Fundamental Programming

with C, C++, and Python



Irfan Subakti

Profitina

profitina.com

Fundamental Programming

with C, C++ and Python

Irfan Subakti



Copyright © 2026 Irfan Subakti. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **xxiv**
Who this book is for xxiv
One idea, several languages xxiv
What is in it xxiv
How to read it xxv
About the author xxvi
Acknowledgments xxvi

CHAPTER 1 Computers, C and Your First Programs **1**
1.1 A Receipt Written by Hand 1
1.2 What a Computer Does 2
1.2.1 Hardware and software 2
1.2.2 Programs and programming languages 2
1.3 C and Python: Two Languages, One Way of Thinking 3
1.3.1 Why C 3
1.3.2 Why Python as well 3
1.3.3 Compiled and interpreted 4
1.4 Setting Up Your Workbench 4
1.4.1 Windows 10 and 11 5
1.4.2 macOS 6
1.4.3 Linux (Ubuntu, Debian and relatives) 6
1.4.4 Visual Studio Code and its two extensions 6
1.4.5 The code folder of this chapter 6
1.5 Your First Program, Line by Line 7
1.6 From Source Text to Running Program 8
1.6.1 What `gcc` really does 8
1.6.2 When something is wrong: errors and warnings 8
1.7 Variables and Memory 9
1.7.1 A named box 9
1.7.2 Reading input with `scanf` 10
1.7.3 Four basic types 11
1.8 Arithmetic 12
1.8.1 Five operators and one trap 12
1.8.2 Precedence: who goes first 13
1.8.3 The same arithmetic in Python 14
1.9 Decisions with `if` 14
1.10 Profitina Laundry Cashier v0.1 16
1.11 Making It Better: Decimal Weights and a Minimum Charge 17
1.12 Using It for Real 19
1.13 Common Mistakes, and How to Avoid Them 19
1.14 Chapter Summary 20
1.15 Exercises 20
1.16 Solutions to the Exercises 21
References 22

CHAPTER 2 Structured Program Development: Algorithms, Pseudocode and Control **24**
2.1 An Evening at the Counter 24
2.2 Algorithms: Actions in the Right Order 24
2.3 Pseudocode 25
2.4 Three Structures Are Enough 26

2.4.1 Sequence, selection and repetition ... 26
2.4.2 Reading a flowchart ... 27
2.4.3 Blocks ... 27
2.5 Selection ... 27
2.5.1 `if` and `if`…`else` ... 27
2.5.2 Chains of decisions ... 27
2.5.3 The conditional operator ... 29
2.6 Repetition Controlled by a Counter ... 30
2.6.1 The `while` loop ... 30
2.6.2 Counting to five ... 30
2.7 Top-Down, Stepwise Refinement: Repetition Controlled by a Sentinel ... 32
2.7.1 A harder problem ... 32
2.7.2 The top and the first refinement ... 32
2.7.3 The second refinement ... 32
2.7.4 The program ... 34
2.7.5 Checking the design by hand: a trace table ... 34
2.7.6 The cast ... 35
2.8 Nested Control Structures ... 36
2.9 Assignment Operators and Increment Operators ... 37
2.9.1 Compound assignment ... 37
2.9.2 Increment and decrement ... 38
2.10 Profitina Laundry Cashier v0.2 ... 39
2.11 Making It Better: Guarding Against Impossible Input ... 43
2.12 Using It for Real ... 43
2.13 Common Mistakes, and How to Avoid Them ... 44
2.14 Chapter Summary ... 44
2.15 Exercises ... 45
2.16 Solutions to the Exercises ... 45
References ... 47

CHAPTER 3 Program Control: for, do…while, switch and Logical Operators ... 48

3.1 A New Washing Machine and a Neater Report ... 48
3.2 The `for` Statement ... 48
3.2.1 Counting loops in one line ... 48
3.2.2 Python's `for` and `range` ... 50
3.3 A `for` Showcase: Compound Interest and Field Widths ... 51
3.3.1 The formula ... 51
3.3.2 The program ... 51
3.3.3 Field widths ... 52
3.3.4 Reading the table like an owner ... 53
3.4 The `do`…`while` Statement ... 53
3.4.1 At least once ... 53
3.4.2 Ask until the answer is valid ... 54
3.5 The `switch` Statement ... 55
3.5.1 One decision, many cases ... 55
3.5.2 Falling through ... 56
3.5.3 `switch` or an `else if` chain? ... 57
3.5.4 Python's `match` ... 57
3.6 `break` and `continue` ... 58
3.7 Logical Operators ... 59
3.7.1 `&&`, `||` and `!` ... 59
3.7.2 Short-circuit evaluation ... 60
3.7.3 De Morgan's laws ... 61
3.7.4 From a business rule to a condition ... 61
3.8 Choosing the Right Structure ... 62

3.9 Profitina Laundry Cashier v0.3 ... 62
3.10 Making It Better: A Price List for the Wall ... 65
3.11 Using It for Real ... 66
3.12 Common Mistakes, and How to Avoid Them ... 67
3.13 Chapter Summary ... 67
3.14 Exercises ... 68
3.15 Solutions to the Exercises ... 68
References ... 69

CHAPTER 4 Evaluation and Discussion I: Five Problems Solved Step by Step ... 70

4.1 Where We Left Off ... 70
4.2 How to Approach a Programming Problem ... 71
4.2.1 Seven steps ... 71
4.2.2 How each problem is laid out, and where the code lives ... 71
4.3 Problem 4.1: Areas of Plane Figures ... 72
4.3.1 The problem ... 72
4.3.2 Understanding it by hand ... 72
4.3.3 Designing it on paper ... 72
4.3.4 The C solution ... 72
4.3.5 The Python solution ... 74
4.3.6 A tempting wrong solution ... 74
4.4 Problem 4.2: The Guessing Game ... 75
4.4.1 The problem ... 75
4.4.2 Understanding it by hand ... 75
4.4.3 Designing it on paper ... 75
4.4.4 The C solution ... 76
4.4.5 The Python solution ... 77
4.4.6 A tempting wrong solution ... 77
4.5 Problem 4.3: 100 to the Power n ... 78
4.5.1 The problem ... 78
4.5.2 Understanding it by hand ... 78
4.5.3 Designing it on paper ... 79
4.5.4 The C solution ... 79
4.5.5 The Python solution ... 80
4.5.6 A tempting wrong solution ... 81
4.6 Problem 4.4: Rock, Paper, Scissors ... 82
4.6.1 The problem ... 82
4.6.2 Understanding it by hand ... 82
4.6.3 Designing it on paper ... 83
4.6.4 The C solution ... 83
4.6.5 The Python solution ... 85
4.6.6 A tempting wrong solution ... 85
4.7 Problem 4.5: The Goblin Battle ... 86
4.7.1 The problem ... 86
4.7.2 Understanding it by hand ... 87
4.7.3 Designing it on paper ... 87
4.7.4 The C solution ... 88
4.7.5 The Python solution ... 90
4.7.6 A tempting wrong solution ... 91
4.8 The Problems Side by Side ... 92
4.9 Using It for Real ... 93
4.10 Common Mistakes, and How to Avoid Them ... 93
4.11 Chapter Summary ... 94
4.12 Exercises ... 94
4.13 Solutions to the Exercises ... 95

References 95

CHAPTER 5 Functions: Divide, Name and Conquer 96

5.1 Mbak Rina's Cashier Has Grown Too Long 96
5.2 The Idea in Plain Words 97
5.2.1 A function is a named piece of work 97
5.2.2 What happens during a call 97
5.2.3 Divide and conquer: why small pieces win 98
5.3 Your First Functions 99
5.3.1 One rule, written once 99
5.3.2 Prototypes, `void` functions and pass-by-value 100
5.3.3 Python's view: a name must exist when it is called 102
5.4 Borrowing Functions: The Standard Library 102
5.5 Scope and Lifetime: Where Names Live 105
5.6 Randomness You Can Repeat 106
5.7 Recursion: Functions That Call Themselves 110
5.7.1 A base case and a smaller problem 110
5.7.2 A useful recursion: rupiah with dots 111
5.7.3 The price of recursion: Fibonacci 111
5.7.4 Recursion or a loop? 115
5.8 Building the Cashier, Version 4 115
5.8.1 One file of rules, one file of story 115
5.8.2 Compiling several files 119
5.9 Running It 120
5.10 Testing It 120
5.11 Making It Better 122
5.12 Using It for Real 123
5.13 Common Mistakes, and How to Avoid Them 123
5.14 Chapter Summary 124
5.15 Exercises 124
5.16 Solutions to the Exercises 125
References 129

CHAPTER 6 Arrays: Many Values, One Name 131

6.1 The Notebook at the End of the Day 131
6.2 The Idea in Plain Words 132
6.2.1 An array is a row of numbered boxes 132
6.2.2 Why the index starts at 0 132
6.2.3 Nobody checks the fence 133
6.2.4 Python's list 133
6.3 Declaring, Filling and Walking Through an Array 133
6.4 Arrays as Counters 136
6.5 Passing Arrays to Functions 137
6.6 Sorting: Bubble Sort 140
6.7 Searching: Linear and Binary 142
6.8 Two-Dimensional Arrays 146
6.9 Building the Cashier, Version 5 149
6.10 Running It 154
6.11 Testing It 155
6.12 Making It Better 157
6.13 Using It for Real 158
6.14 Common Mistakes, and How to Avoid Them 159
6.15 Chapter Summary 159
6.16 Exercises 160

6.17 Solutions to the Exercises 160
References 163

CHAPTER 7 Pointers: Addresses as Values 164
7.1 One Call, Three Answers 164
7.2 The Idea in Plain Words 165
7.2.1 Every variable has an address 165
7.2.2 Following the arrow 165
7.2.3 The type of a pointer, and the pointer to nothing 166
7.2.4 Pointers you have already used 166
7.3 Your First Pointer 166
7.4 Pass-by-Reference: Changing the Caller's Variables 168
7.5 Output Parameters: One Call, Several Answers 171
7.6 Arrays and Pointers: Two Notations, One Memory 174
7.6.1 The name of an array is the address of its first element 174
7.6.2 Pointer arithmetic 174
7.6.3 Walking with a pointer 175
7.6.4 Why pointers and not only indexes? 176
7.7 `const` and Pointers 177
7.8 Pointers to Functions: Behavior as a Parameter 177
7.9 Building the Cashier, Version 6 180
7.10 Running It 184
7.11 Testing It 185
7.12 Making It Better: Pointer Safety 187
7.13 Using It for Real 188
7.14 Common Mistakes, and How to Avoid Them 189
7.15 Chapter Summary 189
7.16 Exercises 190
7.17 Solutions to the Exercises 190
References 193

CHAPTER 8 Evaluation and Discussion II: Functions, Arrays and Pointers, Solved Step by Step 195
8.1 Where We Left Off 195
8.2 How to Approach a Problem 196
8.2.1 The eight steps 196
8.2.2 How each problem is laid out, and where the code lives 196
8.3 Problem 8.1: Statistics Functions 197
8.3.1 The problem 197
8.3.2 Understanding it by hand 197
8.3.3 Designing it on paper 198
8.3.4 The C solution 198
8.3.5 The Python solution 200
8.3.6 A tempting wrong solution 201
8.4 Problem 8.2: Fibonacci Two Ways 204
8.4.1 The problem 204
8.4.2 Understanding it by hand 204
8.4.3 Designing it on paper 204
8.4.4 The C solution 205
8.4.5 The Python solution 207
8.4.6 Measuring the difference 209
8.4.7 A tempting wrong solution 209
8.5 Problem 8.3: Palindromes 211
8.5.1 The problem 211
8.5.2 Understanding it by hand 211

8.5.3 Designing it on paper 212
8.5.4 The C solution 212
8.5.5 The Python solution 214
8.5.6 A tempting wrong solution 215
8.6 Problem 8.4: The Determinant of a 3 × 3 Matrix 217
8.6.1 The problem 217
8.6.2 Understanding it by hand 218
8.6.3 Designing it on paper 218
8.6.4 The C solution 219
8.6.5 The Python solution 220
8.6.6 A tempting wrong solution 221
8.7 Problem 8.5: Finding a Receipt 222
8.7.1 The problem 222
8.7.2 Understanding it by hand, and the design 222
8.7.3 The solutions 222
8.7.4 A tempting wrong solution 224
8.8 Problem 8.6: Merging Two Shifts 224
8.8.1 The problem 224
8.8.2 Understanding it by hand, and the design 224
8.8.3 The solutions 225
8.8.4 A tempting wrong solution 226
8.9 The Problems Side by Side 227
8.10 Common Mistakes, and How to Avoid Them 227
8.11 Chapter Summary 228
8.12 Exercises 228
8.13 Solutions to the Exercises 229
References 229

CHAPTER 9 Characters, Strings and Formatted I/O 231
9.1 Where We Left Off: Mbak Rina's Names 231
9.2 The Idea in Plain Words: Text Is Numbers 232
9.2.1 A character is a small number 232
9.2.2 A string is an array that ends with a zero 232
9.2.3 The same program in Python 235
9.3 One Character at a Time: `<ctype.h>` 236
9.4 The String Library and Its Sharp Edges 240
9.5 Formatted Output: the Full Dial of `printf` 243
9.6 A Receipt That Lines Up 245
9.7 Reading Text Safely 250
9.8 Building Cashier v0.7 251
9.9 Running and Testing It 262
9.10 Making It Better 264
9.11 Using It for Real 264
9.12 Common Mistakes, and How to Avoid Them 265
9.13 Chapter Summary 265
9.14 Exercises 266
9.15 Solutions to the Exercises 266
References 269

CHAPTER 10 Structures, Enums and a First Linked List 271
10.1 Where We Left Off: Three Arrays That Must Agree 271
10.2 The Idea in Plain Words: One Record, One Name 272
10.2.1 Declaring a structure 272
10.2.2 What a structure looks like in memory 272
10.2.3 The first program 273

10.3 Structures and Functions 277
10.4 `enum` and `typedef`: Names Instead of Numbers 282
10.5 Memory on Demand: `malloc` and `free` 285
10.6 A Structure That Points to Itself: the Linked List 286
10.6.1 The idea 286
10.6.2 Walking the list 287
10.6.3 Adding a node 287
10.6.4 Removing a node 287
10.6.5 Freeing the whole list 288
10.6.6 The program 288
10.7 Building Cashier v0.8 295
10.8 Running and Testing It 309
10.9 Making It Better 310
10.10 Using It for Real 311
10.11 Common Mistakes, and How to Avoid Them 311
10.12 Chapter Summary 312
10.13 Exercises 312
10.14 Solutions to the Exercises 313
References 316

CHAPTER 11 File Processing: Data That Survives Power-Off 317
11.1 Where We Left Off: Four O'Clock, and the Power Goes Off 317
11.2 The Idea in Plain Words: Files and Streams 317
11.2.1 Streams and the `FILE` pointer 317
11.2.2 Modes 318
11.3 Writing and Reading a Text File 319
11.4 The `feof` Trap 320
11.5 Reading Real Data: a CSV File Line by Line 322
11.6 Append Mode: How Logs Grow 327
11.7 Binary Files and Random Access 329
11.8 Saving Without Losing: Write, Close, Rename 333
11.9 Building Cashier v0.9 334
11.9.1 One program, several files 334
11.9.2 The storage module 336
11.9.3 The main program 339
11.10 Running and Testing It 343
11.11 Making It Better 344
11.12 Using It for Real 344
11.13 Common Mistakes, and How to Avoid Them 345
11.14 Chapter Summary 345
11.15 Exercises 346
11.16 Solutions to the Exercises 346
References 349

CHAPTER 12 Evaluation and Discussion III: Lists, Trees and a Knight 351
12.1 Where We Left Off 351
12.2 How to Approach a Data-Structure Problem 351
12.2.1 The steps 351
12.2.2 Where the code lives 352
12.3 Problem 12.1: Sort the Linked List, Then Insert at the End 352
12.3.1 The problem 352
12.3.2 Understanding it by hand 352
12.3.3 Designing it on paper 353
12.3.4 The C solution 354

12.3.5 The Python solution 356
12.3.6 Running and testing it 357
12.3.7 A tempting wrong solution 358
12.3.8 Check yourself 360
12.4 Problem 12.2: A Tree Grown from a File 360
12.4.1 The problem 360
12.4.2 Understanding it by hand 360
12.4.3 Designing it on paper 361
12.4.4 The C solution 362
12.4.5 The Python solution 367
12.4.6 Running and testing it 370
12.4.7 A tempting wrong solution 372
12.4.8 Check yourself 372
12.5 Problem 12.3: The Knight's Shortest Path 372
12.5.1 The problem 372
12.5.2 Understanding it by hand 373
12.5.3 Designing it on paper 374
12.5.4 The C solution 374
12.5.5 The Python solution 378
12.5.6 Running and testing it 380
12.5.7 A tempting wrong solution 381
12.5.8 Check yourself 382
12.6 The Problems Side by Side 382
12.7 Using It for Real 383
12.8 Common Mistakes, and How to Avoid Them 383
12.9 Chapter Summary 384
12.10 Exercises 384
12.11 Solutions to the Exercises 384
References 388

CHAPTER 13 C++ as a Better C 389
13.1 Bu Rina's Receipts 389
13.2 The Idea in Plain Words 390
13.2.1 One family, two dialects 390
13.2.2 What "better C" means 390
13.2.3 Compiling a C++ program 391
13.3 Streams: `cout` and `cin` 391
13.4 `std::string`: Text That Fits 393
13.5 `std::vector`: An Array That Grows 395
13.6 References: Changing the Caller's Variable Without Pointers 397
13.7 Overloading and Default Arguments 399
13.8 `new` and `delete` 402
13.9 Building It: Cashier v1.0 403
13.10 Running It 410
13.11 Testing It 411
13.12 Making It Better: Refusing Bad Lines Politely 413
13.13 Using It for Real 414
13.14 Common Mistakes, and How to Avoid Them 415
13.15 Chapter Summary 415
13.16 Exercises 416
13.17 Solutions to the Exercises 416
References 417

CHAPTER 14 Classes and Data Abstraction 418
14.1 Free Laundry, by Accident 418

14.2 The Idea in Plain Words ... 419
14.2.1 Data abstraction ... 419
14.2.2 The invariant ... 419
14.2.3 From `struct` to `class` ... 419
14.3 The Problem, Measured ... 420
14.4 Building the `Order` Class ... 421
14.4.1 The header: the interface ... 422
14.4.2 The implementation ... 423
14.4.3 Using the class ... 425
14.4.4 The same class in Python ... 427
14.4.5 Two questions beginners ask ... 429
14.4.6 Tracing a refused order step by step ... 429
14.5 Constructors and Destructors: Born Valid, Cleaned Up Automatically ... 430
14.6 A First Look at Inheritance ... 432
14.7 Building It: The Shop ... 437
14.7.1 The `Shop` class ... 437
14.7.2 The program around it ... 439
14.8 Running It ... 442
14.9 Testing It ... 443
14.10 Making It Better ... 444
14.11 Using It for Real ... 444
14.12 Common Mistakes, and How to Avoid Them ... 445
14.13 Chapter Summary ... 445
14.14 Exercises ... 446
14.15 Solutions to the Exercises ... 446
References ... 447

CHAPTER 15 Operator Overloading and a Complete Case Study ... 448

15.1 Receipts That Add Up ... 448
15.2 The Idea in Plain Words ... 448
15.2.1 An operator is a function with a special name ... 448
15.2.2 The rules ... 449
15.3 A Money Class: `Rupiah` ... 450
15.3.1 The class ... 450
15.3.2 The class at work ... 453
15.3.3 Tracing one expression by hand ... 454
15.3.4 The same class in Python ... 454
15.4 A Second Example: Fractions and Euclid's Algorithm ... 456
15.5 The Complete Case Study: Profitina Laundry ... 461
15.5.1 The data ... 461
15.5.2 The rules ... 462
15.5.3 The program ... 462
15.5.4 Following two orders through `take` ... 468
15.6 Running It ... 472
15.7 Testing It ... 474
15.8 Making It Better ... 474
15.9 Using It for Real ... 475
15.10 Common Mistakes, and How to Avoid Them ... 475
15.11 Chapter Summary ... 476
15.12 Exercises ... 476
15.13 Solutions to the Exercises ... 477
References ... 478

CHAPTER 16 Evaluation and Discussion IV: Classes, Overloading and the Whole Book, Solved Step by Step 479
16.1 Where We Left Off 479
16.2 How to Approach a Problem 480
16.2.1 The seven steps 480
16.2.2 How each problem is laid out, and where the code lives 480
16.3 Problem 16.1: Chess Pieces 480
16.3.1 The problem 480
16.3.2 Understanding it by hand 481
16.3.3 Designing it on paper 481
16.3.4 The C++ solution 482
16.3.5 The Python solution 486
16.3.6 A tempting wrong solution 487
16.4 Problem 16.2: Overloading `add` 488
16.4.1 The problem 488
16.4.2 Understanding it by hand 488
16.4.3 Designing it on paper 489
16.4.4 The C++ solution 489
16.4.5 The Python solution 490
16.4.6 A tempting wrong solution 490
16.5 Problem 16.3: Finding Anagrams 491
16.5.1 The problem 491
16.5.2 Understanding it by hand 492
16.5.3 Designing it on paper 492
16.5.4 The C++ solution 493
16.5.5 The Python solution 495
16.5.6 A tempting wrong solution 495
16.6 Problem 16.4: Matrix Multiplication 496
16.6.1 The problem 496
16.6.2 Understanding it by hand 496
16.6.3 Designing it on paper 497
16.6.4 The C++ solution 498
16.6.5 The Python solution 500
16.6.6 A tempting wrong solution 501
16.7 Problem 16.5: When Can I Pick It Up? 502
16.7.1 The problem 502
16.7.2 Understanding it by hand 503
16.7.3 Designing it on paper 503
16.7.4 The C++ solution 503
16.7.5 The Python solution 506
16.7.6 A tempting wrong solution 508
16.8 The Problems Side by Side 509
16.9 Using It for Real 510
16.10 Common Mistakes, and How to Avoid Them 510
16.11 Chapter Summary 511
16.12 Exercises 511
16.13 Solutions to the Exercises 512
References 512

List of Figures

Figure 1.1 Two routes from source text to output: a C program passes through the preprocessor, the compiler and the linker once and becomes an executable file; a Python program is read and run by the interpreter every time. 4

Figure 1.2 Three variables after a run of `add_two`: `scanf` wrote 45 and 72 to the addresses of `integer1` and `integer2`, and the assignment read both and wrote 117 into `sum`. 10

Figure 2.1 The three control structures: sequence, selection and repetition; each has exactly one entry and one exit. 26

Figure 2.2 The sentinel-controlled average: read before the loop, test at the top, read again at the end of the body, and divide only when there is at least one order. 33

Figure 3.1 The anatomy of a `for` statement: the initialization (1) runs once; then the condition (2), the body (3) and the increment (4) repeat until the condition is false. 49

Figure 3.2 `while` tests before the body, so the body may never run; `do`…`while` tests after the body, so the body always runs at least once. 54

Figure 4.1 The rule of rock, paper, scissors as a cycle: every object beats the next one along the arrows. 82

Figure 4.2 One turn of the battle: choose the damage, hit the first living goblin, let the living goblins hit back, print a line, and decide whether the battle goes on. 88

Figure 5.1 A call of `bill_for(7000, 3)`: the arguments are copied into the parameters, the body runs, and `return` sends one value back to the place of the call, where `main` continues. 98

Figure 5.2 The call stack at the deepest point of `factorial(4)`: each call has its own frame with its own `n`, and every frame below the top waits for the answer of the frame above it. 111

Figure 5.3 Every call made by `fib(5)`: fifteen calls to compute one number, and the amber calls repeat work already done elsewhere in the tree. 112

Figure 6.1 The array `bill` with eight elements: one name, indexes 0 to 7, and the elements side by side in memory, eight bytes apart on a computer where a `long` takes eight bytes. 132

Figure 6.2 Bubble sort on six bills: after each pass one more value (shaded) is in its final place at the right; pass 4 makes no swap, so the sort stops early. 141

Figure 6.3 The first three rows of the week table, as we picture them and as they lie in memory: row after row, so that `kg[r][c]` lies $3r + c$ elements after the start (four bytes per `int`). 148

Figure 7.1 Three variables in memory: `p` holds the address of `kg`, drawn as an arrow; `*p` follows the arrow, so `*p = 7` changes `kg`. 165

Figure 7.2 Left: `swap_copies` works on copies, and `main` keeps 9 and 4. Right: `swap` receives the addresses of `main`'s variables and exchanges the values there. 169

Figure 7.3 Pointer arithmetic: `p + k` points `k` elements further, whatever the size of an element, so `bill[i]`, `*(bill + i)` and `p[i]` name the same box. 174

Figure 8.1 The work of the two methods, as counted by `fibonacci.c`: loop steps grow like n, recursive calls like 1.618^n (logarithmic scale). 207

Figure 8.2 The two-pointer scan on `kasurrusak`: the amber letters are compared, the gray ones are done; after five equal pairs the pointers cross and the word is a palindrome. 211

Figure 8.3 Expansion along the first row: each first-row entry (amber) multiplies the 2×2 determinant of the cells left when its row and column are removed (teal), with the signs $+, -, +$. 218

Figure 9.1 Two strings in memory: `name` holds four letters and the terminator in exactly five bytes, while `buffer` reserves sixteen bytes of which the string uses six. The grey numbers under `name` are the ASCII codes. 233

Figure 9.2 The parts of the conversion specification `%-9.2f`: the flag `-` left-aligns, the width 9 is the minimum number of characters, the precision 2 gives the digits after the decimal point, and the type `f` says that a `double` is printed. 244

Figure 10.1 The layout of `struct Order` in memory: four members at offsets 0, 4, 28 and 32, and two bytes of padding after `customer` so that `tenths` starts at a multiple of four. 272

Figure 10.2 A singly linked list of orders: (a) `head` points at the first node and every `next` at the following one, the last at `NULL`; (b) appending changes the last node's `next` and the `tail` pointer; (c) removing a node from the middle bridges over it with one assignment. 286
Figure 11.1 A binary file of four 32-byte records: by Equation (11.1), record 2 starts at byte $2 \cdot 32 = 64$, and `fseek` goes there directly. 329
Figure 11.2 Two ways to save: (a) overwriting the file in place leaves it empty or half written during the save; (b) writing a new file and then renaming it over the old one leaves, at every moment, one complete file. 333
Figure 12.1 Growing the tree of `tree5.txt`: after each command the new node (amber) becomes the current node, and a command `u...ur` first climbs one level per `u` from it. 361
Figure 12.2 The number of knight moves needed from the start (7,0) to every square; the target (0,7) needs 6, and the amber squares show one shortest path. 373
Figure 13.1 Three ways to pass `points`: a value parameter gets a copy, a pointer parameter gets the address, and a reference parameter is another name for the caller's own variable. 397
Figure 14.1 The `Order` class: a public interface around private data; outside code can only call the public functions, and the constructor checks every value. 421
Figure 14.2 The life cycle of an order: `advance()` moves it forward twice, `pickUp()` hands it over, and every other call is refused without changing the state. 425
Figure 15.1 What the compiler does with `wash + iron` for class operands: it looks for a non-member `operator+(wash, iron)` or a member `wash.operator+(iron)` and calls the one it finds. 449
Figure 15.2 The case study: three data files are loaded into the `Shop`, each order is checked and turned into a `Receipt`, and the receipts are printed and summarized in a report written both to the screen and to `report.txt`. 461
Figure 16.1 The squares each piece may reach from (3, 3) on an empty board, with the condition that decides them. 481
Figure 16.2 The five windows of length 4 in `aabaabaa`: four are anagrams of `aaba`; the last window starts at $n - m = 4$. 492
Figure 16.3 Daily kilograms per service times the price per kilogram gives the daily revenue; $C00$ combines row 0 of A with column 0 of B 497

List of Tables

Table 1.1 The same multiplication at three levels of language; the machine and assembly lines are typical of an x86-64 processor. 3
Table 1.2 The workbench of this book; all tools are free and run on Windows, macOS and Linux. 5
Table 1.3 The precedence of the operators met so far; when in doubt, add parentheses. 13
Table 2.1 A trace of **Pseudocode 2.6** for the input 4, 7, 3, 0: the sentinel is read at the end of the third iteration and is never added. 35
Table 2.2 The compound assignment operators of C; Python has the same operators, with `//=` for integer division. 38
Table 3.1 Field widths and precision in `printf`; Python's format specifications inside f-strings use the same numbers, with `>` for right and `<` for left alignment. 52
Table 3.2 The truth table of the logical operators AND (`&&`), OR and NOT (`!`); in C a condition is false when it is 0 and true otherwise. 60
Table 3.3 Choosing a control structure: the question to ask and the statement that answers it. 62
Table 4.1 The powers of 100 against the limits of C's integer types: `int` stops after $n = 4$, `long long` after $n = 9$. 79
Table 4.2 All nine combinations checked against Equation (4.3): remainder 0 is a draw, 1 means player 1 wins, 2 means player 2 wins. 83
Table 4.3 The five problems of this chapter, their structures, and the trap of each tempting wrong solution. 93

Table 5.1 The standard headers used most in this book; a header holds prototypes, the library holds the compiled functions. 103
Table 5.2 Scope and lifetime of the four kinds of variables in C. 105
Table 7.1 Where `const` stands in a pointer declaration decides what is protected: the variable pointed to, the pointer, or both. 177
Table 8.1 The six problems of this chapter and the tools from Chapters 5 to 7 that each one exercises most. 196
Table 8.2 The standard deviation of the ten values worked by hand: the deviations add up to zero, and the squared deviations to 166.40. 198
Table 8.3 Measured running time of the recursive and iterative methods; the recursive times grow about elevenfold from $n = 35$ to $n = 40$, as $1.6185 \approx 11$ predicts. 209
Table 8.4 The six problems side by side: the cost of each solution, the trap of its tempting wrong solution, and the test input that exposes the trap. 227
Table 9.1 The most used functions of `<ctype.h>`. Characters that are not letters pass through `toupper` and `tolower` unchanged. 236
Table 9.2 The everyday functions of `<string.h>` (and `snprintf` from `<stdio.h>`), each with its safety rule. 240
Table 10.1 The work of common operations on n orders, for an array and for a linked list that keeps pointers to its first and last node. 287
Table 11.1 The modes of `fopen`. Adding `b` gives the same mode for a binary stream. 318
Table 12.1 The three problems compared: structure, key operation, the idea that makes it work, cost, and the mistake to avoid. 383
Table 13.1 The migration table: the same jobs in C, in C++ and in Python. 391
Table 14.1 The three access specifiers and who may use a member in each. 419
Table 15.1 The operators overloaded in this chapter, their usual form in C++ and their Python counterparts. 449
Table 16.1 The five problems of this chapter side by side. 510

List of Listings

Listing 1.1 `hello.c`: the first C program prints two lines of text. 7
Listing 1.2 `hello.py`: the same two lines in Python. 7
Listing 1.3 `add_two.c`: reading two integers and adding them. 9
Listing 1.4 `add_two.py`: the same program in Python. 10
Listing 1.5 `types.c`: an `int`, a `double` and a `char`, and the range of `int`. 11
Listing 1.6 `types.py`: the same values in Python, which has no `char` type and no limit on `int`. 12
Listing 1.7 `arithmetic.c`: the five operators, integer division, and precedence. 12
Listing 1.8 `arithmetic.py`: the same results in Python, which needs `//`, `int(...)` and `math.fmod` to imitate C. 14
Listing 1.9 `compare.c`: the six comparison operators, each guarding one `printf` 14
Listing 1.10 `compare.py`: the same comparisons in Python. 15
Listing 1.11 `receipt.c`: cashier v0.1 prints a receipt for the wash, dry and fold service. 16
Listing 1.12 `receipt.py`: cashier v0.1 in Python. 16
Listing 1.13 `receipt_min.c`: cashier v0.1, improved, with decimal weights and a minimum charge. 17
Listing 1.14 `receipt_min.py`: the improved cashier in Python. 18
Listing 1.15 `wrong_average.c`: `1 / 2` is 0, so half of 7 kg becomes 0.00 kg. 19
Listing 1.16 `change.c`: change split into notes with `/` and `%`. 21
Listing 2.1 `discount.c`: an `if`…`else if`…`else` chain, a separate `if`, and the conditional operator. 28
Listing 2.2 `discount.py`: `if`, `elif` and `else`, and Python's conditional expression. 30
Listing 2.3 `average_counter.c`: a `while` loop controlled by a counter. 31

Listing 2.4 `average_counter.py`: the same counter-controlled loop in Python. 31
Listing 2.5 `average_sentinel.c`: sentinel-controlled repetition, a cast, and a guard against dividing by zero. 34
Listing 2.6 `average_sentinel.py`: the same algorithm in Python. 35
Listing 2.7 `survey.c`: an `if`…`else` nested in a `while`. 36
Listing 2.8 `survey.py`: the same nested structure in Python, where indentation alone shows that the `if` belongs to the `while`. 37
Listing 2.9 `increment.c`: compound assignment, and prefix versus postfix increment. 38
Listing 2.10 `increment.py`: the same results in Python, which has compound assignment but no `++`. 39
Listing 2.11 `cashier_v02.c`: cashier v0.2 serves a queue of customers until the till is closed. 40
Listing 2.12 `cashier_v02.py`: cashier v0.2 in Python. 42
Listing 2.13 `heavy_count.c`: counting the orders heavier than 5 kg, with negative weights rejected. 45
Listing 3.1 `for_basics.c`: counting up, counting down, counting in steps, and a running sum. 49
Listing 3.2 `for_basics.py`: the same loops with `for` and `range`. 50
Listing 3.3 `savings.c`: a deposit growing at compound interest, printed as an aligned table. 51
Listing 3.4 `savings.py`: the same table in Python. 52
Listing 3.5 `validate.c`: `do`…`while` keeps asking until the weight is in range. 54
Listing 3.6 `validate.py`: Python's form of "ask until valid" is `while True` with `break`. 55
Listing 3.7 `service_count.c`: `switch` with shared cases, `break` and `default`. 55
Listing 3.8 `service_count.py`: `match`, where `case "W" | "w"` lists two alternatives and `case _` catches everything else. 57
Listing 3.9 `break_continue.c`: `break` ends the loop at 5; `continue` skips only 5. 58
Listing 3.10 `logic.c`: the truth table of `&&`, `||` and `!`, and a division protected by short-circuit evaluation. 60
Listing 3.11 `logic.py`: `and`, `or` and `not`, with the same short-circuit protection. 61
Listing 3.12 `cashier_v03.c`: cashier v0.3 with `switch`, validated weights and a formatted daily report. 62
Listing 3.13 `cashier_v03.py`: cashier v0.3 in Python, with `match` and the `while True` idiom. 64
Listing 3.14 `price_table.c`: nested `for` loops print a price list with aligned columns. 65
Listing 4.1 `geometry.c`: the areas of four plane figures, with negative lengths rejected. 72
Listing 4.2 `geometry.py`: the same areas in Python. 74
Listing 4.3 `geometry_wrong.c`: `1 / 2` is evaluated first, as integer division. 74
Listing 4.4 `guess.c`: the guessing game as a read-ahead `while` loop. 76
Listing 4.5 `guess.py`: the same read-ahead loop in Python. 77
Listing 4.6 `guess_wrong.c`: the body judges every guess, including the winning one. 77
Listing 4.7 `power100.c`: 100 to the power n, printed digit by digit, never overflowing. 79
Listing 4.8 `power100.py`: the same digit-printing loop in Python. 80
Listing 4.9 `power100_wrong.c`: `pow` returns a `double`, which has only about 16 significant digits. 81
Listing 4.10 `rps.c`: validated input, names printed with `switch`, and the winner found with one remainder. 83
Listing 4.11 `rps.py`: the same game with `while True` validation and `match`. 85
Listing 4.12 `rps_wrong.c`: without `+ 3`, C's `%` gives −2 for rock against scissors. 85
Listing 4.13 `battle.c`: the battle as one `while` loop over five state variables, printed as a table. 88
Listing 4.14 `battle.py`: the same battle in Python. 90
Listing 4.15 `battle_wrong.c`: without the "is it alive?" checks, dead goblins keep fighting and HP goes negative. 91
Listing 5.1 `first_function.c`: one bill rule, defined once above `main` and called four times. 99
Listing 5.2 `first_function.py`: the same rule as a Python function. 99
Listing 5.3 `prototypes.c`: prototypes at the top, `main` first, `void` functions, a function built from another, and a demonstration of pass-by-value. 100
Listing 5.4 `prototypes.py`: the same program in Python, where a function must exist when it is called, not when it is read. 102

Listing 5.5 `rounding.c`: `floor`, `round` and `ceil` from `<math.h>`, and the laundry's charging rule as a function. 103
Listing 5.6 `rounding.py`: the same table in Python, with a small function that rounds halves the way C does. 104
Listing 5.7 `scope.c`: a global counter, locals, a block that hides a name, and a `static` receipt counter. 105
Listing 5.8 `scope.py`: the same program in Python, with `global` declarations and a module variable in the role of C's `static`. 106
Listing 5.9 `lucky_draw.c`: a repeatable random-number generator with hidden `static` state, dice made from it, and a fairness check. 107
Listing 5.10 `lucky_draw.py`: the same generator in Python, with the 32-bit wrap-around made explicit. 108
Listing 5.11 `recursion.c`: factorial, rupiah with dots, and Fibonacci with a counter of calls. 112
Listing 5.12 `recursion.py`: the same three recursions in Python. 113
Listing 5.13 `laundry.h`: the promises of the business-rules module, with its constants and an include guard 115
Listing 5.14 `laundry.c`: Profitina Laundry's business rules, one function per rule. 115
Listing 5.15 `cashier.c`: the cashier, version 4, whose `main` tells the story and leaves the work to named functions. 117
Listing 5.16 `laundry.py`: the business rules as a Python module. 118
Listing 5.17 `cashier.py`: the cashier, version 4, in Python, importing the rules from `laundry.py`. 118
Listing 5.18 `test_laundry.c`: twelve checks of the business rules against hand-worked answers 120
Listing 5.19 `test_laundry.py`: the same checks in Python, with one more for the rupiah text. 121
Listing 5.20 `ex5_gcd.c`: Euclid's algorithm, recursively, and the least common multiple built from it. 125
Listing 5.21 `ex5_gcd.py`: the same in Python. 126
Listing 5.22 `ex5_digits.c`: the digit sum, recursively and with a loop, and a check digit. 126
Listing 5.23 `ex5_digits.py`: the same in Python. 127
Listing 5.24 `ex5_hanoi.c`: the Towers of Hanoi, the classic problem that is easy recursively and hard with a loop. 128
Listing 5.25 `ex5_hanoi.py`: the same in Python. 129
Listing 6.1 `day_bills.c`: eight bills in one array, printed, added, searched for the biggest and counted. 133
Listing 6.2 `day_bills.py`: the same walks through a Python list. 135
Listing 6.3 `service_count.c`: an array of counters, indexed by the service code, and a bar of stars for each. 136
Listing 6.4 `service_count.py`: the same counters in Python. 137
Listing 6.5 `array_functions.c`: functions that receive an array and its size; `const` promises not to change it, and `add_to_all` changes the caller's array. 137
Listing 6.6 `array_functions.py`: the same functions on Python lists. 139
Listing 6.7 `sort_search.c`: bubble sort with a printed trace of each pass, linear search, and binary search with a printed trace of each look. 142
Listing 6.8 `sort_search.py`: the same sort and searches in Python. 144
Listing 6.9 `week_table.c`: kilograms per day and service in a two-dimensional array, with row totals, column totals and the busiest day. 147
Listing 6.10 `week_table.py`: the same table as a Python list of lists. 148
Listing 6.11 `bills.h`: the promises of the module for arrays of bills. 149
Listing 6.12 `bills.c`: total, index of the maximum, copying, sorting from biggest to smallest, and the median. 149
Listing 6.13 `day_report.c`: the cashier, version 5, which stores the day in parallel arrays and prints the closing report. 151
Listing 6.14 `bills.py`: the same tools for Python lists. 152
Listing 6.15 `day_report.py`: the cashier, version 5, in Python. 153

Listing 6.16 `test_bills.c`: nine checks of the array tools, including the empty array and the promise that `median` leaves its input unchanged. 155
Listing 6.17 `test_bills.py`: the same checks in Python. 156
Listing 6.18 `ex6_reverse.c`: reversing an array in place with two indexes. 160
Listing 6.19 `ex6_reverse.py`: the same in Python (idiomatic Python would write `a.reverse()`). 161
Listing 6.20 `ex6_insertion.c`: insertion sort with a trace after each insertion. 161
Listing 6.21 `ex6_insertion.py`: the same in Python. 162
Listing 7.1 `first_pointer.c`: taking an address, following it, writing through it, re-aiming the pointer, copying it, and `NULL`. 166
Listing 7.2 `first_pointer.py`: one-element lists as boxes that two names can share, which is the part of pointers that Python has. 168
Listing 7.3 `swap.c`: swapping copies changes nothing in `main`; swapping through pointers changes `main`'s variables. Three compare-and-swap steps then sort three numbers. 168
Listing 7.4 `swap.py`: Python cannot swap two variables of its caller; it swaps with a tuple assignment in the caller, or changes a shared list. 170
Listing 7.5 `out_params.c`: `split_weight` gives two answers through pointers; `day_stats` gives three answers and returns whether they are valid. 172
Listing 7.6 `out_params.py`: the Python way: return a tuple, or `None` when there is no valid answer. 173
Listing 7.7 `pointer_walk.c`: three names for one element, a pointer walk, pointer difference, a negative index from a moved pointer, and a word reversed with two pointers. 175
Listing 7.8 `pointer_walk.py`: the same walk with indexes, which is how Python expresses what C does with pointers. 176
Listing 7.9 `sort_by.c`: one bubble sort that asks a function which of two values comes first, three such functions in an array, and the library's `qsort` with its comparison function. 177
Listing 7.10 `sort_by.py`: functions passed as values in Python, and the library's `sorted`. 179
Listing 7.11 `stats.h`: the promises of the statistics module. 180
Listing 7.12 `stats.c`: `split_weight`, the four-answer `day_stats` with a pointer walk, and the comparison function for `qsort`. 181
Listing 7.13 `closing.c`: the cashier, version 6: the closing summary from one call to `day_stats`. 181
Listing 7.14 `stats.py`: the statistics module in Python, returning tuples. 183
Listing 7.15 `closing.py`: the cashier, version 6, in Python. 183
Listing 7.16 `test_stats.c`: fourteen checks, including rounding at a kilogram boundary, an empty day, a null pointer and the promise that refused calls leave the answers untouched. 185
Listing 7.17 `test_stats.py`: the same checks in Python. 186
Listing 7.18 `ex7_strlen.c`: string length by pointer difference, and counting a character with a pointer walk. 191
Listing 7.19 `ex7_strlen.py`: the same counts in Python, whose strings carry no terminating zero. 191
Listing 7.20 `ex7_change.c`: change in banknotes, each call reducing the caller's remaining amount through a pointer. 192
Listing 7.21 `ex7_change.py`: the same with a returned tuple. 192
Listing 8.1 `stats.c`: the given skeleton with the five bodies completed; `INT_MIN` and `INT_MAX` start the searches, the cast comes before the division, and the deviation uses `sqrt`. 198
Listing 8.2 `stats.py`: the same five functions in Python, where `/` always gives a fractional result. 200
Listing 8.3 `wrong_stats.c`: a tempting wrong solution that starts the minimum at 0 and divides two whole numbers. 202
Listing 8.4 `wrong_stats.py`: the same two mistakes in Python, where the whole-number division is written `//`. 202
Listing 8.5 `fibonacci.c`: the iterative method with the sum through an output parameter, the recursive method with a call counter, and a table of the work for larger n. 205
Listing 8.6 `fibonacci.py`: the same two methods in Python; the iterative one returns a tuple. 207
Listing 8.7 `wrong_fibonacci.c`: a tempting wrong solution that starts the sequence at 1, 1. 209
Listing 8.8 `wrong_fibonacci.py`: the same mistake in Python. 210

Listing 8.9 `palindrome.c`: three ways to decide a palindrome, which must agree: two pointers, recursion on two pointers, and XOR over the pairs. 212
Listing 8.10 `palindrome.py`: the same three methods in Python, with indexes. 214
Listing 8.11 `wrong_palindrome.c`: a tempting wrong solution whose partner index is `n - i` instead of `n - 1 - i`. 216
Listing 8.12 `wrong_palindrome.py`: the same mistake in Python, with an explicit zero character to imitate C. 216
Listing 8.13 `determinant.c`: reading 3 × 3 matrices with nested loops and computing each determinant by expansion along the first row. 219
Listing 8.14 `determinant.py`: the same determinant in Python, with the matrix built from slices. 220
Listing 8.15 `wrong_determinant.c`: a tempting wrong solution that adds the middle term instead of subtracting it. 221
Listing 8.16 `wrong_determinant.py`: the same mistake in Python. 221
Listing 8.17 `find_receipt.c`: recursive binary search over sorted receipt numbers, printing each call indented by its depth. 222
Listing 8.18 `find_receipt.py`: the same recursion in Python; `f"{'':{2 * depth}}"` prints the indentation. 223
Listing 8.19 `merge_shifts.c`: merging two sorted arrays with three walking pointers, including the case of an empty evening shift. 225
Listing 8.20 `merge_shifts.py`: the same merge in Python with two indexes; `extend` copies the rest of the unfinished list. 226
Listing 9.1 `string_basics.c`: a string is a `char` array ending in `'\0'`; `sizeof` counts bytes, `strlen` counts characters. 233
Listing 9.2 `string_basics.py`: the same lines in Python, whose strings know their length and cannot be changed in place. 235
Listing 9.3 `char_classes.c`: classifying characters, keeping the digits of a phone number, and tidying a name. 236
Listing 9.4 `char_classes.py`: the same three tasks with Python's string methods. 239
Listing 9.5 `string_library.c`: copying, joining, bounded formatting, comparing, searching and cutting strings. 240
Listing 9.6 `string_library.py`: Python's counterparts of the `<string.h>` functions. 242
Listing 9.7 `format_dial.c`: flags, widths, precisions and types of `printf`, one at a time. 244
Listing 9.8 `format_dial.py`: the same specifications with Python's `%` operator. 245
Listing 9.9 `receipt.c`: a receipt laid out with widths, `%*s` centering and thousands separators. 246
Listing 9.10 `receipt.py`: the same receipt in Python. 248
Listing 9.11 `cashier_v7.c`: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name. 251
Listing 9.12 `cashier_v7.py`: the same cashier in Python. 258
Listing 9.13 `word_stats.c`: counting lines, words and letters, the longest word and the most common letter. 267
Listing 9.14 `word_stats.py`: the same statistics in Python. 268
Listing 10.1 `order_struct.c`: declaring, initializing, copying and comparing structures, and measuring their layout. 273
Listing 10.2 `order_struct.py`: a data class for the order, and `ctypes` to measure the C layout. 275
Listing 10.3 `order_functions.c`: structures passed by value and by pointer, a function that returns a pointer into an array, and `qsort` with a comparison function. 277
Listing 10.4 `order_functions.py`: the same functions in Python, where every argument is a reference. 280
Listing 10.5 `service_enum.c`: enumerations for services and statuses, and a service table built with designated initializers. 282
Listing 10.6 `service_enum.py`: `IntEnum` and a dictionary as the service table. 284
Listing 10.7 `order_list.c`: a linked list of orders with `push_front`, `append`, `find`, `remove_id` and `free_all`. 288
Listing 10.8 `order_list.py`: the same linked list in Python, built from `Node` objects. 292

Listing 10.9 `cashier_v8.c`: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list. 295
Listing 10.10 `cashier_v8.py`: cashier v0.8 in Python. 303
Listing 10.11 `list_reverse.c`: reversing a linked list in place. 314
Listing 10.12 `list_reverse.py`: the same in Python. 315
Listing 11.1 `write_read_text.c`: writing three orders to a text file and reading them back. 319
Listing 11.2 `write_read_text.py`: the same file written and read in Python. 320
Listing 11.3 `feof_trap.c`: the loop `while (!feof(fp))` runs once too often. 320
Listing 11.4 `feof_trap.py`: an imitation of the C loop in Python. 321
Listing 11.5 `csv_report.c`: reading a CSV file line by line, skipping damaged lines with a message, and totaling by service and by day. 323
Listing 11.6 `csv_report.py`: the same report in Python. 326
Listing 11.7 `activity_log.c`: appending one numbered entry to a log file on every run. 327
Listing 11.8 `activity_log.py`: the same log in Python. 328
Listing 11.9 `binary_records.c`: fixed-size records written with `fwrite`, reached with `fseek`, and changed in place. 330
Listing 11.10 `binary_records.py`: the same records with Python's `struct` module. 331
Listing 11.11 `shop.h`: the types and the functions of the shop module, protected by an include guard. 334
Listing 11.12 `storage.h`: loading and saving the shop. 336
Listing 11.13 `storage.c`: the CSV loader, which checks every field of every line, and the saver, which writes a temporary file and renames it. 336
Listing 11.14 `cashier_v9.c`: cashier v0.9, which loads the orders at start and saves after every change. 339
Listing 11.15 `storage.py`: loading and saving in Python. 340
Listing 11.16 `cashier_v9.py`: cashier v0.9 in Python. 341
Listing 11.17 `find_customer.c`: searching a CSV file line by line. 347
Listing 11.18 `find_customer.py`: the same search in Python. 348
Listing 12.1 `list_sort.c`: bubble sort on a linked list by swapping data, then insertion at the end. 354
Listing 12.2 `list_sort.py`: the same solution in Python. 356
Listing 12.3 `wrong_list_sort.c`: a single pass of neighbor swaps, which is not a sort. 358
Listing 12.4 `wrong_list_sort.py`: the same mistake in Python. 359
Listing 12.5 `tree_from_file.c`: growing a binary tree from a command file, printing it in three orders and sideways, and freeing it. 362
Listing 12.6 `tree_from_file.py`: the same tree builder in Python. 367
Listing 12.7 `knight.c`: breadth-first search with a linked-list queue for the fewest knight moves. 374
Listing 12.8 `knight.py`: the same search in Python. 378
Listing 12.9 `knight_dfs.c`: depth-first search reports the first path it finds, not the shortest. 381
Listing 12.10 `knight_dfs.py`: the same mistake in Python. 381
Listing 12.11 `list_insertion_sort.c`: insertion sort by relinking nodes. 385
Listing 12.12 `list_insertion_sort.py`: the same in Python. 386
Listing 13.1 `hello_laundry.cpp`: reading a name and a weight with `cin` and printing with `cout` 391
Listing 13.2 `hello_laundry.py`: the same program in Python. 393
Listing 13.3 `strings_demo.cpp`: the everyday operations of `std::string`. 393
Listing 13.4 `strings_demo.py`: the same operations with Python's `str`. 395
Listing 13.5 `vector_demo.cpp`: a day's weights in a `std::vector`. 395
Listing 13.6 `vector_demo.py`: Python's `list` does what `std::vector` does. 397
Listing 13.7 `references.cpp`: value, pointer and reference parameters, and a function with two results. 398
Listing 13.8 `references.py`: what Python does instead of references. 399
Listing 13.9 `overload_default.cpp`: three overloads of `printLine` and a default argument in `charge`. 400

Listing 13.10 `overload_default.py`: default arguments in Python, and three names instead of overloads. 401
Listing 13.11 `new_delete.cpp`: `new[]` with `delete[]`, `new` with `delete`, and a `vector` that needs neither. 402
Listing 13.12 `new_delete.py`: Python frees memory by itself. 402
Listing 13.13 `cashier.cpp`: Profitina Laundry cashier v1.0 in C++. 403
Listing 13.14 `cashier.py`: the same cashier in Python. 408
Listing 13.15 `tests/run_tests.py`: compiles, runs and compares every program of a chapter. 411
Listing 14.1 `struct_problem.cpp`: an open structure accepts any value in any field. 420
Listing 14.2 `struct_problem.py`: a Python dataclass is just as open. 420
Listing 14.3 `order.h`: the interface of the `Order` class. 422
Listing 14.4 `order.cpp`: the implementation of the `Order` class. 423
Listing 14.5 `order_main.cpp`: creating, refusing and advancing orders. 425
Listing 14.6 `order.py`: the `Order` class in Python. 427
Listing 14.7 `order_main.py`: using the Python `Order`; it prints exactly what `order_main.cpp` prints. 428
Listing 14.8 `raii_demo.cpp`: constructors and destructors announce themselves. 430
Listing 14.9 `raii_demo.py`: the same guarantee with Python's `with` statement. 431
Listing 14.10 `services.cpp`: a base class `Service` and three kinds of service. 432
Listing 14.11 `services.py`: the same hierarchy in Python. 435
Listing 14.12 `shop.h`: the interface of the `Shop` class. 437
Listing 14.13 `shop.cpp`: the implementation of the `Shop` class. 437
Listing 14.14 `shop_main.cpp`: reads commands and passes them to the shop. 439
Listing 14.15 `shop.py`: the `Shop` class in Python. 440
Listing 14.16 `shop_main.py`: the command loop in Python. 441
Listing 15.1 `rupiah.h`: an amount of money in whole rupiah, with its operators. 450
Listing 15.2 `rupiah_demo.cpp`: every operator of `Rupiah` in use. 453
Listing 15.3 `rupiah.py`: the `Rupiah` class in Python. 455
Listing 15.4 `rupiah_demo.py`: the Python demo prints exactly what `rupiah_demo.cpp` prints. 455
Listing 15.5 `fraction.cpp`: a `Fraction` class built on Euclid's gcd, with arithmetic, comparison and output operators. 457
Listing 15.6 `fraction.py`: the `Fraction` class in Python. 459
Listing 15.7 `laundry.cpp`: the complete Profitina Laundry program in C++. 462
Listing 15.8 `laundry.py`: the complete Profitina Laundry program in Python. 469
Listing 16.1 `chess.cpp`: the given `Piece` and `Pawn`, and the four added pieces. 482
Listing 16.2 `chess.py`: the chess pieces in Python; the output is identical. 486
Listing 16.3 `wrong_rook.cpp`: the rook's condition with `||` and the bishop's without $dx \neq 0$. 487
Listing 16.4 `add_overload.cpp`: `add` for two to five integers, for two `double`s, and a variadic template. 489
Listing 16.5 `add_overload.py`: one Python function for any number of arguments. 490
Listing 16.6 `wrong_add.cpp`: a defaulted `add` next to the given one; the call with two arguments is ambiguous. 490
Listing 16.7 `anagram.cpp`: anagram search with `std::is_permutation` and with a sliding count. 493
Listing 16.8 `anagram.py`: the same two methods in Python. 495
Listing 16.9 `wrong_anagram.py`: the loop stops one window too early. 495
Listing 16.10 `matrix.cpp`: matrix multiplication with a dimension check. 498
Listing 16.11 `matrix.py`: matrix multiplication in Python. 500
Listing 16.12 `wrong_matrix.cpp`: B's indices swapped, tested with the identity matrix and with an ordinary matrix. 501
Listing 16.13 `pickup_time.cpp`: the `Time` class with its operators, and the pick-up times of five orders. 503
Listing 16.14 `pickup_time.py`: the `Time` class in Python. 506

Listing 16.15 `wrong_time.py`: hours and minutes kept apart, without carrying.....508

List of Pseudocode

Pseudocode 2.1 Pseudocode for the receipt program of Chapter 1. 25
Pseudocode 2.2 Pseudocode for the discount: a chain of three cases, then a separate decision for members..... 28
Pseudocode 2.3 Counter-controlled repetition: exactly five weights are read. 30
Pseudocode 2.4 The top: one sentence that states the whole task. 32
Pseudocode 2.5 The first refinement: initialize, process, finish. 32
Pseudocode 2.6 The second refinement: sentinel-controlled repetition with a guard against division by zero. 33
Pseudocode 2.7 Pseudocode for the survey: a selection nested inside a counter-controlled repetition, followed by a second selection. 36
Pseudocode 2.8 The design of cashier v0.2: a sentinel loop around a selection, and a guarded average at the end. 39
Pseudocode 2.9 The refined weight input: ask again until the weight is sensible. 43
Pseudocode 4.1 The design of Problem 4.1: a sequence of reads, one guard, and four formulas. 72
Pseudocode 4.2 The design of Problem 4.2: a read-ahead loop whose sentinel is the answer itself. 75
Pseudocode 4.3 The design of Problem 4.3: print the digits of $102n$ instead of computing the number. 79
Pseudocode 4.4 The design of Problem 4.4: two validating loops, a `switch` for the names, and the rule of Equation (4.3). 83
Pseudocode 4.5 The design of Problem 4.5: five state variables and one loop, one iteration per turn. 88
Pseudocode 6.1 Bubble sort with early exit: each pass carries the biggest remaining value to the end of the unsorted part. 141
Pseudocode 8.1 The iterative Fibonacci: two neighbors of the sequence move forward n times, and the sum collects every value that `prev` takes. 204
Pseudocode 12.1 Bubble sort on a singly linked list, swapping the numbers of neighboring nodes. 353
Pseudocode 12.2 Building a binary tree from navigation commands, with a parent pointer in every node. 361
Pseudocode 12.3 Breadth-first search for the fewest knight moves, with the queue as a linked list. 374
Pseudocode 16.1 The move shared by all pieces; MOVE-CONDITION is different for each kind of piece. 482
Pseudocode 16.2 Finding anagrams by testing every window with a permutation check. 492
Pseudocode 16.3 Matrix multiplication with the dimension check first. 497
Pseudocode 16.4 Computing the pick-up time from the drop-off time and the service. 503

List of Equations

(1.1) 12
(2.1) 28
(2.2) 33
(3.1) 51
(3.2) 61
(4.1) 72
(4.2) 78

(4.3) 83
(5.1) 103
(5.2) 107
(5.3) 110
(5.4) 111
(5.5) 112
(6.1) 132
(6.2) 135
(6.3) 142
(6.4) 146
(6.5) 147
(7.1) 165
(7.2) 171
(7.3) 174
(7.4) 174
(7.5) 179
(8.1) 197
(8.2) 197
(8.3) 204
(8.4) 217
(8.5) 218
(8.6) 225
(9.1) 233
(9.2) 244
(10.1)....273
(11.1)....329
(12.1)....353
(12.2)....373
(13.1)....392
(13.2)....401
(14.1)....419
(14.2)....425
(15.1)....452
(15.2)....456
(15.3)....456
(15.4)....457
(15.5)....462
(16.1)....481
(16.2)....489
(16.3)....492
(16.4)....496
(16.5)....497
(16.6)....503

Preface

Every program you have ever used was once a list of small, exact instructions that somebody wrote down: take this number, compare it with that one, repeat until the pile is empty, print the result. A computer is fast and tireless, but it does only what those instructions say — no more, no less, and never what you merely meant. Learning to write such instructions clearly, so that they are right every time and still readable next month, is what programming is, and it is the whole subject of this book. Keep that sentence in mind and nothing in these pages will feel arbitrary.

The book grew out of the Fundamental Programming course (EF234101) taught in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS), Surabaya, and it has been rebuilt so that it stands entirely on its own. Its sixteen chapters follow the rhythm of a sixteen-week semester, but you need no classroom, no lecturer and no other document to learn from it.

Who this book is for

You are assumed to know nothing. Not “nothing about C” — nothing. If you have never written a line of code, never installed a compiler, never opened a terminal, you are exactly the reader this book was written for. Every term is introduced in plain words before it is used, every program is traced on paper before it is run, and every program in the book was run before its output was printed here.

That principle makes the book long. A shorter book would assume you already knew the hard parts, and such a book is no use to the person who does not.

One idea, several languages

Programming is a way of thinking, not a feature of one language, and this book treats it that way. The language it teaches first is **C**, because C hides almost nothing: you see the variables sitting in memory, the addresses that point at them, and the exact moment a file is written to disk. After twelve chapters of C, the book moves on to **C++**, and shows what a class, a reference and an overloaded operator add to the C you already know. And every program, in both parts, is given again, complete and runnable, in **Python**, doing the same thing and printing the same output, line for line. Seeing one idea written two ways is the quickest route to understanding the idea itself rather than the habits of one tool, and it shows you what Python is quietly doing for you — so that whichever language your next course or job uses, you will already know what lies underneath it.

What is in it

Chapters 1 to 3 build the ground: what a computer does and how a C program becomes a running one, with a full set-up of VS Code, a C compiler and Python on Windows, macOS and Linux; your first programs, variables in memory, arithmetic and decisions with `if`; then algorithms and pseudocode, the three control structures that are enough for any program, loops controlled by a counter or a sentinel, and top-down stepwise refinement; then `for`, `do`…`while`, `switch`, `break` and `continue`, the logical operators, and how to choose the right structure. Chapters 5 to 7 give programs their shape: functions, the standard library, scope and lifetime, repeatable randomness

and recursion; arrays, used as counters, passed to functions, sorted with bubble sort, searched linearly and by halves, and laid out in two dimensions; and pointers, with pass-by-reference, output parameters, the close kinship of arrays and pointers, `const`, and pointers to functions. Chapters 9 to 11 let programs handle real data: characters, strings, `<ctype.h>`, the string library and its sharp edges, the full dial of `printf` and safe input; structures, `enum` and `typedef`, memory on demand with `malloc` and `free`, and a first linked list; and files, text and binary, the `feof` trap, reading a CSV line by line, append mode, random access, and saving without losing. Chapters 13 to 15 make the step to C++: streams, `std::string`, `std::vector`, references, overloading and `new` and `delete`; classes, with constructors and destructors that make every object born valid and cleaned up automatically, and a first look at inheritance; and operator overloading, with a money class and a fraction class, ending in a complete case study.

Chapters 4, 8, 12 and 16 are **Evaluation and Discussion** chapters. Each closes one quarter of the book by solving complete problems in full — every problem restated, designed on paper, programmed, tested, run, and explained — with the tempting wrong solution shown and taken apart, and a "check yourself" variant at the end of every problem. They range from areas of plane figures, a guessing game and rock, paper, scissors, through statistics, Fibonacci two ways, palindromes and a 3 × 3 determinant, to sorting a linked list, growing a tree from a file and a knight's shortest path, and finally chess pieces as a class hierarchy, anagrams and matrix multiplication.

Throughout, one small business grows chapter by chapter: **Profitina Laundry**, a laundry on a busy lane in Surabaya, run by its owner Pak Hendra with Mbak Rina at the counter. It begins with receipts written by hand and a tired person multiplying weights by prices in her head. Its cashier program then grows one version per chapter: a first receipt with a minimum charge, a loop that serves customers until closing time, a price list for the wall, functions that hold every price in exactly one place, a day's bills in an array, sorted and searched, names on the receipt, orders as records in a linked list, a day's work that survives a power cut, and at last a complete shop written in C++ with classes and a money type that cannot go wrong. Each chapter also has a *Using It for Real* section that shows how the same ideas carry over to other small businesses and everyday problems.

Every program in this book was run before its output was printed. The programs are collected, chapter by chapter, in the accompanying `Code` folder, each chapter with its `c` (or `cpp`) and `python` versions side by side, its tests, and a README that explains how to build and run each one in a terminal and in VS Code on Windows, macOS and Linux. Every listing in the book is the file in that folder, character for character.

How to read it

With an editor open and a compiler ready. Type the examples in rather than copying them; the typing is where the errors happen, and the errors are where the learning happens. Before you run a program, trace it on paper — the variables in their boxes, the values they hold after each line — and predict what it will print; then run it and compare. Break each example deliberately once it works, and read what the compiler or the failure says. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

Numbered figures, tables, listings, pseudocode and equations are all live links: in the electronic edition, click "Listing 5.3" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at ITS since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform, and Profitina Laundry, the running example of this book, is borrowed from its family of case studies. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason this book prefers a story to a list whenever a story will do.

Acknowledgments

To the students who worked through the earlier forms of this material and asked the questions that showed where an explanation was still thin, and to the colleagues at ITS with whom the course has been taught and refined over the years. The book owes its standard, too, to the classic textbooks of programming in C and C++ and to the official documentation of C, C++ and Python, which are cited wherever they are used.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Computers, C and Your First Programs

Every program you have ever used, from the cashier app at a minimarket to the operating system of your phone, began as a few lines of plain text that someone typed, compiled and ran. This chapter takes you from nothing to that point. You will learn what a computer really does with a program, why this book teaches C and Python side by side, and how to set up a free and complete workbench on Windows, macOS or Linux: Visual Studio Code, the GCC compiler and Python 3.14. Then you will write, compile and run your first programs, and understand every line of them: printing text, storing numbers in variables, reading what the user types, calculating with the five arithmetic operators, and making a first decision with `if`. The chapter ends with version 0.1 of the cashier program for Profitina Laundry, a small neighborhood laundry whose story runs through the whole book, and with an improved version that handles decimal weights and a minimum charge. By the end you will be able to write, run and explain small programs in both languages, and you will know the handful of mistakes that trip up almost every beginner.

1.1 A Receipt Written by Hand

Profitina Laundry is a small laundry on a busy street near a university. Students, office workers and families bring their clothes in plastic bags; Mbak Rina, who runs the counter, weighs each bag on a digital scale, writes the weight and the price on a paper receipt, and tells the customer when to come back. Most customers choose the everyday service, *wash, dry and fold*, which costs Rp 8,000 per kilogram. Some are in a hurry and pay Rp 12,000 per kilogram for the one-day express service; others only need their clothes ironed, at Rp 6,000 per kilogram.

On a quiet morning this works well. On a Monday after a long weekend, with ten people waiting and bags piled on the counter, it does not. Mbak Rina multiplies 4.3 by 8,000 in her head and writes Rp 34,000 instead of Rp 34,400. She forgets that bags under three kilograms are charged as three kilograms, a rule the owner introduced because a tiny bag takes a washing machine just as long as a full one. At the end of the day she adds up forty receipts on a calculator, and the total does not match the money in the drawer. Nobody is dishonest; the arithmetic is simply too much for a tired person in a hurry.

This is exactly the kind of work that computers do better than people. A computer does not get tired, does not forget a rule once the rule is written down, and multiplies 4.3 by 8,000 correctly every single time. But a computer does nothing by itself. Someone has to write down, in a language the computer can follow, what a receipt is, how the price is calculated and what happens with a bag under three kilograms. That written-down procedure is a **program**, and writing it is **programming**.

This book teaches you to write such programs from the very beginning. We will build the Profitina Laundry cashier step by step: in this chapter it prints one receipt; in Chapter 2 it serves a whole queue of customers; in Chapter 3 it validates what the cashier types and prints a daily report; and in later chapters it will remember customers, sort orders, save everything to files, and finally become a C++ program with classes. Along the way you will also meet everyday problems that have nothing to do with laundry, because the same few ideas solve all of them.

1.2 What a Computer Does

1.2.1 Hardware and software

A computer is a machine that does two things extremely fast: it stores data, and it carries out instructions that change that data. The physical parts of the machine are its **hardware**: the processor, the memory, the storage, the screen, the keyboard, the network card. The instructions are its **software**. A laptop without software is a box of metal and glass that switches on and does nothing useful; the same laptop with a spreadsheet program becomes a calculator, and with a cashier program it becomes a cash register.

Four hardware parts matter for programming, and it pays to picture them from the start:

- The **processor** (CPU, central processing unit) carries out instructions, one after another, billions of them per second. Each instruction is tiny: add two numbers, compare two numbers, copy a number from one place to another, jump to another instruction.
- The **main memory** (RAM, random-access memory) holds the program that is running and the data it works on. Memory is a long row of numbered cells called bytes; the number of a cell is its **address**. Memory is fast but forgets everything when the power goes off.
- The **storage** (an SSD or a hard disk) keeps files when the power is off: your programs, your photographs, the laundry's records. It is much slower than memory.
- The **input and output devices** connect the machine to people: the keyboard and the mouse bring data in, the screen and the printer bring results out.

Everything inside the machine is stored as **bits**, switches that are either off (0) or on (1). Eight bits make a **byte**, which can hold 256 different patterns, enough for one letter of the alphabet or a small number. Larger numbers take several bytes; a whole number in C normally takes four bytes, which is 32 bits. You will rarely think in bits in this book, but it explains a fact that surprises every beginner: numbers in a computer have limits. A four-byte whole number cannot be larger than 2,147,483,647, and Chapter 4 solves a problem that is all about that limit.

1.2.2 Programs and programming languages

The processor understands only its own **machine language**: numbered instructions written as patterns of bits. A few bytes such as `01 d8` might mean "add the number in register B to the number in register A" on one kind of processor and something completely different on another. Nobody writes programs like that any more. The next step up is **assembly language**, where each machine instruction gets a short name such as `add` or `mov`; it is readable, but it is still one line per tiny instruction, and it is different for every kind of processor.

A **high-level language** such as C, C++, Java or Python lets us write in terms of the problem instead of the machine. We write `total = weight * price;` and a program called a translator turns that line into the dozens of machine instructions it needs. **Table 1.1** compares the three levels on one small task.

Table 1.1 The same multiplication at three levels of language; the machine and assembly lines are typical of an x86-64 processor.

Level	What the multiplication of weight and price looks like	Who reads it
Machine language	`8b 45 f8 0f af 45 fc 89 45 f4`	the processor only
Assembly language	`mov eax, [weight]` / `imul eax, [price]` / `mov [total], eax`	specialists
High-level language (C)	`total = weight * price;`	every programmer

A high-level program has two further advantages. It is **portable**: the same C program can be translated for a Windows laptop, a MacBook with an Apple processor and a Linux server, and it runs on all three. And it is **readable**: another person, or you in six months, can understand what `total = weight * price;` does without knowing anything about the processor.

1.3 C and Python: Two Languages, One Way of Thinking

1.3.1 Why C

C was created by Dennis Ritchie at Bell Labs between 1969 and 1973 as the language in which the UNIX operating system was rewritten [1]. Its classic description is the book by Kernighan and Ritchie [2], and the language is defined today by an international standard; this book uses the version called C17, published as ISO/IEC 9899:2018 [3]. More than fifty years later C is still everywhere: the kernels of Linux, Windows and macOS, the databases that store the world's records, the interpreter that runs Python itself, and the small computers inside washing machines, cars and phones are all written largely in C. In the TIOBE index of September 2026, which ranks languages by how much they are discussed and searched on the web, C stood in second place with a rating of 10.28%, behind only Python with 17.76% [4], [5].

For a first programming course C has a special virtue: it hides very little. When you store a number, you choose how many bytes it takes. When a function needs to change your variable, you pass its address. When your program reads outside an array, nothing stops it. This makes C less comfortable than Python, but it means that by the end of this book you will know what every other language is doing under the hood.

1.3.2 Why Python as well

Python was created by Guido van Rossum and first released in 1991; in 2026 it is the most widely used language in the world [4], the first language of many schools, and the everyday tool of data analysis and artificial intelligence. It is also a very different language from C. It does the bookkeeping for you: numbers grow as large as they need to, lists grow when you add to them, and memory is released automatically.

This book therefore gives every program twice, in C and in Python, and the two versions always print exactly the same output, character for character. Reading them side by side teaches you two things at once. First, the *ideas* of programming (a variable, a decision, a loop, a function) are the same in every language; only the spelling changes. Second, the *differences* are instructive: when Python quietly does something that C makes you do by hand, you learn what that something is. The current version of Python is 3.14; its latest maintenance release at the time of writing is Python 3.14.7, released on August 5, 2026 [6].

1.3.3 Compiled and interpreted

The two languages also differ in *how* a program gets from text to action, and **Figure 1.1** shows both routes. A C program is **compiled**: a translator called the **compiler** reads the whole source file once and writes a new file, the **executable**, that contains machine language. You then run the executable as often as you like, and the compiler is no longer involved. Python is **interpreted**: the program `python3`, the **interpreter**, reads your source file every time you run it, translates it into an intermediate form called bytecode and carries it out at once.

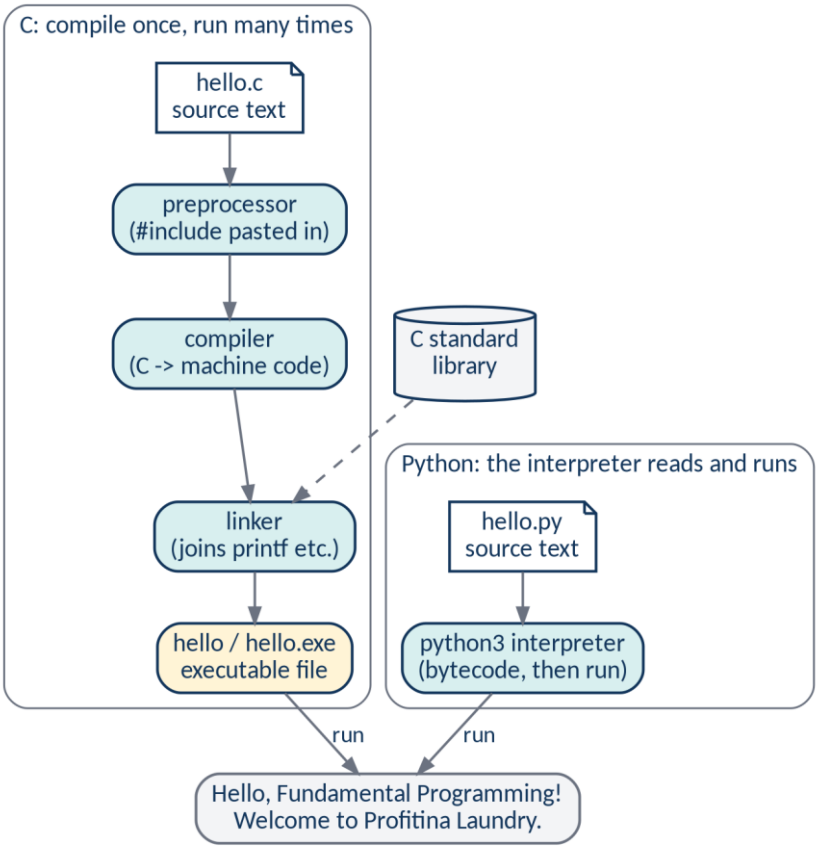


Figure 1.1 Two routes from source text to output: a C program passes through the preprocessor, the compiler and the linker once and becomes an executable file; a Python program is read and run by the interpreter every time.

Compilation costs a step, but it pays off: the compiler checks the whole program before it ever runs, and the finished executable is fast. A loop that adds up many millions of numbers typically runs tens of times faster as compiled C than as plain Python, which is why the parts of Python that must be fast are themselves written in C. For the small programs of this book both are instant.

1.4 Setting Up Your Workbench

A programmer needs three tools: an **editor** to write the source text, a **compiler** or **interpreter** to turn it into action, and a **terminal**, the text window in which you type commands. Everything in this section is free, and the same steps work on a new laptop or an old one. **Table 1.2** lists what we install.

Table 1.2 The workbench of this book; all tools are free and run on Windows, macOS and Linux.

Tool	Version (September 2026)	What it does
Visual Studio Code	1.139 [7]	the editor: colors the code, runs it, helps find errors
GCC (Windows, Linux)	16.2 is the newest release [8]; any version from 9 on works	the C compiler, command `gcc`
Apple clang (macOS)	comes with the Xcode Command Line Tools	a C compiler that also answers to `gcc`
Python	3.14.7 [6]	the interpreter, command `python3` (`python` or `py` on Windows)
VS Code extensions	*C/C++* and *Python*, both by Microsoft	colors, error checking and debugging inside the editor

Every C program in this book is compiled with the same command, whatever the operating system:

```
gcc -std=c17 -Wall -Wextra -O2 hello.c -o hello
```

Read it from left to right: run `gcc`, follow the C17 standard (`-std=c17`), show all the usual warnings and some extra ones (`-Wall -Wextra`), optimize the machine code (`-O2`), translate `hello.c`, and write the executable to a file named `hello` (`-o hello`; on Windows the file becomes `hello.exe`). Programs that use the mathematics library, which we first need in Chapter 3, add `-lm` at the end.

1.4.1 Windows 10 and 11

Windows has no C compiler of its own that works from a terminal, so we take GCC from **MSYS2**, a well-maintained collection of the tools that Linux programmers use; this is also the route that the VS Code documentation describes [9], [10].

1. Download the installer from `https://www.msys2.org` and run it, accepting the folder `C:\msys64`.
2. From the Start menu, open **MSYS2 UCRT64**. The name matters: UCRT64 is the modern variant, and packages installed from another MSYS2 terminal will not fit. Type the command below, press Enter to accept all members of the group, and answer `Y`.
   ```
   pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
   ```
3. Tell Windows where the compiler is. Open **Settings**, search for *environment variables*, choose **Edit environment variables for your account**, select **Path**, click **Edit**, then **New**, and add `C:\msys64\ucrt64\bin`. Close every open terminal afterwards: a terminal reads the Path only when it starts.
4. For Python, install the **Python install manager**, either from the Microsoft Store or from `https://www.python.org/downloads/`. Since Python 3.14 this is the recommended way on Windows; the old full installer still exists but is deprecated [11]. Then open PowerShell and type `py install 3.14`.
5. Install **Visual Studio Code** from `https://code.visualstudio.com`, keeping the ticked option *Add to PATH*.

Now open a *new* PowerShell window and check all three:

```
gcc --version
python --version
code --version
```

The first line should name GCC (MSYS2 builds GCC 15 or 16), the second Python 3.14.7 or a later 3.14 release, and the third the VS Code version, 1.139 or later.

1.4.2 macOS

Apple provides a free C compiler, **clang**, in the Xcode Command Line Tools; you do not need the full Xcode application [12].

1. Open **Terminal** (Applications → Utilities) and type `xcode-select --install`, then click *Install* in the window that appears.
2. Download the macOS installer for Python 3.14 from `https://www.python.org/downloads/` and run it. It installs the command `python3`. Do not remove or change the older Python that macOS may already contain.
3. Download **Visual Studio Code** from `https://code.visualstudio.com`, unpack it, and drag it into the Applications folder. Start it, press Cmd+Shift+P, and run the command *Shell Command: Install 'code' command in PATH*.

Check the result in a new Terminal window with `gcc --version`, `python3 --version` and `code --version`. On a Mac, `gcc --version` answers "Apple clang": the name `gcc` is kept so that commands such as ours work unchanged, and clang accepts the same options.

1.4.3 Linux (Ubuntu, Debian and relatives)

Linux is the home ground of GCC. In a terminal type:

```
sudo apt update
sudo apt install build-essential gdb python3
```

The package `build-essential` brings `gcc`, the standard C library and the tools around them. Ubuntu 26.04 LTS, released in April 2026, ships GCC 15.2 and Python 3.14 as its defaults [13]; on the older Ubuntu 24.04 you get Python 3.12, which runs every Python program of this book, because the newest feature we use, the `match` statement of Chapter 3, arrived in Python 3.10. VS Code is available as a `.deb` package from `https://code.visualstudio.com` or with `sudo snap install code --classic`. On Fedora the corresponding command is `sudo dnf install gcc gdb python3`.

1.4.4 Visual Studio Code and its two extensions

Start VS Code, click the **Extensions** icon on the left (four small squares), and install two extensions published by Microsoft: **C/C++**, which colors C code, checks it while you type and lets you debug it, and **Python**, which does the same for Python and installs the language server Pylance with it [9], [14]. Then choose **File → Open Folder** and open the folder in which you keep your programs, for example the book's `Code_new/Chapter01` folder.

VS Code has a terminal built in: **View → Terminal** opens one at the bottom of the window, already inside the folder you opened. That is where we type every command in this book. You can also run the current Python file with the triangle button at the top right; for C, the terminal is simpler and teaches you what really happens, so we use it throughout.

> **KEY INSIGHT.** Keep one folder per chapter and open that folder, not a single file, in VS Code. The terminal then starts in the right place, the commands of the book work as printed, and the extensions know which files belong together.

1.4.5 The code folder of this chapter

All programs of this chapter are in the folder `Chapter01` of the book's code. The folder `c` holds the C programs, the folder `python` the Python programs with the same names, and the folder `tests` sample inputs together with the exact output each program must print; `README.md` lists the

commands for Windows, macOS and Linux. Typing the programs yourself teaches more than copying them, but when something refuses to work, comparing your file with the book's file is the fastest way to find the difference.

1.5 Your First Program, Line by Line

Tradition says that a first program prints a greeting. Ours greets the reader and the laundry. **Listing 1.1** is the whole program in C.

Listing 1.1 `hello.c`: the first C program prints two lines of text.

```c
/* hello.c - the first program: print two lines of text */
#include <stdio.h>

int main(void)
{
    printf("Hello, Fundamental Programming!\n");
    printf("Welcome to Profitina Laundry.\n");
    return 0;   /* 0 tells the operating system that all went well */
}
```

Six ideas hide in these nine lines, and all of them come back in every program you will ever write in C.

- Text between `/*` and `*/` is a **comment**. The compiler ignores it completely; it is written for people. The first line says what the file is, and the comment after `return 0;` explains a line that a beginner might wonder about. C also accepts comments that start with `//` and run to the end of the line.
- `#include <stdio.h>` is an instruction for the **preprocessor**, the first stage of translation. It pastes in the file `stdio.h` ("standard input and output"), which tells the compiler what `printf` is and how it must be called. Without it the compiler would not know `printf`.
- `int main(void)` begins the **function** `main`. Every C program starts running at the first line of `main`, wherever it stands in the file. The word `int` says that `main` hands back a whole number when it finishes, and `void` says that it receives nothing.
- The braces `{` and `}` enclose the **body** of `main`: the statements that run, one after another, from top to bottom.
- `printf("...\n");` prints the text between the double quotes. The two characters `\n` are an **escape sequence** that means "new line": the cursor jumps to the beginning of the next line. Every statement ends with a semicolon, the way every sentence ends with a full stop.
- `return 0;` ends `main` and reports the number 0 to the operating system, which by convention means "everything went well". Any other number means that something went wrong, and scripts that run programs automatically check this number.

Listing 1.2 is the same program in Python.

Listing 1.2 `hello.py`: the same two lines in Python.

```python
# hello.py - the first program: print two lines of text
print("Hello, Fundamental Programming!")
print("Welcome to Profitina Laundry.")
```

The Python version is shorter because Python does several things for you. There is no `#include`, because `print` is always available. There is no `main`: Python simply runs the file from its first line to its last. There are no braces and no semicolons, because in Python the end of a line ends a statement. And `print` adds the new line by itself. Comments in Python begin with `#`.

To run the C program, open a terminal in the `c` folder, compile, and run the executable. On Linux and macOS you type `./hello`, where `./` means "in this folder"; in Windows PowerShell you type `.\hello.exe`:

```
gcc -std=c17 -Wall -Wextra -O2 hello.c -o hello
./hello
```

To run the Python program, open a terminal in the `python` folder and type `python3 hello.py` (on Windows `python hello.py` or `py hello.py`). Both print the same two lines:

```
Hello, Fundamental Programming!
Welcome to Profitina Laundry.
```

If you see these two lines, your workbench works, and you have written, translated and run a program. Everything else in this book builds on this moment.

1.6 From Source Text to Running Program

1.6.1 What `gcc` really does

The one `gcc` command hides four steps, and the error messages you will meet name them, so it is worth seeing them once. **Figure 1.1** showed them as boxes. The **preprocessor** handles the lines that begin with `#`: it pastes in the contents of `stdio.h` and removes the comments. The **compiler** translates the resulting C text into machine language for your processor; this is where most errors are found. The **assembler** turns that machine language into an *object file*. Finally the **linker** joins your object file with the pieces of the **C standard library** that your program uses, such as the code of `printf`, which you did not write but which must be inside the executable. The result is one file that the operating system can load into memory and start.

1.6.2 When something is wrong: errors and warnings

Everyone makes typing mistakes, and the compiler is the first to notice them. Suppose we forget the semicolon at the end of the first `printf` line. GCC refuses to translate the file and says:

```
hello.c: In function 'main':
hello.c:6:48: error: expected ';' before 'printf'
    6 |     printf("Hello, Fundamental Programming!\n")
      |                                                ^
      |                                                ;
    7 |     printf("Welcome to Profitina Laundry.\n");
      |     ~~~~~~
```

Read an error message from the top. `hello.c:6:48` means "file `hello.c`, line 6, column 48". The word `error` means that no executable was produced. The compiler even shows where the missing character belongs. Notice that it complains about the place *before* `printf` on line 7: an error is often reported slightly after the real mistake, because the compiler notices a problem only when the next word does not fit. So when a message points at a line that looks perfect, look at the line above it.

The same mistake in Python, a missing closing parenthesis, is reported when the interpreter reads the file, before any line runs:

```
  File "hello.py", line 1
    print("Hello, Fundamental Programming!"
         ^
SyntaxError: '(' was never closed
```

A **warning** is different from an error: the program is translated, but the compiler has seen something that is legal C and very probably not what you meant. The options `-Wall -Wextra` switch the useful warnings on, and this book treats every warning as an error in disguise. All programs in this book compile without a single warning; they were also run with GCC's *sanitizers* (`-fsanitize=address,undefined`), which stop a program the moment it reads memory it does not own or computes something the C standard leaves undefined.

> **WARNING.** Never ignore a warning because "the program runs anyway". A program that runs is not yet a program that is correct. In Section 1.13 you will see a missing `&` that produces only a warning and then a program that prints garbage.

There is a third kind of mistake that neither the compiler nor the interpreter can find: the **logic error**. The program is valid and runs, but it computes the wrong thing, for example because you typed `+` where you meant `*`. Only testing finds logic errors, which is why every program in this book is run on examples whose correct answer we know in advance.

1.7 Variables and Memory

1.7.1 A named box

A cashier program must remember things: the weight of the bag, the price per kilogram, the total. A **variable** is a named place in memory that holds one value. **Listing 1.3** reads two whole numbers from the keyboard and prints their sum.

Listing 1.3 `add_two.c`: reading two integers and adding them.

```c
/* add_two.c - read two whole numbers and print their sum */
#include <stdio.h>

int main(void)
{
    int integer1;   /* first number typed by the user */
    int integer2;   /* second number typed by the user */
    int sum;        /* the sum of the two */

    printf("Enter first integer: ");
    if (scanf("%d", &integer1) != 1) return 1;   /* & means "the address of" */
    printf("Enter second integer: ");
    if (scanf("%d", &integer2) != 1) return 1;

    sum = integer1 + integer2;
    printf("Sum is %d\n", sum);
    return 0;
}
```

The line `int integer1;` is a **declaration**. It asks for a box in memory large enough for one `int`, a whole number (an *integer*), and gives the box the name `integer1`. A name may contain letters, digits and the underscore, must not begin with a digit, and is case sensitive: `total` and `Total` are two different variables. Choose names that say what the box holds; `weightKg` is better than `w`.

Three facts about every variable matter, and **Figure 1.2** shows them after the program has run: its **name**, which the program uses; its **value**, what is currently inside; and its **address**, the number of the memory cell where the box begins. The addresses in the figure are examples; they are different on every computer and even on every run.

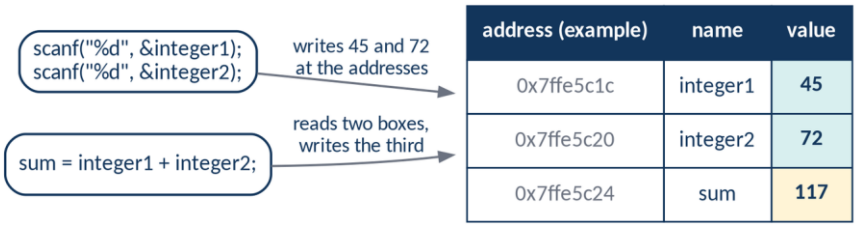


Figure 1.2 Three variables after a run of `add_two`: `scanf` wrote 45 and 72 to the addresses of `integer1` and `integer2`, and the assignment read both and wrote 117 into `sum`.

1.7.2 Reading input with `scanf`

`printf` shows a value; `scanf` does the opposite and puts what the user types into a variable. In `scanf("%d", &integer1)`, the **format** `"%d"` says "read a whole number written in decimal", and `&integer1` is the **address** of the variable, read aloud as "the address of `integer1`". `scanf` needs the address because it must *write* into the box, and a function can write into your box only if you tell it where the box is. `printf`, which only reads the value, gets `integer1` without the `&`.

`scanf` also reports how many values it managed to read. If the user types `abc` instead of a number, it reads nothing and reports 0. Our programs check this with `if (scanf(...) != 1) return 1;`, which means "if not exactly one value was read, stop the program and report failure". You will meet `if` properly in Section 1.9; for now read the line as a safety catch. Besides being good practice, the check is required for a clean compile: on many Linux systems GCC warns when the result of `scanf` is ignored.

The line `sum = integer1 + integer2;` is an **assignment**. The right side is computed first, using the values in the two boxes, and the result is then stored in the box named on the left. The `=` sign in C does not mean "is equal to"; it means "becomes". A statement such as `count = count + 1;`, which is nonsense as mathematics, makes perfect sense as an instruction: take the value of `count`, add one, and put the result back into `count`.

Here is a run in which the user typed 45 and 72:

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Before the first `scanf`, the program prints a **prompt**, a short message that tells the user what to type. A program without prompts waits silently, and the user does not know that it expects anything. Notice that the prompt has no `\n` at its end, so the cursor waits on the same line.

Listing 1.4 is the Python version.

Listing 1.4 `add_two.py`: the same program in Python.

```python
# add_two.py - read two whole numbers and print their sum
integer1 = int(input("Enter first integer: "))    # text typed -> whole number
integer2 = int(input("Enter second integer: "))

total = integer1 + integer2    # "sum" is already a Python function name
print(f"Sum is {total}")
```

Python has no declarations: a variable comes into existence the first time something is assigned to it. The function `input` prints its prompt, waits for the user to press Enter and hands back what was typed as *text*. The text `"45"` is not a number, so `int(...)` converts it into the whole number

45. If the user types something that is not a whole number, `int` stops the program with an error message called a `ValueError`; Chapter 9 shows how to handle such input gracefully in both languages. We call the result `total` instead of `sum` because `sum` is already the name of a function built into Python.

The text between `f"` and `"` is an **f-string**: Python replaces every `{...}` inside it by the value of the expression between the braces. `f"Sum is {total}"` becomes `Sum is 117`. It plays the role of the `%d` in `printf`.

1.7.3 Four basic types

Every C variable has a **type**, which fixes how many bytes the box takes and what kind of value it can hold. This chapter uses four:

- `int`, a whole number, normally four bytes, from −2,147,483,648 to 2,147,483,647;
- `double`, a number with a fraction part such as 4.3, eight bytes, with about 15 to 16 significant digits;
- `char`, one character such as `'A'`, one byte, stored as a small number;
- and, in a few places, `long long`, a whole number of eight bytes that reaches about 9.2×10^{18}; Chapter 4 needs it.

Listing 1.5 shows the first three and the limits of `int`.

Listing 1.5 `types.c`: an `int`, a `double` and a `char`, and the range of `int`.

```c
/* types.c - four basic types and how printf shows them */
#include <stdio.h>
#include <limits.h>

int main(void)
{
    int kilos = 5;           /* a whole number */
    double price = 8000.0;   /* a number with a fraction part */
    char grade = 'A';        /* one character, stored as a small number */

    printf("kilos = %d\n", kilos);
    printf("price = %.2f\n", price);
    printf("bill  = %.2f\n", kilos * price);   /* int times double is double */
    printf("grade = %c, stored as the number %d\n", grade, grade);
    printf("largest int  = %d\n", INT_MAX);
    printf("smallest int = %d\n", INT_MIN);
    return 0;
}
```

```
kilos = 5
price = 8000.00
bill  = 40000.00
grade = A, stored as the number 65
largest int  = 2147483647
smallest int = -2147483648
```

Each type has its own **conversion specification** in `printf`: `%d` for an `int`, `%f` for a `double` (`%.2f` means “with two digits after the decimal point”), and `%c` for a `char`. When `grade` is printed with `%d` instead, we see the number 65 that the computer actually stores for the letter A; this numbering is called ASCII. The constants `INT_MAX` and `INT_MIN` come from the header `limits.h`. In the expression `kilos * price`, an `int` is multiplied by a `double`; C first converts the `int` to a `double`, so the result has a fraction part.

Listing 1.6 gives the same values in Python.

Listing 1.6 `types.py`: the same values in Python, which has no `char` type and no limit on `int`.

```
# types.py - the same values in Python
kilos = 5            # int: a whole number (Python ints never overflow)
price = 8000.0       # float: the same 64-bit number as C's double
grade = "A"          # a string of length one; Python has no char type

print(f"kilos = {kilos}")
print(f"price = {price:.2f}")
print(f"bill  = {kilos * price:.2f}")
print(f"grade = {grade}, stored as the number {ord(grade)}")
print(f"largest int  = {2**31 - 1}")     # the limits of C's 32-bit int
print(f"smallest int = {-2**31}")
```

Python has fewer types, and they behave differently. A Python `int` has no upper limit: `2**100` is computed exactly. A Python `float` is exactly the same eight-byte number as a C `double`, which is why the two programs of this book always agree on fractions. And Python has no separate character type: a single letter is simply a string of length one, and `ord` gives its number. To print the limits of C's `int`, the Python program computes them: `2**31 - 1` means 2 to the power 31, minus 1.

1.8 Arithmetic

1.8.1 Five operators and one trap

C has five arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `%` (remainder, often called *modulus*). The first three behave as in school. Division needs care. When both sides of `/` are whole numbers, C performs **integer division**: it computes the whole-number part of the quotient and throws the fraction away. So `7 / 3` is 2, not 2.333. The operator `%` gives what integer division threw away: `7 % 3` is 1, because 7 is two threes and one left over. The two operators always fit together:

$$a = (a/b) \cdot b + a \,\%\, b \tag{1.1}$$

For $a = 7$ and $b = 3$, Equation (1.1) reads $7 = 2 \cdot 3 + 1$. It is how a cashier counts change, how a clock turns minutes into hours, and how a game decides that "every third turn" has come: `turn % 3 == 0` is true exactly for turns 3, 6, 9 and so on. **Listing 1.7** tries all of this.

Listing 1.7 `arithmetic.c`: the five operators, integer division, and precedence.

```c
/* arithmetic.c - the five arithmetic operators and their precedence */
#include <stdio.h>

int main(void)
{
    printf("7 + 3 = %d\n", 7 + 3);
    printf("7 - 3 = %d\n", 7 - 3);
    printf("7 * 3 = %d\n", 7 * 3);
    printf("7 / 3 = %d\n", 7 / 3);        /* whole-number division */
    printf("7 %% 3 = %d\n", 7 % 3);       /* the remainder */
    printf("-7 / 2 = %d\n", -7 / 2);      /* C cuts toward zero */
    printf("-7 %% 2 = %d\n", -7 % 2);
    printf("7.0 / 2 = %.2f\n", 7.0 / 2);  /* one double makes it exact */
    printf("2 + 3 * 4 = %d\n", 2 + 3 * 4);
    printf("(2 + 3) * 4 = %d\n", (2 + 3) * 4);

    int a = 80, b = 90, c = 85;
    printf("a + b + c / 3 = %d\n", a + b + c / 3);
    printf("(a + b + c) / 3 = %d\n", (a + b + c) / 3);
    return 0;
}
```

```
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7 % 3 = 1
-7 / 2 = -3
-7 % 2 = -1
7.0 / 2 = 3.50
2 + 3 * 4 = 14
(2 + 3) * 4 = 20
a + b + c / 3 = 198
(a + b + c) / 3 = 85
```

Read the output line by line. `7 / 3` gives 2 and `7 % 3` gives 1, as Equation (1.1) promised. For a negative number C cuts the fraction *toward zero*: −3.5 becomes −3, and the remainder then has to be −1 so that Equation (1.1) still holds: $-7 = (-3) \cdot 2 + (-1)$. As soon as one side of `/` is a `double`, as in `7.0 / 2`, the division keeps its fraction and gives 3.5. To print a percent sign with `printf` you must write it twice, `%%`, because a single `%` starts a conversion specification.

1.8.2 Precedence: who goes first

When an expression contains several operators, C does not simply work from left to right. Like school arithmetic, it has rules of **precedence**, summarized in **Table 1.3**. Multiplication, division and remainder bind more tightly than addition and subtraction, so `2 + 3 * 4` is 14, not 20. Parentheses override the rules: `(2 + 3) * 4` is 20.

Table 1.3 The precedence of the operators met so far; when in doubt, add parentheses.

Operators	Meaning	Order
`( )`	parentheses	first; the innermost pair first
`* / %`	multiplication, division, remainder	next; from left to right when several appear
`+ -`	addition, subtraction	next; from left to right
`=`	assignment	last; the value on the right goes into the variable on the left

The last two lines of the output show the most famous precedence bug. To average three scores 80, 90 and 85, a beginner writes `a + b + c / 3`. C computes `c / 3` first (28, with integer division), then adds 80 and 90, and prints 198, which is not even between the three scores. The

correct `(a + b + c) / 3` gives 85. Parentheses cost nothing, and they also tell the reader what you meant.

1.8.3 The same arithmetic in Python

Listing 1.8 prints exactly the same lines in Python, and in doing so it shows where the two languages part ways.

Listing 1.8 `arithmetic.py`: the same results in Python, which needs `//`, `int(...)` and `math.fmod` to imitate C.

```
# arithmetic.py - the same operators in Python, printed like the C program
import math

print(f"7 + 3 = {7 + 3}")
print(f"7 - 3 = {7 - 3}")
print(f"7 * 3 = {7 * 3}")
print(f"7 / 3 = {7 // 3}")               # // is whole-number division
print(f"7 % 3 = {7 % 3}")                # the remainder
print(f"-7 / 2 = {int(-7 / 2)}")         # cut toward zero, as C does
print(f"-7 % 2 = {int(math.fmod(-7, 2))}")   # C's remainder; -7 % 2 gives 1
print(f"7.0 / 2 = {7.0 / 2:.2f}")        # / always gives a fraction
print(f"2 + 3 * 4 = {2 + 3 * 4}")
print(f"(2 + 3) * 4 = {(2 + 3) * 4}")

a, b, c = 80, 90, 85
print(f"a + b + c / 3 = {a + b + c // 3}")
print(f"(a + b + c) / 3 = {(a + b + c) // 3}")
```

In Python, `/` *always* produces a fraction: `7 / 3` is 2.3333333333333335. Integer division has its own operator, `//`. And for negative numbers Python rounds *down*, toward minus infinity, instead of toward zero: `-7 // 2` is −4 and `-7 % 2` is 1 in Python, while C gives −3 and −1. Both choices satisfy Equation (1.1); they simply agree on different quotients. To print C's answers, the Python program uses `int(-7 / 2)`, which cuts toward zero, and `math.fmod`, which computes a remainder the way C does [15], [16]. For positive numbers, which is almost all the time in a laundry, the two languages agree.

> **WARNING.** A tempting one-line formula, `half = 1 / 2 * kilos;`, gives 0 in C for every value of `kilos`, because `1 / 2` is integer division and is computed first. Write `kilos / 2.0` or `0.5 * kilos`. Section 1.13 shows this slip running.

1.9 Decisions with `if`

A program becomes interesting when it can choose. The `if` statement runs a statement only when a **condition** is true. Conditions are built with the six comparison operators `==` (equal to), `!=` (not equal to), `<`, `>`, `<=` and `>=`. **Listing 1.9** compares two numbers in every possible way.

Listing 1.9 `compare.c`: the six comparison operators, each guarding one `printf`.

```c
/* compare.c - the six comparison operators with if */
#include <stdio.h>

int main(void)
{
    int num1, num2;

    printf("Enter two integers: ");
    if (scanf("%d %d", &num1, &num2) != 2) return 1;

    if (num1 == num2)
        printf("%d is equal to %d\n", num1, num2);
    if (num1 != num2)
        printf("%d is not equal to %d\n", num1, num2);
    if (num1 < num2)
        printf("%d is less than %d\n", num1, num2);
    if (num1 > num2)
        printf("%d is greater than %d\n", num1, num2);
    if (num1 <= num2)
        printf("%d is less than or equal to %d\n", num1, num2);
    if (num1 >= num2)
        printf("%d is greater than or equal to %d\n", num1, num2);
    return 0;
}
```

```
Enter two integers: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

The format `"%d %d"` reads two numbers from one line, and `scanf` reports 2 when it has read both. Each `if` tests its condition; when the condition is true, the statement below it runs, and when it is false, the statement is skipped. Of the six conditions, three are true for 3 and 7, and exactly those three lines are printed. The indentation of the `printf` lines is for people only; C would accept everything on one line, but indentation shows at a glance which statement belongs to which `if`.

In C a condition is simply a number: 0 means false and any other value means true. A comparison such as `3 < 7` produces the `int` 1, and `3 > 7` produces 0.

Listing 1.10 `compare.py`: the same comparisons in Python.

```python
# compare.py - the six comparison operators with if
words = input("Enter two integers: ").split()   # "3 7" -> ["3", "7"]
num1 = int(words[0])
num2 = int(words[1])

if num1 == num2:
    print(f"{num1} is equal to {num2}")
if num1 != num2:
    print(f"{num1} is not equal to {num2}")
if num1 < num2:
    print(f"{num1} is less than {num2}")
if num1 > num2:
    print(f"{num1} is greater than {num2}")
if num1 <= num2:
    print(f"{num1} is less than or equal to {num2}")
if num1 >= num2:
    print(f"{num1} is greater than or equal to {num2}")
```

Listing 1.10 shows the Python form. The condition is followed by a colon, and the statements that belong to the `if` are *indented*: in Python, indentation is not decoration but grammar, and four spaces per level is the universal convention. To read two numbers from one line, the program

calls `split` on the typed text, which cuts `"3 7"` into the list `["3", "7"]`; `words[0]` is its first element and `words[1]` its second, and each is converted with `int`.

> **WARNING.** `=` assigns and `==` compares. In C, `if (x = 5)` is legal: it stores 5 in `x` and then tests the 5, which is true, so the statement always runs. GCC warns about it with `-Wall`; Python refuses it outright. Read `==` aloud as "is equal to" and `=` as "becomes", and the two will never be confused.

1.10 Profitina Laundry Cashier v0.1

We now have every piece for the first version of the laundry cashier: variables for the weight, the price and the total; `scanf` to read the weight; one multiplication; and `printf` for the receipt. **Listing 1.11** puts them together.

Listing 1.11 `receipt.c`: cashier v0.1 prints a receipt for the wash, dry and fold service.

```c
/* receipt.c - Profitina Laundry cashier v0.1: one receipt */
#include <stdio.h>

int main(void)
{
    int weightKg;              /* weight of the laundry, whole kilograms */
    int pricePerKg = 8000;     /* wash, dry & fold: rupiah per kilogram */
    int total;                 /* the amount to pay, in rupiah */

    printf("=== Profitina Laundry - cashier v0.1 ===\n");
    printf("Weight (kg): ");
    if (scanf("%d", &weightKg) != 1) return 1;

    total = weightKg * pricePerKg;

    printf("\n--- RECEIPT ---\n");
    printf("Service  : Wash, dry & fold\n");
    printf("Weight   : %d kg\n", weightKg);
    printf("Price/kg : Rp %d\n", pricePerKg);
    printf("TOTAL    : Rp %d\n", total);
    return 0;
}
```

Each variable carries a comment that states its unit: kilograms, rupiah per kilogram, rupiah. Money is stored as a whole number of rupiah, never as a fraction, because a rupiah amount has no cents and whole numbers are exact. The line `printf("\n--- RECEIPT ---\n");` starts with `\n` to leave an empty line between the input and the receipt. Here is a customer with five kilograms:

```
=== Profitina Laundry - cashier v0.1 ===
Weight (kg): 5

--- RECEIPT ---
Service  : Wash, dry & fold
Weight   : 5 kg
Price/kg : Rp 8000
TOTAL    : Rp 40000
```

Listing 1.12 is the Python version. `print()` with nothing between the parentheses prints an empty line, the counterpart of the leading `\n` in C.

Listing 1.12 `receipt.py`: cashier v0.1 in Python.

```python
# receipt.py - Profitina Laundry cashier v0.1: one receipt
price_per_kg = 8000    # wash, dry & fold: rupiah per kilogram

print("=== Profitina Laundry - cashier v0.1 ===")
weight_kg = int(input("Weight (kg): "))

total = weight_kg * price_per_kg

print()
print("--- RECEIPT ---")
print("Service  : Wash, dry & fold")
print(f"Weight   : {weight_kg} kg")
print(f"Price/kg : Rp {price_per_kg}")
print(f"TOTAL    : Rp {total}")
```

Small as it is, this program already follows the pattern of almost every program you will write: **input** (read the weight), **processing** (compute the total) and **output** (print the receipt). When you plan a new program, ask these three questions first: what comes in, what must be computed, and what goes out?

1.11 Making It Better: Decimal Weights and a Minimum Charge

Cashier v0.1 has two problems that Mbak Rina would notice on its first morning. The scale shows 4.3 kg, but the program accepts only whole kilograms. And the shop's rule, that a bag under three kilograms is charged as three kilograms, is missing. **Listing 1.13** fixes both.

Listing 1.13 `receipt_min.c`: cashier v0.1, improved, with decimal weights and a minimum charge.

```c
/* receipt_min.c - cashier v0.1, improved: decimal weight, 3 kg minimum */
#include <stdio.h>

int main(void)
{
    double weight;              /* kilograms, for example 4.3 */
    double charged;             /* the weight we charge for */
    int pricePerKg = 8000;
    int total;

    printf("=== Profitina Laundry - cashier v0.1 (improved) ===\n");
    printf("Weight (kg): ");
    if (scanf("%lf", &weight) != 1) return 1;   /* %lf reads a double */

    charged = weight;
    if (weight < 3.0) {
        charged = 3.0;          /* the shop's minimum charge */
    }
    total = (int)(charged * pricePerKg + 0.5);   /* round to whole rupiah */

    printf("\n--- RECEIPT ---\n");
    printf("Weight   : %.1f kg\n", weight);
    if (weight < 3.0) {
        printf("Minimum  : 3.0 kg charged\n");
    }
    printf("Price/kg : Rp %d\n", pricePerKg);
    printf("TOTAL    : Rp %d\n", total);
    return 0;
}
```

Three changes are worth a closer look. First, the weight is now a `double`, read with `%lf` ("long float"; `scanf` needs the `l` for a `double`, while `printf` accepts `%f` for both) and printed with `%.1f`, one digit after the decimal point, exactly as the scale shows it. Second, the minimum charge is a

decision: we start by charging the real weight and, *if* it is less than 3.0, replace it by 3.0. The braces around the body of the `if` make it clear which statements belong to it; with more than one statement in the body they are required, and many programmers use them always. Third, the total must be a whole number of rupiah, but `charged * pricePerKg` is a `double`. Adding 0.5 and converting to `int` with `(int)` **rounds to the nearest whole number**: the conversion cuts the fraction off, so 34,400.3 + 0.5 becomes 34,400 and 34,400.7 + 0.5 becomes 34,401. The notation `(int)` is called a **cast**; Chapter 2 uses it the other way round.

Two runs show both paths through the program:

```
=== Profitina Laundry - cashier v0.1 (improved) ===
Weight (kg): 4.3

--- RECEIPT ---
Weight   : 4.3 kg
Price/kg : Rp 8000
TOTAL    : Rp 34400
=== Profitina Laundry - cashier v0.1 (improved) ===
Weight (kg): 2.2

--- RECEIPT ---
Weight   : 2.2 kg
Minimum  : 3.0 kg charged
Price/kg : Rp 8000
TOTAL    : Rp 24000
```

The first bag weighs more than three kilograms and pays for its real weight: 4.3 × 8,000 = 34,400. The second weighs only 2.2 kg, so the receipt says that the minimum was applied and the customer pays for 3 kg, Rp 24,000. **Listing 1.14** is the Python version; `float` converts the typed text into a fractional number, and the format `{weight:.1f}` inside the f-string corresponds to `%.1f`.

Listing 1.14 `receipt_min.py`: the improved cashier in Python.

```
# receipt_min.py - cashier v0.1, improved: decimal weight, 3 kg minimum
price_per_kg = 8000

print("=== Profitina Laundry - cashier v0.1 (improved) ===")
weight = float(input("Weight (kg): "))    # float() accepts 4.3

charged = weight
if weight < 3.0:
    charged = 3.0          # the shop's minimum charge
total = int(charged * price_per_kg + 0.5)   # round to whole rupiah

print()
print("--- RECEIPT ---")
print(f"Weight   : {weight:.1f} kg")
if weight < 3.0:
    print("Minimum  : 3.0 kg charged")
print(f"Price/kg : Rp {price_per_kg}")
print(f"TOTAL    : Rp {total}")
```

Why do we test with 4.3 and 2.2 and not with 5? Because a good test is chosen to exercise each path of the program: one weight above the minimum and one below. A third test, exactly 3.0 kg, checks the edge where the two paths meet; we leave it to Exercise 1.4.

1.12 Using It for Real

The receipt program is tiny, but its shape fits a surprising number of small businesses. A photocopy shop near a campus charges per page, a water refill depot per gallon, a parking lot per hour, a rice seller per kilogram, a tailor per piece. In each case the program reads one quantity, multiplies it by a price, perhaps applies a minimum, and prints a receipt. To adapt `receipt_min` to your own business, change three things: the name of the service in the receipt, the price per unit, and the minimum (or remove the minimum if there is none). Keep the unit in every variable's comment, and keep money in whole rupiah.

Two warnings before a program like this is used at a real counter. First, test it on the cases you know by heart: the smallest bag, the largest, one exactly at the minimum, and one with a decimal weight. Write the expected receipts on paper *before* you run the program, and compare. Second, a program is only as correct as its rules. If the shop changes its price, the program must change with it, which is why Chapter 11 moves prices out of the program and into a file that the owner can edit. For now, a comment at the top that states the prices and the date they were valid is a good habit.

1.13 Common Mistakes, and How to Avoid Them

- **Forgetting the semicolon.** Every C statement ends with `;`. The compiler usually reports the error on the *next* line; look one line up.
- **Forgetting `&` in `scanf`.** `scanf("%d", kilos);` passes the value of `kilos`, a random number, where `scanf` expects an address. GCC with `-Wall` says `format '%d' expects argument of type 'int *', but argument 2 has type 'int'`; if you run it anyway, the program usually crashes. `printf` takes values, `scanf` takes addresses.
- **Mixing up the conversion specifications.** Printing a `double` with `%d`, or reading a `double` with `%f` instead of `%lf`, gives nonsense. GCC warns about both; read the warning.
- **Precedence surprises.** `a + b + c / 3` divides only `c`. Use parentheses whenever an expression mixes `+` or `-` with `*`, `/` or `%`.
- **`=` instead of `==`.** `if (x = 5)` assigns and is always true. Compile with `-Wall` and never ignore the warning.
- **Running an old executable.** After you edit `hello.c`, you must compile again; running `./hello` without compiling runs the old version. When a change seems to have no effect, compile first.
- **Mixed indentation in Python.** Python counts spaces. Let VS Code insert four spaces per level and never mix tabs and spaces.
- **Integer division where a fraction was meant.** This slip deserves a program of its own, **Listing 1.15**, because it compiles without a single warning.

Listing 1.15 `wrong_average.c`: `1 / 2` is 0, so half of 7 kg becomes 0.00 kg.

```c
/* wrong_average.c - a classic slip that compiles without a single warning */
#include <stdio.h>

int main(void)
{
    int kilos = 7;
    double half = 1 / 2 * kilos;          /* 1 / 2 is 0 before anything else */
    printf("half of %d kg = %.2f kg\n", kilos, half);
    printf("half of %d kg = %.2f kg\n", kilos, kilos / 2.0);
    return 0;
}
```

```
half of 7 kg = 0.00 kg
half of 7 kg = 3.50 kg
```

The first line is wrong and the second is right. The Python version, `wrong_average.py` in the code folder, makes the same slip with `1 // 2` and prints the same two lines. The cure is to make one side of the division a `double`, as in `kilos / 2.0`, or to multiply by `0.5`.

1.14 Chapter Summary

- A computer stores data in memory and carries out instructions with its processor; hardware is the machine, software the instructions. Memory is a row of numbered bytes, and the number of a byte is its address.
- High-level languages such as C and Python let us write in terms of the problem; a compiler (C) or an interpreter (Python) translates the program for the processor. C is compiled once into an executable; Python is read and run by the interpreter every time (**Figure 1.1**).
- The workbench of this book is VS Code 1.139 with the C/C++ and Python extensions, GCC (MSYS2 on Windows, `build-essential` on Linux) or Apple clang (Xcode Command Line Tools on macOS), and Python 3.14. Every C program is compiled with `gcc -std=c17 -Wall -Wextra -O2`.
- A C program starts in `main`; `#include <stdio.h>` makes `printf` and `scanf` known; statements end with `;`; `return 0;` reports success. Python runs a file from top to bottom, and indentation is part of its grammar.
- A variable has a name, a type, a value and an address (**Figure 1.2**). `scanf` needs the address (`&`) because it writes into the variable; Python's `input` returns text that `int` or `float` converts.
- Integer division drops the fraction and `%` gives the remainder (Equation (1.1)); in Python `/` always keeps the fraction and `//` is integer division, rounding down rather than toward zero. Precedence puts `*`, `/`, `%` before `+`, `-` (**Table 1.3**); parentheses override it.
- `if` runs a statement only when its condition is true; conditions use `==`, `!=`, `<`, `>`, `<=`, `>=`. `=` assigns, `==` compares.
- Cashier v0.1 reads a weight and prints a receipt; the improved version reads decimal weights, applies a three-kilogram minimum and rounds the total to whole rupiah.

1.15 Exercises

1. Write a program `about_me` in C and Python that prints your name on the first line, your city on the second and your favorite food on the third, using three `printf` (or `print`) statements. Then print the same three lines with a *single* `printf`.
2. Without running it, work out what `printf("%d\n", 17 / 5 * 5 + 17 % 5);` prints. Explain, using Equation (1.1), why the expression gives back 17 for any positive whole numbers in place of 17 and 5.
3. Compute by hand, then check with a program: (a) `10 - 4 - 3`; (b) `10 - (4 - 3)`; (c) `2 * 3 % 4`; (d) `-9 / 4` and `-9 % 4` in C; (e) `-9 // 4` and `-9 % 4` in Python.
4. Run `receipt_min` with the weights 3.0, 0.4 and 12.75. Write the expected receipts first. Which of the three runs tests the edge between the two paths, and what should it print?
5. The laundry adds an express surcharge: every order pays Rp 5,000 on top of the weight charge. Change `receipt.c` and `receipt.py` so that the receipt shows a line `Surcharge: Rp 5000` and the new total. Which lines change, and which stay as they are?

6. A customer pays with a banknote and the cashier must give change using as few notes as possible: Rp 50,000, 20,000, 10,000, 5,000, 2,000 and 1,000 notes, and whatever is left as coins. Write a program `change` that reads the bill and the amount paid and prints how many notes of each kind to give. If the amount paid is too small, it prints how much is missing. Use only `/`, `%` and `if`.

1.16 Solutions to the Exercises

1. In C, three statements such as `printf("Name: Dewi\n");` and one combined statement `printf("Name: Dewi\nCity: Surabaya\nFood: rawon\n");` give the same three lines, because each `\n` starts a new line. In Python, `print("Name: Dewi")` three times, or one `print("Name: Dewi\nCity: Surabaya\nFood: rawon")`.

2. `17 / 5` is 3; `3 * 5` is 15 (the operators `/`, `*` and `%` have the same precedence and are applied from left to right); `17 % 5` is 2; `15 + 2` is 17. The program prints 17. In general `a / b * b` is the largest multiple of `b` not above `a` (for positive numbers), and `a % b` is what is left over, so their sum is `a` again. That is Equation (1.1) read from right to left.

3. (a) Subtraction goes from left to right: `(10 - 4) - 3` is 3. (b) The parentheses first: `10 - 1` is 9. (c) `*` and `%` have equal precedence and go from left to right: `6 % 4` is 2. (d) C cuts toward zero: `-9 / 4` is −2 and `-9 % 4` is −1, since $-9 = (-2) \cdot 4 + (-1)$. (e) Python rounds down: `-9 // 4` is −3 and `-9 % 4` is 3, since $-9 = (-3) \cdot 4 + 3$.

4. 3.0 kg is the edge: the condition `weight < 3.0` is false, so no minimum line is printed and the total is 3.0 × 8,000 = Rp 24,000, the same amount that a smaller bag pays. 0.4 kg is below the minimum: the receipt shows `Weight : 0.4 kg`, the line `Minimum : 3.0 kg charged` and Rp 24,000. 12.75 kg pays 12.75 × 8,000 = Rp 102,000; its weight is printed as `12.8 kg`, because `%.1f` rounds for display only, while the total uses the full 12.75.

5. Add a variable `int surcharge = 5000;`, change the total to `total = weightKg * pricePerKg + surcharge;`, and add `printf("Surcharge: Rp %d\n", surcharge);` before the total line. The input, the price and the other receipt lines stay unchanged. In Python the same three changes: `surcharge = 5000`, `total = weight_kg * price_per_kg + surcharge` and `print(f"Surcharge: Rp {surcharge}")`.

6. Each note is handled the same way: divide by its value to get the number of notes, then keep the remainder for the smaller notes. **Listing 1.16** is the C program; `change.py` in the code folder is the Python version, with `//` in place of `/`.

Listing 1.16 `change.c`: change split into notes with `/` and `%`.

```c
/* change.c - split a customer's change into rupiah notes */
#include <stdio.h>

int main(void)
{
    int bill, paid, change;

    printf("Bill (Rp): ");
    if (scanf("%d", &bill) != 1) return 1;
    printf("Paid (Rp): ");
    if (scanf("%d", &paid) != 1) return 1;

    change = paid - bill;
    if (change < 0) {
        printf("Not enough: Rp %d short\n", -change);
        return 0;
    }
    printf("Change: Rp %d\n", change);
    printf("50000 x %d\n", change / 50000);
    change = change % 50000;          /* what is left after the 50000s */
    printf("20000 x %d\n", change / 20000);
    change = change % 20000;
    printf("10000 x %d\n", change / 10000);
    change = change % 10000;
    printf(" 5000 x %d\n", change / 5000);
    change = change % 5000;
    printf(" 2000 x %d\n", change / 2000);
    change = change % 2000;
    printf(" 1000 x %d\n", change / 1000);
    change = change % 1000;
    printf("coins: Rp %d\n", change);
    return 0;
}
```

```
Bill (Rp): 34400
Paid (Rp): 100000
Change: Rp 65600
50000 x 1
20000 x 0
10000 x 1
 5000 x 1
 2000 x 0
 1000 x 0
coins: Rp 600
```

The change is 65,600: one 50,000 note leaves 15,600; no 20,000 note fits; one 10,000 note leaves 5,600; one 5,000 note leaves 600; no 2,000 and no 1,000 note fit, and Rp 600 is given in coins. A second run with a bill larger than the payment prints only the shortage:

```
Bill (Rp): 34400
Paid (Rp): 30000
Not enough: Rp 4400 short
```

References

[1] D. M. Ritchie, “The development of the C language,” in *Proc. 2nd ACM SIGPLAN Conf. History of Programming Languages (HOPL-II)*, Cambridge, MA, USA, 1993, pp. 201–208.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988.

[3] *Information Technology — Programming Languages — C*, ISO/IEC 9899:2018, International Organization for Standardization, Geneva, Switzerland, 2018.

[4] TIOBE Software, "TIOBE Index for September 2026." Accessed: Sep. 25, 2026. [Online]. Available: https://www.tiobe.com/tiobe-index/

[5] TechRepublic, "TIOBE Index for September 2026: Top 10 Most Popular Programming Languages," Sep. 2026. Accessed: Sep. 25, 2026. [Online]. Available: https://www.techrepublic.com/article/news-tiobe-index-language-rankings/

[6] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. Accessed: Sep. 25, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

[7] Microsoft, "Visual Studio Code updates, version 1.139," Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/updates

[8] GNU Project, "GCC, the GNU Compiler Collection" (GCC 16.2 released August 7, 2026). Accessed: Sep. 25, 2026. [Online]. Available: https://gcc.gnu.org/

[9] Microsoft, "Using GCC with MinGW," *Visual Studio Code Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://code.visualstudio.com/docs/cpp/config-mingw

[10] MSYS2 Project, "MSYS2: Software Distribution and Building Platform for Windows." Accessed: Sep. 25, 2026. [Online]. Available: https://www.msys2.org/

[11] Python Software Foundation, "Using Python on Windows," *Python 3.14 Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.python.org/3.14/using/windows.html

[12] Microsoft, "Using Clang in Visual Studio Code," *Visual Studio Code Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://code.visualstudio.com/docs/cpp/config-clang-mac

[13] Canonical, "Ubuntu 26.04 LTS summary for LTS users," *Ubuntu Release Notes*. Accessed: Sep. 25, 2026. [Online]. Available: https://documentation.ubuntu.com/release-notes/26.04/summary-for-lts-users/

[14] Microsoft, "Getting Started with Python in VS Code," *Visual Studio Code Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://code.visualstudio.com/docs/python/python-tutorial

[15] Python Software Foundation, "Built-in Types: Numeric Types — int, float, complex," *Python 3.14 Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.python.org/3.14/library/stdtypes.html

[16] cppreference.com, "Arithmetic operators (C)." Accessed: Sep. 25, 2026. [Online]. Available: https://en.cppreference.com/w/c/language/operator_arithmetic

Chapter 2

Structured Program Development: Algorithms, Pseudocode and Control

Programs are not typed; they are designed. Before a single line of C or Python is written, a programmer decides *what* must be done and *in which order*, and writes that decision down in a form that is easy to check and easy to change. This chapter teaches that discipline. You will learn what an algorithm is, how to write one in pseudocode, and how to refine a one-sentence description of a task, step by step, into a program. You will meet the result that every program can be built from only three kinds of structure (sequence, selection and repetition), and you will learn the C and Python statements for each: `if`, `if`…`else` and its chains, the conditional operator, and the `while` loop, controlled either by a counter or by a special value called a sentinel. Along the way come the compound assignment operators and the increment operators that every C programmer uses daily. The chapter ends with version 0.2 of the Profitina Laundry cashier, which serves a whole queue of customers, rejects unknown services and prints an end-of-day summary.

2.1 An Evening at the Counter

At seven in the evening Profitina Laundry closes. Pak Hendra, the owner, drops by and asks Mbak Rina the same three questions every day: How many customers came? How much money did we take? And what was the average bill? The answers matter to him. The number of customers tells him whether the new banner at the corner is working; the takings must match the money in the drawer; and the average bill tells him whether customers are choosing the more expensive express service.

With the cashier of Chapter 1, Mbak Rina would have to run the program once per customer and add up forty receipts by hand at closing time, which is exactly the job the program was meant to take away from her. What she needs is a program that keeps running for the whole day, serves one customer after another, and answers Pak Hendra's three questions when the till is closed. That program needs two new abilities. It must **repeat** the same steps for every customer, and it must **choose** between several prices depending on the service. Repetition and selection are the subject of this chapter.

There is a second, less visible lesson in Pak Hendra's questions. "The average bill" sounds simple, but a program must be told precisely what it means: the takings divided by the number of customers, with how many decimals, and what to print on a day on which, because of a power cut, nobody came at all. Deciding such details *before* writing code is the heart of structured program development.

2.2 Algorithms: Actions in the Right Order

An **algorithm** is a procedure for solving a problem, given as a list of **actions** to carry out and the **order** in which to carry them out. Both halves matter. Consider Mbak Rina's routine for a bag of laundry: weigh the bag, write the receipt, put a numbered tag on the bag, put the clothes into the

machine. Change the order to "put the clothes into the machine, then weigh the bag" and the result is a wet bag on the scale and a wrong receipt. The actions are the same; the order is wrong.

> **DEFINITION.** An *algorithm* is a finite list of precise actions, together with the order in which they are executed, that solves a problem for every allowed input. Specifying the order in which the statements of a program run is called *program control*.

Three properties make a list of steps an algorithm rather than a wish. It must be **precise**: "add a little detergent" is fine for a person but useless for a machine, which needs "add 30 ml of detergent". It must be **finite**: it must stop after a finite number of steps for every input. And it must be **correct**: for every allowed input it must produce the right result, including the unusual inputs such as a day with no customers.

Everyday life is full of algorithms. A recipe is one; so are the instructions for assembling a cupboard, the steps for withdrawing money from an ATM, and the rules by which a referee decides a penalty. What distinguishes the algorithms in this book is only that the one who carries them out is a computer, which follows them *exactly*, without the common sense that lets a person repair a vague recipe. That is why we write them down carefully first.

Let us test these properties on a familiar task: making a glass of sweet iced tea for a customer who waits at the counter. A first attempt might read: *put a tea bag in a glass, pour water, add sugar, add ice, stir, serve*. Read it as a machine would, and the gaps appear at once. Which water, hot or cold? A tea bag in cold water gives colored water, not tea, so the algorithm is not correct. How much sugar? "Add sugar" is not precise. And "stir" does not say when to stop, so it is not even finite. A better version says: *boil 250 ml of water; pour it over one tea bag; wait three minutes; remove the bag; stir in two teaspoons of sugar until no grains are visible; add ice until the glass is full; serve*. Every action is now precise, every step ends, and the order is right: the sugar dissolves in the hot tea, which it would not do in iced tea. The same questions (is each step precise, does each step end, is the order right, is the result right for every allowed input?) are the questions we ask of every algorithm in this book.

2.3 Pseudocode

Pseudocode is an informal language for writing algorithms. It looks like a mixture of English and a programming language, and it has no fixed grammar: its only purpose is to help you think about the steps and their order before you think about semicolons and braces. Pseudocode is never compiled or run; it is read by people, at design time, where changing a line costs seconds.

In this book, pseudocode follows a few simple conventions, which you can see in **Pseudocode 2.1**, the algorithm of the receipt program of Chapter 1:

Pseudocode 2.1 Pseudocode for the receipt program of Chapter 1.

```
RECEIPT
1  price ← 8000
2  print "Weight (kg): "
3  read weight
4  total ← weight × price
5  print the receipt with weight, price and total
```

The name of the procedure stands at the top. Each line is numbered so that we can refer to it. The arrow `←` means assignment ("becomes"), so that it is not confused with the `=` of comparison. Words such as `read`, `print`, `if`, `else` and `while` have their usual meaning, and indentation shows

which lines belong together, exactly as in Python. Pseudocode contains only *executable* actions: declarations such as `int weight;` are not actions and are left out.

A good rule for every program in this book: spend the first ten minutes away from the keyboard. Read the problem, write down what comes in and what must come out, and write the pseudocode. Programmers who design first almost always finish sooner than those who start typing at once, because the mistakes they catch on paper never have to be found in a running program.

2.4 Three Structures Are Enough

2.4.1 Sequence, selection and repetition

Early programs were often a tangle of jumps. A statement called `goto` let the program continue at any other line, and programs that used it freely were so hard to follow that they were nicknamed "spaghetti code". In 1966 Corrado Böhm and Giuseppe Jacopini proved a remarkable result: *every* algorithm can be written using only three kinds of control structure, combined by stacking and nesting [1]. Two years later Edsger Dijkstra's letter "Go To Statement Considered Harmful" argued that programmers should therefore stop using `goto` [2], and the style that followed, **structured programming**, is the style of this book and of almost all programs written today [3].

Figure 2.1 shows the three structures as **flowcharts**, diagrams in which a rectangle is an action, a diamond is a decision, and arrows show the order.

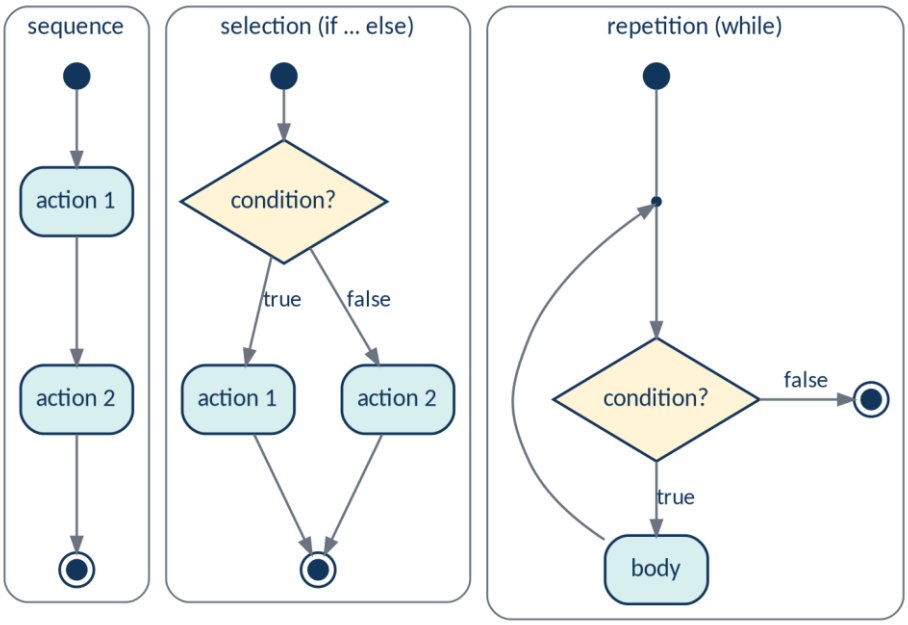


Figure 2.1 The three control structures: sequence, selection and repetition; each has exactly one entry and one exit.

- In a **sequence**, actions run one after another, in the order they are written. Every program so far has been a sequence.
- In a **selection**, a condition decides which of two paths runs. C and Python offer `if` (one path or nothing), `if`…`else` (one of two paths), and chains of these for many paths; C also has `switch`, which Chapter 3 covers.

- In a **repetition**, an action runs again and again while a condition holds. C offers `while`, `for` and `do`…`while`; Python offers `while` and `for`. This chapter uses `while`; Chapter 3 adds the others.

The most important property is visible in the figure: every structure has **exactly one entry and one exit**. That is what makes structured programs readable. You can replace any rectangle in the figure by a whole structure of its own, which is called **nesting**, or place structures one after another, which is called **stacking**, and the result still reads from top to bottom like a page of text.

2.4.2 Reading a flowchart

Flowcharts use a small set of symbols. A filled circle marks where the structure is entered, and a circle inside a ring marks where it is left; many books draw these as ovals labeled *start* and *end*. A rounded rectangle is an action, such as an assignment or a calculation. A diamond is a decision: it contains a condition, and exactly two arrows leave it, one labeled *true* (or *yes*) and one labeled *false* (or *no*). Arrows show the order in which things happen. Some books draw input and output as parallelograms; in this book they are ordinary actions, such as “print prompt, read weight” in **Figure 2.2**.

To read a flowchart, put your finger on the entry and follow the arrows, choosing a branch at every diamond according to the data you have in mind. Flowcharts are excellent for *seeing* the shape of a small algorithm, especially where a loop goes back, and for explaining it to someone else; for larger algorithms they become crowded, and pseudocode is easier to write and to change. This book uses both: flowcharts to show a structure once, pseudocode to design programs.

2.4.3 Blocks

A structure often needs to control more than one statement. C groups several statements into one **block** (also called a *compound statement*) with braces: wherever C allows one statement, it allows `{`, any number of statements, and `}`. Python forms blocks by indentation: all lines indented one level more than the `if` or `while` line belong to it, and the block ends when the indentation returns. This is why indentation matters in both languages, for the compiler in Python and for the reader in C.

2.5 Selection

2.5.1 `if` and `if`…`else`

Chapter 1 used the single-selection `if`: a statement runs when the condition is true and is skipped otherwise. The **double-selection** `if`…`else` chooses between two actions:

```
if (weight < 3.0)
    printf("Minimum charge applied\n");
else
    printf("Charged by weight\n");
```

When the condition is true, the first statement runs; otherwise the one after `else` runs. Exactly one of the two always runs. In Python the same choice is written with colons and indentation, `if weight < 3.0:` followed by an indented line, then `else:` and another indented line.

2.5.2 Chains of decisions

Often there are more than two cases. Profitina Laundry gives a discount on large bills: 10% on a bill of Rp 100,000 or more, 5% on a bill of Rp 50,000 or more, and nothing below that. Customers with a member card get five percentage points more. The algorithm is **Pseudocode 2.2**.

Pseudocode 2.2 Pseudocode for the discount: a chain of three cases, then a separate decision for members.

```
DISCOUNT
1  read bill, member
2  if bill ≥ 100000
3      percent ← 10
4  else if bill ≥ 50000
5      percent ← 5
6  else
7      percent ← 0
8  if member = 1
9      percent ← percent + 5
10 discount ← bill × percent / 100   (whole rupiah)
11 print member or guest, percent, discount, bill − discount
```

In symbols, with bill b and percentage p, the discount in whole rupiah is

$$d = \left\lfloor \frac{b \cdot p}{100} \right\rfloor \tag{2.1}$$

where $\lfloor x \rfloor$, the *floor* of x, is the largest whole number that is not greater than x. For positive numbers this is exactly what C's integer division computes, so Equation (2.1) becomes `bill * percent / 100` in C and `bill * percent // 100` in Python. Multiplying before dividing matters: `bill / 100 * percent` would throw away the last two digits of the bill before multiplying.

The order of the tests in the chain matters, too. The conditions are checked from the top, and the first one that is true wins; the rest are skipped. A bill of Rp 120,000 satisfies both `bill >= 100000` and `bill >= 50000`, and because the 10% test comes first, it gets 10%. Had we written the 5% test first, every large bill would get only 5%. When the cases of a chain overlap, put the most demanding condition first. **Listing 2.1** is the program.

Listing 2.1 `discount.c`: an `if`…`else if`…`else` chain, a separate `if`, and the conditional operator.

```c
/* discount.c - nested if...else and the conditional operator ?: */
#include <stdio.h>

int main(void)
{
    int bill;       /* the bill before discount, rupiah */
    int member;     /* 1 = has a member card, 0 = no card */
    int percent;

    printf("Bill (Rp): ");
    if (scanf("%d", &bill) != 1) return 1;
    printf("Member card? (1 = yes, 0 = no): ");
    if (scanf("%d", &member) != 1) return 1;

    if (bill >= 100000) {
        percent = 10;
    } else if (bill >= 50000) {
        percent = 5;
    } else {
        percent = 0;
    }
    if (member == 1) {
        percent = percent + 5;      /* members get five points more */
    }

    int discount = bill * percent / 100;
    printf("Customer : %s\n", member == 1 ? "member" : "guest");
    printf("Discount : %d%% = Rp %d\n", percent, discount);
    printf("To pay   : Rp %d\n", bill - discount);
    return 0;
}
```

Formally, C has no `else if`: the second `if` is simply the statement that follows the first `else`, and the chain is a *nested* `if`…`else` written without extra indentation. Written with full indentation it would drift further and further to the right, so every C programmer writes it as shown. Here are three customers:

```
Bill (Rp): 120000
Member card? (1 = yes, 0 = no): 1
Customer : member
Discount : 15% = Rp 18000
To pay   : Rp 102000
Bill (Rp): 60000
Member card? (1 = yes, 0 = no): 0
Customer : guest
Discount : 5% = Rp 3000
To pay   : Rp 57000
Bill (Rp): 30000
Member card? (1 = yes, 0 = no): 1
Customer : member
Discount : 5% = Rp 1500
To pay   : Rp 28500
```

The first customer has a large bill and a member card: 10% plus 5% is 15%, and 15% of Rp 120,000 is Rp 18,000. The second gets the middle case, 5% of Rp 60,000. The third falls through the chain to 0% but gets the member's 5%, which is Rp 1,500 on Rp 30,000.

2.5.3 The conditional operator

The last `printf` in **Listing 2.1** contains C's only operator with three operands, the **conditional operator** `?:`. The expression `member == 1 ? "member" : "guest"` is read "if `member` equals 1, then `"member"`, otherwise `"guest"`". It is an `if`…`else` that produces a *value* instead of running a

statement, which lets it sit inside a `printf`. Use it for small choices of a value; when either branch does real work, write a full `if`…`else`, which is easier to read.

Listing 2.2 shows the Python version. Python writes the chain with the keyword `elif`, a contraction of "else if" [4], and its conditional expression puts the condition in the middle: `'member' if member == 1 else 'guest'`.

Listing 2.2 `discount.py`: `if`, `elif` and `else`, and Python's conditional expression.

```python
# discount.py - if/elif/else and the conditional expression
bill = int(input("Bill (Rp): "))
member = int(input("Member card? (1 = yes, 0 = no): "))

if bill >= 100000:
    percent = 10
elif bill >= 50000:
    percent = 5
else:
    percent = 0
if member == 1:
    percent = percent + 5           # members get five points more

discount = bill * percent // 100
print(f"Customer : {'member' if member == 1 else 'guest'}")
print(f"Discount : {percent}% = Rp {discount}")
print(f"To pay   : Rp {bill - discount}")
```

> **WARNING.** In C, an `else` belongs to the nearest `if` above it that has no `else` yet, whatever the indentation suggests. In `if (a) if (b) x(); else y();` the `else` belongs to `if (b)`. This is called the *dangling else*. Braces remove every doubt, which is one reason why this book uses them whenever an `if` contains another `if`.

2.6 Repetition Controlled by a Counter

2.6.1 The `while` loop

The `while` statement repeats a statement, or a block, as long as a condition is true:

```
while (condition) {
    statements
}
```

First the condition is tested. If it is false, the whole loop is skipped and the program continues after the closing brace. If it is true, the body runs once, and then the condition is tested again. Something inside the body must eventually make the condition false; otherwise the loop runs forever, which is called an **infinite loop**. One run of the body is called an **iteration**. The exact rules for `if`, `while` and the other statements of C are collected in the C reference [5].

2.6.2 Counting to five

Suppose Mbak Rina wants the average weight of the first five orders of the morning. The number of repetitions is known in advance, so a variable called a **counter** controls the loop: it starts at 1, the loop runs while it is at most 5, and each iteration adds 1 to it. This is **counter-controlled repetition**, and **Pseudocode 2.3** describes it.

Pseudocode 2.3 Counter-controlled repetition: exactly five weights are read.

```
AVERAGE-OF-FIVE
1  total ← 0
2  counter ← 1
3  while counter ≤ 5
4      print "Weight of order counter (kg): "
5      read weight
6      total ← total + weight
7      counter ← counter + 1
8  average ← total / 5   (whole kilograms)
9  print total and average
```

The variable `total` is an **accumulator**: it starts at 0 and each iteration adds one weight to it, so that after the loop it holds the sum. Forgetting to set it to 0 first is a classic mistake; in C an uninitialized local variable holds whatever happened to be in that memory, and the sum is garbage. **Listing 2.3** is the C program.

Listing 2.3 `average_counter.c`: a `while` loop controlled by a counter.

```c
/* average_counter.c - average weight of exactly five orders */
#include <stdio.h>

int main(void)
{
    int counter = 1;    /* number of the order being read */
    int weight;         /* weight of one order, kg */
    int total = 0;      /* running total of the weights */

    while (counter <= 5) {                  /* repeat exactly five times */
        printf("Weight of order %d (kg): ", counter);
        if (scanf("%d", &weight) != 1) return 1;
        total = total + weight;
        counter = counter + 1;
    }
    printf("Total weight: %d kg\n", total);
    printf("Average weight: %d kg\n", total / 5);   /* whole-number division */
    return 0;
}
```

```
Weight of order 1 (kg): 4
Weight of order 2 (kg): 7
Weight of order 3 (kg): 3
Weight of order 4 (kg): 6
Weight of order 5 (kg): 6
Total weight: 26 kg
Average weight: 5 kg
```

The loop body ran five times, once for each value of `counter` from 1 to 5; after the fifth iteration `counter` became 6, the condition `counter <= 5` was false, and the program continued with the first `printf` after the loop. The true average of 4, 7, 3, 6 and 6 is 26/5 = 5.2, but the program prints 5, because `total / 5` divides two integers. The next section fixes this.

Listing 2.4 is the same loop in Python. Note how the prompt uses an f-string to include the order number, where C uses `%d`.

Listing 2.4 `average_counter.py`: the same counter-controlled loop in Python.

```python
# average_counter.py - average weight of exactly five orders
counter = 1     # number of the order being read
total = 0       # running total of the weights

while counter <= 5:                          # repeat exactly five times
    weight = int(input(f"Weight of order {counter} (kg): "))
    total = total + weight
    counter = counter + 1

print(f"Total weight: {total} kg")
print(f"Average weight: {total // 5} kg")    # whole-number division
```

2.7 Top-Down, Stepwise Refinement: Repetition Controlled by a Sentinel

2.7.1 A harder problem

Pak Hendra's real question is harder: *what is the average weight of all orders of the day?* Nobody knows in the morning how many orders there will be. We need a way to tell the program that the data has ended. A common solution is a **sentinel value**, also called a *flag value* or *signal value*: a special value that cannot be a real piece of data and therefore means "no more data". No bag of laundry weighs 0 kg, so 0 can be our sentinel. When the cashier types 0, the day is over. This is **sentinel-controlled repetition**.

To design the program we use a technique that works for problems of any size: **top-down, stepwise refinement** [6]. We start from a single statement of the whole task, the **top**, and refine it again and again, each time splitting a statement into a few smaller ones, until every line is simple enough to be translated directly into C or Python.

2.7.2 The top and the first refinement

The top, **Pseudocode 2.4**, is one sentence:

Pseudocode 2.4 The top: one sentence that states the whole task.

```
AVERAGE-WEIGHT
1  determine the average weight of the day's orders
```

This is complete but far too vague to program. The first refinement, **Pseudocode 2.5**, divides it into three phases that almost every program has, initialize, process and finish:

Pseudocode 2.5 The first refinement: initialize, process, finish.

```
AVERAGE-WEIGHT
1  initialize the variables
2  read, add and count the weights until the sentinel arrives
3  compute and print the average
```

2.7.3 The second refinement

Now each line is refined in turn. *Initialize the variables*: we need a running `total` and a `count` of the orders, both starting at 0. *Read, add and count the weights until the sentinel arrives*: this is a loop, and here the refinement reveals a subtle point. The loop must test the weight *before* it adds it, because the sentinel 0 must never be counted as an order. So we read the first weight *before* the loop, test it at the top of the loop, and read the next weight at the *end* of the body. *Compute and print the average*: here the refinement reveals a second point, which the one-line top hid

completely. If the very first value is the sentinel, `count` is 0, and dividing by 0 would be a disaster. The program must check this case. **Pseudocode 2.6** is the result.

Pseudocode 2.6 The second refinement: sentinel-controlled repetition with a guard against division by zero.

```
AVERAGE-WEIGHT
1  total ← 0
2  count ← 0
3  print "Weight in kg (0 to finish): "
4  read weight
5  while weight ≠ 0
6      total ← total + weight
7      count ← count + 1
8      print "Weight in kg (0 to finish): "
9      read weight
10 if count ≠ 0
11     average ← total / count   (with decimals)
12     print count, total and average
13 else
14     print "No orders were entered."
```

Figure 2.2 draws the same algorithm as a flowchart. Follow the arrows: the first weight is read once; the diamond tests it; the loop adds, counts, reads the next weight and returns to the diamond; only when the sentinel arrives does the path leave the loop, and a second diamond protects the division.

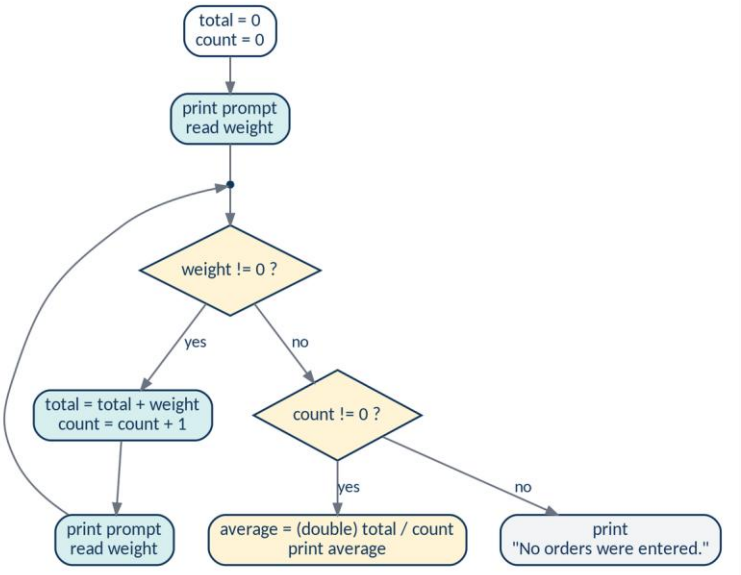


Figure 2.2 The sentinel-controlled average: read before the loop, test at the top, read again at the end of the body, and divide only when there is at least one order.

Every line of **Pseudocode 2.6** can now be translated directly, and the refinement is finished. Notice that the two most important decisions of the program, reading ahead and guarding the division, were discovered on paper. The mathematical definition we are implementing is the ordinary mean: for n weights $w_1, w_2, \ldots, w_n$,

$$\bar{w} = \frac{w_1 + w_2 + \cdots + w_n}{n} = \frac{1}{n}\sum_{i=1}^{n} w_i \tag{2.2}$$

and Equation (2.2) is defined only for $n \geq 1$, which is exactly the case that line 10 of the pseudocode checks.

2.7.4 The program

Listing 2.5 is the translation into C.

Listing 2.5 `average_sentinel.c`: sentinel-controlled repetition, a cast, and a guard against dividing by zero.

```c
/* average_sentinel.c - average weight of any number of orders; 0 ends */
#include <stdio.h>

int main(void)
{
    int weight;         /* weight of one order, or the sentinel 0 */
    int total = 0;
    int count = 0;

    printf("Weight in kg (0 to finish): ");          /* read before the loop */
    if (scanf("%d", &weight) != 1) return 1;
    while (weight != 0) {
        total = total + weight;
        count = count + 1;
        printf("Weight in kg (0 to finish): ");      /* read again at the end */
        if (scanf("%d", &weight) != 1) return 1;
    }

    if (count != 0) {
        double average = (double) total / count;     /* cast: keep the fraction */
        printf("Orders: %d, total %d kg\n", count, total);
        printf("Average weight: %.2f kg\n", average);
    } else {
        printf("No orders were entered.\n");
    }
    return 0;
}
```

```
Weight in kg (0 to finish): 4
Weight in kg (0 to finish): 7
Weight in kg (0 to finish): 3
Weight in kg (0 to finish): 0
Orders: 3, total 14 kg
Average weight: 4.67 kg
Weight in kg (0 to finish): 0
No orders were entered.
```

The first run averages three orders: 14 kg divided by 3 is 4.666..., printed with two decimals as 4.67. The second run is the day of the power cut: the very first value is the sentinel, the loop body never runs, and instead of dividing by zero the program says so.

2.7.5 Checking the design by hand: a trace table

Before we trust a loop, we can *execute it by hand*. A **trace table** has one column for each variable and one row for each step; we write down the value of every variable after each line that changes it. **Table 2.1** traces **Pseudocode 2.6** for the input 4, 7, 3, 0, the same input as the first run.

Table 2.1 A trace of **Pseudocode 2.6** for the input 4, 7, 3, 0: the sentinel is read at the end of the third iteration and is never added.

Step (line)	weight	total	count	weight ≠ 0?
before the loop (1–4)	4	0	0	yes
1st iteration (6–9)	7	4	1	yes
2nd iteration (6–9)	3	11	2	yes
3rd iteration (6–9)	0	14	3	no, leave the loop
after the loop (10–12)	0	14	3	average 14 / 3

The table answers the questions that matter before any code exists. How many times does the body run? Three, once per real weight. Is the sentinel ever added to `total`? No: it is read at the end of the third iteration and immediately stops the loop. Is `count` right? Yes, 3. A trace table takes two minutes and finds most loop errors, especially the ones where a loop runs once too often or once too few, which programmers call *off-by-one errors*.

2.7.6 The cast

The average in the first run has a fraction because of one small piece of code, `(double) total`. Both `total` and `count` are `int`s, and dividing them would throw the fraction away, as in the counter program. The **cast operator** `(double)` makes a temporary `double` copy of `total`; the variable itself stays an `int`. Now the division has a `double` on one side and an `int` on the other, and C applies a rule called **promotion**: before an operation, the operand of the "smaller" type is converted to the "larger" type. So `count` is converted to `double` as well, and the division keeps its fraction.

The position of the cast matters. `(double) (total / count)` looks similar but is wrong: the parentheses make the integer division happen *first*, the fraction is already lost, and the cast only turns 4 into 4.0. Cast an operand, not a finished result.

Python needs no cast: its `/` always produces a fraction. **Listing 2.6** is the Python program.

Listing 2.6 `average_sentinel.py`: the same algorithm in Python.

```python
# average_sentinel.py - average weight of any number of orders; 0 ends
total = 0
count = 0

weight = int(input("Weight in kg (0 to finish): "))      # read before the loop
while weight != 0:
    total = total + weight
    count = count + 1
    weight = int(input("Weight in kg (0 to finish): "))  # read again at the end

if count != 0:
    average = total / count                  # / always keeps the fraction
    print(f"Orders: {count}, total {total} kg")
    print(f"Average weight: {average:.2f} kg")
else:
    print("No orders were entered.")
```

The format `{average:.2f}` rounds to two decimals for printing, like `%.2f` in C. Both languages round the same value in the same way, because a Python `float` and a C `double` are the same kind of number.

> **KEY INSIGHT.** The two loop patterns of this chapter are worth memorizing. *Counter-controlled:* initialize the counter, test it at the top, increase it at the end of the body. *Sentinel-*

> *controlled:* read the first value before the loop, test it at the top, process it, and read the next value at the end of the body. Almost every loop bug in a first program comes from mixing the two, such as reading only inside the loop and then processing the sentinel as if it were data.

2.8 Nested Control Structures

Control structures can be placed inside each other. Profitina Laundry asks ten customers at the end of the week whether they were satisfied: 1 means satisfied, 2 means not satisfied. If more than eight are satisfied, Pak Hendra treats the team to lunch. The loop runs a known number of times, ten, so it is counter-controlled; inside the loop, each answer must be classified, so an `if`…`else` sits *inside* the `while`. After the loop, another decision checks the result. **Pseudocode 2.7** shows the design.

Pseudocode 2.7 Pseudocode for the survey: a selection nested inside a counter-controlled repetition, followed by a second selection.

```
SURVEY
1  satisfied ← 0
2  unhappy ← 0
3  customer ← 1
4  while customer ≤ 10
5      read answer
6      if answer = 1
7          satisfied ← satisfied + 1
8      else
9          unhappy ← unhappy + 1
10     customer ← customer + 1
11 print satisfied and unhappy
12 if satisfied > 8
13     print "Excellent week: treat the team to lunch!"
```

Listing 2.7 `survey.c`: an `if`…`else` nested in a `while`.

```c
/* survey.c - ten customers rate the service: 1 = satisfied, 2 = not */
#include <stdio.h>

int main(void)
{
    int satisfied = 0;
    int unhappy = 0;
    int customer = 1;
    int answer;

    while (customer <= 10) {
        printf("Customer %d (1 = satisfied, 2 = not): ", customer);
        if (scanf("%d", &answer) != 1) return 1;
        if (answer == 1) {                  /* an if inside a while */
            satisfied = satisfied + 1;
        } else {
            unhappy = unhappy + 1;
        }
        customer = customer + 1;
    }
    printf("Satisfied: %d\n", satisfied);
    printf("Not satisfied: %d\n", unhappy);
    if (satisfied > 8) {
        printf("Excellent week: treat the team to lunch!\n");
    }
    return 0;
}
```

```
Customer 1 (1 = satisfied, 2 = not): 1
Customer 2 (1 = satisfied, 2 = not): 1
Customer 3 (1 = satisfied, 2 = not): 2
Customer 4 (1 = satisfied, 2 = not): 1
Customer 5 (1 = satisfied, 2 = not): 1
Customer 6 (1 = satisfied, 2 = not): 1
Customer 7 (1 = satisfied, 2 = not): 1
Customer 8 (1 = satisfied, 2 = not): 1
Customer 9 (1 = satisfied, 2 = not): 1
Customer 10 (1 = satisfied, 2 = not): 1
Satisfied: 9
Not satisfied: 1
Excellent week: treat the team to lunch!
```

Nine of ten are satisfied, which is more than eight, so the closing message appears. **Listing 2.7** translates **Pseudocode 2.7** line by line; the only additions are the prompt, which tells the customer what to type, and the check on `scanf`. **Listing 2.8** is the Python version, with exactly the same structure of `while`, `if` and `else`, and it prints the same lines.

Listing 2.8 `survey.py`: the same nested structure in Python, where indentation alone shows that the `if` belongs to the `while`.

```python
# survey.py - ten customers rate the service: 1 = satisfied, 2 = not
satisfied = 0
unhappy = 0
customer = 1

while customer <= 10:
    answer = int(input(f"Customer {customer} (1 = satisfied, 2 = not): "))
    if answer == 1:                          # an if inside a while
        satisfied = satisfied + 1
    else:
        unhappy = unhappy + 1
    customer = customer + 1

print(f"Satisfied: {satisfied}")
print(f"Not satisfied: {unhappy}")
if satisfied > 8:
    print("Excellent week: treat the team to lunch!")
```

Compare the two listings once more. In C, the braces show where the body of the `while` ends; the closing brace after `customer = customer + 1;` is the end of the loop, and the `printf` lines after it run once. In Python, the same information is carried by indentation alone: `customer = customer + 1` is indented one level, so it belongs to the loop, while the `print` lines start at the left margin, so they run after it. Move `customer = customer + 1` one level to the left in Python, and the loop never ends; move it outside the closing brace in C, and the same happens. Nested structures are easy to read only when the indentation tells the truth, which is why VS Code's automatic formatting is worth using. The program assumes that every answer is 1 or 2: any other number is counted as "not satisfied". Exercise 2.4 asks you to reject such answers instead.

2.9 Assignment Operators and Increment Operators

2.9.1 Compound assignment

Statements such as `total = total + weight;` are so common that C has a shorter form: `total += weight;`. Every binary arithmetic operator has such a **compound assignment operator**, listed in **Table 2.2**. The variable is named only once, which saves typing and removes the chance of writing `total = totl + weight` by accident.

Table 2.2 The compound assignment operators of C; Python has the same operators, with `//=` for integer division.

Operator	Example	Means	Result if c was 12
`+=`	`c += 7`	`c = c + 7`	19
`-=`	`c -= 4`	`c = c - 4`	8
`*=`	`c *= 5`	`c = c * 5`	60
`/=`	`c /= 3`	`c = c / 3`	4
`%=`	`c %= 5`	`c = c % 5`	2

2.9.2 Increment and decrement

Adding 1 is the most common change of all, and C has two operators just for it: `++` adds 1 and `--` subtracts 1. Each comes in two forms. The **prefix** form `++c` adds 1 first and then uses the new value in the surrounding expression; the **postfix** form `c++` uses the old value and adds 1 afterwards. When the operator stands alone as a statement, `++c;` and `c++;` do the same thing; the difference only shows when the value is used in the same expression. **Listing 2.9** shows both operators at work.

Listing 2.9 `increment.c`: compound assignment, and prefix versus postfix increment.

```c
/* increment.c - compound assignment, and ++c versus c++ */
#include <stdio.h>

int main(void)
{
    int total = 40000;
    total += 12000;                 /* total = total + 12000 */
    printf("total after += : %d\n", total);
    total -= 2000;
    printf("total after -= : %d\n", total);
    total *= 2;
    printf("total after *= : %d\n", total);
    total /= 4;
    printf("total after /= : %d\n", total);
    total %= 7;
    printf("total after %%= : %d\n", total);

    int c = 5;
    printf("c++ shows %d, ", c++);  /* use 5, then add 1 */
    printf("afterwards c = %d\n", c);
    c = 5;
    printf("++c shows %d, ", ++c);  /* add 1, then use 6 */
    printf("afterwards c = %d\n", c);
    return 0;
}
```

```
total after += : 52000
total after -= : 50000
total after *= : 100000
total after /= : 25000
total after %= : 3
c++ shows 5, afterwards c = 6
++c shows 6, afterwards c = 6
```

Follow the first half line by line: 40,000 plus 12,000 is 52,000; minus 2,000 is 50,000; times 2 is 100,000; divided by 4 is 25,000; and 25,000 modulo 7 is 3, because $25000 = 3571 \cdot 7 + 3$. In the second half, `c++` shows 5 and leaves 6 behind, while `++c` shows 6 at once.

Python deliberately has no `++` or `--`: `c += 1` is the only way, and it is a statement, not an expression, so the question of "before or after" never arises. **Listing 2.10** imitates the two C forms by writing the two steps in the two orders.

Listing 2.10 `increment.py`: the same results in Python, which has compound assignment but no `++`.

```
# increment.py - compound assignment; Python has no ++ operator
total = 40000
total += 12000                  # total = total + 12000
print(f"total after += : {total}")
total -= 2000
print(f"total after -= : {total}")
total *= 2
print(f"total after *= : {total}")
total //= 4                     # // keeps it a whole number, like C's /
print(f"total after /= : {total}")
total %= 7
print(f"total after %= : {total}")

c = 5
print(f"c++ shows {c}, ", end="")    # use 5 ...
c += 1                               # ... then add 1
print(f"afterwards c = {c}")
c = 5
c += 1                               # add 1 first ...
print(f"++c shows {c}, ", end="")    # ... then use 6
print(f"afterwards c = {c}")
```

> **WARNING.** Never use a variable twice in an expression that also increments it, such as `printf("%d %d", c, c++);` or `x = x++;`. The C standard does not say in which order the parts are evaluated, and the result may differ between compilers or even between optimization levels. GCC warns with `-Wall` (`operation on 'c' may be undefined`). Increment in a statement of its own.

2.10 Profitina Laundry Cashier v0.2

Everything is ready for a cashier that stays open all day. It must repeat for every customer until the till is closed (sentinel-controlled repetition, with 0 as the sentinel for "close the till"), choose the price for the service (a selection chain), reject unknown services, and accumulate the takings and the number of customers (compound assignment and increment). At closing time it answers Pak Hendra's three questions. **Pseudocode 2.8** is the design after two refinements.

Pseudocode 2.8 The design of cashier v0.2: a sentinel loop around a selection, and a guarded average at the end.

```
CASHIER-V02
1  revenue ← 0
2  customers ← 0
3  print the menu
4  read service
5  while service ≠ 0
6      if service is 1, 2 or 3
7          price ← the price of that service
8          read weight
9          bill ← weight × price
10         revenue ← revenue + bill
11         customers ← customers + 1
12         print the bill
13     else
14         print "Unknown service, try again."
15     read service
16 print customers and revenue
17 if customers > 0
18     print revenue / customers   (with two decimals)
```

Listing 2.11 is the C program. It follows the pseudocode closely, with one C-specific detail: an unknown service prints its message, reads the next service, and uses the statement `continue`, which skips the rest of the body and jumps straight back to the loop test. Without it, the program would go on to ask for a weight for a service that does not exist.

Listing 2.11 `cashier_v02.c`: cashier v0.2 serves a queue of customers until the till is closed.

```c
/* cashier_v02.c - Profitina Laundry cashier v0.2: a queue of customers */
#include <stdio.h>

int main(void)
{
    int service;            /* 1..3, or the sentinel 0 = close the till */
    int weightKg;
    int price;              /* price per kg of the chosen service */
    int bill;               /* bill of one customer */
    int revenue = 0;        /* all bills of the day */
    int customers = 0;

    printf("=== Profitina Laundry - cashier v0.2 ===\n");
    printf("1 = Wash, dry & fold (8000/kg)  2 = Express (12000/kg)\n");
    printf("3 = Ironing only (6000/kg)      0 = close the till\n");
    printf("Service: ");
    if (scanf("%d", &service) != 1) return 1;

    while (service != 0) {
        if (service == 1) {
            price = 8000;
        } else if (service == 2) {
            price = 12000;
        } else if (service == 3) {
            price = 6000;
        } else {
            printf("Unknown service, try again.\n");
            printf("Service: ");
            if (scanf("%d", &service) != 1) return 1;
            continue;       /* back to the loop test */
        }
        printf("Weight (kg): ");
        if (scanf("%d", &weightKg) != 1) return 1;
        bill = weightKg * price;
        revenue += bill;
        ++customers;
        printf("Customer #%d pays Rp %d\n", customers, bill);
        printf("Service: ");
        if (scanf("%d", &service) != 1) return 1;
    }
```

Listing 2.11 (Continued) *cashier_v02.c: cashier v0.2 serves a queue of customers until the till is closed.*

```c
    printf("\n--- END OF DAY ---\n");
    printf("Customers served : %d\n", customers);
    printf("Revenue          : Rp %d\n", revenue);
    if (customers > 0) {
        printf("Average bill     : Rp %.2f\n", (double) revenue / customers);
    }
    return 0;
}
```

Here is a short day: a 5 kg wash, a 3 kg express order, a mistyped service 7, a 2 kg ironing order, and closing time.

```
=== Profitina Laundry - cashier v0.2 ===
1 = Wash, dry & fold (8000/kg)  2 = Express (12000/kg)
3 = Ironing only (6000/kg)      0 = close the till
Service: 1
Weight (kg): 5
Customer #1 pays Rp 40000
Service: 2
Weight (kg): 3
Customer #2 pays Rp 36000
Service: 7
Unknown service, try again.
Service: 3
Weight (kg): 2
Customer #3 pays Rp 12000
Service: 0

--- END OF DAY ---
Customers served : 3
Revenue          : Rp 88000
Average bill     : Rp 29333.33
```

Check the numbers by hand, as Pak Hendra would: 5 × 8,000 = 40,000; 3 × 12,000 = 36,000; 2 × 6,000 = 12,000. The total is Rp 88,000 from three customers, and the average is 88,000 / 3 = 29,333.33. The mistyped 7 was rejected and not counted. **Listing 2.12** is the Python version.

Listing 2.12 `cashier_v02.py`: cashier v0.2 in Python.

```python
# cashier_v02.py - Profitina Laundry cashier v0.2: a queue of customers
revenue = 0             # all bills of the day
customers = 0

print("=== Profitina Laundry - cashier v0.2 ===")
print("1 = Wash, dry & fold (8000/kg)  2 = Express (12000/kg)")
print("3 = Ironing only (6000/kg)      0 = close the till")
service = int(input("Service: "))

while service != 0:
    if service == 1:
        price = 8000
    elif service == 2:
        price = 12000
    elif service == 3:
        price = 6000
    else:
        print("Unknown service, try again.")
        service = int(input("Service: "))
        continue        # back to the loop test
    weight_kg = int(input("Weight (kg): "))
    bill = weight_kg * price
    revenue += bill
    customers += 1
    print(f"Customer #{customers} pays Rp {bill}")
    service = int(input("Service: "))

print()
print("--- END OF DAY ---")
print(f"Customers served : {customers}")
print(f"Revenue          : Rp {revenue}")
if customers > 0:
    print(f"Average bill     : Rp {revenue / customers:.2f}")
```

Python's `continue` does the same as C's. Notice that the Python program needs no variable declarations at the top: `price`, `weight_kg` and `bill` come into existence inside the loop, the first time they are assigned.

2.11 Making It Better: Guarding Against Impossible Input

Cashier v0.2 rejects an unknown service, but it happily accepts a weight of 0 or −3 kg and prints a bill of Rp 0 or a negative bill that *reduces* the day's takings. In a real shop that is a problem: one slip of the finger, and the end-of-day total no longer matches the drawer. The fix is a second guard, this time on the weight. In pseudocode, lines 8 and 9 of **Pseudocode 2.8** become **Pseudocode 2.9**:

Pseudocode 2.9 The refined weight input: ask again until the weight is sensible.

```
8a  read weight
8b  while weight < 1 or weight > 50
8c      print "Please type a weight from 1 to 50."
8d      read weight
9   bill ← weight × price
```

The upper limit of 50 kg is a business decision, not a law of nature: no single customer of the shop has ever brought more, and a typed 500 is surely a mistake. The loop reads ahead and tests at the top, the same pattern as the sentinel loop. Chapter 3 introduces a loop that fits this "ask until valid" job even better, `do`…`while`, and the logical operator `||` that joins the two tests; cashier v0.3 in Section 3.9 uses both. For now, notice what the refinement gave us: a new requirement ("reject impossible weights") turned into four lines of pseudocode *before* any code, and the place where they belong is obvious from the structure.

2.12 Using It for Real

The shape of cashier v0.2, a loop that reads a request, checks it, handles it according to its kind and adds it to a running total, is the shape of every point-of-sale program, from a warung's cash register to a web shop. To adapt it to your own business, list your services and their prices, give each a number in the menu, and replace the chain of `if`…`else if` accordingly; the loop, the sentinel and the summary stay the same. A photocopy shop might offer 1 = black and white, 2 = color, 3 = binding; a small rice seller might offer three kinds of rice by the kilogram.

Take a drinking-water refill depot as a worked example. The depot sells refills of a 19-liter gallon at Rp 6,000 for ordinary filtered water and Rp 8,000 for mineral water, and a new gallon with water at Rp 45,000. The menu becomes 1 = refill filtered, 2 = refill mineral, 3 = new gallon, 0 = close. The weight in cashier v0.2 becomes the *number of gallons*, and the bill is the number of gallons times the price. Nothing else changes: the loop still runs until 0, unknown numbers are still rejected with `continue`, and the owner still gets the number of customers, the takings and the average bill at the end of the day. If the depot also delivers, a fourth menu item with a fixed delivery fee, and a counter of deliveries, fit into the same chain. Writing the adapted pseudocode first, as in **Pseudocode 2.8**, takes five minutes and shows at once which lines change.

Three practical points apply to every such adaptation. Keep money in whole rupiah, as we do; when an average needs decimals, compute it only for display, as the summary does. Choose a sentinel that can never be real data: 0 works for a service number here because no service is numbered 0, but it would not work for, say, a discount amount, which may genuinely be 0. And test the empty day: close the till before the first customer and check that the program prints a summary with zero customers and does not divide by zero.

2.13 Common Mistakes, and How to Avoid Them

- **Forgetting to initialize an accumulator.** `int total;` without `= 0` starts with whatever happened to be in memory. GCC with `-Wall` often warns (`'total' may be used uninitialized`). Initialize every counter and total where you declare it.
- **Processing the sentinel.** A loop that reads inside the body and adds before testing counts the 0 as an order, or, with a sentinel such as −1, subtracts it. Read before the loop and again at the end of the body.
- **An infinite loop.** Forgetting `counter = counter + 1` or the second `read` means that the condition never changes. If a program stops responding, press Ctrl+C in the terminal and look for the statement that should change the loop condition.
- **Dividing integers when a fraction is wanted.** `total / count` with two `int`s drops the fraction; `(double) (total / count)` is too late. Write `(double) total / count`. In Python use `/`, not `//`.
- **Dividing by zero.** Guard every division by a count with `if (count != 0)`. In C, integer division by zero crashes the program; in Python it raises `ZeroDivisionError`.
- **A misplaced semicolon.** `while (weight != 0);` or `if (bill >= 100000);` ends the statement at the semicolon: the `while` loops forever on an empty body, and the `if` controls nothing. GCC warns about the `if` with `-Wextra`.
- **The wrong order in an `else if` chain.** Overlapping conditions must be ordered from the most demanding to the least; otherwise the first, weaker test catches everything.
- **Relying on indentation in C.** Without braces, only the *first* statement after `if` or `while` belongs to it, however the next lines are indented. Use braces whenever the body has more than one statement.
- **A sentinel that can be real data.** A sentinel must be impossible as data. A weight of 0 cannot occur, so 0 is safe; a temperature of 0 degrees or a discount of 0 rupiah can occur, and a program that stops at such a value loses data without any error message. When no impossible value exists, ask the user first how many values will follow and use a counter instead.
- **Comparing fractions with `==`.** `if (average == 4.67)` is almost never true, because 14.0 / 3 is 4.666… in the computer and only *printed* as 4.67. Compare whole numbers with `==`, and compare fractions with `<` or `>` or against a small tolerance; Chapter 3 returns to this.
- **Testing only the happy path.** A program tested only with ordinary input has not been tested. For every loop, try the empty case (the sentinel first), the single case (one value), and a typical case; for every chain, try one value in each branch and the values exactly on the boundaries, such as a bill of exactly Rp 50,000 and Rp 100,000.

2.14 Chapter Summary

- An algorithm is a finite list of precise actions in a definite order that solves a problem for every allowed input. Pseudocode describes an algorithm for people, with numbered lines, `←` for assignment and indentation for structure.
- Every algorithm can be built from three control structures, sequence, selection and repetition, each with one entry and one exit, combined by stacking and nesting (**Figure 2.1**); structured programs need no `goto`.
- Selection: `if`, `if`…`else`, and chains written `else if` in C and `elif` in Python, where the first true condition wins. The conditional operator `?:` (Python: `x if c else y`) chooses between two values.

- Counter-controlled repetition runs a known number of times (**Pseudocode 2.3**); sentinel-controlled repetition runs until a special value arrives, reading before the loop and at the end of the body (**Pseudocode 2.6**, **Figure 2.2**).
- Top-down, stepwise refinement turns a one-sentence top into runnable pseudocode and brings design decisions, such as the guard against dividing by zero in Equation (2.2), to light before any code is written.
- A cast such as `(double) total` makes the division keep its fraction by promoting the other operand; casting the finished quotient is too late.
- Compound assignment (`+=`, `-=`, `*=`, `/=`, `%=`) abbreviates updating a variable (**Table 2.2**); `++c` increments before use, `c++` after. Python has compound assignment but no `++`.
- Cashier v0.2 serves customers until the sentinel 0, rejects unknown services with `continue`, and prints the number of customers, the takings and the average bill.

2.15 Exercises

1. Write pseudocode, with two refinements, for this task: "read the weights of the day's orders and report how many weighed more than 5 kg". Then program it in C and Python, rejecting negative weights with a message.
2. Rewrite the discount chain of **Listing 2.1** so that it tests `bill < 50000` first. Which conditions must the other branches use so that the results stay the same?
3. What does this fragment print, and why? `int c = 3; c += c++;` Do not run it; explain instead why a careful programmer never writes it.
4. Change the survey so that an answer other than 1 or 2 is rejected with the message `Please type 1 or 2.` and asked again for the same customer. Which variable must *not* be increased when an answer is rejected?
5. Trace **Pseudocode 2.6** by hand for the input 5, 0 and write down the values of `total`, `count` and `weight` after every line. How many times is the loop body executed?
6. Extend cashier v0.2 so that the summary also prints how many customers chose each service. How many new variables do you need, and where are they increased?

2.16 Solutions to the Exercises

1. Top: *count the heavy orders of the day*. First refinement: initialize; read and classify the weights until the sentinel; print the result. Second refinement: `orders ← 0`, `heavy ← 0`; read the first weight; while it is not 0: if it is negative print a message, else increase `orders` and, if it is above 5, increase `heavy`; read the next weight; finally print `heavy` and `orders`. **Listing 2.13** is the C program; `heavy_count.py` is the Python version.

Listing 2.13 `heavy_count.c`: counting the orders heavier than 5 kg, with negative weights rejected.

```c
/* heavy_count.c - how many orders of the day weigh more than 5 kg? */
#include <stdio.h>

int main(void)
{
    int weight;
    int orders = 0;
    int heavy = 0;

    printf("Weight in kg (0 to finish): ");
    if (scanf("%d", &weight) != 1) return 1;
    while (weight != 0) {
        if (weight < 0) {
            printf("A weight cannot be negative; ignored.\n");
        } else {
            ++orders;
            if (weight > 5) {
                ++heavy;
            }
        }
        printf("Weight in kg (0 to finish): ");
        if (scanf("%d", &weight) != 1) return 1;
    }
    printf("%d of %d orders weigh more than 5 kg\n", heavy, orders);
    return 0;
}
```

```
Weight in kg (0 to finish): 4
Weight in kg (0 to finish): 7
Weight in kg (0 to finish): -2
A weight cannot be negative; ignored.
Weight in kg (0 to finish): 6
Weight in kg (0 to finish): 3
Weight in kg (0 to finish): 0
2 of 4 orders weigh more than 5 kg
```

The negative weight is neither an order nor heavy; of the four real orders, 7 kg and 6 kg are above 5 kg.

2. Test from the bottom up: `if (bill < 50000) percent = 0; else if (bill < 100000) percent = 5; else percent = 10;`. The middle branch is reached only when `bill >= 50000` is already known, so it needs only `bill < 100000`. Both versions give the same result for every bill; the one that reads more naturally is the better one.

3. Nobody can say. `c += c++` both uses and changes `c` without a defined order between the two, which the C standard calls *undefined behavior*: the compiler may produce 6, 7 or anything else, and GCC warns. The fix is to decide what you mean and write it in separate statements, such as `c += c; c++;`.

4. Put the reading into an inner loop: read the answer; while it is neither 1 nor 2, print the message and read again. Only after a valid answer is `satisfied` or `unhappy` increased, and only then is `customer` increased; `customer` must stay the same while an answer is being repeated, or a rejected answer would use up a customer's turn.

5. Lines 1–2: `total` = 0, `count` = 0. Lines 3–4: `weight` = 5. Line 5: 5 ≠ 0, enter the body. Line 6: `total` = 5. Line 7: `count` = 1. Lines 8–9: `weight` = 0. Line 5: 0 ≠ 0 is false, leave the loop. Line 10: `count` = 1 ≠ 0. Line 11: `average` = 5.0. The body ran exactly once.

6. Three counters, `nWash`, `nExpress` and `nIroning`, all initialized to 0 before the loop. Each is increased in its own branch of the chain, next to the line that sets the price. The summary prints

the three counters; their sum must equal `customers`, which is a useful check. Cashier v0.3 in Chapter 3 does exactly this.

References

[1] C. Böhm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Commun. ACM*, vol. 9, no. 5, pp. 366–371, May 1966.

[2] E. W. Dijkstra, "Go To statement considered harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968.

[3] P. J. Deitel and H. M. Deitel, *C How to Program*, 9th ed. Hoboken, NJ, USA: Pearson, 2022.

[4] Python Software Foundation, "More Control Flow Tools," *The Python Tutorial*, Python 3.14. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.python.org/3.14/tutorial/controlflow.html

[5] cppreference.com, "Statements (C): if, while, continue." Accessed: Sep. 25, 2026. [Online]. Available: https://en.cppreference.com/w/c/language/statements

[6] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, pp. 221–227, Apr. 1971.

Chapter 3

Program Control: for, do...while, switch and Logical Operators

Chapter 2 gave you the ideas of selection and repetition and one statement for each kind. This chapter completes the toolbox. You will meet the `for` statement, which gathers everything a counting loop needs into one line; the `do`...`while` statement, which runs its body at least once and is made for "ask until the answer is valid"; the `switch` statement, which chooses among many fixed cases; the statements `break` and `continue`, which leave a loop early or skip one round; and the logical operators `&&`, `||` and `!`, which let one condition say several things at once. Python has its own forms of all of these, `for` over a `range`, `while True` with `break`, and the `match` statement, and each is shown beside its C counterpart. The chapter ends with version 0.3 of the Profitina Laundry cashier, which checks every weight it is given, chooses services with `switch`, and prints a formatted daily report with columns that line up. After this chapter you know every control structure of C, and Chapter 4 puts them all to work on five complete problems.

3.1 A New Washing Machine and a Neater Report

Profitina Laundry has three washing machines, and the oldest one is starting to leak. Pak Hendra wants to replace it next year and plans to put money aside in a savings deposit at the bank. Before he decides how much, he wants to see a simple table: if he deposits one million rupiah today at 5% interest per year, how much will be in the account after one year, two years, and so on up to ten? Nobody wants to compute ten powers by hand. A program can print the table in a moment, with the years in one column and the amounts neatly aligned in another.

At the counter, meanwhile, cashier v0.2 has shown its weak spots. Mbak Rina once typed 60 instead of 6 for the weight of a bag, and the day's takings jumped by Rp 432,000 that nobody had paid. The program accepted the weight without a word. Pak Hendra also wants more from the daily report than three numbers: how many customers chose each service, and what the largest bill of the day was, printed as a table he can photograph and send to the family group chat.

Both wishes need the tools of this chapter. The savings table is a loop that counts from 1 to 10, the natural job of `for`. Checking the weight means asking *until* the answer is sensible, which is what `do`...`while` is for. Choosing among services is a multi-way decision, the job of `switch`. And "the weight must be at least 1 *and* at most 50" is a condition with two parts, joined by a logical operator. The interest rates in this chapter are examples for the arithmetic only; a real deposit's rate and terms come from the bank.

3.2 The `for` Statement

3.2.1 Counting loops in one line

The counter-controlled loop of Chapter 2 needed three separate pieces: the counter was set before the loop, tested at the top, and increased at the bottom of the body. Spread over several

lines, each piece is easy to forget, and forgetting the increase gives an infinite loop. The `for` statement gathers all three into its header:

```
for (initialization; condition; increment)
    statement
```

Figure 3.1 takes one `for` apart and shows the order in which its parts run. (1) The initialization runs once, before anything else. (2) The condition is tested; if it is false, the loop ends at once. (3) If it is true, the body runs. (4) The increment runs, and control returns to (2). So the order is 1, 2, 3, 4, 2, 3, 4, 2, …, until the test in step 2 fails.

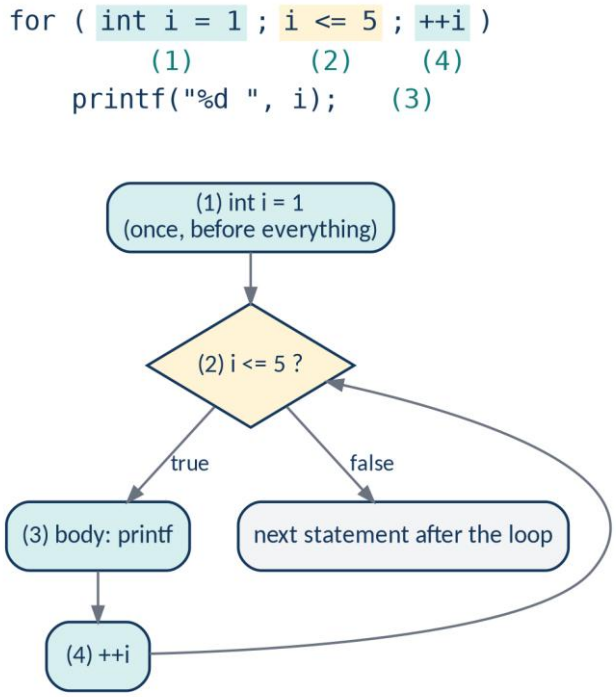


Figure 3.1 The anatomy of a `for` statement: the initialization (1) runs once; then the condition (2), the body (3) and the increment (4) repeat until the condition is false.

A `for` loop is exactly equivalent to this `while` loop, and it helps to see the translation once:

```
initialization;
while (condition) {
    statement
    increment;
}
```

The `for` form is preferred whenever a loop is controlled by a counter, because the reader sees at a glance where the counter starts, where it stops and how it moves. **Listing 3.1** shows four typical counting loops: counting up, counting down, counting in steps of 7, and adding up the even numbers from 2 to 100.

Listing 3.1 `for_basics.c`: counting up, counting down, counting in steps, and a running sum.

```c
/* for_basics.c - the for statement: up, down, in steps, and a sum */
#include <stdio.h>

int main(void)
{
    for (int counter = 1; counter <= 5; ++counter)   /* start; test; step */
        printf("%d ", counter);
    printf("\n");

    for (int i = 5; i >= 1; --i)             /* counting down */
        printf("%d ", i);
    printf("\n");

    for (int i = 7; i <= 77; i += 7)         /* steps of 7 */
        printf("%d ", i);
    printf("\n");

    int sum = 0;
    for (int i = 2; i <= 100; i += 2)        /* even numbers 2..100 */
        sum += i;
    printf("Sum of the even numbers 2..100: %d\n", sum);
    return 0;
}
```

```
1 2 3 4 5
5 4 3 2 1
7 14 21 28 35 42 49 56 63 70 77
Sum of the even numbers 2..100: 2550
```

The counter is declared inside the header, `for (int counter = 1; …)`. Such a variable exists only inside the loop: after the loop it is gone, and another loop may use the same name again, as the three loops that use `i` do. This keeps each counter close to where it is used and prevents an old counter from being used by mistake later; the C standard has allowed it since C99, and our `-std=c17` includes it [1]. The body of each loop is a single statement, so no braces are needed; the same rule as for `if` applies, and with more than one statement the braces are required.

The last loop adds 2 + 4 + 6 + … + 100. There are 50 terms, and pairing the first with the last (2 + 100), the second with the second last (4 + 98), and so on gives 25 pairs of 102, which is 2,550, the same as the program prints. Checking a program against a result worked out another way is a habit worth keeping.

3.2.2 Python's `for` and `range`

Python's `for` is built differently. It does not count by itself; it walks through a sequence of values, one by one, and a function called `range` produces the sequence of numbers to walk through [2]. **Listing 3.2** prints exactly the same lines.

Listing 3.2 `for_basics.py`: the same loops with `for` and `range`.

```python
# for_basics.py - Python's for walks through a range of numbers
for counter in range(1, 6):              # 1, 2, 3, 4, 5 (6 is not included)
    print(counter, end=" ")
print()

for i in range(5, 0, -1):                # counting down
    print(i, end=" ")
print()

for i in range(7, 78, 7):                # steps of 7
    print(i, end=" ")
print()

total = 0
for i in range(2, 101, 2):               # even numbers 2..100
    total += i
print(f"Sum of the even numbers 2..100: {total}")
```

`range(start, stop, step)` produces the numbers from `start` up to, but **not including**, `stop`, in steps of `step`. So `range(1, 6)` produces 1, 2, 3, 4, 5, and `range(5, 0, -1)` counts down from 5 to 1. The rule "stop is not included" confuses beginners at first, but it has a pleasant consequence: `range(0, n)`, which can be shortened to `range(n)`, produces exactly `n` numbers. `print(counter, end=" ")` prints a space instead of a new line at the end, the counterpart of `printf("%d ", counter)`.

> **WARNING.** Do not control a loop with a fractional counter such as `for (double x = 0.1; x != 1.0; x += 0.1)`. The value 0.1 cannot be stored exactly in binary, the sum after nine steps is 0.9999999999999999 and not 1.0, and the loop never ends. Count with whole numbers and compute the fraction inside the body, for example `x = i / 10.0` [3].

3.3 A `for` Showcase: Compound Interest and Field Widths

3.3.1 The formula

Pak Hendra's deposit earns **compound interest**: each year the bank adds 5% of what is in the account, and next year's interest is computed on the larger amount. If p is the first deposit (the principal), r the yearly rate written as a fraction (5% is 0.05), and n the number of years, the amount a after n years is

$$a = p\,(1 + r)^n \tag{3.1}$$

Why a power? After one year the account holds $p + p \cdot r = p(1 + r)$. The second year multiplies *that* amount by $(1 + r)$ again, giving $p(1 + r)^2$, and every further year adds one more factor. For $p = 1{,}000{,}000$ and $r = 0.05$, Equation (3.1) gives 1,050,000 after one year and 1,102,500 after two, since $1.05^2 = 1.1025$.

3.3.2 The program

Listing 3.3 prints the table for ten years. It uses the function `pow` from the mathematics library: `pow(x, y)` computes x^y. The header `math.h` declares it, and on Linux the program must be linked with the mathematics library by adding `-lm` at the end of the compile command: `gcc -std=c17 -Wall -Wextra -O2 savings.c -o savings -lm`. On macOS and with MSYS2 the option is harmless.

Listing 3.3 `savings.c`: a deposit growing at compound interest, printed as an aligned table.

```c
/* savings.c - a deposit growing at compound interest, year by year */
#include <stdio.h>
#include <math.h>

int main(void)
{
    double principal = 1000000.0;   /* first deposit, rupiah */
    double rate = 0.05;             /* 5% interest per year */

    printf("%4s%21s\n", "Year", "Amount on deposit");
    for (int year = 1; year <= 10; ++year) {
        double amount = principal * pow(1.0 + rate, year);
        printf("%4d%21.2f\n", year, amount);   /* widths 4 and 21 */
    }
    return 0;
}
```

```
Year    Amount on deposit
   1           1050000.00
   2           1102500.00
   3           1157625.00
   4           1215506.25
   5           1276281.56
   6           1340095.64
   7           1407100.42
   8           1477455.44
   9           1551328.22
  10           1628894.63
```

3.3.3 Field widths

The table lines up because of **field widths** in the conversion specifications. `%4d` prints an integer right-aligned in a field of 4 characters, padding with spaces on the left; `%21.2f` prints a `double` with 2 decimals, right-aligned in a field of 21 characters. The header line uses the same widths with `%4s` and `%21s` for the two words, so that the headings stand exactly above their columns. A minus sign in front of the width, as in `%-18s`, aligns to the left instead; cashier v0.3 uses it for the labels of its report. **Table 3.1** collects the forms used in this book [4].

Table 3.1 Field widths and precision in `printf`; Python's format specifications inside f-strings use the same numbers, with `>` for right and `<` for left alignment.

Specification	Meaning	`printf` of 42 or 3.14159 gives
`%d`	integer, as wide as needed	`42`
`%5d`	integer, right-aligned in 5 characters	`42`
`%-5d`	integer, left-aligned in 5 characters	`42`
`%.2f`	double with 2 decimals	`3.14`
`%8.2f`	double with 2 decimals, right-aligned in 8 characters	`3.14`
`%-10s`	text, left-aligned in 10 characters	`Express`

Listing 3.4 is the Python version. The power is written `(1.0 + rate) ** year`, and the format specifications inside the f-string carry the same widths: `{year:4d}`, `{amount:21.2f}`, and `{'Year':>4}` for a text right-aligned in 4 characters.

Listing 3.4 `savings.py`: the same table in Python.

```python
# savings.py - a deposit growing at compound interest, year by year
principal = 1000000.0   # first deposit, rupiah
rate = 0.05             # 5% interest per year

print(f"{'Year':>4}{'Amount on deposit':>21}")
for year in range(1, 11):
    amount = principal * (1.0 + rate) ** year
    print(f"{year:4d}{amount:21.2f}")         # widths 4 and 21
```

The two programs agree to the last cent because both compute with the same 64-bit floating-point numbers, and Python's `**` on two floats calls the same C library function as `pow`. Notice, though, that money is computed here in `double`, which Chapter 1 advised against. For a *forecast* printed with two decimals that is acceptable. For a real account, where every rupiah must be exact after thousands of operations, banks compute in whole units and round by fixed rules at fixed moments; we will do the same whenever money is stored, as the cashier does.

3.3.4 Reading the table like an owner

A table is only useful if someone draws a conclusion from it. Suppose the new washing machine costs Rp 1,500,000. The table shows that the deposit passes that amount during the ninth year: after eight years it holds Rp 1,477,455.44, not yet enough, and after nine years Rp 1,551,328.22. So if the bank's rate stays at 5%, a deposit of one million made today pays for the machine in nine years, which is far too slow for a machine that is already leaking. Pak Hendra therefore decides to add money every month instead of waiting, a different computation that you can already write with the loops of this chapter.

The question "after how many years is the target reached?" is itself a small loop, but not a counting loop: we do not know in advance how many years it will take. The natural structure is a `while` loop that multiplies the amount by $1 + r$ and counts the years *while* the amount is still below the target. Choosing between `for` and `while` is exactly this question: do we know the number of repetitions before the loop starts? For the table we do (ten years), so `for` is right; for the target we do not, so `while` is right.

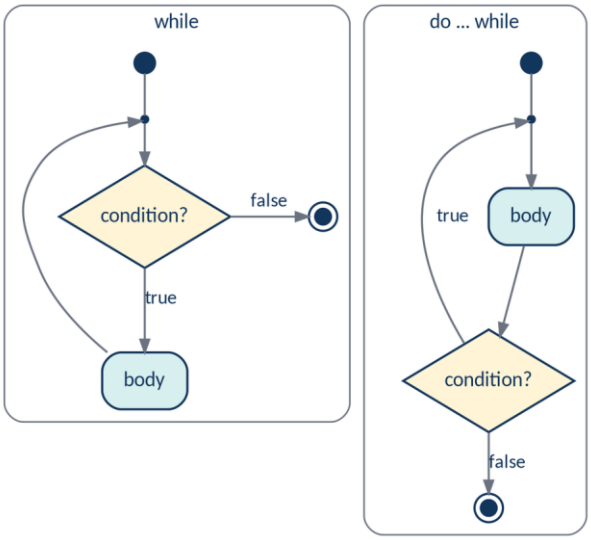
3.4 The `do`...`while` Statement

3.4.1 At least once

The `while` loop tests its condition *before* the body, so its body may run zero times. Some tasks, however, must do something at least once before there is anything to test. Asking for a weight is such a task: we cannot check the weight until we have read one. The `do`...`while` statement tests its condition *after* the body:

```
do {
    statements
} while (condition);
```

The body runs; then the condition is tested; if it is true, the body runs again. **Figure 3.2** compares the two loops. In `while` the decision comes first and the body may be skipped entirely; in `do`...`while` the body comes first and always runs at least once. Note the semicolon after the condition: it is required, because `while (condition);` ends the `do` statement.



the body of do ... while always runs at least once

Figure 3.2 `while` tests before the body, so the body may never run; `do`...`while` tests after the body, so the body always runs at least once.

3.4.2 Ask until the answer is valid

Listing 3.5 asks for a weight until it lies between 1 and 50 kg. The condition `weight < 1 || weight > 50` means "the weight is less than 1 **or** greater than 50"; the operator `||` is explained in Section 3.7.

Listing 3.5 `validate.c`: `do`...`while` keeps asking until the weight is in range.

```c
/* validate.c - do...while: ask again until the weight makes sense */
#include <stdio.h>

int main(void)
{
    int weight;

    do {
        printf("Weight in kg (1-50): ");
        if (scanf("%d", &weight) != 1) return 1;
        if (weight < 1 || weight > 50) {
            printf("Please type a weight from 1 to 50.\n");
        }
    } while (weight < 1 || weight > 50);     /* note the semicolon */

    printf("Accepted: %d kg\n", weight);
    return 0;
}
```

```
Weight in kg (1-50): 0
Please type a weight from 1 to 50.
Weight in kg (1-50): 75
Please type a weight from 1 to 50.
Weight in kg (1-50): 12
Accepted: 12 kg
```

The first answer, 0, is too small; the second, 75, too large; each time the message is printed and the question is asked again. The third answer, 12, makes the condition false, the loop ends, and

the weight is accepted. The body ran three times, and it would have run once, never zero times, even if the first answer had been fine.

Python has no `do`...`while`. The Python idiom is a loop that runs "forever", `while True:`, with a `break` that leaves it as soon as the answer is valid, as in **Listing 3.6**. The test is written where it belongs, after the input, and `break` is the only way out.

Listing 3.6 `validate.py`: Python's form of "ask until valid" is `while True` with `break`.

```
# validate.py - Python has no do...while: loop forever, break when valid
while True:
    weight = int(input("Weight in kg (1-50): "))
    if weight < 1 or weight > 50:
        print("Please type a weight from 1 to 50.")
    else:
        break                           # the only way out of the loop

print(f"Accepted: {weight} kg")
```

Python can also write the double comparison as `1 <= weight <= 50`, which reads like mathematics; C cannot, because in C `1 <= weight <= 50` means `(1 <= weight) <= 50`, compares 0 or 1 with 50, and is always true. The next program shows the Python form.

3.5 The `switch` Statement

3.5.1 One decision, many cases

Profitina Laundry writes a one-letter code on the tag of every bag: W for wash, dry and fold, E for express, I for ironing only. At the end of the day Mbak Rina types the codes of all tags, and the program counts each kind. An `if`...`else if` chain would work, but when one integer or character is compared with many constant values, C offers a clearer statement, `switch` [5]:

```
switch (expression) {
case constant1:
    statements
    break;
case constant2:
    statements
    break;
default:
    statements
    break;
}
```

The expression is evaluated once; control jumps to the `case` whose constant equals its value, or to `default` if none does. The statements after that label run until a `break`, which leaves the whole `switch`. **Listing 3.7** counts the service codes.

Listing 3.7 `service_count.c`: `switch` with shared cases, `break` and `default`.

```c
/* service_count.c - count service codes with switch; Q finishes */
#include <stdio.h>

int main(void)
{
    char code;
    int wash = 0, express = 0, ironing = 0, unknown = 0;

    printf("Service codes (W, E, I; Q to finish):\n");
    while (scanf(" %c", &code) == 1 && code != 'Q' && code != 'q') {
        switch (code) {
        case 'W':
        case 'w':                   /* both cases share one action */
            ++wash;
            break;
        case 'E':
        case 'e':
            ++express;
            break;
        case 'I':
        case 'i':
            ++ironing;
            break;
        default:                    /* anything else */
            ++unknown;
            break;
        }
    }
    printf("Wash, dry & fold: %d\n", wash);
    printf("Express         : %d\n", express);
    printf("Ironing only    : %d\n", ironing);
    printf("Unknown codes   : %d\n", unknown);
    return 0;
}
```

```
Service codes (W, E, I; Q to finish):
W w E
I x W
Q
Wash, dry & fold: 3
Express         : 1
Ironing only    : 1
Unknown codes   : 1
```

The codes may be typed several per line, in capitals or not. The format `" %c"` with a space before `%c` skips any spaces and line breaks and then reads one character, so the program sees W, w, E, I, x, W and finally Q, which ends the loop. The loop condition joins three tests with `&&` (“and”): `scanf` must have read a character, and the character must be neither `Q` nor `q`. Of the codes read, three are wash (W, w, W), one is express, one is ironing, and `x` is unknown.

3.5.2 Falling through

The labels `case 'W':` and `case 'w':` stand directly one above the other, with no statement between them. This works because of a property of `switch` called **fall-through**: once control has jumped to a label, it simply continues downward through the following labels until it meets a `break`. A lowercase `w` jumps to its label and falls into the statements of `W`. Here that is what we want; but a *forgotten* `break` has the same effect, and then one case silently runs the statements of the next. **Figure 3.1** and **Figure 3.2** showed loops as pictures; for `switch`, remember one rule instead: every group of statements ends with `break`, including the last one, so that a case added later cannot fall into it. GCC warns about accidental fall-through with `-Wextra`, and C23 adds the attribute `[[fallthrough]]` to mark an intended one.

A `switch` can test only integer values: `int`, `char` (a small integer) and the enumerations of Chapter 10. It cannot test ranges such as "from 1 to 50", fractional numbers or strings; for those, use `if`...`else if`.

3.5.3 `switch` or an `else if` chain?

The same counting could be written without `switch`:

```
if (code == 'W' || code == 'w')
    ++wash;
else if (code == 'E' || code == 'e')
    ++express;
else if (code == 'I' || code == 'i')
    ++ironing;
else
    ++unknown;
```

Both versions are correct and do the same work. The `switch` has two advantages for this kind of decision. It names the variable being tested only once, at the top, so the reader knows at a glance that the whole statement is about `code`; in the chain, every condition could in principle test something different, and the reader must check each one. And it lists the cases as a column of constants, which reads like a menu. The chain has its own advantage: it can test ranges and combinations, such as `bill >= 100000`, which `switch` cannot. A good rule is to use `switch` when the question is "which of these fixed values is it?" and a chain when the question is "which of these conditions holds?".

3.5.4 Python's `match`

Since version 3.10, Python has a statement that can play the role of `switch`: `match`, with `case` clauses that are tried from top to bottom [6]. **Listing 3.8** is the Python program.

Listing 3.8 `service_count.py`: `match`, where `case "W" | "w"` lists two alternatives and `case _` catches everything else.

```
# service_count.py - count service codes with match; Q finishes
import sys

wash = express = ironing = unknown = 0
finished = False

print("Service codes (W, E, I; Q to finish):")
for line in sys.stdin:                    # one typed line at a time
    for code in line:
        if code.isspace():
            continue                      # skip spaces, like " %c" in C
        if code == "Q" or code == "q":
            finished = True
            break
        match code:
            case "W" | "w":               # both cases share one action
                wash += 1
            case "E" | "e":
                express += 1
            case "I" | "i":
                ironing += 1
            case _:                       # anything else
                unknown += 1
    if finished:
        break

print(f"Wash, dry & fold: {wash}")
print(f"Express         : {express}")
print(f"Ironing only    : {ironing}")
print(f"Unknown codes   : {unknown}")
```

A `case` may list alternatives joined by `|`, which replaces C's stacked labels; `case _` plays the role of `default`. There is no fall-through and no `break`: after one `case` has run, the `match` is finished. (A `break` inside a `match`, as in the next loop, therefore leaves the enclosing *loop*, not the `match`.) `match` can do much more than compare constants, for example take a list apart, but for this book it is simply Python's `switch`. Reading the codes also looks different: the program walks through the input line by line with `for line in sys.stdin`, then through each line character by character with `for code in line`, skipping spaces with `isspace` as the C format `" %c"` does. When it meets a Q it sets the variable `finished` and leaves both loops, because a `break` leaves only the innermost loop.

3.6 `break` and `continue`

The statement `break`, which ends a `switch`, also ends a loop: when it runs inside `while`, `for` or `do`...`while`, the loop stops at once and control continues after it. The statement `continue`, which cashier v0.2 already used, skips only the *rest of the current iteration*: in a `while` or `do`...`while` control jumps to the condition test, and in a `for` it jumps to the increment first, so the counter still advances. **Listing 3.9** shows the difference on the numbers 1 to 10.

Listing 3.9 `break_continue.c`: `break` ends the loop at 5; `continue` skips only 5.

```c
/* break_continue.c - leave a loop early, or skip one round */
#include <stdio.h>

int main(void)
{
    for (int x = 1; x <= 10; ++x) {
        if (x == 5)
            break;              /* leave the loop at once */
        printf("%d ", x);
    }
    printf("\n");

    for (int x = 1; x <= 10; ++x) {
        if (x == 5)
            continue;           /* skip the rest of this round only */
        printf("%d ", x);
    }
    printf("\n");
    return 0;
}
```

```
1 2 3 4
1 2 3 4 6 7 8 9 10
```

The first loop prints 1 to 4 and stops when `x` reaches 5. The second prints every number except 5. The Python version, `break_continue.py` in the code folder, has the same two loops with `range(1, 11)` and prints the same two lines.

Both statements are useful but should be used with care. A loop with several `break`s has several exits, and the property that made structured programs readable, one entry and one exit, is lost. Use `break` for the clear cases, such as leaving a `while True` loop when valid input has arrived or leaving a search as soon as the item is found, and prefer a condition that says when the loop ends wherever that is just as easy.

A `break` leaves only the *innermost* loop or `switch` around it. When a search runs in two nested loops, say over the rows and columns of the price list, and must stop completely as soon as it finds a price above Rp 80,000, a `break` in the inner loop only ends the current row; the outer loop carries on with the next one. The structured solution is a flag, a variable such as `found` that starts at 0, is set to 1 where the item is found, and is tested in the condition of the outer loop: `for (kg = 1; kg <= 10 && !found; ++kg)`. The Python program `service_count.py` uses exactly this technique with its variable `finished`.

3.7 Logical Operators

3.7.1 `&&`, `||` and `!`

Conditions so far compared one thing. Real rules often have several parts: a weight is valid if it is at least 1 *and* at most 50; a customer gets a discount if the bill is large *or* the customer is a member; the loop continues if the code is *not* Q. C builds such conditions with three **logical operators** [1]:

- `&&` (logical AND) is true when both operands are true;
- `||` (logical OR) is true when at least one operand is true;
- `!` (logical NOT) turns true into false and false into true.

Python spells them as words: `and`, `or` and `not`. **Table 3.2** is the **truth table** of the three operators, with 0 for false and 1 for true, as C writes them.

Table 3.2 The truth table of the logical operators AND (`&&`), OR and NOT (`!`); in C a condition is false when it is 0 and true otherwise.

a	b	a AND b	a OR b	NOT a
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

The operators have their own precedence: `!` binds most tightly, then the comparisons, then `&&`, and `||` last. So `weight >= 1 && weight <= 50` needs no parentheses, and `a || b && c` means `a || (b && c)`. When `&&` and `||` are mixed, parentheses make the meaning clear to the reader and should always be written.

3.7.2 Short-circuit evaluation

C evaluates `&&` and `||` from left to right and stops as soon as the answer is known. If the left operand of `&&` is false, the whole expression is false whatever the right operand is, so the right operand is **not evaluated at all**; likewise the right operand of `||` is skipped when the left one is true. This is called **short-circuit evaluation**, and it is not only a speed trick but a tool for writing safe conditions. **Listing 3.10** prints the truth table with two nested `for` loops and then uses short-circuiting to protect a division.

Listing 3.10 `logic.c`: the truth table of `&&`, `||` and `!`, and a division protected by short-circuit evaluation.

```c
/* logic.c - &&, || and !, and short-circuit evaluation */
#include <stdio.h>

int main(void)
{
    printf("a b | a&&b a||b  !a\n");
    for (int a = 0; a <= 1; ++a) {
        for (int b = 0; b <= 1; ++b) {
            printf("%d %d |  %d    %d    %d\n", a, b, a && b, a || b, !a);
        }
    }

    int orders = 0, kilos = 0;
    /* when orders is 0, && stops before the division is tried */
    if (orders != 0 && kilos / orders > 5) {
        printf("Big orders on average.\n");
    } else {
        printf("No verdict: no orders yet.\n");
    }
    return 0;
}
```

```
a b | a&&b a||b  !a
0 0 |  0    0    1
0 1 |  0    1    1
1 0 |  0    1    0
1 1 |  1    1    0
No verdict: no orders yet.
```

In the condition `orders != 0 && kilos / orders > 5`, the division by `orders` is evaluated only when `orders != 0` is true. With no orders, the left operand is false, the division never happens, and the program cannot crash. Written in the other order, `kilos / orders > 5 && orders != 0`,

the same condition divides by zero first. Order the parts of a condition so that each protects the next.

Listing 3.11 is the Python version. Python's `and` and `or` short-circuit in the same way. One detail differs: they do not always return `True` or `False` but return one of their operands (`0 and 1` is 0, `1 or 0` is 1), which is why the program wraps them in `int(...)` to print 0 or 1 exactly as C does.

Listing 3.11 `logic.py`: `and`, `or` and `not`, with the same short-circuit protection.

```
# logic.py - and, or and not, and short-circuit evaluation
print("a b | a&&b a||b  !a")
for a in range(2):
    for b in range(2):
        print(f"{a} {b} |  {int(a and b)}    {int(a or b)}    {int(not a)}")

orders, kilos = 0, 0
# when orders is 0, "and" stops before the division is tried
if orders != 0 and kilos // orders > 5:
    print("Big orders on average.")
else:
    print("No verdict: no orders yet.")
```

3.7.3 De Morgan's laws

The validation loop of Section 3.4 repeats *while the weight is invalid*: `weight < 1 || weight > 50`. The weight is *valid* when `weight >= 1 && weight <= 50`. The two conditions are each other's negation, and the connection between them is one of **De Morgan's laws**:

$$\neg(A \wedge B) \equiv \neg A \vee \neg B \qquad \neg(A \vee B) \equiv \neg A \wedge \neg B \tag{3.2}$$

In words: "not (A and B)" is the same as "not A, or not B", and "not (A or B)" is the same as "not A, and not B". With A standing for `weight >= 1` and B for `weight <= 50`, "not valid" is `!(weight >= 1 && weight <= 50)`, which by Equation (3.2) equals `!(weight >= 1) || !(weight <= 50)`, that is, `weight < 1 || weight > 50`. When you must negate a condition, apply Equation (3.2) on paper instead of guessing; turning `&&` into `||` is where most negation errors come from.

3.7.4 From a business rule to a condition

Most conditions in real programs start life as a sentence spoken by someone who runs the business. Translating such sentences carefully is a skill of its own, and three rules of Profitina Laundry show how it is done.

"Express orders are accepted only until 10 in the morning, except for members." The order is accepted when the service is not express, or the hour is before 10, or the customer is a member: `service != 2 || hour < 10 || member == 1`. Read it back in words to check it: a non-express order is always accepted; an express order is accepted early in the day; and a member's express order is accepted at any time. All three agree with the sentence.

"A bag gets the free fragrance if it weighs at least 5 kg and is not ironing only." That is `weight >= 5 && service != 3`. Here both parts must hold, so the operator is `&&`.

"Close the till when the cashier types 0, or when it is past 21:00." The loop must *continue* while neither is true: `!(service == 0 || hour >= 21)`, which by Equation (3.2) is `service != 0 && hour < 21`. Loop conditions are often easier to write as the negation of the stopping rule, and De Morgan's laws turn that negation into a plain condition.

Two habits make such translations reliable. Write the condition, then read it back aloud in plain words and compare it with the original sentence. And test it on a small table of cases, one row for each combination of the parts, just like **Table 3.2**; with two or three parts that is four or eight rows, a few minutes of work that catches most mistakes.

3.8 Choosing the Right Structure

You now know every control structure of C. Which one to use is usually decided by one question each, as **Table 3.3** summarizes.

Table 3.3 Choosing a control structure: the question to ask and the statement that answers it.

Situation	Structure in C	In Python
a known number of repetitions, or a counter that moves regularly	`for`	`for … in range(…)`
repeat while a condition holds, possibly zero times	`while`	`while`
do something at least once, then decide whether to repeat	`do…while`	`while True` … `break`
one of two paths	`if…else`	`if` … `else`
one of many paths decided by ranges or complex conditions	`if…else if` chain	`if` … `elif`
one of many paths decided by one integer or character value	`switch`	`match`
a small choice between two values inside an expression	`?:`	`x if c else y`

All three loops can do every job: a `while` can count, and a `for` with an empty increment can wait for a sentinel. The point of choosing well is the reader. A `for` says “this counts”; a `do…while` says “this happens at least once”; a `switch` says “this is one decision among fixed cases”. Picking the structure that says what you mean is the cheapest documentation there is.

Four small cases show how the choice is made in practice. *Print the receipt numbers 1 to 40 of a day*: the number of repetitions is known, so `for`. *Read bags from the scale until the cashier types 0*: the number is unknown and there may be no bag at all, so `while` with a sentinel. *Ask for the customer’s phone number until it has between 10 and 13 digits*: at least one question is needed, so `do…while`. *Choose the service from a menu number*: one integer, several fixed values, so `switch`. In each case the question from **Table 3.3** decides, and a reader who knows the conventions understands the purpose of each loop before reading its body.

3.9 Profitina Laundry Cashier v0.3

Cashier v0.3 uses every tool of this chapter. The main loop is a `do…while`, because the cashier is asked for a service at least once. A `switch` decides what to do with the service: 0 closes the till, 1 to 3 are the services, and everything else is unknown. Inside the service case, a second `do…while` asks for the weight until it lies between 1 and 50 kg, which prevents Mbak Rina’s 60-for-6 slip. Three counters record how many customers chose each service, a variable `largest` remembers the largest bill, and the report is printed with field widths. **Listing 3.12** is the program.

Listing 3.12 `cashier_v03.c`: cashier v0.3 with `switch`, validated weights and a formatted daily report.

```c
/* cashier_v03.c - Profitina Laundry cashier v0.3: switch, checks, report */
#include <stdio.h>

int main(void)
{
    int service, weightKg, price, bill;
    int nWash = 0, nExpress = 0, nIroning = 0;
    int revenue = 0, largest = 0;

    printf("=== Profitina Laundry - cashier v0.3 ===\n");
    do {
        printf("Service (1 wash, 2 express, 3 ironing, 0 close): ");
        if (scanf("%d", &service) != 1) return 1;

        switch (service) {
        case 0:
            break;                          /* leaves the switch only */
        case 1: case 2: case 3:
            do {                            /* ask until the weight is sane */
                printf("Weight in kg (1-50): ");
                if (scanf("%d", &weightKg) != 1) return 1;
            } while (weightKg < 1 || weightKg > 50);

            if (service == 1) {
                price = 8000;
                ++nWash;
            } else if (service == 2) {
                price = 12000;
                ++nExpress;
            } else {
                price = 6000;
                ++nIroning;
            }
            bill = weightKg * price;
            revenue += bill;
            if (bill > largest)
                largest = bill;
            printf("Bill: Rp %d\n", bill);
            break;
        default:
            printf("Unknown service %d.\n", service);
            break;
        }
    } while (service != 0);
```

Listing 3.12 (Continued) *cashier_v03.c: cashier v0.3 with switch, validated weights and a formatted daily report.*

```c
    int customers = nWash + nExpress + nIroning;
    printf("\n--- DAILY REPORT ---\n");
    printf("%-18s%6d\n", "Wash, dry & fold", nWash);
    printf("%-18s%6d\n", "Express", nExpress);
    printf("%-18s%6d\n", "Ironing only", nIroning);
    printf("%-18s%6d\n", "Customers", customers);
    printf("%-18s%12d\n", "Revenue (Rp)", revenue);
    printf("%-18s%12d\n", "Largest bill (Rp)", largest);
    if (customers > 0)
        printf("%-18s%12.2f\n", "Average (Rp)", (double) revenue / customers);
    return 0;
}
```

Three details deserve attention. First, `case 0: break;` looks strange: the `break` leaves only the `switch`, not the `do`...`while`, and the loop then ends because its condition `service != 0` is false. Second, `case 1: case 2: case 3:` uses fall-through to send three services to the same

statements, where an `if`...`else if` chain then picks the price and increases the right counter. Third, finding the largest bill is a small algorithm of its own: `largest` starts at 0, smaller than any real bill, and every new bill that is larger replaces it, so after the loop it holds the maximum. Here is a day with a mistyped weight and an unknown service:

```
=== Profitina Laundry - cashier v0.3 ===
Service (1 wash, 2 express, 3 ironing, 0 close): 1
Weight in kg (1-50): 5
Bill: Rp 40000
Service (1 wash, 2 express, 3 ironing, 0 close): 2
Weight in kg (1-50): 0
Weight in kg (1-50): 60
Weight in kg (1-50): 3
Bill: Rp 36000
Service (1 wash, 2 express, 3 ironing, 0 close): 4
Unknown service 4.
Service (1 wash, 2 express, 3 ironing, 0 close): 3
Weight in kg (1-50): 2
Bill: Rp 12000
Service (1 wash, 2 express, 3 ironing, 0 close): 1
Weight in kg (1-50): 7
Bill: Rp 56000
Service (1 wash, 2 express, 3 ironing, 0 close): 0

--- DAILY REPORT ---
Wash, dry & fold       2
Express                1
Ironing only           1
Customers              4
Revenue (Rp)          144000
Largest bill (Rp)      56000
Average (Rp)        36000.00
```

The express customer's weight was typed as 0 and then 60, and each time the program simply asked again until 3 was typed; the unknown service 4 was reported and not counted. Check the report by hand: two wash orders (5 kg and 7 kg, Rp 40,000 and Rp 56,000), one express order of 3 kg (Rp 36,000) and one ironing order of 2 kg (Rp 12,000) give Rp 144,000 from four customers, an average of Rp 36,000.00, and the largest bill is Rp 56,000.

Listing 3.13 is the Python version. The main loop is `while True` with a `break` at its end when the service is 0, the `switch` becomes a `match` with `case 1 | 2 | 3`, and the weight loop is the `while True`...`break` idiom of Section 3.4 with the Python comparison `1 <= weight_kg <= 50`.

Listing 3.13 `cashier_v03.py`: cashier v0.3 in Python, with `match` and the `while True` idiom.

```python
# cashier_v03.py - Profitina Laundry cashier v0.3: match, checks, report
n_wash = n_express = n_ironing = 0
revenue = largest = 0

print("=== Profitina Laundry - cashier v0.3 ===")
while True:
    service = int(input("Service (1 wash, 2 express, 3 ironing, 0 close): "))
    match service:
        case 0:
            pass
        case 1 | 2 | 3:
            while True:                       # ask until the weight is sane
                weight_kg = int(input("Weight in kg (1-50): "))
                if 1 <= weight_kg <= 50:
                    break
            if service == 1:
                price = 8000
                n_wash += 1
            elif service == 2:
                price = 12000
                n_express += 1
            else:
                price = 6000
                n_ironing += 1
            bill = weight_kg * price
            revenue += bill
            if bill > largest:
                largest = bill
            print(f"Bill: Rp {bill}")
        case _:
            print(f"Unknown service {service}.")
    if service == 0:
        break

customers = n_wash + n_express + n_ironing
print()
print("--- DAILY REPORT ---")
print(f"{'Wash, dry & fold':<18}{n_wash:6d}")
print(f"{'Express':<18}{n_express:6d}")
print(f"{'Ironing only':<18}{n_ironing:6d}")
print(f"{'Customers':<18}{customers:6d}")
print(f"{'Revenue (Rp)':<18}{revenue:12d}")
print(f"{'Largest bill (Rp)':<18}{largest:12d}")
if customers > 0:
    print(f"{'Average (Rp)':<18}{revenue / customers:12.2f}")
```

The report's format specifications match C's: `{'Revenue (Rp)':<18}` is `%-18s`, a text left-aligned in 18 characters, and `{revenue:12d}` is `%12d`. The two programs print the same report, space for space.

3.10 Making It Better: A Price List for the Wall

Customers keep asking Mbak Rina what their bag will cost before it is weighed. A printed price list on the wall, showing the price of 1 to 10 kilograms for every service, would answer the question before it is asked. This is a table with ten rows and three columns of prices, and a table is a loop inside a loop: the outer loop runs over the rows (kilograms), and for each row the inner loop runs over the columns (services). **Listing 3.14** prints it.

Listing 3.14 `price_table.c`: nested `for` loops print a price list with aligned columns.

```c
/* price_table.c - a price table for 1..10 kg with nested for loops */
#include <stdio.h>

int main(void)
{
    printf("%3s%10s%10s%10s\n", "kg", "Wash", "Express", "Ironing");
    for (int kg = 1; kg <= 10; ++kg) {
        printf("%3d", kg);
        for (int service = 1; service <= 3; ++service) {
            int price = service == 1 ? 8000 : service == 2 ? 12000 : 6000;
            printf("%10d", kg * price);
        }
        printf("\n");
    }
    return 0;
}
```

```
 kg      Wash   Express   Ironing
  1      8000     12000      6000
  2     16000     24000     12000
  3     24000     36000     18000
  4     32000     48000     24000
  5     40000     60000     30000
  6     48000     72000     36000
  7     56000     84000     42000
  8     64000     96000     48000
  9     72000    108000     54000
 10     80000    120000     60000
```

For each value of `kg`, the inner loop runs three times, once per service, and prints one price in a field of 10 characters; after the inner loop, `printf("\n")` ends the row. In total the inner body runs 10 × 3 = 30 times. The price of the service is chosen with a *nested* conditional operator, `service == 1 ? 8000 : service == 2 ? 12000 : 6000`, which is read from left to right like an `else if` chain; for three values that is still readable, for more a `switch` would be clearer. The Python version, `price_table.py`, uses `print(..., end="")` to stay on the same line and the conditional expression `8000 if service == 1 else 12000 if service == 2 else 6000`; it prints the same table.

A printed table is also a simple test of the cashier. Every entry must equal the bill that cashier v0.3 prints for the same weight and service; if the table says Rp 36,000 for 3 kg of express and the cashier says something else, one of them is wrong. Two programs that compute the same thing in different ways are a powerful check on each other.

3.11 Using It for Real

The tools of this chapter appear in almost every program that talks to a person. A `do`...`while` that asks until the answer is valid belongs wherever people type numbers: an age between 0 and 120, a quantity between 1 and the stock on the shelf, a month between 1 and 12. Choose the limits from the business, not from the computer: 50 kg is right for this laundry and wrong for a rice wholesaler. A `switch` over a menu number or a code letter fits every small point-of-sale program, and the report with field widths works for any shop that wants its day on one screen.

Two limitations of cashier v0.3 remain, and they are worth knowing before it is used at a real counter. If the cashier types a letter where a number is expected, `scanf` fails and the program ends, as the `return 1` says; Chapter 9 reads whole lines of text and handles such input gracefully. And when the program ends, everything is forgotten: the report is on the screen but nowhere else. Chapter 11 writes each day's orders to a file, so that nothing is lost when the power goes off.

The same pattern adapts easily to other small businesses. A school canteen could let the cashier type one letter per item sold, N for nasi goreng, M for mie, T for es teh, and count them with a `switch` exactly as `service_count` counts service codes; at the end of the break the canteen knows what sold and what to cook more of tomorrow. A tailor could validate the measurements typed for a shirt with `do`...`while`, rejecting a chest width of 5 cm or 500 cm before it reaches the cutting table. A small clinic could print a table of appointment times with nested `for` loops, one row per doctor and one column per half hour. In each case the structures stay the same and only the words, the limits and the prices change.

Until then, the report can at least be saved by redirecting the program's output to a file, as in `./cashier_v03 > report.txt`, a trick that works for every program in this book.

3.12 Common Mistakes, and How to Avoid Them

- **A semicolon after the `for` header.** `for (int i = 1; i <= 5; ++i);` has an empty body; the statement below it runs once, after the loop. GCC warns about the misleading indentation with `-Wall`.
- **Off-by-one limits.** `for (i = 1; i < 10; ++i)` runs 9 times, not 10. Decide whether the last value is included, write `<=` or `<` accordingly, and check the first and last iteration by hand. In Python remember that `range(1, 10)` stops at 9.
- **A missing `break` in a `switch`.** Without it, the next case's statements run too. End every case with `break`, and compile with `-Wextra` to catch accidental fall-through.
- **Forgetting the semicolon after `do`...`while`.** `} while (weight < 1 || weight > 50)` without `;` is a syntax error.
- **Writing `1 <= x <= 50` in C.** It compiles, compares 0 or 1 with 50 and is always true. Write `x >= 1 && x <= 50`. (Python's `1 <= x <= 50` is correct.)
- **`&` or `|` instead of `&&` or `||`.** The single-character operators work on bits and do not short-circuit; the condition may still look right on simple tests and fail later.
- **Negating a condition without De Morgan.** The opposite of `x >= 1 && x <= 50` is not `x < 1 && x > 50`, which is never true, but `x < 1 || x > 50`. Apply Equation (3.2).
- **A fractional loop counter.** Count with integers; compute fractions from the counter inside the body.
- **Expecting `break` to leave an `if`.** `break` belongs to loops and `switch` only; inside an `if` that is inside a loop, it leaves the loop. To skip the rest of an `if`, use `else`.
- **Changing the counter inside the body of a `for`.** A statement such as `i = i + 2;` inside `for (int i = 0; i < 10; ++i)` makes the loop skip values in a way that the header no longer shows. If the counter must move differently, say so in the header.
- **Testing a fraction or a string in a `switch`.** `switch` accepts only integer values; `case 2.5:` or `case "W":` does not compile. Use `if`...`else if` for those.
- **Forgetting `-lm`.** On Linux, a program that calls `pow` compiles but fails to link with `undefined reference to 'pow'` unless `-lm` is added at the end of the command.

3.13 Chapter Summary

- `for (initialization; condition; increment)` gathers a counting loop into one header; the initialization runs once, then condition, body and increment repeat (**Figure 3.1**). A counter

declared in the header exists only inside the loop. Python's `for` walks through `range(start, stop, step)`, which excludes `stop`.

- Compound interest follows Equation (3.1); `pow` from `math.h` computes the power, linked with `-lm` on Linux. Field widths such as `%4d`, `%21.2f` and `%-18s` align tables (**Table 3.1**); Python uses the same numbers inside f-strings.
- `do`...`while` tests after the body, so the body runs at least once (**Figure 3.2**); it is the natural loop for "ask until valid". Python writes it as `while True` with `break`.
- `switch` jumps to the `case` equal to an integer or character value, runs until `break`, and falls through otherwise; `default` catches the rest. Python 3.10 and later has `match`, with `|` for alternatives and `case _` for the rest, and no fall-through.
- `break` leaves the innermost loop or `switch`; `continue` skips the rest of the current iteration.
- `&&`, `||` and `!` (Python: `and`, `or`, `not`) combine conditions (**Table 3.2**); they short-circuit, which lets the left part of a condition protect the right part. De Morgan's laws (Equation (3.2)) negate combined conditions correctly.
- The structure that says what you mean is the right one (**Table 3.3**). Cashier v0.3 combines them all: a `do`...`while` main loop, a `switch` on the service, a validated weight, per-service counters, the largest bill, and a report with aligned columns.

3.14 Exercises

1. Write a program that prints the odd numbers from 1 to 19 on one line, separated by spaces, first with `for` in C and then with `range` in Python. Then print them from 19 down to 1.
2. Rewrite the loop of **Listing 3.5** with `while` instead of `do`...`while`. Which statement must now appear twice, and why?
3. Profitina Laundry is open from 07:00 to 21:00 on weekdays (1 = Monday to 5 = Friday), from 08:00 to 18:00 on Saturday (6), and closed on Sunday (7). Write a program that reads a day number and prints the opening hours, using `switch` with fall-through for the five weekdays. Reject numbers outside 1 to 7.
4. Using De Morgan's laws, write the negation of each condition without the `!` operator: (a) `service >= 1 && service <= 3`; (b) `code == 'Q' || code == 'q'`; (c) `bill >= 100000 || member == 1`.
5. Extend `savings` so that it also prints the interest earned in each year (the amount of this year minus the amount of last year) in a third column of width 15. What is the interest in year 10?
6. What does this loop print, and how often does its body run? `for (int i = 0; i < 10; i += 3) printf("%d ", i);` Write the same loop in Python.

3.15 Solutions to the Exercises

1. C: `for (int i = 1; i <= 19; i += 2) printf("%d ", i);` followed by `printf("\n");`, and for the reverse `for (int i = 19; i >= 1; i -= 2)`. Python: `for i in range(1, 20, 2): print(i, end=" ")` and `for i in range(19, 0, -2)`; the stops are 20 and 0 because the stop value is not included.

2. With `while`, the question must be asked once *before* the loop and again inside it, because `while` tests before the body and there is no weight to test until one has been read:

```
printf("Weight in kg (1-50): ");
if (scanf("%d", &weight) != 1) return 1;
while (weight < 1 || weight > 50) {
    printf("Please type a weight from 1 to 50.\n");
    printf("Weight in kg (1-50): ");
    if (scanf("%d", &weight) != 1) return 1;
}
```

This is the read-ahead pattern of the sentinel loop in Chapter 2. It is correct, but the prompt and the `scanf` appear twice; `do`...`while` needs them only once, which is why it is the better fit.

3. After reading `day` and rejecting values outside 1 to 7 with an `if`, the `switch` stacks the weekday labels:

```
switch (day) {
case 1: case 2: case 3: case 4: case 5:
    printf("Open 07:00-21:00\n");
    break;
case 6:
    printf("Open 08:00-18:00\n");
    break;
default:
    printf("Closed\n");
    break;
}
```

In Python, `match day:` with `case 1 | 2 | 3 | 4 | 5:`, `case 6:` and `case _:`.

4. (a) `service < 1 || service > 3`; (b) `code != 'Q' && code != 'q'`, which is exactly the loop condition of **Listing 3.7**; (c) `bill < 100000 && member != 1`. In each case `&&` and `||` swap and every comparison is reversed, as Equation (3.2) says.

5. Keep the previous amount in a variable `previous`, starting at the principal. In the loop compute `amount`, print `amount - previous` with `%15.2f`, and then set `previous = amount`. The interest in year 10 is 1,628,894.63 − 1,551,328.22 = 77,566.41: 5% of the amount at the end of year 9.

6. It prints `0 3 6 9` and the body runs four times: after 9 the counter becomes 12, and `12 < 10` is false. In Python: `for i in range(0, 10, 3): print(i, end=" ")`.

References

[1] *Information Technology — Programming Languages — C*, ISO/IEC 9899:2018, International Organization for Standardization, Geneva, Switzerland, 2018.

[2] Python Software Foundation, “More Control Flow Tools: for statements and the range() function,” *The Python Tutorial*, Python 3.14. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.python.org/3.14/tutorial/controlflow.html

[3] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[4] cppreference.com, “printf, fprintf, sprintf, snprintf (C).” Accessed: Sep. 25, 2026. [Online]. Available: https://en.cppreference.com/w/c/io/fprintf

[5] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988.

[6] B. Bucher and G. van Rossum, “PEP 634 – Structural Pattern Matching: Specification,” Python Enhancement Proposals, 2020. [Online]. Available: https://peps.python.org/pep-0634/

Chapter 4

Evaluation and Discussion I: Five Problems Solved Step by Step

This chapter adds no new statement to your toolbox. Instead, it takes everything Chapters 1 to 3 have given you (variables and arithmetic, input and output, `if` and `switch`, the three loops and the logical operators) and puts it to work on five complete problems of increasing size. The first computes the areas of four plane figures and is pure sequence. The second is a guessing game for two players, a sentinel loop in disguise. The third prints 100 to the power n for any n, and is really a lesson about the limits of numbers in a computer. The fourth is rock, paper, scissors, with input validation and a compact rule for the winner. The fifth, a turn-based battle between a boy and three goblins, combines state variables, a loop, careful ordering and every guard you have learned. Each problem is solved from nothing, always in the same order: the problem stated and restated in plain words; examples worked by hand; a design on paper; a complete C program with its real output and a walk through its important lines; the same program in Python with identical output; a *tempting wrong solution* together with the exact way it fails; and a *check yourself* variant with its answer. By the end you will own a method for attacking any small programming problem, and five fully worked examples of it.

4.1 Where We Left Off

Before we solve anything, let us lay out the toolkit. Naming the tool a problem needs is already half of the work, and each of the five problems below reaches for a different handful.

From **Chapter 1** you have the workbench and the basic vocabulary. You can compile a C program with `gcc -std=c17 -Wall -Wextra -O2` and run a Python program with `python3`. You know that a variable has a name, a type, a value and an address; that `scanf` needs the address and checks what it read; that `int` has a limit of 2,147,483,647 while a Python `int` has none; that integer division drops the fraction and `%` gives the remainder; and that `if` runs a statement only when its condition is true.

From **Chapter 2** you have the method and the first loops. You design with pseudocode and top-down, stepwise refinement; you know sequence, selection and repetition; you can write counter-controlled and sentinel-controlled `while` loops, reading ahead before a sentinel loop; you can protect a division with a check; and you can use `+=`, `++` and the cast `(double)`.

From **Chapter 3** you have the rest of the control structures: `for` for counting, `do`…`while` for “ask until valid”, `switch` (and Python’s `match`) for fixed cases, `break` and `continue`, and the logical operators `&&`, `||` and `!` with their short-circuit evaluation and De Morgan’s laws.

That is enough for all five problems. Nothing below uses functions, arrays or strings beyond printing text; those arrive in Chapters 5, 6 and 9, and several of these programs become shorter then, which is a good reason to solve them now, the long way, and understand every line.

4.2 How to Approach a Programming Problem

4.2.1 Seven steps

Every problem in this chapter is solved with the same seven steps, a programmer's version of the classic advice of Pólya for solving mathematical problems: understand the problem, make a plan, carry it out, and look back [1]. They are not a ritual; each one exists because skipping it is the usual cause of a particular kind of failure.

1. **Restate the problem in your own words.** Write down what comes in, what must go out, and every decision the statement leaves open. Short statements are the most dangerous, because every decision they do not make, you must make yourself, and write down.
2. **Work examples by hand.** Choose small inputs, compute the correct output on paper, and include at least one unusual input: zero, a negative number, the first possible answer, the largest one.
3. **Design on paper.** Choose the control structures and write pseudocode. For a loop, decide what is read before it, what is tested, and what changes in the body.
4. **Trace the design** on one of the hand examples, line by line, with a small trace table like the one in Chapter 2.
5. **Write the program** from the pseudocode, one line at a time, and compile with warnings on. Fix every warning.
6. **Test** with the hand examples of step 2 and compare the output character for character.
7. **Attack your own program.** Look for the input that breaks it: the boundary, the invalid value, the case that makes a loop run zero times or forever. Every problem below ends with a tempting wrong solution precisely to train this step.

> **KEY INSIGHT.** Most of the time spent on a problem should go into steps 1 to 4, before a line of code exists. A design that is right on paper becomes a program in minutes; a program typed without a design often takes hours of guessing to repair.

4.2.2 How each problem is laid out, and where the code lives

Each problem section has the same parts: **the problem**, in a box, followed by our restatement; **understanding it by hand**, with the examples of step 2; **designing it on paper**, with pseudocode; **the C solution**, a complete program with its real output; **the Python solution**, which prints exactly the same output; **a tempting wrong solution**, a program that looks right, its real output, and why it fails; and a **check yourself** variant with its answer.

All programs are in the folder `Chapter04` of the book's code: `c` and `python` hold the solutions, `c/tempting` and `python/tempting` the tempting wrong solutions, and `tests` the inputs and expected outputs used in this chapter. Every C program compiles with `gcc -std=c17 -Wall -Wextra -O2` without a single warning and was also run under the AddressSanitizer and UndefinedBehaviorSanitizer (`-fsanitize=address,undefined`), which stop a program that reads memory it does not own or overflows an `int` [2]. Programs that read input can be run by typing the values, or with the input file redirected, for example `./guess < ../tests/guess.in` on Linux and macOS or `Get-Content ..\tests\guess.in | .\guess.exe` in Windows PowerShell.

4.3 Problem 4.1: Areas of Plane Figures

4.3.1 The problem

> **PROBLEM 4.1.** Write a program that reads the values base, base1, base2, height, side_length and radius (all of them real numbers) and prints the areas of a triangle, a trapezoid, a square and a circle.

The statement is short, so we make its open decisions explicit. *Input:* six real numbers, asked for one at a time with a prompt, in the order given; the triangle uses `base` and `height`, the trapezoid uses `base1`, `base2` and the same `height`, the square uses `side_length`, and the circle uses `radius`. *Output:* the four areas, each on its own labeled line, with two decimals. *Invalid input:* a length cannot be negative; if any of the six values is negative, the program says so and prints no areas. A length of 0 is allowed and gives an area of 0.

4.3.2 Understanding it by hand

The four formulas come from school geometry. With base b and height h, a triangle has area $\frac{1}{2}bh$; with parallel sides b_1 and b_2 and height h, a trapezoid has area $\frac{1}{2}(b_1 + b_2)h$; a square with side s has area s^2; and a circle with radius r has area πr^2:

$$A_{\triangle} = \frac{1}{2}\, b\, h, \qquad A_{\text{trap}} = \frac{b_1 + b_2}{2}\, h, \qquad A_{\square} = s^2, \qquad A_{\circ} = \pi r^2 \tag{4.1}$$

For the example $b = 10$, $b_1 = 8$, $b_2 = 4$, $h = 5$, $s = 3$ and $r = 2$, Equation (4.1) gives a triangle of $\frac{1}{2} \cdot 10 \cdot 5 = 25$, a trapezoid of $\frac{12}{2} \cdot 5 = 30$, a square of 9 and a circle of $\pi \cdot 4 \approx 12.566$, printed as 12.57. A second example checks the circle alone: radius 1 gives $\pi \approx 3.14$. If a program prints 3.00 for that circle, it has divided or multiplied whole numbers somewhere.

4.3.3 Designing it on paper

This is a pure sequence with one guard: read, check, compute, print. **Pseudocode 4.1** is the design.

Pseudocode 4.1 The design of Problem 4.1: a sequence of reads, one guard, and four formulas.

```
GEOMETRY
1  read base, base1, base2, height, side, radius   (one prompt each)
2  if any of the six values is negative
3      print "Lengths cannot be negative." and stop
4  print 0.5 × base × height
5  print (base1 + base2) / 2 × height
6  print side × side
7  print PI × radius × radius
```

C has no built-in constant for π in standard C17, so we define it ourselves as a `const double` with 15 significant digits, more than a `double` can use in any case.

4.3.4 The C solution

Listing 4.1 follows the design line by line.

Listing 4.1 `geometry.c`: the areas of four plane figures, with negative lengths rejected.

```c
/* geometry.c - Problem 4.1: areas of four plane figures */
#include <stdio.h>

int main(void)
{
    const double PI = 3.14159265358979;
    double base, base1, base2, height, side, radius;

    printf("Triangle base: ");
    if (scanf("%lf", &base) != 1) return 1;
    printf("Trapezoid base1: ");
    if (scanf("%lf", &base1) != 1) return 1;
    printf("Trapezoid base2: ");
    if (scanf("%lf", &base2) != 1) return 1;
    printf("Height (triangle and trapezoid): ");
    if (scanf("%lf", &height) != 1) return 1;
    printf("Square side length: ");
    if (scanf("%lf", &side) != 1) return 1;
    printf("Circle radius: ");
    if (scanf("%lf", &radius) != 1) return 1;

    if (base < 0 || base1 < 0 || base2 < 0 || height < 0 || side < 0
        || radius < 0) {
        printf("Lengths cannot be negative.\n");
        return 1;
    }

    printf("\nTriangle area  = %10.2f\n", 0.5 * base * height);
    printf("Trapezoid area = %10.2f\n", (base1 + base2) / 2.0 * height);
    printf("Square area    = %10.2f\n", side * side);
    printf("Circle area    = %10.2f\n", PI * radius * radius);
    return 0;
}
```

```
Triangle base: 10
Trapezoid base1: 8
Trapezoid base2: 4
Height (triangle and trapezoid): 5
Square side length: 3
Circle radius: 2

Triangle area  =      25.00
Trapezoid area =      30.00
Square area    =       9.00
Circle area    =      12.57
```

The output matches the hand computation. Three details are worth a look. All six values are `double` and read with `%lf`, so a length such as 2.5 is accepted. The guard joins six comparisons with `||`: the whole condition is true as soon as one length is negative, and short-circuit evaluation stops at the first such one. Its line is split in two, which C allows anywhere a space is allowed, because the full condition would be longer than a line. Finally, the formulas are written with `0.5` and `2.0`, so that every operation involves a `double` and no integer division can creep in. A run with a negative length shows the guard:

```
Triangle base: -1
Trapezoid base1: 8
Trapezoid base2: 4
Height (triangle and trapezoid): 5
Square side length: 3
Circle radius: 2
Lengths cannot be negative.
```

4.3.5 The Python solution

Listing 4.2 is the same program in Python; `float(input(...))` reads a real number, and `raise SystemExit(1)` ends the program with status 1, the counterpart of `return 1` in C's `main`.

Listing 4.2 `geometry.py`: the same areas in Python.

```
# geometry.py - Problem 4.1: areas of four plane figures
PI = 3.14159265358979       # the same constant as the C program

base = float(input("Triangle base: "))
base1 = float(input("Trapezoid base1: "))
base2 = float(input("Trapezoid base2: "))
height = float(input("Height (triangle and trapezoid): "))
side = float(input("Square side length: "))
radius = float(input("Circle radius: "))

if (base < 0 or base1 < 0 or base2 < 0 or height < 0 or side < 0
        or radius < 0):
    print("Lengths cannot be negative.")
    raise SystemExit(1)

print()
print(f"Triangle area  = {0.5 * base * height:10.2f}")
print(f"Trapezoid area = {(base1 + base2) / 2.0 * height:10.2f}")
print(f"Square area    = {side * side:10.2f}")
print(f"Circle area    = {PI * radius * radius:10.2f}")
```

A long condition in Python must stay inside parentheses to continue on the next line, which is why the guard is wrapped in `( … )`. Python's standard library does provide π as `math.pi`, but using the same constant as the C program guarantees identical output.

4.3.6 A tempting wrong solution

The most tempting shortcut is to copy the school formula literally: "one half times base times height", `1 / 2 * base * height`. **Listing 4.3** does exactly that, for the triangle only.

Listing 4.3 `geometry_wrong.c`: `1 / 2` is evaluated first, as integer division.

```
/* geometry_wrong.c - a tempting wrong triangle: 1 / 2 is 0 */
#include <stdio.h>

int main(void)
{
    double base, height;

    printf("Triangle base: ");
    if (scanf("%lf", &base) != 1) return 1;
    printf("Height: ");
    if (scanf("%lf", &height) != 1) return 1;

    printf("Triangle area = %.2f\n", 1 / 2 * base * height);
    return 0;
}
```

```
Triangle base: 10
Height: 5
Triangle area = 0.00
```

The program compiles without a warning and prints 0.00 for every triangle. The expression is evaluated from left to right: `1 / 2` divides two `int` constants, gives 0, and everything multiplied by 0 is 0. That `base` and `height` are `double` does not help, because the damage is done before they are involved. The Python version in `python/tempting` makes the same mistake with `1 // 2` and

prints the same line; with Python's ordinary `/` the slip could not happen, since `1 / 2` is 0.5 there. The cure in C is to write `0.5`, or `1.0 / 2`, or to put the doubles first: `base * height / 2`.

> **CHECK YOURSELF.** Add a rectangle with `length` and `width`. Which lines of **Listing 4.1** change, and what does the new line print for length 6 and width 4?
>
> *Answer.* Three places change: a new declaration and two new prompts with `scanf` calls; two more comparisons in the negative-length guard; and one new line `printf("Rectangle area = %10.2f\n", length * width);`. For 6 and 4 it prints `Rectangle area =      24.00`. Nothing else changes, which is the benefit of a pure sequence: new figures are simply appended.

4.4 Problem 4.2: The Guessing Game

4.4.1 The problem

> **PROBLEM 4.2.** Write a guessing game. The program reads the answer, then keeps reading guesses until a guess equals the answer. After each wrong guess it prints "Your guess is too low!" or "Your guess is too high!". When the guess is right, it congratulates the player.

In our words: two people play. Player 1 types a secret whole number; player 2 then guesses until the guess is right, and after each wrong guess the program says in which direction the secret lies. We make three decisions. The messages are printed exactly as given, including the exclamation marks, because a program that other programs or people test is judged on its exact output. The congratulation also reports the secret and the number of tries, with "try" for one and "tries" for more. And the secret is visible on the screen when player 1 types it; a real game would let the computer choose a random number, which needs the functions of Chapter 5, so for now player 1 should hide the screen or scroll it away.

4.4.2 Understanding it by hand

Take the secret 37. Player 2 guesses 50: too high. Then 25: too low. Then 37: right, after 3 tries. A clever player halves the remaining range each time, 1 to 100 needs at most 7 guesses that way, but the program does not care how the player guesses. The unusual case is a correct *first* guess: the program must print only the congratulation, with "1 try", and never a "too low" or "too high".

4.4.3 Designing it on paper

The structure is the sentinel loop of Chapter 2 in disguise. The "sentinel" is not a fixed value like 0 but the answer itself: read a guess before the loop, loop *while* the guess differs from the answer, give feedback and read again inside. Inside the loop we already know that the guess is wrong, so a simple `if`…`else` decides between low and high. **Pseudocode 4.2** shows it.

Pseudocode 4.2 The design of Problem 4.2: a read-ahead loop whose sentinel is the answer itself.

```
GUESS
1  read answer
2  tries ← 1
3  read guess
4  while guess ≠ answer
5      if guess < answer
6          print "Your guess is too low!"
7      else
8          print "Your guess is too high!"
9      tries ← tries + 1
10     read guess
11 print congratulations with answer and tries
```

4.4.4 The C solution

Listing 4.4 `guess.c`: the guessing game as a read-ahead `while` loop.

```c
/* guess.c - Problem 4.2: the guessing game for two players */
#include <stdio.h>

int main(void)
{
    int answer, guess;
    int tries = 1;

    printf("Player 1, type the secret number: ");
    if (scanf("%d", &answer) != 1) return 1;

    printf("Player 2, your guess: ");
    if (scanf("%d", &guess) != 1) return 1;
    while (guess != answer) {
        if (guess < answer) {
            printf("Your guess is too low!\n");
        } else {
            printf("Your guess is too high!\n");
        }
        ++tries;
        printf("Player 2, your guess: ");
        if (scanf("%d", &guess) != 1) return 1;
    }
    printf("Congratulations! You found %d in %d %s.\n", answer, tries,
           tries == 1 ? "try" : "tries");
    return 0;
}
```

```
Player 1, type the secret number: 37
Player 2, your guess: 50
Your guess is too high!
Player 2, your guess: 25
Your guess is too low!
Player 2, your guess: 37
Congratulations! You found 37 in 3 tries.

Player 1, type the secret number: 7
Player 2, your guess: 7
Congratulations! You found 7 in 1 try.
```

The first run is the hand example: two wrong guesses with the right feedback, then the congratulation after 3 tries. In the second run the first guess is right: the loop condition is false at once, the body never runs, and the congratulation says “1 try”. The word is chosen by the conditional operator `tries == 1 ? "try" : "tries"`, printed with `%s`, the conversion for text. The counter `tries` starts at 1, not 0, because the guess read before the loop is already the first try.

4.4.5 The Python solution

Listing 4.5 `guess.py`: the same read-ahead loop in Python.

```python
# guess.py - Problem 4.2: the guessing game for two players
tries = 1

answer = int(input("Player 1, type the secret number: "))

guess = int(input("Player 2, your guess: "))
while guess != answer:
    if guess < answer:
        print("Your guess is too low!")
    else:
        print("Your guess is too high!")
    tries += 1
    guess = int(input("Player 2, your guess: "))

word = "try" if tries == 1 else "tries"
print(f"Congratulations! You found {answer} in {tries} {word}.")
```

Listing 4.5 mirrors the C program statement for statement. It prints the same two runs; the choice of "try" or "tries" is made with Python's conditional expression before the last `print`.

4.4.6 A tempting wrong solution

A `do`...`while` seems made for this problem: "read a guess, judge it, repeat until it is right". **Listing 4.6** writes exactly that.

Listing 4.6 `guess_wrong.c`: the body judges every guess, including the winning one.

```c
/* guess_wrong.c - a tempting wrong loop: judges the winning guess too */
#include <stdio.h>

int main(void)
{
    int answer, guess;

    printf("Player 1, type the secret number: ");
    if (scanf("%d", &answer) != 1) return 1;

    do {
        printf("Player 2, your guess: ");
        if (scanf("%d", &guess) != 1) return 1;
        if (guess < answer)
            printf("Your guess is too low!\n");
        else
            printf("Your guess is too high!\n");
    } while (guess != answer);
    printf("Congratulations!\n");
    return 0;
}
```

```
Player 1, type the secret number: 37
Player 2, your guess: 50
Your guess is too high!
Player 2, your guess: 25
Your guess is too low!
Player 2, your guess: 37
Your guess is too high!
Congratulations!
```

The winning guess 37 is followed by "Your guess is too high!" and only then by the congratulation. The body of a `do`...`while` runs completely before the condition is tested, so the right guess is

judged too, and the `else` branch, which assumed "not low means high", fires for "equal". The program works on every wrong guess and fails on the one that matters most. Two repairs are possible: keep the read-ahead `while` of **Listing 4.4**, in which the body only ever sees wrong guesses, or add a third case `else if (guess > answer)` so that equality prints nothing. The first is cleaner, because it puts the test where the meaning is: judge only while the guess is wrong.

> **CHECK YOURSELF.** Limit the game to 5 tries: after the fifth wrong guess, the program prints "Out of tries! The number was 37." instead of the congratulation. What becomes of the loop condition, and how does the program know after the loop which message to print?
>
> *Answer.* The loop runs while `guess != answer && tries < 5`, increasing `tries` and reading again inside. After the loop, one of the two conditions has failed; test `if (guess == answer)` to print the congratulation, and print the "Out of tries" message otherwise. Testing `tries == 5` instead would be wrong, because a player who hits the answer exactly on the fifth try also has `tries == 5`.

4.5 Problem 4.3: 100 to the Power n

4.5.1 The problem

> **PROBLEM 4.3.** Write a program that reads an integer n and prints the value of 100^n. Hint: an `int` has 32 bits, and its range is from −2,147,483,648 to 2,147,483,647.

The hint is the whole problem. $100^5 = 10{,}000{,}000{,}000$ is already more than four times the largest `int`, so the obvious program, which multiplies 100 by itself n times in an `int`, is wrong for every $n \geq 5$. The problem asks whether you notice, and what you do about it. We decide: the program must print 100^n *exactly*, digit for digit, for every n a person could reasonably type; we accept n from −100,000 to 100,000; and for negative n we print the exact decimal fraction, since $100^{-2} = 0.0001$ is a perfectly good number. We also report whether the value would have fitted into an `int` or a `long long`.

4.5.2 Understanding it by hand

The key observation is that we do not need to *compute* the number in order to *print* it. Since $100 = 10^2$,

$$100^n = (10^2)^n = 10^{2n} \tag{4.2}$$

and 10^{2n}, written in decimal, is a 1 followed by $2n$ zeros. So $100^0 = 1$, $100^1 = 100$, $100^3 = 1\,000\,000$ (a 1 and six zeros), and 100^{12} is a 1 and 24 zeros. For a negative exponent $n = -m$, Equation (4.2) gives 10^{-2m}, which is written "0." followed by $2m - 1$ zeros and then a 1: $100^{-1} = 0.01$ and $100^{-2} = 0.0001$.

How far do the built-in types reach? **Table 4.1** compares the powers with the limits: an `int` holds $100^4 = 10^8$ but not 10^{10}; a `long long`, whose largest value is about 9.22×10^{18}, holds $100^9 = 10^{18}$ but not 10^{20}.

Table 4.1 The powers of 100 against the limits of C's integer types: `int` stops after $n = 4$, `long long` after $n = 9$.

n	100^n	Fits in `int` (max about 2.1×10^9)?	Fits in `long long` (max about 9.2×10^{18})?
4	10^8	yes	yes
5	10^{10}	no	yes
9	10^{18}	no	yes
10	10^{20}	no	no

4.5.3 Designing it on paper

The design prints digits instead of computing a number, as **Pseudocode 4.3** shows. A `for` loop prints the zeros, because their number, $2n$ or $2m - 1$, is known before the loop starts.

Pseudocode 4.3 The design of Problem 4.3: print the digits of 10^{2n} instead of computing the number.

```
POWER-OF-100
1  read n
2  if n < -100000 or n > 100000
3      print a message and stop
4  print "100^n = "
5  if n ≥ 0
6      print "1"
7      repeat 2n times: print "0"
8  else
9      print "0."
10     repeat -2n - 1 times: print "0"
11     print "1"
12 print whether n fits int (0 ≤ n ≤ 4) and long long (0 ≤ n ≤ 9)
```

4.5.4 The C solution

Listing 4.7 is **Pseudocode 4.3** written in C; the two `for` loops print the zeros, and the conditional operator chooses "yes" or "no" for the two types.

Listing 4.7 `power100.c`: 100 to the power n, printed digit by digit, never overflowing.

```c
/* power100.c - Problem 4.3: print 100 to the power n, exactly, for any n */
#include <stdio.h>

int main(void)
{
    int n;

    printf("n: ");
    if (scanf("%d", &n) != 1) return 1;
    if (n < -100000 || n > 100000) {
        printf("Please keep n between -100000 and 100000.\n");
        return 1;
    }

    printf("100^%d = ", n);
    if (n >= 0) {
        printf("1");                          /* 100^n = 1 and 2n zeros */
        for (int i = 1; i <= 2 * n; ++i)
            printf("0");
    } else {
        printf("0.");                         /* 100^-m = 0. (2m-1 zeros) 1 */
        for (int i = 1; i <= -2 * n - 1; ++i)
            printf("0");
        printf("1");
    }
    printf("\n");
    printf("fits in int: %s, fits in long long: %s\n",
           n >= 0 && n <= 4 ? "yes" : "no", n >= 0 && n <= 9 ? "yes" : "no");
    return 0;
}
```

```
n: 0
100^0 = 1
fits in int: yes, fits in long long: yes
n: 3
100^3 = 1000000
fits in int: yes, fits in long long: yes
n: 12
100^12 = 1000000000000000000000000
fits in int: no, fits in long long: no
n: -2
100^-2 = 0.0001
fits in int: no, fits in long long: no
```

Count the zeros in the third run: 24, so the number has 25 digits, far beyond any built-in integer type, and yet no arithmetic ever went near a limit. The largest number the program computes is the loop bound `2 * n`, at most 200,000. That is why the range check on n exists: without it, a user typing 2,000,000,000 would make `2 * n` overflow the `int`, which the C standard calls undefined behavior [3] and the sanitizer reports. For $n = 0$ the loop runs zero times and the output is simply 1, and for $n = -2$ the loop prints $2 \cdot 2 - 1 = 3$ zeros between "0." and the closing 1.

4.5.5 The Python solution

Listing 4.8 `power100.py`: the same digit-printing loop in Python.

```python
# power100.py - Problem 4.3: print 100 to the power n, exactly, for any n
n = int(input("n: "))
if n < -100000 or n > 100000:
    print("Please keep n between -100000 and 100000.")
    raise SystemExit(1)

print(f"100^{n} = ", end="")
if n >= 0:
    print("1", end="")                      # 100^n = 1 and 2n zeros
    for i in range(2 * n):
        print("0", end="")
else:
    print("0.", end="")                     # 100^-m = 0. (2m-1 zeros) 1
    for i in range(-2 * n - 1):
        print("0", end="")
    print("1", end="")
print()
fits_int = "yes" if 0 <= n <= 4 else "no"
fits_ll = "yes" if 0 <= n <= 9 else "no"
print(f"fits in int: {fits_int}, fits in long long: {fits_ll}")
```

Listing 4.8 prints the same four runs. It deserves a remark, because Python could have taken a shortcut: its integers have no limit, so `print(100 ** n)` prints 100^{12} exactly [4], and even `"1" + "0" * (2 * n)` builds the digits in one expression. We keep the loop so that the two programs have the same design, but it is worth knowing that the whole difficulty of this problem is a property of C's fixed-size integers, not of the mathematics.

4.5.6 A tempting wrong solution

C does have a power function, `pow` from Chapter 3. **Listing 4.9** lets it do the work and prints the result with no decimals.

Listing 4.9 `power100_wrong.c`: `pow` returns a `double`, which has only about 16 significant digits.

```c
/* power100_wrong.c - a tempting wrong answer: let pow() do it */
#include <stdio.h>
#include <math.h>

int main(void)
{
    int n;

    printf("n: ");
    if (scanf("%d", &n) != 1) return 1;
    printf("100^%d = %.0f\n", n, pow(100.0, n));
    return 0;
}
```

```
n: 3
100^3 = 1000000
n: 12
100^12 = 999999999999999983222784
```

For $n = 3$ the answer is right, and a programmer who tests only small values will believe the program works. For $n = 12$ it prints 999999999999999983222784 instead of 1 followed by 24 zeros. The reason is that `pow` returns a `double`, which stores about 15 to 17 significant decimal digits [5]. The number 10^{24} has no exact `double` representation; the nearest `double` is 999,999,999,999,999,983,222,784, and `%.0f` faithfully prints *that* number. The Python version, `100.0 ** n` formatted with `:.0f`, prints the same wrong digits, because it computes with the

same `double`. The lesson is general: a `double` is exact only for whole numbers up to 2^{53}, about 9×10^{15}, and anything larger is rounded.

> **CHECK YOURSELF.** Change the program to print 1000^n exactly for $n \geq 0$. How many zeros does the loop print, and up to which n does 1000^n still fit in a `long long`?
>
> *Answer.* $1000^n = 10^{3n}$, so the loop prints $3n$ zeros after the 1. A `long long` holds 10^{18}, which is 1000^6, but not $10^{21} = 1000^7$, so the answer is $n \leq 6$.

4.6 Problem 4.4: Rock, Paper, Scissors

4.6.1 The problem

> **PROBLEM 4.4.** Write a rock, paper, scissors game for two players. Each player enters 1 (rock), 2 (paper) or 3 (scissors); an invalid number must be asked again. The program announces the result: rock beats scissors, scissors beats paper, and paper beats rock. As long as the process is the same, the design is free.

We decide the details. Each player is asked with a prompt that lists the three numbers; any other number prints "Please type 1, 2 or 3." and asks the same player again. The result is printed in words, first what each player showed and then the verdict: "It is a draw.", "Player 1 wins!" or "Player 2 wins!". One game per run; playing several rounds is the check-yourself variant.

4.6.2 Understanding it by hand

There are $3 \times 3 = 9$ combinations and three verdicts. **Figure 4.1** draws the rule as a cycle: each object beats the one its arrow points to. Rock (1) beats scissors (3), scissors (3) beats paper (2), and paper (2) beats rock (1).

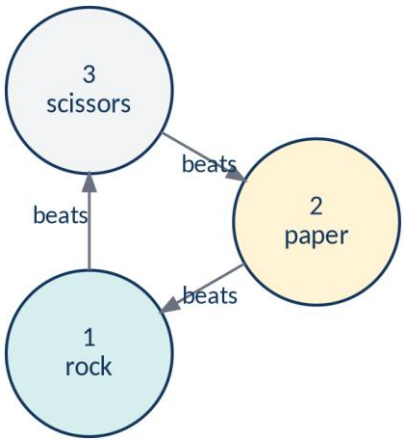


Figure 4.1 The rule of rock, paper, scissors as a cycle: every object beats the next one along the arrows.

Nine `if` statements, one per combination, would work and are easy to check. But the cycle suggests a shorter rule. Going once around the cycle in the direction $1 \to 3 \to 2 \to 1$, the winner's number is always one *more* than the loser's, counting modulo 3: paper (2) beats rock (1), scissors (3) beats paper (2), and rock (1) beats scissors (3), since $1 \equiv 3 + 1 \pmod 3$. So player 1 wins exactly when $p_1 - p_2$ leaves remainder 1 when divided by 3. Because $p_1 - p_2$ can be negative and C's `%` keeps the sign of the dividend (Chapter 1), we add 3 first, which does not change the remainder but makes the number positive; so player 1 wins exactly when

$$(p_1 - p_2 + 3) \bmod 3 = 1 \tag{4.3}$$

A formula like this must be checked, not trusted. **Table 4.2** evaluates Equation (4.3) for all nine cases.

Table 4.2 All nine combinations checked against Equation (4.3): remainder 0 is a draw, 1 means player 1 wins, 2 means player 2 wins.

p_1	p_2	p_1-p_2+3	Mod 3	Verdict by the formula	Correct?
1 rock	1 rock	3	0	draw ($p_1 = p_2$)	yes
1 rock	2 paper	2	2	player 2	yes
1 rock	3 scissors	1	1	player 1	yes
2 paper	1 rock	4	1	player 1	yes
2 paper	2 paper	3	0	draw	yes
2 paper	3 scissors	2	2	player 2	yes
3 scissors	1 rock	5	2	player 2	yes
3 scissors	2 paper	4	1	player 1	yes
3 scissors	3 scissors	3	0	draw	yes

4.6.3 Designing it on paper

Two “ask until valid” loops, one per player, are the `do`…`while` of Chapter 3. Printing the names is a multi-way choice on one integer, a `switch`. The verdict is an `if`…`else if`…`else` with Equation (4.3) in the middle. **Pseudocode 4.4** puts it together.

Pseudocode 4.4 The design of Problem 4.4: two validating loops, a `switch` for the names, and the rule of Equation (4.3).

```
ROCK-PAPER-SCISSORS
1  repeat: read p1, and if it is not 1, 2 or 3 print a message
2  until p1 is 1, 2 or 3
3  repeat the same for p2
4  print the names of p1 and p2
5  if p1 = p2
6      print "It is a draw."
7  else if (p1 - p2 + 3) mod 3 = 1
8      print "Player 1 wins!"
9  else
10     print "Player 2 wins!"
```

4.6.4 The C solution

Listing 4.10 translates **Pseudocode 4.4**: two `do`…`while` loops, two `switch` statements and one chain.

Listing 4.10 `rps.c`: validated input, names printed with `switch`, and the winner found with one remainder.

```c
/* rps.c - Problem 4.4: rock, paper, scissors for two players */
#include <stdio.h>

int main(void)
{
    int p1, p2;

    do {
        printf("Player 1 (1 = rock, 2 = paper, 3 = scissors): ");
        if (scanf("%d", &p1) != 1) return 1;
        if (p1 < 1 || p1 > 3)
            printf("Please type 1, 2 or 3.\n");
    } while (p1 < 1 || p1 > 3);
    do {
        printf("Player 2 (1 = rock, 2 = paper, 3 = scissors): ");
        if (scanf("%d", &p2) != 1) return 1;
        if (p2 < 1 || p2 > 3)
            printf("Please type 1, 2 or 3.\n");
    } while (p2 < 1 || p2 > 3);

    printf("Player 1 shows ");
    switch (p1) {
    case 1: printf("rock"); break;
    case 2: printf("paper"); break;
    default: printf("scissors"); break;
    }
    printf(", player 2 shows ");
    switch (p2) {
    case 1: printf("rock"); break;
    case 2: printf("paper"); break;
    default: printf("scissors"); break;
    }
    printf(".\n");

    if (p1 == p2)
        printf("It is a draw.\n");
    else if ((p1 - p2 + 3) % 3 == 1)        /* 1 beats 3, 2 beats 1, 3 beats 2 */
        printf("Player 1 wins!\n");
    else
        printf("Player 2 wins!\n");
    return 0;
}
```

```
Player 1 (1 = rock, 2 = paper, 3 = scissors): 0
Please type 1, 2 or 3.
Player 1 (1 = rock, 2 = paper, 3 = scissors): 4
Please type 1, 2 or 3.
Player 1 (1 = rock, 2 = paper, 3 = scissors): 2
Player 2 (1 = rock, 2 = paper, 3 = scissors): 1
Player 1 shows paper, player 2 shows rock.
Player 1 wins!
Player 1 (1 = rock, 2 = paper, 3 = scissors): 1
Player 2 (1 = rock, 2 = paper, 3 = scissors): 3
Player 1 shows rock, player 2 shows scissors.
Player 1 wins!
Player 1 (1 = rock, 2 = paper, 3 = scissors): 3
Player 2 (1 = rock, 2 = paper, 3 = scissors): 3
Player 1 shows scissors, player 2 shows scissors.
It is a draw.
```

The first run tests the validation: player 1 types 0 and 4, each rejected, then 2 (paper); player 2 types 1 (rock); paper beats rock. The second run is the case where the “+ 3” matters: rock against scissors gives $1 - 3 + 3 = 1$, remainder 1, player 1 wins. The third is a draw. Each `switch` prints

one word and ends every case with `break`; the last case uses `default` for 3, which is safe because the loops above guarantee that only 1, 2 or 3 can arrive.

4.6.5 The Python solution

Listing 4.11 `rps.py`: the same game with `while True` validation and `match`.

```
# rps.py - Problem 4.4: rock, paper, scissors for two players
while True:
    p1 = int(input("Player 1 (1 = rock, 2 = paper, 3 = scissors): "))
    if 1 <= p1 <= 3:
        break
    print("Please type 1, 2 or 3.")
while True:
    p2 = int(input("Player 2 (1 = rock, 2 = paper, 3 = scissors): "))
    if 1 <= p2 <= 3:
        break
    print("Please type 1, 2 or 3.")

print("Player 1 shows ", end="")
match p1:
    case 1: print("rock", end="")
    case 2: print("paper", end="")
    case _: print("scissors", end="")
print(", player 2 shows ", end="")
match p2:
    case 1: print("rock", end="")
    case 2: print("paper", end="")
    case _: print("scissors", end="")
print(".")

if p1 == p2:
    print("It is a draw.")
elif (p1 - p2 + 3) % 3 == 1:                # 1 beats 3, 2 beats 1, 3 beats 2
    print("Player 1 wins!")
else:
    print("Player 2 wins!")
```

In **Listing 4.11** the validating loops use the `while True`…`break` idiom, and each `match` has its `case` and its statement on one line, which Python allows when the statement is short. The formula is written exactly as in C. In Python the `+ 3` is not even necessary, because Python's `%` always returns a remainder with the sign of the divisor; we keep it so that the formula is the same in both languages.

4.6.6 A tempting wrong solution

Having found the cycle, it is tempting to write the formula without the `+ 3`: "player 1 wins when $(p_1 - p_2) \bmod 3 = 1$". On paper, in mathematics, that is correct. **Listing 4.12** tries it in C and prints the remainder as well.

Listing 4.12 `rps_wrong.c`: without `+ 3`, C's `%` gives −2 for rock against scissors.

```c
/* rps_wrong.c - a tempting wrong rule: (p1 - p2) % 3 without the + 3 */
#include <stdio.h>

int main(void)
{
    int p1, p2;

    printf("Player 1 (1-3): ");
    if (scanf("%d", &p1) != 1) return 1;
    printf("Player 2 (1-3): ");
    if (scanf("%d", &p2) != 1) return 1;

    printf("(p1 - p2) %% 3 = %d\n", (p1 - p2) % 3);
    if (p1 == p2)
        printf("It is a draw.\n");
    else if ((p1 - p2) % 3 == 1)
        printf("Player 1 wins!\n");
    else
        printf("Player 2 wins!\n");
    return 0;
}
```

```
Player 1 (1-3): 1
Player 2 (1-3): 3
(p1 - p2) % 3 = -2
Player 2 wins!
```

Rock against scissors: $1 - 3 = -2$, and in C `-2 % 3` is −2, not 1, because C's remainder has the sign of the number being divided. The test `== 1` fails, and the program declares that scissors beat rock. Scissors against paper does not suffer, because $3 - 2 = 1$ is positive. The error hits exactly the one combination where the winner has the *smaller* number, rock over scissors, which is why a quick test with paper against rock passes and the bug survives. In Python the plain expression `(p1 - p2) % 3` would give 1 and be correct; the Python version in `python/tempting` imitates C with `math.fmod` to show the same failure. The formula of Equation (4.3), with its `+ 3`, is correct in both languages.

> **CHECK YOURSELF.** Turn the program into a "best of three": rounds are played until one player has won two rounds; draws do not count. What loop do you need around the game, which variables, and what is its condition?
>
> *Answer.* Two counters `wins1` and `wins2`, both 0, and a loop `while (wins1 < 2 && wins2 < 2)` around the reading and judging; a win increases the winner's counter and a draw increases nothing. After the loop, `wins1 == 2` tells who won the match. The loop is a `while`, not a `for`, because the number of rounds is not known in advance: it is at least two and, with draws, unbounded.

4.7 Problem 4.5: The Goblin Battle

4.7.1 The problem

> **PROBLEM 4.5.** A boy fights three goblins, turn by turn. The boy has 200 hit points (HP) and an attack of 10; every third turn he uses a skill that deals 20 instead. Each goblin has 165 HP; their attacks are 3, 4 and 1. In every turn the boy attacks first, then the goblins attack him. Print the battle turn by turn until one side is wiped out, then print the number of turns and the winner's remaining HP.

The statement leaves several decisions open, and each changes the result, so we make them explicit. *Whom does the boy attack?* The first goblin that is still alive: goblin 1 until it falls, then

goblin 2, then goblin 3. *Who hits back?* Only goblins that are still alive *after* the boy's attack; a goblin killed in this turn does not strike. *Can HP go below zero?* No: a hit that exceeds the remaining HP leaves 0, both for goblins and for the boy. *What is printed?* One line per turn, as a table: the turn, the kind of attack, the target, the three goblins' HP, the damage the goblins dealt back, and the boy's HP. *What is "the winner's remaining HP"?* For the boy, his HP; for the goblins, the sum of their remaining HP.

4.7.2 Understanding it by hand

Simulate the first turns on paper. Turn 1: the boy deals 10 to goblin 1 (165 → 155); all three goblins are alive and deal 3 + 4 + 1 = 8; the boy has 192. Turn 2: the same, goblin 1 has 145 and the boy 184. Turn 3 is divisible by 3, so the skill deals 20: goblin 1 has 125, the boy 176. The pattern is clear: every three turns goblin 1 loses 10 + 10 + 20 = 40 HP while the boy loses 24.

We can predict the outcome before writing any code. Goblin 1 needs 165 HP of damage: 12 turns deal 4 × 40 = 160, and turn 13 finishes it, so goblin 1 falls in turn 13 and does not hit back in that turn. From then on only 4 + 1 = 5 damage comes back per turn. Goblin 2 falls in turn 26 and goblin 3 in turn 39. The boy takes 8 × 12 = 96 in turns 1 to 12, 5 × 13 = 65 in turns 13 to 25, and 1 × 13 = 13 in turns 26 to 38, a total of 174, and nothing in turn 39. He wins after 39 turns with 200 − 174 = 26 HP. A prediction like this is the best possible test: if the program disagrees, one of the two is wrong, and finding out which teaches something either way.

4.7.3 Designing it on paper

The **state** of the battle is five numbers: the boy's HP, the three goblins' HP, and the turn number. One iteration of the loop is one complete turn, and **Figure 4.2** shows its steps. The loop continues while the boy is alive *and* at least one goblin is alive, `boy > 0 && (g1 > 0 || g2 > 0 || g3 > 0)`, a condition that uses both logical operators and needs its parentheses.

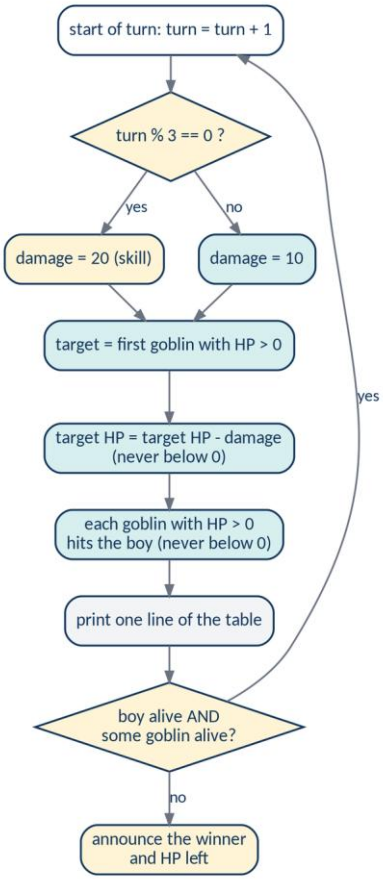


Figure 4.2 One turn of the battle: choose the damage, hit the first living goblin, let the living goblins hit back, print a line, and decide whether the battle goes on.

In pseudocode, **Pseudocode 4.5**:

Pseudocode 4.5 The design of Problem 4.5: five state variables and one loop, one iteration per turn.

```
BATTLE
1  boy ← 200; g1, g2, g3 ← 165; turn ← 0
2  print the table header
3  while boy > 0 and (g1 > 0 or g2 > 0 or g3 > 0)
4      turn ← turn + 1
5      damage ← 20 if turn mod 3 = 0, else 10
6      target ← the first goblin with HP > 0
7      target's HP ← max(0, target's HP – damage)
8      back ← 3·[g1 > 0] + 4·[g2 > 0] + 1·[g3 > 0]
9      boy ← max(0, boy – back)
10     print turn, attack, target, g1, g2, g3, back, boy
11 if boy > 0
12     print "The boy wins" with turn and boy
13 else
14     print "The goblins win" with turn and g1 + g2 + g3
```

On line 8, $[g_1 > 0]$ means 1 if goblin 1 is alive and 0 otherwise, so only living goblins contribute. The order of lines 7 and 8 encodes the decision that a goblin killed in this turn does not hit back.

4.7.4 The C solution

Listing 4.13 `battle.c`: the battle as one `while` loop over five state variables, printed as a table.

```c
/* battle.c - Problem 4.5: one boy against three goblins, turn by turn */
#include <stdio.h>

int main(void)
{
    int boy = 200;                          /* hit points (HP) */
    int g1 = 165, g2 = 165, g3 = 165;
    int turn = 0;

    printf("Turn  Attack  Target  G1   G2   G3   Hit back  Boy\n");
    while (boy > 0 && (g1 > 0 || g2 > 0 || g3 > 0)) {
        ++turn;
        int damage = turn % 3 == 0 ? 20 : 10;   /* skill every third turn */
        int target;

        if (g1 > 0) {                       /* the first goblin still alive */
            target = 1;
            g1 = g1 > damage ? g1 - damage : 0;
        } else if (g2 > 0) {
            target = 2;
            g2 = g2 > damage ? g2 - damage : 0;
        } else {
            target = 3;
            g3 = g3 > damage ? g3 - damage : 0;
        }

        int back = 0;                       /* only living goblins hit */
        if (g1 > 0) back += 3;
        if (g2 > 0) back += 4;
        if (g3 > 0) back += 1;
        boy = boy > back ? boy - back : 0;

        printf("%4d  %-6s     G%d  %3d  %3d  %3d  %8d  %3d\n", turn,
               damage == 20 ? "skill" : "normal", target, g1, g2, g3, back, boy);
    }

    if (boy > 0)
        printf("The boy wins after %d turns with %d HP left.\n", turn, boy);
    else
        printf("The goblins win after %d turns with %d HP left in total.\n",
               turn, g1 + g2 + g3);
    return 0;
}
```

```
Turn  Attack  Target  G1   G2   G3   Hit back  Boy
   1  normal     G1  155  165  165         8  192
   2  normal     G1  145  165  165         8  184
   3  skill      G1  125  165  165         8  176
   4  normal     G1  115  165  165         8  168
   5  normal     G1  105  165  165         8  160
   6  skill      G1   85  165  165         8  152
   7  normal     G1   75  165  165         8  144
   8  normal     G1   65  165  165         8  136
   9  skill      G1   45  165  165         8  128
  10  normal     G1   35  165  165         8  120
  11  normal     G1   25  165  165         8  112
  12  skill      G1    5  165  165         8  104
  13  normal     G1    0  165  165         5   99
  14  normal     G2    0  155  165         5   94
  15  skill      G2    0  135  165         5   89
  16  normal     G2    0  125  165         5   84
  17  normal     G2    0  115  165         5   79
  18  skill      G2    0   95  165         5   74
  19  normal     G2    0   85  165         5   69
  20  normal     G2    0   75  165         5   64
  21  skill      G2    0   55  165         5   59
  22  normal     G2    0   45  165         5   54
  23  normal     G2    0   35  165         5   49
  24  skill      G2    0   15  165         5   44
  25  normal     G2    0    5  165         5   39
  26  normal     G2    0    0  165         1   38
  27  skill      G3    0    0  145         1   37
  28  normal     G3    0    0  135         1   36
  29  normal     G3    0    0  125         1   35
  30  skill      G3    0    0  105         1   34
  31  normal     G3    0    0   95         1   33
  32  normal     G3    0    0   85         1   32
  33  skill      G3    0    0   65         1   31
  34  normal     G3    0    0   55         1   30
  35  normal     G3    0    0   45         1   29
  36  skill      G3    0    0   25         1   28
  37  normal     G3    0    0   15         1   27
  38  normal     G3    0    0    5         1   26
  39  skill      G3    0    0    0         0   26
The boy wins after 39 turns with 26 HP left.
```

The program agrees with the prediction: goblin 1 falls in turn 13 (and the damage back drops from 8 to 5 in that very turn), goblin 2 in turn 26, goblin 3 in turn 39, and the boy wins with 26 HP. The HP never goes below 0 because every subtraction is guarded by a conditional operator: `g1 = g1 > damage ? g1 - damage : 0;` subtracts when enough HP is left and sets 0 otherwise. The damage back is built up with `back += …` only for goblins with HP above 0. The table uses field widths so that the columns line up for the whole battle: `%4d` for the turn, `%-6s` for the kind of attack, `%3d` for each HP. Finally, `damage == 20 ? "skill" : "normal"` names the attack from the damage, so the rule “every third turn” lives in one place only, the line that chooses the damage.

4.7.5 The Python solution

Listing 4.14 `battle.py`: the same battle in Python.

```
# battle.py - Problem 4.5: one boy against three goblins, turn by turn
boy = 200                                   # hit points (HP)
g1 = g2 = g3 = 165
turn = 0

print("Turn  Attack  Target  G1   G2   G3   Hit back  Boy")
while boy > 0 and (g1 > 0 or g2 > 0 or g3 > 0):
    turn += 1
    damage = 20 if turn % 3 == 0 else 10    # skill every third turn

    if g1 > 0:                              # the first goblin still alive
        target = 1
        g1 = g1 - damage if g1 > damage else 0
    elif g2 > 0:
        target = 2
        g2 = g2 - damage if g2 > damage else 0
    else:
        target = 3
        g3 = g3 - damage if g3 > damage else 0

    back = 0                                # only living goblins hit
    if g1 > 0:
        back += 3
    if g2 > 0:
        back += 4
    if g3 > 0:
        back += 1
    boy = boy - back if boy > back else 0

    kind = "skill" if damage == 20 else "normal"
    print(f"{turn:4d}  {kind:<6s}     G{target}  {g1:3d}  {g2:3d}  {g3:3d}"
          f"  {back:8d}  {boy:3d}")

if boy > 0:
    print(f"The boy wins after {turn} turns with {boy} HP left.")
else:
    print(f"The goblins win after {turn} turns with {g1 + g2 + g3} HP left "
          "in total.")
```

Listing 4.14 prints the same 39 lines and the same verdict. The statement `g1 = g2 = g3 = 165` assigns one value to three variables at once. The long `print` is split into two f-strings written side by side inside the parentheses; Python joins adjacent string literals into one, which is a convenient way to keep lines short.

4.7.6 A tempting wrong solution

The tempting version forgets the guards. Goblins always hit back, dead or alive, and HP is simply decreased, without a floor at zero; to keep the output short, it prints only every fifth turn and the last one.

Listing 4.15 `battle_wrong.c`: without the “is it alive?” checks, dead goblins keep fighting and HP goes negative.

```c
/* battle_wrong.c - a tempting wrong battle: nobody checks who is alive */
#include <stdio.h>

int main(void)
{
    int boy = 200;
    int g1 = 165, g2 = 165, g3 = 165;
    int turn = 0;

    while (boy > 0 && g1 + g2 + g3 > 0) {
        ++turn;
        int damage = turn % 3 == 0 ? 20 : 10;
        if (g1 > 0)
            g1 -= damage;               /* may go below zero */
        else if (g2 > 0)
            g2 -= damage;
        else
            g3 -= damage;
        boy -= 3 + 4 + 1;               /* every goblin always hits back */
        if (turn % 5 == 0 || boy <= 0)
            printf("turn %2d: G1 %4d  G2 %4d  G3 %4d  boy %4d\n",
                   turn, g1, g2, g3, boy);
    }
    printf("%s after %d turns.\n", boy > 0 ? "The boy wins" : "The goblins win",
           turn);
    return 0;
}
```

```
turn  5: G1  105  G2  165  G3  165  boy  160
turn 10: G1   35  G2  165  G3  165  boy  120
turn 15: G1   -5  G2  135  G3  165  boy   80
turn 20: G1   -5  G2   75  G3  165  boy   40
turn 25: G1   -5  G2    5  G3  165  boy    0
The goblins win after 25 turns.
```

The result is not a small inaccuracy but the opposite verdict. From turn 13 on, goblin 1 has −5 HP, prints as a negative number, and still deals its 3 damage every turn; the boy keeps losing 8 HP per turn instead of 5 and then 1, and falls in turn 25. Two missing guards change who wins. Notice also that the loop condition `g1 + g2 + g3 > 0` is wrong in a subtler way: with negative HP allowed, a sum above 0 does not mean that any goblin is alive. The correct condition tests each goblin separately, as **Listing 4.13** does.

> **CHECK YOURSELF.** (a) What happens if goblin 2's attack is 6 instead of 4? (b) With the original attacks, what is the smallest starting HP with which the boy still wins?
>
> *Answer.* (a) The goblins win: with 3 + 6 + 1 = 10 damage per turn while all three are alive, the boy falls in turn 24, when goblin 2 still has 15 HP and goblin 3 is untouched; the program prints "The goblins win after 24 turns with 180 HP left in total." (b) The boy loses exactly 174 HP in the whole battle, as computed in Section 4.7.2, and the battle does not depend on his HP as long as he survives. So he needs 175 HP: he then wins after 39 turns with 1 HP left. With 174 HP he falls in turn 38, one turn before the last goblin.

4.8 The Problems Side by Side

Table 4.3 lists the five problems with the structures they needed and the trap each tempting solution fell into. Read the last column once more: none of the five traps is exotic. Integer division, judging after the fact, the limits of a type, the sign of a remainder and a missing guard are the errors that appear in real programs every day.

Table 4.3 The five problems of this chapter, their structures, and the trap of each tempting wrong solution.

Problem	Main structures	Trap of the tempting solution
4.1 Plane figures	sequence, one guard joining six tests with OR	`1 / 2` is integer division, so every triangle has area 0
4.2 Guessing game	read-ahead `while`, `if`...`else`	a `do`...`while` judges the winning guess as "too high"
4.3 100^n	`if`...`else`, `for` over digits	`pow` returns a `double` with about 16 digits
4.4 Rock, paper, scissors	`do`...`while` validation, `switch`, one remainder	without `+ 3`, C's `%` is negative for rock against scissors
4.5 Goblin battle	state variables, `while` with AND and OR, guarded updates	no "alive" checks: dead goblins fight, HP goes negative, the verdict flips

4.9 Using It for Real

The five problems look like exercises, but their shapes are everywhere in real software. Problem 4.1 is every calculator page: read measurements, validate them, apply formulas, print results; a tile shop computing how many boxes a floor needs is the same program with different formulas. Problem 4.2 is every retry loop: a PIN pad that says "wrong PIN" until the right one arrives, with the check-yourself limit of tries that every bank card has. Problem 4.3 is a reminder that the numbers in a program have limits; money counted in whole rupiah fits comfortably into a `long long` (up to about nine quintillion rupiah), but a price multiplied by a large quantity in an `int` can overflow long before that. Problem 4.4 is every validated menu. And Problem 4.5 is every simulation: a queue at the laundry counter simulated minute by minute, stock in a warehouse day by day, or savings month by month all keep a few state variables and update them in one loop, with guards that keep the state sensible.

For Profitina Laundry, the battle's pattern answers a real question. If three washing machines each take 60 minutes per load and customers bring a bag every 15 minutes on a busy morning, how long does the queue grow by noon? The state is the number of bags waiting and the minutes left on each machine; one iteration is one minute; and the guards keep the counts from going negative. With the tools of Chapters 1 to 3 you can already write that simulation.

4.10 Common Mistakes, and How to Avoid Them

- **Deciding the unstated details silently.** Every problem here left decisions open: which goblin is attacked, whether HP may be negative, what "invalid" means. Write your decisions down at the top of the program in a comment; a reader, or a tester, then knows what the program promises.
- **Testing only the typical case.** Each tempting solution passed an ordinary test and failed an unusual one: a first-try guess, a large exponent, rock against scissors. Test the boundaries and the special cases on purpose.
- **Mismatched output text.** When a statement gives exact messages, such as "Your guess is too low!", print them character for character. Automatic tests compare text exactly, and a missing exclamation mark fails them.
- **Trusting a formula without checking every case.** Equation (4.3) is correct, but only a table of all nine cases proves it. When the number of cases is small, check them all.
- **Overflow that the compiler cannot see.** `int` arithmetic that exceeds 2,147,483,647 is undefined behavior in C, and the program may print anything. Estimate the size of your numbers before choosing the type; run with `-fsanitize=undefined` while testing.

- **Printing inside the wrong branch.** In the battle, printing before the goblins hit back, or after the loop instead of inside it, produces a table that disagrees with the rules. Trace one turn on paper and compare it with the first line of output.
- **Leaving the program half-validated.** Rejecting negative lengths but not letters, or checking player 1 but not player 2, gives a false sense of safety. Validate every input the same way.

4.11 Chapter Summary

- A small problem is solved in seven steps: restate it, work examples by hand, design on paper, trace the design, write the program, test it against the hand examples, and attack it with unusual inputs.
- Problem 4.1 is a pure sequence with a guard; its formulas (Equation (4.1)) must be written with `double` constants such as `0.5`, because `1 / 2` is 0 in integer arithmetic.
- Problem 4.2 is a read-ahead `while` loop whose sentinel is the answer; a `do`…`while` that judges before testing mislabels the winning guess.
- Problem 4.3 shows that `int` stops after 100^4 and `long long` after 100^9 (**Table 4.1**); printing the digits of 10^{2n} (Equation (4.2)) is exact for any n, while `pow` returns a rounded `double`.
- Problem 4.4 validates both players with `do`…`while`, prints names with `switch`, and decides the winner with $(p_1 - p_2 + 3) \bmod 3$ (Equation (4.3)), checked on all nine cases (**Table 4.2**); without `+ 3`, C's negative remainder breaks one case.
- Problem 4.5 keeps five state variables and runs one loop iteration per turn (**Figure 4.2**); guarding every attack with "is it alive?" and never letting HP fall below 0 decides the winner, and a prediction on paper (26 HP left after 39 turns) confirms the program.
- The tempting wrong solutions (**Table 4.3**) fail on everyday traps, and each passes an ordinary test; the unusual test is the one that finds them.

4.12 Exercises

1. Extend Problem 4.1 with the area of a parallelogram (base and height) and the perimeter of each figure. Which new value is needed for the trapezoid's perimeter, and why can the program not compute it from the given values?
2. In the guessing game, print after each wrong guess the range in which the answer must lie, for example "The number is between 26 and 49." Which two variables do you need, and how are they updated?
3. Write a program that reads n and prints 2^n exactly for $0 \le n \le 62$ using a `long long` and a `for` loop. Why is 62 the limit, and what happens at 63 under `-fsanitize=undefined`?
4. Rock, paper, scissors has an extended version with lizard and Spock, five objects in which each object beats two others. Number them 1 to 5 in the order rock, Spock, paper, lizard, scissors. Show that player 1 wins exactly when $(p_1 - p_2 + 5) \bmod 5$ is 1 or 2 by checking the case rock against scissors.
5. In the battle, let the boy's skill heal him by 5 HP (up to his starting 200) whenever he uses it. Where does the new statement go, and what is the result?
6. Explain in two sentences why the loop condition `g1 + g2 + g3 > 0` is correct in **Listing 4.13** if HP never goes below 0, but wrong in **Listing 4.15**.

4.13 Solutions to the Exercises

1. The parallelogram's area is base × height, one more line of the same kind. The perimeters of the triangle and the trapezoid need the lengths of their slanted sides, which the input does not contain: a height and a base do not determine a triangle's other two sides. The program must either ask for the missing sides or be limited to special figures, such as a right triangle, whose third side follows from $\sqrt{b^2 + h^2}$. The square's perimeter is $4s$ and the circle's circumference $2\pi r$.

2. Keep `low` and `high`, starting at the smallest and largest allowed secret, for example 1 and 100. After a guess that is too low, set `low = guess + 1`; after one that is too high, set `high = guess - 1`; then print "The number is between %d and %d.". For the secret 37 with guesses 50 and 25, the program says 1 to 49 and then 26 to 49.

3. Start with `long long p = 1;` and multiply by 2 in a loop `for (int i = 1; i <= n; ++i) p *= 2;`, printing with `%lld`. A `long long` holds up to $2^{63} - 1$, so 2^{62} is the largest power of 2 that fits. At $n = 63$ the last multiplication overflows; the sanitizer stops the program with "signed integer overflow", and without it the program may print a negative number. In Python, `2 ** 63` is simply 9223372036854775808.

4. Rock is 1 and scissors is 5: $(1 - 5 + 5) \bmod 5 = 1$, so player 1 wins, which is right, since rock crushes scissors. The pattern works because in the order rock, Spock, paper, lizard, scissors each object beats the one right before it and the one two places before it, counting around the circle of five.

5. After the boy's attack and before the goblins hit back, add: if the damage was 20, set `boy = boy + 5 > 200 ? 200 : boy + 5;`. The battle itself does not change, since the goblins' HP does not depend on the boy's; he heals 5 HP in each of the 13 skill turns and never reaches the cap of 200, so he wins after 39 turns with 26 + 13 × 5 = 91 HP. The order of healing and the goblins' attack must be stated, because it would change the result if the boy's HP were low.

6. When HP never goes below 0, the sum is greater than 0 exactly when at least one goblin has HP above 0, so the test is correct, though less clear than testing each goblin. When HP can be negative, a dead goblin at −5 and a living goblin at 3 give a sum of −2, so the battle would end while a goblin is still alive; the sum no longer means "someone is alive".

References

[1] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ, USA: Princeton Univ. Press, 1957.

[2] GCC Project, "Program Instrumentation Options: -fsanitize=address, -fsanitize=undefined," *Using the GNU Compiler Collection (GCC)*. Accessed: Sep. 25, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[3] *Information Technology — Programming Languages — C*, ISO/IEC 9899:2018, International Organization for Standardization, Geneva, Switzerland, 2018.

[4] Python Software Foundation, "Built-in Types: Numeric Types — int, float, complex," *Python 3.14 Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.python.org/3.14/library/stdtypes.html

[5] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

Chapter 5
Functions: Divide, Name and Conquer

Every program in Chapters 1 to 3 lived inside one function, `main`, and grew longer each time we taught it something new. That works for thirty lines and fails for three hundred: nobody can hold three hundred lines in their head, and when a rule such as a price appears in three places, one of the three is always forgotten when the price changes. This chapter teaches the cure that every working programmer uses: cut the program into small pieces, give each piece a name, and let `main` tell the story by calling those names. By the end of the chapter you will be able to define and call your own functions in C and in Python; explain what a prototype promises and why the compiler wants it; say exactly what happens to an argument when it is passed to a parameter; borrow ready-made functions from the standard library, including the mathematics library that needs `-lm`; decide where a variable is visible and how long it lives (local, global, block and `static`); build a small random-number generator whose results are the same on every computer; write recursive functions, trace them on paper and predict what they cost; and split the Profitina Laundry cashier into a header file, a file of business rules, a short `main` and a test program that checks every rule against answers worked by hand.

5.1 Mbak Rina’s Cashier Has Grown Too Long

Profitina Laundry is a small laundry on a busy lane in Surabaya. Its owner, Pak Hendra, and Mbak Rina, who runs the counter, wash, dry, fold and iron for students from the nearby campus, for families whose washing machine has broken, and for three small hotels that send their bed sheets every morning. Mbak Rina’s nephew Dimas, who studies informatics, wrote the cashier program she uses at the counter: she types the service and the weight, and the program prints the amount to pay. The program was born in Chapter 1, learned to loop over a queue of customers in Chapter 2, and learned `switch` and a daily total in Chapter 3.

Last month Pak Hendra rewrote the price list. *Wash and fold* now costs Rp 7,000 per kilogram, a new *wash and iron* service costs Rp 10,000, *iron only* stays at Rp 6,000, express became a surcharge of half the price on any service, and the minimum charge dropped from three kilograms to two. Dimas opened the program and searched for `8000`, the old price of wash, dry and fold. He found it in the calculation of the bill and changed it. What he did not notice was that the same number appeared a second time, in the line that printed the price list for the customer, and a third time, in the calculation of the daily total. For four days the receipts said one thing and the daily report said another, and it was Mbak Rina’s careful evening count of the cash drawer that discovered the difference.

Nothing in that story is unusual. Anyone who has kept a recipe book knows the problem. A good recipe for *nasi goreng* does not explain again how to make the chili paste: it says “two spoons of sambal (see page 12)”. The sambal recipe is written once, on page 12, and every dish that needs it points there. When the cook finds a better sambal, page 12 changes and every dish improves at once. A program needs the same habit. The rule “wash and fold costs Rp 7,000 per kilogram” should be written in exactly one place, under a name, and every part of the program that needs the price should ask that name.

That named, reusable piece of a program is a **function**. You have used functions since your first program: `printf` and `scanf` are functions that someone else wrote, and `main` is a function that you write. In this chapter you start writing the others yourself. The benefit is not only for Mbak Rina. A tailor who charges per piece, a photocopy kiosk that charges per page, a motorcycle workshop that charges per job: each of them has a handful of rules, and each rule deserves a name and a home.

5.2 The Idea in Plain Words

5.2.1 A function is a named piece of work

> **DEFINITION.** A *function* is a named block of code that does one job. It may receive input values, called *parameters*, and it may give back one result, called its *return value*. Writing the function (its header and body) is called *defining* it; using it by writing its name followed by parentheses is called *calling* it. The values written between the parentheses of a call are the *arguments*.

Think of a function as a small machine with a label on it. The label says what the machine is called and what goes in: `bill_for(price_per_kg, kg)`. You put in two numbers, the machine does its work, and out comes one number, the bill. Whoever uses the machine does not need to see its insides; they only need the label. This is exactly how you have used `printf` for four chapters without ever reading its code.

A C function definition has four parts. Read this one from left to right:

```
long   bill_for   (int price_per_kg, int kg)   { ... }
^^^^   ^^^^^^^^    ^^^^^^^^^^^^^^^^^^^^^^^^   ^^^^^^^
type   name         parameters                 body
```

The **return type** (`long`) says what kind of value comes out. The **name** follows the same rules as a variable name. The **parameter list** declares, like ordinary variable declarations separated by commas, the inputs the function expects. The **body** between braces is ordinary C code, and somewhere in it a `return` statement says which value comes out. A function that gives nothing back has the return type `void`, and a function that takes nothing in has the parameter list `(void)`, as `main` has had since Chapter 1.

5.2.2 What happens during a call

When the computer meets a call such as `bill_for(7000, 3)`, four things happen in order, and **Figure 5.1** shows them.

1. The arguments are evaluated: `7000` and `3` are already values, but an argument may be any expression, such as `price * 2`.
2. Fresh variables are created for the parameters, and each argument is **copied** into its parameter: `price_per_kg` receives 7000 and `kg` receives 3.
3. The body runs. Here it computes `7000 * 3` and reaches `return bill;`.
4. The returned value, 21000, travels back to the place of the call, the parameters and local variables of the function disappear, and the caller continues exactly where it left off, now holding 21000 as the value of the call.

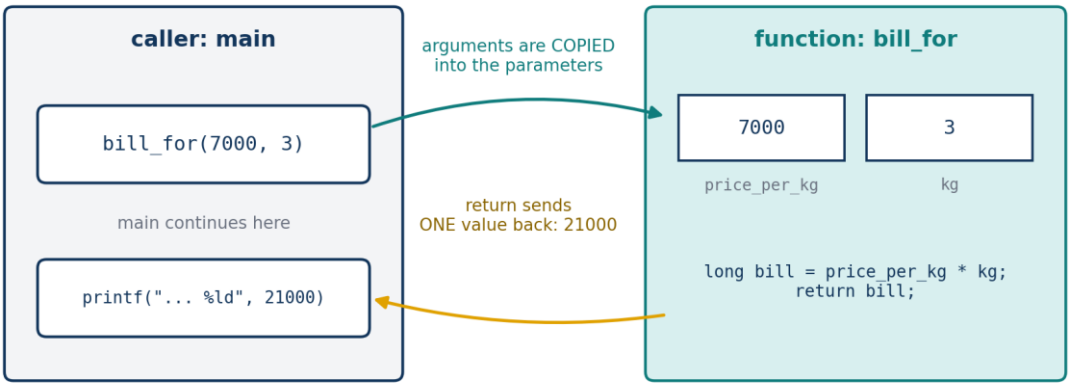


Figure 5.1 A call of `bill_for(7000, 3)`: the arguments are copied into the parameters, the body runs, and `return` sends one value back to the place of the call, where `main` continues.

Step 2 deserves a second reading, because it explains a surprise that every beginner meets. The parameter receives a *copy*. If the function changes its parameter, it changes its own copy, and the caller's variable is untouched. This rule is called **pass-by-value**, and C uses it for every argument. Chapter 7 shows how a function can nevertheless change a variable of its caller, by receiving the variable's address instead of its value; for now, a function talks back to its caller only through its return value.

5.2.3 Divide and conquer: why small pieces win

Why go to the trouble? Programmers have four answers, and each is worth remembering because each is a question you can ask of your own code.

- **One rule, one place.** The price of a service lives in one function. When it changes, one line changes, and Mbak Rina's four days of wrong reports cannot happen.
- **A name explains.** `charged_kg(weight)` says what it does. The three lines inside it do not, until you have read them carefully. A `main` built from well-named calls reads like the story of the program.
- **A piece can be tested alone.** You can call `bill_for` with numbers whose answer you have worked out by hand and compare, without typing a whole day of orders. Section 5.10 does exactly this.
- **A piece can be reused.** The function that prints an amount as `Rp 1.234.567` will print receipts, the daily report and the price list, and Chapter 6 will reuse it unchanged.

The strategy of cutting a big problem into smaller ones, solving each, and combining the answers is called **divide and conquer**, and in software design it goes back to the idea of *stepwise refinement* described by Niklaus Wirth [1]: start from a one-line description of the whole program, and refine each line into a few smaller steps until every step is easy to write. Kernighan and Ritchie put the practical rule in one sentence in their classic book: functions break large computing tasks into smaller ones, and let people build on what others have done instead of starting over [2].

5.3 Your First Functions

5.3.1 One rule, written once

Listing 5.1 is the smallest useful program with a function of our own. The rule "a bill is the price per kilogram times the kilograms" lives in `bill_for`, and `main` calls it four times for three customers.

Listing 5.1 `first_function.c`: one bill rule, defined once above `main` and called four times.

```c
/* first_function.c - one bill rule, written once, used three times */
#include <stdio.h>

/* definition: the header names the inputs, the body computes the answer */
long bill_for(int price_per_kg, int kg)
{
    long bill = (long) price_per_kg * kg;
    return bill;
}

int main(void)
{
    printf("Bu Ani     : Rp %ld\n", bill_for(7000, 3));
    printf("Pak Budi   : Rp %ld\n", bill_for(10000, 5));

    long both = bill_for(7000, 4) + bill_for(6000, 2);   /* two calls */
    printf("Mbak Citra : Rp %ld\n", both);
    return 0;
}
```

Look at the cast `(long) price_per_kg * kg`. The parameters are `int`, and on some computers an `int` holds at most 32767 (the C standard guarantees no more), while on every desktop computer today it holds a little over two billion. The product of two `int` values is computed as an `int`, so without the cast a large order could overflow before the result reaches the `long` variable. Converting one factor to `long` first makes the whole multiplication happen in `long`. It is the same lesson as the cast before a division in Chapter 2: the type of an operation is decided by its operands, not by the variable that will receive the result.

Listing 5.2 is the same program in Python. The keyword `def` starts a definition, the parameters have no declared types, and the body is the indented block under the header.

Listing 5.2 `first_function.py`: the same rule as a Python function.

```python
# first_function.py - one bill rule, written once, used three times


# definition: the header names the inputs, the body computes the answer
def bill_for(price_per_kg, kg):
    bill = price_per_kg * kg
    return bill


def main():
    print(f"Bu Ani     : Rp {bill_for(7000, 3)}")
    print(f"Pak Budi   : Rp {bill_for(10000, 5)}")

    both = bill_for(7000, 4) + bill_for(6000, 2)   # two calls
    print(f"Mbak Citra : Rp {both}")


if __name__ == "__main__":
    main()
```

Both programs print the same three lines:

```
Bu Ani     : Rp 21000
Pak Budi   : Rp 50000
Mbak Citra : Rp 40000
```

The Python program ends with two lines that you will see at the bottom of every Python program in this book: `if __name__ == "__main__": main()`. Python runs a file from top to bottom. The `def` statements only *create* the functions; nothing is called until the last line calls `main`. The test on `__name__` means "only if this file was started as a program, not imported as a module by another file". Section 5.8 uses exactly this: `cashier.py` imports the functions of `laundry.py` without running anything in it.

5.3.2 Prototypes, `void` functions and pass-by-value

In **Listing 5.1** the function was defined above `main`, so the compiler had read its header before it met the first call. Larger programs prefer the opposite order, `main` first, because `main` is the table of contents of the program. Then the compiler meets a call to a function it has not seen yet. C solves this with a **prototype**: the header of the function followed by a semicolon, placed before the first call. A prototype is a promise: "a function with this name, these parameter types and this return type will be defined somewhere". With the promise the compiler can check every call, convert arguments to the parameter types, and complain when a call has the wrong number of arguments.

Listing 5.3 puts four prototypes at the top, `main` next, and the definitions last. It also shows two new kinds of function. `print_rule` returns nothing (`void`) and exists only for what it does, printing a line of symbols. `max3` is built from `max2`: functions may call other functions, and a small function reused inside a bigger one is the everyday form of divide and conquer.

Listing 5.3 `prototypes.c`: prototypes at the top, `main` first, `void` functions, a function built from another, and a demonstration of pass-by-value.

```c
/* prototypes.c - prototypes, void functions and pass-by-value */
#include <stdio.h>

void print_rule(char symbol, int width);    /* prototypes: the promises */
int max2(int a, int b);
int max3(int a, int b, int c);
long add_express_fee(long bill);

int main(void)
{
    print_rule('=', 32);
    printf("Heaviest bag today: %d kg\n", max3(4, 9, 6));
    print_rule('-', 32);

    long bill = 30000;
    long with_fee = add_express_fee(bill);
    printf("bill in main     : %ld\n", bill);      /* still 30000 */
    printf("with express fee : %ld\n", with_fee);
    print_rule('=', 32);
    return 0;
}

void print_rule(char symbol, int width)
{
    for (int i = 0; i < width; i++)
        putchar(symbol);
    putchar('\n');
}

int max2(int a, int b)
{
    return a > b ? a : b;
}

int max3(int a, int b, int c)
{
    return max2(max2(a, b), c);                 /* a function using a function */
}

long add_express_fee(long bill)
{
    bill = bill + bill / 2;                     /* changes the COPY only */
    printf("bill in function : %ld\n", bill);
    return bill;
}
```

The last function is the experiment that proves pass-by-value. `add_express_fee` receives the bill, adds half of it, and prints the new value of its parameter: 45000. Back in `main`, the variable `bill` still holds 30000, because the function only ever had a copy. The caller gets the new amount only through the return value, which `main` stores in `with_fee`.

```
================================
Heaviest bag today: 9 kg
--------------------------------
bill in function : 45000
bill in main     : 30000
with express fee : 45000
================================
```

> **WARNING:** In old C, a call to a function without a prototype was allowed, and the compiler guessed that the function returned `int`. The guess was often wrong and the bugs were strange. Since C99 the language no longer allows it [3], and `gcc` reports it. Always give a prototype, or define the function before its first use.

5.3.3 Python's view: a name must exist when it is called

Python has no prototypes. **Listing 5.4** defines `main` first and the helpers after it, exactly like the C program, and it works, because Python does not look up the name `max3` when it *reads* `main`; it looks it up when `main` *runs*, and by then the last line of the file has run every `def`. If you moved the call `main()` to the top of the file, Python would stop with `NameError: name 'print_rule' is not defined`.

Listing 5.4 `prototypes.py`: the same program in Python, where a function must exist when it is called, not when it is read.

```
# prototypes.py - Python needs no prototypes: a name must exist when it is CALLED


def main():
    print_rule("=", 32)
    print(f"Heaviest bag today: {max3(4, 9, 6)} kg")
    print_rule("-", 32)

    bill = 30000
    with_fee = add_express_fee(bill)
    print(f"bill in main     : {bill}")        # still 30000
    print(f"with express fee : {with_fee}")
    print_rule("=", 32)


def print_rule(symbol, width):
    print(symbol * width)


def max2(a, b):
    return a if a > b else b


def max3(a, b, c):
    return max2(max2(a, b), c)               # a function using a function


def add_express_fee(bill):
    bill = bill + bill // 2                  # rebinds the LOCAL name only
    print(f"bill in function : {bill}")
    return bill


if __name__ == "__main__":
    main()                                   # every def above has run by now
```

The Python output is identical, including the proof of pass-by-value. Python's rule is worded differently (a parameter is a new *name* for the argument's object, and assigning to the parameter rebinds only that name [4]) but for numbers the effect is the same as in C: the caller's variable does not change. Chapter 7 returns to the difference when it matters, for lists.

5.4 Borrowing Functions: The Standard Library

Nobody writes a square root from scratch. C comes with a **standard library**, a collection of functions described by the C standard itself [3], grouped by subject. To use a group, you `#include` its **header**, a file of prototypes. **Table 5.1** lists the headers this book uses most.

Table 5.1 The standard headers used most in this book; a header holds prototypes, the library holds the compiled functions.

Header	What it offers	Examples
`<stdio.h>`	input and output	`printf`, `scanf`, `putchar`
`<math.h>`	mathematics on `double`	`sqrt`, `pow`, `ceil`, `floor`, `round`, `fabs`
`<stdlib.h>`	general utilities	`abs`, `rand`, `srand`, `qsort`, `malloc`
`<string.h>`	character strings	`strlen`, `strcpy`, `strcmp`
`<ctype.h>`	single characters	`isdigit`, `toupper`
`<time.h>`	clock and calendar	`time`, `clock`
`<limits.h>`	limits of the integer types	`INT_MAX`, `INT_MIN`
`<stdint.h>`	integers of exact width	`uint32_t`, `int64_t`

Profitina Laundry needs one mathematical function right away. Mbak Rina charges **every started kilogram**: a bag of 3.4 kg is charged as 4 kg. She also charges at least 2 kg, because washing a single shirt costs her almost as much water and electricity as washing a small bag. In mathematical notation, with w the weight in kilograms and $\lceil w \rceil$ the *ceiling* of w, the smallest whole number that is not less than w:

$$k = \max(2,\ \lceil w \rceil) \tag{5.1}$$

The C library has the ceiling as `ceil` in `<math.h>`. **Listing 5.5** prints a small table that compares `floor` (round down), `round` (round to the nearest whole number) and `ceil` (round up), and applies Equation (5.1) in a function `charged_kg`.

Listing 5.5 `rounding.c`: `floor`, `round` and `ceil` from `<math.h>`, and the laundry's charging rule as a function.

```
/* rounding.c - borrowing ceil, floor and round from the math library */
#include <stdio.h>
#include <math.h>

int charged_kg(double weight)
{
    int kg = (int) ceil(weight);          /* every started kilogram counts */
    return kg < 2 ? 2 : kg;               /* but never less than 2 kg */
}

int main(void)
{
    double weights[] = {0.8, 2.0, 2.1, 3.5, 4.0, 7.25};
    printf("weight   floor   round    ceil  charged\n");
    for (int i = 0; i < 6; i++) {
        double w = weights[i];
        printf("%6.2f  %6.1f  %6.1f  %6.1f  %5d kg\n",
               w, floor(w), round(w), ceil(w), charged_kg(w));
    }
    printf("sqrt(2) = %.6f, pow(1.1, 12) = %.4f\n", sqrt(2.0), pow(1.1, 12));
    return 0;
}
```

```
weight   floor   round    ceil  charged
  0.80     0.0     1.0     1.0      2 kg
  2.00     2.0     2.0     2.0      2 kg
  2.10     2.0     2.0     3.0      3 kg
  3.50     3.0     4.0     4.0      4 kg
  4.00     4.0     4.0     4.0      4 kg
  7.25     7.0     7.0     8.0      8 kg
sqrt(2) = 1.414214, pow(1.1, 12) = 3.1384
```

Read the row for 2.10: `floor` and `round` both give 2, but the customer started a third kilogram, so `ceil` gives 3 and the charge is 3 kg. The row for 0.80 shows the minimum at work: `ceil` gives 1, and `charged_kg` raises it to 2.

On Linux, a program that uses `<math.h>` must be linked with the mathematics library by adding `-lm` at the *end* of the command line, because the linker reads libraries in order and only takes from a library the functions that earlier files asked for [5]:

```
gcc -std=c17 -Wall -Wextra -o rounding rounding.c -lm
```

On macOS and on Windows with MSYS2 the mathematics functions are part of the main C library and `-lm` is accepted but not needed. Every build command in this book includes it, so that one command works everywhere.

Listing 5.6 is the Python version. Python's `math` module has `floor`, `ceil`, `sqrt` and `pow` [6], but its built-in `round` is different from C's: it rounds a number exactly halfway between two whole numbers to the *even* one, so `round(2.5)` is 2 and `round(3.5)` is 4. This rule, called *banker's rounding*, avoids a small upward bias when many halves are rounded [7]. C's `round` always rounds halves away from zero. To print exactly the same table, the Python program defines `c_round`, which does what C does.

Listing 5.6 `rounding.py`: the same table in Python, with a small function that rounds halves the way C does.

```
# rounding.py - borrowing ceil, floor and round from the math module
import math


def charged_kg(weight):
    kg = math.ceil(weight)                  # every started kilogram counts
    return 2 if kg < 2 else kg              # but never less than 2 kg


def c_round(x):
    """Round half away from zero, like C's round(); Python's round() does not."""
    return math.floor(x + 0.5) if x >= 0 else math.ceil(x - 0.5)


def main():
    weights = [0.8, 2.0, 2.1, 3.5, 4.0, 7.25]
    print("weight   floor   round    ceil  charged")
    for w in weights:
        print(f"{w:6.2f}  {math.floor(w):6.1f}  {c_round(w):6.1f}  "
              f"{math.ceil(w):6.1f}  {charged_kg(w):5d} kg")
    print(f"sqrt(2) = {math.sqrt(2.0):.6f}, pow(1.1, 12) = {math.pow(1.1, 12):.4f}")


if __name__ == "__main__":
    main()
```

5.5 Scope and Lifetime: Where Names Live

Every variable has two properties that beginners rarely think about until they cause a bug. Its **scope** is the part of the program text where its name can be used. Its **lifetime**, which the C standard calls *storage duration*, is the stretch of running time during which the variable exists and keeps its value [3]. **Table 5.2** summarizes the four kinds you need.

Table 5.2 Scope and lifetime of the four kinds of variables in C.

Kind	Declared	Visible (scope)	Exists (lifetime)
local	inside a function	from the declaration to the end of its block	from entering the block to leaving it
parameter	in the function header	the whole function body	during one call
global	outside every function	from the declaration to the end of the file	the whole run of the program
`static` local	inside a function, with `static`	like a local	the whole run of the program

Listing 5.7 shows all four at the counter of Profitina Laundry. The global `orders_today` is visible to every function, which is convenient and dangerous at once: any function can change it, so when its value is wrong, every function is a suspect. The local `fee` inside `take_order` is created at every call and destroyed at its end. The block in `main` declares a second `kg` that *hides* the outer one until the closing brace. And `next_receipt_number` keeps its counter in a `static` local variable: only this function can see it, but it survives from one call to the next, so the receipt numbers go 1, 2, 3.

Listing 5.7 `scope.c`: a global counter, locals, a block that hides a name, and a `static` receipt counter.

```c
/* scope.c - where a name is visible, and how long a variable lives */
#include <stdio.h>

int orders_today = 0;                   /* global: visible to every function */

int next_receipt_number(void)
{
    static int last = 0;                /* static: keeps its value between calls */
    last++;
    return last;
}

void take_order(int kg)
{
    int fee = kg * 7000;                /* local: born at the call, gone after */
    orders_today++;
    printf("receipt #%d: %d kg, Rp %d\n", next_receipt_number(), kg, fee);
}

int main(void)
{
    int kg = 3;
    take_order(kg);
    take_order(5);
    {
        int kg = 10;                    /* a new kg that hides the outer one */
        take_order(kg);
    }
    printf("kg in main is still %d\n", kg);
    printf("orders today: %d\n", orders_today);
    return 0;
}
```

```
receipt #1: 3 kg, Rp 21000
receipt #2: 5 kg, Rp 35000
receipt #3: 10 kg, Rp 70000
kg in main is still 3
orders today: 3
```

The `static` receipt counter is the best of both worlds and worth copying: the number must survive between calls, as a global would, but no other function has any business changing it, so it is hidden inside the one function that owns it. Globals, by contrast, are best kept for values that really belong to the whole program, and even then used sparingly.

Python draws the lines differently, as **Listing 5.8** shows. A variable assigned inside a function is local to that function unless the function declares it `global`. A variable at the top level of the file is global to the module. There is no block scope: the variable created inside an `if` or a loop is still visible after it, which is why the Python program uses a different name, `inner_kg`, to imitate the C block. And there are no `static` locals; the usual substitutes are a module variable with a leading underscore, as here, or an object (Chapter 14 introduces classes in C++, and Python has them too).

Listing 5.8 `scope.py`: the same program in Python, with `global` declarations and a module variable in the role of C's `static`.

```
# scope.py - where a name is visible, and how long a variable lives

orders_today = 0                        # global: visible to every function
_last_receipt = 0                       # plays the role of C's static variable


def next_receipt_number():
    global _last_receipt
    _last_receipt += 1
    return _last_receipt


def take_order(kg):
    global orders_today                 # we ASSIGN to it, so say so
    fee = kg * 7000                     # local: born at the call, gone after
    orders_today += 1
    print(f"receipt #{next_receipt_number()}: {kg} kg, Rp {fee}")


def main():
    kg = 3
    take_order(kg)
    take_order(5)
    inner_kg = 10                       # Python has no block scope
    take_order(inner_kg)
    print(f"kg in main is still {kg}")
    print(f"orders today: {orders_today}")


if __name__ == "__main__":
    main()
```

5.6 Randomness You Can Repeat

Mbak Rina wants a small promotion for Saturdays: every customer whose bill is at least Rp 50,000 rolls two dice, and a double wins a voucher for free ironing. Dimas will write the dice in the program, so that there is no argument over a die that fell off the counter.

The C library offers `rand()` in `<stdlib.h>`, which returns a *pseudo-random* whole number between 0 and `RAND_MAX`, and `srand(seed)`, which chooses where the sequence starts. The word *pseudo* matters: the numbers come from a formula, so the same seed always gives the same sequence. That is a feature. A program whose random numbers can be repeated can be tested, and a surprising result can be investigated.

There is one problem for a book that prints its outputs: the formula behind `rand` is not fixed by the standard, so the numbers differ between Windows, macOS and Linux, and Python's `random` module uses yet another generator, the Mersenne Twister [8]. To get identical numbers everywhere, we write our own generator, and it teaches two ideas of this chapter at once: a `static` variable that keeps its value between calls, and a function with a *hidden state*. The formula is the *linear congruential generator* that the C standard itself shows as an example implementation of `rand` [3], analyzed at length by Knuth [9]:

$$x_{k+1} = (1103515245\,x_k + 12345) \bmod 2^{32} \tag{5.2}$$

Each new state x_{k+1} is computed from the previous one x_k; `my_rand` returns bits 16 to 30 of the state, a number from 0 to 32767. Using the type `uint32_t` from `<stdint.h>`, an unsigned integer of exactly 32 bits, makes "mod 2^{32}" happen by itself, because unsigned arithmetic in C wraps around at the width of the type. **Listing 5.9** is the whole promotion.

Listing 5.9 `lucky_draw.c`: a repeatable random-number generator with hidden `static` state, dice made from it, and a fairness check.

```c
/* lucky_draw.c - a Saturday promotion with our own random numbers:
   customers whose bill is at least Rp 50.000 roll two dice,
   and a double wins a free ironing voucher. */
#include <stdio.h>
#include <stdint.h>

static uint32_t state = 1;              /* hidden state of the generator */

void my_srand(uint32_t seed)
{
    state = seed;
}

int my_rand(void)                       /* 0 .. 32767, the ISO C example */
{
    state = state * 1103515245u + 12345u;
    return (int) ((state >> 16) & 0x7FFF);
}

int roll_die(void)
{
    return 1 + my_rand() % 6;           /* scale 0..32767 down to 1..6 */
}

int main(void)
{
    int counts[7] = {0};
    my_srand(2026);
    int winners = 0;
    for (int customer = 1; customer <= 12; customer++) {
        int a = roll_die(), b = roll_die();
        counts[a]++;
        counts[b]++;
        printf("customer %2d rolls %d and %d%s\n", customer, a, b,
               a == b ? "  -> free ironing!" : "");
        if (a == b)
            winners++;
    }
    printf("winners: %d of 12\n", winners);

    for (int i = 0; i < 6000; i++)      /* a fairness check: 6000 more rolls */
        counts[roll_die()]++;
    for (int face = 1; face <= 6; face++)
        printf("face %d: %d\n", face, counts[face]);
    return 0;
}
```

`roll_die` turns a number from 0 to 32767 into a face from 1 to 6 with `1 + my_rand() % 6`: the remainder of a division by 6 is 0 to 5, and adding 1 shifts it to 1 to 6. Because 32768 is not a multiple of 6, faces 1 and 2 are very slightly more likely (5462 chances each out of 32768, against 5461 for the others), a difference far too small to matter for a voucher but important for a lottery or for cryptography, where much better generators are required.

The Python version in **Listing 5.10** must imitate the 32-bit wrap-around, because Python integers never overflow; `& 0xFFFFFFFF` keeps the lowest 32 bits, which is exactly "mod 2^{32}".

Listing 5.10 `lucky_draw.py`: the same generator in Python, with the 32-bit wrap-around made explicit.

```
# lucky_draw.py - a Saturday promotion with our own random numbers:
# customers whose bill is at least Rp 50.000 roll two dice,
# and a double wins a free ironing voucher.

_state = 1                              # hidden state of the generator


def my_srand(seed):
    global _state
    _state = seed & 0xFFFFFFFF


def my_rand():                          # 0 .. 32767, the ISO C example
    global _state
    _state = (_state * 1103515245 + 12345) & 0xFFFFFFFF   # 32-bit wrap-around
    return (_state >> 16) & 0x7FFF


def roll_die():
    return 1 + my_rand() % 6            # scale 0..32767 down to 1..6


def main():
    counts = [0] * 7
    my_srand(2026)
    winners = 0
    for customer in range(1, 13):
        a = roll_die()
        b = roll_die()
        counts[a] += 1
        counts[b] += 1
        prize = "  -> free ironing!" if a == b else ""
        print(f"customer {customer:2d} rolls {a} and {b}{prize}")
        if a == b:
            winners += 1
    print(f"winners: {winners} of 12")

    for _ in range(6000):               # a fairness check: 6000 more rolls
        counts[roll_die()] += 1
    for face in range(1, 7):
        print(f"face {face}: {counts[face]}")


if __name__ == "__main__":
    main()
```

Both programs print, on every computer:

```
customer  1 rolls 5 and 1
customer  2 rolls 2 and 3
customer  3 rolls 3 and 6
customer  4 rolls 3 and 2
customer  5 rolls 4 and 1
customer  6 rolls 1 and 3
customer  7 rolls 1 and 5
customer  8 rolls 4 and 3
customer  9 rolls 3 and 1
customer 10 rolls 3 and 3  -> free ironing!
customer 11 rolls 4 and 1
customer 12 rolls 4 and 4  -> free ironing!
winners: 2 of 12
face 1: 1014
face 2: 1020
face 3: 969
face 4: 988
face 5: 1020
face 6: 1013
```

Two customers out of twelve win, close to the one in six that a fair pair of dice promises, and the 6024 rolls in total land between 969 and 1020 times on each face, close to the expected 1004. For a real Saturday, `main` would call `my_srand((uint32_t) time(NULL))` instead of `my_srand(2026)`, so that each day starts at a different place; `time` from `<time.h>` returns the number of seconds since 1970, a different seed every second.

5.7 Recursion: Functions That Call Themselves

5.7.1 A base case and a smaller problem

A function may call any function, including itself. A function that does so is **recursive**. At first this sounds like a trick that could never stop, and a careless recursive function indeed never stops. A careful one has two parts.

> **DEFINITION.** A *recursive* function solves a problem by calling itself on a smaller instance of the same problem. It has a *base case*, an instance small enough to answer directly, and a *recursive step*, which reduces every other instance to a smaller one and combines the answer.

The factorial is the classic first example. The factorial of n, written $n!$, is the product $1 \times 2 \times \cdots \times n$, and $0! = 1$ by agreement. The product of the first n numbers is the product of the first $n-1$ numbers, times n:

$$n! = \begin{cases} 1 & \text{if } n \le 1, \\ n \cdot (n-1)! & \text{if } n > 1. \end{cases} \tag{5.3}$$

Equation (5.3) translates into C line by line, as the first function of **Listing 5.11** shows. Trace `factorial(4)` by hand. It is not the base case, so it must compute `4 * factorial(3)`, and it cannot finish the multiplication until `factorial(3)` has answered. `factorial(3)` in turn waits for `factorial(2)`, which waits for `factorial(1)`. That one is the base case and returns 1 at once. Now the waiting calls finish in reverse order: `factorial(2)` returns $2 \times 1 = 2$, `factorial(3)` returns $3 \times 2 = 6$, and `factorial(4)` returns $4 \times 6 = 24$.

Where do the waiting calls wait? Each call gets its own **frame** (also called an *activation record*): a small area of memory that holds its parameters and local variables and the place to return to. The frames sit on top of each other on the **call stack**; a call pushes a new frame, a return removes the top one. **Figure 5.2** shows the stack at the deepest point of `factorial(4)`. Four frames of the

same function exist at the same time, each with its own `n`, and this is why recursion works at all: the parameter `n` of one call is a different variable from the `n` of the next.

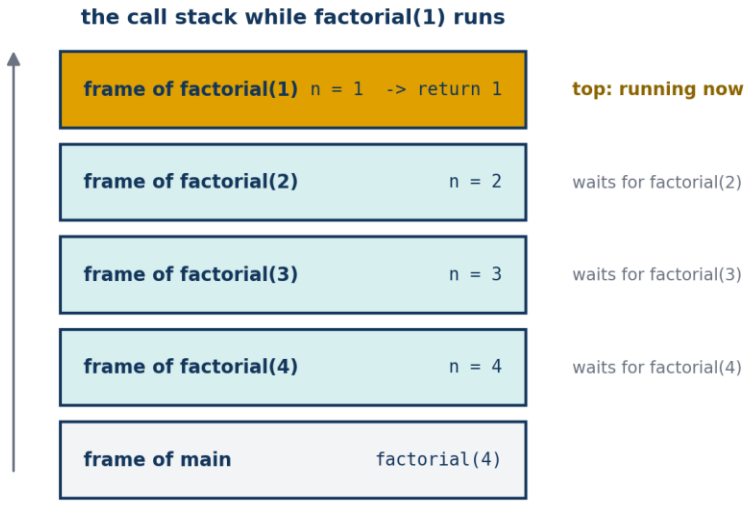


Figure 5.2 The call stack at the deepest point of `factorial(4)`: each call has its own frame with its own `n`, and every frame below the top waits for the answer of the frame above it.

The stack is large but not endless: a few megabytes on a typical desktop. A recursive function without a correct base case calls itself until the stack is full, and the program crashes with a *stack overflow* (on Linux the message is usually `Segmentation fault`). Python protects itself with a limit on the depth of recursion, 1000 by default, and stops with `RecursionError` [10].

5.7.2 A useful recursion: rupiah with dots

Indonesians write one million two hundred thirty-four thousand five hundred sixty-seven rupiah as Rp 1,234,567, with a dot between each group of three digits. `printf` cannot do this by itself, but recursion does it in six lines. The idea: to print 1234567, first print everything except the last three digits (that is 1234, a smaller problem of the same kind), then print a dot and the last three digits, 567, always with three digits, so that 5007 becomes 5.007 and not 5.7. The base case is a number below 1000, which is printed as it is. This is `print_thousands` in **Listing 5.11**. Notice where the recursive call stands: *before* the `printf`. The deepest call prints first, which is exactly the order in which the digit groups must appear.

5.7.3 The price of recursion: Fibonacci

The Fibonacci numbers start with 0 and 1, and every next number is the sum of the two before it: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … The definition is itself recursive,

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \ \text{ for } n \geq 2, \tag{5.4}$$

and Equation (5.4) translates into the three-line function `fib` of **Listing 5.11**. It is correct, and it is a lesson in cost. **Figure 5.3** draws every call made by `fib(5)`. The root calls `fib(4)` and `fib(3)`; `fib(4)` calls `fib(3)` again, and the whole subtree below it is computed twice. The amber boxes are calls whose answer had already been computed somewhere else in the tree.

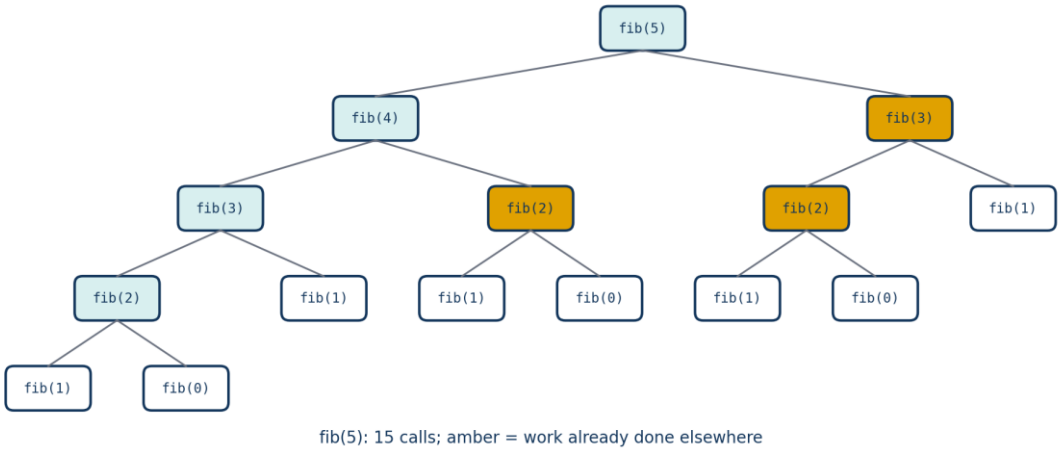


Figure 5.3 Every call made by `fib(5)`: fifteen calls to compute one number, and the amber calls repeat work already done elsewhere in the tree.

Let $C(n)$ be the number of calls that `fib(n)` makes, counting itself. For $n < 2$ it is one call. Otherwise it is one call plus all the calls of the two recursive calls:

$$C(n) = C(n-1) + C(n-2) + 1, \qquad C(0) = C(1) = 1. \tag{5.5}$$

From Equation (5.5): $C(2) = 3, C(3) = 5, C(4) = 9, C(5) = 15$, which matches the fifteen boxes of **Figure 5.3**. Adding 1 to both sides shows that $C(n) + 1$ obeys the Fibonacci rule itself, and working out the first values gives $C(n) = 2F(n+1) - 1$. The number of calls therefore grows as fast as the Fibonacci numbers, which grow by a factor of about 1.618 at every step [11]. Each increase of n by 5 multiplies the work by about $1.618^5 \approx 11$. The program counts the calls in a global variable to show it.

Listing 5.11 `recursion.c`: factorial, rupiah with dots, and Fibonacci with a counter of calls.

```c
/* recursion.c - functions that call themselves */
#include <stdio.h>

long factorial(int n)
{
    if (n <= 1)                         /* base case: stop here */
        return 1;
    return n * factorial(n - 1);        /* recursive step: a smaller problem */
}

void print_thousands(long amount)       /* 1234567 -> 1.234.567 */
{
    if (amount < 1000) {
        printf("%ld", amount);
        return;
    }
    print_thousands(amount / 1000);     /* first everything but the last 3 */
    printf(".%03ld", amount % 1000);    /* then the last three digits */
}

long calls = 0;                         /* how many times fib was entered */

long fib(int n)
{
    calls++;
    if (n < 2)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main(void)
{
    for (int n = 0; n <= 5; n++)
        printf("%d! = %ld\n", n, factorial(n));

    long amounts[] = {7, 28000, 187500, 1234567};
    for (int i = 0; i < 4; i++) {
        printf("Rp ");
        print_thousands(amounts[i]);
        printf("\n");
    }
```

Listing 5.11 (Continued) *recursion.c: factorial, rupiah with dots, and Fibonacci with a counter of calls.*

```c
    int ns[] = {5, 10, 20, 25, 30};
    for (int i = 0; i < 5; i++) {
        calls = 0;
        long f = fib(ns[i]);
        printf("fib(%2d) = %6ld after %8ld calls\n", ns[i], f, calls);
    }
    return 0;
}
```

Listing 5.12 is the Python version. `thousands` returns a string instead of printing, because joining strings is easy in Python; the recursion is the same.

Listing 5.12 `recursion.py`: the same three recursions in Python.

```python
# recursion.py - functions that call themselves


def factorial(n):
    if n <= 1:                          # base case: stop here
        return 1
    return n * factorial(n - 1)         # recursive step: a smaller problem


def thousands(amount):                  # 1234567 -> "1.234.567"
    if amount < 1000:
        return str(amount)
    return thousands(amount // 1000) + f".{amount % 1000:03d}"


calls = 0                               # how many times fib was entered


def fib(n):
    global calls
    calls += 1
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)


def main():
    global calls
    for n in range(6):
        print(f"{n}! = {factorial(n)}")

    for amount in [7, 28000, 187500, 1234567]:
        print("Rp " + thousands(amount))

    for n in [5, 10, 20, 25, 30]:
        calls = 0
        f = fib(n)
        print(f"fib({n:2d}) = {f:6d} after {calls:8d} calls")


if __name__ == "__main__":
    main()
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
Rp 7
Rp 28.000
Rp 187.500
Rp 1.234.567
fib( 5) =      5 after       15 calls
fib(10) =     55 after      177 calls
fib(20) =   6765 after    21891 calls
fib(25) =  75025 after   242785 calls
fib(30) = 832040 after  2692537 calls
```

The last five lines are the lesson. `fib(30)` makes 2,692,537 calls to compute 832040, and each line is about eleven times the one before, as Equation (5.5) predicted. A simple loop that keeps the last two numbers computes `fib(30)` in 30 steps. Chapter 8 measures the two ways against each other with a stopwatch.

5.7.4 Recursion or a loop?

Every recursive function can be rewritten with a loop, and every loop as a recursion. Choose by clarity and by cost. The factorial reads well either way, and the loop saves the stack. The rupiah printer is clearer recursive, because the problem itself is "the rest, then the last group". Fibonacci written as a direct copy of its definition is a trap: clear, but exponentially slow. Problems whose data are themselves nested, such as the folders on a disk or the Towers of Hanoi in Exercise 5.6, are where recursion is at its best, because the loop version would have to manage its own stack by hand. Abelson and Sussman call the Fibonacci pattern *tree recursion* and use it for the same warning: a process that branches into two at every step repeats most of its work [12].

5.8 Building the Cashier, Version 4

5.8.1 One file of rules, one file of story

Now we rebuild Mbak Rina's cashier with everything above. The business rules move into their own source file, `laundry.c`, and their prototypes into a **header file**, `laundry.h`. Any program that needs the rules includes the header: the cashier, the day report of Chapter 6, the closing summary of Chapter 7 and the test program of this chapter. **Listing 5.13** is the header. Its first two lines and its last line are an *include guard*: if a file is accidentally included twice, the second copy is skipped, because `LAUNDRY_H` is already defined.

Listing 5.13 `laundry.h`: the promises of the business-rules module, with its constants and an include guard.

```
/* laundry.h - Profitina Laundry's business rules: the promises (Chapter 5) */
#ifndef LAUNDRY_H
#define LAUNDRY_H

#define MIN_KG 2
#define MAX_KG 50
#define EXPRESS_PERCENT 50

int price_per_kg(int service);          /* rupiah per kg; 0 = unknown service */
void print_service(int service);        /* name, padded to 12 characters */
int charged_kg(double weight);          /* started kilograms, at least MIN_KG */
long bill_for(int service, double weight, int express);
void print_rupiah(long amount);         /* 1234567 -> Rp 1.234.567 */
int next_receipt_number(void);          /* 1, 2, 3, ... */

#endif
```

The constants are written with `#define`. The **preprocessor**, which runs before the compiler, replaces every `MIN_KG` in the program by `2`. The price of express service, the minimum and the largest believable bag each have one home and one name.

Listing 5.14 keeps the promises. Each function does one thing. `price_per_kg` knows the price list, and the value 0 means "unknown service". `print_service` knows the names. `charged_kg` applies Equation (5.1). `bill_for` combines the price and the charged kilograms and adds the express fee. `print_rupiah` prints amounts with dots, using the recursion of Section 5.7.2, whose helper is marked `static` at file level: in front of a function, `static` means *private to this file*, so no other file can call `print_thousands` or collide with its name. `next_receipt_number` is the `static` counter of Section 5.5.

Listing 5.14 `laundry.c`: Profitina Laundry's business rules, one function per rule.

```c
/* laundry.c - Profitina Laundry's business rules: one function per rule */
#include <stdio.h>
#include <math.h>
#include "laundry.h"

int price_per_kg(int service)
{
    switch (service) {
    case 1: return 7000;                /* wash & fold */
    case 2: return 10000;               /* wash & iron */
    case 3: return 6000;                /* iron only */
    default: return 0;
    }
}

void print_service(int service)
{
    switch (service) {
    case 1: printf("%-12s", "Wash & fold"); break;
    case 2: printf("%-12s", "Wash & iron"); break;
    case 3: printf("%-12s", "Iron only"); break;
    default: printf("%-12s", "Unknown"); break;
    }
}

int charged_kg(double weight)
{
    int kg = (int) ceil(weight);
    return kg < MIN_KG ? MIN_KG : kg;
}

long bill_for(int service, double weight, int express)
{
    long bill = (long) price_per_kg(service) * charged_kg(weight);
    if (express)
        bill += bill * EXPRESS_PERCENT / 100;
    return bill;
}
```

Listing 5.14 (Continued) *laundry.c: Profitina Laundry's business rules, one function per rule.*

```c
static void print_thousands(long amount)   /* static: private to this file */
{
    if (amount < 1000) {
        printf("%ld", amount);
        return;
    }
    print_thousands(amount / 1000);
    printf(".%03ld", amount % 1000);
}

void print_rupiah(long amount)
{
    printf("Rp ");
    print_thousands(amount);
}

int next_receipt_number(void)
{
    static int last = 0;
    return ++last;
}
```

With the rules in place, the cashier itself in **Listing 5.15** is short enough to read as a story: while there is an order, read it, check it, compute the bill, print the receipt, and add to the day's totals.

Two helper functions live in this file because they belong to the counter and not to the business: `order_is_valid` explains why an order is refused, and `print_receipt` formats one line.

Listing 5.15 `cashier.c`: the cashier, version 4, whose `main` tells the story and leaves the work to named functions.

```
/* cashier.c - Profitina Laundry cashier, version 4: main tells the story,
   the functions in laundry.c do the work.
   Input: one order per line, "service weight express", ending with 0. */
#include <stdio.h>
#include "laundry.h"

int order_is_valid(int service, double weight)
{
    if (price_per_kg(service) == 0) {
        printf("skipped: unknown service %d\n", service);
        return 0;
    }
    if (weight <= 0 || weight > MAX_KG) {
        printf("skipped: %.1f kg is not a laundry bag\n", weight);
        return 0;
    }
    return 1;
}

void print_receipt(int service, double weight, int express, long bill)
{
    printf("#%03d  ", next_receipt_number());
    print_service(service);
    printf(" %5.1f kg -> %2d kg", weight, charged_kg(weight));
    printf(express ? "  express  " : "           ");
    print_rupiah(bill);
    printf("\n");
}

int main(void)
{
    int service, express, orders = 0;
    double weight;
    long revenue = 0;

    printf("PROFITINA LAUNDRY - cashier v4\n");
    while (scanf("%d", &service) == 1 && service != 0) {
        if (scanf("%lf %d", &weight, &express) != 2)
            break;
        if (!order_is_valid(service, weight))
            continue;
        long bill = bill_for(service, weight, express);
        print_receipt(service, weight, express, bill);
        orders++;
        revenue += bill;
    }
```

Listing 5.15 (Continued) *cashier.c: the cashier, version 4, whose main tells the story and leaves the work to named functions.*

```
    printf("orders: %d, revenue: ", orders);
    print_rupiah(revenue);
    printf("\n");
    return 0;
}
```

The loop condition `scanf("%d", &service) == 1 && service != 0` stops at the end of the input and at the service code 0, which the counter uses to close the day. The `continue` in the loop skips the rest of the body for an invalid order, and the next order is read.

The Python module **Listing 5.16** holds the same rules. The price list and the names become *dictionaries*, tables that map a key to a value, and `.get(service, 0)` answers 0 for an unknown key, the same convention as in C. Where the C functions print, the Python functions return strings, and the caller prints them; either design works, and Chapter 9 shows how C builds strings too.

Listing 5.16 `laundry.py`: the business rules as a Python module.

```
# laundry.py - Profitina Laundry's business rules: one function per rule
import math

MIN_KG = 2
MAX_KG = 50
EXPRESS_PERCENT = 50

_PRICES = {1: 7000, 2: 10000, 3: 6000}   # wash & fold, wash & iron, iron only
_NAMES = {1: "Wash & fold", 2: "Wash & iron", 3: "Iron only"}
_last_receipt = 0


def price_per_kg(service):
    return _PRICES.get(service, 0)       # 0 = unknown service


def service_name(service):
    return f"{_NAMES.get(service, 'Unknown'):<12}"


def charged_kg(weight):
    kg = math.ceil(weight)
    return MIN_KG if kg < MIN_KG else kg


def bill_for(service, weight, express):
    bill = price_per_kg(service) * charged_kg(weight)
    if express:
        bill += bill * EXPRESS_PERCENT // 100
    return bill


def _thousands(amount):                  # leading _: private to this module
    if amount < 1000:
        return str(amount)
    return _thousands(amount // 1000) + f".{amount % 1000:03d}"


def rupiah(amount):
    return "Rp " + _thousands(amount)


def next_receipt_number():
    global _last_receipt
    _last_receipt += 1
    return _last_receipt
```

Listing 5.17 is the Python cashier. It reads all of the input at once with `sys.stdin.read().split()`, which gives a list of the words, and walks through the list three words at a time.

Listing 5.17 `cashier.py`: the cashier, version 4, in Python, importing the rules from `laundry.py`.

```python
# cashier.py - Profitina Laundry cashier, version 4: main tells the story,
# the functions in laundry.py do the work.
# Input: one order per line, "service weight express", ending with 0.
import sys
from laundry import (MAX_KG, price_per_kg, service_name, charged_kg,
                     bill_for, rupiah, next_receipt_number)


def order_is_valid(service, weight):
    if price_per_kg(service) == 0:
        print(f"skipped: unknown service {service}")
        return False
    if weight <= 0 or weight > MAX_KG:
        print(f"skipped: {weight:.1f} kg is not a laundry bag")
        return False
    return True


def print_receipt(service, weight, express, bill):
    tag = "  express  " if express else "           "
    print(f"#{next_receipt_number():03d}  {service_name(service)}"
          f" {weight:5.1f} kg -> {charged_kg(weight):2d} kg{tag}{rupiah(bill)}")


def main():
    words = sys.stdin.read().split()
    pos, orders, revenue = 0, 0, 0
    print("PROFITINA LAUNDRY - cashier v4")
    while pos < len(words) and int(words[pos]) != 0:
        if pos + 2 >= len(words):
            break
        service = int(words[pos])
        weight = float(words[pos + 1])
        express = int(words[pos + 2])
        pos += 3
        if not order_is_valid(service, weight):
            continue
        bill = bill_for(service, weight, express)
        print_receipt(service, weight, express, bill)
        orders += 1
        revenue += bill
    print(f"orders: {orders}, revenue: {rupiah(revenue)}")


if __name__ == "__main__":
    main()
```

5.8.2 Compiling several files

A program in several files is compiled by naming all of its `.c` files; the header is not named, because the `#include` lines bring it in:

```
gcc -std=c17 -Wall -Wextra -o cashier cashier.c laundry.c -lm
```

Behind this one command, `gcc` compiles each `.c` file separately into an *object file* and then the **linker** joins the object files and the libraries into one program, matching every call in `cashier.c` to its definition in `laundry.c`. If a function was promised in the header but never defined, the compiler is satisfied and the linker complains with `undefined reference to ...`; that message always means “a promise was not kept”. In VS Code, the build task of Chapter 1 compiles only the open file, so a program of several files is built from the integrated terminal (**Terminal > New Terminal**) with the command above.

For Python nothing changes: `python3 cashier.py` finds `laundry.py` in the same folder and imports it.

5.9 Running It

Here is a real morning at the counter, as Mbak Rina types it: one order per line with the service, the weight and 1 for express or 0 for normal, and 0 to close.

```
1 3.4 0
2 1.2 1
3 5 0
7 2 0
2 6.5 0
1 0 0
1 12.75 1
0
```

```
PROFITINA LAUNDRY - cashier v4
#001  Wash & fold    3.4 kg ->  4 kg           Rp 28.000
#002  Wash & iron    1.2 kg ->  2 kg  express  Rp 30.000
#003  Iron only      5.0 kg ->  5 kg           Rp 30.000
skipped: unknown service 7
#004  Wash & iron    6.5 kg ->  7 kg           Rp 70.000
skipped: 0.0 kg is not a laundry bag
#005  Wash & fold   12.8 kg -> 13 kg  express  Rp 136.500
orders: 5, revenue: Rp 294.500
```

Check the lines by hand, because that is the habit that would have caught the four days of wrong reports. The first bag weighs 3.4 kg, so four kilograms are charged at Rp 7,000: Rp 28,000. The second is 1.2 kg of wash and iron as express: the minimum of 2 kg at Rp 10,000 gives Rp 20,000, and half again for express makes Rp 30,000. Service 7 does not exist and is refused with its reason, and so is a bag of 0 kg, probably a typing slip. The last order, 12.75 kg of wash and fold as express, is printed as 12.8 kg because `%5.1f` shows one decimal; it is charged as 13 kg, Rp 91,000, plus half, Rp 136,500. Five valid orders make Rp 294,500.

5.10 Testing It

A test program calls each function with inputs whose answers were worked out by hand, and compares. **Listing 5.18** checks twelve such facts. The helper `check` counts the checks and reports only failures, with the line number of the failing check, which the predefined macro `__LINE__` supplies.

Listing 5.18 `test_laundry.c`: twelve checks of the business rules against hand-worked answers.

```c
/* test_laundry.c - checks every business rule against answers worked by hand */
#include <stdio.h>
#include "laundry.h"

static int checks = 0, failures = 0;

static void check(long got, long expected, int line)
{
    checks++;
    if (got != expected) {
        failures++;
        printf("FAILED on line %d: got %ld, expected %ld\n", line, got, expected);
    }
}

int main(void)
{
    check(price_per_kg(1), 7000, __LINE__);
    check(price_per_kg(2), 10000, __LINE__);
    check(price_per_kg(3), 6000, __LINE__);
    check(price_per_kg(9), 0, __LINE__);
    check(charged_kg(0.8), 2, __LINE__);      /* the minimum */
    check(charged_kg(2.0), 2, __LINE__);      /* exactly 2 stays 2 */
    check(charged_kg(2.1), 3, __LINE__);      /* a started kilogram */
    check(bill_for(1, 3.4, 0), 28000, __LINE__);
    check(bill_for(2, 1.2, 1), 30000, __LINE__);
    check(bill_for(3, 5.0, 1), 45000, __LINE__);
    check(next_receipt_number(), 1, __LINE__);
    check(next_receipt_number(), 2, __LINE__);
    printf("%d checks, %d failures\n", checks, failures);
    return failures != 0;
}
```

The edge cases are the valuable checks: 0.8 kg must become the minimum of 2 kg, exactly 2.0 kg must stay 2 (a careless rule such as “the whole kilograms plus one” would charge 3), and 2.1 kg must become 3. The receipt counter must start at 1 and count up. The program returns 1 to the operating system when a check fails, which lets a script notice failures automatically; `run_tests.py` in the chapter’s code folder compiles and runs every program of the chapter this way.

Listing 5.19 is the same test in Python, plus one check that the C version cannot do easily: that `rupiah(1234567)` is the text `Rp 1.234.567`.

Listing 5.19 `test_laundry.py`: the same checks in Python, with one more for the rupiah text.

```
# test_laundry.py - checks every business rule against answers worked by hand
import inspect
import sys
from laundry import (price_per_kg, charged_kg, bill_for, rupiah,
                     next_receipt_number)

checks = 0
failures = 0


def check(got, expected):
    global checks, failures
    checks += 1
    if got != expected:
        failures += 1
        line = inspect.stack()[1].lineno
        print(f"FAILED on line {line}: got {got}, expected {expected}")


def main():
    check(price_per_kg(1), 7000)
    check(price_per_kg(2), 10000)
    check(price_per_kg(3), 6000)
    check(price_per_kg(9), 0)
    check(charged_kg(0.8), 2)            # the minimum
    check(charged_kg(2.0), 2)            # exactly 2 stays 2
    check(charged_kg(2.1), 3)            # a started kilogram
    check(bill_for(1, 3.4, 0), 28000)
    check(bill_for(2, 1.2, 1), 30000)
    check(bill_for(3, 5.0, 1), 45000)
    check(next_receipt_number(), 1)
    check(next_receipt_number(), 2)
    if rupiah(1234567) != "Rp 1.234.567":  # Python can test text as well
        print("FAILED: rupiah(1234567)")
    print(f"{checks} checks, {failures} failures")
    return 1 if failures else 0


if __name__ == "__main__":
    sys.exit(main())
```

```
12 checks, 0 failures
```

Twelve checks and no failures. Write the test before you change a rule, run it after, and you will know within a second whether the change broke anything. Python has a full testing framework, `unittest`, in its standard library [13]; we use the small `check` function so that the C and Python tests look alike and print the same line.

5.11 Making It Better

The value of the split shows the first time Mbak Rina changes her mind. Suppose she adds a fourth service, *bed cover*, at Rp 15,000 per kilogram. In version 3 the price list, the names and the validity check were spread over `main`. Now exactly two functions change, `price_per_kg` and `print_service`, each by one `case` line, and the cashier, the check `order_is_valid` and every future report pick up the new service without a single edit, because they all ask `price_per_kg`. Add one line to the test program, `check(price_per_kg(4), 15000, __LINE__);`, and run it.

Suppose instead that the express fee drops from 50% to 40%. `EXPRESS_PERCENT` changes in the header, `bill_for` uses it, and the test `check(bill_for(2, 1.2, 1), 30000, ...)` now fails with `got 28000, expected 30000`. That failure is good news: the test has noticed the change, and you

update the expected answer consciously instead of discovering the new prices from an angry customer.

Two small habits make functions like these better still. First, keep the output out of the rules: `bill_for` computes and returns, and it never prints. A function that computes and prints at once cannot be reused where the printing is not wanted, such as in the day report of Chapter 6. Second, give every function a comment that states its promise, as `laundry.h` does, including what it returns for bad input (`0 = unknown service`). A reader of the header should never need to open the `.c` file to use a function correctly.

5.12 Using It for Real

The cashier is Mbak Rina's, but the shape fits almost any small business that charges by a rule. To adapt it, change the rules, not the story.

- **A tailor** charges by piece and by kind of work: hemming trousers Rp 15,000, taking in a shirt Rp 25,000, a new kebaya by quotation. `price_per_kg` becomes `price_of(job)`, and `charged_kg` disappears or becomes a rule for rush jobs.
- **A photocopy kiosk** near a campus charges per page, cheaper in black and white than in color, with a discount above 100 pages. The discount is one more function, `discount_for(pages)`, used by `bill_for`; the receipt printer does not change.
- **A motorcycle workshop** charges a service fee plus parts. The order line gets a fourth number, the price of the parts, and `bill_for` adds it; everything else stays.
- **A catering kitchen** sells boxes by the dozen: the minimum charge and the rounding up of `charged_kg` become "at least 10 boxes, in steps of 5", the same shape as Equation (5.1) with a different step.

In each case, write the rules first as a header of prototypes with a one-line promise each, write a test program with answers worked out on paper by the owner, and only then write `main`. The owner can read and check the test far more easily than the code.

5.13 Common Mistakes, and How to Avoid Them

- **Expecting a function to change its argument.** `add_express_fee(bill);` alone does nothing to `bill`; the result is returned and thrown away. Write `bill = add_express_fee(bill);`, or wait for the pointers of Chapter 7.
- **Forgetting the `return`.** A non-`void` function that reaches its closing brace without `return` gives back an undefined value. `gcc -Wall` warns: *control reaches end of non-void function*. Never ignore that warning.
- **Calling without a prototype.** The error *implicit declaration of function* means a call came before the definition and no prototype was given. Add the prototype or include the right header (for example `<math.h>` for `sqrt`).
- **Forgetting `-lm`.** *undefined reference to `ceil`* on Linux means the mathematics library was not linked. Add `-lm` at the end of the command.
- **Integer overflow inside a product.** `price * kg` with two `int` values is computed as `int`, whatever the type of the variable that receives it. Cast one factor, as in `(long) price * kg`.
- **A recursion without a reachable base case.** `factorial(-1)` with the test `n == 1` instead of `n <= 1` never reaches the base case and ends in a stack overflow. Make the base case catch every small or odd input.

- **Tree recursion for large inputs.** A recursion that calls itself twice on inputs only a little smaller, like `fib`, repeats exponential amounts of work. Count the calls, as in **Listing 5.11**, before you trust it with $n = 50$.
- **Python's `round` is not C's.** `round(2.5)` is 2 in Python and `round(2.5)` is 3.0 in C. When money is rounded, write down which rule you mean and use it on purpose.
- **A local that hides a global.** Declaring `int orders_today = 0;` inside a function creates a new local that hides the global; the global never changes. `gcc -Wshadow` warns about names that hide other names.

5.14 Chapter Summary

- A function is a named piece of work with parameters and a return value. Define it once, call it wherever it is needed; the caller needs only its header.
- Arguments are copied into parameters (pass-by-value): a function cannot change its caller's variables through its parameters, and gives its answer through `return`.
- A prototype is a promise of a function's name, parameter types and return type, placed before the first call; headers are files of prototypes, and `#include` brings them in. Python needs no prototypes, only that a name exists when it is called.
- The standard library provides tested functions by subject: `<math.h>` (with `-lm` on Linux) for `ceil`, `floor`, `round`, `sqrt`, `pow`; `<stdlib.h>`, `<string.h>`, `<time.h>` and others. The laundry charges max(2, $\lceil w \rceil$) kilograms, Equation (5.1).
- Scope is where a name is visible, lifetime is how long a variable exists: locals and parameters live during one call, globals and `static` locals for the whole run. Prefer locals, use `static` for private memory, keep globals rare.
- A pseudo-random generator is a function with hidden state; with our own formula, Equation (5.2), the numbers repeat on every computer and in both languages.
- A recursive function has a base case and a recursive step; each active call has its own frame on the call stack. Recursion is natural for nested problems, and dangerous when it repeats work, as `fib` does with $2F(n+1) - 1$ calls.
- A program of several files keeps rules in a module (`laundry.h`, `laundry.c`), the story in `main`, and a test program that checks each rule against hand-worked answers.

5.15 Exercises

1. **Tracing.** Without running it, write what this program prints, then check by running it. Explain each line with the words *copy* and *return*. `int twice(int x) { x = x * 2; return x; }` and in `main`: `int a = 5; int b = twice(a); printf("%d %d\n", a, b); twice(b); printf("%d\n", b);`
2. **A function of your own.** Write `long round_to_500(long amount)` that rounds an amount to the nearest multiple of Rp 500, halves upward (Rp 28,250 becomes Rp 28,500, Rp 28,249 becomes Rp 28,000), using only whole-number arithmetic. Give five test values with their expected results.
3. **Scope detective.** A program has a global `int total = 0;` and a function `void add(int x) { int total = 0; total += x; }`. After `add(5); add(7);`, what is the global `total`, and why? How would you fix `add`, and what warning option of `gcc` would have pointed at the problem?
4. **Euclid, recursively.** The greatest common divisor of two numbers satisfies $\gcd(a, 0) = a$ and $\gcd(a, b) = \gcd(b, a \bmod b)$. Write it as a recursive function that prints each call, and use it to

find how often two pickup rounds that come every 4 and every 6 days fall on the same day (their least common multiple, $a \cdot b/\gcd(a, b)$).

5. **Digit sums.** Write the sum of the decimal digits of a number twice, recursively and with a loop, and a *check digit* that repeats the digit sum until one digit is left (2026 → 10 → 1). Print both sums and the check digit for 0, 7, 2026 and 987654321.
6. **The Towers of Hanoi.** Three laundry baskets of different sizes are stacked on shelf A, the biggest at the bottom. Move them to shelf C, one basket at a time, never putting a bigger basket on a smaller one, using shelf B for help. Write a recursive function that prints the moves for n baskets and counts them. How many moves are needed for n baskets?
7. **Counting calls.** Using Equation (5.5), compute $C(10)$ by hand and check it against the output of **Listing 5.11**. Then estimate how many calls `fib(40)` makes, and explain why a loop needs only 40 steps.

5.16 Solutions to the Exercises

1. The program prints `5 10` and then `10`. In `twice(a)` the parameter `x` receives a *copy* of 5, doubles its copy and returns 10, which `main` stores in `b`; `a` is untouched. The call `twice(b)` doubles a copy of 10 and *returns* 20, but nobody stores the returned value, so it is lost and `b` is still 10.

2. Add 250 and cut to a multiple of 500 with whole-number division: `return (amount + 250) / 500 * 500;`. Tests: 28250 → 28500, 28249 → 28000, 28000 → 28000, 0 → 0, 136499 → 136500. The first two are the edges of the rule and the most valuable; the half goes upward because 28250 + 250 = 28500 is exactly a multiple.

3. The global `total` is still 0. Inside `add`, the declaration `int total = 0;` creates a new local variable that hides the global for the whole function; the addition changes the local, which disappears at the end of each call. The fix is to delete the local declaration, so that `total += x;` refers to the global. `gcc -Wshadow` reports that the local declaration shadows a global one.

4. **Listing 5.20** and **Listing 5.21** follow the two equations directly; each call strictly decreases the second argument, so the base case $b = 0$ is always reached.

Listing 5.20 `ex5_gcd.c`: Euclid's algorithm, recursively, and the least common multiple built from it.

```c
/* ex5_gcd.c - Exercise 5.4: Euclid's greatest common divisor, recursively */
#include <stdio.h>

int gcd(int a, int b)
{
    printf("  gcd(%d, %d)\n", a, b);
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

int main(void)
{
    int g = gcd(84, 36);
    printf("gcd = %d\n", g);
    int a = 4, b = 6;                       /* pickup rounds every 4 and 6 days */
    int lcm = a / gcd(a, b) * b;
    printf("both rounds meet every %d days\n", lcm);
    return 0;
}
```

Listing 5.21 `ex5_gcd.py`: the same in Python.

```python
# ex5_gcd.py - Exercise 5.4: Euclid's greatest common divisor, recursively


def gcd(a, b):
    print(f"  gcd({a}, {b})")
    if b == 0:
        return a
    return gcd(b, a % b)


def main():
    g = gcd(84, 36)
    print(f"gcd = {g}")
    a, b = 4, 6                             # pickup rounds every 4 and 6 days
    lcm = a // gcd(a, b) * b
    print(f"both rounds meet every {lcm} days")


if __name__ == "__main__":
    main()
```

```
  gcd(84, 36)
  gcd(36, 12)
  gcd(12, 0)
gcd = 12
  gcd(4, 6)
  gcd(6, 4)
  gcd(4, 2)
  gcd(2, 0)
both rounds meet every 12 days
```

The least common multiple is computed as `a / gcd(a, b) * b` rather than `a * b / gcd(a, b)`: dividing first keeps the intermediate number small, which avoids overflow for large inputs. The two rounds meet every 12 days.

5. Listing 5.22 and **Listing 5.23**. The recursion splits off the last digit with `% 10` and recurses on the rest with `/ 10`; the base case is a number with one digit. The check digit is itself recursive: it applies the digit sum until the result has one digit.

Listing 5.22 `ex5_digits.c`: the digit sum, recursively and with a loop, and a check digit.

```c
/* ex5_digits.c - Exercise 5.5: digit sum, recursively and with a loop */
#include <stdio.h>

int digit_sum_rec(long n)
{
    if (n < 10)
        return (int) n;
    return (int) (n % 10) + digit_sum_rec(n / 10);
}

int digit_sum_loop(long n)
{
    int sum = 0;
    while (n > 0) {
        sum += (int) (n % 10);
        n /= 10;
    }
    return sum;
}

int check_digit(long receipt)            /* repeat until one digit is left */
{
    int s = digit_sum_rec(receipt);
    return s < 10 ? s : check_digit(s);
}

int main(void)
{
    long numbers[] = {0, 7, 2026, 987654321};
    for (int i = 0; i < 4; i++)
        printf("%9ld: %2d %2d  check digit %d\n", numbers[i],
               digit_sum_rec(numbers[i]), digit_sum_loop(numbers[i]),
               check_digit(numbers[i]));
    return 0;
}
```

Listing 5.23 `ex5_digits.py`: the same in Python.

```python
# ex5_digits.py - Exercise 5.5: digit sum, recursively and with a loop


def digit_sum_rec(n):
    if n < 10:
        return n
    return n % 10 + digit_sum_rec(n // 10)


def digit_sum_loop(n):
    total = 0
    while n > 0:
        total += n % 10
        n //= 10
    return total


def check_digit(receipt):                # repeat until one digit is left
    s = digit_sum_rec(receipt)
    return s if s < 10 else check_digit(s)


def main():
    for n in [0, 7, 2026, 987654321]:
        print(f"{n:9d}: {digit_sum_rec(n):2d} {digit_sum_loop(n):2d}"
              f"  check digit {check_digit(n)}")


if __name__ == "__main__":
    main()
```

```
        0:  0  0  check digit 0
        7:  7  7  check digit 7
     2026: 10 10  check digit 1
987654321: 45 45  check digit 9
```

6. To move n baskets from A to C, move the top $n-1$ from A to B (using C for help), move the biggest from A to C, and move the $n-1$ from B to C (using A for help). Moving zero baskets is the base case. **Listing 5.24** and **Listing 5.25** print the moves for three baskets.

Listing 5.24 `ex5_hanoi.c`: the Towers of Hanoi, the classic problem that is easy recursively and hard with a loop.

```c
/* ex5_hanoi.c - Exercise 5.6: the Towers of Hanoi with laundry baskets */
#include <stdio.h>

int moves = 0;

void hanoi(int n, char from, char to, char spare)
{
    if (n == 0)
        return;
    hanoi(n - 1, from, spare, to);       /* clear the way */
    moves++;
    printf("move basket %d from %c to %c\n", n, from, to);
    hanoi(n - 1, spare, to, from);       /* put the rest back on top */
}

int main(void)
{
    hanoi(3, 'A', 'C', 'B');
    printf("%d moves\n", moves);
    return 0;
}
```

Listing 5.25 `ex5_hanoi.py`: the same in Python.

```python
# ex5_hanoi.py - Exercise 5.6: the Towers of Hanoi with laundry baskets

moves = 0


def hanoi(n, source, target, spare):
    global moves
    if n == 0:
        return
    hanoi(n - 1, source, spare, target)  # clear the way
    moves += 1
    print(f"move basket {n} from {source} to {target}")
    hanoi(n - 1, spare, target, source)  # put the rest back on top


def main():
    hanoi(3, "A", "C", "B")
    print(f"{moves} moves")


if __name__ == "__main__":
    main()
```

```
move basket 1 from A to C
move basket 2 from A to B
move basket 1 from C to B
move basket 3 from A to C
move basket 1 from B to A
move basket 2 from B to C
move basket 1 from A to C
7 moves
```

If $M(n)$ is the number of moves, $M(0) = 0$ and $M(n) = 2M(n-1) + 1$, which gives $M(n) = 2^n - 1$: 7 moves for 3 baskets, 1023 for 10.

7. From Equation (5.5): $C(2) = 3$, $C(3) = 5$, $C(4) = 9$, $C(5) = 15$, $C(6) = 25$, $C(7) = 41$, $C(8) = 67$, $C(9) = 109$, $C(10) = 177$, which is the number the program prints for `fib(10)`. With $C(n) = 2F(n+1) - 1$ and $F(41) = 165580141$, `fib(40)` makes 331,160,281 calls. A loop keeps only the last two numbers and adds them once per step, so it needs 40 additions: each Fibonacci number is computed once instead of being recomputed in every branch of the tree.

References

[1] N. Wirth, “Program development by stepwise refinement,” *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, Apr. 1971.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988.

[3] ISO/IEC 9899:2018, *Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, 2018.

[4] Python Software Foundation, “Defining functions,” in *The Python Tutorial*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/tutorial/controlflow.html#defining-functions

[5] Free Software Foundation, “Options for linking,” in *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

[6] Python Software Foundation, “math — Mathematical functions,” Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/math.html

[7] Python Software Foundation, “Built-in functions: round,” Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/functions.html#round

[8] Python Software Foundation, “random — Generate pseudo-random numbers,” Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/random.html

[9] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[10] Python Software Foundation, “sys.getrecursionlimit,” Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/sys.html#sys.getrecursionlimit

[11] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.

[12] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

[13] Python Software Foundation, “unittest — Unit testing framework,” Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/unittest.html

Chapter 6

Arrays: Many Values, One Name

The cashier of Chapter 5 prints a correct receipt and forgets it the moment the next customer steps up. That is fine for a receipt and useless for everything the owner asks at closing time: which order was the biggest, what the typical bill was, how many customers wanted ironing, which day of the week is the busiest. To answer, a program must remember many values of the same kind, and giving each one its own variable name does not scale past a handful. This chapter introduces the **array**, one name for a whole row of values, and the three things every programmer does with rows: count, sort and search. By the end of the chapter you will be able to declare, initialize and walk through arrays in C and lists in Python; explain why indexes start at 0 and what happens when a program steps outside an array; use an array as a set of counters; pass arrays to functions safely with a size and `const`; sort with bubble sort and insertion sort and count their work; search with linear and binary search and say which one to use when; store a table in a two-dimensional array and compute the address of any cell; and build version 5 of the Profitina Laundry cashier, which remembers the whole day and prints a report with totals per service, the average, the median and the three biggest bills.

6.1 The Notebook at the End of the Day

When Profitina Laundry closes at nine in the evening, Mbak Rina sits down with a cup of tea and the stack of carbon copies of the day's receipts. She wants to know four things. How much money came in, and how much was the average bill? Which order was the biggest, so that she can thank that customer next time? How many orders were wash-and-fold, wash-and-iron and iron-only, so that she knows whether to buy more detergent or more ironing starch? And, looking back over the week, which day is the busiest, so that her part-time helper comes on the right days?

With paper she answers the first question by adding the receipts twice, and the rest by sorting the receipts into piles on the table. Dimas wants the cashier to do it. The difficulty is that the program of Chapter 5 kept only running totals. A total is enough to compute a sum and an average, but not a median, not the three biggest bills and not a sorted list, because those need every single bill at the end of the day, not only their sum.

Declaring `long bill1, bill2, bill3, ... bill100;` is not a solution. A loop cannot say "the next variable", because `bill1` and `bill2` are unrelated names to the compiler, and a hundred almost identical `if` statements would be needed to store the fifty-seventh order in `bill57`. What Dimas needs is one name for the whole stack of receipts, and a number that says which receipt he means. That is exactly what the pile of carbon copies on the table already is: one stack, and the receipts inside it numbered from the top.

The same need appears wherever there is a list of things of the same kind: the daily sales of a warung over a month, the rainfall of each day at a fruit and durian orchard, the marks of a class, the kilograms of rice in each sack of a delivery. In each case the owner asks the same questions: total, average, biggest, how many of each kind, in order please, and where is this one.

6.2 The Idea in Plain Words

6.2.1 An array is a row of numbered boxes

> **DEFINITION.** An *array* is a fixed number of variables of the same type, stored side by side in memory under one name. Each variable is an *element* of the array, and it is selected by an *index*, a whole number written in square brackets: `bill[0]`, `bill[1]`, and so on. In C the first index is 0, and an array of n elements has the indexes 0 to $n - 1$.

The declaration `long bill[8];` asks for eight `long` variables in a row. **Figure 6.1** draws them. The name `bill` stands for the whole row; `bill[4]` is the fifth box, which here holds 136500. Everything you can do with an ordinary `long` variable, you can do with an element: read it, assign it, pass it to `printf`, use it in a condition.

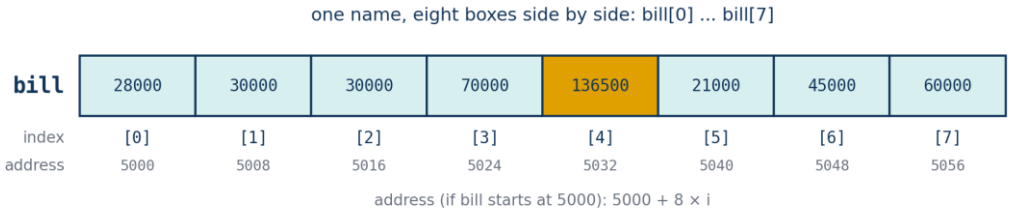


Figure 6.1 The array `bill` with eight elements: one name, indexes 0 to 7, and the elements side by side in memory, eight bytes apart on a computer where a `long` takes eight bytes.

The power of the array is that the index can be a variable. `bill[i]` means "the box whose number is currently in `i`", so a single loop in which `i` runs from 0 to 7 visits every box. That is the whole difference between eight variables and one array: the eight variables have eight names, the array has one name and a number that a loop can count.

6.2.2 Why the index starts at 0

The boxes of an array sit next to each other. If the array starts at memory address a, and each element takes s bytes, then element number i starts at

$$\text{address}(\text{bill}[i]) = a + i \cdot s \tag{6.1}$$

This is why C counts from 0: the index is the *distance* from the start of the array, measured in elements. The first element is at distance 0 from the start, and Equation (6.1) needs no correction. With $a = 5000$ and $s = 8$, as in **Figure 6.1**, `bill[4]` is at $5000 + 4 \cdot 8 = 5032$. The computer finds any element with one multiplication and one addition, however long the array is; this is why reading `bill[4]` is as fast as reading `bill[0]`, and why arrays are the fastest structure there is for "give me element number i". (Real addresses are much larger numbers, and on 64-bit Windows a `long` takes four bytes, not eight; the formula is the same.)

The same formula explains why arrays are fixed in size. The elements must lie side by side, so the space for all of them is reserved at once, when the array is created. There is no room to add a ninth box after `bill[7]`, because the bytes after it may already belong to another variable. A program that needs to grow must reserve more room than it expects to use, as the cashier of Section 6.9 does, or ask for a new, bigger block and copy the elements over, which is what a Python list does quietly behind `append`.

6.2.3 Nobody checks the fence

C trusts the programmer. When a program writes `bill[8]` in an array of eight elements, the compiler does not stop it and the running program does not check it: Equation (6.1) simply computes the address just past the end, and the program reads or writes whatever happens to live there, perhaps another variable, perhaps nothing it is allowed to touch. The C standard calls this *undefined behavior* [1]: anything may happen, including the program appearing to work until the day it does not. The SEI CERT C Coding Standard lists it as one of its most important rules, ARR30-C, because out-of-bounds writes have caused countless security holes [2].

The defense is a habit, not a tool: every loop over an array runs `for (i = 0; i < n; i++)` with `<` and never `<=`, every index computed from input is checked before it is used, and the size of the array is a named constant. The build flags `-fsanitize=address,undefined`, which every program of this book passes, make `gcc` insert checks that stop the program with a clear report at the first out-of-bounds access; use them while you develop [3].

6.2.4 Python's list

Python has no fixed-size array in the core language; its **list** plays the part. A list is written in square brackets, `bill = [28000, 30000, 30000]`, and indexed exactly like a C array, from 0. Three differences matter. A list knows its length, `len(bill)`, so no separate count is needed. A list can grow and shrink, with `bill.append(45000)` and `bill.pop()`. And Python *does* check the fence: `bill[8]` in a list of eight elements stops the program with `IndexError: list index out of range` [4]. A list may even hold values of different types, although the lists in this book never do.

There is also a difference in what the boxes hold. A C array of `long` holds the numbers themselves, eight bytes each, packed together. A Python list holds *references*: each element points to a separate number object somewhere else in memory. This is what lets a list hold values of any type and grow freely, and it costs memory and speed: a list of a million numbers takes several times the memory of a C array of a million `long` values, and walking through it is slower. For the few hundred orders of a laundry the difference does not matter at all. For millions of measurements, Python programmers use the standard `array` module or the NumPy library, which store the numbers packed together as C does.

6.3 Declaring, Filling and Walking Through an Array

Listing 6.1 declares the day's eight bills with an **initializer**, a list of values in braces, and then walks through the array four times: to print it, to add it up, to find the biggest bill and to count the bills of Rp 50,000 or more.

Listing 6.1 `day_bills.c`: eight bills in one array, printed, added, searched for the biggest and counted.

```c
/* day_bills.c - eight bills under one name: an array */
#include <stdio.h>

#define N 8

int main(void)
{
    long bill[N] = {28000, 30000, 30000, 70000, 136500, 21000, 45000, 60000};

    printf("index  bill\n");
    for (int i = 0; i < N; i++)          /* i runs 0, 1, ..., N-1 */
        printf("%5d  %6ld\n", i, bill[i]);

    long total = 0;
    for (int i = 0; i < N; i++)
        total += bill[i];

    int best = 0;                        /* index of the biggest bill so far */
    for (int i = 1; i < N; i++)
        if (bill[i] > bill[best])
            best = i;

    int big = 0;
    for (int i = 0; i < N; i++)
        if (bill[i] >= 50000)
            big++;

    printf("total   : %ld\n", total);
    printf("average : %.2f\n", (double) total / N);
    printf("biggest : %ld at index %d\n", bill[best], best);
    printf("bills of Rp 50000 or more: %d\n", big);
    printf("elements: %zu\n", sizeof bill / sizeof bill[0]);  /* bytes / bytes */
    return 0;
}
```

A few rules of the initializer are worth knowing. If the braces hold fewer values than the array has elements, the rest are set to zero, so `int count[4] = {0};` makes four zeros; that is the standard way to start a set of counters. If the size is left out, as in `long bill[] = {28000, 30000};`, the compiler counts the values and makes the array exactly that long. An array declared inside a function *without* an initializer is not set to zero; its elements hold whatever was in memory before, which is why every counter array in this book starts with `= {0}`.

It pays to trace one of these loops by hand once, the way the computer runs it. Before the adding loop, `total` is 0. With `i` = 0 the body adds `bill[0]`, 28000, and `total` becomes 28000; with `i` = 1 it adds 30000 and `total` becomes 58000; then 88000, 158000, 294500, 315500, 360500 and finally 420500 when `i` = 7. Now `i++` makes `i` equal to 8, the test `i < N` is false, and the loop ends without ever touching `bill[8]`, which does not exist. The same four lines (start, test, body, step) run in every loop of this chapter; only the body changes.

The search for the biggest bill keeps the *index* of the best bill so far, `best`, not its value. Knowing where the biggest bill is means knowing its value as well (`bill[best]`), and in the cashier of Section 6.9 it also gives the order number. The loop starts at 1 because `best = 0` has already looked at element 0.

The last line prints the number of elements with `sizeof bill / sizeof bill[0]`: the size in bytes of the whole array divided by the size of one element. This trick works only where the compiler can see the array's declaration, as in `main` here; Section 6.5 shows where it fails.

The average is the total divided by the number of values, the arithmetic **mean**:

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i \tag{6.2}$$

In Equation (6.2), x_i is element i and the symbol $\sum$ means "add up for i from 0 to $n-1$", which is exactly the second loop. The cast `(double) total / N` makes the division exact instead of whole-number division.

The Python version, **Listing 6.2**, walks through the list with `range(N)`, which produces the indexes 0 to $N-1$, so that the two programs look alike. Idiomatic Python would often write `for b in bill:` and use `sum(bill)` and `max(bill)` from the standard library; we write the loops ourselves because this chapter is about what those library functions do inside.

Listing 6.2 `day_bills.py`: the same walks through a Python list.

```
# day_bills.py - eight bills under one name: a list

N = 8


def main():
    bill = [28000, 30000, 30000, 70000, 136500, 21000, 45000, 60000]

    print("index  bill")
    for i in range(N):                  # i runs 0, 1, ..., N-1
        print(f"{i:5d}  {bill[i]:6d}")

    total = 0
    for i in range(N):
        total += bill[i]

    best = 0                            # index of the biggest bill so far
    for i in range(1, N):
        if bill[i] > bill[best]:
            best = i

    big = 0
    for i in range(N):
        if bill[i] >= 50000:
            big += 1

    print(f"total   : {total}")
    print(f"average : {total / N:.2f}")
    print(f"biggest : {bill[best]} at index {best}")
    print(f"bills of Rp 50000 or more: {big}")
    print(f"elements: {len(bill)}")      # a list knows its own length


if __name__ == "__main__":
    main()
```

```
index  bill
    0   28000
    1   30000
    2   30000
    3   70000
    4  136500
    5   21000
    6   45000
    7   60000
total   : 420500
average : 52562.50
biggest : 136500 at index 4
bills of Rp 50000 or more: 3
elements: 8
```

Check the numbers by hand once: the eight bills add up to 420500, and 420500 / 8 = 52562.5. Three bills (70000, 136500 and 60000) are at least 50000. The mean is higher than five of the eight bills, because the one big order of Rp 136,500 pulls it up; Section 6.12 comes back to this.

6.4 Arrays as Counters

The second question is "how many of each service?". Here the array plays a new role: its *elements* are counters, and the *index* says which thing is counted. **Listing 6.3** has the service codes of the day's twenty orders and an array `count` of four counters. For each order with code `s`, `count[s]++` adds one to counter number `s`. The value of the data becomes the index. Nothing needs to be compared or searched: the service code goes straight to its counter.

Listing 6.3 `service_count.c`: an array of counters, indexed by the service code, and a bar of stars for each.

```c
/* service_count.c - an array of counters: how often was each service ordered? */
#include <stdio.h>

int main(void)
{
    int service[20] = {1, 2, 1, 3, 1, 1, 2, 3, 1, 2,
                       1, 1, 3, 2, 1, 9, 1, 2, 1, 1};
    int count[4] = {0};                 /* count[1..3]; count[0] unused */
    int unknown = 0;

    for (int i = 0; i < 20; i++) {
        int s = service[i];
        if (s >= 1 && s <= 3)           /* guard: never index outside 0..3 */
            count[s]++;                 /* the value itself is the index */
        else
            unknown++;
    }

    for (int s = 1; s <= 3; s++) {
        printf("service %d: %2d  ", s, count[s]);
        for (int k = 0; k < count[s]; k++)
            putchar('*');
        putchar('\n');
    }
    printf("unknown codes: %d\n", unknown);
    return 0;
}
```

One order has the impossible code 9, a typing slip. Without the guard `if (s >= 1 && s <= 3)`, the statement `count[9]++` would write past the end of the four-element array: exactly the undefined behavior of Section 6.2.3, and a bug that might go unnoticed for weeks while it quietly

changes some other variable. With the guard, the slip is counted separately and reported. Counter 0 is never used; wasting one element to let the service code be the index directly is a good trade.

The Python version, **Listing 6.4**, creates the counters with `[0] * 4`, a list of four zeros, and draws each bar with `"*" * count[s]`, a string repeated `count[s]` times.

Listing 6.4 `service_count.py`: the same counters in Python.

```python
# service_count.py - a list of counters: how often was each service ordered?


def main():
    service = [1, 2, 1, 3, 1, 1, 2, 3, 1, 2,
               1, 1, 3, 2, 1, 9, 1, 2, 1, 1]
    count = [0] * 4                      # count[1..3]; count[0] unused
    unknown = 0

    for s in service:
        if 1 <= s <= 3:                  # guard: Python would raise IndexError
            count[s] += 1                # the value itself is the index
        else:
            unknown += 1

    for s in range(1, 4):
        print(f"service {s}: {count[s]:2d}  " + "*" * count[s])
    print(f"unknown codes: {unknown}")


if __name__ == "__main__":
    main()
```

```
service 1: 11  ***********
service 2:  5  *****
service 3:  3  ***
unknown codes: 1
```

The same pattern works when the thing to count is not a small code but a range of values. To see how the bills are spread, count them in bands of Rp 25,000: a bill of `b` rupiah falls in band `b / 25000`, so Rp 28,000 is in band 1 (25.000 to 49.999) and Rp 136,500 in band 5. With `int band[6] = {0};` and a guard that puts every bill of Rp 125,000 or more in the last band, `band[b / 25000]++` builds a *histogram*, a picture of how many bills fall in each range. Whole-number division does the classification, and the array does the counting.

Eleven of twenty orders, more than half, are wash-and-fold. Mbak Rina now knows which detergent to buy in bulk. This *frequency counting* is one of the most useful patterns in programming: counting votes per candidate, sales per product, visitors per hour, letters per kind in a text (Chapter 9) all follow it.

6.5 Passing Arrays to Functions

Every question of the evening needs the same small jobs: add up, find the biggest, compute an average. Those jobs deserve functions, as Chapter 5 taught, and the functions need the array as a parameter. **Listing 6.5** shows how. The parameter is written `const long a[]`, with empty brackets, and the number of elements comes as a second parameter `n`.

Listing 6.5 `array_functions.c`: functions that receive an array and its size; `const` promises not to change it, and `add_to_all` changes the caller's array.

```c
/* array_functions.c - handing an array to a function */
#include <stdio.h>

long total(const long a[], int n)        /* const: we promise not to change a */
{
    long sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}

double average(const long a[], int n)
{
    return n > 0 ? (double) total(a, n) / n : 0.0;
}

int index_of_max(const long a[], int n)
{
    int best = 0;
    for (int i = 1; i < n; i++)
        if (a[i] > a[best])
            best = i;
    return best;
}

void add_to_all(long a[], int n, long amount)   /* changes the CALLER's array */
{
    for (int i = 0; i < n; i++)
        a[i] += amount;
}

void print_all(const long a[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%ld%s", a[i], i < n - 1 ? " " : "\n");
}

int main(void)
{
    long morning[5] = {28000, 30000, 30000, 70000, 21000};
    long evening[3] = {136500, 45000, 60000};
```

Listing 6.5 (Continued) *array_functions.c: functions that receive an array and its size; const promises not to change it, and add_to_all changes the caller's array.*

```c
    printf("morning: ");
    print_all(morning, 5);
    printf("total %ld, average %.2f, biggest at %d\n", total(morning, 5),
           average(morning, 5), index_of_max(morning, 5));
    printf("evening: total %ld, average %.2f, biggest at %d\n",
           total(evening, 3), average(evening, 3), index_of_max(evening, 3));

    add_to_all(evening, 3, 5000);        /* delivery fee for every evening bag */
    printf("evening after delivery fee: ");
    print_all(evening, 3);
    return 0;
}
```

Two facts about array parameters surprise everyone once.

An array is not copied. Chapter 5 said that C copies every argument into its parameter. For an array, C passes only the address of its first element, as if the call were `total(&morning[0], 5)`. The function works on the caller's own boxes. This is fast (copying a thousand bills at every call would be slow), and it means that a function *can* change the caller's array, as `add_to_all` does

when it adds the delivery fee of Rp 5,000 to every evening bill. Chapter 7 explains the mechanism, which is a pointer.

The function does not know the length. Because only an address arrives, `sizeof a` inside the function gives the size of an address (eight bytes on most computers), not of the array, and `gcc -Wall` warns if you try. That is why every function here takes `n` as a second parameter, and why the same function can serve the five-element morning and the three-element evening.

The keyword `const` in `const long a[]` is a promise that the function only reads the array. If the body tried `a[0] = 0;`, the compiler would refuse. Put `const` on every array parameter that is only read: it documents the intention, it lets the caller pass arrays that must not change, and it turns an accidental assignment into a compile error instead of a bug.

Try it once to see the promise enforced: add the line `a[0] = 0;` to `total` and compile. `gcc` stops with *error: assignment of read-only location*, pointing at the line. A mistake that would otherwise have changed Mbak Rina's data silently is caught before the program even exists.

In Python, **Listing 6.6**, a list is passed the same way: the function receives the caller's list itself, not a copy, so `add_to_all` changes it. Python has no `const`; a function that must not change a list is simply written not to, or makes its own copy with `list(a)`.

Listing 6.6 `array_functions.py`: the same functions on Python lists.

```
# array_functions.py - handing a list to a function


def total(a):
    s = 0
    for x in a:
        s += x
    return s


def average(a):
    return total(a) / len(a) if a else 0.0


def index_of_max(a):
    best = 0
    for i in range(1, len(a)):
        if a[i] > a[best]:
            best = i
    return best


def add_to_all(a, amount):               # changes the CALLER's list
    for i in range(len(a)):
        a[i] += amount


def print_all(a):
    print(" ".join(str(x) for x in a))


def main():
    morning = [28000, 30000, 30000, 70000, 21000]
    evening = [136500, 45000, 60000]

    print("morning: ", end="")
    print_all(morning)
    print(f"total {total(morning)}, average {average(morning):.2f}, "
          f"biggest at {index_of_max(morning)}")
    print(f"evening: total {total(evening)}, average {average(evening):.2f}, "
          f"biggest at {index_of_max(evening)}")

    add_to_all(evening, 5000)            # delivery fee for every evening bag
    print("evening after delivery fee: ", end="")
    print_all(evening)
```

Listing 6.6 (Continued) *array_functions.py: the same functions on Python lists.*

```
if __name__ == "__main__":
    main()
```

```
morning: 28000 30000 30000 70000 21000
total 179000, average 35800.00, biggest at 3
evening: total 241500, average 80500.00, biggest at 0
evening after delivery fee: 141500 50000 65000
```

6.6 Sorting: Bubble Sort

Sorting is putting things in order: bills from small to big, names from A to Z. When Knuth wrote his volume on sorting, computer makers estimated that a quarter of all running time was spent sorting [5], and although the share has changed, sorted data still make almost everything else

easier: the median is the middle of a sorted list, the top three are its end, and a sorted list can be searched in a few steps instead of many.

Bubble sort is the simplest method to understand. Walk from left to right and compare each pair of neighbors; when a pair is in the wrong order, swap it. After one such *pass*, the biggest value has been carried, swap by swap, all the way to the right end, like a bubble rising to the surface, and it is in its final place. Repeat the pass on the part that is not yet in place. After $n - 1$ passes all values are in order; and if a whole pass makes no swap at all, the array is already sorted and we can stop early. **Pseudocode 6.1** writes the method in pseudocode, the plain step-by-step language of Chapter 2.

Pseudocode 6.1 Bubble sort with early exit: each pass carries the biggest remaining value to the end of the unsorted part.

```
BUBBLE-SORT(A, n)
1  for pass = 1 to n - 1
2      swapped = FALSE
3      for i = 0 to n - pass - 1
4          if A[i] > A[i + 1]
5              exchange A[i] with A[i + 1]
6              swapped = TRUE
7      if not swapped
8          return            // no swap in a whole pass: sorted
```

Figure 6.2 shows the passes on six bills taken from the counter. After pass 1, the biggest bill, 136500, is at the right end. After pass 2, 70000 is next to it. Pass 4 swaps nothing, and the method stops without the fifth pass.

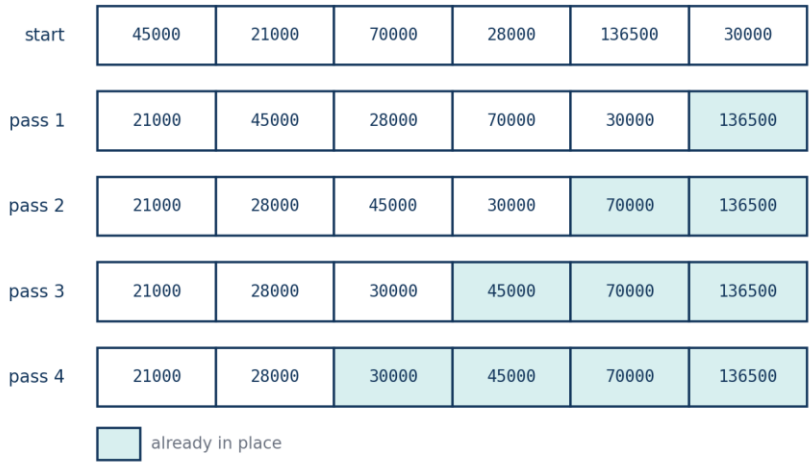


Figure 6.2 Bubble sort on six bills: after each pass one more value (shaded) is in its final place at the right; pass 4 makes no swap, so the sort stops early.

Follow pass 1 comparison by comparison, because every sorting method is built from exactly these small steps. The array starts as 45000, 21000, 70000, 28000, 136500, 30000. Compare positions 0 and 1: 45000 is bigger than 21000, so swap, giving 21000, 45000, 70000, 28000, 136500, 30000. Compare positions 1 and 2: 45000 and 70000 are in order, no swap. Compare 2 and 3: 70000 is bigger than 28000, swap, giving 21000, 45000, 28000, 70000, 136500, 30000. Compare 3 and 4: 70000 and 136500 are in order. Compare 4 and 5: 136500 is bigger than 30000, swap, giving 21000, 45000, 28000, 70000, 30000, 136500. Five comparisons and three swaps, and the biggest bill has traveled from position 4 to the end. Notice that 136500 did not move until the last comparison, and then moved only one place: a big value is carried right in one

pass, but a small value near the end, like 30000, moves left by only one place per pass. That asymmetry is why bubble sort needs many passes when small values start at the right.

> **KEY INSIGHT:** After pass p, the last p positions hold the p biggest values, in their final order. This fact, true after every pass, is the *loop invariant* of bubble sort, and it explains both why the inner loop may stop at `n - pass` (the tail is already right) and why $n-1$ passes are always enough (after them, the last $n-1$ positions are right, so the first one must be too).

How much work is it? Pass 1 compares $n-1$ pairs, pass 2 compares $n-2$, and so on down to 1. In the worst case, a list in reverse order, every pass is needed and the number of comparisons is

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2} \tag{6.3}$$

For the 6 bills of **Figure 6.2** that is at most 15 comparisons, harmless. For a year of orders, say 20,000, Equation (6.3) gives almost 200 million comparisons, which a modern computer still does in well under a second, but which grows fourfold every time the list doubles. Bubble sort is the right method to learn first and almost never the right method to use for large data; Astrachan's history of the algorithm is a lively account of why it is taught anyway [6]. Better methods, such as merge sort and quicksort, need about $n\log_2 n$ comparisons; the library function `qsort` of Chapter 7 and Python's `sorted` use such methods [7].

One more property of bubble sort matters in practice. It swaps two neighbors only when the left one is *strictly* bigger, so two equal values never change places: two bills of Rp 30,000 from different customers stay in the order in which the receipts were written. A sort with this property is called **stable**. Stability matters as soon as the data carry more than the key being sorted: sort the orders by bill, and a stable sort keeps orders with the same bill in receipt order, which is what a reader of the report expects. Insertion sort is stable too; Python's `sorted` is guaranteed stable [7]; C's `qsort` is not.

6.7 Searching: Linear and Binary

To find a value in an array that is *not* sorted, there is only one way: look at the elements one by one from the front until you find it or run out. That is **linear search**, the function `linear_search` of **Listing 6.7**. It returns the index where the value was found, or -1, which can never be an index, to say "not there". In the worst case it looks at all n elements.

In a *sorted* array there is a much better way, the one you use in a dictionary. Open it in the middle. If the word you want comes before the middle word, it is in the left half, and you never need to look at the right half again; otherwise it is in the right half. Each look throws away half of what is left. That is **binary search**. The variables `low` and `high` mark the part still in question, `mid` is its middle, and the loop ends when the value is found or the part is empty (`low > high`).

Listing 6.7 `sort_search.c`: bubble sort with a printed trace of each pass, linear search, and binary search with a printed trace of each look.

```c
/* sort_search.c - bubble sort with a trace, then linear and binary search */
#include <stdio.h>

void print_all(const long a[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%7ld", a[i]);
    printf("\n");
}

void bubble_sort(long a[], int n)
{
    for (int pass = 1; pass < n; pass++) {
        int swapped = 0;
        for (int i = 0; i < n - pass; i++) {    /* the tail is already sorted */
            if (a[i] > a[i + 1]) {
                long t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
                swapped = 1;
            }
        }
        printf("pass %d:", pass);
        print_all(a, n);
        if (!swapped)                           /* nothing moved: done early */
            break;
    }
}

int linear_search(const long a[], int n, long key)
{
    for (int i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;                                  /* -1 means "not there" */
}
```

Listing 6.7 (Continued) *sort_search.c: bubble sort with a printed trace of each pass, linear search, and binary search with a printed trace of each look.*

```c
int binary_search(const long a[], int n, long key)   /* a must be sorted */
{
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        printf("  low=%d mid=%d high=%d looks at %ld\n", low, mid, high, a[mid]);
        if (a[mid] == key)
            return mid;
        if (a[mid] < key)
            low = mid + 1;                      /* key is in the right half */
        else
            high = mid - 1;                     /* key is in the left half */
    }
    return -1;
}

int main(void)
{
    long bill[6] = {45000, 21000, 70000, 28000, 136500, 30000};
    printf("start: ");
    print_all(bill, 6);
    printf("linear search for 28000: index %d\n", linear_search(bill, 6, 28000));

    bubble_sort(bill, 6);
    printf("binary search for 70000:\n");
    printf("found at index %d\n", binary_search(bill, 6, 70000));
    printf("binary search for 50000:\n");
    printf("found at index %d\n", binary_search(bill, 6, 50000));
    return 0;
}
```

The middle is computed as `low + (high - low) / 2` rather than `(low + high) / 2`. The two are equal in mathematics, but for a very large array `low + high` can exceed the largest `int`, and the second form then gives a negative index. This bug hid in the binary search of Java's standard library for nine years before Joshua Bloch reported it in 2006 [8]; the first form never overflows.

Listing 6.8 is the Python version. Swapping two elements is one line in Python, `a[i], a[i + 1] = a[i + 1], a[i]`, where C needs a temporary variable `t`.

Listing 6.8 `sort_search.py`: the same sort and searches in Python.

```
# sort_search.py - bubble sort with a trace, then linear and binary search


def print_all(a):
    print("".join(f"{x:7d}" for x in a))


def bubble_sort(a):
    n = len(a)
    for pass_no in range(1, n):
        swapped = False
        for i in range(n - pass_no):         # the tail is already sorted
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        print(f"pass {pass_no}:", end="")
        print_all(a)
        if not swapped:                      # nothing moved: done early
            break


def linear_search(a, key):
    for i in range(len(a)):
        if a[i] == key:
            return i
    return -1                                # -1 means "not there"


def binary_search(a, key):                   # a must be sorted
    low, high = 0, len(a) - 1
    while low <= high:
        mid = low + (high - low) // 2
        print(f"  low={low} mid={mid} high={high} looks at {a[mid]}")
        if a[mid] == key:
            return mid
        if a[mid] < key:
            low = mid + 1                    # key is in the right half
        else:
            high = mid - 1                   # key is in the left half
    return -1
```

Listing 6.8 (Continued) *sort_search.py: the same sort and searches in Python.*

```
def main():
    bill = [45000, 21000, 70000, 28000, 136500, 30000]
    print("start: ", end="")
    print_all(bill)
    print(f"linear search for 28000: index {linear_search(bill, 28000)}")

    bubble_sort(bill)
    print("binary search for 70000:")
    print(f"found at index {binary_search(bill, 70000)}")
    print("binary search for 50000:")
    print(f"found at index {binary_search(bill, 50000)}")


if __name__ == "__main__":
    main()
```

```
start:   45000  21000  70000  28000 136500  30000
linear search for 28000: index 3
pass 1:  21000  45000  28000  70000  30000 136500
pass 2:  21000  28000  45000  30000  70000 136500
pass 3:  21000  28000  30000  45000  70000 136500
pass 4:  21000  28000  30000  45000  70000 136500
binary search for 70000:
  low=0 mid=2 high=5 looks at 30000
  low=3 mid=4 high=5 looks at 70000
found at index 4
binary search for 50000:
  low=0 mid=2 high=5 looks at 30000
  low=3 mid=4 high=5 looks at 70000
  low=3 mid=3 high=3 looks at 45000
found at index -1
```

The output shows the passes of **Figure 6.2** and then two binary searches in the sorted array. For 70000 the first look, at index 2, sees 30000, which is smaller, so the search continues in the right half, indexes 3 to 5; the second look finds 70000 at index 4. For 50000, which is not there, the third look narrows the part to the single index 3, sees 45000, and the next step would make `low` greater than `high`: not found.

How many looks can binary search need? Each look at least halves the part that is left. Starting from n elements, after k looks at most $n/2^k$ remain, and the search must stop when fewer than one remains. The largest number of looks is therefore

$$k_{\max} = \lfloor \log_2 n \rfloor + 1 \tag{6.4}$$

where $\log_2 n$ is the power to which 2 must be raised to give n, and $\lfloor \cdot \rfloor$ rounds down. For $n = 6$, $\log_2 6 \approx 2.58$, so Equation (6.4) gives 3 looks, exactly what the search for 50000 needed. For a million receipts it gives 20 looks, against up to a million for linear search. That is the reward for sorting first, and it is why a telephone book, a dictionary and the index of this book are sorted [9].

Which search should a program use? It depends on how often the data change and how often they are searched. Sorting costs far more than one linear search, so for a list that is searched once, linear search wins. For a list that is sorted once and searched thousands of times, such as the price list of a supermarket or the receipt numbers of a month, sorting once and then searching with binary search wins by a wide margin. Mbak Rina's cashier searches for a receipt perhaps five times a day in a list of fifty: linear search is the honest choice, and the simpler code is worth more than the saved microseconds. The day the laundry keeps a year of receipts and a customer asks "where is my order 18342?", binary search over the sorted receipt numbers becomes the right tool, and Chapter 8 builds exactly that.

6.8 Two-Dimensional Arrays

Mbak Rina's last question looks over the whole week: kilograms washed per day and per service. The data form a *table*, seven rows for the days and three columns for the services. C stores a table in a **two-dimensional array**, declared with two sizes: `int kg[7][3];`. The element in row `d` and column `s` is `kg[d][s]`.

Listing 6.9 fills the table with an initializer (one inner pair of braces per row), prints it with a total for each row, adds a line with the total of each column, and finds the busiest day. The day names are a two-dimensional array of characters, `char day[7][4]`: seven rows of four characters,

enough for three letters and the `'\0'` that ends every string in C (Chapter 9 explains strings in full).

Listing 6.9 `week_table.c`: kilograms per day and service in a two-dimensional array, with row totals, column totals and the busiest day.

```
/* week_table.c - a two-dimensional array: kilograms per day and service */
#include <stdio.h>

#define DAYS 7
#define SERVICES 3

int main(void)
{
    char day[DAYS][4] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
    int kg[DAYS][SERVICES] = {          /* rows: days; columns: services 1..3 */
        {42, 18, 10}, {38, 20, 12}, {40, 15, 9}, {45, 22, 11},
        {51, 30, 14}, {66, 41, 20}, {58, 35, 16}
    };
    int col_total[SERVICES] = {0};
    int grand = 0, busiest = 0, busiest_kg = -1;

    printf("day   fold  iron  only  total\n");
    for (int d = 0; d < DAYS; d++) {
        int row_total = 0;
        printf("%s ", day[d]);
        for (int s = 0; s < SERVICES; s++) {
            printf("%6d", kg[d][s]);
            row_total += kg[d][s];
            col_total[s] += kg[d][s];
        }
        printf("%7d\n", row_total);
        grand += row_total;
        if (row_total > busiest_kg) {
            busiest_kg = row_total;
            busiest = d;
        }
    }
    printf("all ");
    for (int s = 0; s < SERVICES; s++)
        printf("%6d", col_total[s]);
    printf("%7d\n", grand);
    printf("busiest day: %s with %d kg\n", day[busiest], busiest_kg);
    return 0;
}
```

The pattern of two nested loops, the outer one over the rows and the inner one over the columns, is the standard way to visit every cell of a table. The row total is reset for each row inside the outer loop; the column totals are an array of three counters that live across all rows.

How is a table stored in memory, which is one long row of bytes? Row after row: first the three elements of row 0, then the three of row 1, and so on. This is called *row-major* order, and **Figure 6.3** draws it for the first three days. The address of `kg[r][c]` in an array with C columns and elements of s bytes is

$$\text{address}(\text{kg}[r][c]) = a + (r \cdot C + c) \cdot s \tag{6.5}$$

because r whole rows of C elements come before row r, and c elements come before column c within it. This is why C must know the number of columns of a two-dimensional array parameter (`int kg[][3]`): without C, Equation (6.5) cannot be computed.

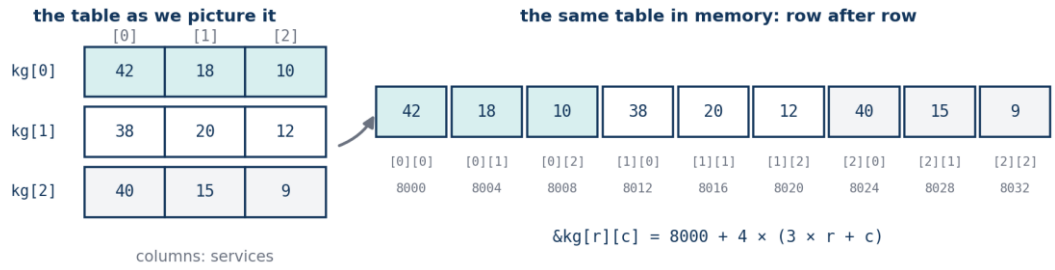


Figure 6.3 The first three rows of the week table, as we picture them and as they lie in memory: row after row, so that `kg[r][c]` lies $3r + c$ elements after the start (four bytes per `int`).

A function that receives the whole table must be told the number of columns in its parameter, for the reason just given, while the number of rows may be passed separately like the length of a one-dimensional array: `int row_total(const int kg[][3], int d)` adds up row `d`, and `int column_total(const int kg[][3], int days, int s)` adds up column `s` over `days` rows. Inside, `kg[d][s]` works exactly as in `main`. C99 also allows the number of columns itself to be a parameter, `int total(int rows, int cols, const int t[rows][cols])`, a *variable-length array* parameter; it is handy but optional in C11 and later, so this book keeps fixed column counts.

Python has no two-dimensional array; the usual substitute is a **list of lists**, one inner list per row, and `kg[d][s]` works the same way, as **Listing 6.10** shows.

Listing 6.10 `week_table.py`: the same table as a Python list of lists.

```
# week_table.py - a list of lists: kilograms per day and service


def main():
    day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
    kg = [                                  # rows: days; columns: services 1..3
        [42, 18, 10], [38, 20, 12], [40, 15, 9], [45, 22, 11],
        [51, 30, 14], [66, 41, 20], [58, 35, 16],
    ]
    col_total = [0] * 3
    grand, busiest, busiest_kg = 0, 0, -1

    print("day   fold  iron  only  total")
    for d in range(7):
        row_total = 0
        line = f"{day[d]} "
        for s in range(3):
            line += f"{kg[d][s]:6d}"
            row_total += kg[d][s]
            col_total[s] += kg[d][s]
        print(f"{line}{row_total:7d}")
        grand += row_total
        if row_total > busiest_kg:
            busiest_kg, busiest = row_total, d
    print("all " + "".join(f"{c:6d}" for c in col_total) + f"{grand:7d}")
    print(f"busiest day: {day[busiest]} with {busiest_kg} kg")


if __name__ == "__main__":
    main()
```

```
day   fold  iron  only  total
Mon     42    18    10     70
Tue     38    20    12     70
Wed     40    15     9     64
Thu     45    22    11     78
Fri     51    30    14     95
Sat     66    41    20    127
Sun     58    35    16    109
all    340   181    92    613
busiest day: Sat with 127 kg
```

Saturday is the busiest day with 127 kg, followed by Sunday, so the part-time helper is needed at the weekend. Across the week, wash-and-fold accounts for 340 of the 613 kg, more than half, matching the counts of Section 6.4.

> **WARNING:** In Python, never create a table with `[[0] * 3] * 7`. It makes *one* inner list and seven references to it, so changing `kg[0][0]` changes the first cell of every row. Write `[[0] * 3 for _ in range(7)]`, which makes seven separate rows [10].

6.9 Building the Cashier, Version 5

Now the cashier learns to remember. Version 5 reads the day's orders into arrays and then answers the evening questions. It reuses `laundry.h` and `laundry.c` of Chapter 5 unchanged (the prices, the charging rule and the rupiah printer) and adds a second small module for arrays of bills. **Listing 6.11** is its header, and **Listing 6.12** its code.

Listing 6.11 `bills.h`: the promises of the module for arrays of bills.

```c
/* bills.h - small tools for arrays of bills (Chapter 6) */
#ifndef BILLS_H
#define BILLS_H

long total(const long a[], int n);
int index_of_max(const long a[], int n);
void copy_bills(const long from[], long to[], int n);
void sort_descending(long a[], int n);
long median(const long a[], int n);      /* 0 for an empty array */

#endif
```

Listing 6.12 `bills.c`: total, index of the maximum, copying, sorting from biggest to smallest, and the median.

```c
/* bills.c - small tools for arrays of bills */
#include "bills.h"

#define MAX_COPY 1000

long total(const long a[], int n)
{
    long sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}

int index_of_max(const long a[], int n)
{
    int best = 0;
    for (int i = 1; i < n; i++)
        if (a[i] > a[best])
            best = i;
    return best;
}

void copy_bills(const long from[], long to[], int n)
{
    for (int i = 0; i < n; i++)
        to[i] = from[i];
}

void sort_descending(long a[], int n)    /* bubble sort, biggest first */
{
    for (int pass = 1; pass < n; pass++) {
        int swapped = 0;
        for (int i = 0; i < n - pass; i++) {
            if (a[i] < a[i + 1]) {
                long t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
                swapped = 1;
            }
        }
        if (!swapped)
            break;
    }
}
```

Listing 6.12 (Continued) *bills.c: total, index of the maximum, copying, sorting from biggest to smallest, and the median.*

```c
long median(const long a[], int n)
{
    long sorted[MAX_COPY];
    if (n <= 0 || n > MAX_COPY)
        return 0;
    copy_bills(a, sorted, n);            /* never sort the caller's data */
    sort_descending(sorted, n);
    if (n % 2 == 1)
        return sorted[n / 2];
    return (sorted[n / 2 - 1] + sorted[n / 2]) / 2;
}
```

The **median** is the middle value of the sorted data: half of the bills are at least as big, half at most as big. For an odd number of values it is the one in the middle; for an even number it is the mean of the two middle ones, here in whole rupiah. The function `median` sorts a *copy*, because a function whose job is to answer a question must not reorder the caller's data as a side effect: the order of

the bills is the order of the receipts, and the report later prints the biggest order by its receipt number.

The report itself, **Listing 6.13**, stores the orders in three **parallel arrays**: `service[i]`, `weight[i]` and `bill[i]` together describe order number *i*. The loop reads at most `MAX_ORDERS` orders, which protects the arrays from an unexpectedly busy day. The per-service totals are arrays indexed by the service code, the counting pattern of Section 6.4 applied to counts, kilograms and money at once.

Listing 6.13 `day_report.c`: the cashier, version 5, which stores the day in parallel arrays and prints the closing report.

```
/* day_report.c - Profitina Laundry cashier, version 5: it remembers the day.
   Input: one order per line, "service weight express", ending with 0. */
#include <stdio.h>
#include "laundry.h"
#include "bills.h"

#define MAX_ORDERS 100

int main(void)
{
    int service[MAX_ORDERS];            /* parallel arrays: order i is */
    double weight[MAX_ORDERS];          /* service[i], weight[i], bill[i] */
    long bill[MAX_ORDERS];
    int n = 0, s, express;
    double w;

    while (n < MAX_ORDERS && scanf("%d", &s) == 1 && s != 0) {
        if (scanf("%lf %d", &w, &express) != 2)
            break;
        if (price_per_kg(s) == 0 || w <= 0 || w > MAX_KG)
            continue;                   /* the v4 cashier already said why */
        service[n] = s;
        weight[n] = w;
        bill[n] = bill_for(s, w, express);
        n++;
    }
    if (n == 0) {
        printf("no orders today\n");
        return 0;
    }

    int count[4] = {0};
    double kg[4] = {0};
    long revenue[4] = {0};
    for (int i = 0; i < n; i++) {
        count[service[i]]++;
        kg[service[i]] += weight[i];
        revenue[service[i]] += bill[i];
    }
```

Listing 6.13 (Continued) *day_report.c: the cashier, version 5, which stores the day in parallel arrays and prints the closing report.*

```
    printf("PROFITINA LAUNDRY - day report (v5)\n");
    printf("service       orders      kg  revenue\n");
    for (int k = 1; k <= 3; k++) {
        print_service(k);
        printf("  %5d  %6.1f  ", count[k], kg[k]);
        print_rupiah(revenue[k]);
        printf("\n");
    }
    long sum = total(bill, n);
    int best = index_of_max(bill, n);
    printf("orders %d, revenue ", n);
    print_rupiah(sum);
    printf(", average ");
    print_rupiah(sum / n);
    printf("\nbiggest: order #%03d, ", best + 1);
    print_rupiah(bill[best]);
    printf("\nmedian bill: ");
    print_rupiah(median(bill, n));

    long top[MAX_ORDERS];
    copy_bills(bill, top, n);
    sort_descending(top, n);
    printf("\ntop three:");
    for (int i = 0; i < 3 && i < n; i++) {
        printf("  ");
        print_rupiah(top[i]);
    }
    printf("\n");
    return 0;
}
```

Read the input loop of `day_report.c` slowly, because its condition does three jobs in a fixed order. `n < MAX_ORDERS` is tested first: when the arrays are full, the loop stops *before* `scanf` reads another order, so no order is read and then lost. Next, `scanf("%d", &s) == 1` fails at the end of the input, and `s != 0` stops at the closing code. Because `&&` evaluates from left to right and stops at the first false part (Chapter 3), the order of the three tests matters. Inside the loop, an invalid order is skipped with `continue` *before* anything is stored, so the three arrays always hold exactly `n` valid orders at the same positions; this *invariant*, a fact that stays true at every turn of the loop, is what makes the later loops over `i < n` safe.

Parallel arrays work, but they are fragile: sorting `bill` alone would separate each bill from its service and weight. That is why the report sorts a copy, `top`. Chapter 10 replaces the three arrays by one array of *structures*, in which each element holds all the facts of one order together.

The Python versions follow the same design. **Listing 6.14** is the module; `median` copies the list with `list(a)` before sorting.

Listing 6.14 `bills.py`: the same tools for Python lists.

```
# bills.py - small tools for lists of bills


def total(a):
    s = 0
    for x in a:
        s += x
    return s


def index_of_max(a):
    best = 0
    for i in range(1, len(a)):
        if a[i] > a[best]:
            best = i
    return best


def sort_descending(a):                  # bubble sort, biggest first
    n = len(a)
    for pass_no in range(1, n):
        swapped = False
        for i in range(n - pass_no):
            if a[i] < a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        if not swapped:
            break


def median(a):
    if not a:
        return 0
    s = list(a)                          # never sort the caller's data
    sort_descending(s)
    n = len(s)
    if n % 2 == 1:
        return s[n // 2]
    return (s[n // 2 - 1] + s[n // 2]) // 2
```

Listing 6.15 is the report. Here the lists start empty and grow with `append`, so no `n` is needed: `len(bill)` is always the number of stored orders.

Listing 6.15 `day_report.py`: the cashier, version 5, in Python.

```
# day_report.py - Profitina Laundry cashier, version 5: it remembers the day.
# Input: one order per line, "service weight express", ending with 0.
import sys
from laundry import (MAX_KG, price_per_kg, service_name, bill_for, rupiah)
from bills import total, index_of_max, sort_descending, median

MAX_ORDERS = 100


def main():
    words = sys.stdin.read().split()
    service, weight, bill = [], [], []   # parallel lists: order i is
    pos = 0                              # service[i], weight[i], bill[i]
    while (len(bill) < MAX_ORDERS and pos < len(words)
           and int(words[pos]) != 0 and pos + 2 < len(words)):
        s, w = int(words[pos]), float(words[pos + 1])
        express = int(words[pos + 2])
        pos += 3
        if price_per_kg(s) == 0 or w <= 0 or w > MAX_KG:
            continue                     # the v4 cashier already said why
        service.append(s)
        weight.append(w)
        bill.append(bill_for(s, w, express))
    n = len(bill)
    if n == 0:
        print("no orders today")
        return

    count, kg, revenue = [0] * 4, [0.0] * 4, [0] * 4
    for i in range(n):
        count[service[i]] += 1
        kg[service[i]] += weight[i]
        revenue[service[i]] += bill[i]

    print("PROFITINA LAUNDRY - day report (v5)")
    print("service       orders      kg  revenue")
    for k in range(1, 4):
        print(f"{service_name(k)}  {count[k]:5d}  {kg[k]:6.1f}  "
              f"{rupiah(revenue[k])}")
    s = total(bill)
    best = index_of_max(bill)
    print(f"orders {n}, revenue {rupiah(s)}, average {rupiah(s // n)}")
    print(f"biggest: order #{best + 1:03d}, {rupiah(bill[best])}")
    print(f"median bill: {rupiah(median(bill))}")
```

Listing 6.15 (Continued) *day_report.py: the cashier, version 5, in Python.*

```
    top = list(bill)
    sort_descending(top)
    print("top three:" + "".join("  " + rupiah(x) for x in top[:3]))


if __name__ == "__main__":
    main()
```

6.10 Running It

A Saturday at the counter, with twelve orders of which one has the unknown service 7:

```
1 3.4 0
2 1.2 1
3 5 0
2 6.5 0
1 12.75 1
1 4.2 0
3 2.5 0
2 3 1
1 7.9 0
7 3 0
3 1 0
1 5.5 1
0
```

```
PROFITINA LAUNDRY - day report (v5)
service       orders      kg  revenue
Wash & fold        5    33.8  Rp 318.500
Wash & iron        3    10.7  Rp 145.000
Iron only          3     8.5  Rp 60.000
orders 11, revenue Rp 523.500, average Rp 47.590
biggest: order #005, Rp 136.500
median bill: Rp 35.000
top three:  Rp 136.500  Rp 70.000  Rp 63.000
```

Verify the report by hand. The five wash-and-fold orders weigh 3.4 + 12.75 + 4.2 + 7.9 + 5.5 = 33.75 kg, printed with one decimal as 33.8. Their bills are Rp 28,000, 136,500, 35,000 (4.2 kg is charged as 5 kg), 56.000 (8 kg) and 63.000 (6 kg at Rp 7,000 plus half for express), which add up to Rp 318,500. The eleven valid orders bring Rp 523,500; divided by 11 that is 47590.9, and the report prints Rp 47,590 because `sum / n` is whole-number division. Sorted from the biggest, the bills are 136500, 70000, 63000, 56000, 45000, 35000, 30000, 30000, 28000, 18000, 12000; the sixth of eleven is the median, Rp 35,000.

The other two services check the same way. Wash-and-iron has three orders: 1.2 kg as express (the minimum 2 kg at Rp 10,000 is Rp 20,000, plus half, Rp 30,000), 6.5 kg (7 kg, Rp 70,000) and 3 kg as express (Rp 30,000 plus half, Rp 45,000), together Rp 145,000 for 10.7 kg. Iron-only has 5 kg (Rp 30,000), 2.5 kg (3 kg, Rp 18,000) and 1 kg (the minimum, Rp 12,000), together Rp 60,000 for 8.5 kg. The three revenues add up to 318500 + 145000 + 60000 = 523500, the day's total, and the counts 5 + 3 + 3 to the eleven orders: the parts agree with the whole. The order with service 7 appears nowhere, as it should. A report whose parts do not add up to its whole has a bug, and this cross-check is the quickest way to find one.

The mean of Rp 47,590 is higher than seven of the eleven bills. The median of Rp 35,000 says better what a typical customer pays, because one order of Rp 136,500 pulls the mean up but moves the median only one place. When Mbak Rina wants to know "what does a normal customer spend?", the median is the honest answer; when she wants to know "how much money came in per order?", the mean is.

6.11 Testing It

Listing 6.16 tests the module on small arrays whose answers are obvious: five values whose total is 150 and whose middle value is 30, a one-element array, and an even-length array whose median is the mean of 2 and 3 in whole numbers, 2. Two checks are about behavior rather than results: after `median(a, 5)` the array `a` must be unchanged, and an empty day (`n` = 0) must give a total of 0 rather than crash.

Listing 6.16 `test_bills.c`: nine checks of the array tools, including the empty array and the promise that `median` leaves its input unchanged.

```c
/* test_bills.c - checks the array tools on small arrays worked by hand */
#include <stdio.h>
#include "bills.h"

static int checks = 0, failures = 0;

static void check(long got, long expected, int line)
{
    checks++;
    if (got != expected) {
        failures++;
        printf("FAILED on line %d: got %ld, expected %ld\n", line, got, expected);
    }
}

int main(void)
{
    long a[5] = {30, 10, 50, 20, 40};
    long one[1] = {7};
    long even[4] = {4, 1, 3, 2};

    check(total(a, 5), 150, __LINE__);
    check(total(a, 0), 0, __LINE__);           /* an empty day */
    check(index_of_max(a, 5), 2, __LINE__);
    check(index_of_max(one, 1), 0, __LINE__);
    check(median(a, 5), 30, __LINE__);
    check(median(even, 4), 2, __LINE__);       /* (3 + 2) / 2 in whole rupiah */
    check(a[0], 30, __LINE__);                 /* median left a untouched */
    sort_descending(a, 5);
    check(a[0], 50, __LINE__);
    check(a[4], 10, __LINE__);
    printf("%d checks, %d failures\n", checks, failures);
    return failures != 0;
}
```

Listing 6.17 is the same test in Python. The empty day is written `total([])`, an empty list, and the promise about the input is checked in the same way, by looking at `a[0]` after the call.

Listing 6.17 `test_bills.py`: the same checks in Python.

```
# test_bills.py - checks the list tools on small lists worked by hand
import inspect
import sys
from bills import total, index_of_max, sort_descending, median

checks = 0
failures = 0


def check(got, expected):
    global checks, failures
    checks += 1
    if got != expected:
        failures += 1
        line = inspect.stack()[1].lineno
        print(f"FAILED on line {line}: got {got}, expected {expected}")


def main():
    a = [30, 10, 50, 20, 40]
    check(total(a), 150)
    check(total([]), 0)                  # an empty day
    check(index_of_max(a), 2)
    check(index_of_max([7]), 0)
    check(median(a), 30)
    check(median([4, 1, 3, 2]), 2)       # (3 + 2) // 2 in whole rupiah
    check(a[0], 30)                      # median left a untouched
    sort_descending(a)
    check(a[0], 50)
    check(a[4], 10)
    print(f"{checks} checks, {failures} failures")
    return 1 if failures else 0


if __name__ == "__main__":
    sys.exit(main())
```

```
9 checks, 0 failures
```

Notice also what is *not* tested here: the printing of the report. The report's output is checked another way, by comparing the whole output against a file of expected output, which is exactly what `run_tests.py` does for every program of the chapter. The two kinds of test complement each other: a function test points precisely at the function that is wrong, and an output test makes sure that all the pieces are connected correctly.

Why test such small arrays, when the real day has dozens of orders? Because a small array can be checked in your head, and because the mistakes that matter live at the edges: the first element, the last element, one element, no elements, an even or an odd count. A function that is right on those five cases is almost always right on the large ones, and when it is wrong, the small failing case shows *why* in a way that a report of eleven orders never does. When a bug is found later in real data, the first step is to shrink the data to the smallest array that still shows the bug, and to add that array to the test program before fixing anything, so that the bug can never come back unnoticed.

6.12 Making It Better

Guard the capacity. A C array has a fixed size. The loop condition `n < MAX_ORDERS` stops reading at 100 orders instead of writing past the end; a stronger version would print a warning that orders were ignored, so that a record day is not silently cut short. Choosing the capacity is a real

decision: 100 orders is three times Mbak Rina's busiest day, and 100 `long` values take 800 bytes. Chapter 10 shows how to ask for memory while the program runs, when the size cannot be known in advance.

Keep the order of the data. Every function that must sort for its answer sorts a copy. An alternative, useful when the data are large, is to sort an array of *indexes*: an array `order` holding 0 to $n - 1$, sorted by `bill[order[i]]`, so that the parallel arrays themselves never move. Exercise 6.7 asks for it.

Report robust numbers. As Section 6.10 showed, one unusual order moves the mean a lot and the median little. A report meant for decisions should show both, and perhaps also the smallest and biggest bill, so that the reader sees how spread out the day was. Chapter 8 adds the *standard deviation*, a single number for that spread.

Round the average consciously. `sum / n` cuts off the fraction; `(sum + n / 2) / n` rounds to the nearest rupiah. The difference is one rupiah here, but choosing the rule on purpose, and saying it, is the habit that matters when money is involved.

Keep the data after closing. When the program ends, the arrays vanish, and tomorrow's report cannot compare with today. Chapter 11 writes each order to a file as it is taken, so that the day survives a power cut and a month of files can be read back into arrays for a monthly report. The functions of `bills.c` will not change at all: they do not care whether the array was filled from the keyboard or from a file, which is the reward for keeping input, computation and output in separate functions.

6.13 Using It for Real

The day report fits any business that records many items of the same kind.

- **A warung** records each sale's amount and a product code; the counting pattern gives sales per product, the median gives the typical basket, and the week table becomes sales per day and per product.
- **A fruit and durian orchard** records the kilograms picked per tree per week; a two-dimensional array `kg[week][tree]` with row and column totals shows the best week and the best tree, and sorting the column totals ranks the trees.
- **A small clinic's front desk** records the waiting time of each patient; the mean, the median and the longest wait tell whether one more receptionist is needed in the morning. (Such a report is about the queue only; decisions about patients belong to the doctor or midwife.)
- **A school canteen** counts the drinks sold per hour with an array `count[24]` indexed by the hour, and the busiest hour tells when two sellers are needed.
- **A motorcycle parking attendant** records the hour each motorcycle enters and leaves; an array of 24 counters gives the number of motorcycles present at each hour, and its maximum tells whether the lot is big enough on market day.

In every case, decide first which values are stored per item (the parallel arrays, later the structure), which questions are answered at the end (totals, counts, order, search), and the largest number of items that can occur, which becomes the capacity.

6.14 Common Mistakes, and How to Avoid Them

- **Off by one at the end.** `for (i = 0; i <= n; i++)` visits $n + 1$ elements, one past the end. Write `i < n`, always.
- **Starting at 1.** Mathematics books often number from 1; C arrays start at 0. `bill[1]` is the *second* bill.
- **Uninitialized counters.** `int count[4];` inside a function holds garbage. Write `int count[4] = {0};`.
- **Using a value as an index without checking it.** `count[s]++` with an `s` from the input writes anywhere when `s` is out of range. Check the range first.
- **`sizeof` on an array parameter.** Inside a function, `sizeof a / sizeof a[0]` divides the size of an address by the size of an element; the result has nothing to do with the array. Pass the length as a parameter.
- **Sorting the caller's data by accident.** A function that sorts its array parameter changes the caller's array. Sort a copy when the order matters to the caller.
- **Binary search on unsorted data.** It returns wrong answers without any warning. Sort first, or use linear search.
- **`(low + high) / 2`.** Overflows for very large arrays. Write `low + (high - low) / 2`.
- **Python's shared rows.** `[[0] * 3] * 7` makes seven references to one row. Use a list comprehension.
- **Assigning arrays.** `a = b;` for two C arrays does not compile; copy element by element (as `copy_bills` does). In Python, `a = b` does compile, but makes `a` a second name for the same list; use `a = list(b)` for a copy.

6.15 Chapter Summary

- An array is a fixed number of elements of one type under one name, stored side by side; element i is at distance i from the start, Equation (6.1), which is why indexes run from 0 to $n - 1$ and why any element is reached in one step.
- C does not check indexes: stepping outside an array is undefined behavior. Loop with `i < n`, check indexes from input, and develop with `-fsanitize=address,undefined`. Python lists check and raise `IndexError`.
- Initializers set elements in braces, missing ones become zero; `= {0}` starts a set of counters. Using the data value as the index turns an array into counters for each kind.
- An array parameter receives the address of the caller's array: the function can change it, and does not know its length, so the length is passed as a parameter and `const` marks read-only arrays.
- Bubble sort carries the biggest remaining value to the end in each pass and needs up to $n(n - 1)/2$ comparisons, Equation (6.3); insertion sort is a close cousin. Library sorts are much faster for large data.
- Linear search works on any array and may look at all n elements; binary search needs sorted data and at most $\lfloor \log_2 n \rfloor + 1$ looks, Equation (6.4).
- A two-dimensional array stores a table row after row; the address of `kg[r][c]` follows Equation (6.5). Python uses a list of lists.

- The cashier, version 5, stores the day in parallel arrays and reports totals per service, the mean, the median and the top three; the median describes the typical customer better when one order is unusual.

6.16 Exercises

1. **Tracing indexes.** For `int a[5] = {4, 8, 15, 16, 23};`, what are `a[0]`, `a[4]`, `a[a[0] - 3]`, and the value of `s` after `int s = 0; for (int i = 1; i < 5; i += 2) s += a[i];`? Which expression would be undefined behavior: `a[5]`, `a[-1]`, or both?
2. **Addresses.** An array `double price[10]` starts at address 2000, and a `double` takes 8 bytes. What is the address of `price[7]`? A table `int t[4][6]` starts at 9000 with 4-byte `int` values; what is the address of `t[2][5]`?
3. **Reverse in place.** Write `void reverse(long a[], int n)` that reverses an array without a second array, by swapping the first and last elements, then the second and second-to-last, and so on. Test it on arrays of odd and even length.
4. **The smallest and its position.** Write `int index_of_min(const long a[], int n)`. What should it return for `n = 0`, and how should the caller deal with that?
5. **Insertion sort.** Write insertion sort, the way card players sort a hand: take the cards one by one from the left and insert each into its place among the cards already sorted, shifting bigger cards one place to the right. Print the array after each insertion for the six bills of **Figure 6.2**. How many comparisons does it make on an array that is already sorted?
6. **Counting comparisons.** Using Equation (6.3) and Equation (6.4), fill in a table for n = 10, 1000 and 1,000,000: the worst-case comparisons of bubble sort, and the worst-case looks of linear and of binary search.
7. **Sorting indexes.** Write a function that fills `int order[]` with 0 to $n - 1$ and sorts it so that `bill[order[0]]` is the biggest bill, *without moving the elements of* `bill`. Why is this useful with parallel arrays?

6.17 Solutions to the Exercises

1. `a[0]` is 4, `a[4]` is 23, and `a[a[0] - 3]` is `a[1]`, which is 8. The loop visits `i` = 1 and 3, so `s` = 8 + 16 = 24. Both `a[5]` (one past the end) and `a[-1]` (one before the start) are undefined behavior.

2. By Equation (6.1), `price[7]` is at $2000 + 7 \cdot 8 = 2056$. By Equation (6.5) with $C = 6$ and $s = 4$, `t[2][5]` is at $9000 + (2 \cdot 6 + 5) \cdot 4 = 9000 + 68 = 9068$.

3. Two indexes start at the two ends and move toward each other, swapping, until they meet or cross (**Listing 6.18**, **Listing 6.19**). In an array of odd length the middle element stays where it is.

Listing 6.18 `ex6_reverse.c`: reversing an array in place with two indexes.

```c
/* ex6_reverse.c - Exercise 6.3: reverse an array in place */
#include <stdio.h>

void reverse(long a[], int n)
{
    for (int i = 0, j = n - 1; i < j; i++, j--) {   /* two indexes meet */
        long t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}

int main(void)
{
    long odd[5] = {1, 2, 3, 4, 5};
    long even[4] = {10, 20, 30, 40};
    reverse(odd, 5);
    reverse(even, 4);
    for (int i = 0; i < 5; i++)
        printf("%ld ", odd[i]);
    printf("\n");
    for (int i = 0; i < 4; i++)
        printf("%ld ", even[i]);
    printf("\n");
    return 0;
}
```

Listing 6.19 `ex6_reverse.py`: the same in Python (idiomatic Python would write `a.reverse()`).

```python
# ex6_reverse.py - Exercise 6.3: reverse a list in place


def reverse(a):
    i, j = 0, len(a) - 1
    while i < j:                          # two indexes meet
        a[i], a[j] = a[j], a[i]
        i += 1
        j -= 1


def main():
    odd = [1, 2, 3, 4, 5]
    even = [10, 20, 30, 40]
    reverse(odd)
    reverse(even)
    print("".join(f"{x} " for x in odd))
    print("".join(f"{x} " for x in even))


if __name__ == "__main__":
    main()
```

```
5 4 3 2 1
40 30 20 10
```

4. The function mirrors `index_of_max` with `<` instead of `>`. For `n = 0` there is no smallest element, so no index is correct; return -1, the same convention as the searches, and let the caller test for it before using the result as an index: `int k = index_of_min(a, n); if (k >= 0) printf("%ld\n", a[k]);`.

5. Listing 6.20 and **Listing 6.21**. The `while` loop shifts every bigger element one place right and stops at the first element that is not bigger, where the picked-up value is written.

Listing 6.20 `ex6_insertion.c`: insertion sort with a trace after each insertion.

```c
/* ex6_insertion.c - Exercise 6.5: insertion sort, the card player's sort */
#include <stdio.h>

void insertion_sort(long a[], int n)
{
    for (int i = 1; i < n; i++) {
        long key = a[i];                /* the card we pick up */
        int j = i - 1;
        while (j >= 0 && a[j] > key) {  /* shift bigger cards to the right */
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
        printf("after i=%d:", i);
        for (int k = 0; k < n; k++)
            printf(" %ld", a[k]);
        printf("\n");
    }
}

int main(void)
{
    long bill[6] = {45000, 21000, 70000, 28000, 136500, 30000};
    insertion_sort(bill, 6);
    return 0;
}
```

Listing 6.21 `ex6_insertion.py`: the same in Python.

```python
# ex6_insertion.py - Exercise 6.5: insertion sort, the card player's sort


def insertion_sort(a):
    for i in range(1, len(a)):
        key = a[i]                      # the card we pick up
        j = i - 1
        while j >= 0 and a[j] > key:    # shift bigger cards to the right
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
        print(f"after i={i}:" + "".join(f" {x}" for x in a))


def main():
    insertion_sort([45000, 21000, 70000, 28000, 136500, 30000])


if __name__ == "__main__":
    main()
```

```
after i=1: 21000 45000 70000 28000 136500 30000
after i=2: 21000 45000 70000 28000 136500 30000
after i=3: 21000 28000 45000 70000 136500 30000
after i=4: 21000 28000 45000 70000 136500 30000
after i=5: 21000 28000 30000 45000 70000 136500
```

On an array that is already sorted, the `while` condition fails at once for every `i`, so insertion sort makes only $n - 1$ comparisons. This is its great advantage over bubble sort without early exit, and the reason library sorts use insertion sort for short or nearly sorted pieces.

6. Bubble sort: $n(n-1)/2$ = 45, 499,500 and 499,999,500,000. Linear search: 10, 1000 and 1,000,000 looks. Binary search, $\lfloor \log_2 n \rfloor + 1$: 4, 10 and 20 looks.

7. Fill `order[i] = i`, then run bubble sort on `order`, comparing `bill[order[i]] < bill[order[i + 1]]` and swapping the two *indexes* when they are in the wrong order. Afterward `order[0]` is the

position of the biggest bill, and `service[order[0]]` and `weight[order[0]]` still describe the same order, because none of the parallel arrays moved. This is how a report can list the orders by size and still print their receipt numbers, which are `order[i] + 1`.

References

[1] ISO/IEC 9899:2018, *Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, 2018.

[2] Carnegie Mellon University Software Engineering Institute, "ARR30-C. Do not form or use out-of-bounds pointers or array subscripts," in *SEI CERT C Coding Standard*. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/ARR30-C.+Do+not+form+or+use+out-of-bounds+pointers+or+array+subscripts

[3] Free Software Foundation, "Program instrumentation options," in *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[4] Python Software Foundation, "Data structures: more on lists," in *The Python Tutorial*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[6] O. Astrachan, "Bubble sort: an archaeological algorithmic analysis," in *Proc. 34th SIGCSE Technical Symposium on Computer Science Education*, Reno, NV, USA, 2003, pp. 1–5.

[7] Python Software Foundation, "Sorting techniques," Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/howto/sorting.html

[8] J. Bloch, "Extra, extra - read all about it: nearly all binary searches and mergesorts are broken," Google Research Blog, Jun. 2, 2006. [Online]. Available: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[10] Python Software Foundation, "How do I create a multidimensional list?," in *Programming FAQ*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/faq/programming.html#how-do-i-create-a-multidimensional-list

Chapter 7
Pointers: Addresses as Values

Chapter 5 left two promises open. A function can give back only one value, yet the owner of a shop asks for four at closing time. And a function receives copies of its arguments, yet `scanf` somehow fills the caller's variables, and the array functions of Chapter 6 somehow changed the caller's array. Both mysteries have the same answer: an *address*, the number that says where a variable lives in memory, is itself a value that can be stored in a variable, passed to a function and followed back to the variable it names. Such a variable is a **pointer**, and it is the idea that gives C its power and its reputation. This chapter takes pointers slowly, with pictures. By the end of the chapter you will be able to explain what `&` and `*` do and draw the memory of a small program with its arrows; write functions that change their caller's variables, such as `swap`; return several answers from one call through output parameters, and a status that says whether the answers are valid; walk through arrays with pointers, compute with pointer arithmetic and explain why `a[i]` and `*(a + i)` are the same thing; read and write `const` in pointer declarations; pass functions to functions, and use the library's `qsort` with a comparison function; recognize the classic pointer mistakes (the null pointer, the uninitialized pointer, the pointer to a variable that no longer exists) and catch them with the sanitizer; say what Python does instead; and build version 6 of the Profitina Laundry cashier, whose closing summary gets the total, the average, the smallest and the biggest bill from a single call.

7.1 One Call, Three Answers

At closing time Mbak Rina asks the cashier for a summary: how many orders, how many kilograms were washed, the revenue, the average bill, the smallest and the biggest. Dimas wants one function, `day_stats`, to walk once through the day's bills and produce all of it. A function in C returns one value. He could write four functions, `total`, `average`, `smallest` and `biggest`, each walking through the array again, and for fifty orders nobody would notice the waste. But the problem returns in forms where four functions are not a way out: splitting a weight such as 3.45 kg into 3 kg and 450 g gives two numbers that belong together, and a function that reads an order from the keyboard must give back a service, a weight and an express flag, and also say whether the reading succeeded at all.

The second problem is on a shelf. Two bags wait for pickup, and Dimas's program keeps their weights in two variables, `left` and `right`. To keep the lighter bag on the left, he wants a function `swap(left, right)`. Chapter 5 already proved that this cannot work with values: the function receives copies, swaps the copies, and the shelf does not change.

Think about how the laundry itself solves a related problem. When a customer leaves a bag, Mbak Rina does not hand over the bag's contents to the helper who will wash it. She hands over a *tag* with a shelf number: "shelf 12". The helper walks to shelf 12 and works on the bag that is there, and when Mbak Rina looks at shelf 12 later, she sees the washed bag. The tag is small, easy to copy and easy to pass on, and it lets anybody who holds it reach the one real bag. A pointer is exactly such a tag: a small value that says *where* something is, so that a function holding it can reach the caller's real variable instead of a copy.

Profitina Laundry has used such tags for years without calling them pointers. The receipt that the customer takes home carries a number, and that number leads to exactly one bag on exactly one shelf. When the customer comes back, nobody searches the whole shop: the number says where to look. If the receipt is lost, the bag still exists, but nobody can find it quickly; if two receipts carry the same number by mistake, two customers will claim one bag. Every one of these everyday situations has a twin in programs with pointers, and this chapter meets them all.

7.2 The Idea in Plain Words

7.2.1 Every variable has an address

The memory of a computer is a long row of numbered bytes, like a very long row of numbered lockers at a market. Every variable occupies some consecutive lockers: an `int` usually four, a `double` eight. The number of the first locker is the variable's **address**. The operator `&` (read "address of") gives it: if `kg` is an `int` variable, `&kg` is the number of the locker where `kg` begins. You have written `&` since your first `scanf("%d", &kg)`; now you know what you handed to `scanf`: not the value of `kg`, which `scanf` does not need, but the place where `kg` lives, which is exactly what `scanf` needs in order to put a number there.

> **DEFINITION.** A *pointer* is a variable whose value is the address of another variable. If `p` holds the address of `kg`, we say that `p` *points to* `kg`. The declaration `int *p;` makes `p` a pointer to an `int`. The operator `&` gives the address of a variable; the operator `*`, applied to a pointer, gives the variable it points to, a step called *dereferencing* or *following* the pointer.

7.2.2 Following the arrow

Figure 7.1 draws the memory of three variables. `kg` lives at address 1000 and holds 7; `other` lives at 1004 and holds 12; `p` lives at 1008 and holds the number 1000, the address of `kg`. We draw that number as an arrow from `p` to `kg`, because an arrow is what it means. The addresses in the figure are small invented numbers; real addresses are much larger and change from one run to the next, but the picture is the same.

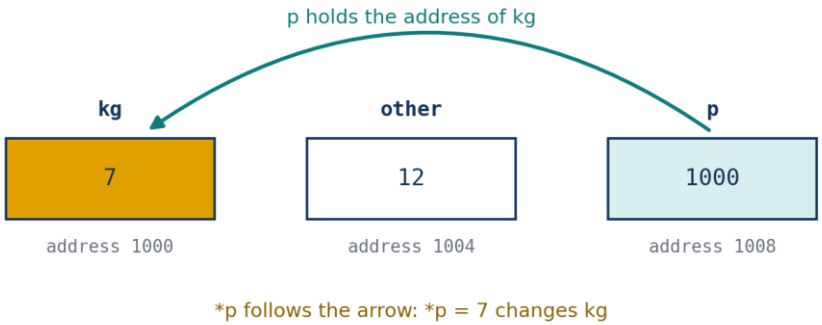


Figure 7.1 Three variables in memory: `p` holds the address of `kg`, drawn as an arrow; `*p` follows the arrow, so `*p = 7` changes `kg`.

With `p` pointing to `kg`, the expression `*p` *is* `kg`: reading `*p` reads `kg`, and assigning `*p = 7` writes 7 into `kg`. Written as a rule, for any variable `x`,

$$*(\&x) \equiv x \tag{7.1}$$

Equation (7.1) says that following the address of a variable leads back to the variable itself: `&` and `*` undo each other. The symbol ≡ means "is the same thing as", not merely "has the same value": `*(&x)` can be assigned to, exactly as `x` can.

The star appears in two places with two different jobs, which confuses everyone at first. In a **declaration**, `int *p;`, the star says "p is a pointer to int". In an **expression**, `*p = 7;`, the star says "follow p". A way to read the declaration that makes both agree: `int *p` says that `*p` is an `int`.

Practice the reading on a few declarations. `double *price;` says that `*price` is a `double`, so `price` is a pointer to a `double`. `long *a, b;` is a trap: the star belongs to `a` only, so `a` is a pointer to `long` and `b` is a plain `long`; write one declaration per line to avoid it. `int **pp;` says that `**pp` is an `int`, so `*pp` is a pointer to `int` and `pp` is a pointer to a pointer, which Chapter 10 needs for linked lists. And in `int *p = &kg;` the star belongs to the declaration: the value `&kg` goes into `p`, not into `*p`.

7.2.3 The type of a pointer, and the pointer to nothing

A pointer knows the type of what it points to: `int *` points to an `int`, `long *` to a `long`, `double *` to a `double`. The address alone would say only where the first byte is; the type says how many bytes belong to the variable and how to read them. The compiler refuses to put the address of a `double` into an `int *` without a cast, because following that pointer would read eight bytes of a `double` as if they were an `int`, which is nonsense.

A pointer may also point to nothing. The constant `NULL` (from `<stddef.h>`, also available through `<stdio.h>` and `<stdlib.h>`) is a pointer value guaranteed not to be the address of any variable. It is used to say "no variable here": a search that finds nothing may return `NULL`, and a function may refuse to work when it receives `NULL`. Following a null pointer is undefined behavior, and on desktop systems it crashes the program at once [1], [2]. The inventor of the null reference, Tony Hoare, later called it his "billion-dollar mistake", because of the countless crashes caused by programs that followed a null pointer without checking [3]. The rule is simple: before following a pointer that might be `NULL`, test it.

7.2.4 Pointers you have already used

You have met pointers since Chapter 1 without the name. The `&` in `scanf("%d", &kg)` produces a pointer to `kg`, and `scanf` writes the number it reads through that pointer. The text in `printf("Hello\n")` is an array of characters, and what `printf` receives is a pointer to its first character. The array parameter `const long a[]` of Chapter 6 is a pointer, which is why the functions of that chapter could change the caller's array and why they could not know its length. This chapter does not add a strange new feature to C; it explains the feature that has been working under the surface all along, and gives you the words and the pictures to use it on purpose.

7.3 Your First Pointer

Listing 7.1 performs everything in **Figure 7.1**. It stores the address of `kg` in `p`, writes through `p`, aims `p` at another variable, copies the pointer into `q`, and finally declares a pointer that points nowhere.

Listing 7.1 `first_pointer.c`: taking an address, following it, writing through it, re-aiming the pointer, copying it, and `NULL`.

```c
/* first_pointer.c - an address is a value you can store, copy and follow */
#include <stdio.h>

int main(void)
{
    int kg = 5;
    int *p = &kg;                       /* p holds the address of kg */
    printf("kg = %d, *p = %d\n", kg, *p);

    *p = 7;                             /* write THROUGH the pointer */
    printf("after *p = 7: kg = %d\n", kg);

    int other = 12;
    p = &other;                         /* aim p at another variable */
    *p += 1;
    printf("other = %d, kg = %d\n", other, kg);
    printf("p points to other: %s\n", p == &other ? "yes" : "no");

    int *q = p;                         /* copying a pointer: two arrows */
    *q = 20;
    printf("after *q = 20: other = %d, *p = %d\n", other, *p);

    int *nowhere = NULL;                /* a pointer that points at nothing */
    if (nowhere == NULL)
        printf("nowhere is NULL: never follow it\n");
    return 0;
}
```

```
kg = 5, *p = 5
after *p = 7: kg = 7
other = 13, kg = 7
p points to other: yes
after *q = 20: other = 20, *p = 20
nowhere is NULL: never follow it
```

Follow the program in **Figure 7.1** line by line. After `int *p = &kg;` the arrow goes from `p` to `kg`, and `*p` reads 5. After `*p = 7;` the value in `kg` is 7, although the statement never names `kg`: that is the whole point. Then `p = &other;` moves the arrow (without a star, the assignment changes `p` itself, not what it points to), and `*p += 1` makes `other` 13 while `kg` stays 7. Copying `q = p` makes a second arrow to the same variable; writing through `q` changes `other`, and reading through `p` sees the change. Two pointers to one variable are like two tags with the same shelf number: whoever uses either tag works on the same bag.

The program deliberately prints no addresses. `printf("%p\n", (void *) p);` would print the real address, a number like `0x7ffd5c3a1b2c`, but it would be different on your computer and on every run (modern systems place variables at random addresses as a security measure), and this book prints only output that you can reproduce exactly. Try it once yourself: the value is meaningless, and only its relation to other addresses matters, as Section 7.6 shows.

Kernighan and Ritchie, who designed C's pointers, warned that they "have been lumped with the goto statement as a marvelous way to create impossible-to-understand programs", and that with discipline they can instead be used to achieve clarity and simplicity [4]. The discipline is what the rest of this chapter teaches.

Python has no pointers, and deliberately so: a Python program cannot take the address of a variable or follow one [5]. What Python does have is *references*. Every Python name refers to an object, and two names can refer to the same object. When the object can be changed, as a list can, a change made through one name is seen through the other. **Listing 7.2** uses this to imitate the C

program: a list with one element plays the part of a variable in memory, and a second name for the same list plays the part of a pointer.

Listing 7.2 `first_pointer.py`: one-element lists as boxes that two names can share, which is the part of pointers that Python has.

```
# first_pointer.py - Python has no pointers; a one-element list plays a box
# that two names can share, which is the part of pointers Python does have.


def main():
    kg = [5]                            # a box that holds one number
    p = kg                              # p refers to the SAME box
    print(f"kg = {kg[0]}, *p = {p[0]}")

    p[0] = 7                            # write THROUGH the shared box
    print(f"after *p = 7: kg = {kg[0]}")

    other = [12]
    p = other                           # p now refers to another box
    p[0] += 1
    print(f"other = {other[0]}, kg = {kg[0]}")
    print(f"p points to other: {'yes' if p is other else 'no'}")

    q = p                               # copying a reference: two arrows
    q[0] = 20
    print(f"after *q = 20: other = {other[0]}, *p = {p[0]}")

    nowhere = None                      # Python's NULL
    if nowhere is None:
        print("nowhere is NULL: never follow it")


if __name__ == "__main__":
    main()
```

The output is identical, and the imitation teaches the essential idea: after `p = kg`, the names `p` and `kg` refer to *one* box, and `p is kg` is `True`. What Python cannot imitate is arithmetic on addresses, the subject of Section 7.6.

7.4 Pass-by-Reference: Changing the Caller's Variables

Now the shelf. **Listing 7.3** contains two functions. `swap_copies` receives two `int` values, the old way, and swaps its own copies. `swap` receives two *addresses*, `int *a` and `int *b`, and swaps what they point at. The call passes the addresses with `&`: `swap(&shelf_left, &shelf_right)`.

Listing 7.3 `swap.c`: swapping copies changes nothing in `main`; swapping through pointers changes `main`'s variables. Three compare-and-swap steps then sort three numbers.

```c
/* swap.c - why a function needs addresses to change the caller's variables */
#include <stdio.h>

void swap_copies(int a, int b)          /* swaps its own copies only */
{
    int t = a;
    a = b;
    b = t;
}

void swap(int *a, int *b)               /* swaps what the arrows point at */
{
    int t = *a;
    *a = *b;
    *b = t;
}

int main(void)
{
    int shelf_left = 9, shelf_right = 4;     /* kg of two bags on a shelf */
    swap_copies(shelf_left, shelf_right);
    printf("after swap_copies: left %d, right %d\n", shelf_left, shelf_right);
    swap(&shelf_left, &shelf_right);
    printf("after swap:        left %d, right %d\n", shelf_left, shelf_right);

    int bag[3] = {7, 3, 5};
    if (bag[0] > bag[1]) swap(&bag[0], &bag[1]);   /* three compare-swaps */
    if (bag[1] > bag[2]) swap(&bag[1], &bag[2]);   /* sort three numbers */
    if (bag[0] > bag[1]) swap(&bag[0], &bag[1]);
    printf("sorted bags: %d %d %d\n", bag[0], bag[1], bag[2]);
    return 0;
}
```

Figure 7.2 puts the two calls side by side. On the left, `swap_copies` has its own boxes `a` and `b`, filled with copies of 9 and 4; it swaps them to 4 and 9, and when it returns, its boxes vanish and `main` still has 9 and 4. On the right, the boxes of `swap` hold arrows into `main`. `int t = *a;` reads `main`'s `left` (9) into `t`; `*a = *b;` copies `main`'s `right` (4) into `main`'s `left`; `*b = t;` puts 9 into `main`'s `right`. When `swap` returns, its boxes vanish, but the work was done on `main`'s variables, and it stays done.

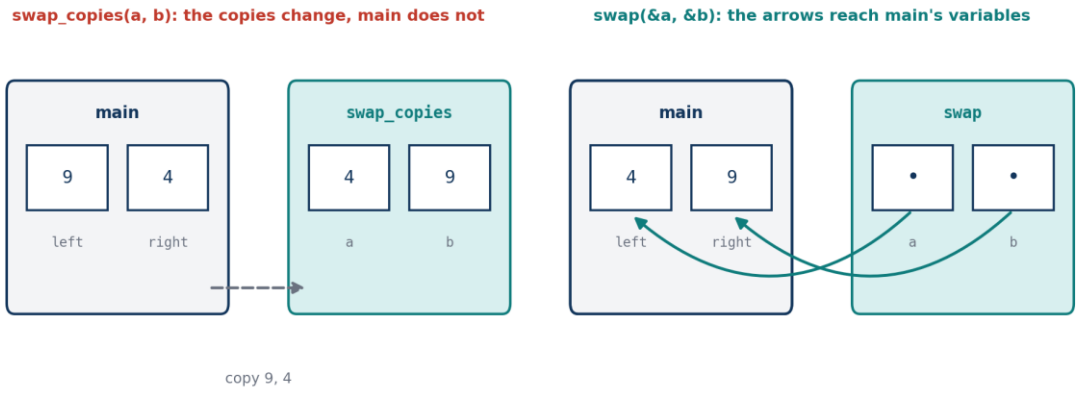


Figure 7.2 Left: `swap_copies` works on copies, and `main` keeps 9 and 4. Right: `swap` receives the addresses of `main`'s variables and exchanges the values there.

```
after swap_copies: left 9, right 4
after swap:        left 4, right 9
sorted bags: 3 5 7
```

Now `scanf` holds no more mystery. When you write `scanf("%d", &kg)`, you hand `scanf` the address of `kg`, and inside the library a statement equivalent to `*p = number;` puts the number that was typed into your variable. Forgetting the `&`, writing `scanf("%d", kg)`, hands `scanf` the *value* of `kg`, some random number, which `scanf` then uses as an address; the typed number is written into a random place in memory. `gcc -Wall` catches this particular mistake, because it knows what `scanf` expects.

This way of passing, where the function receives the address and can therefore change the caller's variable, is called **pass-by-reference**. Strictly speaking, C still passes by value: the function receives a *copy of the address*. But a copy of an address leads to the same variable, just as a photocopy of a shelf tag leads to the same shelf, and the effect is the reference that the caller wanted.

The last lines of `main` sort three bags with three calls of the kind `if (bag[0] > bag[1]) swap(&bag[0], &bag[1]);`. The address of an array element is taken with `&` like any other address. After the first two steps the biggest weight is at the end; the third puts the other two in order. This little *sorting network* is bubble sort for exactly three elements, written out.

Check the three steps by hand on `{7, 3, 5}`. Step one compares 7 and 3, swaps: `{3, 7, 5}`. Step two compares 7 and 5, swaps: `{3, 5, 7}`, and the biggest is at the end. Step three compares 3 and 5, no swap. The output says `3 5 7`. Try the other five orders of three numbers on paper; each ends sorted, which is a small proof that the three steps are enough.

Python cannot pass the address of a variable, so its `swap_copies` fails just as in C, and it offers two other ways, both shown in **Listing 7.4**. For two separate variables, the natural Python is the tuple assignment `left, right = right, left`, done in the caller. For elements of a list, a function can change the list itself, because it receives a reference to the same list: `swap_items(bag, 0, 1)`. The Python FAQ discusses exactly these substitutes for output parameters [6].

Listing 7.4 `swap.py`: Python cannot swap two variables of its caller; it swaps with a tuple assignment in the caller, or changes a shared list.

```
# swap.py - Python passes references to objects, but rebinding a parameter
# never reaches the caller; return the new values or change a shared list.


def swap_copies(a, b):                  # rebinds its own names only
    a, b = b, a


def swap_items(a, i, j):                # changes the caller's list in place
    a[i], a[j] = a[j], a[i]


def main():
    shelf_left, shelf_right = 9, 4
    swap_copies(shelf_left, shelf_right)
    print(f"after swap_copies: left {shelf_left}, right {shelf_right}")
    shelf_left, shelf_right = shelf_right, shelf_left   # the Python way
    print(f"after swap:        left {shelf_left}, right {shelf_right}")

    bag = [7, 3, 5]
    if bag[0] > bag[1]:
        swap_items(bag, 0, 1)
    if bag[1] > bag[2]:
        swap_items(bag, 1, 2)
    if bag[0] > bag[1]:
        swap_items(bag, 0, 1)
    print(f"sorted bags: {bag[0]} {bag[1]} {bag[2]}")


if __name__ == "__main__":
    main()
```

7.5 Output Parameters: One Call, Several Answers

With pointers, a function can give back as many results as it likes: the caller passes the addresses of variables that will receive the results, and the function writes into them. Such parameters are called **output parameters**. **Listing 7.5** has two functions of this kind.

`split_weight` turns a weight in kilograms into whole kilograms and grams, for the receipt of a customer who wants “3 kg 450 g” instead of “3.45 kg”. It first converts to whole grams, rounded to the nearest gram, and then divides. With w the weight in kilograms and g the whole number of grams,

$$g = \operatorname{round}(1000\,w), \qquad \text{kg} = \lfloor g/1000 \rfloor, \qquad \text{grams} = g \bmod 1000 \tag{7.2}$$

The rounding in Equation (7.2) is not decoration. The number 3.45 cannot be stored exactly in binary; the `double` closest to it is a tiny bit more, and 1000 times it is 3450.0000000000005. Converting that straight to an integer would happen to give 3450, but for other weights the error falls on the other side: 0.29 kg times 1000 is 289.99999999999994, which a plain conversion turns into 289 grams. `lround` from `<math.h>` rounds to the nearest whole number and removes the problem.

`day_stats` walks through the bills once and fills three answers: the total, the average and the biggest bill. It also *returns* a value, 1 or 0, which says whether the answers are valid. For an empty day there is no average (it would be a division by zero) and no biggest bill, so the function returns 0 and does not touch the answers. The caller tests the status before it uses the answers, as `main` shows.

Listing 7.5 `out_params.c`: `split_weight` gives two answers through pointers; `day_stats` gives three answers and returns whether they are valid.

```c
/* out_params.c - one call, several answers: output parameters */
#include <stdio.h>
#include <math.h>

void split_weight(double weight, int *kg, int *grams)
{
    long total_grams = lround(weight * 1000);
    *kg = (int) (total_grams / 1000);
    *grams = (int) (total_grams % 1000);
}

/* returns 1 and fills the three answers; returns 0 for an empty day */
int day_stats(const long *bill, int n, long *total, double *average,
              long *biggest)
{
    if (n <= 0)
        return 0;
    *total = 0;
    *biggest = bill[0];
    for (int i = 0; i < n; i++) {
        *total += bill[i];
        if (bill[i] > *biggest)
            *biggest = bill[i];
    }
    *average = (double) *total / n;
    return 1;
}

int main(void)
{
    int kg, grams;
    split_weight(3.45, &kg, &grams);
    printf("3.45 kg is %d kg and %d g\n", kg, grams);

    long bill[5] = {28000, 30000, 30000, 70000, 136500};
    long total, biggest;
    double average;
    if (day_stats(bill, 5, &total, &average, &biggest))
        printf("total %ld, average %.2f, biggest %ld\n", total, average, biggest);
    if (!day_stats(bill, 0, &total, &average, &biggest))
        printf("an empty day has no statistics\n");
    return 0;
}
```

```
3.45 kg is 3 kg and 450 g
total 294500, average 58900.00, biggest 136500
an empty day has no statistics
```

Trace the call `day_stats(bill, 5, &total, &average, &biggest)` through memory once. `main` owns five variables that matter here: the array `bill` and the three answer variables, which hold garbage because they were never given a value. The call creates the parameters of `day_stats`: `bill` receives the address of `main`'s array, `n` receives 5, and `total`, `average` and `biggest` receive the *addresses* of `main`'s three answer variables. Inside the function, the names `total`, `average` and `biggest` are pointers, and every statement that stores an answer does so through a star: `*total = 0;` writes zero into `main`'s `total`, `*total += bill[i];` adds into it, `*biggest = bill[i];` replaces `main`'s `biggest`. When the function returns 1, its parameters vanish, and `main` finds its three variables filled. Nothing was returned except the status; the answers were delivered to `main`'s door.

Notice the names. Inside `day_stats` the parameter is called `total` and is a pointer; inside `main` the variable is called `total` and is a `long`. They are different variables that happen to share a name, in different scopes (Chapter 5). Some programmers mark output parameters with a prefix such as `out_total` to make the difference visible; whichever style you choose, keep it consistent.

The pattern "results through pointers, success as the return value" is everywhere in C. `scanf` itself follows it: it writes the values it reads through the pointers it receives, and returns the number of values it managed to read, which is why this book always writes `scanf(...) == 1` in its loops.

Python, in **Listing 7.6**, does not need output parameters, because a function can return a *tuple*, several values in one, and the caller unpacks them: `kg, grams = split_weight(3.45)`. For "no valid answer", Python returns `None`, its version of `NULL`, and the caller tests `answer is not None`. The program also defines `c_lround`, because Python's `round` rounds halves to even (Chapter 5) while C's `lround` rounds them away from zero.

Listing 7.6 `out_params.py`: the Python way: return a tuple, or `None` when there is no valid answer.

```
# out_params.py - one call, several answers: Python returns a tuple


def split_weight(weight):
    total_grams = c_lround(weight * 1000)
    return total_grams // 1000, total_grams % 1000


def c_lround(x):                        # C's lround: halves away from zero
    return int(x + 0.5) if x >= 0 else -int(-x + 0.5)


def day_stats(bill):
    """Return (total, average, biggest), or None for an empty day."""
    if not bill:
        return None
    total, biggest = 0, bill[0]
    for b in bill:
        total += b
        if b > biggest:
            biggest = b
    return total, total / len(bill), biggest


def main():
    kg, grams = split_weight(3.45)
    print(f"3.45 kg is {kg} kg and {grams} g")

    bill = [28000, 30000, 30000, 70000, 136500]
    answer = day_stats(bill)
    if answer is not None:
        total, average, biggest = answer
        print(f"total {total}, average {average:.2f}, biggest {biggest}")
    if day_stats(bill[:0]) is None:
        print("an empty day has no statistics")


if __name__ == "__main__":
    main()
```

7.6 Arrays and Pointers: Two Notations, One Memory

7.6.1 The name of an array is the address of its first element

Chapter 6 said that an array passed to a function arrives as the address of its first element. Here is the full truth. In almost every expression, the name of an array is converted to a pointer to its first element: `bill` means `&bill[0]`. The exceptions are `sizeof bill`, which gives the size of the whole array, and `&bill`. This conversion is called *array-to-pointer decay*, and it is why `long *p = bill;` needs no `&`.

The exception for `sizeof` is where arrays and pointers visibly part ways. With `long bill[5]; long *p = bill;`, the expression `sizeof bill` is the size of the whole array, 40 bytes on a computer with 8-byte `long` values, while `sizeof p` is the size of an address, 8 bytes on a 64-bit computer whatever it points to. An array *is* its elements; a pointer only *knows where* they are. The two behave alike when you index or walk them, and differently when you ask how big they are or try to assign to them (`bill = p;` does not compile, `p = bill;` does).

7.6.2 Pointer arithmetic

Adding a whole number k to a pointer moves it k *elements*, not k bytes. If `p` points to `bill[0]`, then `p + 1` points to `bill[1]`, `p + 3` to `bill[3]`. The compiler multiplies by the size of the element for you:

$$\text{address}(p + k) = \text{address}(p) + k \cdot \text{sizeof}(*p) \tag{7.3}$$

Equation (7.3) is the address formula of Section 6.2.2 in a new form. It leads directly to the central identity of C: for an array or pointer `a` and an index `i`,

$$a[i] \equiv *(a + i) \tag{7.4}$$

The C standard in fact *defines* the brackets by Equation (7.4) [1]: `a[i]` is a shorthand for "move `i` elements from `a` and follow". **Figure 7.3** draws it on five bills. Because `p` holds the same address as `bill`, `p[2]` works as well as `bill[2]`; the brackets do not care whether they are given an array name or a pointer.

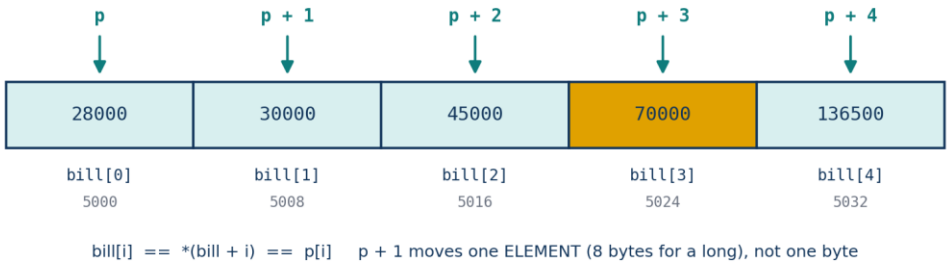


Figure 7.3 Pointer arithmetic: `p + k` points `k` elements further, whatever the size of an element, so `bill[i]`, `*(bill + i)` and `p[i]` name the same box.

Work one example with the numbers of **Figure 7.3**. The array starts at 5000 and a `long` takes 8 bytes. `p` holds 5000; by Equation (7.3), `p + 3` holds $5000 + 3 \cdot 8 = 5024$, the address of `bill[3]`, and `*(p + 3)` is 70000. Had the array been of `int`, four bytes each, `p + 3` would hold 5012, again the address of element 3. The program writes the same `p + 3` in both cases and the compiler supplies the right multiplier, which is why pointer code is written in elements and never in bytes.

Subtracting two pointers into the same array gives the number of elements between them, the opposite of Equation (7.3). If `p` points to `bill[0]` and `last` to `bill[4]`, then `last - p` is 4, whatever the size of a `long`. The type of the result is `ptrdiff_t`, printed with `%td`. Comparing two pointers into the same array with `<` tells which one comes first. Arithmetic and comparison are defined only within one array (and one position past its end); subtracting pointers into two different arrays is undefined [1].

7.6.3 Walking with a pointer

Listing 7.7 shows all of this in action. It reads `bill[2]` three ways, adds up the array by moving a pointer `q` from the first element to one past the last, measures the distance between two pointers, moves a pointer three elements forward and reads the element *before* it with `p[-1]`, and finally reverses a word in place with two pointers walking toward each other, the same two-index idea as Exercise 6.3.

Listing 7.7 `pointer_walk.c`: three names for one element, a pointer walk, pointer difference, a negative index from a moved pointer, and a word reversed with two pointers.

```
/* pointer_walk.c - arrays and pointers: two notations, one memory */
#include <stdio.h>
#include <string.h>

int main(void)
{
    long bill[5] = {28000, 30000, 45000, 70000, 136500};
    long *p = bill;                      /* the same as &bill[0] */

    printf("bill[2] = %ld, *(bill + 2) = %ld, p[2] = %ld\n",
           bill[2], *(bill + 2), p[2]);

    long sum = 0;
    for (long *q = bill; q < bill + 5; q++)   /* walk the array with a pointer */
        sum += *q;
    printf("sum by walking: %ld\n", sum);

    long *last = &bill[4];
    printf("last - p = %td elements apart\n", last - p);

    p += 3;                              /* move three ELEMENTS, not bytes */
    printf("after p += 3, *p = %ld and p[-1] = %ld\n", *p, p[-1]);

    char word[] = "laundry";             /* a string is an array of char */
    char *left = word;
    char *right = word + strlen(word) - 1;
    while (left < right) {               /* two pointers walk toward each other */
        char t = *left;
        *left = *right;
        *right = t;
        left++;
        right--;
    }
    printf("reversed: %s\n", word);
    return 0;
}
```

The loop `for (long *q = bill; q < bill + 5; q++)` deserves a close look, because it is the classic C idiom. `bill + 5` is the address just past the last element: a pointer may point there (it is how the end is marked), but it must never be followed. The loop continues while `q` is before the end, and `q++` moves one element at a time. The body reads the current element with `*q`. No index appears at all.

The word is stored as `char word[] = "laundry";`, an array of eight characters: the seven letters and the zero character `'\0'` that marks the end of every string in C. `strlen(word)` counts the letters before the zero, 7, so `word + strlen(word) - 1` points to the last letter, `y`. The two pointers swap the characters they point to and step toward each other until they meet in the middle. Chapter 8 uses exactly this walk to decide whether a word is a palindrome, and Chapter 9 is all about strings.

```
bill[2] = 45000, *(bill + 2) = 45000, p[2] = 45000
sum by walking: 309500
last - p = 4 elements apart
after p += 3, *p = 70000 and p[-1] = 45000
reversed: yrdnual
```

Python has no pointer arithmetic; its programs use indexes, which is what **Listing 7.8** does. Python strings cannot be changed in place, so the word is first turned into a list of characters, reversed, and joined back into a string.

Listing 7.8 `pointer_walk.py`: the same walk with indexes, which is how Python expresses what C does with pointers.

```
# pointer_walk.py - what C does with pointers, Python does with indexes


def main():
    bill = [28000, 30000, 45000, 70000, 136500]
    p = 0                                   # an index plays the pointer
    print(f"bill[2] = {bill[2]}, *(bill + 2) = {bill[0 + 2]}, p[2] = {bill[p + 2]}")

    total = 0
    for q in range(len(bill)):              # walk with an index
        total += bill[q]
    print(f"sum by walking: {total}")

    last = 4
    print(f"last - p = {last - p} elements apart")

    p += 3
    print(f"after p += 3, *p = {bill[p]} and p[-1] = {bill[p - 1]}")

    word = list("laundry")                  # Python strings cannot change: use a list
    left, right = 0, len(word) - 1
    while left < right:                     # two indexes walk toward each other
        word[left], word[right] = word[right], word[left]
        left += 1
        right -= 1
    print(f"reversed: {''.join(word)}")


if __name__ == "__main__":
    main()
```

7.6.4 Why pointers and not only indexes?

If `a[i]` and `*(a + i)` are the same, why have both? Partly history: early C compilers produced faster code for pointer walks than for indexes, and a generation of C code was written in that style; today's compilers produce the same code for both. The lasting reasons are different. A pointer can point into the *middle* of an array and be handed to a function as if the array started there: `total(bill + 3, 2)` adds up `bill[3]` and `bill[4]` with the unchanged function of Chapter 6. A pointer can point to a single variable as easily as to an array element, which is what `swap`

needs. And in Chapter 10 pointers connect separate pieces of memory into structures, such as the linked list, that arrays cannot express.

7.7 `const` and Pointers

A pointer involves two things that might be protected: the pointer itself and the variable it points to. The position of `const` says which, as **Table 7.1** shows. Read the declaration from right to left: `const long *p` is "p is a pointer to a long that is const".

Table 7.1 Where `const` stands in a pointer declaration decides what is protected: the variable pointed to, the pointer, or both.

Declaration	May change *p?	May change p?	Typical use
`long *p`	yes	yes	an output parameter; a walk that writes
`const long *p`	no	yes	a read-only parameter; a walk that only reads
`long *const p`	yes	no	a pointer fixed to one place
`const long *const p`	no	no	a fixed, read-only view

The second row is the one you will write most. The parameter `const long *bill` of `day_stats` promises that the function only reads the bills; the walk `for (const long *p = bill; ...)` inside it moves the pointer (allowed) but never writes through it (not allowed). The parameter `const long a[]` of Chapter 6 is exactly the same thing written with brackets: in a parameter list, `long a[]` and `long *a` mean the same.

7.8 Pointers to Functions: Behavior as a Parameter

Mbak Rina sometimes wants the bills from cheapest to dearest, sometimes the other way round, and her accountant wants the bills that are whole thousands first, because those were probably paid in cash without change. Three sorts that differ only in one comparison would be three almost identical functions. The better design writes the sort once and hands it the comparison as a parameter.

A function also lives at an address in memory, and C lets a program store and pass that address. The declaration `int (*before)(long, long)` makes `before` a **pointer to a function** that takes two `long` values and returns an `int`. The parentheses around `*before` are required; without them, `int *before(long, long)` would declare a function that returns a pointer. The name of a function, like the name of an array, turns into its address, so `sort_by(a, 6, cheaper_first)` passes the function `cheaper_first`, and inside `sort_by` the call `before(a[i + 1], a[i])` calls whichever function was passed.

Listing 7.9 writes the three rules as small functions, keeps them in an *array of function pointers*, `rule[3]`, and sorts the same six bills three times, once with each rule. Then it sorts them with the standard library's `qsort` [7].

Listing 7.9 `sort_by.c`: one bubble sort that asks a function which of two values comes first, three such functions in an array, and the library's `qsort` with its comparison function.

```c
/* sort_by.c - passing behavior as a parameter: pointers to functions */
#include <stdio.h>
#include <stdlib.h>

int cheaper_first(long a, long b) { return a < b; }
int dearer_first(long a, long b) { return a > b; }
int round_thousands_first(long a, long b)    /* whole thousands before others */
{
    return (a % 1000 == 0) && (b % 1000 != 0);
}

/* bubble sort that asks "before(x, y)?" instead of using < itself */
void sort_by(long a[], int n, int (*before)(long, long))
{
    for (int pass = 1; pass < n; pass++)
        for (int i = 0; i < n - pass; i++)
            if (before(a[i + 1], a[i])) {
                long t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
}

int compare_long(const void *x, const void *y)  /* the shape qsort wants */
{
    long a = *(const long *) x, b = *(const long *) y;
    return (a > b) - (a < b);            /* -1, 0 or 1; never a - b */
}

void print_all(const char *title, const long a[], int n)
{
    printf("%-18s", title);
    for (int i = 0; i < n; i++)
        printf(" %6ld", a[i]);
    printf("\n");
}

int main(void)
{
    long start[6] = {45500, 21000, 70000, 28500, 136500, 30000};
    long a[6];
    int (*rule[3])(long, long) = {cheaper_first, dearer_first,
                                  round_thousands_first};
    const char *name[3] = {"cheaper first:", "dearer first:", "round first:"};
```

Listing 7.9 (Continued) *sort_by.c: one bubble sort that asks a function which of two values comes first, three such functions in an array, and the library's qsort with its comparison function.*

```c
    print_all("as taken:", start, 6);
    for (int r = 0; r < 3; r++) {
        for (int i = 0; i < 6; i++)
            a[i] = start[i];
        sort_by(a, 6, rule[r]);
        print_all(name[r], a, 6);
    }
    for (int i = 0; i < 6; i++)
        a[i] = start[i];
    qsort(a, 6, sizeof a[0], compare_long);
    print_all("qsort:", a, 6);
    return 0;
}
```

```
as taken:           45500  21000  70000  28500 136500  30000
cheaper first:      21000  28500  30000  45500  70000 136500
dearer first:      136500  70000  45500  30000  28500  21000
round first:        21000  70000  30000  45500  28500 136500
qsort:              21000  28500  30000  45500  70000 136500
```

The rule “whole thousands first” gives an interesting result: 21000, 70000 and 30000 come first, *in the order in which they were taken*, then 45500, 28500 and 136500, also in their original order. The bubble sort of this listing swaps only when `before(a[i + 1], a[i])` is true, that is, only when the right value *must* come first, so values that the rule considers equal never change places. The sort is **stable**, as Chapter 6 defined, and the stable order is what gives this rule its meaning.

In a real cashier, the rule would be chosen by the user. With the array of function pointers, the choice is one index: read a number `r` from a menu, check that `0 <= r && r < 3` (an index from the keyboard is checked before use, as Chapter 6 taught), and call `sort_by(a, n, rule[r])`. Adding a fourth rule means writing one small function and adding it to the array; neither the sort nor the menu code changes. This is the same “one rule, one place” of Chapter 5, applied to behavior instead of prices.

`qsort` has a more general interface, because it sorts arrays of *any* type. It receives the address of the array, the number of elements, the size of one element and a comparison function whose parameters are `const void *`, pointers to “something” that the comparison function must convert to the real type. The comparison returns a negative number when the first element comes first, zero when the two are equal, and a positive number otherwise. For two numbers a and b that is the *sign* of their difference:

$$\mathrm{cmp}(a, b) = [a > b] - [a < b] \in \{-1,\ 0,\ 1\} \tag{7.5}$$

where $[\cdot]$ is 1 when the condition holds and 0 otherwise, exactly what C’s comparison operators produce. The tempting shortcut `return a - b;` computes the same sign when the numbers are small, and fails when they are large: for `int` values near two billion, $a - b$ overflows and the sign comes out wrong, and returning a `long` difference as an `int` cuts it. Equation (7.5) never overflows. Note also that `qsort` is not required to be stable [7].

How does `qsort` use the comparison function? It does not know the type of the elements, only their size, so it moves them as blocks of bytes, and whenever it needs to know which of two elements comes first, it calls the comparison function with the addresses of the two elements. `compare_long` converts each `const void *` back to `const long *`, follows it to get the two numbers, and answers with Equation (7.5). The same `qsort` therefore sorts arrays of `int`, of `double`, of strings or of the structures of Chapter 10: only the comparison function changes. This is the same idea as `sort_by`, a general algorithm with the specific decision handed in from outside, and it is how the standard libraries of most languages are designed.

In Python, functions are values like any other: `rule = [cheaper_first, dearer_first, round_thousands_first]` is an ordinary list, and `sort_by(a, rule[r])` passes a function without any special syntax, as **Listing 7.10** shows. The library sort is `sorted(a)`, which is stable; to sort by a rule it takes a `key` function (`sorted(a, reverse=True)` for dearest first, or `sorted(a, key=lambda b: b % 1000 != 0)` for whole thousands first), and a C-style comparison can be adapted with `functools.cmp_to_key` [8].

Listing 7.10 `sort_by.py`: functions passed as values in Python, and the library’s `sorted`.

```
# sort_by.py - passing behavior as a parameter: functions are values


def cheaper_first(a, b):
    return a < b


def dearer_first(a, b):
    return a > b


def round_thousands_first(a, b):         # whole thousands before others
    return a % 1000 == 0 and b % 1000 != 0


def sort_by(a, before):
    """Bubble sort that asks before(x, y) instead of using < itself."""
    n = len(a)
    for pass_no in range(1, n):
        for i in range(n - pass_no):
            if before(a[i + 1], a[i]):
                a[i], a[i + 1] = a[i + 1], a[i]


def print_all(title, a):
    print(f"{title:<18}" + "".join(f" {x:6d}" for x in a))


def main():
    start = [45500, 21000, 70000, 28500, 136500, 30000]
    rule = [cheaper_first, dearer_first, round_thousands_first]
    name = ["cheaper first:", "dearer first:", "round first:"]

    print_all("as taken:", start)
    for r in range(3):
        a = list(start)
        sort_by(a, rule[r])
        print_all(name[r], a)
    print_all("qsort:", sorted(start))   # the library sort, as C's qsort


if __name__ == "__main__":
    main()
```

7.9 Building the Cashier, Version 6

The closing summary of version 6 brings the chapter together. A new module, `stats`, holds the statistics with output parameters. **Listing 7.11** is its header.

Listing 7.11 `stats.h`: the promises of the statistics module.

```
/* stats.h - closing-time statistics with output parameters (Chapter 7) */
#ifndef STATS_H
#define STATS_H

void split_weight(double weight, int *kg, int *grams);
int day_stats(const long *bill, int n, long *total, double *average,
              long *smallest, long *biggest);
int compare_long(const void *x, const void *y);

#endif
```

In **Listing 7.12**, `day_stats` gives four answers, the total, the average, the smallest and the biggest bill, from one walk with a `const` pointer. It refuses a null pointer as well as an empty day. And it

writes the answers only at the end, after the whole walk: until then it works in local variables. This detail has a reason. If the caller happened to pass the address of the same variable for two answers, or if the walk were interrupted, answers written early could be half-finished or overwrite each other; computing in locals and writing once at the end means the caller sees either all four answers or none.

Listing 7.12 `stats.c`: `split_weight`, the four-answer `day_stats` with a pointer walk, and the comparison function for `qsort`.

```
/* stats.c - closing-time statistics with output parameters */
#include <math.h>
#include <stddef.h>
#include "stats.h"

void split_weight(double weight, int *kg, int *grams)
{
    long total_grams = lround(weight * 1000);
    *kg = (int) (total_grams / 1000);
    *grams = (int) (total_grams % 1000);
}

int day_stats(const long *bill, int n, long *total, double *average,
              long *smallest, long *biggest)
{
    if (bill == NULL || n <= 0)
        return 0;                       /* nothing to report: answers untouched */
    long sum = 0, lo = bill[0], hi = bill[0];
    for (const long *p = bill; p < bill + n; p++) {
        sum += *p;
        if (*p < lo)
            lo = *p;
        if (*p > hi)
            hi = *p;
    }
    *total = sum;                       /* write the answers only at the end */
    *average = (double) sum / n;
    *smallest = lo;
    *biggest = hi;
    return 1;
}

int compare_long(const void *x, const void *y)
{
    long a = *(const long *) x, b = *(const long *) y;
    return (a > b) - (a < b);
}
```

The program itself, **Listing 7.13**, reads the orders with a function `read_orders` that fills two arrays through pointer parameters and returns how many orders it stored. Then one call to `day_stats` produces the four statistics, `split_weight` turns the total weight into kilograms and grams, and `qsort` sorts the bills from cheapest to dearest, which are printed with a pointer walk in thousands of rupiah. It reuses `laundry.c` of Chapter 5 unchanged.

Listing 7.13 `closing.c`: the cashier, version 6: the closing summary from one call to `day_stats`.

```c
/* closing.c - Profitina Laundry cashier, version 6: the closing-time summary.
   Input: one order per line, "service weight express", ending with 0. */
#include <stdio.h>
#include <stdlib.h>
#include "laundry.h"
#include "stats.h"

#define MAX_ORDERS 100

int read_orders(long *bill, double *weight, int max)   /* returns how many */
{
    int n = 0, s, express;
    double w;
    while (n < max && scanf("%d", &s) == 1 && s != 0) {
        if (scanf("%lf %d", &w, &express) != 2)
            break;
        if (price_per_kg(s) == 0 || w <= 0 || w > MAX_KG)
            continue;
        bill[n] = bill_for(s, w, express);
        weight[n] = w;
        n++;
    }
    return n;
}

int main(void)
{
    long bill[MAX_ORDERS], total = 0, smallest = 0, biggest = 0;
    double weight[MAX_ORDERS], average = 0, kg_sum = 0;
    int n = read_orders(bill, weight, MAX_ORDERS);

    printf("PROFITINA LAUNDRY - closing summary (v6)\n");
    if (!day_stats(bill, n, &total, &average, &smallest, &biggest)) {
        printf("no orders today\n");
        return 0;
    }
    for (int i = 0; i < n; i++)
        kg_sum += weight[i];
    int kg, grams;
    split_weight(kg_sum, &kg, &grams);
```

Listing 7.13 (Continued) *closing.c: the cashier, version 6: the closing summary from one call to day_stats.*

```c
    printf("orders   : %d\n", n);
    printf("washed   : %d kg %d g\n", kg, grams);
    printf("revenue  : ");
    print_rupiah(total);
    printf("\naverage  : ");
    print_rupiah((long) (average + 0.5));
    printf("\nsmallest : ");
    print_rupiah(smallest);
    printf("\nbiggest  : ");
    print_rupiah(biggest);

    qsort(bill, (size_t) n, sizeof bill[0], compare_long);
    printf("\ncheapest first:");
    for (long *p = bill; p < bill + n; p++)
        printf(" %ld", *p / 1000);
    printf(" (thousands)\n");
    return 0;
}
```

Look at the header of `read_orders`: `int read_orders(long *bill, double *weight, int max)`. The first two parameters are written as pointers, where Chapter 6 would have written `long bill[]` and `double weight[]`. In a parameter list the two spellings mean exactly the same, and

the choice is a message to the reader: brackets say "this is an array", a star says "this is an address". Both are true here. Inside, `bill[n] = ...` uses the brackets on the pointer, by Equation (7.4). The third parameter, `max`, is the capacity, which the function needs because a pointer cannot tell how many elements lie behind it; without it, a busy day would make `read_orders` write past the end of `main`'s arrays, and no compiler would notice.

The Python module, **Listing 7.14**, returns tuples; `day_stats` returns `None` for an empty day or for `None` instead of a list.

Listing 7.14 `stats.py`: the statistics module in Python, returning tuples.

```
# stats.py - closing-time statistics; Python returns several values at once


def c_lround(x):                          # C's lround: halves away from zero
    return int(x + 0.5) if x >= 0 else -int(-x + 0.5)


def split_weight(weight):
    total_grams = c_lround(weight * 1000)
    return total_grams // 1000, total_grams % 1000


def day_stats(bill):
    """Return (total, average, smallest, biggest), or None when empty."""
    if not bill:
        return None
    total, lo, hi = 0, bill[0], bill[0]
    for b in bill:
        total += b
        lo = min(lo, b)
        hi = max(hi, b)
    return total, total / len(bill), lo, hi
```

Listing 7.15 `closing.py`: the cashier, version 6, in Python.

```
# closing.py - Profitina Laundry cashier, version 6: the closing-time summary.
# Input: one order per line, "service weight express", ending with 0.
import sys
from laundry import MAX_KG, price_per_kg, bill_for, rupiah
from stats import split_weight, day_stats

MAX_ORDERS = 100


def read_orders(words, limit):
    """Return (bills, weights) of the valid orders."""
    bill, weight, pos = [], [], 0
    while (len(bill) < limit and pos < len(words) and int(words[pos]) != 0
           and pos + 2 < len(words)):
        s, w = int(words[pos]), float(words[pos + 1])
        express = int(words[pos + 2])
        pos += 3
        if price_per_kg(s) == 0 or w <= 0 or w > MAX_KG:
            continue
        bill.append(bill_for(s, w, express))
        weight.append(w)
    return bill, weight


def main():
    bill, weight = read_orders(sys.stdin.read().split(), MAX_ORDERS)
    print("PROFITINA LAUNDRY - closing summary (v6)")
    answer = day_stats(bill)
    if answer is None:
        print("no orders today")
        return
    total, average, smallest, biggest = answer
    kg_sum = 0.0
    for w in weight:
        kg_sum += w
    kg, grams = split_weight(kg_sum)

    print(f"orders   : {len(bill)}")
    print(f"washed   : {kg} kg {grams} g")
    print(f"revenue  : {rupiah(total)}")
    print(f"average  : {rupiah(int(average + 0.5))}")
    print(f"smallest : {rupiah(smallest)}")
    print(f"biggest  : {rupiah(biggest)}")
    print("cheapest first:" + "".join(f" {b // 1000}" for b in sorted(bill))
          + " (thousands)")
```

Listing 7.15 (Continued) *closing.py: the cashier, version 6, in Python.*

```
if __name__ == "__main__":
    main()
```

Listing 7.15 is the Python program; `read_orders` returns the two lists as a tuple, and `sorted(bill)` gives a sorted copy, leaving `bill` in receipt order.

7.10 Running It

The same Saturday as in Chapter 6:

```
1 3.4 0
2 1.2 1
3 5 0
2 6.5 0
1 12.75 1
1 4.2 0
3 2.5 0
2 3 1
1 7.9 0
7 3 0
3 1 0
1 5.5 1
0
```

```
PROFITINA LAUNDRY - closing summary (v6)
orders   : 11
washed   : 52 kg 950 g
revenue  : Rp 523.500
average  : Rp 47.591
smallest : Rp 12.000
biggest  : Rp 136.500
cheapest first: 12 18 28 30 30 35 45 56 63 70 136 (thousands)
```

The eleven valid orders weigh 33.75 + 10.7 + 8.5 = 52.95 kg, shown as 52 kg 950 g. The revenue, Rp 523,500, is the same as in Chapter 6, as it must be. The average is now Rp 47,591 where Chapter 6 printed Rp 47,590: the exact average is 47590.909..., and this version rounds it to the nearest rupiah with `(long) (average + 0.5)`, while Chapter 6 cut off the fraction with whole-number division. Neither is wrong, but a shop should choose one rule and keep it, or two reports of the same day will disagree by a rupiah and cost an evening of searching. The smallest bill is Rp 12,000, the iron-only order of 1 kg charged at the minimum of 2 kg, and the biggest is Rp 136,500. The last line lists the bills sorted by `qsort`, in thousands.

Check the sorted line against the list worked out by hand in Section 6.10, read from the other end: 12, 18, 28, 30, 30, 35, 45, 56, 63, 70 and 136 thousand. The two 30s are the orders of wash-and-iron 1.2 kg express and iron-only 5 kg; `qsort` promises nothing about which comes first, and for this report it does not matter, because only the amounts are printed. Whole-number division by 1000 prints 136500 as 136, which is why the line says "(thousands)": a label that tells the reader what was cut off is part of an honest report.

7.11 Testing It

Listing 7.16 tests the edges of the module. `split_weight(0.9996, ...)` must round up to 1 kg and 0 g, not report 0 kg and 1000 g. `day_stats` must give the right four answers for four small bills, and must return 0 for an empty day *and leave the answers untouched*: the test first fills `total` with the right value, calls `day_stats` with `n` = 0, and checks that `total` still holds it. A null pointer must be refused. The test ends with `qsort` and `compare_long`.

Listing 7.16 `test_stats.c`: fourteen checks, including rounding at a kilogram boundary, an empty day, a null pointer and the promise that refused calls leave the answers untouched.

```c
/* test_stats.c - checks split_weight and day_stats, including the edge cases */
#include <stdio.h>
#include <stdlib.h>
#include "stats.h"

static int checks = 0, failures = 0;

static void check(long got, long expected, int line)
{
    checks++;
    if (got != expected) {
        failures++;
        printf("FAILED on line %d: got %ld, expected %ld\n", line, got, expected);
    }
}

int main(void)
{
    int kg = -1, g = -1;
    split_weight(3.45, &kg, &g);
    check(kg, 3, __LINE__);
    check(g, 450, __LINE__);
    split_weight(0.9996, &kg, &g);          /* rounds up to a whole kg */
    check(kg, 1, __LINE__);
    check(g, 0, __LINE__);

    long bill[4] = {30, 10, 50, 20}, total = -1, lo = -1, hi = -1;
    double avg = -1;
    check(day_stats(bill, 4, &total, &avg, &lo, &hi), 1, __LINE__);
    check(total, 110, __LINE__);
    check((long) (avg * 100), 2750, __LINE__);
    check(lo, 10, __LINE__);
    check(hi, 50, __LINE__);
    check(day_stats(bill, 0, &total, &avg, &lo, &hi), 0, __LINE__);
    check(total, 110, __LINE__);            /* untouched after a refusal */
    check(day_stats(NULL, 3, &total, &avg, &lo, &hi), 0, __LINE__);

    qsort(bill, 4, sizeof bill[0], compare_long);
    check(bill[0], 10, __LINE__);
    check(bill[3], 50, __LINE__);
    printf("%d checks, %d failures\n", checks, failures);
    return failures != 0;
}
```

The average is checked as `(long) (avg * 100)` against 2750, because comparing two `double` values for exact equality is fragile; the average 27.5 is exact in binary, but in general a test of a computed `double` should compare with a small tolerance. The Python test, **Listing 7.17**, checks the same facts; there, "untouched" means that the caller's own variables do not change when `day_stats` returns `None`.

Listing 7.17 `test_stats.py`: the same checks in Python.

```
# test_stats.py - checks split_weight and day_stats, including the edge cases
import inspect
import sys
from stats import split_weight, day_stats

checks = 0
failures = 0


def check(got, expected):
    global checks, failures
    checks += 1
    if got != expected:
        failures += 1
        line = inspect.stack()[1].lineno
        print(f"FAILED on line {line}: got {got}, expected {expected}")


def main():
    kg, g = split_weight(3.45)
    check(kg, 3)
    check(g, 450)
    kg, g = split_weight(0.9996)         # rounds up to a whole kg
    check(kg, 1)
    check(g, 0)

    bill = [30, 10, 50, 20]
    answer = day_stats(bill)
    check(answer is not None, True)
    total, avg, lo, hi = answer
    check(total, 110)
    check(int(avg * 100), 2750)
    check(lo, 10)
    check(hi, 50)
    check(day_stats([]), None)
    check(total, 110)                    # our own variables are untouched
    check(day_stats(None), None)

    bill.sort()
    check(bill[0], 10)
    check(bill[3], 50)
    print(f"{checks} checks, {failures} failures")
    return 1 if failures else 0


if __name__ == "__main__":
    sys.exit(main())
```

Listing 7.17 (Continued) *test_stats.py: the same checks in Python.*

```
14 checks, 0 failures
```

7.12 Making It Better: Pointer Safety

Pointers give a program the power to reach anywhere in its memory, and that includes places it should not reach. Four rules keep the power under control.

Initialize every pointer. `int *p;` inside a function holds a random address; `*p = 5;` then writes into a random place. Give every pointer a value at once, the address of a real variable or `NULL`.

Test before following. A function that receives a pointer from outside, like `day_stats`, should decide what a `NULL` means and test for it. The SEI CERT C rule EXP34-C, "Do not dereference null pointers", exists because this one line of testing is so often forgotten [2].

Never return the address of a local variable. A function's local variables vanish when it returns (Chapter 5). A pointer to one of them points to memory that will be reused by the next call, a *dangling pointer*:

```
long *biggest_bill_wrong(const long *bill, int n)
{
    long best = bill[0];
    for (int i = 1; i < n; i++)
        if (bill[i] > best)
            best = bill[i];
    return &best;          /* WRONG: best disappears at the return */
}
```

`gcc` warns *function returns address of local variable*, and the SEI CERT rule DCL30-C forbids it [9]. Return the value, or a pointer into the caller's array (`return &bill[k];` is fine, because the caller's array still exists), or write the answer through an output parameter.

Know who owns the memory. Every pointer points into memory that belongs to somebody: `main`'s local variables, a caller's array, a string literal, or (from Chapter 10) a block that the program asked for. A pointer is safe to follow only while the owner still exists and still allows it. A function that receives a pointer *borrows* the memory for the duration of the call; it must not keep the pointer in a global variable to use later, because by then the owner may be gone. Asking "whose memory is this, and how long will it live?" for every pointer you write prevents most pointer bugs before they are typed.

Develop with the sanitizer. Every program of this book is also compiled with `-fsanitize=address,undefined`, which makes the running program check every memory access and stop with a report that names the line of a null dereference, an out-of-bounds access or the use of a dangling pointer [10]. A bug that silently corrupts memory in a normal build becomes a loud, precise error message. Use these flags every time you write pointer code, and remove them only for the version you hand to the user.

7.13 Using It for Real

The shapes of this chapter appear in every program that keeps accounts, whatever the business.

- **Output parameters with a status.** A market stall's scale reads a weight: `int read_weight(double *kg)` returns 1 and fills `*kg`, or returns 0 when the scale sends nothing. The caller never uses a weight that was not read.
- **Several statistics from one pass.** A fruit and durian orchard computes, from one walk over the week's harvest, the total, the average per tree, the weakest and the strongest tree, with the `day_stats` pattern unchanged.
- **Sorting by a rule chosen at run time.** A photocopy kiosk sorts its orders by price for the owner, by time for the queue and by customer name for the monthly bill: one sort, three comparison functions, and the rule chosen from a menu with an array of function pointers.
- **A pointer into the middle.** A warung's sales of a month are in one array of 31 days; `total(sales + 7, 7)` is the second week, without copying anything.

- **Two pointers toward each other.** A tailor checking whether a strip of fabric has a symmetric pattern, or a clerk checking whether a number plate reads the same from both ends, uses the two-pointer walk of `pointer_walk.c`; Chapter 8 turns it into a palindrome checker.
- **Splitting quantities.** `split_weight` becomes `split_duration(int minutes, int *hours, int *mins)` for a rental of motorcycles, or `split_money(long rupiah, long *notes_100k, long *rest)` for a cash drawer (Exercise 7.4).

7.14 Common Mistakes, and How to Avoid Them

- **Forgetting `&` in the call.** `swap(left, right)` passes values where addresses are expected; `gcc` warns *passing argument 1 makes pointer from integer without a cast*. Treat that warning as an error.
- **Forgetting `*` in the function.** `a = b;` inside `swap` exchanges the local copies of the addresses and changes nothing; `*a = *b;` changes the variables.
- **An uninitialized pointer.** `int *p; *p = 5;` writes to a random address. Initialize at the declaration.
- **Following `NULL`.** Test pointers that may be null before using them.
- **Returning the address of a local.** The variable dies at the return; the pointer dangles. Return a value or use an output parameter.
- **Confusing the two stars.** `int *p = &x;` declares and initializes `p`; later, `*p = 7;` writes to `x`. `p = 7;` would put the number 7 into the pointer (and draw a warning).
- **Pointer arithmetic in bytes.** `p + 1` moves one element, not one byte; do not multiply by `sizeof` yourself.
- **Walking past the end.** A pointer may point one past the last element, never be followed there. Loop with `q < a + n`.
- **`sizeof` on a pointer.** `sizeof p` is the size of an address, not of the array it points into; the length must travel as a separate parameter.
- **`a - b` in a comparison function.** It overflows for large values; use Equation (7.5).
- **Declaring two pointers in one line.** `long *a, b;` makes only `a` a pointer. Write `long *a; long *b;`.
- **Comparing pointers instead of values.** `if (p == q)` asks whether two pointers point to the same place; `if (*p == *q)` asks whether the two values are equal. Both are useful, and they are not the same question.
- **Expecting Python to change a caller's number.** Rebinding a parameter never reaches the caller. Return the new value, return a tuple, or change a shared list.

7.15 Chapter Summary

- Every variable has an address; `&x` gives it, and a pointer is a variable that stores it. `*p` follows the pointer to the variable, so `*(&x)` is `x`, Equation (7.1). `NULL` points nowhere and must never be followed.
- Passing addresses lets a function change its caller's variables (pass-by-reference, built from pass-by-value of an address), as `swap` and `scanf` do.
- Output parameters give a function several answers; a returned status says whether the answers are valid. Python returns tuples and `None` instead.

- The name of an array turns into the address of its first element. Pointer arithmetic moves by elements, Equation (7.3), and `a[i]` is defined as `*(a + i)`, Equation (7.4). Subtracting pointers into one array counts the elements between them.
- `const` before the type protects the variable pointed to; `const` after the star protects the pointer, **Table 7.1**.
- A pointer to a function passes behavior as a parameter; `qsort` sorts any array with a comparison function that returns the sign of the comparison, Equation (7.5), never a raw difference.
- The cashier, version 6, gets four statistics from one call, splits the day's weight with Equation (7.2), and sorts the bills with `qsort`.
- Initialize pointers, test for `NULL`, never return the address of a local, and develop with `-fsanitize=address,undefined`.

7.16 Exercises

1. **Draw the memory.** For `int a = 3, b = 8; int *p = &a, *q = &b; *p = *q + 1; p = q; *p = 20;`, draw the boxes and arrows after each statement, and give the values of `a`, `b`, `*p` and `*q` at the end.
2. **Minimum and maximum.** Write `void min_max(const int *a, int n, int *lo, int *hi)` that finds both in one walk with a pointer. What should the function do when `n` is 0, and how will the caller know?
3. **Length and counting with pointers.** Write `size_t my_strlen(const char *s)` that walks to the terminating `'\0'` and returns the distance, and `int count_char(const char *s, char c)`. Test them on the empty string, on `"laundry"` and on `"cuci setrika express"`.
4. **Change in banknotes.** A customer pays Rp 200,000 for a bill of Rp 136,500. Write `int take_notes(long *remaining, long note)` that returns how many notes of value `note` fit into `*remaining` and reduces `*remaining` accordingly, and use it for the notes Rp 100,000, 50,000, 20,000, 10,000, 5,000, 2,000 and 1.000.
5. **Which are legal?** With `long bill[5]; long *p = bill; const long *c = bill; long *const k = bill;`, which of these compile, and what does each do: `p++`, `c++`, `k++`, `*p = 1`, `*c = 1`, `*k = 1`, `p = c`?
6. **Sort by a new rule.** Add a comparison function that puts bills of Rp 50,000 or more first (the big orders, for delivery planning) and otherwise keeps the original order. Why must the sort you give it be stable?
7. **Explain the crash.** A program declares `char *name;` and then calls `scanf("%s", name);`. It crashes on some computers and seems to work on others. Explain, and fix it with an array.

7.17 Solutions to the Exercises

1. After `*p = *q + 1;`, `a` is 9 (8 + 1) and `b` is 8. After `p = q;`, both pointers point to `b`, and `a` is no longer reachable through a pointer. After `*p = 20;`, `b` is 20. Final: `a` = 9, `b` = 20, `*p` = 20, `*q` = 20.

2. Walk with `const int *p` from `a` to `a + n`, starting both answers at `a[0]`. For `n` = 0 there is no minimum; like `day_stats`, return an `int` status instead of `void`: `int min_max(...)` returns 0 and leaves `*lo` and `*hi` untouched, and the caller tests the status. Chapter 8 solves a closely related problem in full.

3. **Listing 7.18** and **Listing 7.19** solve it; the C version walks a pointer `p` to the zero and returns `p - s`, the number of characters passed, by the pointer difference of Section 7.6.2. `count_char`

moves its own copy of the parameter `s`; the caller's pointer is not affected, because the pointer itself was passed by value.

Listing 7.18 `ex7_strlen.c`: string length by pointer difference, and counting a character with a pointer walk.

```c
/* ex7_strlen.c - Exercise 7.3: string length and counting, with pointers */
#include <stdio.h>

size_t my_strlen(const char *s)
{
    const char *p = s;
    while (*p != '\0')                  /* walk to the terminating zero */
        p++;
    return (size_t) (p - s);            /* distance = number of characters */
}

int count_char(const char *s, char c)
{
    int count = 0;
    for (; *s; s++)                     /* s is our own copy of the address */
        if (*s == c)
            count++;
    return count;
}

int main(void)
{
    const char *words[3] = {"", "laundry", "cuci setrika express"};
    for (int i = 0; i < 3; i++)
        printf("\"%s\": length %zu, %d letter e, %d spaces\n", words[i],
               my_strlen(words[i]), count_char(words[i], 'e'),
               count_char(words[i], ' '));
    return 0;
}
```

Listing 7.19 `ex7_strlen.py`: the same counts in Python, whose strings carry no terminating zero.

```python
# ex7_strlen.py - Exercise 7.3: string length and counting, walking by index


def my_strlen(s):
    n = 0
    for _ in s:                         # Python strings carry no '\0'
        n += 1
    return n


def count_char(s, c):
    count = 0
    for ch in s:
        if ch == c:
            count += 1
    return count


def main():
    for w in ["", "laundry", "cuci setrika express"]:
        print(f'"{w}": length {my_strlen(w)}, {count_char(w, "e")} letter e, '
              f'{count_char(w, " ")} spaces')


if __name__ == "__main__":
    main()
```

```
"": length 0, 0 letter e, 0 spaces
"laundry": length 7, 0 letter e, 0 spaces
"cuci setrika express": length 20, 3 letter e, 2 spaces
```

4. **Listing 7.20** passes the address of `change` to every call, so each call takes its notes from what the previous calls left. In Python, **Listing 7.21**, the function returns the count and the new remainder as a tuple.

Listing 7.20 `ex7_change.c`: change in banknotes, each call reducing the caller's remaining amount through a pointer.

```c
/* ex7_change.c - Exercise 7.4: change in banknotes; each call takes what it
   can from the remaining amount, through a pointer */
#include <stdio.h>

int take_notes(long *remaining, long note)
{
    int count = (int) (*remaining / note);
    *remaining -= count * note;          /* the caller's amount shrinks */
    return count;
}

int main(void)
{
    long notes[7] = {100000, 50000, 20000, 10000, 5000, 2000, 1000};
    long paid = 200000, bill = 136500;
    long change = paid - bill;
    printf("change for Rp %ld: Rp %ld\n", bill, change);
    for (int i = 0; i < 7; i++) {
        int k = take_notes(&change, notes[i]);
        if (k > 0)
            printf("  %d x %ld\n", k, notes[i]);
    }
    printf("left as coins: Rp %ld\n", change);
    return 0;
}
```

Listing 7.21 `ex7_change.py`: the same with a returned tuple.

```python
# ex7_change.py - Exercise 7.4: change in banknotes; each call returns the
# count AND the new remaining amount, since Python has no output parameters


def take_notes(remaining, note):
    count = remaining // note
    return count, remaining - count * note


def main():
    notes = [100000, 50000, 20000, 10000, 5000, 2000, 1000]
    paid, bill = 200000, 136500
    change = paid - bill
    print(f"change for Rp {bill}: Rp {change}")
    for note in notes:
        k, change = take_notes(change, note)
        if k > 0:
            print(f"  {k} x {note}")
    print(f"left as coins: Rp {change}")


if __name__ == "__main__":
    main()
```

```
change for Rp 136500: Rp 63500
  1 x 50000
  1 x 10000
  1 x 2000
  1 x 1000
left as coins: Rp 500
```

The change of Rp 63,500 is one note of 50.000, one of 10.000, one of 2.000 and one of 1.000, with Rp 500 left for coins. This greedy method, always taking the biggest note that fits, gives the fewest notes for the rupiah series, although not for every possible set of note values.

5. `p++` compiles and moves `p` to `bill[1]`. `c++` compiles (the pointer is not const). `k++` does not compile (the pointer is const). `*p = 1` compiles and writes `bill[0]`. `*c = 1` does not compile (the pointed-to value is const through `c`). `*k = 1` compiles. `p = c` draws a warning (*discards const qualifier*): it would let `p` write where `c` promised not to; C++ refuses it outright.

6. `int big_first(long a, long b) { return a >= 50000 && b < 50000; }` says that `a` must come before `b` only when `a` is big and `b` is not. Among two big bills, or two small ones, the rule says nothing, and only a stable sort keeps them in receipt order; an unstable sort such as `qsort` may mix them up, and the delivery list would change from run to run.

7. `char *name;` is an uninitialized pointer: it holds a random address, and `scanf` writes the typed characters there. Where that address happens to be in writable memory the program seems to work while it damages some other data; elsewhere it crashes. The fix is to give `scanf` real memory: `char name[50]; scanf("%49s", name);` (the array name turns into the address of its first element, so no `&` is needed, and the width 49 leaves room for the `'\0'`).

References

[1] ISO/IEC 9899:2018, *Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, 2018.

[2] Carnegie Mellon University Software Engineering Institute, “EXP34-C. Do not dereference null pointers,” in *SEI CERT C Coding Standard*. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/EXP34-C.+Do+not+dereference+null+pointers

[3] C. A. R. Hoare, “Null references: the billion dollar mistake,” presented at QCon London, 2009. [Online]. Available: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988.

[5] Python Software Foundation, “Data model: objects, values and types,” in *The Python Language Reference*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types

[6] Python Software Foundation, “How do I write a function with output parameters (call by reference)?,” in *Programming FAQ*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/faq/programming.html#how-do-i-write-a-function-with-output-parameters-call-by-reference

[7] cppreference.com, “qsort, qsort_s.” [Online]. Available: https://en.cppreference.com/w/c/algorithm/qsort

[8] Python Software Foundation, “functools — Higher-order functions and operations on callable objects,” Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/functools.html#functools.cmp_to_key

[9] Carnegie Mellon University Software Engineering Institute, “DCL30-C. Declare objects with appropriate storage durations,” in *SEI CERT C Coding Standard*. [Online]. Available:

https://wiki.sei.cmu.edu/confluence/display/c/DCL30-C.+Declare+objects+with+appropriate+storage+durations

[10] Free Software Foundation, "Program instrumentation options," in *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Chapter 8

Evaluation and Discussion II: Functions, Arrays and Pointers, Solved Step by Step

This chapter adds no new feature of C. It takes the three tools of Chapters 5 to 7 (functions, arrays and pointers) and puts them to work on six complete problems, each solved slowly and in full. Four are classic problems of a first programming course: five statistics functions completed inside a skeleton that must not be changed; the Fibonacci numbers computed twice, with a loop and with recursion, and the two ways measured against each other; deciding whether a word is a palindrome, in three different ways that must agree; and the determinant of a 3×3 matrix with the question whether the matrix can be inverted. Two more are new and combine the tools once again: finding a receipt number by recursive binary search, and merging the sorted bills of two shifts with two walking pointers. For every problem you will see the statement in plain words, a solution worked out by hand before any code is written, a design on paper, the complete program in C and in Python with its real output, a *tempting wrong solution* that looks right and is not, taken apart to show exactly where it fails, and a "check yourself" question with its answer. By the end of the chapter you will be able to attack a new problem with the same eight steps, find the test case that exposes a wrong program, and explain why the right program is right.

8.1 Where We Left Off

Before solving anything, let us lay out the toolkit, because naming the tool a problem needs is already half of its solution.

From **Chapter 5** you have **functions**. A function has a name, parameters and a return value; its arguments are copied into its parameters; a prototype promises its shape to the compiler; the standard library supplies `sqrt`, `ceil` and their relatives from `<math.h>`, linked with `-lm`; a variable's scope and lifetime decide where it is visible and how long it lives; and a recursive function solves a problem by calling itself on a smaller instance, with a base case that stops it. You also saw that recursion has a price: `fib(30)` makes 2,692,537 calls.

From **Chapter 6** you have **arrays**: a row of elements of one type under one name, indexed from 0 to $n-1$; loops that walk them with `i < n`; counters indexed by the data; arrays passed to functions as the address of their first element, together with their length; bubble sort, insertion sort, linear search and binary search; and two-dimensional arrays stored row after row.

From **Chapter 7** you have **pointers**: `&` takes an address and `*` follows it; a function that receives addresses can change its caller's variables and return several answers through output parameters; `a[i]` is `*(a + i)`, so a pointer can walk through an array and two pointers can walk toward each other; `const` protects what a pointer points to; and a function can be passed as a parameter.

Table 8.1 lists the six problems and the tools each one exercises most.

Table 8.1 The six problems of this chapter and the tools from Chapters 5 to 7 that each one exercises most.

Problem	Task	Main tools
8.1	Five statistics functions in a given skeleton	array parameters, loops, a cast, `sqrt`
8.2	Fibonacci with a loop and with recursion, and the sum	output parameter, recursion, counting calls, timing
8.3	Is a word a palindrome?	strings as arrays, two pointers, recursion, XOR
8.4	3×3 determinant and invertibility	two-dimensional array, one formula transcribed exactly
8.5	Finding a receipt number	recursive binary search on a sorted array
8.6	Merging the bills of two shifts	two walking pointers, an output array

8.2 How to Approach a Problem

8.2.1 The eight steps

The mathematician George Pólya described problem solving in four phases: understand the problem, devise a plan, carry out the plan, look back [1]. For programs the same phases unfold into eight steps, and every problem of this chapter follows them in order.

1. **Restate the problem in your own words**, and write down every decision that the statement leaves open: what exactly is read, what exactly is printed, what happens with odd input.
2. **Solve a small example by hand**, on paper, to the last digit. If you cannot do it by hand, you cannot tell the computer how to do it, and you will have nothing to compare the program's output with.
3. **Design on paper**: which functions, with which parameters and return values; which arrays; which loop or recursion. Write the design as a few lines of pseudocode or as function prototypes.
4. **Estimate the cost**: how many steps for an input of size n? A design that needs 2^n steps is useless for $n = 60$, whatever the language.
5. **Write the code**, one function at a time, compiling after each one with `-Wall -Wextra`, and fix every warning.
6. **Test**: the example of step 2, the edge cases (the smallest input, the empty input, the biggest input, equal values), and the inputs that a wrong program would get wrong.
7. **Look for the tempting wrong solution.** Ask what a hurried programmer would write, and find the input on which it fails. If your own program passes that input, you understand the problem.
8. **Look back**: could it be simpler, faster or clearer? What would change if the problem changed a little?

> **KEY INSIGHT:** Step 2 is the step that beginners skip and experts never do. A hand-worked example is the only thing that tells you what the right output *is*; without it, a program that prints plausible numbers looks correct, and the error surfaces only when somebody else checks.

8.2.2 How each problem is laid out, and where the code lives

Each problem section has the same six parts: the problem, understanding it by hand, the design on paper, the solution in C, the solution in Python, and a tempting wrong solution followed by a "check yourself" question. Every program is complete and was compiled with `gcc -std=c17 -O2 -Wall -Wextra` and, a second time, with `-fsanitize=address,undefined`, and every output

printed here is the output that the program actually produced. The Python version of every program prints exactly the same text. All files are in the folder `Chapter08` of the book's code, with the inputs in `tests` and a script `run_tests.py` that compiles every C program, runs it and its Python twin, and compares both outputs with the expected ones.

> **WARNING:** Read each problem and try steps 1 to 3 yourself *before* reading the solution. Following a solution is easy; producing one is the skill. The solutions are here to compare with, not to replace your attempt.

8.3 Problem 8.1: Statistics Functions

8.3.1 The problem

> **PROBLEM 8.1.** You are given a complete `main` function, ten whole numbers in an array, and the prototypes of five functions: `max`, `min`, `sum`, `average` and `sDeviation`, each taking a pointer to the array and its size as an `unsigned int`. Complete the five function bodies, without changing `main`, the names or the prototypes, so that the program prints the largest value, the smallest value, the sum, the average and the standard deviation, the last two with three decimals.

The skeleton decides much for us, and one of the decisions is part of the problem: the given code is fixed. In real teams this is common; the prototypes are an agreement with other people's code, and "improving" a name (`Max`, or `int max(int *a, int n)`) breaks the agreement even when the program still works. The ten numbers, in our version, are the kilograms of laundry that one customer, a small boarding house, brought on ten visits: 12, 7, 15, 9, 20, 11, 8, 14, 10 and 18. The standard deviation is the *population* standard deviation, dividing by the number of values n; Section 8.3.6 comes back to this choice.

8.3.2 Understanding it by hand

The largest value is 20 and the smallest 7, by looking. The sum is $12 + 7 + 15 + 9 + 20 + 11 + 8 + 14 + 10 + 18 = 124$. The average, the arithmetic mean, is

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i = \frac{124}{10} = 12.4 \tag{8.1}$$

The **standard deviation** measures how far the values typically lie from the mean. For each value compute its distance from the mean, $x_i - \bar{x}$, square it (so that distances below and above the mean do not cancel, and big distances count more), take the mean of the squares, and finally the square root, which brings the unit back to kilograms:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} (x_i - \bar{x})^2} \tag{8.2}$$

Table 8.2 carries out Equation (8.2) by hand for the ten values.

Table 8.2 The standard deviation of the ten values worked by hand: the deviations add up to zero, and the squared deviations to 166.40.

x_i	$x_i - x$	$(x_i - x)^2$
12	−0.4	0.16
7	−5.4	29.16
15	2.6	6.76
9	−3.4	11.56
20	7.6	57.76
11	−1.4	1.96
8	−4.4	19.36
14	1.6	2.56
10	−2.4	5.76
18	5.6	31.36
sum 124	**0.0**	**166.40**

The column of deviations adds up to exactly zero, as it always must (the mean is the balance point of the data), which is a useful check of the arithmetic. The mean of the squares is $166.40/10 = 16.64$, and $\sigma = \sqrt{16.64} \approx 4.0792$, printed with three decimals as 4.079. So the program must print 20, 7, 124, 12.400 and 4.079. In words: this customer brings about 12.4 kg per visit, give or take about 4 kg.

8.3.3 Designing it on paper

The skeleton fixes the five prototypes, so the design is the body of each.

- `max`: start with `INT_MIN`, the smallest `int` there is (from `<limits.h>`), and let every element that is bigger replace it. Starting at `INT_MIN` means the first element always replaces it, so no special case for the first element is needed. `min` mirrors it with `INT_MAX` and `<`.
- `sum`: an accumulator that starts at 0.
- `average`: one line, `(double) sum(a, aSize) / aSize`. The cast must convert an operand *before* the division; this is the cast lesson of Chapter 2 once more.
- `sDeviation`: compute the mean once, then accumulate the squared deviations in a `double`, divide by `aSize` and take `sqrt`. The function reuses `average`, which reuses `sum`: divide and conquer, Chapter 5.

The size parameter is `unsigned int`. A loop counter declared `int i` and compared with `i < aSize` mixes signed and unsigned values, and `-Wextra` warns about it (*comparison of integer expressions of different signedness*). Declaring the counter `unsigned int i` keeps the build silent and is the honest type: an index is never negative. Cost: each function walks the array once, so all five together take a few dozen steps for ten numbers; `sDeviation` walks it three times (twice inside `average` and `sum`, once itself), still linear in n.

8.3.4 The C solution

Listing 8.1 is the completed skeleton. `main` and the five prototypes are exactly as given; only the bodies are ours.

Listing 8.1 `stats.c`: the given skeleton with the five bodies completed; `INT_MIN` and `INT_MAX` start the searches, the cast comes before the division, and the deviation uses `sqrt`.

```c
/* stats.c - Problem 8.1: five statistics functions completed inside the
   given skeleton (main and the prototypes are exactly as given) */
#include <stdio.h>
#include <limits.h>
#include <math.h>

int max(int *a, unsigned int aSize);
int min(int *a, unsigned int aSize);
int sum(int *a, unsigned int aSize);
double average(int *a, unsigned int aSize);
double sDeviation(int *a, unsigned int aSize);

int main(void)
{
    int kg[10] = {12, 7, 15, 9, 20, 11, 8, 14, 10, 18};
    unsigned int n = 10;

    printf("Maximum value      : %d\n", max(kg, n));
    printf("Minimum value      : %d\n", min(kg, n));
    printf("Sum                : %d\n", sum(kg, n));
    printf("Average            : %.3f\n", average(kg, n));
    printf("Standard deviation : %.3f\n", sDeviation(kg, n));
    return 0;
}

int max(int *a, unsigned int aSize)
{
    int result = INT_MIN;                /* smaller than any real value */
    for (unsigned int i = 0; i < aSize; i++)
        if (a[i] > result)
            result = a[i];
    return result;
}

int min(int *a, unsigned int aSize)
{
    int result = INT_MAX;                /* bigger than any real value */
    for (unsigned int i = 0; i < aSize; i++)
        if (a[i] < result)
            result = a[i];
    return result;
}
```

Listing 8.1 (Continued) *stats.c: the given skeleton with the five bodies completed; INT_MIN and INT_MAX start the searches, the cast comes before the division, and the deviation uses sqrt.*

```c
int sum(int *a, unsigned int aSize)
{
    int result = 0;
    for (unsigned int i = 0; i < aSize; i++)
        result += a[i];
    return result;
}

double average(int *a, unsigned int aSize)
{
    return (double) sum(a, aSize) / aSize;   /* cast BEFORE dividing */
}

double sDeviation(int *a, unsigned int aSize)
{
    double mean = average(a, aSize);
    double squares = 0.0;
    for (unsigned int i = 0; i < aSize; i++) {
        double d = a[i] - mean;          /* distance from the mean */
        squares += d * d;
    }
    return sqrt(squares / aSize);        /* population standard deviation */
}
```

```
Maximum value      : 20
Minimum value      : 7
Sum                : 124
Average            : 12.400
Standard deviation : 4.079
```

The five lines match the hand computation. Build it with `-lm` at the end of the command on Linux, because of `sqrt` (Chapter 5):

```
gcc -std=c17 -Wall -Wextra -o stats stats.c -lm
```

8.3.5 The Python solution

Python already has built-in functions named `max`, `min` and `sum`; defining our own functions with those names would hide the built-ins for the rest of the file. **Listing 8.2** therefore names them `max_`, `min_` and `sum_`, the usual Python convention for a name that would otherwise clash. Python has no `INT_MIN`, because its integers have no limit; `-math.inf` and `math.inf`, minus and plus infinity, play the same role.

Listing 8.2 `stats.py`: the same five functions in Python, where `/` always gives a fractional result.

```python
# stats.py - Problem 8.1: the same five statistics functions in Python
import math


def max_(a):                                # max_ : Python already has max()
    result = -math.inf                      # smaller than any real value
    for x in a:
        if x > result:
            result = x
    return result


def min_(a):
    result = math.inf                       # bigger than any real value
    for x in a:
        if x < result:
            result = x
    return result


def sum_(a):
    result = 0
    for x in a:
        result += x
    return result


def average(a):
    return sum_(a) / len(a)                 # / always gives a float in Python


def s_deviation(a):
    mean = average(a)
    squares = 0.0
    for x in a:
        d = x - mean                        # distance from the mean
        squares += d * d
    return math.sqrt(squares / len(a))      # population standard deviation
```

Listing 8.2 (Continued) *stats.py: the same five functions in Python, where / always gives a fractional result.*

```python
def main():
    kg = [12, 7, 15, 9, 20, 11, 8, 14, 10, 18]
    print(f"Maximum value      : {max_(kg)}")
    print(f"Minimum value      : {min_(kg)}")
    print(f"Sum                : {sum_(kg)}")
    print(f"Average            : {average(kg):.3f}")
    print(f"Standard deviation : {s_deviation(kg):.3f}")


if __name__ == "__main__":
    main()
```

The output is identical. The standard library module `statistics` offers `mean` and `pstdev`, which compute the same two numbers [2]; writing them yourself once is the way to know what those library functions do.

8.3.6 A tempting wrong solution

Two shortcuts tempt everyone. The first starts `min` at 0 “because a counter starts at 0”. The second writes the average as `sum(a, aSize) / aSize`, which looks exactly like the formula. **Listing 8.3** makes both mistakes, and **Listing 8.4** repeats them in Python.

Listing 8.3 `wrong_stats.c`: a tempting wrong solution that starts the minimum at 0 and divides two whole numbers.

```c
/* wrong_stats.c - a tempting wrong solution to Problem 8.1 (do not copy) */
#include <stdio.h>
#include <math.h>

int min(int *a, unsigned int aSize)
{
    int result = 0;                      /* WRONG: 0 is smaller than every value */
    for (unsigned int i = 0; i < aSize; i++)
        if (a[i] < result)
            result = a[i];
    return result;
}

int sum(int *a, unsigned int aSize)
{
    int result = 0;
    for (unsigned int i = 0; i < aSize; i++)
        result += a[i];
    return result;
}

double average(int *a, unsigned int aSize)
{
    return sum(a, aSize) / aSize;        /* WRONG: whole-number division first */
}

double sDeviation(int *a, unsigned int aSize)
{
    double mean = average(a, aSize);
    double squares = 0.0;
    for (unsigned int i = 0; i < aSize; i++)
        squares += (a[i] - mean) * (a[i] - mean);
    return sqrt(squares / aSize);
}

int main(void)
{
    int kg[10] = {12, 7, 15, 9, 20, 11, 8, 14, 10, 18};
    unsigned int n = 10;

    printf("Minimum value      : %d\n", min(kg, n));
    printf("Average            : %.3f\n", average(kg, n));
    printf("Standard deviation : %.3f\n", sDeviation(kg, n));
    return 0;
}
```

Listing 8.4 `wrong_stats.py`: the same two mistakes in Python, where the whole-number division is written `//`.

```python
# wrong_stats.py - a tempting wrong solution to Problem 8.1 (do not copy)
import math


def min_(a):
    result = 0                          # WRONG: 0 is smaller than every value
    for x in a:
        if x < result:
            result = x
    return result


def average(a):
    return sum(a) // len(a)             # WRONG: // is whole-number division


def s_deviation(a):
    mean = average(a)
    return math.sqrt(sum((x - mean) ** 2 for x in a) / len(a))


def main():
    kg = [12, 7, 15, 9, 20, 11, 8, 14, 10, 18]
    print(f"Minimum value      : {min_(kg)}")
    print(f"Average            : {average(kg):.3f}")
    print(f"Standard deviation : {s_deviation(kg):.3f}")


if __name__ == "__main__":
    main()
```

```
Minimum value      : 0
Average            : 12.000
Standard deviation : 4.099
```

The minimum is 0, a value that is not in the data at all: every value is compared with 0, none is smaller, and 0 survives. The average is 12.000 because `124 / 10` with two whole numbers is 12; the fraction is thrown away *before* the result is converted to `double` for the return. And the standard deviation, 4.099, is wrong in a subtle way: it is computed correctly *from the wrong mean*. Using 12 instead of 12.4, the squared deviations add up to 168 instead of 166.40, and $\sqrt{16.8} \approx 4.099$. A wrong number that is close to the right one is the most dangerous kind, because it passes a quick glance; only the hand computation of Section 8.3.2 exposes it.

Why does the wrong minimum go unnoticed so often? Because many test arrays contain 0 or a negative number, and then starting at 0 happens to give the right answer. The test that exposes it is an array of positive values only, like ours. Starting at `INT_MAX`, or at the first element `a[0]`, is right for every input.

> **CHECK YOURSELF.** A statistics program gives 4.300 instead of 4.079 for the same ten values, and the programmer who wrote it insists it is correct. What did that program do differently, and when is that choice the right one?
>
> *Answer.* It divided by $n - 1 = 9$ instead of $n = 10$: $\sqrt{166.40/9} = \sqrt{18.49} \approx 4.300$. That is the *sample* standard deviation, the right choice when the ten values are a sample used to estimate the spread of a larger population (Python's `statistics.stdev`); dividing by n is right when the ten values are the whole population of interest (`statistics.pstdev`) [2]. Neither is a bug; the bug is not saying which one a program computes.

8.4 Problem 8.2: Fibonacci Two Ways

8.4.1 The problem

> **PROBLEM 8.2.** Read a whole number n. Compute the n-th Fibonacci number $F(n)$, starting with $F(0) = 0$ and $F(1) = 1$, and the sum of the sequence from $F(0)$ to $F(n)$, in two ways: with a loop, and with recursion. Then compare how much work the two ways need for values of n above 30. For $n = 9$ the sequence is 0 1 1 2 3 5 8 13 21 34, so $F(9) = 34$ and the sum is 88.

The sample fixes the convention, and it must be fixed first: some books start the sequence with 1, 1, and then every number is shifted by one place. We decide the rest. The program prints the first sixteen numbers of the sequence (or all of them when n is smaller), the two results of each method, and then a table of the work that recursion needs for n = 20, 25, 30 and 32. Values of n above 90 are refused, because the sum up to $F(90)$ is the largest that fits in a 64-bit `long long`; for n above 40 the recursive method is skipped, for a reason the table will make obvious.

8.4.2 Understanding it by hand

For $n = 9$: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. The sum is $0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 = 88$. There is a shortcut worth knowing. Continue the sequence two more places: $F(10) = 55$, $F(11) = 89$. And $89 - 1 = 88$. In general

$$\sum_{i=0}^{n} F\,(i) = F(n+2) - 1 \tag{8.3}$$

The proof is by induction, the mathematical twin of recursion. For $n = 0$: the left side is $F(0) = 0$ and the right side is $F(2) - 1 = 1 - 1 = 0$. If Equation (8.3) holds for n, then adding $F(n + 1)$ to both sides gives $F(n + 2) - 1 + F(n + 1) = F(n + 3) - 1$ on the right, which is Equation (8.3) for $n + 1$ [3]. The recursive method uses this identity for its sum, so that it does not have to call `fib` once for every term.

The cost of the recursive method was worked out in Chapter 5: `fib(n)` makes $C(n) = 2F(n + 1) - 1$ calls. For $n = 32$ that is $2 \cdot 3524578 - 1 = 7049155$ calls. The loop needs n steps.

8.4.3 Designing it on paper

The iterative method keeps two rolling variables, `prev` and `curr`, which always hold two neighbors of the sequence, and a running `sum`. The problem wants two results from one function, so the function returns $F(n)$ and gives the sum through an output parameter, `long long *sum`, the pattern of Chapter 7. In pseudocode:

Pseudocode 8.1 The iterative Fibonacci: two neighbors of the sequence move forward n times, and the sum collects every value that `prev` takes.

```
FIB-ITER(n)
1  prev = 0, curr = 1, sum = 0          // F(0) and F(1)
2  repeat n times
3      sum = sum + prev
4      next = prev + curr
5      prev = curr, curr = next
6  sum = sum + prev                     // prev is now F(n)
7  return prev and sum
```

Pseudocode 8.1 can be traced for $n = 2$: before the loop `prev` = 0, `curr` = 1; the first turn adds 0 to the sum and moves on to 1, 1; the second adds 1 and moves on to 1, 2; after the loop `prev` = 1 = $F(2)$, and adding it makes the sum $0 + 1 + 1 = 2$, which is $F(4) - 1 = 3 - 1$. The recursive method is the three-line function of Chapter 5 with a global call counter, plus `fib_rec_sum(n)`, which returns `fib_rec(n + 2) - 1` by Equation (8.3).

8.4.4 The C solution

Listing 8.5 follows the design line by line; `main` reads n, checks its range, prints the sequence, runs both methods and builds the table of work.

Listing 8.5 `fibonacci.c`: the iterative method with the sum through an output parameter, the recursive method with a call counter, and a table of the work for larger n.

```c
/* fibonacci.c - Problem 8.2: the n-th Fibonacci number and the sum of the
   sequence up to it, iteratively and recursively, and what each one costs.
   Input: n (0..90). Run with --time to measure seconds as well. */
#include <stdio.h>
#include <string.h>
#include <time.h>

long long calls = 0;

long long fib_iter(int n, long long *sum)
{
    long long prev = 0, curr = 1;        /* F(0) and F(1) */
    *sum = 0;
    for (int i = 0; i < n; i++) {
        *sum += prev;
        long long next = prev + curr;
        prev = curr;
        curr = next;
    }
    *sum += prev;                        /* prev is now F(n) */
    return prev;
}

long long fib_rec(int n)
{
    calls++;
    if (n < 2)
        return n;                        /* F(0) = 0, F(1) = 1 */
    return fib_rec(n - 1) + fib_rec(n - 2);
}

long long fib_rec_sum(int n)             /* F(0) + ... + F(n) = F(n+2) - 1 */
{
    return fib_rec(n + 2) - 1;
}
```

Listing 8.5 (Continued) *fibonacci.c: the iterative method with the sum through an output parameter, the recursive method with a call counter, and a table of the work for larger n.*

```
int main(int argc, char *argv[])
{
    int n;
    int timing = argc > 1 && strcmp(argv[1], "--time") == 0;
    if (scanf("%d", &n) != 1 || n < 0 || n > 90) {
        printf("please give n from 0 to 90\n");
        return 1;
    }
    long long s;
    printf("sequence:");
    for (int i = 0; i <= n && i <= 15; i++)
        printf(" %lld", fib_iter(i, &s));
    printf(n > 15 ? " ...\n" : "\n");

    long long f = fib_iter(n, &s);
    printf("iterative: F(%d) = %lld, sum = %lld\n", n, f, s);
    if (n <= 40) {
        f = fib_rec(n);
        printf("recursive: F(%d) = %lld, sum = %lld\n", n, f, fib_rec_sum(n));
    }

    int sizes[4] = {20, 25, 30, 32};
    printf("   n   loop steps   recursive calls\n");
    for (int k = 0; k < 4; k++) {
        calls = 0;
        clock_t t0 = clock();
        fib_rec(sizes[k]);
        double secs = (double) (clock() - t0) / CLOCKS_PER_SEC;
        printf("%4d %12d %17lld", sizes[k], sizes[k], calls);
        if (timing)
            printf("   %.3f seconds", secs);
        printf("\n");
    }
    return 0;
}
```

The input is a single number:

```
9
```

```
sequence: 0 1 1 2 3 5 8 13 21 34
iterative: F(9) = 34, sum = 88
recursive: F(9) = 34, sum = 88
   n   loop steps   recursive calls
  20           20             21891
  25           25            242785
  30           30           2692537
  32           32           7049155
```

Both methods agree with the sample, 34 and 88. The table is the lesson of the problem: while the loop needs 32 steps for $n = 32$, the recursion needs over seven million calls, and every increase of n by 5 multiplies the calls by about 11. **Figure 8.1** draws the two columns on a logarithmic scale, where each grid line is ten times the one below it; the loop's bars barely rise, the recursion's climb a factor of ten every two or three steps of n.

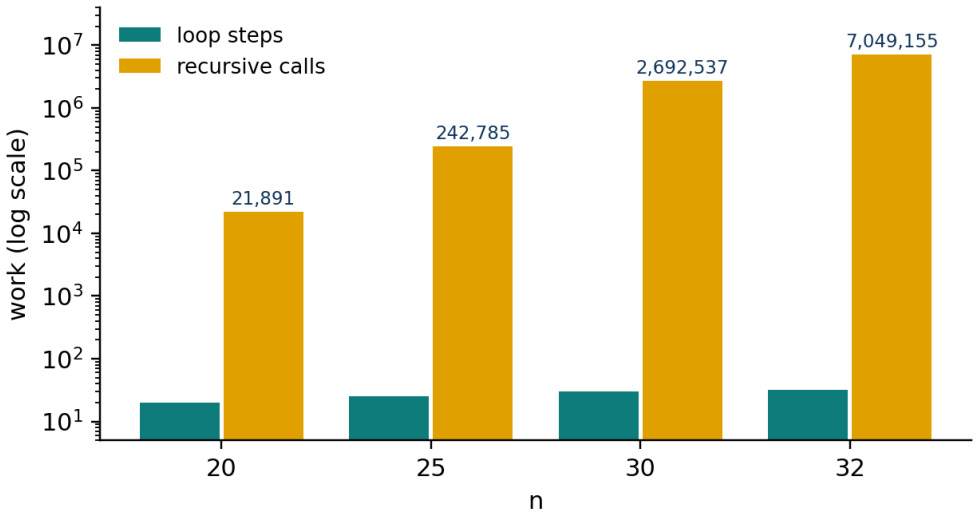


Figure 8.1 The work of the two methods, as counted by `fibonacci.c`: loop steps grow like n, recursive calls like 1.618^n (logarithmic scale).

8.4.5 The Python solution

In **Listing 8.6** the iterative function returns a tuple of two values instead of using an output parameter, and the sliding step is the tuple assignment `prev, curr = curr, prev + curr`, which needs no temporary variable.

Listing 8.6 `fibonacci.py`: the same two methods in Python; the iterative one returns a tuple.

```python
# fibonacci.py - Problem 8.2: the n-th Fibonacci number and the sum of the
# sequence up to it, iteratively and recursively, and what each one costs.
# Input: n (0..90). Run with --time to measure seconds as well.
import sys
import time

calls = 0


def fib_iter(n):
    """Return (F(n), F(0) + ... + F(n))."""
    prev, curr, total = 0, 1, 0          # F(0) and F(1)
    for _ in range(n):
        total += prev
        prev, curr = curr, prev + curr
    return prev, total + prev            # prev is now F(n)


def fib_rec(n):
    global calls
    calls += 1
    if n < 2:
        return n                         # F(0) = 0, F(1) = 1
    return fib_rec(n - 1) + fib_rec(n - 2)


def fib_rec_sum(n):                      # F(0) + ... + F(n) = F(n+2) - 1
    return fib_rec(n + 2) - 1


def main():
    global calls
    timing = len(sys.argv) > 1 and sys.argv[1] == "--time"
    words = sys.stdin.read().split()
    n = int(words[0]) if words and words[0].lstrip("-").isdigit() else -1
    if not 0 <= n <= 90:
        print("please give n from 0 to 90")
        return 1
    seq = " ".join(str(fib_iter(i)[0]) for i in range(min(n, 15) + 1))
    print(f"sequence: {seq}" + (" ..." if n > 15 else ""))
```

Listing 8.6 (Continued) *fibonacci.py: the same two methods in Python; the iterative one returns a tuple.*

```python
    f, s = fib_iter(n)
    print(f"iterative: F({n}) = {f}, sum = {s}")
    if n <= 40:
        f = fib_rec(n)
        print(f"recursive: F({n}) = {f}, sum = {fib_rec_sum(n)}")

    print("   n   loop steps   recursive calls")
    for size in [20, 25, 30, 32]:
        calls = 0
        t0 = time.process_time()
        fib_rec(size)
        secs = time.process_time() - t0
        line = f"{size:4d} {size:12d} {calls:17d}"
        if timing:
            line += f"   {secs:.3f} seconds"
        print(line)
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

The output is identical. Python integers never overflow, so the Python program could go beyond $n = 90$; the limit is kept so that both programs accept the same inputs.

8.4.6 Measuring the difference

The problem asks to *compare the speed*, and counting calls is only half of it. Both programs accept `--time`, which adds a column of seconds measured with `clock()` in C and `time.process_time()` in Python; the seconds differ from computer to computer and from run to run, which is why they are not part of the normal output. **Table 8.3** shows measurements on the computer used for this book (an Intel Xeon processor at 2.1 GHz), for the recursive $F(35)$ and $F(40)$, and for the loop.

Table 8.3 Measured running time of the recursive and iterative methods; the recursive times grow about elevenfold from $n = 35$ to $n = 40$, as $1.618^5 \approx 11$ predicts.

Method	$F(35)$	$F(40)$
C recursion, `gcc -O0` (no optimization)	0.069 s	0.807 s
C recursion, `gcc -O2`	0.014 s	0.162 s
Python recursion	1.05 s	11.3 s
loop, C or Python	below 0.0001 s	below 0.0001 s

Three lessons are in this table. First, the growth: from 35 to 40, every recursive time grows about elevenfold, exactly what the call count predicts, whatever the language or the compiler. Second, the optimizer: `gcc -O2` made the same C recursion five times faster than `-O0`, by rearranging the calls cleverly, but it could not change the exponential growth. Third, the language: Python runs the same recursion about seventy times slower than optimized C, because every Python function call does much more work. And the loop beats them all by a factor of many thousands, in either language. A better algorithm beats a faster language and a better compiler, every time the input grows large enough [4].

8.4.7 A tempting wrong solution

The most common mistake is the convention. **Listing 8.7** starts the sequence with 1, 1, as many books and websites do, and **Listing 8.8** does the same in Python.

Listing 8.7 `wrong_fibonacci.c`: a tempting wrong solution that starts the sequence at 1, 1.

```c
/* wrong_fibonacci.c - a tempting wrong solution to Problem 8.2 (do not copy):
   it starts the sequence at 1, 1 instead of 0, 1 */
#include <stdio.h>

int main(void)
{
    int n;
    if (scanf("%d", &n) != 1)
        return 1;
    long long a = 1, b = 1, sum = 0, last = 0;   /* WRONG: F(0) is 0, not 1 */
    printf("sequence:");
    for (int i = 0; i <= n; i++) {
        printf(" %lld", a);
        sum += a;
        last = a;
        long long next = a + b;
        a = b;
        b = next;
    }
    printf("\nF(%d) = %lld, sum = %lld\n", n, last, sum);
    return 0;
}
```

Listing 8.8 `wrong_fibonacci.py`: the same mistake in Python.

```python
# wrong_fibonacci.py - a tempting wrong solution to Problem 8.2 (do not copy):
# it starts the sequence at 1, 1 instead of 0, 1
import sys


def main():
    n = int(sys.stdin.read().split()[0])
    a, b, total, last = 1, 1, 0, 0      # WRONG: F(0) is 0, not 1
    seq = []
    for _ in range(n + 1):
        seq.append(str(a))
        total += a
        last = a
        a, b = b, a + b
    print("sequence: " + " ".join(seq))
    print(f"F({n}) = {last}, sum = {total}")


if __name__ == "__main__":
    main()
```

```
sequence: 1 1 2 3 5 8 13 21 34 55
F(9) = 55, sum = 143
```

The program prints ten numbers, as asked, and they are all Fibonacci numbers, which is why the mistake survives a quick look. But the ten are shifted by one place: $F(9)$ comes out as 55 instead of 34, and the sum as 143 instead of 88. The sample in the problem statement exists precisely to catch this, and the lesson is general: before writing a line, compute the sample by hand and write the expected output next to your code.

A second classic mistake is overflow. With `int` instead of `long long`, the sum overflows at $n = 45$ ($F(47) - 1$ exceeds $2^{31} - 1$) and the program prints a negative number; signed overflow is undefined behavior in C, and the sanitizer reports it.

> **CHECK YOURSELF.** How many calls does `fib_rec_sum(9)` make, and why is it more than `fib_rec(9)`?
>
> *Answer.* `fib_rec_sum(9)` calls `fib_rec(11)`, which makes $C(11) = 2F(12) - 1 = 2 \cdot 144 - 1 = 287$ calls, while `fib_rec(9)` makes $2F(10) - 1 = 109$. Computing the sum through

Equation (8.3) costs one recursion two places further, about $1.618^2 \approx 2.6$ times the work of $F(9)$ alone. Adding up `fib_rec(i)` separately for every i from 0 to 9 is no better: $\sum_{i=0}^{9}(2F(i+1)-1) = 2 \cdot 143 - 10 = 276$ calls, almost the same.

8.5 Problem 8.3: Palindromes

8.5.1 The problem

PROBLEM 8.3. Read words and report, for each one, whether it is a *palindrome*, a word that reads the same from left to right and from right to left, such as `racecar`, `level` or `amanaplanacanalpanama` ("a man, a plan, a canal: Panama" without spaces). The words are in lowercase letters without spaces. As an optional challenge, decide it also in a way that uses more than plain loops and comparisons, for example recursion or bit operations.

We decide the details. The program reads words until the end of the input, each up to 99 letters, and prints each word with its verdict and, at the end, how many were palindromes. It decides every word in three ways, the standard two-pointer scan, a recursion and a scan with the XOR operator, and it warns if the three ever disagree. The test input contains sixteen palindromes, among them the Indonesian words `katak` (frog), `malam` (night), `kodok` (toad) and the phrase `kasurrusak` ("kasur rusak", a broken mattress, without the space), and four words that are not.

8.5.2 Understanding it by hand

Figure 8.2 checks `kasurrusak` with two fingers, one on the first letter and one on the last. They show `k` and `k`: equal, so both fingers move one letter inward. Then `a` and `a`, `s` and `s`, `u` and `u`, `r` and `r`: all equal. Now the left finger has passed the right one, nothing is left to compare, and the word is a palindrome. Five comparisons for ten letters: the scan compares each pair once and stops in the middle.

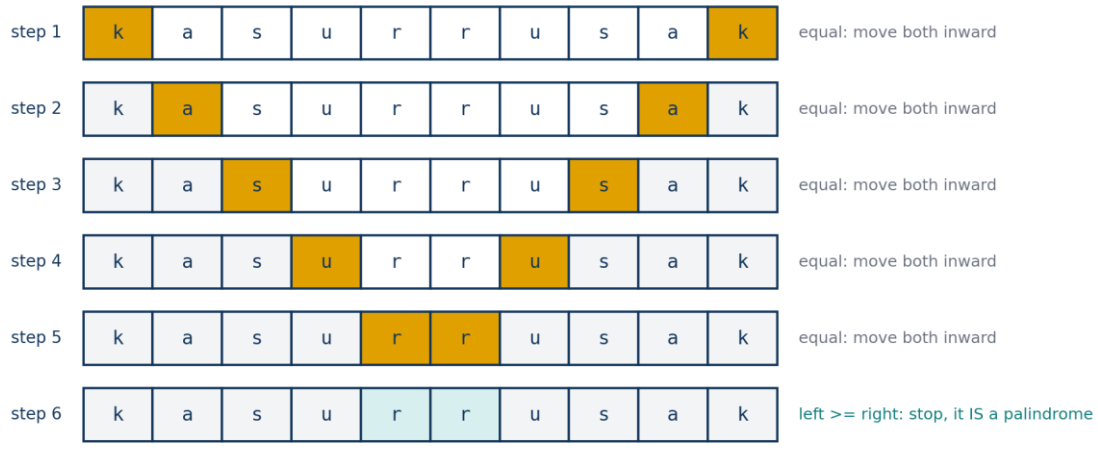


Figure 8.2 The two-pointer scan on `kasurrusak`: the amber letters are compared, the gray ones are done; after five equal pairs the pointers cross and the word is a palindrome.

For `laundry` the first comparison, `l` against `y`, already fails, and the answer is "not a palindrome" at once. The first mismatch decides; there is no need to look further. A word of one letter, `x`, has no pair at all, and is a palindrome. For a word of n letters the scan makes at most $\lfloor n/2 \rfloor$ comparisons.

8.5.3 Designing it on paper

Two pointers. `left` starts at the first character, `right` at the last, which is `s + strlen(s) - 1`. While `left < right`: if `*left != *right`, answer 0; otherwise move `left` forward and `right` back. When the loop ends, answer 1. This is exactly the reversal walk of Chapter 7, with a comparison instead of a swap.

Recursion. A word is a palindrome when its first and last letters are equal *and* the part between them is a palindrome. The base case is a part of zero or one letters, which is always a palindrome. The function receives the two pointers: `is_pal_rec(left, right)`.

XOR. The bitwise operator `^` (exclusive or) compares two numbers bit by bit and gives 1 in every position where they differ, so `x ^ y` is 0 exactly when `x == y`. OR-ing together the XOR of every pair collects a 1 bit wherever any pair differs; the word is a palindrome exactly when the collection is 0 at the end. This scan has no `if` inside the loop at all, which is the spirit of the challenge; it always looks at every pair instead of stopping at the first mismatch.

Reading the word needs care: `scanf("%99s", word)` into `char word[100]`, with *no* `&`, because the name of an array is already the address of its first element (Chapter 7), and with the width 99, which stops `scanf` from writing past the end of the array when somebody types a longer word, leaving room for the terminating `'\0'`.

8.5.4 The C solution

Listing 8.9 contains the three deciding functions and a `main` that reads words until the input ends.

Listing 8.9 `palindrome.c`: three ways to decide a palindrome, which must agree: two pointers, recursion on two pointers, and XOR over the pairs.

```c
/* palindrome.c - Problem 8.3: does a word read the same in both directions?
   Three ways that must agree: two pointers, recursion, and XOR.
   Input: words separated by spaces or new lines, up to 99 letters each. */
#include <stdio.h>
#include <string.h>

int is_palindrome(const char *s)         /* the standard two-pointer scan */
{
    const char *left = s;
    const char *right = s + strlen(s) - 1;
    while (left < right) {               /* stop in the middle */
        if (*left != *right)
            return 0;                    /* the first mismatch decides */
        left++;
        right--;
    }
    return 1;
}

int is_pal_rec(const char *left, const char *right)
{
    if (left >= right)                   /* zero or one letter left */
        return 1;
    return *left == *right && is_pal_rec(left + 1, right - 1);
}

int is_pal_xor(const char *s)            /* equal letters XOR to zero */
{
    size_t n = strlen(s);
    unsigned diff = 0;
    for (size_t i = 0; i < n / 2; i++)
        diff |= (unsigned char) s[i] ^ (unsigned char) s[n - 1 - i];
    return diff == 0;
}

int main(void)
{
    char word[100];                      /* 99 letters and the '\0' */
    int yes = 0, total = 0;
    while (scanf("%99s", word) == 1) {   /* no & : word is already an address */
        int a = is_palindrome(word);
        int b = is_pal_rec(word, word + strlen(word) - 1);
        int c = is_pal_xor(word);
        printf("%-22s %s%s\n", word, a ? "palindrome" : "not a palindrome",
               (a == b && b == c) ? "" : "  (METHODS DISAGREE)");
        yes += a;
        total++;
    }
    printf("%d of %d words are palindromes\n", yes, total);
    return 0;
}
```

Listing 8.9 (Continued) *palindrome.c: three ways to decide a palindrome, which must agree: two pointers, recursion on two pointers, and XOR over the pairs.*

The input has twenty words:

```
amanaplanacanalpanama neveroddoreven racecar level
kayak rotator civic madam
refer stats noon malam
katak kasurrusak kodok
x ab laundry profitina
abcdefghijklmnlkjihgfedcba
```

```
amanaplanacanalpanama  palindrome
neveroddoreven         palindrome
racecar                palindrome
level                  palindrome
kayak                  palindrome
rotator                palindrome
civic                  palindrome
madam                  palindrome
refer                  palindrome
stats                  palindrome
noon                   palindrome
malam                  palindrome
katak                  palindrome
kasurrusak             palindrome
kodok                  palindrome
x                      palindrome
ab                     not a palindrome
laundry                not a palindrome
profitina              not a palindrome
abcdefghijklmnlkjihgfedcba not a palindrome
16 of 20 words are palindromes
```

All sixteen palindromes are recognized, and the three methods never disagree (otherwise the line would end with a warning). Look at the last word, `abcdefghijklmnlkjihgfedcba`: it is a palindrome except for the two letters in the middle, `m` and `n`. The two-pointer scan compares twelve equal pairs before it finds the difference; a test like this one, which fails only at the last possible comparison, is the one that exposes a scan that stops one step too early.

8.5.5 The Python solution

Listing 8.10 follows the same three methods with indexes instead of pointers. The XOR is applied to `ord(c)`, the character code, because Python's `^` works on integers, not on characters.

Listing 8.10 `palindrome.py`: the same three methods in Python, with indexes.

```
# palindrome.py - Problem 8.3: does a word read the same in both directions?
# Three ways that must agree: two indexes, recursion, and XOR.
# Input: words separated by spaces or new lines.
import sys


def is_palindrome(s):                    # the standard two-index scan
    left, right = 0, len(s) - 1
    while left < right:                  # stop in the middle
        if s[left] != s[right]:
            return False                 # the first mismatch decides
        left += 1
        right -= 1
    return True


def is_pal_rec(s, left, right):
    if left >= right:                    # zero or one letter left
        return True
    return s[left] == s[right] and is_pal_rec(s, left + 1, right - 1)


def is_pal_xor(s):                       # equal letters XOR to zero
    diff = 0
    n = len(s)
    for i in range(n // 2):
        diff |= ord(s[i]) ^ ord(s[n - 1 - i])
    return diff == 0


def main():
    yes = total = 0
    for word in sys.stdin.read().split():
        word = word[:99]                 # the C program reads at most 99
        a = is_palindrome(word)
        b = is_pal_rec(word, 0, len(word) - 1)
        c = is_pal_xor(word)
        verdict = "palindrome" if a else "not a palindrome"
        warn = "" if a == b == c else "  (METHODS DISAGREE)"
        print(f"{word:<22} {verdict}{warn}")
        yes += a
        total += 1
    print(f"{yes} of {total} words are palindromes")
    # the Python shortcut, for comparison: word == word[::-1]
```

Listing 8.10 (Continued) *palindrome.py: the same three methods in Python, with indexes.*

```
if __name__ == "__main__":
    main()
```

The last comment in the program shows what an experienced Python programmer would actually write: `word == word[::-1]`, comparing the word with its reverse, made by a *slice* with step -1. It is correct and short, but it builds a reversed copy of the word and compares all n letters, where the two-index scan compares at most $n/2$ and builds nothing; for a single word the difference does not matter, and knowing what the short form costs is part of knowing it.

8.5.6 A tempting wrong solution

The two-pointer scan is often written with one index `i` and the partner computed from it. The partner of position `i` in a word of length `n` is position `n - 1 - i`: the partner of the first letter, 0, is

the last letter, `n - 1`. The tempting mistake forgets the `- 1`, as in **Listing 8.11** and its Python twin **Listing 8.12**.

Listing 8.11 `wrong_palindrome.c`: a tempting wrong solution whose partner index is `n - i` instead of `n - 1 - i`.

```c
/* wrong_palindrome.c - a tempting wrong solution to Problem 8.3 (do not copy):
   the partner of s[i] is written as s[n - i] */
#include <stdio.h>
#include <string.h>

int is_palindrome(const char *s)
{
    size_t n = strlen(s);
    for (size_t i = 0; i < n / 2; i++)
        if (s[i] != s[n - i])            /* WRONG: s[n] is the '\0' */
            return 0;
    return 1;
}

int main(void)
{
    char word[100];
    while (scanf("%99s", word) == 1)
        printf("%-22s %s\n", word, is_palindrome(word) ? "palindrome"
                                                       : "not a palindrome");
    return 0;
}
```

Listing 8.12 `wrong_palindrome.py`: the same mistake in Python, with an explicit zero character to imitate C.

```python
# wrong_palindrome.py - a tempting wrong solution to Problem 8.3 (do not copy):
# the partner of s[i] is written as s[n - i]
import sys


def is_palindrome(s):
    n = len(s)
    padded = s + "\0"                    # imitate C's terminating zero
    for i in range(n // 2):
        if padded[i] != padded[n - i]:   # WRONG: s[n] is the '\0'
            return False
    return True


def main():
    for word in sys.stdin.read().split():
        verdict = "palindrome" if is_palindrome(word) else "not a palindrome"
        print(f"{word:<22} {verdict}")


if __name__ == "__main__":
    main()
```

```
amanaplanacanalpanama  not a palindrome
neveroddoreven         not a palindrome
racecar                not a palindrome
level                  not a palindrome
kayak                  not a palindrome
rotator                not a palindrome
civic                  not a palindrome
madam                  not a palindrome
refer                  not a palindrome
stats                  not a palindrome
noon                   not a palindrome
malam                  not a palindrome
katak                  not a palindrome
kasurrusak             not a palindrome
kodok                  not a palindrome
x                      palindrome
ab                     not a palindrome
laundry                not a palindrome
profitina              not a palindrome
abcdefghijklmnlkjihgfedcba not a palindrome
```

Every palindrome of two or more letters is rejected. The first comparison, with `i` = 0, compares `s[0]` with `s[n]`, and `s[n]` is not the last letter but the terminating `'\0'` of the string, which no letter equals. Only `x`, with $n/2 = 0$ comparisons, slips through as a palindrome. The mistake is instructive because in C it does not crash: `s[n]` is inside the array (it is the zero), so even the sanitizer is silent. In Python, without the explicit zero, `s[n]` would raise `IndexError`, which is why the Python version appends `"\0"` to imitate C faithfully.

> **CHECK YOURSELF.** Change the program so that it also accepts phrases with capital letters and spaces, such as “Kasur rusak” or “Never odd or even”. Which two things must change, and why does reading with `%99s` stand in the way?
>
> *Answer.* The comparison must skip every character that is not a letter and compare letters ignoring case: move `left` forward while `*left` is not a letter (`isalpha` from `<ctype.h>`), move `right` back likewise, and compare `tolower(*left)` with `tolower(*right)`. And the input must be read by lines, not by words, because `%99s` stops at the first space and would split “Kasur rusak” into two words; Chapter 9 introduces `fgets` for this.

8.6 Problem 8.4: The Determinant of a 3 × 3 Matrix

8.6.1 The problem

> **PROBLEM 8.4.** Read a 3 × 3 matrix of whole numbers, compute its determinant, and report whether the matrix is invertible: it is invertible exactly when its determinant is not zero.

A matrix is a table of numbers, and a 3 × 3 matrix is a two-dimensional array of three rows and three columns. Why would a laundry care? Here is a real question. Mbak Rina buys detergent, softener and ironing starch from one shop, and the prices changed without a price list. She has three receipts: two packs of detergent, one of softener and one of starch cost Rp 60,000; one detergent, three softeners and two starches cost Rp 72,000; and one detergent alone cost Rp 20,000. Can she work out the price of each item? The quantities form the matrix

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 0 & 0 \end{pmatrix} \tag{8.4}$$

and the prices can be recovered exactly when A is invertible, that is, when its determinant is not zero [5]. If two receipts had the same quantities, they would say the same thing twice, the matrix would have two equal rows, its determinant would be 0, and no amount of arithmetic could separate the three prices. Our program reads several matrices, one group of nine numbers after another, row by row, and prints each with its determinant and verdict.

8.6.2 Understanding it by hand

Name the nine entries by letters, row by row: a, b, c in the first row, d, e, f in the second, g, h, i in the third. Expanding along the first row, the determinant is

$$\det A = a(ei - fh) - b(di - fg) + c(dh - eg) \tag{8.5}$$

Figure 8.3 shows where each term comes from. Each entry of the first row is multiplied by the determinant of the 2×2 matrix that remains when its row and its column are crossed out (the teal cells), and the three products are added with the signs $+, -, +$. The minus sign on the middle term is the part that is easy to forget.

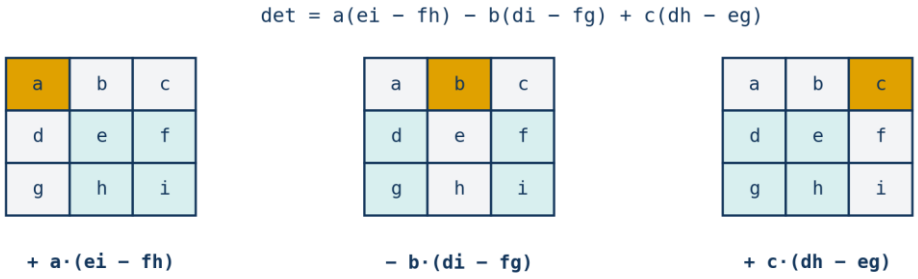


Figure 8.3 Expansion along the first row: each first-row entry (amber) multiplies the 2×2 determinant of the cells left when its row and column are removed (teal), with the signs $+, -, +$.

Work one example to the end. For the matrix with rows $(2, -3, 1)$, $(2, 0, -1)$ and $(1, 4, 5)$:

- $a(ei - fh) = 2 \cdot (0 \cdot 5 - (-1) \cdot 4) = 2 \cdot 4 = 8$;
- $-b(di - fg) = -(-3) \cdot (2 \cdot 5 - (-1) \cdot 1) = 3 \cdot 11 = 33$;
- $c(dh - eg) = 1 \cdot (2 \cdot 4 - 0 \cdot 1) = 8$;

so $\det = 8 + 33 + 8 = 49$, not zero: invertible. For Mbak Rina's receipts, Equation (8.4): $2(3 \cdot 0 - 2 \cdot 0) - 1(1 \cdot 0 - 2 \cdot 1) + 1(1 \cdot 0 - 3 \cdot 1) = 0 + 2 - 3 = -1$. Not zero, so the prices can be recovered; Exercise 8.5 does it. Two quick tests will be useful: the *identity matrix*, with 1 on the diagonal and 0 elsewhere, has determinant 1, and any matrix with two equal rows has determinant 0.

8.6.3 Designing it on paper

The matrix lives in `long long m[3][3]`; `long long` because products of three entries can grow quickly. Reading uses two nested loops, row and column, not nine separate `scanf` calls, in a function `read_matrix` that returns 0 when the input runs out, so that `main` can loop over as many matrices as there are. The determinant is one function, `det3`, that transcribes Equation (8.5) with `m[0][0]` for a, `m[0][1]` for b, and so on; the letters are written in a comment above it, so that each term of the code can be checked against its letter. The parameter is `const long long m[3][3]`: the function only reads, and the column count 3 must be known, as Chapter 6 explained.

8.6.4 The C solution

Listing 8.13 keeps the letter layout as a comment above `det3`, so that each of the three terms can be checked against Equation (8.5).

Listing 8.13 `determinant.c`: reading 3×3 matrices with nested loops and computing each determinant by expansion along the first row.

```c
/* determinant.c - Problem 8.4: the determinant of a 3x3 matrix, and whether
   the matrix is invertible. Input: groups of nine whole numbers, row by row. */
#include <stdio.h>

/*  | a b c |
    | d e f |   det = a(ei - fh) - b(di - fg) + c(dh - eg)
    | g h i |                                              */
long long det3(const long long m[3][3])
{
    return m[0][0] * (m[1][1] * m[2][2] - m[1][2] * m[2][1])
         - m[0][1] * (m[1][0] * m[2][2] - m[1][2] * m[2][0])
         + m[0][2] * (m[1][0] * m[2][1] - m[1][1] * m[2][0]);
}

int read_matrix(long long m[3][3])
{
    for (int r = 0; r < 3; r++)
        for (int c = 0; c < 3; c++)
            if (scanf("%lld", &m[r][c]) != 1)
                return 0;
    return 1;
}

int main(void)
{
    long long m[3][3];
    int number = 0;
    while (read_matrix(m)) {
        long long d = det3(m);
        printf("matrix %d:\n", ++number);
        for (int r = 0; r < 3; r++)
            printf("  %5lld %5lld %5lld\n", m[r][0], m[r][1], m[r][2]);
        printf("  determinant = %lld, %s\n", d,
               d != 0 ? "invertible" : "not invertible");
    }
    return 0;
}
```

The test input holds four matrices: the identity, the example worked above, a matrix whose third row repeats the first, and Mbak Rina's receipts.

```
1 0 0
0 1 0
0 0 1

2 -3 1
2 0 -1
1 4 5

1 2 3
4 5 6
1 2 3

2 1 1
1 3 2
1 0 0
```

```
matrix 1:
      1     0     0
      0     1     0
      0     0     1
  determinant = 1, invertible
matrix 2:
      2    -3     1
      2     0    -1
      1     4     5
  determinant = 49, invertible
matrix 3:
      1     2     3
      4     5     6
      1     2     3
  determinant = 0, not invertible
matrix 4:
      2     1     1
      1     3     2
      1     0     0
  determinant = -1, invertible
```

All four agree with the hand computations: 1, 49, 0 and −1.

8.6.5 The Python solution

In **Listing 8.14** the nine numbers of each group are cut into three rows with slices, `flat[0:3]`, `flat[3:6]` and `flat[6:9]`, which build the list of lists of Chapter 6.

Listing 8.14 `determinant.py`: the same determinant in Python, with the matrix built from slices.

```python
# determinant.py - Problem 8.4: the determinant of a 3x3 matrix, and whether
# the matrix is invertible. Input: groups of nine whole numbers, row by row.
import sys

#  | a b c |
#  | d e f |   det = a(ei - fh) - b(di - fg) + c(dh - eg)
#  | g h i |


def det3(m):
    return (m[0][0] * (m[1][1] * m[2][2] - m[1][2] * m[2][1])
            - m[0][1] * (m[1][0] * m[2][2] - m[1][2] * m[2][0])
            + m[0][2] * (m[1][0] * m[2][1] - m[1][1] * m[2][0]))


def main():
    numbers = [int(w) for w in sys.stdin.read().split()]
    number = 0
    for start in range(0, len(numbers) - 8, 9):
        flat = numbers[start:start + 9]
        m = [flat[0:3], flat[3:6], flat[6:9]]
        d = det3(m)
        number += 1
        print(f"matrix {number}:")
        for row in m:
            print(f"  {row[0]:5d} {row[1]:5d} {row[2]:5d}")
        print(f"  determinant = {d}, "
              + ("invertible" if d != 0 else "not invertible"))


if __name__ == "__main__":
    main()
```

8.6.6 A tempting wrong solution

The classic transcription error flips the sign of the middle term, as in **Listing 8.15** and **Listing 8.16**: all three terms are added.

Listing 8.15 `wrong_determinant.c`: a tempting wrong solution that adds the middle term instead of subtracting it.

```c
/* wrong_determinant.c - a tempting wrong solution to Problem 8.4 (do not copy):
   the middle term is added instead of subtracted */
#include <stdio.h>

long long det3(const long long m[3][3])
{
    return m[0][0] * (m[1][1] * m[2][2] - m[1][2] * m[2][1])
         + m[0][1] * (m[1][0] * m[2][2] - m[1][2] * m[2][0])   /* WRONG sign */
         + m[0][2] * (m[1][0] * m[2][1] - m[1][1] * m[2][0]);
}

int main(void)
{
    long long m[3][3];
    int number = 0;
    for (;;) {
        for (int k = 0; k < 9; k++)
            if (scanf("%lld", &m[k / 3][k % 3]) != 1)
                return 0;
        printf("matrix %d: determinant = %lld\n", ++number, det3(m));
    }
}
```

Listing 8.16 `wrong_determinant.py`: the same mistake in Python.

```python
# wrong_determinant.py - a tempting wrong solution to Problem 8.4 (do not copy):
# the middle term is added instead of subtracted
import sys


def det3(m):
    return (m[0][0] * (m[1][1] * m[2][2] - m[1][2] * m[2][1])
            + m[0][1] * (m[1][0] * m[2][2] - m[1][2] * m[2][0])  # WRONG sign
            + m[0][2] * (m[1][0] * m[2][1] - m[1][1] * m[2][0]))


def main():
    numbers = [int(w) for w in sys.stdin.read().split()]
    for k, start in enumerate(range(0, len(numbers) - 8, 9), 1):
        f = numbers[start:start + 9]
        print(f"matrix {k}: determinant = {det3([f[0:3], f[3:6], f[6:9]])}")


if __name__ == "__main__":
    main()
```

```
matrix 1: determinant = 1
matrix 2: determinant = -17
matrix 3: determinant = 24
matrix 4: determinant = -5
```

Look at the first line. The wrong program gives the right answer, 1, for the identity matrix, because the middle term is multiplied by $b = 0$ and its sign does not matter. The identity matrix, the test that everybody tries first, cannot catch this bug. The matrix with two equal rows does: its determinant must be 0, and the wrong program says 24. The example worked by hand catches it too (-17 instead of 49), and so do Mbak Rina's receipts (-5 instead of -1). The general lesson of

testing is visible here: a test is valuable only if the bug you fear would change its result. Choose test inputs in which every part of the formula matters; an input with many zeros tests little [6].

> **CHECK YOURSELF.** Without computing, what is the determinant of a matrix whose second row is twice its first, such as rows (1,2,3), (2,4,6) and (0,1,5)? Check your answer with Equation (8.5).
>
> *Answer.* Zero: a row that is a multiple of another adds no new information, just like a repeated receipt, so the matrix is not invertible. By Equation (8.5): $1(4 \cdot 5 - 6 \cdot 1) - 2(2 \cdot 5 - 6 \cdot 0) + 3(2 \cdot 1 - 4 \cdot 0) = 14 - 20 + 6 = 0$.

8.7 Problem 8.5: Finding a Receipt

8.7.1 The problem

> **PROBLEM 8.5.** The receipt numbers of the laundry's orders are kept in a sorted array. Write a recursive function that finds a given receipt number and returns its index, or -1 if the number is not in the array, and print every step of the search.

Customers come back for their laundry with a receipt, and sometimes with only the number, read over the telephone. The array holds twelve receipt numbers from 1003 to 1066, sorted, with gaps where orders were canceled. The program looks for three numbers: one in the middle part, the first one, and one that is not there.

8.7.2 Understanding it by hand, and the design

This is the binary search of Chapter 6, written as a recursion. The part still in question runs from index `low` to index `high`. If it is empty (`low > high`), the number is not there: base case, answer -1. Otherwise look at the middle element, `mid = low + (high - low) / 2`. If it is the number, answer `mid`: the second base case. If the number is smaller, the answer is the answer for the left part, `low` to `mid - 1`; otherwise for the right part, `mid + 1` to `high`. Each call works on a part at most half as big, so the recursion is at most $\lfloor \log_2 n \rfloor + 1$ calls deep: four for twelve receipts.

By hand, for 1047 in the array 1003, 1008, 1015, 1021, 1022, 1030, 1034, 1041, 1047, 1052, 1060, 1066: the first call covers 0 to 11, the middle is index 5, 1030, smaller than 1047, so continue with 6 to 11; the middle is index 8, 1047: found. Two calls. The function receives a pointer to the array, the two bounds and the number, plus a `depth` used only to indent the printed steps, so that the depth of the recursion is visible on the screen.

8.7.3 The solutions

Listing 8.17 and **Listing 8.18** implement the design; `main` searches for the three numbers in turn.

Listing 8.17 `find_receipt.c`: recursive binary search over sorted receipt numbers, printing each call indented by its depth.

```c
/* find_receipt.c - Problem 8.5: find a receipt number in a sorted array by
   recursive binary search, and report every step */
#include <stdio.h>

int find(const int *a, int low, int high, int key, int depth)
{
    if (low > high)
        return -1;                          /* empty part: not there */
    int mid = low + (high - low) / 2;
    printf("%*scall %d: [%d..%d], middle a[%d] = %d\n", 2 * depth, "",
           depth + 1, low, high, mid, a[mid]);
    if (a[mid] == key)
        return mid;
    if (key < a[mid])
        return find(a, low, mid - 1, key, depth + 1);
    return find(a, mid + 1, high, key, depth + 1);
}

int main(void)
{
    int receipt[12] = {1003, 1008, 1015, 1021, 1022, 1030,
                       1034, 1041, 1047, 1052, 1060, 1066};
    int keys[3] = {1047, 1003, 1050};
    for (int k = 0; k < 3; k++) {
        printf("looking for %d\n", keys[k]);
        int at = find(receipt, 0, 11, keys[k], 0);
        if (at >= 0)
            printf("found at index %d\n", at);
        else
            printf("not in the book\n");
    }
    return 0;
}
```

Listing 8.18 `find_receipt.py`: the same recursion in Python; `f"{'':{2 * depth}}"` prints the indentation.

```python
# find_receipt.py - Problem 8.5: find a receipt number in a sorted list by
# recursive binary search, and report every step


def find(a, low, high, key, depth):
    if low > high:
        return -1                           # empty part: not there
    mid = low + (high - low) // 2
    print(f"{'':{2 * depth}}call {depth + 1}: [{low}..{high}], "
          f"middle a[{mid}] = {a[mid]}")
    if a[mid] == key:
        return mid
    if key < a[mid]:
        return find(a, low, mid - 1, key, depth + 1)
    return find(a, mid + 1, high, key, depth + 1)


def main():
    receipt = [1003, 1008, 1015, 1021, 1022, 1030,
               1034, 1041, 1047, 1052, 1060, 1066]
    for key in [1047, 1003, 1050]:
        print(f"looking for {key}")
        at = find(receipt, 0, len(receipt) - 1, key, 0)
        print(f"found at index {at}" if at >= 0 else "not in the book")


if __name__ == "__main__":
    main()
```

```
looking for 1047
call 1: [0..11], middle a[5] = 1030
  call 2: [6..11], middle a[8] = 1047
found at index 8
looking for 1003
call 1: [0..11], middle a[5] = 1030
  call 2: [0..4], middle a[2] = 1015
    call 3: [0..1], middle a[0] = 1003
found at index 0
looking for 1050
call 1: [0..11], middle a[5] = 1030
  call 2: [6..11], middle a[8] = 1047
    call 3: [9..11], middle a[10] = 1060
      call 4: [9..9], middle a[9] = 1052
not in the book
```

The number 1050 is not in the array; the fourth call narrows the part to index 9 alone, sees 1052, and the next call would have an empty part. Four calls, as predicted.

8.7.4 A tempting wrong solution

The tempting shortcut writes the recursive calls with `mid` instead of `mid - 1` or `mid + 1`: `find(a, low, mid, ...)` and `find(a, mid, high, ...)`. It looks harmless, since `mid` has already been checked. But when the part has two elements, `low = 8` and `high = 9` for example, `mid` is 8, and if the number is bigger, the next call gets `low = 8, high = 9` again: the part never shrinks, the recursion never reaches a base case, and the program ends with a stack overflow. The rule that prevents it is the rule of every recursion: *each call must work on a strictly smaller problem*. Excluding `mid`, which has been checked, guarantees it.

> **CHECK YOURSELF.** Why does the recursive version use as much memory on the call stack as it makes calls, and why is that harmless here, when it was dangerous for `fib`?
>
> *Answer.* Every active call keeps its frame on the stack until it returns, and here each call waits for the one it makes, so the stack is as deep as the number of calls, at most $\lfloor \log_2 n \rfloor + 1$: 20 frames for a million receipts, 31 for two billion. `fib` was dangerous not because of its depth, which is only n, but because of its total number of calls; the binary search makes only one recursive call per level, so its total and its depth are the same small number.

8.8 Problem 8.6: Merging Two Shifts

8.8.1 The problem

> **PROBLEM 8.6.** The morning shift and the evening shift each keep their bills sorted from smallest to biggest. Write a function that merges the two sorted arrays into one sorted array, walking through both with pointers, and returns the number of bills in the result. It must also work when one shift had no orders.

8.8.2 Understanding it by hand, and the design

Think of two piles of receipts face up, each sorted with the smallest on top. To make one sorted pile, compare the two top receipts, take the smaller, and repeat. When one pile is empty, put the rest of the other pile on the result as it is: it is already sorted, and everything in it is at least as big as everything taken so far. With morning 12, 28, 30, 45, 136.5 and evening 18, 30, 56, 70 (in thousands): 12 against 18, take 12; 28 against 18, take 18; 28 against 30, take 28; 30 against 30, take the morning 30; 45 against 30, take the evening 30; 45 against 56, take 45; 136.5 against 56, take 56; 136.5 against 70, take 70; the evening pile is empty, so put 136.5 on the result.

Three pointers carry this out: `pa` walks the morning array, `pb` the evening array, and `po` the output. The heart is one line, `*po++ = (*pa <= *pb) ? *pa++ : *pb++;`, which copies the smaller front bill to the output and moves the two pointers involved. The comparison `<=` takes the morning bill when the two are equal, which keeps equal bills in shift order: the merge is stable. If the arrays have n_a and n_b elements, every step outputs one element, so the merge takes exactly $n_a + n_b$ steps, and it makes at most

$$n_a + n_b - 1 \tag{8.6}$$

comparisons, because after the last comparison at least one element is copied without one. This step is the heart of *merge sort*, which sorts by splitting an array in halves, sorting each half recursively and merging the two halves, in about $n\log_2 n$ comparisons [7].

8.8.3 The solutions

Listing 8.19 returns the number of bills written, computed as the pointer difference `po - out` of Chapter 7; **Listing 8.20** builds a new list instead.

Listing 8.19 `merge_shifts.c`: merging two sorted arrays with three walking pointers, including the case of an empty evening shift.

```c
/* merge_shifts.c - Problem 8.6: merge the sorted bills of the morning and
   the evening shift into one sorted list, walking both with pointers */
#include <stdio.h>

int merge(const long *a, int na, const long *b, int nb, long *out)
{
    const long *pa = a, *pb = b;
    const long *enda = a + na, *endb = b + nb;
    long *po = out;
    while (pa < enda && pb < endb)       /* take the smaller front bill */
        *po++ = (*pa <= *pb) ? *pa++ : *pb++;
    while (pa < enda)                    /* one shift is finished: */
        *po++ = *pa++;                   /* copy the rest of the other */
    while (pb < endb)
        *po++ = *pb++;
    return (int) (po - out);
}

void print_all(const char *title, const long *a, int n)
{
    printf("%-9s", title);
    for (int i = 0; i < n; i++)
        printf(" %ld", a[i] / 1000);
    printf("\n");
}

int main(void)
{
    long morning[5] = {12000, 28000, 30000, 45000, 136500};
    long evening[4] = {18000, 30000, 56000, 70000};
    long day[9];
    int n = merge(morning, 5, evening, 4, day);
    print_all("morning:", morning, 5);
    print_all("evening:", evening, 4);
    print_all("day:", day, n);
    n = merge(morning, 5, evening, 0, day);   /* a closed evening shift */
    print_all("no eve:", day, n);
    return 0;
}
```

Listing 8.20 `merge_shifts.py`: the same merge in Python with two indexes; `extend` copies the rest of the unfinished list.

```
# merge_shifts.py - Problem 8.6: merge the sorted bills of the morning and
# the evening shift into one sorted list, walking both with indexes


def merge(a, b):
    out = []
    i = j = 0
    while i < len(a) and j < len(b):     # take the smaller front bill
        if a[i] <= b[j]:
            out.append(a[i])
            i += 1
        else:
            out.append(b[j])
            j += 1
    out.extend(a[i:])                    # one shift is finished:
    out.extend(b[j:])                    # copy the rest of the other
    return out


def print_all(title, a):
    print(f"{title:<9}" + "".join(f" {x // 1000}" for x in a))


def main():
    morning = [12000, 28000, 30000, 45000, 136500]
    evening = [18000, 30000, 56000, 70000]
    print_all("morning:", morning)
    print_all("evening:", evening)
    print_all("day:", merge(morning, evening))
    print_all("no eve:", merge(morning, []))   # a closed evening shift


if __name__ == "__main__":
    main()
```

```
morning:  12 28 30 45 136
evening:  18 30 56 70
day:      12 18 28 30 30 45 56 70 136
no eve:   12 28 30 45 136
```

The merged day is sorted, has all nine bills, and the empty evening shift simply returns the morning bills. For the nine bills of the example, Equation (8.6) allows at most 8 comparisons; the hand trace above made exactly 8.

8.8.4 A tempting wrong solution

The tempting version has only the first loop: while both piles have receipts, take the smaller. It passes a test in which both piles happen to run out together, and loses bills in every other case: with the example data, 136.5 would never be copied, and with an empty evening shift the result would be empty. The two small loops after the main one, which copy the rest of whichever pile is left, are the part that the tempting version forgets. The test with an empty array, which the problem statement demands, is exactly the test that exposes it.

> **CHECK YOURSELF.** What goes wrong if the comparison is written `*pa < *pb` instead of `*pa <= *pb`?
>
> *Answer.* The output is still sorted and complete, so every test of the numbers alone passes. What changes is the order of equal bills: when the two front bills are equal, the evening bill is taken first. The merge is no longer stable. For bills alone nobody can tell; for bills that carry a

> receipt number or a shift name (Chapter 10), equal bills would come out with the evening orders before the morning ones.

8.9 The Problems Side by Side

Table 8.4 collects what the six problems teach: the cost of each solution and the trap that the tempting wrong solution fell into, with the test that exposes it.

Table 8.4 The six problems side by side: the cost of each solution, the trap of its tempting wrong solution, and the test input that exposes the trap.

Problem	Cost	The trap	The test that exposes it
8.1 Statistics	n steps per function	`min` starting at 0; whole-number average	positive values only; a mean that is not whole
8.2 Fibonacci	loop n; recursion $2F(n+1)-1$	sequence starting at 1, 1	the given sample, $n = 9$
8.3 Palindrome	at most $n/2$ comparisons	partner at `n - i`	any palindrome of two or more letters
8.4 Determinant	a fixed 14 operations	sign of the middle term	two equal rows (the identity does not catch it)
8.5 Receipt	at most $\lfloor \log_2 n \rfloor + 1$ calls	recursing on `mid`	a two-element part, number on the right
8.6 Merge	$n_a + n_b$ steps	forgetting the leftovers	an empty array; unequal lengths

Read the last column once more. In every case the exposing test is an *edge*: a value at a boundary, an empty input, a repeated value, a sample computed by hand. Kernighan and Pike call this *testing at the boundaries*, and it is the single most productive testing habit a programmer can have [6].

8.10 Common Mistakes, and How to Avoid Them

- **Changing given code.** When prototypes and `main` are given, they are an agreement. Complete the bodies; do not rename or "improve" what was given.
- **Casting the result instead of an operand.** `(double) (sum / n)` casts after the whole-number division has thrown the fraction away. Write `(double) sum / n`.
- **Starting a minimum or maximum at 0.** Start at `INT_MAX`/`INT_MIN` or at the first element.
- **Mixing signed and unsigned counters.** Compare `unsigned int` sizes with `unsigned int` counters; `-Wextra` tells you where.
- **Skipping the sample.** Compute the given example by hand first; it catches convention errors such as the Fibonacci start.
- **Recursion on large inputs without counting.** Count the calls, or reason with a recurrence, before you trust a recursion with $n = 40$.
- **An off-by-one partner index.** In an array of length `n`, the partner of `i` is `n - 1 - i`; `s[n]` is the terminating zero of a string.
- **`&` on an array in `scanf`.** `scanf("%99s", word)` needs no `&`, and always give a width.
- **Weak test matrices.** The identity matrix hides sign errors; test with a matrix in which every entry matters and with one whose determinant must be 0.
- **A recursion that does not shrink.** Each recursive call must work on a strictly smaller part; exclude what has been checked.
- **Forgetting the rest after a merge.** After the main loop, copy what remains of each input.

8.11 Chapter Summary

- Every problem follows eight steps: restate, solve by hand, design on paper, estimate the cost, code, test, find the tempting wrong solution, look back. The hand-worked example is the step that decides what "correct" means.
- Problem 8.1 completed five statistics functions inside a fixed skeleton; the mean Equation (8.1) needs a cast before the division, and the standard deviation Equation (8.2) is the square root of the mean squared deviation, worked by hand in **Table 8.2**.
- Problem 8.2 computed Fibonacci numbers with a loop in n steps and with recursion in $2F(n+1)-1$ calls, used the identity Equation (8.3) for the sum, and measured the difference (**Figure 8.1**, **Table 8.3**): a better algorithm beats a faster language.
- Problem 8.3 decided palindromes with two pointers, with recursion and with XOR; the partner of position `i` is `n - 1 - i`.
- Problem 8.4 computed 3×3 determinants by Equation (8.5), decided invertibility, and showed that the identity matrix cannot catch a sign error.
- Problem 8.5 wrote binary search as a recursion whose every call works on a strictly smaller part; Problem 8.6 merged two sorted arrays with walking pointers in $n_a + n_b$ steps and at most Equation (8.6) comparisons.
- In every problem, the test that exposes the wrong solution is an edge case: **Table 8.4**.

8.12 Exercises

1. **Statistics, extended.** Add a sixth function, `double median(int *a, unsigned int aSize)`, to Problem 8.1 without changing the given prototypes, and compute by hand what it returns for the ten values. It must not reorder the caller's array.
2. **Fibonacci without the sum trick.** Write a recursive function `long long fib_sum_rec(int n)` that computes $\sum_{i=0}^{n} F(i)$ directly from the definition, $S(n) = S(n-1) + F(n)$, and count its calls for $n = 9$. Compare with the 287 calls of `fib_rec_sum(9)`.
3. **Palindromic numbers.** A receipt number such as 12321 is a palindrome. Write `int is_pal_number(long n)` that decides it *without* converting the number to a string, by building the reversed number with `% 10` and `/ 10`. Which numbers ending in 0 can be palindromes?
4. **Sarrus.** For 3×3 matrices, the rule of Sarrus writes the determinant as $aei + bfg + cdh - ceg - bdi - afh$. Show that it agrees with Equation (8.5) by expanding both, and compute the determinant of Mbak Rina's receipts, Equation (8.4), with it.
5. **Recovering the prices.** Cramer's rule says that the first unknown of the system $A\mathbf{x} = \mathbf{b}$ is $x_1 = \det A_1 / \det A$, where A_1 is A with its first column replaced by $\mathbf{b}$, and likewise for the others. Use it to find the prices of detergent, softener and starch from the three receipts of Section 8.6.1 (totals Rp 60,000, Rp 72,000 and Rp 20,000).
6. **Merge three shifts.** A third, night shift is added. Merge three sorted arrays into one, first by merging two and then the result with the third, and count the steps for shifts of sizes n_1, n_2, n_3. Does the order in which you merge them matter?
7. **Iterative binary search from the recursive one.** Turn `find` of Problem 8.5 into a loop by replacing each recursive call with an update of `low` or `high`. Why is this always possible for a function whose recursive call is the last thing it does?

8.13 Solutions to the Exercises

1. Copy the array into a local array of 10 elements (or of a fixed capacity), sort the copy with bubble sort or insertion sort from Chapter 6, and return the middle: for even `aSize`, the mean of the elements at `aSize / 2 - 1` and `aSize / 2`, as a `double`. The sorted values are 7, 8, 9, 10, 11, 12, 14, 15, 18, 20, so the median is $(11 + 12)/2 = 11.5$, a little below the mean 12.4 because the two biggest visits, 18 and 20 kg, pull the mean up.

2. `if (n < 0) return 0; return fib_sum_rec(n - 1) + fib_rec(n);`. The calls to `fib_sum_rec` itself are $n + 2$ (from n down to -1), 11 for $n = 9$, and each level also calls `fib_rec(i)`, with $2F(i + 1) - 1$ calls, for i = 0 to 9: $\sum_{i=0}^{9}(2F(i + 1) - 1) = 2 \cdot (F(1) + \cdots + F(10)) - 10 = 2 \cdot 143 - 10 = 276$. Together 287 calls, exactly as many as `fib_rec_sum(9)`, and not by coincidence: in general the direct method makes $(n + 2) + (2F(n + 3) - n - 3) = 2F(n + 3) - 1$ calls, which is $C(n + 2)$, the cost of `fib_rec(n + 2)`. Both ways are exponential; only the loop is not.

3. Build the reverse: `long r = 0, m = n; while (m > 0) { r = r * 10 + m % 10; m /= 10; } return r == n;`. For 12321 the loop builds 1, 12, 123, 1232, 12321, equal to n. A number ending in 0 would need to start with 0 to be a palindrome, which no number written without leading zeros does; the only such palindrome is 0 itself (and negative numbers are not palindromes because of the minus sign).

4. Expanding Equation (8.5): $a(ei - fh) - b(di - fg) + c(dh - eg) = aei - afh - bdi + bfg + cdh - ceg$, which is Sarrus' six terms in another order. For Equation (8.4), with a, b, c = 2,1,1; d, e, f = 1,3,2; g, h, i = 1,0,0: $aei + bfg + cdh = 0 + 2 + 0 = 2$ and $ceg + bdi + afh = 3 + 0 + 0 = 3$, so det = $2 - 3 = -1$, as before.

5. $\det A = -1$. Replacing the first column by (60000,72000,20000): $\det A_1 = 60000(3 \cdot 0 - 2 \cdot 0) - 1(72000 \cdot 0 - 2 \cdot 20000) + 1(72000 \cdot 0 - 3 \cdot 20000) = 0 + 40000 - 60000 = -20000$, so detergent costs \$-20000 / -1 = \$ Rp 20,000, which the third receipt confirms. Replacing the second column: $\det A_2 = 2(72000 \cdot 0 - 2 \cdot 20000) - 60000(1 \cdot 0 - 2 \cdot 1) + 1(1 \cdot 20000 - 72000 \cdot 1) = -80000 + 120000 - 52000 = -12000$, softener Rp 12,000. Replacing the third: $\det A_3 = 2(3 \cdot 20000 - 72000 \cdot 0) - 1(1 \cdot 20000 - 72000 \cdot 1) + 60000(1 \cdot 0 - 3 \cdot 1) = 120000 + 52000 - 180000 = -8000$, starch Rp 8,000. Check with the first receipt: $2 \cdot 20000 + 12000 + 8000 = 60000$.

6. Merging the first two takes $n_1 + n_2$ steps and produces $n_1 + n_2$ bills; merging that with the third takes $n_1 + n_2 + n_3$ steps. Total $2n_1 + 2n_2 + n_3$. The order matters: the two arrays merged first are copied twice, so merge the two *smallest* first. With sizes 50, 40 and 5, merging 40 and 5 first costs $45 + 95 = 140$ steps; merging 50 and 40 first costs $90 + 95 = 185$.

7. Replace `return find(a, low, mid - 1, key, depth + 1);` by `high = mid - 1;` and the other call by `low = mid + 1;`, wrap the body in `while (low <= high)`, and return -1 after the loop; the result is the `binary_search` of Chapter 6. It is always possible when the recursive call is the last action (a *tail call*), because nothing remains to be done after it returns: the call only changes the arguments and starts over, which is what one turn of a loop does.

References

[1] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, Princeton Science Library ed. Princeton, NJ, USA: Princeton University Press, 2014.

[2] Python Software Foundation, “statistics — Mathematical statistics functions,” Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/statistics.html

[3] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[4] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

[5] G. Strang, *Introduction to Linear Algebra*, 6th ed. Wellesley, MA, USA: Wellesley-Cambridge Press, 2023.

[6] B. W. Kernighan and R. Pike, *The Practice of Programming*. Reading, MA, USA: Addison-Wesley, 1999.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

Chapter 9
Characters, Strings and Formatted I/O

Almost everything a program receives from people is text: a name typed at the counter, a phone number copied from a chat, a command, a line of a file. Almost everything it gives back is text as well: a receipt, a table, a message. This chapter teaches you how C stores text, which turns out to be surprisingly simple, and how to handle it without the mistakes that have made string handling the most famous source of bugs in the history of C. By the end of the chapter you will be able to explain what a string is in memory and why it ends with a zero byte; classify and convert single characters with `<ctype.h>`; use the everyday functions of `<string.h>` and state the one rule that keeps each of them safe; control every column of your output with the flags, widths and precisions of `printf`; read whole lines safely with `fgets` and pick them apart; and build version 0.7 of the Profitina Laundry cashier, which understands typed commands, writes every customer's name the same way, and finds a customer from part of a name. As always, every program is given in C and in Python, and both print exactly the same output.

9.1 Where We Left Off: Mbak Rina's Names

By the end of Chapter 7 the Profitina Laundry cashier could compute bills, keep a day's orders in arrays, and return several results from one function through pointers. Chapter 8 then put those tools to work on a set of complete problems. But one thing the cashier still cannot do is deal with *names*. It numbers its orders 1, 2, 3, and Mbak Rina, who runs the counter, has to keep a paper notebook next to the computer to remember which number belongs to which customer.

Look at a page of that notebook and you will see why a program would help. The same regular customer appears as "andi", "ANDI PRATAMA", "Andi Pratama" with two spaces, and "andi p." in four different handwritings, because four different people worked the counter that week. Phone numbers are written as "0812-3456-7890", "0812 3456 7890" and "+62 812 3456 7890". When Andi comes back on Friday to collect his laundry and says "Pratama", Mbak Rina leafs through three pages to find him. And the receipts that the cashier prints have columns that wander: a long name pushes the price to the right, a short one pulls it to the left, and customers squint to find the total.

Every one of these problems is a problem of *text*. Tidying a name means looking at its letters one by one and changing some of them. Cleaning a phone number means keeping the digits and throwing the rest away. Finding "Pratama" inside "Andi Pratama" is a search for one piece of text inside another. Lining up a receipt means telling `printf` exactly how many columns each value may use. And understanding a command typed at the keyboard, such as `add fold 3.5 andi pratama`, means cutting a line into words and deciding what each word means.

C has a reputation for being awkward with text, and there is some truth in it: there is no built-in "string type" that grows and shrinks by itself as in Python. But the idea behind C's strings is small enough to understand completely in one section, and once you understand it, every string function in the library becomes an easy loop that you could have written yourself. That is how this chapter proceeds: first the idea, then the character and string libraries, then formatted output and safe input, and finally cashier v0.7, which uses all of them.

9.2 The Idea in Plain Words: Text Is Numbers

9.2.1 A character is a small number

A computer stores only numbers. To store the letter `A`, it stores a number that everybody has agreed stands for `A`. The agreement used by C on every computer you are likely to meet is ASCII, the American Standard Code for Information Interchange, which gives the numbers 0 to 127 to the English letters, the digits, the punctuation marks, the space and a few control characters such as the newline [1], [2]. In ASCII, `A` is 65, `B` is 66 and so on up to `Z`, which is 90; the small letters start at `a`, which is 97; the digit `0` is 48, `1` is 49, up to `9`, which is 57; the space is 32 and the newline is 10.

In C, the type `char` holds one such number, and a character constant written in single quotes, such as `'A'`, is simply another way of writing the number 65. You can add to it, compare it and subtract it like any other integer. Two small facts follow that you will use again and again:

- The digits are consecutive, so the *value* of a digit character `c` is `c - '0'`. For example, `'7' - '0'` is $55 - 48 = 7$.
- The small letters are consecutive and the capital letters are consecutive, and each small letter lies exactly 32 places above its capital: `'a' - 'A'` is $97 - 65 = 32$. Converting case is therefore an addition or a subtraction, although, as Section 9.3 shows, the library does it for you more safely.

Names such as “Dewi Anggraini” fit in ASCII. Names with letters outside English, such as “Chloé”, need more than one byte per letter in the UTF-8 encoding that modern systems use. This chapter works with ASCII text, which covers everything the laundry types; Section 9.11 says what changes for other scripts.

9.2.2 A string is an array that ends with a zero

A *string* in C is not a special type. It is an array of `char` that holds the codes of the characters one after another, followed by one extra byte with the value 0, written `'\0'` and called the *null terminator* or *null character*. The terminator is the whole trick: it marks where the text ends, so no separate length needs to be stored.

> **DEFINITION.** A *string* in C is a sequence of characters stored in consecutive bytes of a `char` array and ended by the null character `'\0'` (the byte with value 0). A *string literal* is text written between double quotes in the program, such as `"Andi"`; the compiler stores it with the terminator already added. The *length* of a string is the number of characters before the terminator.

Figure 9.1 shows the two arrays of the first program of this chapter. The declaration `char name[] = "Andi";` makes an array just large enough for the literal: four letters plus the terminator, five bytes. The declaration `char buffer[16] = "Bella";` asks for sixteen bytes; the first six receive `B`, `e`, `l`, `l`, `a` and the terminator, and because an array initialized with fewer values than its size gets zeros in the rest, the other ten bytes are zero as well.

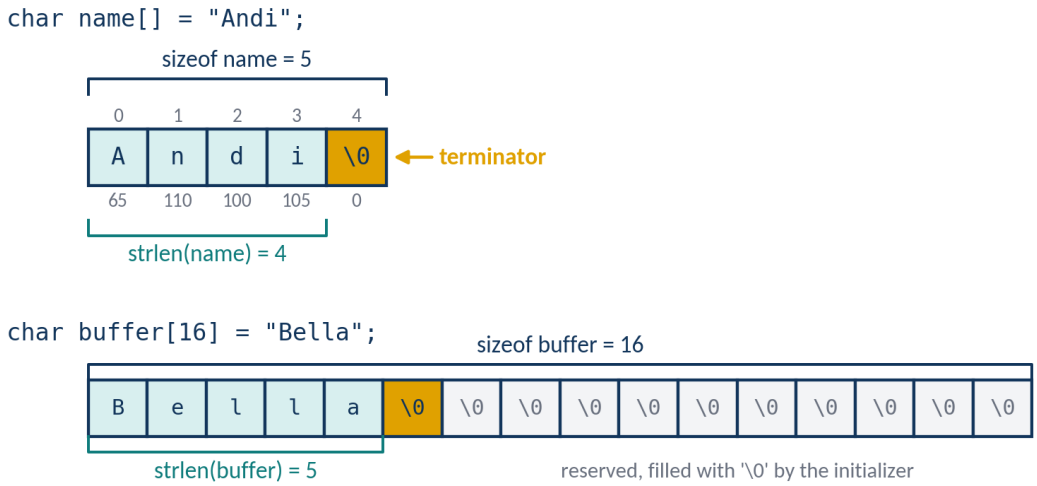


Figure 9.1 Two strings in memory: `name` holds four letters and the terminator in exactly five bytes, while `buffer` reserves sixteen bytes of which the string uses six. The grey numbers under `name` are the ASCII codes.

Two questions about a string sound alike and have different answers. How many bytes does the array occupy? That is `sizeof`, which the compiler knows from the declaration: 5 for `name`, 16 for `buffer`. How long is the text? That is `strlen`, a function that walks along the array counting characters until it meets the terminator: 4 for `name`, 5 for `buffer`. Written as a formula for an array declared from a literal of n characters,

$$\text{sizeof name} = n + 1, \qquad \text{strlen(name)} = n \tag{9.1}$$

and Equation (9.1) is worth remembering, because forgetting the "+ 1" for the terminator is the most common way to make an array one byte too small.

Every string function in C, without exception, is a loop that walks until it meets `'\0'`. **Listing 9.1** writes one such loop by hand, `my_strlen`, and then looks at the arrays of **Figure 9.1** byte by byte.

Listing 9.1 `string_basics.c`: a string is a `char` array ending in `'\0'`; `sizeof` counts bytes, `strlen` counts characters.

```c
/* string_basics.c - a C string is an array of char that ends with '\0'. */
#include <stdio.h>
#include <string.h>

/* Count the characters before the terminator, exactly as strlen does. */
static size_t my_strlen(const char *s)
{
    size_t n = 0;
    while (s[n] != '\0')
        n++;
    return n;
}

int main(void)
{
    char name[] = "Andi";            /* 5 bytes: 'A' 'n' 'd' 'i' '\0' */
    char buffer[16] = "Bella";       /* 16 bytes reserved, 6 of them used */
    const char *shop = "Profitina Laundry";  /* points at a read-only literal */

    printf("name   = \"%s\"  sizeof = %zu  strlen = %zu\n",
           name, sizeof name, strlen(name));
    printf("buffer = \"%s\"  sizeof = %zu  strlen = %zu\n",
           buffer, sizeof buffer, strlen(buffer));
    printf("shop   = \"%s\"  my_strlen = %zu\n", shop, my_strlen(shop));

    printf("bytes of name:");
    for (size_t i = 0; i < sizeof name; i++)
        printf(" %d", name[i]);
    printf("\n");

    name[0] = 'a';                   /* an array holds its own copy: we may edit */
    printf("after name[0] = 'a': %s\n", name);

    buffer[3] = '\0';                /* an earlier terminator cuts the string */
    printf("after buffer[3] = '\\0': %s (strlen %zu)\n", buffer, strlen(buffer));

    printf("'A' = %d, 'a' = %d, '0' = %d, ' ' = %d\n", 'A', 'a', '0', ' ');
    printf("'7' - '0' = %d, 'a' - 'A' = %d\n", '7' - '0', 'a' - 'A');
    return 0;
}
```

Compile and run it the way Chapter 1 showed (`gcc -std=c17 -O2 -Wall -Wextra -o string_basics string_basics.c`, then `./string_basics`). It prints:

```
name   = "Andi"  sizeof = 5  strlen = 4
buffer = "Bella"  sizeof = 16  strlen = 5
shop   = "Profitina Laundry"  my_strlen = 17
bytes of name: 65 110 100 105 0
after name[0] = 'a': andi
after buffer[3] = '\0': Bel (strlen 3)
'A' = 65, 'a' = 97, '0' = 48, ' ' = 32
'7' - '0' = 7, 'a' - 'A' = 32
```

Read the output line by line against the code. The byte dump `65 110 100 105 0` is **Figure 9.1** in numbers: the four codes and the terminator. The line `name[0] = 'a'` changes the first byte of the array, and printing it again shows `andi`; this is allowed because `name` is an array that holds its *own copy* of the letters. Writing `buffer[3] = '\0'` puts a terminator in the middle of `Bella`: nothing after it has been erased, `l`, `a` and the old terminator are still there, but every string function now stops at index 3 and sees only `Bel`. The last two lines confirm the arithmetic of Section 9.2.1.

> **WARNING.** `const char *shop = "Profitina Laundry";` does not make an array; it makes a *pointer* (Chapter 7) to a literal that the compiler has stored somewhere, often in memory that may not be written. `shop[0] = 'p';` compiles but is undefined behavior and usually crashes.

> Declare such pointers `const`, as the listing does, and the compiler will refuse the assignment. When you need text you may change, declare an array.

9.2.3 The same program in Python

Python stores strings differently: a Python string is an object that knows its own length and never needs a terminator, and it can never be changed after it is made (strings are *immutable*). To change the first letter you build a new string from pieces. **Listing 9.2** prints the same lines as the C program by showing what C would store; the `sizeof` values are computed as the text length plus one, which is exactly Equation (9.1).

Listing 9.2 `string_basics.py`: the same lines in Python, whose strings know their length and cannot be changed in place.

```
# string_basics.py - Python strings know their own length; C strings end in '\0'.
# This version prints the same lines as string_basics.c by showing the bytes
# a C program would store.


def my_strlen(s):
    """Count characters up to the (imagined) terminator, like strlen."""
    n = 0
    for _ in s:
        n += 1
    return n


def main():
    name = "Andi"                  # C stores 5 bytes: 'A' 'n' 'd' 'i' '\0'
    buffer = "Bella"               # C reserved 16 bytes for it
    shop = "Profitina Laundry"

    print('name   = "%s"  sizeof = %d  strlen = %d'
          % (name, len(name) + 1, len(name)))
    print('buffer = "%s"  sizeof = %d  strlen = %d' % (buffer, 16, len(buffer)))
    print('shop   = "%s"  my_strlen = %d' % (shop, my_strlen(shop)))

    raw = name.encode("ascii") + b"\0"
    print("bytes of name:" + "".join(" %d" % b for b in raw))

    name = "a" + name[1:]          # Python strings are immutable: build a new one
    print("after name[0] = 'a': %s" % name)

    buffer = buffer[:3]            # what an early '\0' does in C
    print("after buffer[3] = '\\0': %s (strlen %d)" % (buffer, len(buffer)))

    print("'A' = %d, 'a' = %d, '0' = %d, ' ' = %d"
          % (ord("A"), ord("a"), ord("0"), ord(" ")))
    print("'7' - '0' = %d, 'a' - 'A' = %d"
          % (ord("7") - ord("0"), ord("a") - ord("A")))


if __name__ == "__main__":
    main()
```

Run it with `python3 string_basics.py` (on Windows, `python` or `py`). The output is identical to the C program's, byte for byte; the test runner of Section 9.9 checks that for every program of the chapter.

9.3 One Character at a Time: `<ctype.h>`

Before we handle whole strings, we need tools for single characters. Is this character a letter? A digit? A space? What is its capital form? The header `<ctype.h>` answers these questions with a family of small functions, listed in **Table 9.1** [2], [3]. Each takes one character and returns a true (non-zero) or false (zero) answer, or the converted character.

Table 9.1 The most used functions of `<ctype.h>`. Characters that are not letters pass through `toupper` and `tolower` unchanged.

Function	True for	Examples that are true
`isalpha(c)`	a letter	`'A'`, `'z'`
`isdigit(c)`	a decimal digit	`'0'` to `'9'`
`isalnum(c)`	a letter or a digit	`'b'`, `'7'`
`isspace(c)`	a white-space character	`' '`, `'\t'`, `'\n'`, `'\r'`
`isupper(c)` / `islower(c)`	a capital / small letter	`'Q'` / `'q'`
`ispunct(c)`	a punctuation mark	`'!'`, `'-'`, `','`
`toupper(c)` / `tolower(c)`	(returns the converted letter)	`toupper('a')` is `'A'`

One detail looks fussy but matters. These functions take an `int` that must be either the value `EOF` or a value that fits in an `unsigned char`, that is, 0 to 255. On most computers a plain `char` is *signed*, so a byte above 127 (which appears in UTF-8 text) is stored as a negative number, and passing it straight to `isalpha` is undefined behavior. The cure is a cast: `isalpha((unsigned char)c)`. Every program in this book does so, and so should yours [4].

Now the laundry's problems. **Listing 9.3** contains three small functions. `classify` counts letters, digits, spaces and other characters in a string, the kind of check a form makes before accepting input. `digits_only` copies only the digits of a phone number into a second array, which turns every way of writing a number into one canonical form, and `valid_mobile` then checks that the result looks like an Indonesian mobile number: it starts with `08` and has between 10 and 13 digits. `tidy_name` takes a name typed in any mixture of case and spacing and writes it in *title case*: no spaces at the ends, one space between words, the first letter of each word capital and the rest small.

Listing 9.3 `char_classes.c`: classifying characters, keeping the digits of a phone number, and tidying a name.

```c
/* char_classes.c - <ctype.h>: classify characters, clean a phone number and
 * tidy a customer name the way the Profitina Laundry counter receives them. */
#include <ctype.h>
#include <stdio.h>
#include <string.h>

/* Count letters, digits, spaces and everything else in s. */
static void classify(const char *s)
{
    int letters = 0, digits = 0, spaces = 0, others = 0;
    for (size_t i = 0; s[i] != '\0'; i++) {
        unsigned char c = (unsigned char)s[i];   /* ctype needs unsigned */
        if (isalpha(c))
            letters++;
        else if (isdigit(c))
            digits++;
        else if (isspace(c))
            spaces++;
        else
            others++;
    }
    printf("\"%s\": letters=%d digits=%d spaces=%d others=%d\n",
           s, letters, digits, spaces, others);
}

/* Copy only the digits of src into dst (size bytes); return how many. */
static size_t digits_only(const char *src, char *dst, size_t size)
{
    size_t n = 0;
    for (size_t i = 0; src[i] != '\0' && n + 1 < size; i++)
        if (isdigit((unsigned char)src[i]))
            dst[n++] = src[i];
    dst[n] = '\0';
    return n;
}

/* An Indonesian mobile number: starts with 08 and has 10 to 13 digits. */
static int valid_mobile(const char *digits)
{
    size_t n = strlen(digits);
    return n >= 10 && n <= 13 && digits[0] == '0' && digits[1] == '8';
}
```

Listing 9.3 (Continued) *char_classes.c: classifying characters, keeping the digits of a phone number, and tidying a name.*

```c
/* Trim, squeeze inner spaces to one, and write each word as Title case. */
static void tidy_name(const char *src, char *dst, size_t size)
{
    size_t n = 0;
    int new_word = 1;
    for (size_t i = 0; src[i] != '\0' && n + 1 < size; i++) {
        unsigned char c = (unsigned char)src[i];
        if (isspace(c)) {
            new_word = 1;
            continue;
        }
        if (new_word && n > 0 && n + 2 < size)
            dst[n++] = ' ';          /* one space between words */
        dst[n++] = (char)(new_word ? toupper(c) : tolower(c));
        new_word = 0;
    }
    dst[n] = '\0';
}

int main(void)
{
    const char *phones[] = {"0812-3456-7890", "+62 812 3456 7890", "0812 34"};
    const char *names[] = {"  aNDi   pratama ", "BUDI", "citra  dewi lestari"};
    char buf[32];

    classify("Order PL-0042: 3.5 kg!");
    for (int i = 0; i < 3; i++) {
        size_t n = digits_only(phones[i], buf, sizeof buf);
        printf("phone %-18s -> %-13s (%zu digits) %s\n", phones[i], buf, n,
               valid_mobile(buf) ? "ok" : "check with customer");
    }
    for (int i = 0; i < 3; i++) {
        tidy_name(names[i], buf, sizeof buf);
        printf("name  [%s] -> [%s]\n", names[i], buf);
    }
    return 0;
}
```

Look closely at the two copying functions, because their pattern appears in every safe string routine in this book. Each receives the destination array *and its size*, and the loop condition `n + 1 < size` makes sure that at least one byte is always left for the terminator. After the loop, `dst[n] = '\0'` ends the new string. A copying loop that forgets either half, the size check or the terminator, is a bug waiting for the first long input.

`tidy_name` uses a small *state variable*, `new_word`, that remembers whether the previous character was a space (or whether we are at the very start). Whenever a space arrives, the function only sets `new_word = 1` and copies nothing, so any run of spaces collapses into nothing. When a non-space arrives after spaces, the function first writes one single space, but only if something has already been written (`n > 0`), which is how the leading spaces disappear; then it writes the character in capitals if it starts a word and in small letters otherwise. Trailing spaces never produce anything because no character comes after them. This way of thinking, “what have I just seen, and what does that mean for the current character”, is the heart of all text processing.

The output:

```
"Order PL-0042: 3.5 kg!": letters=9 digits=6 spaces=3 others=4
phone 0812-3456-7890     -> 081234567890  (12 digits) ok
phone +62 812 3456 7890  -> 6281234567890 (13 digits) check with customer
phone 0812 34            -> 081234        (6 digits) check with customer
name  [  aNDi   pratama ] -> [Andi Pratama]
name  [BUDI] -> [Budi]
name  [citra  dewi lestari] -> [Citra Dewi Lestari]
```

The international form `+62 812 3456 7890` keeps its country code 62 and fails the check. That is deliberate: the program does not *guess* what a customer meant, it asks the cashier to check. Exercise 9.3 extends `digits_only` so that it converts a leading `62` into `0`.

The Python version, **Listing 9.4**, uses the string methods `isalpha`, `isdigit` and `isspace`, which behave like their C cousins for ASCII text, and it builds new strings instead of filling arrays. `tidy_name` becomes one line: `src.split()` with no argument splits at any run of white space and drops empty pieces, which is exactly the trimming and squeezing the C loop does by hand.

Listing 9.4 `char_classes.py`: the same three tasks with Python's string methods.

```
# char_classes.py - classify characters, clean a phone number and tidy a
# customer name the way the Profitina Laundry counter receives them.


def classify(s):
    """Count letters, digits, spaces and everything else in s."""
    letters = digits = spaces = others = 0
    for c in s:
        if c.isalpha():
            letters += 1
        elif c.isdigit():
            digits += 1
        elif c.isspace():
            spaces += 1
        else:
            others += 1
    print('"%s": letters=%d digits=%d spaces=%d others=%d'
          % (s, letters, digits, spaces, others))


def digits_only(src, size=32):
    """Keep only the digits (at most size - 1 of them, as in the C buffer)."""
    return "".join(c for c in src if c.isdigit())[:size - 1]


def valid_mobile(digits):
    """An Indonesian mobile number: starts with 08 and has 10 to 13 digits."""
    return 10 <= len(digits) <= 13 and digits.startswith("08")


def tidy_name(src, size=32):
    """Trim, squeeze inner spaces to one, and write each word as Title case."""
    words = [w[0].upper() + w[1:].lower() for w in src.split()]
    return " ".join(words)[:size - 1]


def main():
    phones = ["0812-3456-7890", "+62 812 3456 7890", "0812 34"]
    names = ["  aNDi   pratama ", "BUDI", "citra  dewi lestari"]
```

Listing 9.4 (Continued) *char_classes.py: the same three tasks with Python's string methods.*

```
    classify("Order PL-0042: 3.5 kg!")
    for p in phones:
        d = digits_only(p)
        verdict = "ok" if valid_mobile(d) else "check with customer"
        print("phone %-18s -> %-13s (%d digits) %s" % (p, d, len(d), verdict))
    for n in names:
        print("name  [%s] -> [%s]" % (n, tidy_name(n)))


if __name__ == "__main__":
    main()
```

9.4 The String Library and Its Sharp Edges

The header `<string.h>` offers ready-made functions for the most common work on whole strings: measuring, copying, joining, comparing and searching [2], [3]. **Table 9.2** lists the ones you will use most, each with the rule that keeps it safe. The rules exist because none of these functions checks anything: they trust that the arrays you give them are large enough and that every string ends with its terminator.

Table 9.2 The everyday functions of `<string.h>` (and `snprintf` from `<stdio.h>`), each with its safety rule.

Function	What it does	The rule that keeps it safe
`strlen(s)`	number of characters before `'\0'`	`s` must be terminated, or the walk runs on through memory
`strcpy(dst, src)`	copies `src`, terminator included, into `dst`	`dst` must hold `strlen(src) + 1` bytes; prefer `snprintf`
`strcat(dst, src)`	appends `src` to the end of the string in `dst`	`dst` must hold both strings plus one byte
`snprintf(dst, size, fmt, ...)`	formats into `dst`, never writing more than `size` bytes	pass the true size; the return value tells how long the full text was
`strcmp(a, b)`	negative, zero or positive by character codes	zero means *equal*; never compare strings with `==`
`strchr(s, c)`	pointer to the first `c` in `s`, or `NULL`	check for `NULL` before using the result
`strstr(s, part)`	pointer to the first occurrence of `part`, or `NULL`	check for `NULL` before using the result
`strtok(s, delims)`	cuts `s` into pieces at the delimiter characters	it *writes* `'\0'` into `s`; skips empty fields; one string at a time

Listing 9.5 tries each of them on laundry text.

Listing 9.5 `string_library.c`: copying, joining, bounded formatting, comparing, searching and cutting strings.

```c
/* string_library.c - the everyday functions of <string.h>, each with the one
 * rule that keeps it safe. */
#include <stdio.h>
#include <string.h>

static int sign(int x) { return (x > 0) - (x < 0); }

int main(void)
{
    char dst[24];

    strcpy(dst, "Wash");                 /* copy: dst must be big enough */
    strcat(dst, " & fold");              /* append: same rule */
    printf("strcpy + strcat : \"%s\" (%zu of %zu bytes used)\n",
           dst, strlen(dst) + 1, sizeof dst);

    char wish[48];                       /* a longer text, built at run time */
    strcpy(wish, dst);
    strcat(wish, " plus ironing and delivery");
    int need = snprintf(dst, sizeof dst, "%s", wish);   /* never overflows */
    printf("snprintf        : \"%s\" (wanted %d chars, kept %zu)\n",
           dst, need, strlen(dst));

    printf("strcmp(\"Andi\", \"Andi\")  -> %d\n", sign(strcmp("Andi", "Andi")));
    printf("strcmp(\"Andi\", \"Budi\")  -> %d\n", sign(strcmp("Andi", "Budi")));
    printf("strcmp(\"andi\", \"Andi\")  -> %d\n", sign(strcmp("andi", "Andi")));
    printf("strcmp(\"Andi\", \"Andika\")-> %d\n", sign(strcmp("Andi", "Andika")));

    const char *code = "PL-0042";
    const char *dash = strchr(code, '-');          /* first '-' or NULL */
    printf("strchr          : number part \"%s\" at index %d\n",
           dash + 1, (int)(dash - code));

    const char *menu = "fold,wash-iron,iron";
    printf("strstr          : \"wash-iron\" %s, \"dry-clean\" %s\n",
           strstr(menu, "wash-iron") ? "found" : "missing",
           strstr(menu, "dry-clean") ? "found" : "missing");

    char row[] = "Andi Pratama,fold,3.5";      /* strtok edits this array */
    int field = 0;
    for (char *tok = strtok(row, ","); tok != NULL; tok = strtok(NULL, ","))
        printf("strtok field %d  : \"%s\"\n", ++field, tok);
    printf("row afterwards  : \"%s\" (the commas became '\\0')\n", row);
    return 0;
}
```

It prints:

```
strcpy + strcat : "Wash & fold" (12 of 24 bytes used)
snprintf        : "Wash & fold plus ironin" (wanted 37 chars, kept 23)
strcmp("Andi", "Andi")  -> 0
strcmp("Andi", "Budi")  -> -1
strcmp("andi", "Andi")  -> 1
strcmp("Andi", "Andika")-> -1
strchr          : number part "0042" at index 2
strstr          : "wash-iron" found, "dry-clean" missing
strtok field 1  : "Andi Pratama"
strtok field 2  : "fold"
strtok field 3  : "3.5"
row afterwards  : "Andi Pratama" (the commas became '\0')
```

Walk through the results in order.

Copying and joining. After `strcpy` and `strcat`, `dst` holds `Wash & fold`: 11 characters plus the terminator, 12 of its 24 bytes. That was safe only because we counted. The next line shows the tool that makes counting unnecessary: `snprintf(dst, sizeof dst, "%s", wish)` formats the

37-character text but never writes more than `sizeof dst`, that is 24, bytes. It keeps the first 23 characters and the terminator, and it returns 37, the length the full text *would* have had, so the program can notice that something was cut. Truncating a customer's note is annoying; writing 37 bytes into a 24-byte array is a disaster, as the warning below explains.

Comparing. `strcmp(a, b)` compares the two strings character by character, from the left, and stops at the first position where they differ. If `a`'s character there has the smaller code, the result is negative; if larger, positive; if the strings are equal to the end, zero. The standard promises only the *sign* of the result, not its size, which is why the program prints the sign. `"Andi"` before `"Budi"` is negative because 65 is less than 66. `"andi"` after `"Andi"` is positive because small letters have larger codes than capitals: to a computer, `z` comes after `Z`. And `"Andi"` before `"Andika"` is negative because when one string runs out first, its terminator 0 is compared with `k`, and 0 is smaller: a prefix comes first, as in a dictionary.

Searching. `strchr(code, '-')` returns a *pointer* to the dash inside `"PL-0042"`, not an index; subtracting the start of the string, `dash - code`, turns it into the index 2 (pointer arithmetic from Chapter 7), and `dash + 1` is a pointer to the rest of the text, `"0042"`, which `printf` prints like any other string. `strstr` looks for a whole piece of text. Both return `NULL` when they find nothing, and the program tests for that before using the result.

Cutting. `strtok` finds the first comma, writes a terminator over it, and returns a pointer to the first piece; each later call with `NULL` continues from where the previous one stopped. The last line of the output shows the price: the array `row` has been changed, its commas are now terminators, and printing it shows only the first field. That is why `strtok` may never be used on a string literal, and why Chapter 11 writes its own splitter for data files, where empty fields such as `Andi,,3.5` must be kept and `strtok` would silently skip them.

> **WARNING.** Copying more bytes into an array than it can hold does not stop the program with an error. The extra bytes are written over whatever lies next in memory: another variable, the return address of the function, the data of another customer. This *buffer overflow* has been one of the most exploited programming errors for decades. In the 2025 list of the 25 most dangerous software weaknesses published by MITRE, *out-of-bounds write* is still in fifth place and the *classic buffer overflow*, a copy without checking the size of the input, in eleventh [5]. The rule of this book: every copy into an array states the size of that array, through a size check in your own loop or through `snprintf`.

The Python version, **Listing 9.6**, shows the same ideas with Python's operators. Strings are joined with `+`, compared with `<`, `==` and `>` (the comparison is by character codes, exactly like `strcmp`), searched with `find` and `in`, and cut with `split`. Python cannot overflow a string, because a new, larger string is made whenever needed; to print the same "kept 23" line, the program cuts the text to 23 characters itself.

Listing 9.6 `string_library.py`: Python's counterparts of the `<string.h>` functions.

```
# string_library.py - the Python counterparts of the <string.h> functions,
# printing the same lines as string_library.c.


def sign(x):
    return (x > 0) - (x < 0)


def cmp(a, b):
    """strcmp: negative, zero or positive by character codes."""
    return (a > b) - (a < b)


def main():
    size = 24
    dst = "Wash"
    dst += " & fold"
    print('strcpy + strcat : "%s" (%d of %d bytes used)'
          % (dst, len(dst) + 1, size))

    full = dst + " plus ironing and delivery"     # a longer text
    dst = full[:size - 1]                 # what snprintf keeps
    print('snprintf        : "%s" (wanted %d chars, kept %d)'
          % (dst, len(full), len(dst)))

    print('strcmp("Andi", "Andi")  -> %d' % sign(cmp("Andi", "Andi")))
    print('strcmp("Andi", "Budi")  -> %d' % sign(cmp("Andi", "Budi")))
    print('strcmp("andi", "Andi")  -> %d' % sign(cmp("andi", "Andi")))
    print('strcmp("Andi", "Andika")-> %d' % sign(cmp("Andi", "Andika")))

    code = "PL-0042"
    i = code.find("-")                    # -1 plays the role of NULL
    print('strchr          : number part "%s" at index %d' % (code[i + 1:], i))

    menu = "fold,wash-iron,iron"
    print('strstr          : "wash-iron" %s, "dry-clean" %s'
          % ("found" if "wash-iron" in menu else "missing",
             "found" if "dry-clean" in menu else "missing"))

    row = "Andi Pratama,fold,3.5"
    for field, tok in enumerate(row.split(","), start=1):
        print('strtok field %d  : "%s"' % (field, tok))
    print('row afterwards  : "%s" (the commas became \'\\0\')' % row.split(",")[0])


```

Listing 9.6 (Continued) *string_library.py: Python's counterparts of the <string.h> functions.*

```
if __name__ == "__main__":
    main()
```

9.5 Formatted Output: the Full Dial of `printf`

You have used `%d`, `%f` and `%s` since Chapter 1. Each of these is the last letter of a *conversion specification*, and a specification may carry up to four more parts between the `%` and that letter. **Figure 9.2** takes one apart.

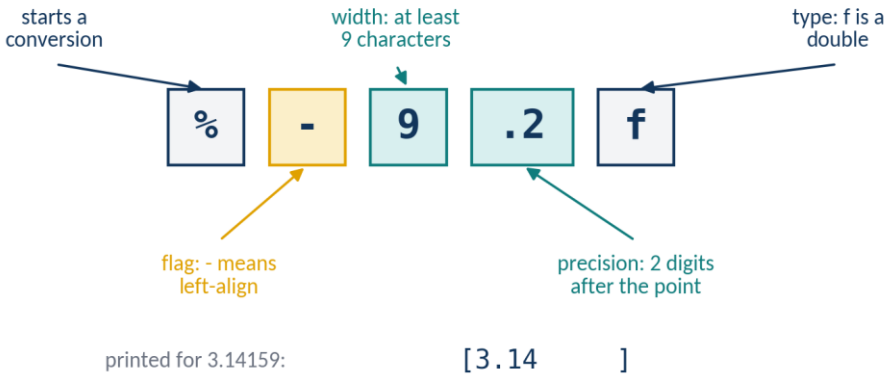


Figure 9.2 The parts of the conversion specification `%-9.2f`: the flag `-` left-aligns, the width 9 is the minimum number of characters, the precision 2 gives the digits after the decimal point, and the type `f` says that a `double` is printed.

In general a specification has the form

$$\%\ [\text{flags}]\ [\text{width}]\ [.\text{precision}]\ \text{type} \tag{9.2}$$

where the parts in brackets are optional [2], [6]. The *width* is a minimum: a value that needs fewer characters is padded with spaces (on the left, unless the flag `-` asks for the right), and a value that needs more is never cut, it simply pushes the rest of the line along. The *precision* means different things for different types: digits after the point for `f` and `e`, significant digits for `g`, and the *maximum* number of characters for `s`. The *flags* are `-` (left-align), `0` (pad numbers with zeros), `+` (always print a sign), a space (a space where a plus sign would go) and `#` (the "alternate form", for example `0x` before hexadecimal). A width or precision written as `*` is taken from the argument list, which lets the program compute column widths while it runs.

Listing 9.7 turns the knobs of Equation (9.2) one at a time. Every field is printed between square brackets, so that you can see exactly where it starts and ends, spaces included.

Listing 9.7 `format_dial.c`: flags, widths, precisions and types of `printf`, one at a time.

```c
/* format_dial.c - the conversion specification %[flags][width][.precision]type
 * of printf, turned one knob at a time. Brackets show where each field ends. */
#include <stdio.h>

int main(void)
{
    printf("[%d] [%6d] [%-6d] [%06d] [%+d]\n", 42, 42, 42, 42, 42);
    printf("[%f] [%.2f] [%9.2f] [%-9.2f]\n", 3.14159, 3.14159, 3.14159, 3.14159);
    printf("[%.2f] [%.2f] [%.1f]\n", 2.675, 1.005, 0.25);
    printf("[%e] [%.3e] [%g] [%g]\n", 1234.5, 0.000123, 1234.5, 0.0000125);
    printf("[%s] [%10s] [%-10s] [%.3s]\n", "Andi", "Andi", "Andi", "Andika");
    printf("[%c] [%3c] [%%]\n", 'A', 'B');
    printf("[%x] [%X] [%o] [%#x]\n", 255, 255, 8, 255);
    printf("[%*d] [%-*s]\n", 8, 42, 8, "left");
    int n = printf("Rp %d\n", 28000);
    printf("the line above had %d characters, newline included\n", n);
    return 0;
}
```

```
[42] [    42] [42    ] [000042] [+42]
[3.141590] [3.14] [     3.14] [3.14     ]
[2.67] [1.00] [0.2]
[1.234500e+03] [1.230e-04] [1234.5] [1.25e-05]
[Andi] [      Andi] [Andi      ] [And]
[A] [  B] [%]
[ff] [FF] [10] [0xff]
[      42] [left    ]
Rp 28000
the line above had 9 characters, newline included
```

Most lines are the knobs doing what **Figure 9.2** says. Three deserve a comment.

The third line is a lesson about money. `2.675` printed with two decimals gives `2.67`, not `2.68`, and `1.005` gives `1.00`. `printf` is not wrong: the numbers 2.675 and 1.005 cannot be stored exactly in binary, and the nearest `double` values are 2.67499999999999982236431605997495353221893310546875 and 1.00499999999999989341858963598497211933135986328125, which do round down [7]. `0.25` *is* stored exactly, and there the rule "round half to even" gives `0.2`. This is why every program in this book keeps rupiah in whole numbers of type `int` or `long` and weights in whole tenths of a kilogram: integers are exact, and a cashier must never lose a rupiah to rounding.

The `%g` type chooses by itself between the plain and the scientific notation, whichever is shorter, and drops trailing zeros. And the return value of `printf` is the number of characters it wrote; the program uses it to show that `"Rp 28000\n"` is 9 characters, the newline included.

In Python, the `%` operator on strings understands the same conversion specifications, so **Listing 9.8** is almost a copy. Python's `format` method and f-strings use a different mini-language (`{:<9.2f}`) with the same ideas; this book uses `%` because it matches C exactly.

Listing 9.8 `format_dial.py`: the same specifications with Python's `%` operator.

```
# format_dial.py - Python's % operator uses the same conversion specifications
# as printf, so every line below matches format_dial.c.


def main():
    print("[%d] [%6d] [%-6d] [%06d] [%+d]" % (42, 42, 42, 42, 42))
    print("[%f] [%.2f] [%9.2f] [%-9.2f]" % (3.14159, 3.14159, 3.14159, 3.14159))
    print("[%.2f] [%.2f] [%.1f]" % (2.675, 1.005, 0.25))
    print("[%e] [%.3e] [%g] [%g]" % (1234.5, 0.000123, 1234.5, 0.0000125))
    print("[%s] [%10s] [%-10s] [%.3s]" % ("Andi", "Andi", "Andi", "Andika"))
    print("[%c] [%3c] [%%]" % ("A", "B"))
    print("[%x] [%X] [%o] [%#x]" % (255, 255, 8, 255))
    print("[%*d] [%-*s]" % (8, 42, 8, "left"))
    line = "Rp %d\n" % 28000
    print(line, end="")
    print("the line above had %d characters, newline included" % len(line))


if __name__ == "__main__":
    main()
```

9.6 A Receipt That Lines Up

Now we can give Mbak Rina a receipt whose columns never wander. **Listing 9.9** prints one for an order of three services. It introduces two helper functions that are pure string work and that the rest of the book keeps using.

`rupiah` writes a number with a dot between groups of three digits, the Indonesian way: 1250000 becomes `1.250.000`. It first lets `snprintf` turn the number into its digits, then copies the digits into the result, inserting a dot before digit `i` whenever the number of digits still to come, `len - i`, is a multiple of three (and `i` is not 0). For `51500`, `len` is 5: before index 2, three digits remain, so a dot goes there, giving `51.500`.

`centered` places a title in the middle of a boxed line. It computes how many spaces go on the left and on the right and prints them with the `*` width: `printf("|%*s%s%*s|\n", left, "", text, right, "")` prints the empty string `""` in a field of `left` characters, which is simply `left` spaces.

The weights are kept in tenths of a kilogram, as `int`: 3.5 kg is stored as 35, and printed with `%d.%d` from `tenths / 10` and `tenths % 10`. The amount for a line is `tenths * price / 10`; for 3.5 kg of wash-and-fold at Rp 7,000 per kilogram that is $35 \times 7000/10 = 24500$, computed exactly. The prices are the price list of Chapter 5: Rp 7,000 per kilogram to wash and fold, Rp 10,000 to wash and iron, Rp 6,000 to iron only.

Listing 9.9 `receipt.c`: a receipt laid out with widths, `%*s` centering and thousands separators.

```
/* receipt.c - a Profitina Laundry receipt laid out with printf widths. */
#include <stdio.h>
#include <string.h>

#define WIDTH 44                    /* characters per receipt line */

/* Write n with dots between thousands: 1250000 -> "1.250.000".
 * out must hold at least 32 bytes. */
static void rupiah(long n, char *out)
{
    char digits[24];
    int len = snprintf(digits, sizeof digits, "%ld", n < 0 ? -n : n);
    int k = 0;
    if (n < 0)
        out[k++] = '-';
    for (int i = 0; i < len; i++) {
        if (i > 0 && (len - i) % 3 == 0)
            out[k++] = '.';         /* a dot before each group of three */
        out[k++] = digits[i];
    }
    out[k] = '\0';
}

/* "081234567890" -> "0812-3456-7890". out must hold 24 bytes. */
static void phone_groups(const char *d, char *out)
{
    int k = 0;
    for (int i = 0; d[i] != '\0' && k < 22; i++) {
        if (i > 0 && i % 4 == 0)
            out[k++] = '-';
        out[k++] = d[i];
    }
    out[k] = '\0';
}

static void rule(char edge, char fill)
{
    putchar(edge);
    for (int i = 0; i < WIDTH - 2; i++)
        putchar(fill);
    printf("%c\n", edge);
}
```

Listing 9.9 (Continued) *receipt.c: a receipt laid out with widths, %*s centering and thousands separators.*

```c
/* Print text in the middle of a boxed line: %*s pads with a computed width. */
static void centered(const char *text)
{
    int len = (int)strlen(text);
    int left = (WIDTH - 2 - len) / 2;
    int right = WIDTH - 2 - len - left;
    printf("|%*s%s%*s|\n", left, "", text, right, "");
}

static void money_line(const char *label, long amount)
{
    char m[32];
    rupiah(amount, m);
    printf("%-30s Rp %10s\n", label, m);
}

int main(void)
{
    const char *service[] = {"Wash & fold", "Iron only", "Wash & iron"};
    const int tenths[] = {35, 20, 15};          /* 3.5 kg, 2.0 kg, 1.5 kg */
    const int price[] = {7000, 6000, 10000};    /* rupiah per kg */
    const int items = 3;
    char phone[24], a[32], b[32];
    long subtotal = 0;

    rule('+', '-');
    centered("PROFITINA LAUNDRY");
    centered("Jl. Kenanga 12, Surabaya");
    rule('+', '-');
    printf("Order    : PL-%06d\n", 42);
    printf("Customer : %s\n", "Andi Pratama");
    phone_groups("081234567890", phone);
    printf("Phone    : %s\n\n", phone);

    printf("%-15s %8s %8s %10s\n", "Service", "Weight", "Price", "Amount");
    for (int i = 0; i < items; i++) {
        long amount = (long)tenths[i] * price[i] / 10;
        rupiah(price[i], a);
        rupiah(amount, b);
        printf("%-15s %3d.%d kg %8s %10s\n",
               service[i], tenths[i] / 10, tenths[i] % 10, a, b);
        subtotal += amount;
    }
    rule('-', '-');
    long discount = subtotal * 10 / 100;
    long total = subtotal - discount;
```

Listing 9.9 (Continued) *receipt.c: a receipt laid out with widths, %*s centering and thousands separators.*

```c
    money_line("Subtotal", subtotal);
    money_line("Member discount 10%", -discount);
    money_line("TOTAL", total);
    money_line("Paid", 100000);
    money_line("Change", 100000 - total);
    rule('+', '-');
    centered("Thank you - ready Friday after 16.00");
    rule('+', '-');
    return 0;
}
```

```
+----------------------------------------+
|            PROFITINA LAUNDRY           |
|         Jl. Kenanga 12, Surabaya       |
+----------------------------------------+
Order    : PL-000042
Customer : Andi Pratama
Phone    : 0812-3456-7890

Service          Weight    Price     Amount
Wash & fold      3.5 kg    7.000     24.500
Iron only        2.0 kg    6.000     12.000
Wash & iron      1.5 kg   10.000     15.000
----------------------------------------
Subtotal                      Rp     51.500
Member discount 10%           Rp     -5.150
TOTAL                         Rp     46.350
Paid                          Rp    100.000
Change                        Rp     53.650
+----------------------------------------+
|   Thank you - ready Friday after 16.00   |
+----------------------------------------+
```

Every column lines up because every value is printed in a field of fixed width: service names left-aligned in 15 characters, weights right-aligned, amounts right-aligned in 10. Try changing `"Andi Pratama"` to a much longer name: the customer line grows, but the table does not move, because its widths do not depend on the name. The Python version, **Listing 9.10**, builds the same strings with `%` and with a join for the phone groups.

Listing 9.10 `receipt.py`: the same receipt in Python.

```
# receipt.py - the same Profitina Laundry receipt, laid out with % widths.

WIDTH = 44


def rupiah(n):
    """1250000 -> '1.250.000' (a dot before each group of three digits)."""
    digits = str(abs(n))
    out = "-" if n < 0 else ""
    for i, d in enumerate(digits):
        if i > 0 and (len(digits) - i) % 3 == 0:
            out += "."
        out += d
    return out


def phone_groups(d):
    """'081234567890' -> '0812-3456-7890'."""
    return "-".join(d[i:i + 4] for i in range(0, len(d), 4))


def rule(edge, fill):
    print(edge + fill * (WIDTH - 2) + edge)


def centered(text):
    left = (WIDTH - 2 - len(text)) // 2
    right = WIDTH - 2 - len(text) - left
    print("|%*s%s%*s|" % (left, "", text, right, ""))


def money_line(label, amount):
    print("%-30s Rp %10s" % (label, rupiah(amount)))


def main():
    service = ["Wash & fold", "Iron only", "Wash & iron"]
    tenths = [35, 20, 15]
    price = [7000, 6000, 10000]
    subtotal = 0
```

Listing 9.10 (Continued) *receipt.py: the same receipt in Python.*

```
    rule("+", "-")
    centered("PROFITINA LAUNDRY")
    centered("Jl. Kenanga 12, Surabaya")
    rule("+", "-")
    print("Order    : PL-%06d" % 42)
    print("Customer : %s" % "Andi Pratama")
    print("Phone    : %s\n" % phone_groups("081234567890"))

    print("%-15s %8s %8s %10s" % ("Service", "Weight", "Price", "Amount"))
    for i in range(len(service)):
        amount = tenths[i] * price[i] // 10
        print("%-15s %3d.%d kg %8s %10s" % (service[i], tenths[i] // 10,
              tenths[i] % 10, rupiah(price[i]), rupiah(amount)))
        subtotal += amount
    rule("-", "-")
    discount = subtotal * 10 // 100
    total = subtotal - discount
    money_line("Subtotal", subtotal)
    money_line("Member discount 10%", -discount)
    money_line("TOTAL", total)
    money_line("Paid", 100000)
    money_line("Change", 100000 - total)
    rule("+", "-")
    centered("Thank you - ready Friday after 16.00")
    rule("+", "-")


if __name__ == "__main__":
    main()
```

9.7 Reading Text Safely

Output is the easy half. Input is where text programs break, because the program cannot choose what the person types. Chapter 1 read numbers with `scanf("%d", ...)`, and `scanf` can read words too, with `%s`. But `%s` has two problems. It stops at the first space, so `Andi Pratama` arrives as two separate words. And a plain `%s` has no idea how large the destination array is: a customer who types a 40-letter word into a 32-byte array causes exactly the overflow of Section 9.4. A width fixes the second problem (`%31s` reads at most 31 characters into a 32-byte array), and the *scan set* `%[^\n]`, "every character except newline", fixes the first; `scanf(" %63[^\n]", line)` reads a whole line of at most 63 characters, skipping leading white space. You will meet both in Chapter 11.

For typed commands, a more robust way is to separate *reading* from *understanding*:

1. Read one whole line into an array with `fgets(buf, size, stdin)`. `fgets` never writes more than `size` bytes: it stops after the newline, or when `size - 1` characters have been read, and always adds the terminator. It returns `NULL` at the end of the input.
2. Look at what arrived. If the last character is `'\n'`, the whole line fit; remove the newline. If there is no newline and the array is full, the line was too long: the rest of it is still waiting in the input and must be read and thrown away, or it would be taken for the next command. A file written on Windows ends its lines with `"\r\n"`, so a trailing `'\r'` is removed too.
3. Only then cut the line into words and decide what they mean.

Cashier v0.7 packs step 1 and 2 into a function `read_line` that returns 1 for a good line, 0 at the end of the input and −1 for a line that was too long. With an array of 80 bytes, a line may have up to 79 characters: `fgets` reads 79 characters, finds the array full without a newline, and peeks at

the next character with `getchar()`; if that is the newline, the line fit exactly, otherwise it was too long.

For step 3, the cashier uses a helper `next_word(&p, out, size)` that skips spaces, copies one word into `out` (cutting it to `size - 1` characters, but always *consuming* the whole word), and moves the pointer `p` past the word. The pointer-to-a-pointer parameter `const char **p` is the technique of Chapter 7 for a function that must change the caller's pointer: after `next_word(&rest, cmd, sizeof cmd)`, the caller's `rest` points just after the command word, ready for the next call. Whatever is left after the last word is the customer's name, which may contain spaces and goes through `tidy_name`.

Why not `sscanf` with `%15s` for the words? `sscanf` is fine for fields of known shape, and Chapter 12 uses it for numbers, but `%15s` on a 20-letter word reads 15 letters and leaves the other 5 to be taken as the *next* word, which silently changes the meaning of the line. `next_word` cuts too long words but never splits one into two.

9.8 Building Cashier v0.7

We now have every piece; **Listing 9.11** and **Listing 9.12** put them together. Cashier v0.7 is a small command interpreter, a *read–evaluate–print loop*: it prints a prompt `>`, reads a line, carries out the command, prints the result and starts again. Its commands are:

- `add <service> <kg> <name>` records an order; the service is `fold`, `wash-iron` or `iron`, the weight is written like `3` or `3.5`, and the name may have several words in any case and spacing.
- `find <part of a name>` lists every order whose customer name contains the given text, ignoring upper and lower case.
- `list` prints the orders of the day in a table with a total.
- `help` shows the commands, and `quit` ends the day.

The orders are kept as in Chapter 6, in arrays: a two-dimensional array `names[MAX_ORDERS][NAME_LEN]`, which is 50 rows of 32 characters each, one row per customer name, and two `int` arrays for the service and the weight in tenths. `names[i]` is row `i`, an ordinary `char` array, so every string function works on it. Keeping three arrays side by side, called *parallel arrays*, works but is fragile; Chapter 10 replaces them with one array of structures.

Three more helpers complete the program. `parse_tenths` turns the text `3.5` into the number 35 without any floating point: it accepts one to three digits, optionally followed by a point and exactly one digit, and nothing else; `3.25`, `.5`, `3.` and `3kg` are refused. `contains_ci` searches for one string inside another while ignoring case, by comparing `tolower` of both characters at every position; the standard library has no such function, and writing it takes eleven lines. `rupiah` is the helper of Section 9.6.

Finally, the command-line option `--echo`. When you type commands yourself, the terminal shows what you type. When the commands come from a file (`./cashier_v7 < session.txt`), nobody types, and the output would show only answers. With `--echo`, the program repeats every command after the prompt, so that a recorded session reads like a conversation. The book's outputs are made that way.

Listing 9.11 `cashier_v7.c`: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name.

```c
/* cashier_v7.c - Profitina Laundry cashier v0.7: orders are typed as text
 * commands, names are tidied, and a customer is found by part of a name.
 * Usage: cashier_v7 [--echo]    (--echo repeats each command after "> ") */
#include <ctype.h>
#include <stdio.h>
#include <string.h>

#define MAX_ORDERS 50
#define NAME_LEN 32
#define LINE_LEN 80
#define N_SERVICES 3

static const char *SERVICE[N_SERVICES] = {"fold", "wash-iron", "iron"};
static const int PRICE[N_SERVICES] = {7000, 10000, 6000};

static char names[MAX_ORDERS][NAME_LEN];   /* one row of chars per customer */
static int service_of[MAX_ORDERS];
static int tenths_of[MAX_ORDERS];          /* weight in tenths of a kg */
static int count = 0;

/* 1 = a line was read, 0 = end of input, -1 = line too long (skipped). */
static int read_line(char *buf, int size)
{
    if (fgets(buf, size, stdin) == NULL)
        return 0;
    size_t n = strlen(buf);
    if (n > 0 && buf[n - 1] == '\n') {
        buf[--n] = '\0';
    } else if (n == (size_t)size - 1) {    /* buffer full: is there more? */
        int c = getchar();
        if (c != EOF && c != '\n') {
            while (c != EOF && c != '\n')  /* throw the rest of the line away */
                c = getchar();
            return -1;
        }
    }
    if (n > 0 && buf[n - 1] == '\r')       /* a file saved on Windows */
        buf[--n] = '\0';
    return 1;
}
```

Listing 9.11 (Continued) *cashier_v7.c: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name.*

```c
/* Copy the next word of *p into out (cut to size - 1 chars) and move *p
 * past it. Returns the word's full length; 0 means there was no word. */
static size_t next_word(const char **p, char *out, size_t size)
{
    const char *s = *p;
    size_t n = 0;
    while (isspace((unsigned char)*s))
        s++;
    while (*s != '\0' && !isspace((unsigned char)*s)) {
        if (n + 1 < size)
            out[n] = *s;
        n++;
        s++;
    }
    out[n < size ? n : size - 1] = '\0';
    *p = s;
    return n;
}

/* Trim, squeeze inner spaces to one, and write each word as Title case. */
static void tidy_name(const char *src, char *dst, size_t size)
{
    size_t n = 0;
    int new_word = 1;
    for (size_t i = 0; src[i] != '\0' && n + 1 < size; i++) {
        unsigned char c = (unsigned char)src[i];
        if (isspace(c)) {
            new_word = 1;
            continue;
        }
        if (new_word && n > 0 && n + 2 < size)
            dst[n++] = ' ';
        dst[n++] = (char)(new_word ? toupper(c) : tolower(c));
        new_word = 0;
    }
    dst[n] = '\0';
}
```

Listing 9.11 (Continued) *cashier_v7.c: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name.*

```c
/* "3" -> 30, "3.5" -> 35. Returns 0 if s is not written like that. */
static int parse_tenths(const char *s, int *out)
{
    int whole = 0, digits = 0, tenth = 0;
    for (; isdigit((unsigned char)*s); s++) {
        if (++digits > 3)
            return 0;
        whole = whole * 10 + (*s - '0');
    }
    if (digits == 0)
        return 0;
    if (*s == '.') {
        if (!isdigit((unsigned char)s[1]))
            return 0;
        tenth = s[1] - '0';
        s += 2;
    }
    if (*s != '\0')
        return 0;
    *out = whole * 10 + tenth;
    return 1;
}

/* Does text contain part, ignoring upper and lower case? */
static int contains_ci(const char *text, const char *part)
{
    size_t n = strlen(part);
    for (size_t i = 0; text[i] != '\0'; i++) {
        size_t k = 0;
        while (k < n && text[i + k] != '\0' &&
               tolower((unsigned char)text[i + k]) ==
               tolower((unsigned char)part[k]))
            k++;
        if (k == n)
            return 1;
    }
    return 0;
}
```

Listing 9.11 (Continued) *cashier_v7.c: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name.*

```
static void rupiah(long n, char *out)      /* out: 32 bytes */
{
    char digits[24];
    int len = snprintf(digits, sizeof digits, "%ld", n), k = 0;
    for (int i = 0; i < len; i++) {
        if (i > 0 && (len - i) % 3 == 0)
            out[k++] = '.';
        out[k++] = digits[i];
    }
    out[k] = '\0';
}

static long amount_of(int i)
{
    return (long)tenths_of[i] * PRICE[service_of[i]] / 10;
}

static void print_order(int i)
{
    char m[32];
    rupiah(amount_of(i), m);
    printf("  #%d %s, %s, %d.%d kg = Rp %s\n", i + 1, names[i],
           SERVICE[service_of[i]], tenths_of[i] / 10, tenths_of[i] % 10, m);
}

static void cmd_add(const char *rest)
{
    char svc[16], kg[16], name[NAME_LEN];
    int s = -1, t = 0;
    if (next_word(&rest, svc, sizeof svc) == 0 ||
        next_word(&rest, kg, sizeof kg) == 0) {
        puts("  usage: add <service> <kg> <name>");
        return;
    }
    for (int i = 0; i < N_SERVICES; i++)
        if (strcmp(svc, SERVICE[i]) == 0)
            s = i;
    if (s < 0) {
        printf("  unknown service '%s' (fold, wash-iron, iron)\n", svc);
        return;
    }
    if (!parse_tenths(kg, &t) || t < 1 || t > 500) {
        puts("  weight must look like 3 or 3.5 (0.1 to 50.0 kg)");
        return;
    }
    tidy_name(rest, name, sizeof name);
```

Listing 9.11 (Continued) *cashier_v7.c: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name.*

```
    if (name[0] == '\0') {
        puts("  the customer's name is missing");
        return;
    }
    if (count == MAX_ORDERS) {
        puts("  the order book is full");
        return;
    }
    strcpy(names[count], name);            /* safe: name fits NAME_LEN */
    service_of[count] = s;
    tenths_of[count] = t;
    print_order(count++);
}

static void cmd_find(const char *rest)
{
    char part[NAME_LEN];
    tidy_name(rest, part, sizeof part);
    if (part[0] == '\0') {
        puts("  usage: find <part of a name>");
        return;
    }
    int hits = 0;
    for (int i = 0; i < count; i++)
        if (contains_ci(names[i], part)) {
            print_order(i);
            hits++;
        }
    if (hits == 0)
        printf("  no customer matches '%s'\n", part);
}

static long total_of_day(void)
{
    long total = 0;
    for (int i = 0; i < count; i++)
        total += amount_of(i);
    return total;
}
```

Listing 9.11 (Continued) *cashier_v7.c: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name.*

```
static void cmd_list(void)
{
    char m[32];
    printf("  %2s  %-20s  %-9s  %7s  %10s\n",
           "No", "Customer", "Service", "Weight", "Amount");
    for (int i = 0; i < count; i++) {
        rupiah(amount_of(i), m);
        printf("  %2d  %-20.20s  %-9s  %2d.%d kg  %10s\n", i + 1, names[i],
               SERVICE[service_of[i]], tenths_of[i] / 10, tenths_of[i] % 10, m);
    }
    rupiah(total_of_day(), m);
    printf("  %d order(s), total Rp %s\n", count, m);
}

static void cmd_help(void)
{
    puts("  add <service> <kg> <name>   service: fold, wash-iron, iron");
    puts("  find <part of a name>       list       help       quit");
}

int main(int argc, char *argv[])
{
    int echo = argc > 1 && strcmp(argv[1], "--echo") == 0;
    char line[LINE_LEN], cmd[16], m[32];

    puts("Profitina Laundry cashier v0.7 - type help for the commands");
    for (;;) {
        printf("> ");
        fflush(stdout);
        int r = read_line(line, sizeof line);
        if (r == 0) {
            printf("\n");
            break;
        }
        if (echo)
            printf(r < 0 ? "%.40s...\n" : "%s\n", line);
        if (r < 0) {
            puts("  line too long (at most 79 characters), ignored");
            continue;
        }
        const char *rest = line;
        if (next_word(&rest, cmd, sizeof cmd) == 0)
            continue;                       /* an empty line */
        if (strcmp(cmd, "add") == 0)
            cmd_add(rest);
        else if (strcmp(cmd, "find") == 0)
```

Listing 9.11 (Continued) *cashier_v7.c: Profitina Laundry cashier v0.7, a command interpreter with tidy names and search by part of a name.*

```
            cmd_find(rest);
        else if (strcmp(cmd, "list") == 0)
            cmd_list();
        else if (strcmp(cmd, "help") == 0)
            cmd_help();
        else if (strcmp(cmd, "quit") == 0)
            break;
        else
            printf("  unknown command '%s' - type help\n", cmd);
    }
    rupiah(total_of_day(), m);
    printf("Bye. %d order(s), Rp %s today.\n", count, m);
    return 0;
}
```

Read `main` first: it is only the loop. Every command is dispatched with `strcmp(cmd, "add") == 0`, never with `cmd == "add"`, which would compare two *addresses* (the array `cmd` and the literal) and be false forever. Then read `cmd_add` from top to bottom: each check that fails prints a clear message and returns, so the order is stored only after *all* checks have passed, and the closing `strcpy` is safe because `tidy_name` has already made sure that `name` fits in `NAME_LEN` bytes.

The Python version keeps each order as a small list `[name, service, tenths]` in one list `orders`, uses `split(None, 1)` to separate the command from the rest of the line and `split(None, 2)` to take the service and the weight off the front of the arguments, and uses `in` on lower-case copies for the search.

Listing 9.12 `cashier_v7.py`: the same cashier in Python.

```
# cashier_v7.py - Profitina Laundry cashier v0.7: orders are typed as text
# commands, names are tidied, and a customer is found by part of a name.
# Usage: python3 cashier_v7.py [--echo]   (--echo repeats each command)
import sys

MAX_ORDERS = 50
NAME_LEN = 32
LINE_LEN = 80
SERVICE = ["fold", "wash-iron", "iron"]
PRICE = [7000, 10000, 6000]

orders = []          # each order is a list [name, service index, tenths]


def read_line():
    """Return (1, line), (0, "") at end of input, or (-1, start) if too long."""
    line = sys.stdin.readline()
    if line == "":
        return 0, ""
    line = line[:-1] if line.endswith("\n") else line
    if len(line) > LINE_LEN - 1:
        return -1, line[:LINE_LEN - 1]
    return 1, line[:-1] if line.endswith("\r") else line


def tidy_name(src, size=NAME_LEN):
    words = [w[0].upper() + w[1:].lower() for w in src.split()]
    return " ".join(words)[:size - 1]


def parse_tenths(s):
    """'3' -> 30, '3.5' -> 35, anything else -> None."""
    whole, dot, tenth = s.partition(".")
    if not (whole.isdigit() and len(whole) <= 3 and whole.isascii()):
        return None
    if dot and not (len(tenth) == 1 and tenth.isdigit()):
        return None
    return int(whole) * 10 + (int(tenth) if dot else 0)
```

Listing 9.12 (Continued) *cashier_v7.py: the same cashier in Python.*

```
def rupiah(n):
    digits = str(n)
    out = ""
    for i, d in enumerate(digits):
        if i > 0 and (len(digits) - i) % 3 == 0:
            out += "."
        out += d
    return out


def amount_of(o):
    return o[2] * PRICE[o[1]] // 10


def print_order(i):
    name, s, t = orders[i]
    print("  #%d %s, %s, %d.%d kg = Rp %s"
          % (i + 1, name, SERVICE[s], t // 10, t % 10,
             rupiah(amount_of(orders[i]))))


def cmd_add(words, rest):
    if len(words) < 2:
        print("  usage: add <service> <kg> <name>")
        return
    svc, kg = words[0][:15], words[1][:15]
    if svc not in SERVICE:
        print("  unknown service '%s' (fold, wash-iron, iron)" % svc)
        return
    t = parse_tenths(kg)
    if t is None or not 1 <= t <= 500:
        print("  weight must look like 3 or 3.5 (0.1 to 50.0 kg)")
        return
    name = tidy_name(rest)
    if name == "":
        print("  the customer's name is missing")
        return
    if len(orders) == MAX_ORDERS:
        print("  the order book is full")
        return
    orders.append([name, SERVICE.index(svc), t])
    print_order(len(orders) - 1)
```

Listing 9.12 (Continued) *cashier_v7.py: the same cashier in Python.*

```
def cmd_find(rest):
    part = tidy_name(rest)
    if part == "":
        print("  usage: find <part of a name>")
        return
    hits = 0
    for i, o in enumerate(orders):
        if part.lower() in o[0].lower():
            print_order(i)
            hits += 1
    if hits == 0:
        print("  no customer matches '%s'" % part)


def total_of_day():
    return sum(amount_of(o) for o in orders)


def cmd_list():
    print("  %2s  %-20s  %-9s  %7s  %10s"
          % ("No", "Customer", "Service", "Weight", "Amount"))
    for i, (name, s, t) in enumerate(orders):
        print("  %2d  %-20.20s  %-9s  %2d.%d kg  %10s"
              % (i + 1, name, SERVICE[s], t // 10, t % 10,
                 rupiah(amount_of(orders[i]))))
    print("  %d order(s), total Rp %s" % (len(orders), rupiah(total_of_day())))


def cmd_help():
    print("  add <service> <kg> <name>   service: fold, wash-iron, iron")
    print("  find <part of a name>       list       help       quit")


def main():
    echo = len(sys.argv) > 1 and sys.argv[1] == "--echo"
    print("Profitina Laundry cashier v0.7 - type help for the commands")
    while True:
        print("> ", end="", flush=True)
        r, line = read_line()
        if r == 0:
            print()
            break
        if echo:
            print(line[:40] + "..." if r < 0 else line)
        if r < 0:
            print("  line too long (at most 79 characters), ignored")
```

Listing 9.12 (Continued) *cashier_v7.py: the same cashier in Python.*

```
            continue
        parts = line.split(None, 1)
        if not parts:
            continue
        cmd, rest = parts[0][:15], parts[1] if len(parts) > 1 else ""
        if cmd == "add":
            words = rest.split(None, 2)
            cmd_add(words, words[2] if len(words) > 2 else "")
        elif cmd == "find":
            cmd_find(rest)
        elif cmd == "list":
            cmd_list()
        elif cmd == "help":
            cmd_help()
        elif cmd == "quit":
            break
        else:
            print("  unknown command '%s' - type help" % cmd)
    print("Bye. %d order(s), Rp %s today." % (len(orders), rupiah(total_of_day())))


if __name__ == "__main__":
    main()
```

9.9 Running and Testing It

The file `data/session9.txt` holds a morning at the counter, including the mistakes a tired cashier makes:

```
help
add fold 3.5 andi   pratama
add dry-clean 2 budi
add wash-iron 2 bUDI santoso
add iron 4.25 citra
add iron 4 citra dewi LESTARI

add fold 1.5
find BUDI
find ari
list
dance
add fold 2 this line is far too long because somebody kept typing a very long name
quit
```

Run it with `./cashier_v7 --echo < ../data/session9.txt` from the `c` folder, or `python3 cashier_v7.py --echo < ../data/session9.txt` from the `python` folder. Both print:

```
Profitina Laundry cashier v0.7 - type help for the commands
> help
  add <service> <kg> <name>   service: fold, wash-iron, iron
  find <part of a name>       list       help       quit
> add fold 3.5 andi   pratama
  #1 Andi Pratama, fold, 3.5 kg = Rp 24.500
> add dry-clean 2 budi
  unknown service 'dry-clean' (fold, wash-iron, iron)
> add wash-iron 2 bUDI santoso
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000
> add iron 4.25 citra
  weight must look like 3 or 3.5 (0.1 to 50.0 kg)
> add iron 4 citra dewi LESTARI
  #3 Citra Dewi Lestari, iron, 4.0 kg = Rp 24.000
>
> add fold 1.5
  the customer's name is missing
> find BUDI
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000
> find ari
  #3 Citra Dewi Lestari, iron, 4.0 kg = Rp 24.000
> list
  No  Customer              Service     Weight      Amount
   1  Andi Pratama          fold        3.5 kg      24.500
   2  Budi Santoso          wash-iron   2.0 kg      20.000
   3  Citra Dewi Lestari    iron        4.0 kg      24.000
  3 order(s), total Rp 68.500
> dance
  unknown command 'dance' - type help
> add fold 2 this line is far too long bec...
  line too long (at most 79 characters), ignored
> quit
Bye. 3 order(s), Rp 68.500 today.
```

Check each answer against the rules. `andi   pratama` became `Andi Pratama`; `bUDI santoso` became `Budi Santoso`. The unknown service `dry-clean`, the weight `4.25` with two decimals and the order without a name were refused with a message, and no order number was used up by them. The empty line did nothing. `find BUDI` found Budi although the case differs, and `find ari` found `Citra Dewi Lestari`, because “ari” is inside “Lestari”: searching by part of a name finds everything that contains the text, which is usually what Mbak Rina wants and occasionally a surprise. The 82-character line was refused as a whole instead of being cut into a strange order, and its first 40 characters were echoed so that the log shows what happened.

A session file is a *test*: its correct output is known and can be compared automatically. Each chapter folder has a runner, `tests/run_tests.py`, that does this for every program of the chapter. It compiles each C program with `-Wall -Wextra -Werror`, so that a single warning fails the test, and with the *sanitizers* of GCC, `-fsanitize=address,undefined`, which add checks to the compiled program: any read or write outside an array, any use of freed memory and any signed overflow stops the program with a report instead of silently doing damage [8]. Then it runs the C and the Python version on the same input and checks that both print exactly the output stored in `tests/expected/`. Leaving out the lines that report each compilation, the runner prints:

```
$ python3 tests/run_tests.py
ok    string_basics
ok    char_classes
ok    string_library
ok    format_dial
ok    receipt
ok    cashier_v7
ok    cashier_v7_eof
ok    word_stats
8 cases, 0 failed
```

The case `cashier_v7_eof` feeds a file whose last line has no newline and no `quit`, the input that catches most home-made line readers. On Windows the sanitizers are not available with MSYS2's GCC; run `python tests/run_tests.py --no-sanitize` there.

9.10 Making It Better

Version 0.7 already fixes the problems of Section 9.1: names are tidy, search works on parts of names, and the tables line up. Three weaknesses remain, and it is worth naming them because the next chapters remove them.

First, the orders live in three parallel arrays. Adding a field, say the phone number, means a fourth array and remembering to fill it in every place an order is created. Chapter 10 groups everything about one order into one `struct`.

Second, the order book has a fixed size of 50 and the day's orders vanish when the program ends or the power fails. Chapter 10 grows the order book one order at a time with `malloc`, and Chapter 11 writes it to a file after every change.

Third, the search is *linear*: every `find` looks at every order. With fifty orders a day this takes microseconds and does not matter. With fifty thousand customers in a year it still takes only milliseconds on a laptop, but a smarter structure would be faster; Chapter 12 builds a tree, and the Data Structures course that follows this book studies search structures in depth.

One improvement fits in this chapter, and Exercise 9.4 asks for it: a `phone` field entered with the order and cleaned with `digits_only`, so that a customer can also be found by the last four digits of the number, which is how many laundries in Indonesia actually find their customers.

9.11 Using It for Real

The techniques of this chapter fit any small business that writes names and numbers by hand. A tailor can tidy customer names and find them by part of a name exactly as the cashier does; a fruit and durian orchard selling through a chat group can clean the phone numbers of its buyers with `digits_only` and print packing slips with the receipt layout; a village clinic's front desk can check that a patient's national identity number has 16 digits before it is stored. The pattern is always the same: read a whole line with a size limit, cut it into fields, check every field and refuse what does not fit with a clear message, then store a *canonical* form so that later searches compare like with like.

When you adapt the code, keep in mind:

- **Decide the limits and state them.** `NAME_LEN` of 32 and lines of 79 characters are this program's choices; a clinic may need longer names. Change the constant in one place, and let the messages say what the limit is.

- **Non-English letters.** Names written in UTF-8 contain bytes above 127. The cashier passes them through safely, because every `ctype` call casts to `unsigned char` and these bytes are neither letters nor spaces in the default "C" locale. But `toupper` does not change them, so "élodie" keeps its small é, and `NAME_LEN` counts bytes, not letters. Handling other scripts properly requires wide characters or a Unicode library, a topic beyond this book; Python's `str.title()` and `str.lower()` handle it out of the box.
- **Personal data.** Names and phone numbers are personal data, protected in Indonesia by Law 27 of 2022 on Personal Data Protection (UU PDP). Store only what the business needs, and do not print a full phone number on a receipt that the customer might throw away; the last four digits are enough to recognize it.

9.12 Common Mistakes, and How to Avoid Them

- **Forgetting the terminator's byte.** `char code[7]; strcpy(code, "PL-0042");` writes 8 bytes into 7. Size every array as the longest text plus one (Equation (9.1)), or better, copy with `snprintf(code, sizeof code, "%s", text)`.
- **Comparing strings with `==`.** `if (cmd == "quit")` compares addresses and is never true. Use `strcmp(cmd, "quit") == 0`. The mirror image is `if (strcmp(a, b))`, which is true when the strings *differ*; write the `== 0` or `!= 0` out in full.
- **Using `sizeof` on a pointer.** Inside a function whose parameter is `char *s`, `sizeof s` is the size of a pointer (8 on a 64-bit machine), not of the caller's array. That is why every function in this chapter that writes into an array receives its size as a separate parameter.
- **Passing a plain `char` to `<ctype.h>`.** `isalpha(c)` with a negative `char` is undefined. Always write `isalpha((unsigned char)c)`.
- **Writing into a string literal.** `char *s = "andi"; s[0] = 'A';` may crash. Use an array, `char s[] = "andi";`, when the text will change, and `const char *` when it will not.
- **Reading words with a bare `%s`.** `scanf("%s", name)` has no size limit. Give a width one less than the array (`%31s`), or read the line with `fgets` and cut it yourself.
- **Forgetting that `fgets` keeps the newline.** Comparing `"quit\n"` with `"quit"` fails. Remove the newline (and a `'\r'`) right after reading, as `read_line` does; the one-liner `buf[strcspn(buf, "\r\n")] = '\0';` does both.
- **Using `strtok` on data with empty fields or on two strings at once.** It skips empty fields and remembers its position in a hidden variable. Chapter 11 writes a field splitter that has neither problem.
- **Printing money as `double`.** `%.2f` of 2.675 prints 2.67. Keep money in integers and format it yourself.

9.13 Chapter Summary

- A character is a small integer; in ASCII, `'A'` is 65, `'a'` is 97 and `'0'` is 48, so a digit's value is `c - '0'`.
- A C string is a `char` array that ends with the null character `'\0'`. `sizeof` gives the bytes of the array, `strlen` the characters before the terminator (Equation (9.1)).
- An array initialized from a literal is a changeable copy; a `const char *` pointing at a literal must not be written through.
- `<ctype.h>` classifies and converts single characters; cast the argument to `unsigned char`.

- `<string.h>` copies, joins, compares and searches strings without checking anything. Give every copy a size (`snprintf`), test `strchr` and `strstr` for `NULL`, compare with `strcmp(...) == 0`, and remember that `strtok` changes its input.
- A `printf` conversion specification is `%[flags][width][.precision]type` (Equation (9.2)); fixed widths make columns line up, `*` takes a width from the arguments, and binary floating point is not for money.
- Safe input reads a whole line with `fgets`, handles lines that are too long, removes the line ending, and only then cuts the line into words.
- Cashier v0.7 is a read–evaluate–print loop with tidy names, validated input and search by part of a name, in C and in Python with identical output, checked by a test runner with sanitizers.

9.14 Exercises

1. Without running it, write down what `printf("[%-6.3s][%06.1f][%+5d]\n", "Profitina", 3.14159, 42);` prints. Then run it to check.
2. Write `int count_char(const char *s, char c)` that returns how many times `c` occurs in `s`, and use it to count the spaces in `"Citra Dewi Lestari"`. Why is the parameter `const`?
3. Extend `digits_only` and `valid_mobile` of **Listing 9.3** so that a number written in the international form, `+62 812 3456 7890` or `62812...`, is converted to the domestic form `0812...` before it is checked. Give the output for the three numbers of the program.
4. Add a `phone` column to cashier v0.7: `add` takes the phone number as a fourth word before the name, stores it cleaned by `digits_only`, and `find` also matches when the search text equals the last four digits of a phone number. List the lines of the program you had to change, and say which of them Chapter 10 would make unnecessary.
5. Explain, byte by byte, what happens in memory when `char buf[8]; strcpy(buf, "Profitina");` runs. How many bytes are written outside `buf`? Rewrite the line so that it is safe, and say what `buf` then contains.
6. Write a program `word_stats` in C and in Python that reads a text from standard input and prints the number of lines, words and letters, the longest word, and the most common letter (ignoring case; on a tie, the letter nearer to `a`). A word is a run of letters. Test it on `data/notes.txt`.

9.15 Solutions to the Exercises

1. `%-6.3s` prints at most 3 characters, `Pro`, left-aligned in 6 columns: `[Pro   ]`. `%06.1f` rounds 3.14159 to `3.1` and pads it with zeros to 6 characters: `[0003.1]`. `%+5d` prints `+42` right-aligned in 5: `[  +42]`. The line is `[Pro   ][0003.1][  +42]`.

2. The loop walks to the terminator and counts matches:

```
int count_char(const char *s, char c)
{
    int n = 0;
    for (; *s != '\0'; s++)
        if (*s == c)
            n++;
    return n;
}
```

For `"Citra Dewi Lestari"` it returns 2. The parameter is `const char *` because the function only reads the string; the `const` promises that to the caller, lets the function accept string literals, and makes the compiler reject an accidental write such as `*s = c`.

3. After collecting the digits into `d`, add: if `d` starts with `62`, shift everything two places to the left, put `'0'` in front and keep the terminator, which in C is `memmove(d + 1, d + 2, strlen(d + 2) + 1); d[0] = '0';` and in Python `d = "0" + d[2:]`. The first number stays `081234567890 (12 digits) ok`; the second becomes `081234567890 (12 digits) ok`; the third stays `081234 (6 digits) check with customer`. Note that `memmove`, not `strcpy`, must be used when source and destination overlap.

4. The changes are: a new array `char phones[MAX_ORDERS][16]`; in `cmd_add`, a third `next_word` for the phone, then `digits_only`, then a length check; the copy into `phones[count]`; in `cmd_find`, a second test, `strlen(part) == 4 && strcmp(phones[i] + strlen(phones[i]) - 4, part) == 0` (after checking that the stored number has at least 4 digits); the help text; and a wider `list`. The new array and the lines that keep it in step with the other three are what Chapter 10 makes unnecessary: with a `struct Order` that has a `phone` member, one assignment of a whole order keeps all fields together.

5. `"Profitina"` has 9 characters, so `strcpy` writes 10 bytes, `P r o f i t i n a \0`, starting at `buf[0]`. `buf` has 8 bytes, `buf[0]` to `buf[7]`, which receive `Profitin`; the `a` and the terminator are written into the two bytes after the array, which belong to something else. The program may seem to work, crash later, or corrupt another variable; under the sanitizer it stops with a *stack-buffer-overflow* report. The safe line is `snprintf(buf, sizeof buf, "%s", "Profitina");`. It writes at most 8 bytes: the first `sizeof buf - 1` = 7 characters, `Profiti`, and the terminator. It returns 9, the length of the full text, so the program can tell that the name was cut.

6. **Listing 9.13** reads line by line with `fgets` and walks each line with the same "am I inside a word?" state as `tidy_name`: a letter extends the current word, anything else ends it. Walking up to *and including* the terminator makes sure that a word at the very end of a line is counted too.

Listing 9.13 `word_stats.c`: counting lines, words and letters, the longest word and the most common letter.

```c
/* word_stats.c - Exercise 9.6: read text from standard input and report the
 * number of lines, words and letters, the longest word and the most common
 * letter. A word is a run of letters; case is ignored when counting letters. */
#include <ctype.h>
#include <stdio.h>
#include <string.h>

#define LINE_LEN 256
#define WORD_LEN 32

int main(void)
{
    char line[LINE_LEN], word[WORD_LEN], longest[WORD_LEN] = "";
    int lines = 0, words = 0, letters = 0, freq[26] = {0};

    while (fgets(line, sizeof line, stdin) != NULL) {
        lines++;
        size_t len = 0;                          /* length of current word */
        for (size_t i = 0;; i++) {
            unsigned char c = (unsigned char)line[i];
            if (isalpha(c)) {
                if (len + 1 < sizeof word)
                    word[len] = (char)c;
                len++;
                letters++;
                freq[tolower(c) - 'a']++;
            } else {
                if (len > 0) {                   /* a word has just ended */
                    words++;
                    word[len < sizeof word ? len : sizeof word - 1] = '\0';
                    if (strlen(word) > strlen(longest))
                        strcpy(longest, word);
                    len = 0;
                }
                if (c == '\0')
                    break;
            }
        }
    }
    int best = 0;
    for (int k = 1; k < 26; k++)
        if (freq[k] > freq[best])
            best = k;
    printf("lines: %d, words: %d, letters: %d\n", lines, words, letters);
    printf("longest word: \"%s\" (%zu letters)\n", longest, strlen(longest));
    printf("most common letter: %c (%d times)\n", 'a' + best, freq[best]);
```

Listing 9.13 (Continued) *word_stats.c: counting lines, words and letters, the longest word and the most common letter.*

```c
    return 0;
}
```

The Python version, **Listing 9.14**, appends a `"\0"` to each line for the same reason, and chooses the most common letter with `max` and a key that prefers the larger count and, on a tie, the smaller index.

Listing 9.14 `word_stats.py`: the same statistics in Python.

```python
# word_stats.py - Exercise 9.6: lines, words, letters, the longest word and
# the most common letter of the text on standard input.
import sys

WORD_LEN = 32


def main():
    lines = words = letters = 0
    longest = ""
    freq = [0] * 26
    for line in sys.stdin:
        lines += 1
        word = ""
        for c in line + "\0":                  # "\0" ends the last word
            if c.isascii() and c.isalpha():
                word += c
                letters += 1
                freq[ord(c.lower()) - ord("a")] += 1
            elif word:
                words += 1
                word = word[:WORD_LEN - 1]
                if len(word) > len(longest):
                    longest = word
                word = ""
    best = max(range(26), key=lambda k: (freq[k], -k))
    print("lines: %d, words: %d, letters: %d" % (lines, words, letters))
    print('longest word: "%s" (%d letters)' % (longest, len(longest)))
    print("most common letter: %c (%d times)" % (chr(ord("a") + best), freq[best]))


if __name__ == "__main__":
    main()
```

With `./word_stats < ../data/notes.txt` both print:

```
lines: 3, words: 26, letters: 121
longest word: "separately" (10 letters)
most common letter: e (15 times)
```

References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988.

[2] ISO/IEC 9899:2018, *Information technology — Programming languages — C* (C17). Geneva, Switzerland: ISO, 2018, §7.4 “Character handling <ctype.h>”, §7.21.6 “Formatted input/output functions”, §7.24 “String handling <string.h>”.

[3] cppreference.com, “Null-terminated byte strings.” [Online]. Available: https://en.cppreference.com/w/c/string/byte (accessed Sep. 2026).

[4] Software Engineering Institute, Carnegie Mellon University, *SEI CERT C Coding Standard*, rules STR31-C “Guarantee that storage for strings has sufficient space for character data and the null terminator” and STR37-C “Arguments to character-handling functions must be representable as an unsigned char.” [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c (accessed Sep. 2026).

[5] MITRE, “2025 CWE Top 25 Most Dangerous Software Weaknesses,” Dec. 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

[6] P. J. Deitel and H. M. Deitel, *C How to Program*, 9th ed. Hoboken, NJ, USA: Pearson, 2022, ch. 8 “Characters and Strings” and ch. 9 “Formatted Input/Output.”

[7] Python Software Foundation, “Floating-Point Arithmetic: Issues and Limitations,” *The Python Tutorial*, Python 3.14. [Online]. Available: https://docs.python.org/3/tutorial/floatingpoint.html (accessed Sep. 2026).

[8] GCC team, “Program Instrumentation Options (-fsanitize=address, -fsanitize=undefined),” *Using the GNU Compiler Collection*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (accessed Sep. 2026).

Chapter 10

Structures, Enums and a First Linked List

An order at Profitina Laundry is one thing: a number, a customer, a service, a weight, an amount and a state such as “washing” or “ready”. Until now our programs have kept these facts in separate arrays and relied on every line of code to keep them in step. This chapter gives the program a way to say what the order *is*. By the end of the chapter you will be able to define a `struct` that groups related values of different types, create, copy and compare structures, and explain why a structure can be larger than the sum of its members; pass structures to functions by value and by pointer and use the `->` operator; give names to codes with `enum` and to types with `typedef`, and build tables indexed by those names; ask for memory while the program runs with `malloc` and give it back with `free`; and build, walk, grow, shrink and free a *singly linked list*, the first data structure in this book whose size is not fixed in advance. The chapter ends with cashier v0.8, in which every order is one structure, the orders in the shop form a linked list, and each order moves from *waiting* to *washing* to *ready* until the customer picks it up.

10.1 Where We Left Off: Three Arrays That Must Agree

Cashier v0.7 of Chapter 9 remembers each order in three *parallel arrays*: the name in `names[i]`, the service in `service_of[i]` and the weight in `tenths_of[i]`. The index `i` is the only thing that ties the three together. As long as every piece of code that touches an order touches all three arrays with the same index, all is well.

Now imagine that Mbak Rina asks for the day’s orders sorted by amount, largest first, so that she can see which bags to wash first. A programmer who writes a sorting loop in a hurry swaps `names[i]` and `names[j]` and forgets to swap `tenths_of[i]` and `tenths_of[j]`. The program compiles, runs and prints a neat table in which Andi’s name stands next to Budi’s weight and bill. Nothing crashes; the table is simply wrong, and a customer is charged for someone else’s laundry. Every new fact about an order, such as a phone number or a pickup date, adds another array that every sorting, deleting and copying loop must remember.

There is a second problem. The arrays have a fixed size, 50 orders, chosen when the program was written. On a quiet Tuesday half of that space is wasted; on the Monday after a long weekend, the 51st customer is turned away by the message “the order book is full”. And there is a third: when a customer picks up an order, v0.7 has no way to take it out of the middle of the arrays except by shifting every later order one place to the left, in all three arrays.

C’s answer to the first problem is the **structure**, a type that bundles several values under one name, so that an order is moved, copied and passed around as one thing. Its answer to the second and third problems is to ask for memory one order at a time while the program runs, and to chain the orders together with pointers into a **linked list**, which grows and shrinks one element at a time and lets an order leave from the middle by changing a single pointer. This chapter builds both, step by step.

10.2 The Idea in Plain Words: One Record, One Name

10.2.1 Declaring a structure

A structure declaration describes a new type: which values it contains, called its *members* (or *fields*), and the type of each.

```
struct Order {
    int id;
    char customer[22];
    int tenths;
    long long amount;
};
```

This declares a type named `struct Order`; it does not yet create any variable. A variable of the type is declared like any other, `struct Order a;`, and each member is reached with the **dot operator**: `a.id`, `a.customer`, `a.amount`. A member can be of any type, including an array such as `customer` or, as we will see, another structure or a pointer.

> **DEFINITION.** A *structure* (`struct`) is a type made of named *members* that may have different types. A variable of a structure type holds one value of every member, stored together in one block of memory. The member `m` of a structure variable `s` is written `s.m`; if `p` is a pointer to a structure, the member is `(*p).m`, which C allows to be written `p->m`.

A structure variable can be initialized with a list of values in braces, given in the order of the members, or with *designated initializers* that name the members, in any order. Members that are not given a value are set to zero, exactly as with arrays:

```
struct Order b = {2, "Budi Santoso", 20, 20000};    /* in order */
struct Order c = {.customer = "Citra", .id = 3};    /* by name  */
```

10.2.2 What a structure looks like in memory

The members of a structure are stored one after another, in the order of the declaration. But they are not always packed tightly. Most processors read an `int` fastest when its address is a multiple of 4, and a `long long` when its address is a multiple of 8; this requirement is the *alignment* of the type. To honor it, the compiler may insert unused bytes, called *padding*, between members and at the end.

Figure 10.1 shows `struct Order` on a typical 64-bit computer. `id` takes bytes 0 to 3. `customer` takes the next 22 bytes, 4 to 25. The next member, `tenths`, is an `int` and must start at a multiple of 4; the next free byte, 26, is not, so the compiler skips two bytes and starts `tenths` at 28. `amount` then starts at 32, which is a multiple of 8, and ends at 39. The whole structure takes 40 bytes although its members add up to only 38.

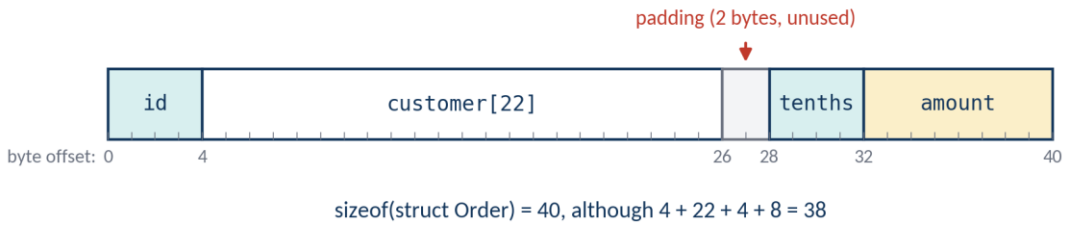


Figure 10.1 The layout of `struct Order` in memory: four members at offsets 0, 4, 28 and 32, and two bytes of padding after `customer` so that `tenths` starts at a multiple of four.

The rule the compiler follows is simple to state. If the next free byte is at offset o and the next member has alignment a, the member starts at the first multiple of a that is not smaller than o:

$$o' = a \cdot \left\lceil \frac{o}{a} \right\rceil \tag{10.1}$$

where $\lceil x \rceil$ means "x rounded up to a whole number". For `tenths`, $o = 26$ and $a = 4$, so $o' = 4 \cdot \lceil 6.5 \rceil = 4 \cdot 7 = 28$. For `amount`, $o = 32$ and $a = 8$, so $o' = 8 \cdot 4 = 32$ and no padding is needed. At the end, the size of the whole structure is rounded up the same way to a multiple of its largest alignment, so that in an array of structures every element is aligned too. The exact alignments are chosen by the platform, not by the C standard [1], so the practical rule is: never compute the size of a structure by hand in a program; ask `sizeof`, and ask `offsetof` (from `<stddef.h>`) where a member starts.

10.2.3 The first program

Listing 10.1 puts the pieces together: filling a structure member by member, initializing two more with lists, copying one, comparing two, and measuring the layout of **Figure 10.1**.

Listing 10.1 `order_struct.c`: declaring, initializing, copying and comparing structures, and measuring their layout.

```c
/* order_struct.c - one struct holds everything about one laundry order. */
#include <stddef.h>
#include <stdio.h>
#include <string.h>

struct Order {
    int id;                 /* 4 bytes                                   */
    char customer[22];      /* 22 bytes                                  */
    int tenths;             /* weight in tenths of a kg; 4 bytes         */
    long long amount;       /* rupiah; 8 bytes                           */
};

int main(void)
{
    struct Order a;                              /* members still unknown */
    a.id = 1;
    strcpy(a.customer, "Andi Pratama");
    a.tenths = 35;
    a.amount = 24500;

    struct Order b = {2, "Budi Santoso", 20, 20000};     /* in order */
    struct Order c = {.customer = "Citra", .id = 3};     /* by name */

    printf("a: #%d %-14s %d.%d kg Rp %lld\n",
           a.id, a.customer, a.tenths / 10, a.tenths % 10, a.amount);
    printf("b: #%d %-14s %d.%d kg Rp %lld\n",
           b.id, b.customer, b.tenths / 10, b.tenths % 10, b.amount);
    printf("c: #%d %-14s %d.%d kg Rp %lld (members not named are 0)\n",
           c.id, c.customer, c.tenths / 10, c.tenths % 10, c.amount);

    struct Order copy = a;          /* copies every member, array included */
    copy.id = 99;
    copy.customer[0] = 'X';
    printf("copy: #%d %s | a is still #%d %s\n",
           copy.id, copy.customer, a.id, a.customer);

    int same = a.id == b.id && strcmp(a.customer, b.customer) == 0 &&
               a.tenths == b.tenths && a.amount == b.amount;
    printf("a and b hold the same order? %s\n", same ? "yes" : "no");
```

Listing 10.1 (Continued) *order_struct.c: declaring, initializing, copying and comparing structures, and measuring their layout.*

```c
    printf("sizeof(struct Order) = %zu, members add up to %zu\n",
           sizeof(struct Order),
           sizeof a.id + sizeof a.customer + sizeof a.tenths + sizeof a.amount);
    printf("offsets: id %zu, customer %zu, tenths %zu, amount %zu\n",
           offsetof(struct Order, id), offsetof(struct Order, customer),
           offsetof(struct Order, tenths), offsetof(struct Order, amount));
    return 0;
}
```

```
a: #1 Andi Pratama   3.5 kg Rp 24500
b: #2 Budi Santoso   2.0 kg Rp 20000
c: #3 Citra          0.0 kg Rp 0 (members not named are 0)
copy: #99 Xndi Pratama | a is still #1 Andi Pratama
a and b hold the same order? no
sizeof(struct Order) = 40, members add up to 38
offsets: id 0, customer 4, tenths 28, amount 32
```

Three facts in this output deserve attention.

Assignment copies everything. `struct Order copy = a;` copies all 40 bytes, *including the array* `customer`. Changing `copy.customer[0]` to `X` leaves `a.customer` untouched: the copy has its own

array. This is remarkable because Chapter 6 showed that arrays cannot be assigned with `=` at all; wrapped in a structure, they can.

Comparison is your job. C has no `==` for structures. Two orders are "the same" if you say so, and the program says so member by member, with `strcmp` for the name. (Comparing the raw bytes with `memcmp` is tempting and wrong: the padding bytes may contain anything.)

The size is 40, not 38. The last two lines confirm **Figure 10.1** and Equation (10.1). Reordering the members from largest to smallest alignment, `amount` first, then the two `int`s, then the array, would bring the size to 38 plus 2 bytes of padding at the end, still 40, because the size must be a multiple of 8. Padding is rarely worth fighting; knowing it exists explains many surprises, for example in Chapter 11 when structures are written to files.

The Python version, **Listing 10.2**, uses a *data class*, a class declared with `@dataclass`, which is the closest Python equivalent of a C structure: named fields, a generated initializer that accepts values in order or by name, and a generated `==` that compares field by field [2]. Python objects are not laid out byte by byte, so to print the same sizes the program describes the C structure with the `ctypes` module, which applies the platform's C alignment rules [3].

Listing 10.2 `order_struct.py`: a data class for the order, and `ctypes` to measure the C layout.

```
# order_struct.py - a class with named fields plays the role of a C struct.
# ctypes.Structure lays the fields out exactly like the C compiler does, so
# the sizes and offsets printed here match order_struct.c.
import ctypes
from dataclasses import dataclass, replace


@dataclass
class Order:
    id: int = 0
    customer: str = ""
    tenths: int = 0            # weight in tenths of a kg
    amount: int = 0            # rupiah


class COrder(ctypes.Structure):
    _fields_ = [("id", ctypes.c_int), ("customer", ctypes.c_char * 22),
                ("tenths", ctypes.c_int), ("amount", ctypes.c_longlong)]


def show(label, o, note=""):
    print("%s: #%d %-14s %d.%d kg Rp %d%s"
          % (label, o.id, o.customer, o.tenths // 10, o.tenths % 10,
             o.amount, note))


def main():
    a = Order()
    a.id = 1
    a.customer = "Andi Pratama"
    a.tenths = 35
    a.amount = 24500
    b = Order(2, "Budi Santoso", 20, 20000)
    c = Order(customer="Citra", id=3)
    show("a", a)
    show("b", b)
    show("c", c, " (members not named are 0)")

    copy = replace(a)                    # a real copy, like struct assignment
    copy.id = 99
    copy.customer = "X" + copy.customer[1:]
    print("copy: #%d %s | a is still #%d %s"
          % (copy.id, copy.customer, a.id, a.customer))
    print("a and b hold the same order? %s" % ("yes" if a == b else "no"))
```

Listing 10.2 (Continued) *order_struct.py: a data class for the order, and ctypes to measure the C layout.*

```
    members = sum(ctypes.sizeof(t) for _, t in COrder._fields_)
    print("sizeof(struct Order) = %d, members add up to %d"
          % (ctypes.sizeof(COrder), members))
    print("offsets: id %d, customer %d, tenths %d, amount %d"
          % tuple(getattr(COrder, f).offset for f, _ in COrder._fields_))


if __name__ == "__main__":
    main()
```

There is one real difference to keep in mind. In Python, `b = a` does *not* copy an object; it makes `b` a second name for the same object, and changing `b.id` changes `a.id`. To copy, the program calls `dataclasses.replace(a)`. C's assignment copies; Python's assignment shares.

10.3 Structures and Functions

A structure can be passed to a function and returned from it like any other value, and here the difference between *by value* and *by pointer*, which Chapter 7 introduced for single numbers, becomes very visible.

Passed **by value**, the function receives a copy of the whole structure. It can read it and change its copy, and the caller's structure is never affected. For a 40-byte order the copying costs little; for a structure holding a 1,000-character note it costs 1,000 bytes on every call.

Passed **by pointer**, the function receives only the address of the caller's structure, 8 bytes on a 64-bit computer, whatever the size of the structure. Through the pointer it can read the members and change them. Because pointers to structures are so common, C has an operator for "the member of the structure this pointer points to": `o->amount` means exactly `(*o).amount`. The parentheses in the long form are necessary because `.` binds more tightly than `*`; the arrow saves them.

When a function only needs to *read* an order, the parameter is declared `const struct Order *o`: the pointer avoids the copy, and `const` promises that the function will not change the order and makes the compiler enforce the promise. This is the normal way to pass a structure in C.

Listing 10.3 shows all of this with an *array of structures*, `struct Order day[4]`, which replaces the parallel arrays of Chapter 9: `day[i]` is the whole order `i`, and `day[i].amount` its amount.

Listing 10.3 `order_functions.c`: structures passed by value and by pointer, a function that returns a pointer into an array, and `qsort` with a comparison function.

```c
/* order_functions.c - structs passed to functions: by value (a copy) and by
 * pointer (the original), the -> operator, and an array of structs. */
#include <stdio.h>
#include <stdlib.h>

struct Order {
    int id;
    char customer[24];
    int tenths;              /* weight in tenths of a kg */
    long amount;             /* rupiah */
};

/* const pointer: we may read the order, never change it */
static void print_order(const struct Order *o)
{
    printf("  #%d %-14s %2d.%d kg  Rp %6ld\n",
           o->id, o->customer, o->tenths / 10, o->tenths % 10, o->amount);
}

static void discount_copy(struct Order o, int percent)    /* gets a copy */
{
    o.amount -= o.amount * percent / 100;
}

static void discount(struct Order *o, int percent)        /* gets the address */
{
    o->amount -= o->amount * percent / 100;     /* same as (*o).amount */
}

static long total(const struct Order orders[], int n)
{
    long sum = 0;
    for (int i = 0; i < n; i++)
        sum += orders[i].amount;
    return sum;
}
```

Listing 10.3 (Continued) *order_functions.c: structures passed by value and by pointer, a function that returns a pointer into an array, and qsort with a comparison function.*

```c
/* Returns a pointer INTO the array, so the caller can change that order. */
static struct Order *largest(struct Order orders[], int n)
{
    struct Order *best = &orders[0];
    for (int i = 1; i < n; i++)
        if (orders[i].amount > best->amount)
            best = &orders[i];
    return best;
}

/* For qsort: larger amount first; equal amounts by smaller id. */
static int by_amount_desc(const void *x, const void *y)
{
    const struct Order *a = x, *b = y;
    if (a->amount != b->amount)
        return a->amount < b->amount ? 1 : -1;
    return a->id - b->id;
}

int main(void)
{
    struct Order day[] = {
        {1, "Andi Pratama", 35, 24500},
        {2, "Budi Santoso", 20, 20000},
        {3, "Citra Lestari", 40, 24000},
        {4, "Dewi Anggraini", 55, 38500},
    };
    int n = (int)(sizeof day / sizeof day[0]);

    discount_copy(day[0], 10);
    printf("after discount_copy(day[0], 10): Rp %ld (unchanged)\n", day[0].amount);
    discount(&day[0], 10);
    printf("after discount(&day[0], 10):     Rp %ld\n", day[0].amount);

    printf("orders of the day, total Rp %ld:\n", total(day, n));
    for (int i = 0; i < n; i++)
        print_order(&day[i]);

    struct Order *big = largest(day, n);
    printf("largest: #%d %s\n", big->id, big->customer);
    big->amount += 5000;                          /* a delivery fee, say */
    printf("with delivery fee: Rp %ld (day[%d] changed)\n",
           day[3].amount, (int)(big - day));
```

Listing 10.3 (Continued) *order_functions.c: structures passed by value and by pointer, a function that returns a pointer into an array, and qsort with a comparison function.*

```c
    qsort(day, (size_t)n, sizeof day[0], by_amount_desc);
    printf("sorted by amount, largest first:\n");
    for (int i = 0; i < n; i++)
        print_order(&day[i]);
    return 0;
}
```

```
after discount_copy(day[0], 10): Rp 24500 (unchanged)
after discount(&day[0], 10):     Rp 22050
orders of the day, total Rp 104550:
  #1 Andi Pratama    3.5 kg  Rp  22050
  #2 Budi Santoso    2.0 kg  Rp  20000
  #3 Citra Lestari   4.0 kg  Rp  24000
  #4 Dewi Anggraini  5.5 kg  Rp  38500
largest: #4 Dewi Anggraini
with delivery fee: Rp 43500 (day[3] changed)
sorted by amount, largest first:
  #4 Dewi Anggraini  5.5 kg  Rp  43500
  #3 Citra Lestari   4.0 kg  Rp  24000
  #1 Andi Pratama    3.5 kg  Rp  22050
  #2 Budi Santoso    2.0 kg  Rp  20000
```

`discount_copy` received a copy and its 10% discount disappeared with the copy; `discount` received `&day[0]` and changed the real order. `largest` returns a pointer *into* the caller's array, so the caller can use it to change the order it found: adding a delivery fee through `big` changed `day[3]`, and `big - day` gives its index by pointer arithmetic.

The last part sorts the orders with the library function `qsort` from `<stdlib.h>` [4]. `qsort` can sort an array of anything, because it is told the number of elements, the size of one element and a *comparison function* (a pointer to a function, Chapter 7). The comparison function receives two `const void *` pointers, which it converts to pointers to orders, and returns a negative number if the first should come first, a positive number if the second should, and zero if they are equal. Here it puts larger amounts first and, for equal amounts, the smaller id first. The second rule matters: `qsort` does not promise to keep equal elements in their original order, so without a tie-breaker two orders with the same amount could come out in either order on different computers. And look at the output: the whole order moved as one, so no name can ever again end up next to someone else's bill. That was the problem of Section 10.1, solved by construction.

In Python, **Listing 10.4**, every argument is passed as a reference to the object, so a function can always change the caller's order; `discount_copy` makes its own copy to behave like C's by-value parameter. Sorting uses `list.sort` with a *key* function; the key `(-o.amount, o.id)` sorts by amount descending and id ascending, the same order as the C comparison.

Listing 10.4 `order_functions.py`: the same functions in Python, where every argument is a reference.

```python
# order_functions.py - Python passes a reference to the object, so a function
# can change an order; to protect the caller's order we pass a copy ourselves.
from dataclasses import dataclass, replace


@dataclass
class Order:
    id: int
    customer: str
    tenths: int
    amount: int


def print_order(o):
    print("  #%d %-14s %2d.%d kg  Rp %6d"
          % (o.id, o.customer, o.tenths // 10, o.tenths % 10, o.amount))


def discount_copy(o, percent):
    o = replace(o)                      # work on a copy, like C by-value
    o.amount -= o.amount * percent // 100


def discount(o, percent):
    o.amount -= o.amount * percent // 100


def total(orders):
    return sum(o.amount for o in orders)


def largest(orders):
    best = orders[0]
    for o in orders[1:]:
        if o.amount > best.amount:
            best = o
    return best


def main():
    day = [Order(1, "Andi Pratama", 35, 24500),
           Order(2, "Budi Santoso", 20, 20000),
           Order(3, "Citra Lestari", 40, 24000),
           Order(4, "Dewi Anggraini", 55, 38500)]
```

Listing 10.4 (Continued) *order_functions.py: the same functions in Python, where every argument is a reference.*

```
    discount_copy(day[0], 10)
    print("after discount_copy(day[0], 10): Rp %d (unchanged)" % day[0].amount)
    discount(day[0], 10)
    print("after discount(&day[0], 10):     Rp %d" % day[0].amount)

    print("orders of the day, total Rp %d:" % total(day))
    for o in day:
        print_order(o)

    big = largest(day)
    print("largest: #%d %s" % (big.id, big.customer))
    big.amount += 5000
    print("with delivery fee: Rp %d (day[%d] changed)"
          % (day[3].amount, day.index(big)))

    day.sort(key=lambda o: (-o.amount, o.id))
    print("sorted by amount, largest first:")
    for o in day:
        print_order(o)


if __name__ == "__main__":
    main()
```

10.4 `enum` and `typedef`: Names Instead of Numbers

In earlier chapters the service was a number: 1 for wash and fold, 2 for wash and iron, 3 for iron only. A line such as `if (service == 2)` works, but a reader must remember what 2 means, and nothing stops a program from storing 7. An **enumeration** gives such codes names:

```
typedef enum { FOLD, WASH_IRON, IRON_ONLY, N_SERVICES } Service;
```

The names in braces become integer constants: `FOLD` is 0, `WASH_IRON` is 1, `IRON_ONLY` is 2, each one more than the previous unless a value is written explicitly (`FOLD = 1`). The last name, `N_SERVICES`, is a common trick: it equals the number of real services, 3, so a table can be declared with `N_SERVICES` elements and a loop can run `for (s = FOLD; s < N_SERVICES; s++)`, and both stay right when a service is added before it.

The keyword **typedef** gives a type a second name. Without it, the type above would be called `enum { ... }` or `enum Service`; with `typedef ... Service;` it is simply `Service`. The same works for structures: `typedef struct { ... } ServiceInfo;` lets us write `ServiceInfo` where we would otherwise write `struct ServiceInfo`. `typedef` creates no new type and no variable, only a shorter name.

Enumerations become especially useful together with **designated initializers for arrays**, which set elements by index: `[WASH_IRON] = {"wash-iron", 10000, 48}` puts that structure in element 1, whatever order the lines are written in. **Listing 10.5** builds the laundry's service table that way, and uses a second enumeration, `Status`, for the life of an order.

Listing 10.5 `service_enum.c`: enumerations for services and statuses, and a service table built with designated initializers.

```c
/* service_enum.c - enum gives names to codes, typedef gives names to types,
 * and a designated initializer builds a table indexed by those names. */
#include <stdio.h>

typedef enum { FOLD, WASH_IRON, IRON_ONLY, N_SERVICES } Service;
typedef enum { WAITING, WASHING, READY, PICKED_UP } Status;

typedef struct {
    const char *name;
    int price_per_kg;        /* rupiah */
    int hours;               /* promised turnaround */
} ServiceInfo;

static const ServiceInfo SERVICES[N_SERVICES] = {
    [FOLD] = {"fold", 7000, 24},
    [WASH_IRON] = {"wash-iron", 10000, 48},
    [IRON_ONLY] = {"iron", 6000, 24},
};

static const char *status_name(Status s)
{
    switch (s) {
    case WAITING:   return "waiting";
    case WASHING:   return "washing";
    case READY:     return "ready";
    case PICKED_UP: return "picked up";
    }
    return "?";              /* only reached if s holds a bad number */
}

static Status next_status(Status s)
{
    return s == PICKED_UP ? PICKED_UP : (Status)(s + 1);
}

int main(void)
{
    printf("%-3s %-10s %6s %6s\n", "#", "service", "Rp/kg", "hours");
    for (Service s = FOLD; s < N_SERVICES; s++)
        printf("%-3d %-10s %6d %6d\n",
               (int)s, SERVICES[s].name, SERVICES[s].price_per_kg,
               SERVICES[s].hours);
```

Listing 10.5 (Continued) *service_enum.c: enumerations for services and statuses, and a service table built with designated initializers.*

```c
    Service pick = WASH_IRON;
    printf("\n3.0 kg %s = Rp %d, ready in %d hours\n", SERVICES[pick].name,
           3 * SERVICES[pick].price_per_kg, SERVICES[pick].hours);

    Status st = WAITING;
    printf("life of an order:");
    for (int step = 0; step < 5; step++) {
        printf(" %s%s", status_name(st), step < 4 ? " ->" : "\n");
        st = next_status(st);
    }
    printf("sizeof(Service) = %zu, READY = %d\n", sizeof(Service), (int)READY);
    return 0;
}
```

```
#   service     Rp/kg  hours
0   fold         7000     24
1   wash-iron   10000     48
2   iron         6000     24

3.0 kg wash-iron = Rp 30000, ready in 48 hours
life of an order: waiting -> washing -> ready -> picked up -> picked up
sizeof(Service) = 4, READY = 2
```

`status_name` uses a `switch` over the enumeration. Because the `switch` lists every value of `Status` and has no `default`, GCC's `-Wall` would warn if someone added a new status, say `DELIVERED`, and forgot to give it a name here: the compiler helps you keep the program complete. The `return "?"` after the `switch` is reached only if a variable holds a number that is not one of the names, which C does not prevent: an `enum` variable is an integer underneath, and `sizeof(Service)` is 4 because it is stored as an `int`. That is the limit of enumerations in C: they make code readable, but they do not make wrong values impossible, so input from outside must still be checked.

Python's `enum` module offers `IntEnum`, whose members are named integers that print as their names when asked and can be used as indexes [5]. **Listing 10.6** builds the same table as a dictionary keyed by the enumeration.

Listing 10.6 `service_enum.py`: `IntEnum` and a dictionary as the service table.

```
# service_enum.py - IntEnum gives names to codes; a dict of tuples is the table.
from enum import IntEnum


class Service(IntEnum):
    FOLD = 0
    WASH_IRON = 1
    IRON_ONLY = 2


class Status(IntEnum):
    WAITING = 0
    WASHING = 1
    READY = 2
    PICKED_UP = 3


SERVICES = {                        # name, price per kg, hours
    Service.FOLD: ("fold", 7000, 24),
    Service.WASH_IRON: ("wash-iron", 10000, 48),
    Service.IRON_ONLY: ("iron", 6000, 24),
}

STATUS_NAME = {Status.WAITING: "waiting", Status.WASHING: "washing",
               Status.READY: "ready", Status.PICKED_UP: "picked up"}


def next_status(s):
    return s if s == Status.PICKED_UP else Status(s + 1)


def main():
    print("%-3s %-10s %6s %6s" % ("#", "service", "Rp/kg", "hours"))
    for s in Service:
        name, price, hours = SERVICES[s]
        print("%-3d %-10s %6d %6d" % (s, name, price, hours))

    name, price, hours = SERVICES[Service.WASH_IRON]
    print("\n3.0 kg %s = Rp %d, ready in %d hours" % (name, 3 * price, hours))
```

Listing 10.6 (Continued) *service_enum.py: IntEnum and a dictionary as the service table.*

```
    st = Status.WAITING
    line = "life of an order:"
    for step in range(5):
        line += " %s%s" % (STATUS_NAME[st], " ->" if step < 4 else "")
        st = next_status(st)
    print(line)
    print("sizeof(Service) = %d, READY = %d" % (4, Status.READY))  # C: an int


if __name__ == "__main__":
    main()
```

10.5 Memory on Demand: `malloc` and `free`

Every variable we have used so far was created in one of two ways. A variable declared outside all functions (or with `static`) exists for the whole run of the program. A variable declared inside a function exists while the function runs and disappears when it returns; Chapter 5 described this as the *automatic* storage on the *stack*. Neither kind lets a program decide *while it runs* how many orders to keep.

The third kind is **dynamic** memory, taken from a large region called the *heap* when the program asks for it, and kept until the program gives it back. Two functions from `<stdlib.h>` do this [1], [4]:

- `malloc(n)` reserves a block of `n` bytes and returns a pointer to its first byte, of type `void *`, which C converts automatically to any other pointer type. If no memory is available it returns `NULL`. The contents of the block are *not* initialized.
- `free(p)` gives back the block that `p` points to, which must be a pointer that `malloc` returned and that has not been freed yet. `free(NULL)` does nothing.

The idiom to create one structure on the heap is

```
Node *n = malloc(sizeof *n);
if (n == NULL) { /* report and stop, or give up this order */ }
```

Writing `sizeof *n` instead of `sizeof(Node)` means “the size of whatever `n` points to”, which stays right if the type of `n` is ever changed.

The block lives until `free`, *independently of the function that created it*. That is the whole point: a function can create an order, return a pointer to it, and the order survives the return. It also brings three duties, which we will call the three rules of `malloc`:

1. **Check for `NULL`.** Writing through a null pointer crashes the program.
2. **Free every block exactly once.** A block that is never freed is a *memory leak*: harmless in a program that runs for a second, fatal in a cashier that runs all day and leaks a little with every order. A block freed twice corrupts the heap.
3. **Never use a block after freeing it.** The pointer still holds the old address, but the memory may already belong to something else. *Use after free* is seventh and *NULL pointer dereference* thirteenth in MITRE’s 2025 list of the most dangerous software weaknesses [6]; both are exactly these rules broken [7].

The sanitizer that the test runner of this book switches on (`-fsanitize=address`) catches all three: it reports every write through a bad pointer, every use of freed memory, and, when the program ends, every block that was never freed.

Python has no `malloc` and no `free`. Every object is created on Python's own heap, and the interpreter frees it automatically when no variable or other object refers to it any more, a service called *garbage collection* (in CPython, mostly by reference counting). That is why the Python versions in this chapter are shorter and have no clean-up code.

10.6 A Structure That Points to Itself: the Linked List

10.6.1 The idea

A structure may contain a pointer to a structure of *its own type*:

```
typedef struct Node {
    int id;
    char customer[24];
    long amount;
    struct Node *next;
} Node;
```

A structure like this is called *self-referential*. Each node holds one order and the address of another node. Starting from one node and following the `next` pointers visits node after node, like following a treasure hunt where every clue tells you where the next clue is hidden. The last node's `next` is `NULL`, which means "there is no next". The address of the first node is kept in a variable called `head`; when the list is empty, `head` is `NULL`. This chain is a **singly linked list** [8], shown in part (a) of **Figure 10.2**.

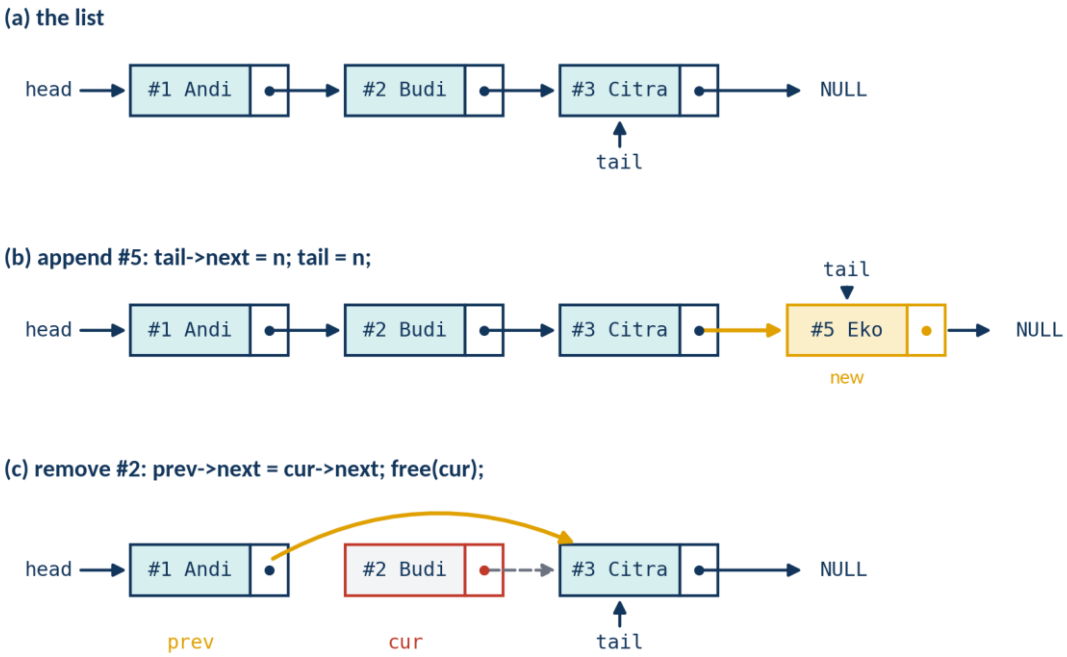


Figure 10.2 A singly linked list of orders: (a) `head` points at the first node and every `next` at the following one, the last at `NULL`; (b) appending changes the last node's `next` and the `tail` pointer; (c) removing a node from the middle bridges over it with one assignment.

Compared with an array, a linked list gives up one thing and gains three. It gives up *direct access*: there is no `list[5]`; to reach the sixth order you must follow five pointers from the head. It gains, first, that nodes are created one at a time with `malloc`, so the list is exactly as long as the day is busy. Second, a node can be inserted or removed anywhere by changing one or two pointers,

without shifting any other node. Third, the nodes never move in memory, so a pointer to a node stays valid as long as the node exists. **Table 10.1** compares the work of the common operations for n orders.

Table 10.1 The work of common operations on n orders, for an array and for a linked list that keeps pointers to its first and last node.

Operation	Array (Chapter 6)	Linked list with head and tail
reach the k-th order	one step	k steps along `next`
add at the end	one step, if there is room	one step, using `tail`
add at the front	shift all n orders	one step
remove order k	shift the $n - k$ orders after it	walk to it, then one step
grow beyond the planned size	impossible without copying	always, one `malloc` per order

For the queue at a laundry counter, where orders are added at the end and leave from anywhere, the list wins.

10.6.2 Walking the list

Almost every list operation is built on one loop, the *traversal*:

```
for (const Node *p = head; p != NULL; p = p->next)
    /* do something with p->id, p->customer, p->amount */
```

`p` starts at the first node; after each node it moves to the node's successor; when it reaches `NULL`, the walk is over. For an empty list, `head` is `NULL`, and the body runs zero times, with no special case. Printing, counting, adding up and searching are all this loop with a different body.

10.6.3 Adding a node

To add at the **front** takes two assignments, and their order matters: first the new node points to the old first node, `n->next = head;`, then the list starts at the new node, `head = n;`. Done the other way around, the old list would be lost, because `head` would already point to `n` when we try to read the old first node from it.

To add at the **end**, we could walk to the last node every time, but that costs n steps. Instead the list keeps a second pointer, `tail`, to its last node. Appending is then, as part (b) of **Figure 10.2** shows, `tail->next = n; tail = n;`, and the only special case is the empty list, where there is no last node and the new node becomes the head as well.

We keep `head`, `tail` and the number of nodes together in one structure, `List`, so that functions can receive the whole list through one pointer and change all three.

10.6.4 Removing a node

To remove a node, the node *before* it must be told to point past it, as part (c) of **Figure 10.2** shows: `prev->next = cur->next;`. The walk therefore keeps two pointers, `prev` one step behind `cur`. Three cases need care: if the node is the first one, there is no `prev`, and `head` moves instead; if it is the last one, `tail` must move back to `prev`; and only after the node has been unlinked may it be freed.

Let us trace it by hand on the list Dewi (#4), Andi (#1), Budi (#2), Citra (#3), removing #2. At the start, `prev` is `NULL` and `cur` is Dewi. Dewi's id is 4, not 2, so the walk moves on: `prev` becomes Dewi and `cur` becomes Andi. Andi's id is 1, so again: `prev` is Andi and `cur` is Budi. Budi's id is 2, and the walk stops with `prev` pointing at Andi and `cur` at Budi. Now `prev` is not `NULL`, so the program

executes `prev->next = cur->next`, which copies Budi's `next`, the address of Citra, into Andi's `next`: Andi now points straight at Citra, and Budi is no longer reachable from the head. `tail` points at Citra, not at Budi, so it stays. Only now is Budi freed, and the count goes from 4 to 3. Had we looked for #9 instead, the walk would have moved all the way to the end, `cur` would have become `NULL`, and the function would have returned 0 without changing anything. Had we removed Dewi, the walk would have stopped at once with `prev` still `NULL`, and `head` would have moved to Andi. Three cases, one loop, and every one of them can be checked with a pencil before a line of code is typed.

One more remark about cost. Removing an order that has already been found takes one assignment, however long the list is; what takes time is *finding* it, which in the worst case walks all n nodes. In an array the search takes just as long, and afterwards every order behind it must still be shifted. That is why the list wins for a queue whose members often leave from the middle.

10.6.5 Freeing the whole list

At the end, every node must be freed. The loop that looks natural, `for (p = head; p != NULL; p = p->next) free(p);`, breaks rule 3: after `free(p)`, the update `p = p->next` reads from a node that no longer exists. The correct loop saves the successor *before* freeing:

```
while (p != NULL) { Node *next = p->next; free(p); p = next; }
```

10.6.6 The program

Listing 10.7 implements everything of this section and exercises it.

Listing 10.7 `order_list.c`: a linked list of orders with `push_front`, `append`, `find`, `remove_id` and `free_all`.

```c
/* order_list.c - a singly linked list of laundry orders: every node is a
 * struct allocated with malloc that points to the next node. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Node {
    int id;
    char customer[24];
    long amount;
    struct Node *next;       /* the next node, or NULL after the last one */
} Node;

typedef struct {
    Node *head;              /* first node, NULL when the list is empty */
    Node *tail;              /* last node, so that appending is quick */
    int count;
} List;

static Node *new_node(int id, const char *customer, long amount)
{
    Node *n = malloc(sizeof *n);
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(EXIT_FAILURE);
    }
    n->id = id;
    snprintf(n->customer, sizeof n->customer, "%s", customer);
    n->amount = amount;
    n->next = NULL;
    return n;
}

static void push_front(List *l, Node *n)
{
    n->next = l->head;       /* 1: the new node points at the old first */
    l->head = n;             /* 2: the list starts at the new node */
    if (l->tail == NULL)
        l->tail = n;
    l->count++;
}
```

Listing 10.7 (Continued) *order_list.c: a linked list of orders with push_front, append, find, remove_id and free_all.*

```
static void append(List *l, Node *n)
{
    if (l->tail == NULL)
        l->head = n;         /* empty list: n is also the first node */
    else
        l->tail->next = n;   /* the old last node points at n */
    l->tail = n;
    l->count++;
}

static void print_list(const List *l)
{
    printf("head");
    for (const Node *p = l->head; p != NULL; p = p->next)
        printf(" -> [#%d %s]", p->id, p->customer);
    printf(" -> NULL   count=%d\n", l->count);
}

static Node *find(const List *l, const char *customer)
{
    for (Node *p = l->head; p != NULL; p = p->next)
        if (strcmp(p->customer, customer) == 0)
            return p;
    return NULL;
}

/* Take the node with this id out of the list and free it; 1 if found. */
static int remove_id(List *l, int id)
{
    Node *prev = NULL, *cur = l->head;
    while (cur != NULL && cur->id != id) {
        prev = cur;
        cur = cur->next;
    }
    if (cur == NULL)
        return 0;
    if (prev == NULL)
        l->head = cur->next;            /* removing the first node */
    else
        prev->next = cur->next;         /* bridge over cur */
    if (l->tail == cur)
        l->tail = prev;                 /* removing the last node */
    free(cur);
    l->count--;
    return 1;
}
```

Listing 10.7 (Continued) *order_list.c: a linked list of orders with push_front, append, find, remove_id and free_all.*

```
static void free_all(List *l)
{
    Node *p = l->head;
    while (p != NULL) {
        Node *next = p->next;            /* read next BEFORE free */
        free(p);
        p = next;
    }
    l->head = l->tail = NULL;
    l->count = 0;
}

int main(void)
{
    List queue = {NULL, NULL, 0};
    print_list(&queue);
    append(&queue, new_node(1, "Andi", 24500));
    append(&queue, new_node(2, "Budi", 20000));
    append(&queue, new_node(3, "Citra", 24000));
    print_list(&queue);
    push_front(&queue, new_node(4, "Dewi", 38500));      /* a VIP customer */
    print_list(&queue);

    long sum = 0;
    for (const Node *p = queue.head; p != NULL; p = p->next)
        sum += p->amount;
    printf("total Rp %ld\n", sum);

    const char *who[] = {"Citra", "Eko"};
    for (int i = 0; i < 2; i++) {
        Node *f = find(&queue, who[i]);
        if (f != NULL)
            printf("find %s: #%d, Rp %ld\n", who[i], f->id, f->amount);
        else
            printf("find %s: not in the list\n", who[i]);
    }
```

Listing 10.7 (Continued) *order_list.c: a linked list of orders with push_front, append, find, remove_id and free_all.*

```
    int ids[] = {4, 2, 3, 9};
    for (int i = 0; i < 4; i++) {
        int ok = remove_id(&queue, ids[i]);
        printf("remove #%d: %s  ", ids[i], ok ? "done" : "no such order");
        print_list(&queue);
    }
    append(&queue, new_node(5, "Eko", 14000));           /* tail still right? */
    print_list(&queue);
    free_all(&queue);
    print_list(&queue);
    return 0;
}
```

```
head -> NULL   count=0
head -> [#1 Andi] -> [#2 Budi] -> [#3 Citra] -> NULL   count=3
head -> [#4 Dewi] -> [#1 Andi] -> [#2 Budi] -> [#3 Citra] -> NULL   count=4
total Rp 107000
find Citra: #3, Rp 24000
find Eko: not in the list
remove #4: done  head -> [#1 Andi] -> [#2 Budi] -> [#3 Citra] -> NULL   count=3
remove #2: done  head -> [#1 Andi] -> [#3 Citra] -> NULL   count=2
remove #3: done  head -> [#1 Andi] -> NULL   count=1
remove #9: no such order  head -> [#1 Andi] -> NULL   count=1
head -> [#1 Andi] -> [#5 Eko] -> NULL   count=2
head -> NULL   count=0
```

Follow the output with **Figure 10.2** in hand. The empty list prints as `head -> NULL`. Three appends build the list of part (a). The VIP customer Dewi goes to the front. Removing #4 removes the *first* node (the `prev == NULL` case); removing #2 removes a middle node, as in part (c); removing #3 removes the *last* node, and `tail` must move back to Andi. The line after that is the proof: appending Eko afterwards puts him right after Andi. Had `remove_id` forgotten to move `tail`, Eko would have been linked behind the freed node #3, and the sanitizer would have stopped the program with a *heap-use-after-free* report. Removing #9, which does not exist, changes nothing. Finally `free_all` gives every node back, and the test runner's leak check confirms that nothing was left behind.

The Python version, **Listing 10.8**, deliberately does *not* use Python's built-in list. It builds the same chain of `Node` objects with a `next` attribute and `None` in place of `NULL`, so that you can see the same algorithm in both languages. `free_all` only sets `head` and `tail` to `None`; once no variable refers to the nodes, Python frees them.

Listing 10.8 `order_list.py`: the same linked list in Python, built from `Node` objects.

```
# order_list.py - the same singly linked list, built from Node objects that
# point to the next Node (Python's own list is NOT used, on purpose).


class Node:
    def __init__(self, id, customer, amount):
        self.id = id
        self.customer = customer[:23]
        self.amount = amount
        self.next = None             # the next node, or None after the last


class List:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0


def push_front(l, n):
    n.next = l.head
    l.head = n
    if l.tail is None:
        l.tail = n
    l.count += 1


def append(l, n):
    if l.tail is None:
        l.head = n
    else:
        l.tail.next = n
    l.tail = n
    l.count += 1


def print_list(l):
    s, p = "head", l.head
    while p is not None:
        s += " -> [#%d %s]" % (p.id, p.customer)
        p = p.next
    print(s + " -> NULL   count=%d" % l.count)
```

Listing 10.8 (Continued) *order_list.py: the same linked list in Python, built from Node objects.*

```
def find(l, customer):
    p = l.head
    while p is not None:
        if p.customer == customer:
            return p
        p = p.next
    return None


def remove_id(l, id):
    prev, cur = None, l.head
    while cur is not None and cur.id != id:
        prev, cur = cur, cur.next
    if cur is None:
        return False
    if prev is None:
        l.head = cur.next
    else:
        prev.next = cur.next
    if l.tail is cur:
        l.tail = prev
    l.count -= 1                        # Python frees the node for us
    return True


def free_all(l):
    l.head = l.tail = None              # no node is reachable any more
    l.count = 0


def main():
    queue = List()
    print_list(queue)
    append(queue, Node(1, "Andi", 24500))
    append(queue, Node(2, "Budi", 20000))
    append(queue, Node(3, "Citra", 24000))
    print_list(queue)
    push_front(queue, Node(4, "Dewi", 38500))
    print_list(queue)

    total, p = 0, queue.head
    while p is not None:
        total += p.amount
        p = p.next
    print("total Rp %d" % total)
```

Listing 10.8 (Continued) *order_list.py: the same linked list in Python, built from Node objects.*

```
    for who in ["Citra", "Eko"]:
        f = find(queue, who)
        if f is not None:
            print("find %s: #%d, Rp %d" % (who, f.id, f.amount))
        else:
            print("find %s: not in the list" % who)

    for id in [4, 2, 3, 9]:
        ok = remove_id(queue, id)
        print("remove #%d: %s  " % (id, "done" if ok else "no such order"), end="")
        print_list(queue)
    append(queue, Node(5, "Eko", 14000))
    print_list(queue)
    free_all(queue)
    print_list(queue)


if __name__ == "__main__":
    main()
```

10.7 Building Cashier v0.8

Cashier v0.8 combines the pieces. An order is one structure:

- `id`, the order number, given out by the shop in increasing order and never reused;
- `customer`, the tidied name (Chapter 9);
- `service`, of type `Service`, and `tenths`, the weight;
- `amount`, computed once when the order is taken;
- `status`, of type `Status`: `WAITING`, `WASHING` or `READY`;
- `next`, the pointer that links the orders in the shop into a list.

All the orders currently in the shop form one linked list, kept in a structure `Shop` together with the counter for the next order number and the day's takings. The commands are:

- `add <service> <kg> <name>` creates an order with `malloc`, fills it in, and appends it to the list, as in Chapter 9 but without any limit on the number of orders;
- `advance <id>` moves an order one step on, from waiting to washing or from washing to ready (the id may be written `3` or `#3`);
- `pickup <id>` hands a *ready* order to the customer: it is unlinked from the list, its amount is added to the takings, and its memory is freed;
- `list` shows all orders in the shop with their status, and `quit` ends the day, frees every remaining order and prints the takings.

The text helpers of Chapter 9 are copied unchanged into the program; Chapter 11 will move them into a file of their own so that they are written only once. Everything else in **Listing 10.9** is new.

Listing 10.9 `cashier_v8.c`: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.

```c
/* cashier_v8.c - Profitina Laundry cashier v0.8: every order is one struct,
 * the orders in the shop form a linked list, and an order moves through
 * waiting -> washing -> ready until the customer picks it up.
 * Usage: cashier_v8 [--echo]    (--echo repeats each command after "> ") */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NAME_LEN 32
#define LINE_LEN 80

typedef enum { FOLD, WASH_IRON, IRON_ONLY, N_SERVICES } Service;
typedef enum { WAITING, WASHING, READY } Status;

static const struct {
    const char *name;
    int price;                                  /* rupiah per kg */
} SERVICES[N_SERVICES] = {
    [FOLD] = {"fold", 7000},
    [WASH_IRON] = {"wash-iron", 10000},
    [IRON_ONLY] = {"iron", 6000},
};
static const char *STATUS_NAME[] = {
    [WAITING] = "waiting", [WASHING] = "washing", [READY] = "ready"};

typedef struct Order {
    int id;
    char customer[NAME_LEN];
    Service service;
    int tenths;                                 /* weight in tenths of a kg */
    long amount;                                /* rupiah */
    Status status;
    struct Order *next;
} Order;

typedef struct {
    Order *head, *tail;                         /* the orders in the shop */
    int count, next_id;
    int picked;                                 /* orders picked up today */
    long takings;                               /* rupiah received today */
} Shop;
```

Listing 10.9 (Continued) *cashier_v8.c: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.*

```
/* ---- text helpers from Chapter 9 ----------------------------------- */
static int read_line(char *buf, int size)
{
    if (fgets(buf, size, stdin) == NULL)
        return 0;
    size_t n = strlen(buf);
    if (n > 0 && buf[n - 1] == '\n') {
        buf[--n] = '\0';
    } else if (n == (size_t)size - 1) {
        int c = getchar();
        if (c != EOF && c != '\n') {
            while (c != EOF && c != '\n')
                c = getchar();
            return -1;
        }
    }
    if (n > 0 && buf[n - 1] == '\r')
        buf[--n] = '\0';
    return 1;
}

static size_t next_word(const char **p, char *out, size_t size)
{
    const char *s = *p;
    size_t n = 0;
    while (isspace((unsigned char)*s))
        s++;
    while (*s != '\0' && !isspace((unsigned char)*s)) {
        if (n + 1 < size)
            out[n] = *s;
        n++;
        s++;
    }
    out[n < size ? n : size - 1] = '\0';
    *p = s;
    return n;
}
```

Listing 10.9 (Continued) *cashier_v8.c: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.*

```
static void tidy_name(const char *src, char *dst, size_t size)
{
    size_t n = 0;
    int new_word = 1;
    for (size_t i = 0; src[i] != '\0' && n + 1 < size; i++) {
        unsigned char c = (unsigned char)src[i];
        if (isspace(c)) {
            new_word = 1;
            continue;
        }
        if (new_word && n > 0 && n + 2 < size)
            dst[n++] = ' ';
        dst[n++] = (char)(new_word ? toupper(c) : tolower(c));
        new_word = 0;
    }
    dst[n] = '\0';
}

static int parse_tenths(const char *s, int *out)
{
    int whole = 0, digits = 0, tenth = 0;
    for (; isdigit((unsigned char)*s); s++) {
        if (++digits > 3)
            return 0;
        whole = whole * 10 + (*s - '0');
    }
    if (digits == 0)
        return 0;
    if (*s == '.') {
        if (!isdigit((unsigned char)s[1]))
            return 0;
        tenth = s[1] - '0';
        s += 2;
    }
    if (*s != '\0')
        return 0;
    *out = whole * 10 + tenth;
    return 1;
}
```

Listing 10.9 (Continued) *cashier_v8.c: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.*

```
static void rupiah(long n, char *out)          /* out: 32 bytes */
{
    char digits[24];
    int len = snprintf(digits, sizeof digits, "%ld", n), k = 0;
    for (int i = 0; i < len; i++) {
        if (i > 0 && (len - i) % 3 == 0)
            out[k++] = '.';
        out[k++] = digits[i];
    }
    out[k] = '\0';
}

/* ---- the list of orders ------------------------------------------- */
static Order *find_id(const Shop *shop, int id)
{
    for (Order *o = shop->head; o != NULL; o = o->next)
        if (o->id == id)
            return o;
    return NULL;
}

static void append(Shop *shop, Order *o)
{
    if (shop->tail == NULL)
        shop->head = o;
    else
        shop->tail->next = o;
    shop->tail = o;
    shop->count++;
}

static Order *unlink_id(Shop *shop, int id)     /* take out, do not free */
{
    Order *prev = NULL, *cur = shop->head;
    while (cur != NULL && cur->id != id) {
        prev = cur;
        cur = cur->next;
    }
    if (cur == NULL)
        return NULL;
    if (prev == NULL)
        shop->head = cur->next;
    else
        prev->next = cur->next;
    if (shop->tail == cur)
        shop->tail = prev;
```

Listing 10.9 (Continued) *cashier_v8.c: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.*

```
    cur->next = NULL;
    shop->count--;
    return cur;
}

static void free_all(Shop *shop)
{
    while (shop->head != NULL) {
        Order *next = shop->head->next;
        free(shop->head);
        shop->head = next;
    }
    shop->tail = NULL;
    shop->count = 0;
}

static void print_order(const Order *o)
{
    char m[32];
    rupiah(o->amount, m);
    printf("  #%d %s, %s, %d.%d kg = Rp %s (%s)\n", o->id, o->customer,
           SERVICES[o->service].name, o->tenths / 10, o->tenths % 10, m,
           STATUS_NAME[o->status]);
}

/* ---- commands ------------------------------------------------------- */
static void cmd_add(Shop *shop, const char *rest)
{
    char svc[16], kg[16], name[NAME_LEN];
    int s = -1, t = 0;
    if (next_word(&rest, svc, sizeof svc) == 0 ||
        next_word(&rest, kg, sizeof kg) == 0) {
        puts("  usage: add <service> <kg> <name>");
        return;
    }
    for (int i = 0; i < N_SERVICES; i++)
        if (strcmp(svc, SERVICES[i].name) == 0)
            s = i;
    if (s < 0) {
        printf("  unknown service '%s' (fold, wash-iron, iron)\n", svc);
        return;
    }
    if (!parse_tenths(kg, &t) || t < 1 || t > 500) {
        puts("  weight must look like 3 or 3.5 (0.1 to 50.0 kg)");
        return;
    }
```

Listing 10.9 (Continued) *cashier_v8.c: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.*

```
    tidy_name(rest, name, sizeof name);
    if (name[0] == '\0') {
        puts("  the customer's name is missing");
        return;
    }
    Order *o = malloc(sizeof *o);
    if (o == NULL) {
        puts("  out of memory - order not taken");
        return;
    }
    o->id = shop->next_id++;
    strcpy(o->customer, name);
    o->service = (Service)s;
    o->tenths = t;
    o->amount = (long)t * SERVICES[s].price / 10;
    o->status = WAITING;
    o->next = NULL;
    append(shop, o);
    print_order(o);
}

/* Reads "#3" or "3" from rest; returns the id, or 0 if there is none. */
static int read_id(const char *rest)
{
    char word[16];
    const char *w = word;
    if (next_word(&rest, word, sizeof word) == 0)
        return 0;
    if (*w == '#')
        w++;
    int id = 0, digits = 0;
    for (; isdigit((unsigned char)*w); w++)
        if (++digits <= 6)
            id = id * 10 + (*w - '0');
    return (*w == '\0' && digits > 0 && digits <= 6) ? id : 0;
}
```

Listing 10.9 (Continued) *cashier_v8.c: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.*

```
static void cmd_advance(Shop *shop, const char *rest)
{
    int id = read_id(rest);
    Order *o = id ? find_id(shop, id) : NULL;
    if (o == NULL)
        printf("  no order #%d in the shop\n", id);
    else if (o->status == READY)
        printf("  #%d is already ready - use pickup\n", id);
    else {
        o->status = (Status)(o->status + 1);
        print_order(o);
    }
}

static void cmd_pickup(Shop *shop, const char *rest)
{
    int id = read_id(rest);
    Order *o = id ? find_id(shop, id) : NULL;
    char m[32];
    if (o == NULL) {
        printf("  no order #%d in the shop\n", id);
        return;
    }
    if (o->status != READY) {
        printf("  #%d is still %s\n", id, STATUS_NAME[o->status]);
        return;
    }
    unlink_id(shop, id);
    rupiah(o->amount, m);
    printf("  #%d %s picked up, paid Rp %s\n", o->id, o->customer, m);
    shop->picked++;
    shop->takings += o->amount;
    free(o);
}

static void cmd_list(const Shop *shop)
{
    char m[32];
    printf("  %3s  %-20s  %-9s  %7s  %8s  %s\n",
           "No", "Customer", "Service", "Weight", "Amount", "Status");
    for (const Order *o = shop->head; o != NULL; o = o->next) {
        rupiah(o->amount, m);
        printf("  %3d  %-20.20s  %-9s  %2d.%d kg  %8s  %s\n", o->id,
               o->customer, SERVICES[o->service].name, o->tenths / 10,
               o->tenths % 10, m, STATUS_NAME[o->status]);
    }
```

Listing 10.9 (Continued) *cashier_v8.c: Profitina Laundry cashier v0.8, with one structure per order and the orders in the shop kept as a linked list.*

```
    printf("  %d order(s) in the shop\n", shop->count);
}

int main(int argc, char *argv[])
{
    int echo = argc > 1 && strcmp(argv[1], "--echo") == 0;
    char line[LINE_LEN], cmd[16], m[32];
    Shop shop = {NULL, NULL, 0, 1, 0, 0};

    puts("Profitina Laundry cashier v0.8 - add, advance, pickup, list, quit");
    for (;;) {
        printf("> ");
        fflush(stdout);
        int r = read_line(line, sizeof line);
        if (r == 0) {
            printf("\n");
            break;
        }
        if (echo)
            printf(r < 0 ? "%.40s...\n" : "%s\n", line);
        if (r < 0) {
            puts("  line too long (at most 79 characters), ignored");
            continue;
        }
        const char *rest = line;
        if (next_word(&rest, cmd, sizeof cmd) == 0)
            continue;
        if (strcmp(cmd, "add") == 0)
            cmd_add(&shop, rest);
        else if (strcmp(cmd, "advance") == 0)
            cmd_advance(&shop, rest);
        else if (strcmp(cmd, "pickup") == 0)
            cmd_pickup(&shop, rest);
        else if (strcmp(cmd, "list") == 0)
            cmd_list(&shop);
        else if (strcmp(cmd, "quit") == 0)
            break;
        else
            printf("  unknown command '%s'\n", cmd);
    }
    rupiah(shop.takings, m);
    printf("Bye. %d picked up (Rp %s), %d still in the shop.\n",
           shop.picked, m, shop.count);
    free_all(&shop);
    return 0;
}
```

Some details are worth reading slowly. The service table is an array of an *anonymous* structure type, declared and initialized in one statement with designated initializers. `unlink_id` is `remove_id` of Section 10.6 without the `free`: it returns the unlinked order so that `cmd_pickup` can still print its name and amount and add them to the takings *before* freeing it; freeing first and printing afterwards would be a use after free. `cmd_advance` changes the status with `(Status)(o->status + 1)`, which relies on the statuses being consecutive, and the check `o->status == READY` before it makes sure that the value never leaves the enumeration. And `main` ends with `free_all(&shop)`, so that a normal end of the day leaves no leaked orders.

The Python version, **Listing 10.10**, has the same structure with classes `Order` and `Shop`, `IntEnum` for services and statuses, and the same linked list.

Listing 10.10 `cashier_v8.py`: cashier v0.8 in Python.

```
# cashier_v8.py - Profitina Laundry cashier v0.8: every order is one object,
# the orders in the shop form a linked list, and an order moves through
# waiting -> washing -> ready until the customer picks it up.
# Usage: python3 cashier_v8.py [--echo]
import sys
from enum import IntEnum

NAME_LEN = 32
LINE_LEN = 80


class Service(IntEnum):
    FOLD = 0
    WASH_IRON = 1
    IRON_ONLY = 2


class Status(IntEnum):
    WAITING = 0
    WASHING = 1
    READY = 2


SERVICES = {Service.FOLD: ("fold", 7000), Service.WASH_IRON: ("wash-iron", 10000),
            Service.IRON_ONLY: ("iron", 6000)}
STATUS_NAME = {Status.WAITING: "waiting", Status.WASHING: "washing",
               Status.READY: "ready"}


class Order:
    def __init__(self, id, customer, service, tenths):
        self.id = id
        self.customer = customer
        self.service = service
        self.tenths = tenths
        self.amount = tenths * SERVICES[service][1] // 10
        self.status = Status.WAITING
        self.next = None
```

Listing 10.10 (Continued) *cashier_v8.py: cashier v0.8 in Python.*

```
class Shop:
    def __init__(self):
        self.head = self.tail = None
        self.count = 0
        self.next_id = 1
        self.picked = 0
        self.takings = 0


# ---- text helpers from Chapter 9 ---------------------------------------
def read_line():
    line = sys.stdin.readline()
    if line == "":
        return 0, ""
    line = line[:-1] if line.endswith("\n") else line
    if len(line) > LINE_LEN - 1:
        return -1, line[:LINE_LEN - 1]
    return 1, line[:-1] if line.endswith("\r") else line


def tidy_name(src, size=NAME_LEN):
    return " ".join(w[0].upper() + w[1:].lower() for w in src.split())[:size - 1]


def parse_tenths(s):
    whole, dot, tenth = s.partition(".")
    if not (whole.isascii() and whole.isdigit() and len(whole) <= 3):
        return None
    if dot and not (len(tenth) == 1 and tenth.isdigit()):
        return None
    return int(whole) * 10 + (int(tenth) if dot else 0)


def rupiah(n):
    digits, out = str(n), ""
    for i, d in enumerate(digits):
        if i > 0 and (len(digits) - i) % 3 == 0:
            out += "."
        out += d
    return out
```

Listing 10.10 (Continued) *cashier_v8.py: cashier v0.8 in Python.*

```
# ---- the list of orders -------------------------------------------------
def find_id(shop, id):
    o = shop.head
    while o is not None:
        if o.id == id:
            return o
        o = o.next
    return None


def append(shop, o):
    if shop.tail is None:
        shop.head = o
    else:
        shop.tail.next = o
    shop.tail = o
    shop.count += 1


def unlink_id(shop, id):
    prev, cur = None, shop.head
    while cur is not None and cur.id != id:
        prev, cur = cur, cur.next
    if cur is None:
        return None
    if prev is None:
        shop.head = cur.next
    else:
        prev.next = cur.next
    if shop.tail is cur:
        shop.tail = prev
    cur.next = None
    shop.count -= 1
    return cur


def print_order(o):
    print("  #%d %s, %s, %d.%d kg = Rp %s (%s)"
          % (o.id, o.customer, SERVICES[o.service][0], o.tenths // 10,
             o.tenths % 10, rupiah(o.amount), STATUS_NAME[o.status]))
```

Listing 10.10 (Continued) *cashier_v8.py: cashier v0.8 in Python.*

```
# ---- commands --------------------------------------------------------
def cmd_add(shop, rest):
    words = rest.split(None, 2)
    if len(words) < 2:
        print("  usage: add <service> <kg> <name>")
        return
    svc, kg = words[0][:15], words[1][:15]
    names = [v[0] for v in SERVICES.values()]
    if svc not in names:
        print("  unknown service '%s' (fold, wash-iron, iron)" % svc)
        return
    t = parse_tenths(kg)
    if t is None or not 1 <= t <= 500:
        print("  weight must look like 3 or 3.5 (0.1 to 50.0 kg)")
        return
    name = tidy_name(words[2] if len(words) > 2 else "")
    if name == "":
        print("  the customer's name is missing")
        return
    o = Order(shop.next_id, name, Service(names.index(svc)), t)
    shop.next_id += 1
    append(shop, o)
    print_order(o)


def read_id(rest):
    words = rest.split()
    if not words:
        return 0
    w = words[0][:15]
    w = w[1:] if w.startswith("#") else w
    ok = w.isascii() and w.isdigit() and len(w) <= 6
    return int(w) if ok else 0


def cmd_advance(shop, rest):
    id = read_id(rest)
    o = find_id(shop, id) if id else None
    if o is None:
        print("  no order #%d in the shop" % id)
    elif o.status == Status.READY:
        print("  #%d is already ready - use pickup" % id)
    else:
        o.status = Status(o.status + 1)
        print_order(o)
```

Listing 10.10 (Continued) *cashier_v8.py: cashier v0.8 in Python.*

```
def cmd_pickup(shop, rest):
    id = read_id(rest)
    o = find_id(shop, id) if id else None
    if o is None:
        print("  no order #%d in the shop" % id)
        return
    if o.status != Status.READY:
        print("  #%d is still %s" % (id, STATUS_NAME[o.status]))
        return
    unlink_id(shop, id)
    print("  #%d %s picked up, paid Rp %s" % (o.id, o.customer, rupiah(o.amount)))
    shop.picked += 1
    shop.takings += o.amount


def cmd_list(shop):
    print("  %3s  %-20s  %-9s  %7s  %8s  %s"
          % ("No", "Customer", "Service", "Weight", "Amount", "Status"))
    o = shop.head
    while o is not None:
        print("  %3d  %-20.20s  %-9s  %2d.%d kg  %8s  %s"
              % (o.id, o.customer, SERVICES[o.service][0], o.tenths // 10,
                 o.tenths % 10, rupiah(o.amount), STATUS_NAME[o.status]))
        o = o.next
    print("  %d order(s) in the shop" % shop.count)


def main():
    echo = len(sys.argv) > 1 and sys.argv[1] == "--echo"
    shop = Shop()
    print("Profitina Laundry cashier v0.8 - add, advance, pickup, list, quit")
    while True:
        print("> ", end="", flush=True)
        r, line = read_line()
        if r == 0:
            print()
            break
        if echo:
            print(line[:40] + "..." if r < 0 else line)
        if r < 0:
            print("  line too long (at most 79 characters), ignored")
            continue
        parts = line.split(None, 1)
        if not parts:
            continue
```

Listing 10.10 (Continued) *cashier_v8.py: cashier v0.8 in Python.*

```
        cmd, rest = parts[0][:15], parts[1] if len(parts) > 1 else ""
        if cmd == "add":
            cmd_add(shop, rest)
        elif cmd == "advance":
            cmd_advance(shop, rest)
        elif cmd == "pickup":
            cmd_pickup(shop, rest)
        elif cmd == "list":
            cmd_list(shop)
        elif cmd == "quit":
            break
        else:
            print("  unknown command '%s'" % cmd)
    print("Bye. %d picked up (Rp %s), %d still in the shop."
          % (shop.picked, rupiah(shop.takings), shop.count))


if __name__ == "__main__":
    main()
```

10.8 Running and Testing It

The session in `data/session10.txt` takes three orders, tries to hand over an order that is not ready, moves orders along, hands one over, and takes a new order afterwards:

```
add fold 3.5 andi pratama
add wash-iron 2 budi santoso
add iron 4 citra dewi lestari
list
pickup 2
advance 2
advance #2
advance 2
pickup 2
advance 1
advance 7
list
add fold 5 dewi
list
quit
```

`./cashier_v8 --echo < ../data/session10.txt` and `python3 cashier_v8.py --echo < ../data/session10.txt` both print:

```
Profitina Laundry cashier v0.8 - add, advance, pickup, list, quit
> add fold 3.5 andi pratama
  #1 Andi Pratama, fold, 3.5 kg = Rp 24.500 (waiting)
> add wash-iron 2 budi santoso
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000 (waiting)
> add iron 4 citra dewi lestari
  #3 Citra Dewi Lestari, iron, 4.0 kg = Rp 24.000 (waiting)
> list
   No  Customer              Service     Weight    Amount  Status
    1  Andi Pratama          fold        3.5 kg    24.500  waiting
    2  Budi Santoso          wash-iron   2.0 kg    20.000  waiting
    3  Citra Dewi Lestari    iron        4.0 kg    24.000  waiting
  3 order(s) in the shop
> pickup 2
  #2 is still waiting
> advance 2
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000 (washing)
> advance #2
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000 (ready)
> advance 2
  #2 is already ready - use pickup
> pickup 2
  #2 Budi Santoso picked up, paid Rp 20.000
> advance 1
  #1 Andi Pratama, fold, 3.5 kg = Rp 24.500 (washing)
> advance 7
  no order #7 in the shop
> list
   No  Customer              Service     Weight    Amount  Status
    1  Andi Pratama          fold        3.5 kg    24.500  washing
    3  Citra Dewi Lestari    iron        4.0 kg    24.000  waiting
  2 order(s) in the shop
> add fold 5 dewi
  #4 Dewi, fold, 5.0 kg = Rp 35.000 (waiting)
> list
   No  Customer              Service     Weight    Amount  Status
    1  Andi Pratama          fold        3.5 kg    24.500  washing
    3  Citra Dewi Lestari    iron        4.0 kg    24.000  waiting
    4  Dewi                  fold        5.0 kg    35.000  waiting
  3 order(s) in the shop
> quit
Bye. 1 picked up (Rp 20.000), 3 still in the shop.
```

Check it against the rules. Budi's order could not be picked up while waiting; two `advance` commands made it ready (once as `2`, once as `#2`); a third was refused. After the pickup, Budi's order has left the list, the list closes the gap by itself, and the new order of Dewi receives number 4, not 2, because order numbers are never reused: a receipt with number 2 is still in Budi's pocket. The day ends with one order picked up for Rp 20,000 and three still in the shop.

The test runner of the chapter (`python3 tests/run_tests.py`) runs this session under AddressSanitizer as well. A version of `cmd_pickup` that frees the order before printing it, or a `free_all` that forgets to save `next`, fails the test with a precise report of the line that used the freed memory, which is how such mistakes are found in practice: not by staring at the code, but by running it under a tool that watches every memory access.

10.9 Making It Better

Keep the list in a useful order. At a busy counter, express orders should be washed first. Instead of appending every order at the end, an `insert_sorted` function can walk the list and link a new order in *before* the first order with a later promised time. It uses the two-pointer walk of

`remove_id`: find the node after which the new one belongs (or `NULL` for the front), then `n->next = prev->next; prev->next = n;`. Exercise 10.4 asks for it.

Find the customer's orders, not only one. `find` stops at the first match. For a customer who brought three bags, a walk that prints every match is more useful; with the traversal loop that is a matter of not returning early.

Do not lose the day. Everything v0.8 knows is still in memory. A power cut at four in the afternoon loses every order of the day, and Mbak Rina is back to her paper notebook. Chapter 11 saves the list to a file after every change, which is the most important improvement of all.

10.10 Using It for Real

A structure plus a linked list is the right shape whenever things arrive and leave in an order you cannot predict and you cannot know in advance how many there will be. A motorcycle repair shop keeps the bikes waiting for repair as a list of structures with the plate number, the owner's name, the problem and a status (*waiting*, *in repair*, *waiting for parts*, *ready*); bikes leave when they are ready, not in the order they came. A village health post keeps the queue of the day's patients with their arrival time and a status, and calls the next one from the head of the list. A fruit and durian orchard that takes orders by chat keeps each order with the buyer, the fruit, the weight and a status such as *picked*, *packed*, *sent*. In each case the steps are the same as in this chapter: write down the facts that belong together and turn them into a structure; write down the states as an enumeration and the allowed moves between them; keep the live records in a list; and give every record a number that is never reused.

A concrete example makes the steps visible. For the motorcycle shop, the structure would hold the plate number as a string of 12 characters, the owner's name of 32, the complaint of 80, an enumeration `BikeStatus` with the four states, and the promised day. The command `advance` of the cashier becomes the mechanic's "next step", `pickup` becomes "hand over and take payment", and a new command, `parts <plate>`, moves a bike to *waiting for parts* and back. Almost all of `cashier_v8.c` can be kept: the list functions do not care what an order contains.

When adapting the cashier, keep the three rules of `malloc` in view, and check every program under the sanitizer before you trust it. A small memory leak in a program that is restarted every morning is harmless; the same leak in a program that runs for months, as shop software often does, eventually stops it.

10.11 Common Mistakes, and How to Avoid Them

- **Forgetting the semicolon after a structure declaration.** `struct Order { ... }` must end with `};`. Without it, the compiler joins the declaration to the next line and reports a confusing error there.
- **Using `.` with a pointer or `->` with a structure.** With `struct Order o;` write `o.amount`; with `struct Order *p;` write `p->amount`. Writing `*p.amount` means `*(p.amount)` and does not compile.
- **Passing large structures by value by accident.** It works but copies the structure on every call, and changes are lost. Pass `const struct X *` to read and `struct X *` to change.
- **Comparing structures with `memcmp`.** Padding bytes make two equal structures look different. Compare member by member.
- **Assuming the size of a structure.** 38 is not 40. Use `sizeof` and `offsetof`.

- **Not checking `malloc`.** Always test the result for `NULL` before writing through it.
- **Freeing and then using.** Read everything you need from a node, unlink it, then free it; and save `p->next` before `free(p)` in a loop.
- **Losing the list while inserting.** At the front, set `n->next = head` *before* `head = n`; in the middle, set `n->next = prev->next` *before* `prev->next = n`.
- **Forgetting `tail`.** Every function that can remove the last node or add to an empty list must update `tail` as well as `head`. Test removing the last node and then appending, as `order_list.c` does.
- **Trusting an `enum` variable.** It can hold any `int`. Check values that come from input or from files before using them as an index.

10.12 Chapter Summary

- A `struct` groups named members of different types into one type; members are reached with `.` on a structure and `->` through a pointer. Assignment copies a whole structure, arrays inside included; comparison must be written member by member.
- Members are stored in order, with padding inserted so that each starts at a multiple of its alignment (Equation (10.1)); `sizeof` and `offsetof` give the real layout (**Figure 10.1**).
- Structures are usually passed by pointer: `const struct X *` to read, `struct X *` to change. An array of structures replaces parallel arrays and keeps each record together, even through `qsort`.
- `enum` names integer codes; `typedef` names types; designated initializers build tables indexed by enumeration names. An enumeration does not stop wrong values.
- `malloc` gives memory that lives until `free`. Check for `NULL`, free exactly once, never use after freeing.
- A self-referential structure with a `next` pointer builds a singly linked list. With `head` and `tail` pointers, adding at either end takes one step, and removing a node takes a walk and one assignment (**Table 10.1**, **Figure 10.2**).
- Cashier v0.8 keeps one structure per order in a linked list, moves orders through the states *waiting*, *washing* and *ready*, frees them when they are picked up, and passes its tests under AddressSanitizer in C and with identical output in Python.

10.13 Exercises

1. Draw `struct Item { char code; int qty; char unit; double price; };` in memory on a 64-bit computer, using Equation (10.1), and give its `sizeof`. Then reorder the members to make it as small as possible and give the new size.
2. Write `long total_waiting(const Shop *shop)` for cashier v0.8, which returns the total amount of the orders that are still `WAITING`. Why is the parameter a `const` pointer?
3. The loop `for (Node *p = head; p != NULL; p = p->next) free(p);` is wrong. Explain, step by step, what happens for a list of two nodes, and name the rule of Section 10.5 it breaks.
4. Write `void insert_sorted(List *l, Node *n)` for the list of `order_list.c`, which keeps the list sorted by amount, smallest first. Test it by inserting orders of 24500, 20000, 38500 and 24000 into an empty list and printing the list after each insertion. Which of the cases “empty list”, “new first node”, “new last node” and “middle” does each insertion exercise?
5. Write a function that reverses a linked list *in place*, by turning every `next` pointer around, without creating or copying any node, and test it on the list 10, 20, 30, 40, 50 in C and in Python.

6. Add a `status` value `DELIVERED` to cashier v0.8 for orders that are brought to the customer's home instead of picked up. List every place in `cashier_v8.c` that must change, and say which of them the compiler would point out by itself.

10.14 Solutions to the Exercises

1. `code` takes byte 0. `qty` needs alignment 4: $o = 1$, $o' = 4\lceil 1/4 \rceil = 4$, so bytes 1 to 3 are padding and `qty` takes 4 to 7. `unit` takes byte 8. `price` needs alignment 8: $o = 9$, $o' = 8\lceil 9/8 \rceil = 16$, so bytes 9 to 15 are padding and `price` takes 16 to 23. The size is 24, of which only 14 bytes are data. Ordered `price, qty, code, unit`, the members take 8 + 4 + 1 + 1 = 14 bytes at offsets 0, 8, 12 and 13, and the size is rounded up to the next multiple of 8, which is 16.

2. The function walks the list and adds up:

```
long total_waiting(const Shop *shop)
{
    long sum = 0;
    for (const Order *o = shop->head; o != NULL; o = o->next)
        if (o->status == WAITING)
            sum += o->amount;
    return sum;
}
```

The pointer avoids copying the `Shop` structure (whose copy would share the same nodes anyway), and `const` documents and enforces that counting does not change the shop. In the session of Section 10.8, just before `quit`, it returns 24.000 + 35.000 = 59.000 (Citra and Dewi).

3. With nodes A → B: `p` is A; `free(A)`; then `p = p->next` reads the `next` member of A, whose memory has just been given back. It may still hold the address of B, so the program seems to work, or it may hold anything, so the loop jumps to a random address. This breaks rule 3, *never use a block after freeing it*; the sanitizer reports a heap-use-after-free on the `p = p->next` line. The fix saves `p->next` before the `free`, as in `free_all`.

4. Walk with `prev` and `cur` until `cur` is `NULL` or `cur->amount` is larger than `n->amount`, then link `n` between them:

```
void insert_sorted(List *l, Node *n)
{
    Node *prev = NULL, *cur = l->head;
    while (cur != NULL && cur->amount <= n->amount) {
        prev = cur;
        cur = cur->next;
    }
    n->next = cur;
    if (prev == NULL)
        l->head = n;
    else
        prev->next = n;
    if (cur == NULL)
        l->tail = n;
    l->count++;
}
```

24500 goes into the empty list (it is head and tail at once); 20000 becomes the new first node; 38500 the new last node; 24000 goes in the middle, between 20000 and 24500. Using `<=` in the loop puts a new order *after* existing orders with the same amount, so orders with equal amounts stay in the order they arrived.

5. Keep three pointers: the part already reversed starts at `prev`, the node being turned is `head`, and its successor is saved before its pointer is turned. **Listing 10.11** shows the function and

Listing 10.12 the Python version; reversing twice gives back the original list, and reversing an empty list returns `NULL`.

Listing 10.11 `list_reverse.c`: reversing a linked list in place.

```
/* list_reverse.c - Exercise 10.5: reverse a linked list in place by turning
 * every next pointer around; no node is copied or allocated. */
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

static Node *push_front(Node *head, int value)
{
    Node *n = malloc(sizeof *n);
    if (n == NULL)
        exit(EXIT_FAILURE);
    n->value = value;
    n->next = head;
    return n;
}

static void print(const char *label, const Node *p)
{
    printf("%-9s", label);
    for (; p != NULL; p = p->next)
        printf("%d -> ", p->value);
    printf("NULL\n");
}

static Node *reverse(Node *head)
{
    Node *prev = NULL;
    while (head != NULL) {
        Node *next = head->next;   /* 1: remember the rest of the list */
        head->next = prev;         /* 2: turn this arrow around */
        prev = head;               /* 3: this node is now the new front */
        head = next;               /* 4: go on with the rest */
    }
    return prev;
}
```

Listing 10.11 (Continued) *list_reverse.c: reversing a linked list in place.*

```
int main(void)
{
    Node *head = NULL;
    for (int v = 50; v >= 10; v -= 10)
        head = push_front(head, v);
    print("before:", head);
    head = reverse(head);
    print("after:", head);
    head = reverse(head);
    print("again:", head);
    print("empty:", reverse(NULL));
    while (head != NULL) {
        Node *next = head->next;
        free(head);
        head = next;
    }
    return 0;
}
```

Listing 10.12 `list_reverse.py`: the same in Python.

```
# list_reverse.py - Exercise 10.5: reverse a linked list in place.


class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next


def show(label, p):
    s = "%-9s" % label
    while p is not None:
        s += "%d -> " % p.value
        p = p.next
    print(s + "NULL")


def reverse(head):
    prev = None
    while head is not None:
        nxt = head.next            # 1: remember the rest of the list
        head.next = prev           # 2: turn this arrow around
        prev = head                # 3: this node is now the new front
        head = nxt                 # 4: go on with the rest
    return prev


def main():
    head = None
    for v in range(50, 0, -10):
        head = Node(v, head)
    show("before:", head)
    head = reverse(head)
    show("after:", head)
    head = reverse(head)
    show("again:", head)
    show("empty:", reverse(None))


if __name__ == "__main__":
    main()
```

Both print:

```
before:  10 -> 20 -> 30 -> 40 -> 50 -> NULL
after:   50 -> 40 -> 30 -> 20 -> 10 -> NULL
again:   10 -> 20 -> 30 -> 40 -> 50 -> NULL
empty:   NULL
```

6. Add `DELIVERED` to `Status` after `READY`; add `[DELIVERED] = "delivered"` to `STATUS_NAME`; add a command `deliver <id>` that, like `pickup`, accepts only `READY` orders, unlinks and frees them, and counts them; and make sure `cmd_advance` still stops at `READY` (its test `o->status == READY` does). The compiler points out none of these by itself in this program, because `STATUS_NAME` is an array, not a `switch`; a `switch` over `Status` without `default`, as in `service_enum.c`, would make `-Wall` warn about the missing case. That is a good reason to turn enumeration values into names with a `switch`.

References

[1] ISO/IEC 9899:2018, *Information technology — Programming languages — C* (C17). Geneva, Switzerland: ISO, 2018, §6.7.2.1 “Structure and union specifiers”, §6.7.2.2 “Enumeration specifiers”, §7.22.3 “Memory management functions”.

[2] Python Software Foundation, “dataclasses — Data Classes,” *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/dataclasses.html (accessed Sep. 2026).

[3] Python Software Foundation, “ctypes — A foreign function library for Python,” *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/ctypes.html (accessed Sep. 2026).

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988, ch. 6 “Structures” and appendix B5.

[5] Python Software Foundation, “enum — Support for enumerations,” *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/enum.html (accessed Sep. 2026).

[6] MITRE, “2025 CWE Top 25 Most Dangerous Software Weaknesses,” Dec. 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

[7] Software Engineering Institute, Carnegie Mellon University, *SEI CERT C Coding Standard*, rules MEM30-C “Do not access freed memory” and MEM31-C “Free dynamically allocated memory when no longer needed.” [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c (accessed Sep. 2026).

[8] P. J. Deitel and H. M. Deitel, *C How to Program*, 9th ed. Hoboken, NJ, USA: Pearson, 2022, ch. 10 “Structures, Unions, Bit Manipulation and Enumerations” and ch. 12 “Data Structures.”

Chapter 11

File Processing: Data That Survives Power-Off

Everything our programs have known so far lived in the computer's working memory, and working memory forgets everything the moment the program ends or the power fails. This chapter teaches programs to remember. By the end of the chapter you will be able to explain what a file, a stream and a `FILE *` are, and choose the right mode when opening a file; write and read text files with `fprintf`, `fscanf` and `fgets`, and write the reading loop that stops at exactly the right moment; read a CSV data file line by line, skip damaged lines with a clear message and report on the rest; add to a log file without ever overwriting it; store fixed-size records in a binary file and jump straight to any one of them with `fseek`; save data so that a power cut at any moment leaves either the old or the new file complete, never a broken one; and split a program into several source files with headers. The chapter ends with cashier v0.9, which saves the laundry's orders after every change and finds them all again after the power comes back.

11.1 Where We Left Off: Four O'Clock, and the Power Goes Off

Cashier v0.8 of Chapter 10 does everything Mbak Rina asked for: tidy names, orders that move from *waiting* to *washing* to *ready*, and a list that grows with the day. By four o'clock on a busy Saturday it holds thirty orders. Then the power goes off. Power cuts are an everyday event in many parts of Indonesia, sometimes announced in advance for maintenance, sometimes not. The laptop's battery died last year, the screen goes black, and when the power returns ten minutes later and the cashier is started again, it greets Mbak Rina with an empty list. Thirty orders, thirty customers who will come back with a receipt number, and the only record is in the memory of a program that no longer runs.

The reason is physical. The working memory of a computer, its RAM, keeps its contents only while it has power, and a program's variables, arrays and linked lists all live there. Data that must outlive the program must be written to *storage*: a solid-state drive, a hard disk or a memory card, which keep their contents without power. Programs use storage through **files**: named sequences of bytes that the operating system keeps for them.

Writing to a file is easy; the second half of this chapter shows that writing to a file *safely* takes a little more thought. If the cashier writes its orders to a file only when Mbak Rina types `quit`, a power cut in the afternoon still loses the whole day. If it rewrites the file after every order, a power cut *during* the rewrite can leave a file that is half old and half new, or empty. The goal of the chapter is a cashier for which neither can happen.

11.2 The Idea in Plain Words: Files and Streams

11.2.1 Streams and the `FILE` pointer

C does not let a program touch the disk directly. It talks to files through **streams**: a stream is a sequence of bytes flowing into the program (input) or out of it (output), and the program reads or

writes it from front to back, one piece after another, like water from a hose. You have used three streams since Chapter 1 without knowing their names. Every C program starts with `stdin`, the standard input (normally the keyboard), `stdout`, the standard output (normally the screen), and `stderr`, the standard error stream, also shown on the screen but kept separate so that error messages are not mixed into output that is saved or piped to another program. `printf(...)` is simply `fprintf(stdout, ...)`, and `scanf(...)` is `fscanf(stdin, ...)` [1], [2].

To use a file, a program **opens** it with `fopen(name, mode)`, which connects a new stream to the file and returns a pointer to an object of type `FILE`. The program never looks inside the `FILE` object; it only passes the pointer to the input and output functions, the way you show a ticket without reading the fine print. If the file cannot be opened, because it does not exist, because the folder is wrong, or because the user may not read it, `fopen` returns `NULL`, and a program that goes on to use `NULL` as a stream crashes. Checking the result of every `fopen` is therefore not optional.

When the program is done with a file, it **closes** it with `fclose(fp)`. Closing matters more than it seems, because streams are *buffered*: to avoid a slow trip to the disk for every character, `fprintf` collects output in a buffer in memory and writes it to the file in large blocks. The last block reaches the file only when the buffer is full, when the program calls `fflush(fp)`, or when the stream is closed. A program that crashes before closing its file may leave the last part of its output unwritten.

> **DEFINITION.** A *file* is a named sequence of bytes kept by the operating system on storage. A *stream* is the program's connection to a file (or to the keyboard or screen), read or written from front to back. In C a stream is represented by a pointer of type `FILE *`, obtained from `fopen` and released with `fclose`. A *text* stream carries lines of characters; a *binary* stream carries bytes exactly as they are.

11.2.2 Modes

The second argument of `fopen` says what the program intends to do with the file. **Table 11.1** lists the modes. The letter `b` may be added to any of them (`"rb"`, `"wb"`, `"r+b"`) to open the file in binary mode, which matters on Windows: there, a text stream turns every `'\n'` the program writes into the two bytes `"\r\n"`, and back again when reading, while a binary stream passes every byte unchanged. On Linux and macOS text and binary mode behave the same.

Table 11.1 The modes of `fopen`. Adding `b` gives the same mode for a binary stream.

Mode	Purpose	If the file exists	If it does not exist
`"r"`	read	read from the start	`fopen` returns `NULL`
`"w"`	write	**emptied at once**	created
`"a"`	append	every write goes to the end	created
`"r+"`	read and write	kept; start at the beginning	`fopen` returns `NULL`
`"w+"`	write and read	**emptied at once**	created
`"a+"`	read and append	kept; writes go to the end	created

The dangerous mode is `"w"`: it empties an existing file *at the moment of opening*, before a single byte has been written. Open the day's order file with `"w"` by mistake instead of `"r"`, and the orders are gone. Section 11.8 turns this danger into a design rule.

Python's built-in `open(name, mode)` uses the same letters, raises an exception such as `FileNotFoundError` instead of returning `NULL`, and is normally used in a `with` statement, which closes the file automatically at the end of the block, even when an error happens inside it [3].

11.3 Writing and Reading a Text File

Listing 11.1 goes through the complete life of a text file: open it for writing, write three orders with `fprintf`, close it, open it again for reading, read the orders back one by one, and close it again.

Listing 11.1 `write_read_text.c`: writing three orders to a text file and reading them back.

```c
/* write_read_text.c - the life of a text file: open, write, close, open
 * again, read until a read fails, close. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp = fopen("orders.txt", "w");          /* "w" empties the file! */
    if (fp == NULL) {
        perror("orders.txt");
        return EXIT_FAILURE;
    }
    fprintf(fp, "%d %ld %s\n", 1, 24500L, "Andi Pratama");
    fprintf(fp, "%d %ld %s\n", 2, 20000L, "Budi Santoso");
    fprintf(fp, "%d %ld %s\n", 3, 24000L, "Citra Dewi Lestari");
    fclose(fp);                    /* only now is everything on the disk */

    fp = fopen("orders.txt", "r");
    if (fp == NULL) {
        perror("orders.txt");
        return EXIT_FAILURE;
    }
    int id, n = 0;
    long amount, total = 0;
    char name[32];
    printf("%-4s %-20s %8s\n", "id", "customer", "amount");
    /* the name is last on the line, so %31[^\n] may read its spaces */
    while (fscanf(fp, "%d %ld %31[^\n]", &id, &amount, name) == 3) {
        printf("%-4d %-20s %8ld\n", id, name, amount);
        total += amount;
        n++;
    }
    fclose(fp);
    printf("%d orders, total %ld\n", n, total);
    return 0;
}
```

```
id   customer              amount
1    Andi Pratama           24500
2    Budi Santoso           20000
3    Citra Dewi Lestari     24000
3 orders, total 68500
--- orders.txt ---
1 24500 Andi Pratama
2 20000 Budi Santoso
3 24000 Citra Dewi Lestari
```

The line after `--- orders.txt ---` is the file itself, as the test runner shows it: three lines of plain text, one per order, which any editor can open. That is the great strength of text files: people can read them, check them and even repair them.

Two details of the reading loop are the heart of this section.

First, the format `"%d %ld %31[^\n]"`. The number and the amount are read with `%d` and `%ld`. The name contains spaces, so `%s` would read only `Andi`; instead the *scan set* `%[^\n]` reads every character up to the end of the line, and the width 31 stops it from writing more than 31 characters

into the 32-byte array `name` (Chapter 9). This works because the name is the *last* field on the line; a field that may contain spaces is always easiest to read when nothing comes after it.

Second, the loop condition. `fscanf` returns the number of fields it managed to read and store: 3 when a whole order was read, fewer when the line was damaged, and the constant `EOF` (a negative number, usually −1) when the end of the file came before anything could be read. The loop `while (fscanf(...) == 3)` therefore continues exactly as long as a complete order arrived, and stops both at the end of the file and at the first damaged line. This is the reading idiom of C: **test the result of the read itself**.

The Python version, **Listing 11.2**, reads the file with a `for` loop over the file object, which gives one line at a time and stops at the end by itself, and cuts each line with `split(" ", 2)`, which splits at the first two spaces only, so that the name keeps its spaces.

Listing 11.2 `write_read_text.py`: the same file written and read in Python.

```
# write_read_text.py - the life of a text file in Python. "with" closes the
# file for us at the end of the block, even if an error happens inside it.


def main():
    with open("orders.txt", "w", encoding="utf-8") as fp:   # "w" empties it!
        fp.write("%d %d %s\n" % (1, 24500, "Andi Pratama"))
        fp.write("%d %d %s\n" % (2, 20000, "Budi Santoso"))
        fp.write("%d %d %s\n" % (3, 24000, "Citra Dewi Lestari"))

    n = total = 0
    print("%-4s %-20s %8s" % ("id", "customer", "amount"))
    with open("orders.txt", encoding="utf-8") as fp:
        for line in fp:                          # one line at a time
            id, amount, name = line.rstrip("\n").split(" ", 2)
            print("%-4d %-20s %8d" % (int(id), name, int(amount)))
            total += int(amount)
            n += 1
    print("%d orders, total %d" % (n, total))


if __name__ == "__main__":
    main()
```

11.4 The `feof` Trap

There is a function `feof(fp)` that answers “has this stream reached the end of the file?”, and many beginners write the reading loop as `while (!feof(fp))`. It looks right. It is wrong, and **Listing 11.3** shows why.

Listing 11.3 `feof_trap.c`: the loop `while (!feof(fp))` runs once too often.

```c
/* feof_trap.c - why "while (!feof(fp))" reads the last record twice.
 * Run write_read_text first: this program reads its orders.txt. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp = fopen("orders.txt", "r");
    if (fp == NULL) {
        perror("orders.txt");
        return EXIT_FAILURE;
    }
    int id = 0, rounds = 0;
    long amount = 0, total = 0;
    char name[32] = "";
    while (!feof(fp)) {      /* WRONG: EOF is known only after a read fails */
        int got = fscanf(fp, "%d %ld %31[^\n]", &id, &amount, name);
        total += amount;             /* the failed last read left the old values */
        rounds++;
        printf("round %d: fscanf gave %2d -> #%d %s %ld\n",
               rounds, got, id, name, amount);
    }
    fclose(fp);
    printf("rounds: %d, total %ld (should be 3 and 68500)\n", rounds, total);
    return 0;
}
```

Run after `write_read_text`, which leaves `orders.txt` behind, it prints:

```
$ step 1
id   customer               amount
1    Andi Pratama            24500
2    Budi Santoso            20000
3    Citra Dewi Lestari      24000
3 orders, total 68500
$ step 2
round 1: fscanf gave  3 -> #1 Andi Pratama 24500
round 2: fscanf gave  3 -> #2 Budi Santoso 20000
round 3: fscanf gave  3 -> #3 Citra Dewi Lestari 24000
round 4: fscanf gave -1 -> #3 Citra Dewi Lestari 24000
rounds: 4, total 92500 (should be 3 and 68500)
```

The output under `step 2` is the trap. The loop ran four times for three orders, and the total is wrong by one whole order. The reason is that `feof` does not look ahead. It reports the end of the file only *after a read has tried to go past it and failed*. After the third order has been read, the stream stands at the end of the file, but no read has failed yet, so `feof` still says "not at the end". The loop runs a fourth time; this time `fscanf` finds nothing and returns −1 without changing `id`, `amount` or `name`; and the program adds the old amount again. Notice that on Ubuntu, GCC would not even compile the first version of this program with the options of this book: the C library there declares `fscanf` with an attribute that asks callers to use its result, and ignoring it produced a warning, which `-Werror` turns into an error. Using the result, as the reading idiom of Section 11.3 does, is the cure.

`feof` has one honest use: *after* a reading loop has stopped, it tells whether the loop stopped because the file ended (`feof` is true) or because of a damaged line or a read error (`feof` is false). The Python version, **Listing 11.4**, imitates the C loop to show the same effect; real Python code never has this problem, because `for line in fp` stops by itself.

Listing 11.4 `feof_trap.py`: an imitation of the C loop in Python.

```
# feof_trap.py - imitates the C loop "while (!feof(fp))": the end of the file
# is noticed only after one read has already failed.


def main():
    with open("orders.txt", encoding="utf-8") as fp:
        lines = fp.read().split("\n")      # the last piece is "" after the last \n
    id, amount, name = 0, 0, ""
    total = rounds = 0
    for piece in lines:                    # one round per read attempt
        got = -1                           # C's EOF
        if piece:                          # a successful read changes the values
            sid, samount, name = piece.split(" ", 2)
            id, amount, got = int(sid), int(samount), 3
        total += amount                    # the failed read left the old values
        rounds += 1
        print("round %d: fscanf gave %2d -> #%d %s %d"
              % (rounds, got, id, name, amount))
    print("rounds: %d, total %d (should be 3 and 68500)" % (rounds, total))


if __name__ == "__main__":
    main()
```

11.5 Reading Real Data: a CSV File Line by Line

Most data that small businesses exchange travels as **CSV**, *comma-separated values*: a text file with one record per line and the fields separated by commas, often with a first line that names the columns. Every spreadsheet program can save a sheet as CSV and open one, so CSV is how a week of orders typed into a spreadsheet by the owner reaches our program, and how our program's reports reach the owner. The format is described in RFC 4180 [4]; the simple form used here has no commas inside fields and no quotes.

Profitina Laundry's owner keeps a sheet of the week's orders and saves it as `data/week.csv`:

```
date,customer,service,kg
2026-09-21,Andi Pratama,fold,3.5
2026-09-21,Budi Santoso,wash-iron,2
2026-09-21,Citra Dewi Lestari,iron,4
2026-09-22,Dewi Anggraini,fold,5.5
2026-09-22,Eko Wibowo,dry-clean,3
2026-09-22,Fajar Nugroho,fold,2.5
2026-09-23,Gita Permata,wash-iron,1.5
2026-09-23,Hadi Susanto,iron
2026-09-23,Indah Sari,fold,6

2026-09-24,Joko Purnomo,iron,3.5
2026-09-24,Kartika Putri,fold,4,5
2026-09-25,Lina Marlina,wash-iron,2.5
2026-09-25,Made Wirawan,fold,3.x
```

A file typed by people contains mistakes, and this one has several: an unknown service, a line with a missing field, a line with an extra field, an empty line and a weight that is not a number. A good reader does not crash, does not stop at the first mistake and does not silently accept nonsense. It reports every damaged line *with its line number*, so that the owner can open the sheet and fix it, and it computes the report from the good lines.

Listing 11.5 reads the file in three steps per line: `fgets` reads the whole line (up to 255 characters), `strcspn(line, "\r\n")` finds the line ending, whether it is `"\n"` or the Windows `"\r\n"`, and a terminator is written there; then `split_csv` cuts the line at the commas. `split_csv` writes a terminator over each comma and remembers where each field starts, like `strtok`, but,

unlike `strtok`, it keeps empty fields: `a,,b` has three fields, the middle one empty. It also counts *all* fields, even more than the array can hold, so the program can tell "3 fields" from "5 fields". The file name comes from the command line, `argv[1]`, and when the file cannot be opened, `perror` prints the name together with the operating system's reason, taken from the variable `errno`, on `stderr`.

Listing 11.5 `csv_report.c`: reading a CSV file line by line, skipping damaged lines with a message, and totaling by service and by day.

```
/* csv_report.c - read a week of orders from a CSV file, skip bad lines with a
 * clear message, and print totals per service and per day.
 * Usage: csv_report <file.csv>      (the file starts with a header line) */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N_SERVICES 3
static const char *SERVICE[N_SERVICES] = {"fold", "wash-iron", "iron"};
static const int PRICE[N_SERVICES] = {7000, 10000, 6000};

/* Split line at commas IN PLACE, keeping empty fields (strtok would skip
 * them). Returns the number of fields; at most max are stored. */
static int split_csv(char *line, char *field[], int max)
{
    int n = 0;
    char *p = line;
    for (;;) {
        if (n < max)
            field[n] = p;
        n++;
        char *comma = strchr(p, ',');
        if (comma == NULL)
            break;
        *comma = '\0';
        p = comma + 1;
    }
    return n;
}

/* "3" -> 30, "3.5" -> 35; -1 if the text is not written like that. */
static int parse_tenths(const char *s)
{
    int whole = 0, digits = 0;
    for (; *s >= '0' && *s <= '9'; s++, digits++)
        whole = whole * 10 + (*s - '0');
    if (digits == 0 || digits > 3)
        return -1;
    if (*s == '\0')
        return whole * 10;
    if (s[0] == '.' && s[1] >= '0' && s[1] <= '9' && s[2] == '\0')
        return whole * 10 + (s[1] - '0');
    return -1;
}
```

Listing 11.5 (Continued) *csv_report.c: reading a CSV file line by line, skipping damaged lines with a message, and totaling by service and by day.*

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: %s <file.csv>\n", argv[0]);
        return EXIT_FAILURE;
    }
    FILE *fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror(argv[1]);                    /* e.g. "No such file or directory" */
        return EXIT_FAILURE;
    }
    char line[256], day[16] = "", *f[4];
    int lineno = 0, good = 0, count[N_SERVICES] = {0}, tenths[N_SERVICES] = {0};
    long amount[N_SERVICES] = {0}, day_amount = 0;
    int day_orders = 0;
    char report[16][48];                    /* one line per day, printed later */
    int days = 0;

    while (fgets(line, sizeof line, fp) != NULL) {
        lineno++;
        line[strcspn(line, "\r\n")] = '\0';     /* cut the line ending */
        if (lineno == 1 || line[0] == '\0')
            continue;                           /* header or empty line */
        int n = split_csv(line, f, 4);
        if (n != 4) {
            printf("line %d: expected 4 fields, found %d - skipped\n", lineno, n);
            continue;
        }
        int s = -1;
        for (int i = 0; i < N_SERVICES; i++)
            if (strcmp(f[2], SERVICE[i]) == 0)
                s = i;
        int t = parse_tenths(f[3]);
        if (s < 0 || t <= 0) {
            printf("line %d: bad %s '%s' - skipped\n", lineno,
                   s < 0 ? "service" : "weight", s < 0 ? f[2] : f[3]);
            continue;
        }
        if (strcmp(f[0], day) != 0) {           /* a new day starts */
            if (day[0] != '\0' && days < 16)
                snprintf(report[days++], sizeof report[0], "%-10s %6d %12ld",
                         day, day_orders, day_amount);
            snprintf(day, sizeof day, "%s", f[0]);
            day_orders = 0;
            day_amount = 0;
        }
```

Listing 11.5 (Continued) *csv_report.c: reading a CSV file line by line, skipping damaged lines with a message, and totaling by service and by day.*

```
        long a = (long)t * PRICE[s] / 10;
        count[s]++;
        tenths[s] += t;
        amount[s] += a;
        day_orders++;
        day_amount += a;
        good++;
    }
    fclose(fp);
    if (day[0] != '\0' && days < 16)
        snprintf(report[days++], sizeof report[0], "%-10s %6d %12ld",
                 day, day_orders, day_amount);

    printf("%d good orders in %s\n\n", good, argv[1]);
    printf("%-10s %6s %10s %12s\n", "service", "orders", "weight", "amount");
    long total = 0;
    for (int i = 0; i < N_SERVICES; i++) {
        printf("%-10s %6d %7d.%d kg %12ld\n", SERVICE[i], count[i],
               tenths[i] / 10, tenths[i] % 10, amount[i]);
        total += amount[i];
    }
    printf("%-10s %6d %10s %12ld\n\n", "all", good, "", total);
    printf("%-10s %6s %12s\n", "date", "orders", "amount");
    for (int i = 0; i < days; i++)
        printf("%s\n", report[i]);
    return 0;
}
```

`./csv_report ../data/week.csv` prints:

```
line 6: bad service 'dry-clean' - skipped
line 9: expected 4 fields, found 3 - skipped
line 13: expected 4 fields, found 5 - skipped
line 15: bad weight '3.x' - skipped
9 good orders in ../data/week.csv

service    orders     weight       amount
fold            4      17.5 kg       122500
wash-iron       3       6.0 kg        60000
iron            2       7.5 kg        45000
all             9                    227500

date       orders       amount
2026-09-21      3        68500
2026-09-22      2        56000
2026-09-23      2        57000
2026-09-24      1        21000
2026-09-25      1        25000
```

The four messages point at exactly the four damaged lines (the empty line 11 is simply skipped); line 13's extra field, `4,5`, is a weight typed with a comma as the decimal separator, the Indonesian habit, which is why it matters that the reader counts fields instead of taking the first four. The totals by day rely on the file being in date order: the program closes a day's line when the date changes, which is the classic way to report *groups* of a sorted file in a single pass. The per-day lines are collected with `snprintf` into an array of strings and printed after the service table.

And when the file does not exist, `./csv_report ../data/nope.csv` prints on `stderr` and ends with exit status 1:

```
../data/nope.csv: No such file or directory
```

That message is `perror`'s work: the operating system's own words, after the name of the file that caused them. Messages of this form, "which file, what went wrong", save hours for the person who has to fix the problem.

The Python version, **Listing 11.6**, uses `split(",")`, which keeps empty fields as C's `split_csv` does, and a dictionary to collect the days. (Python's `csv` module handles quoted fields and commas inside quotes, and is the right tool for CSV files that other programs produced [3]; here we split by hand so that both languages do the same thing.)

Listing 11.6 `csv_report.py`: the same report in Python.

```
# csv_report.py - read a week of orders from a CSV file, skip bad lines with a
# clear message, and print totals per service and per day.
# Usage: python3 csv_report.py <file.csv>
import sys

SERVICE = ["fold", "wash-iron", "iron"]
PRICE = [7000, 10000, 6000]


def parse_tenths(s):
    whole, dot, tenth = s.partition(".")
    if not (whole.isascii() and whole.isdigit() and len(whole) <= 3):
        return -1
    if not dot:
        return int(whole) * 10
    if len(tenth) == 1 and tenth.isascii() and tenth.isdigit():
        return int(whole) * 10 + int(tenth)
    return -1


def main():
    if len(sys.argv) != 2:
        print("usage: csv_report.py <file.csv>", file=sys.stderr)
        sys.exit(1)
    try:
        fp = open(sys.argv[1], encoding="utf-8")
    except OSError as e:
        print("%s: %s" % (sys.argv[1], e.strerror), file=sys.stderr)
        sys.exit(1)
    good = 0
    count, tenths, amount = [0] * 3, [0] * 3, [0] * 3
    per_day = {}                          # date -> [orders, amount], in file order
    with fp:
        for lineno, line in enumerate(fp, start=1):
            line = line.rstrip("\r\n")
            if lineno == 1 or line == "":
                continue
            f = line.split(",")           # keeps empty fields
            if len(f) != 4:
                print("line %d: expected 4 fields, found %d - skipped"
                      % (lineno, len(f)))
                continue
            s = SERVICE.index(f[2]) if f[2] in SERVICE else -1
            t = parse_tenths(f[3])
            if s < 0 or t <= 0:
                print("line %d: bad %s '%s' - skipped" % (lineno,
```

Listing 11.6 (Continued) *csv_report.py: the same report in Python.*

```
                    "service" if s < 0 else "weight", f[2] if s < 0 else f[3]))
                continue
            a = t * PRICE[s] // 10
            count[s] += 1
            tenths[s] += t
            amount[s] += a
            day = per_day.setdefault(f[0], [0, 0])
            day[0] += 1
            day[1] += a
            good += 1

    print("%d good orders in %s\n" % (good, sys.argv[1]))
    print("%-10s %6s %10s %12s" % ("service", "orders", "weight", "amount"))
    for i in range(3):
        print("%-10s %6d %7d.%d kg %12d"
              % (SERVICE[i], count[i], tenths[i] // 10, tenths[i] % 10, amount[i]))
    print("%-10s %6d %10s %12d\n" % ("all", good, "", sum(amount)))
    print("%-10s %6s %12s" % ("date", "orders", "amount"))
    for date, (n, a) in per_day.items():
        print("%-10s %6d %12d" % (date, n, a))


if __name__ == "__main__":
    main()
```

11.6 Append Mode: How Logs Grow

Mode `"a"` opens a file so that every write goes to its *end*, and the file is never emptied. That is exactly what a **log** needs: a record of events that only ever grows. Every server, every cash register and every bank keeps logs, because when something goes wrong, the first question is "what happened, in what order?".

Listing 11.7 counts the lines already in `activity.log` (a missing file simply has none), appends one numbered entry, and prints the whole log.

Listing 11.7 `activity_log.c`: appending one numbered entry to a log file on every run.

```c
/* activity_log.c - append mode: every run adds one line to activity.log and
 * nothing already in the file is ever overwritten.
 * Usage: activity_log [message words...] */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int lines = 0, c;
    FILE *fp = fopen("activity.log", "r");
    if (fp != NULL) {                       /* no file yet is fine: 0 lines */
        while ((c = fgetc(fp)) != EOF)
            if (c == '\n')
                lines++;
        fclose(fp);
    }

    fp = fopen("activity.log", "a");        /* "a": always write at the end */
    if (fp == NULL) {
        perror("activity.log");
        return EXIT_FAILURE;
    }
    fprintf(fp, "entry %d:", lines + 1);
    if (argc == 1)
        fprintf(fp, " cashier opened");
    for (int i = 1; i < argc; i++)
        fprintf(fp, " %s", argv[i]);
    fprintf(fp, "\n");
    fclose(fp);

    fp = fopen("activity.log", "r");
    if (fp == NULL) {
        perror("activity.log");
        return EXIT_FAILURE;
    }
    char line[128];
    while (fgets(line, sizeof line, fp) != NULL)
        printf("  %s", line);
    fclose(fp);
    return 0;
}
```

Run three times, the second time with a message (`./activity_log order #1 taken`), it prints:

```
$ step 1
  entry 1: cashier opened
$ step 2
  entry 1: cashier opened
  entry 2: order #1 taken
$ step 3
  entry 1: cashier opened
  entry 2: order #1 taken
  entry 3: cashier opened
```

Each run found the lines of the runs before and added one. The count is made with `fgetc`, which reads one character at a time and returns `EOF` at the end, and the loop `while ((c = fgetc(fp)) != EOF)` is the character-level form of the reading idiom: read, and test what the read returned. Note that `c` is an `int`, not a `char`: `EOF` is a value outside the range of the characters, and storing it in a `char` could make it look like a real character. Python's version, **Listing 11.8**, catches `FileNotFoundError` for the missing file.

Listing 11.8 `activity_log.py`: the same log in Python.

```python
# activity_log.py - append mode: every run adds one line to activity.log.
# Usage: python3 activity_log.py [message words...]
import sys


def main():
    try:
        with open("activity.log", encoding="utf-8") as fp:
            lines = fp.read().count("\n")
    except FileNotFoundError:            # no file yet is fine: 0 lines
        lines = 0
    message = " ".join(sys.argv[1:]) if len(sys.argv) > 1 else "cashier opened"
    with open("activity.log", "a", encoding="utf-8") as fp:   # "a": at the end
        fp.write("entry %d: %s\n" % (lines + 1, message))
    with open("activity.log", encoding="utf-8") as fp:
        for line in fp:
            print("  " + line, end="")


if __name__ == "__main__":
    main()
```

11.7 Binary Files and Random Access

A text file is read from front to back. To find the 5,000th order in a text file, a program must read the 4,999 before it, because lines have different lengths and there is no way to know where line 5,000 starts without counting the newlines. When every record has *the same size*, there is a faster way.

A **binary** file stores data as the bytes that represent it in memory, not as readable text. `fwrite(ptr, size, count, fp)` copies `count` objects of `size` bytes each from memory to the stream, and `fread(ptr, size, count, fp)` copies them back; both return how many whole objects they moved [1]. If the objects are structures of `s` bytes, record number i (counting from 0) starts at byte

$$\text{offset}(i) = i \cdot s \tag{11.1}$$

of the file, and the function `fseek(fp, offset, SEEK_SET)` moves the stream's position straight there, without reading anything on the way. This is called **random access**, "random" in the sense that any record can be reached in one step, in any order. `fseek(fp, 0, SEEK_END)` moves to the end, and `ftell(fp)` then reports the position, which is the size of the file in bytes. **Figure 11.1** shows the file of the next program.

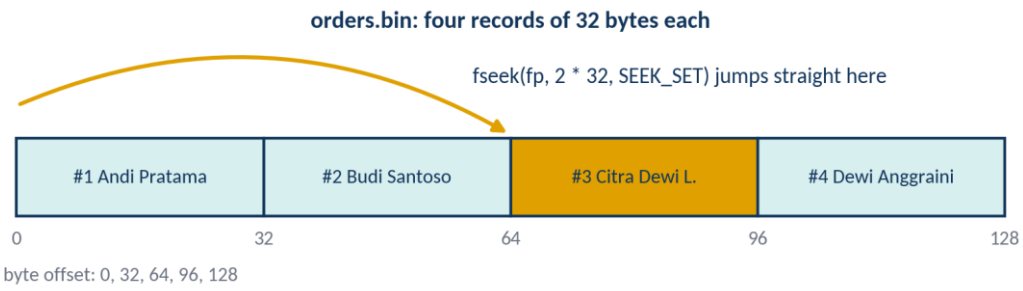


Figure 11.1 A binary file of four 32-byte records: by Equation (11.1), record 2 starts at byte $2 \cdot 32 = 64$, and `fseek` goes there directly.

Listing 11.9 writes four orders as 32-byte records, measures the file, reads record 2 directly, gives it a 10% discount and writes it back *in place*, in the middle of the file, with mode `"r+b"`, which reads and writes without emptying the file.

Listing 11.9 `binary_records.c`: fixed-size records written with `fwrite`, reached with `fseek`, and changed in place.

```c
/* binary_records.c - fixed-size records in a binary file: write them with
 * fwrite, jump to any one with fseek, change it in place, read them back. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    int id;
    char customer[20];       /* the rest of the 20 bytes is filled with '\0' */
    int tenths;
    int amount;
} Record;                    /* 4 + 20 + 4 + 4 = 32 bytes, no padding */

static Record make(int id, const char *customer, int tenths, int amount)
{
    Record r;
    memset(&r, 0, sizeof r);             /* every byte defined, even unused */
    r.id = id;
    snprintf(r.customer, sizeof r.customer, "%s", customer);
    r.tenths = tenths;
    r.amount = amount;
    return r;
}

static void show(const Record *r)
{
    printf("  #%d %-19s %2d.%d kg  Rp %6d\n", r->id, r->customer,
           r->tenths / 10, r->tenths % 10, r->amount);
}

int main(void)
{
    Record day[4] = {make(1, "Andi Pratama", 35, 24500),
                     make(2, "Budi Santoso", 20, 20000),
                     make(3, "Citra Dewi Lestari", 40, 24000),
                     make(4, "Dewi Anggraini", 55, 38500)};
    FILE *fp = fopen("orders.bin", "wb");
    if (fp == NULL) {
        perror("orders.bin");
        return EXIT_FAILURE;
    }
    size_t written = fwrite(day, sizeof(Record), 4, fp);
    fclose(fp);
    printf("wrote %zu records of %zu bytes\n", written, sizeof(Record));
```

Listing 11.9 (Continued) *binary_records.c: fixed-size records written with fwrite, reached with fseek, and changed in place.*

```
    fp = fopen("orders.bin", "r+b");         /* read AND write, no erasing */
    if (fp == NULL) {
        perror("orders.bin");
        return EXIT_FAILURE;
    }
    fseek(fp, 0, SEEK_END);
    long size = ftell(fp);
    printf("file size %ld bytes = %ld records\n",
           size, size / (long)sizeof(Record));

    Record r;
    fseek(fp, 2L * (long)sizeof(Record), SEEK_SET);   /* straight to record 3 */
    if (fread(&r, sizeof r, 1, fp) == 1) {
        printf("record at index 2:\n");
        show(&r);
    }

    r.amount -= r.amount / 10;                        /* a 10% discount ... */
    fseek(fp, 2L * (long)sizeof(Record), SEEK_SET);   /* ... written back in place */
    fwrite(&r, sizeof r, 1, fp);
    fclose(fp);

    fp = fopen("orders.bin", "rb");
    if (fp == NULL) {
        perror("orders.bin");
        return EXIT_FAILURE;
    }
    printf("all records after the change:\n");
    while (fread(&r, sizeof r, 1, fp) == 1)
        show(&r);
    fclose(fp);
    return 0;
}
```

```
wrote 4 records of 32 bytes
file size 128 bytes = 4 records
record at index 2:
  #3 Citra Dewi Lestari   4.0 kg  Rp  24000
all records after the change:
  #1 Andi Pratama         3.5 kg  Rp  24500
  #2 Budi Santoso         2.0 kg  Rp  20000
  #3 Citra Dewi Lestari   4.0 kg  Rp  21600
  #4 Dewi Anggraini       5.5 kg  Rp  38500
```

The record was chosen so that it has no padding: 4 + 20 + 4 + 4 = 32 bytes, all members aligned (Section 10.2.2). The function `make` sets every byte of the record to zero with `memset` before filling it, so that the unused bytes after a short name are zero rather than whatever happened to be in memory. That is tidy, it keeps customer data from one record from leaking into the unused bytes of another, and it makes the file's bytes predictable, which the next paragraph needs.

The Python version, **Listing 11.10**, describes the record with the `struct` module: the format `"<i20sii"` means little-endian (`<`), a 4-byte integer, 20 bytes of text, and two more integers, 32 bytes in total [3]. The test runner compares the two `orders.bin` files byte for byte, and they are identical: a C program and a Python program can share a binary file if both agree on its exact layout.

Listing 11.10 `binary_records.py`: the same records with Python's `struct` module.

```python
# binary_records.py - the same 32-byte records, packed with the struct module
# in the layout the C program uses: "<" little-endian, i int, 20s 20 bytes.
import os
import struct

RECORD = struct.Struct("<i20sii")        # 4 + 20 + 4 + 4 = 32 bytes


def make(id, customer, tenths, amount):
    return RECORD.pack(id, customer.encode("ascii")[:19], tenths, amount)


def show(raw):
    id, name, tenths, amount = RECORD.unpack(raw)
    name = name.split(b"\0", 1)[0].decode("ascii")
    print("  #%d %-19s %2d.%d kg  Rp %6d"
          % (id, name, tenths // 10, tenths % 10, amount))


def main():
    day = [make(1, "Andi Pratama", 35, 24500),
           make(2, "Budi Santoso", 20, 20000),
           make(3, "Citra Dewi Lestari", 40, 24000),
           make(4, "Dewi Anggraini", 55, 38500)]
    with open("orders.bin", "wb") as fp:
        for r in day:
            fp.write(r)
    print("wrote %d records of %d bytes" % (len(day), RECORD.size))

    with open("orders.bin", "r+b") as fp:          # read AND write, no erasing
        size = fp.seek(0, os.SEEK_END)
        print("file size %d bytes = %d records" % (size, size // RECORD.size))
        fp.seek(2 * RECORD.size)                   # straight to record 3
        raw = fp.read(RECORD.size)
        print("record at index 2:")
        show(raw)
        id, name, tenths, amount = RECORD.unpack(raw)
        amount -= amount // 10
        fp.seek(2 * RECORD.size)
        fp.write(RECORD.pack(id, name, tenths, amount))
```

Listing 11.10 (Continued) *binary_records.py: the same records with Python's struct module.*

```python
    print("all records after the change:")
    with open("orders.bin", "rb") as fp:
        while True:
            raw = fp.read(RECORD.size)
            if len(raw) < RECORD.size:
                break
            show(raw)


if __name__ == "__main__":
    main()
```

WARNING. A binary file written by `fwrite` of a structure depends on the computer that wrote it: the size of `int` and `long`, the padding inside the structure, and the *byte order* (whether the lowest byte of a number is stored first, as on the x86 and ARM processors of today's laptops and phones, or last). A file written on one platform may be nonsense on another. Use binary files for data that stays on one machine and needs speed; use text files such as CSV for data that people read or that travels between programs and machines.

11.8 Saving Without Losing: Write, Close, Rename

Back to the power cut of Section 11.1. The cashier must save its orders *after every change*, so that at most the command being typed is lost. The obvious code is: open `shop.csv` with `"w"`, write every order, close. Part (a) of **Figure 11.2** shows what goes wrong. `"w"` empties the file at once; for the next few milliseconds the file is empty and then half-written; only after `fclose` is it complete. A power cut in that window, and the file on disk holds nothing, or half the orders. Rare, but a program that saves after every command saves thousands of times a year.

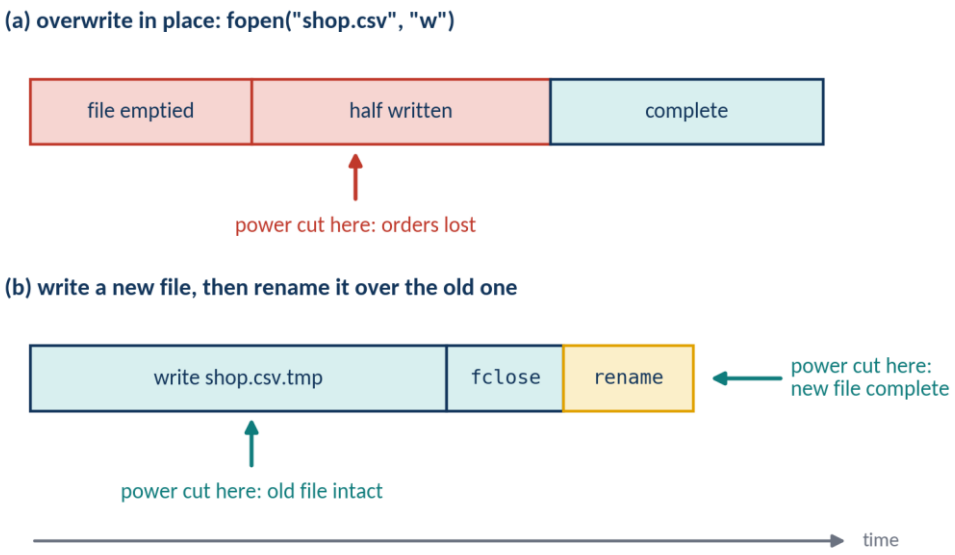


Figure 11.2 Two ways to save: (a) overwriting the file in place leaves it empty or half written during the save; (b) writing a new file and then renaming it over the old one leaves, at every moment, one complete file.

The safe method, part (b), never touches the old file until the new one is complete:

1. Write all the orders to a *new* file, `shop.csv.tmp`.
2. Close it, and check that the close succeeded: `fclose` returns non-zero if the last buffered bytes could not be written, for example because the disk is full. Also check `ferror(fp)`, which reports whether any earlier write failed.
3. Only if everything succeeded, **rename** `shop.csv.tmp` to `shop.csv`, replacing the old file.

A power cut during step 1 or 2 leaves the old `shop.csv` untouched; the half-written `.tmp` file is simply ignored and overwritten next time. The rename itself is done by the operating system in one step: on Linux and macOS, POSIX guarantees that `rename` replaces the target *atomically*, so that at every moment the name `shop.csv` refers either to the complete old file or to the complete new one [5]. Windows is less convenient: its C library's `rename` refuses to replace an existing file [6], so the portable C code removes the old file and renames again when the first rename fails, which leaves a very short window without `shop.csv` (but with a complete `shop.csv.tmp`). Python's `os.replace` performs an atomic replacement on every platform, including Windows [3].

Even this method trusts the operating system to have put the bytes on the disk. After `fclose`, the data may still sit in the operating system's own memory cache for a few seconds; programs that must survive a power cut *at any cost*, such as databases, additionally call a system function (`fsync` on POSIX, `FlushFileBuffers` on Windows) that waits until the disk confirms the write. That lies outside standard C and outside this book; for a laundry, write–close–rename is the big step.

11.9 Building Cashier v0.9

11.9.1 One program, several files

Cashier v0.8 is already over 300 lines, and v0.9 adds loading and saving. It is time to split the program into *modules*, each in its own source file with a clear job, the way Chapter 5 split the business rules into `laundry.c` and `laundry.h`:

- `shop.h` and `shop.c`: everything about orders, the text helpers of Chapter 9, the linked list and the commands of Chapter 10. `shop.c` is cashier v0.8 with `main` removed and the functions made visible to other files; it is in the code folder and is not repeated here.
- `storage.h` and `storage.c`: loading the orders from a CSV file and saving them safely.
- `cashier_v9.c`: `main`, the command loop, now calling `save_shop` after every change.

A **header** file (`.h`) holds the *declarations* that other files need: types, constants and function prototypes, but no function bodies. A source file uses it with `#include "shop.h"` (quotes, not angle brackets, for your own headers). Because one header may be included by another header and by a source file, the same declarations could be read twice, which is an error for type definitions; the **include guard** prevents it [7]: the header's body is wrapped in `#ifndef SHOP_H` / `#define SHOP_H` / `#endif`, so that the second time the preprocessor sees the header, `SHOP_H` is already defined and the body is skipped. **Listing 11.11** is the header of the shop module.

Listing 11.11 `shop.h`: the types and the functions of the shop module, protected by an include guard.

```
/* shop.h - the orders of Profitina Laundry and the commands that change
 * them (Chapter 10's cashier v0.8, moved into a module of its own). */
#ifndef SHOP_H              /* include guard: the body is read only once */
#define SHOP_H

#include <stddef.h>

#define NAME_LEN 32

typedef enum { FOLD, WASH_IRON, IRON_ONLY, N_SERVICES } Service;
typedef enum { WAITING, WASHING, READY, N_STATUS } Status;

typedef struct Order {
    int id;
    char customer[NAME_LEN];
    Service service;
    int tenths;                 /* weight in tenths of a kg */
    long amount;                /* rupiah */
    Status status;
    struct Order *next;
} Order;

typedef struct {
    Order *head, *tail;
    int count, next_id, picked;
    long takings;
} Shop;

extern const char *const SERVICE_NAME[N_SERVICES];
extern const int SERVICE_PRICE[N_SERVICES];
extern const char *const STATUS_NAME[N_STATUS];

/* text helpers (Chapter 9) */
int read_line(char *buf, int size);
size_t next_word(const char **p, char *out, size_t size);
void tidy_name(const char *src, char *dst, size_t size);
int parse_tenths(const char *s, int *out);
void rupiah(long n, char *out);
int index_of(const char *word, const char *const names[], int n);

/* the list of orders (Chapter 10) */
Order *add_order(Shop *shop, int id, Service s, int tenths, Status st,
                 const char *customer);
Order *find_id(const Shop *shop, int id);
void free_all(Shop *shop);
```

Listing 11.11 (Continued) *shop.h: the types and the functions of the shop module, protected by an include guard.*

```
/* commands; each returns 1 if it changed the shop, 0 if not */
int cmd_add(Shop *shop, const char *rest);
int cmd_advance(Shop *shop, const char *rest);
int cmd_pickup(Shop *shop, const char *rest);
void cmd_list(const Shop *shop);

#endif
```

The word `extern` in front of the three tables says “this array exists, and is defined in another file”: `shop.c` defines them once, and every file that includes `shop.h` may use them. The program is compiled by giving all its source files to `gcc` at once:

```
gcc -std=c17 -O2 -Wall -Wextra -o cashier_v9 cashier_v9.c shop.c storage.c
```

11.9.2 The storage module

The orders are saved as a CSV file with a header line:

```
id,service,tenths,status,customer
1,fold,35,washing,Andi Pratama
```

The customer comes *last*, for the same reason as in Section 11.3: if a customer's name ever contains a comma, the loader cuts the line at the first four commas only, and everything after the fourth comma is the name. **Listing 11.12** declares the two functions of the module, and **Listing 11.13** implements them.

Listing 11.12 `storage.h`: loading and saving the shop.

```
/* storage.h - keep the shop's orders in a CSV file between runs. */
#ifndef STORAGE_H
#define STORAGE_H

#include "shop.h"

/* Adds the orders stored in path to shop; returns how many were loaded.
 * A missing file is not an error: it is the very first day. */
int load_shop(Shop *shop, const char *path);

/* Writes every order to path through a temporary file; 1 on success. */
int save_shop(const Shop *shop, const char *path);

#endif
```

Listing 11.13 `storage.c`: the CSV loader, which checks every field of every line, and the saver, which writes a temporary file and renames it.

```c
/* storage.c - the shop's orders in a CSV file:
 *     id,service,tenths,status,customer
 *     1,fold,35,washing,Andi Pratama
 * The customer comes last, so a comma inside a name cannot break a line. */
#include "storage.h"

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

#define HEADER "id,service,tenths,status,customer"

static int is_number(const char *s)
{
    int n = 0;
    for (; *s >= '0' && *s <= '9'; s++)
        n++;
    return *s == '\0' && n > 0 && n <= 6;
}

int load_shop(Shop *shop, const char *path)
{
    FILE *fp = fopen(path, "r");
    if (fp == NULL)
        return 0;                           /* no file yet: the first day */
    char line[128], name[NAME_LEN];
    int lineno = 0, loaded = 0;
    while (fgets(line, sizeof line, fp) != NULL) {
        lineno++;
        line[strcspn(line, "\r\n")] = '\0';
        if (lineno == 1 && strcmp(line, HEADER) == 0)
            continue;
        char *f[5];                         /* cut at the first 4 commas */
        int n = 1;
        f[0] = line;
        while (n < 5) {
            char *comma = strchr(f[n - 1], ',');
            if (comma == NULL)
                break;
            *comma = '\0';
            f[n++] = comma + 1;
        }
        name[0] = '\0';
        int s = n == 5 ? index_of(f[1], SERVICE_NAME, N_SERVICES) : -1;
        int st = n == 5 ? index_of(f[3], STATUS_NAME, N_STATUS) : -1;
        if (n == 5)
```

Listing 11.13 (Continued) *storage.c: the CSV loader, which checks every field of every line, and the saver, which writes a temporary file and renames it.*

```
            tidy_name(f[4], name, sizeof name);
        if (n < 5 || !is_number(f[0]) || s < 0 || !is_number(f[2]) ||
            st < 0 || name[0] == '\0' || find_id(shop, atoi(f[0])) != NULL) {
            printf("  %s line %d is damaged - skipped\n", path, lineno);
            continue;
        }
        if (add_order(shop, atoi(f[0]), (Service)s, atoi(f[2]), (Status)st,
                      name) == NULL)
            break;                          /* out of memory */
        loaded++;
    }
    fclose(fp);
    return loaded;
}

int save_shop(const Shop *shop, const char *path)
{
    char tmp[256];
    snprintf(tmp, sizeof tmp, "%s.tmp", path);
    FILE *fp = fopen(tmp, "w");
    if (fp == NULL) {
        perror(tmp);
        return 0;
    }
    fprintf(fp, "%s\n", HEADER);
    for (const Order *o = shop->head; o != NULL; o = o->next)
        fprintf(fp, "%d,%s,%d,%s,%s\n", o->id, SERVICE_NAME[o->service],
                o->tenths, STATUS_NAME[o->status], o->customer);
    int bad = ferror(fp);                   /* did any write fail? */
    if (fclose(fp) != 0)                    /* the last bytes go out here */
        bad = 1;
    if (bad) {                              /* e.g. the disk is full */
        perror(tmp);
        remove(tmp);
        return 0;
    }
    if (rename(tmp, path) != 0) {           /* Windows will not rename onto */
        remove(path);                       /* an existing file: remove it */
        if (rename(tmp, path) != 0) {       /* first and try once more */
            perror(path);
            return 0;
        }
    }
    return 1;
}
```

The loader treats the file as the untrusted input it is. It accepts the header line only as line 1; it cuts each line into exactly five fields; it checks that the id and the weight are numbers of at most six digits, that the service and the status are names it knows, that the name is not empty after tidying, and that the id is not already used. A line that fails any check is reported with its line number and skipped, and the rest of the file is still loaded: one damaged line must not cost the shop its other orders. `add_order`, from `shop.c`, also raises `next_id` above the largest id loaded, so that new orders never reuse an old number.

The saver is Section 11.8 in C: write to `shop.csv.tmp`, check `ferror` and `fclose`, and rename, with the second attempt for Windows.

11.9.3 The main program

Listing 11.14 is `main`. It loads the orders at start, and after every command that reports a change (the command functions now return 1 when they changed the shop), it saves. If saving fails, it does not pretend: it tells the cashier to write the order on paper.

For the demonstration there is one extra command, `poweroff`, which simulates a power cut: it prints a message and stops the program *immediately* with `_Exit(1)`, a standard C function that ends the program without running any clean-up, without flushing any stream and without saving anything, which is as close to pulling the plug as a program can get. The message is flushed by hand first, so that we can see it.

Listing 11.14 `cashier_v9.c`: cashier v0.9, which loads the orders at start and saves after every change.

```
/* cashier_v9.c - Profitina Laundry cashier v0.9: the orders survive a power
 * cut, because every change is saved to shop.csv at once.
 * Build: gcc -std=c17 -O2 -Wall -Wextra -o cashier_v9 cashier_v9.c shop.c storage.c
 * Usage: cashier_v9 [--echo] */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "shop.h"
#include "storage.h"

#define DB "shop.csv"
#define LINE_LEN 80

int main(int argc, char *argv[])
{
    int echo = argc > 1 && strcmp(argv[1], "--echo") == 0;
    char line[LINE_LEN], cmd[16], m[32];
    Shop shop = {NULL, NULL, 0, 1, 0, 0};

    int loaded = load_shop(&shop, DB);
    printf("Profitina Laundry cashier v0.9 - %d order(s) loaded from %s\n",
           loaded, DB);
    for (;;) {
        printf("> ");
        fflush(stdout);
        int r = read_line(line, sizeof line);
        if (r == 0) {
            printf("\n");
            break;
        }
        if (echo)
            printf(r < 0 ? "%.40s...\n" : "%s\n", line);
        if (r < 0) {
            puts("  line too long (at most 79 characters), ignored");
            continue;
        }
        const char *rest = line;
        int changed = 0;
        if (next_word(&rest, cmd, sizeof cmd) == 0)
            continue;
        if (strcmp(cmd, "add") == 0)
            changed = cmd_add(&shop, rest);
        else if (strcmp(cmd, "advance") == 0)
            changed = cmd_advance(&shop, rest);
        else if (strcmp(cmd, "pickup") == 0)
```

Listing 11.14 (Continued) *cashier_v9.c: cashier v0.9, which loads the orders at start and saves after every change.*

```
            changed = cmd_pickup(&shop, rest);
        else if (strcmp(cmd, "list") == 0)
            cmd_list(&shop);
        else if (strcmp(cmd, "quit") == 0)
            break;
        else if (strcmp(cmd, "poweroff") == 0) {    /* simulate a power cut */
            puts("  *** the power goes off ***");
            fflush(stdout);
            _Exit(1);          /* stop at once: no goodbye, no clean-up */
        } else
            printf("  unknown command '%s'\n", cmd);
        if (changed && !save_shop(&shop, DB))
            puts("  WARNING: not saved - write this order on paper");
    }
    rupiah(shop.takings, m);
    printf("Bye. %d picked up (Rp %s), %d still in the shop.\n",
           shop.picked, m, shop.count);
    free_all(&shop);
    return 0;
}
```

The Python version has the same three parts: `shop.py`, cashier v0.8 without `main`; **Listing 11.15**, `storage.py`, where `split(",", 4)` cuts at the first four commas and `os.replace` does the atomic rename; and **Listing 11.16**, `cashier_v9.py`, where `os._exit(1)` plays the role of `_Exit`.

Listing 11.15 `storage.py`: loading and saving in Python.

```
# storage.py - the shop's orders in a CSV file:
#     id,service,tenths,status,customer
#     1,fold,35,washing,Andi Pratama
# The customer comes last, so a comma inside a name cannot break a line.
import os

from shop import (SERVICE_NAME, STATUS_NAME, add_order, find_id,
                  tidy_name)

HEADER = "id,service,tenths,status,customer"


def is_number(s):
    return s.isascii() and s.isdigit() and len(s) <= 6


def load_shop(shop, path):
    """Add the orders stored in path to shop; return how many were loaded."""
    try:
        fp = open(path, encoding="utf-8")
    except FileNotFoundError:
        return 0                                # no file yet: the first day
    loaded = 0
    with fp:
        for lineno, line in enumerate(fp, start=1):
            line = line.rstrip("\r\n")
            if lineno == 1 and line == HEADER:
                continue
            f = line.split(",", 4)              # cut at the first 4 commas
            ok = (len(f) == 5 and is_number(f[0]) and f[1] in SERVICE_NAME
                  and is_number(f[2]) and f[3] in STATUS_NAME
                  and tidy_name(f[4]) != "" and find_id(shop, int(f[0])) is None)
            if not ok:
                print("  %s line %d is damaged - skipped" % (path, lineno))
                continue
            add_order(shop, int(f[0]), SERVICE_NAME.index(f[1]), int(f[2]),
                      STATUS_NAME.index(f[3]), tidy_name(f[4]))
            loaded += 1
    return loaded
```

Listing 11.15 (Continued) *storage.py: loading and saving in Python.*

```
def save_shop(shop, path):
    """Write every order to path through a temporary file; True on success."""
    tmp = path + ".tmp"
    try:
        with open(tmp, "w", encoding="utf-8") as fp:
            fp.write(HEADER + "\n")
            o = shop.head
            while o is not None:
                fp.write("%d,%s,%d,%s,%s\n" % (o.id, SERVICE_NAME[o.service],
                         o.tenths, STATUS_NAME[o.status], o.customer))
                o = o.next
        os.replace(tmp, path)      # one step, also over an existing file
        return True
    except OSError as e:
        print("%s: %s" % (tmp, e.strerror))
        return False
```

Listing 11.16 `cashier_v9.py`: cashier v0.9 in Python.

```
# cashier_v9.py - Profitina Laundry cashier v0.9: the orders survive a power
# cut, because every change is saved to shop.csv at once.
# Usage: python3 cashier_v9.py [--echo]
import os
import sys

from shop import (Shop, cmd_add, cmd_advance, cmd_list, cmd_pickup,
                  read_line, rupiah)
from storage import load_shop, save_shop

DB = "shop.csv"


def main():
    echo = len(sys.argv) > 1 and sys.argv[1] == "--echo"
    shop = Shop()
    loaded = load_shop(shop, DB)
    print("Profitina Laundry cashier v0.9 - %d order(s) loaded from %s"
          % (loaded, DB))
    while True:
        print("> ", end="", flush=True)
        r, line = read_line()
        if r == 0:
            print()
            break
        if echo:
            print(line[:40] + "..." if r < 0 else line)
        if r < 0:
            print("  line too long (at most 79 characters), ignored")
            continue
        parts = line.split(None, 1)
        if not parts:
            continue
        cmd, rest = parts[0][:15], parts[1] if len(parts) > 1 else ""
        changed = False
        if cmd == "add":
            changed = cmd_add(shop, rest)
        elif cmd == "advance":
            changed = cmd_advance(shop, rest)
        elif cmd == "pickup":
            changed = cmd_pickup(shop, rest)
        elif cmd == "list":
            cmd_list(shop)
        elif cmd == "quit":
            break
        elif cmd == "poweroff":                  # simulate a power cut
```

Listing 11.16 (Continued) *cashier_v9.py: cashier v0.9 in Python.*

```
            print("  *** the power goes off ***", flush=True)
            os._exit(1)                          # stop at once, no clean-up
        else:
            print("  unknown command '%s'" % cmd)
        if changed and not save_shop(shop, DB):
            print("  WARNING: not saved - write this order on paper")
    print("Bye. %d picked up (Rp %s), %d still in the shop."
          % (shop.picked, rupiah(shop.takings), shop.count))


if __name__ == "__main__":
    main()
```

11.10 Running and Testing It

The test of v0.9 runs the program twice in the same folder, starting without any `shop.csv`. The first session takes three orders, moves two of them on, and then the power goes off; the last line of the file, typed after the power cut, is never read:

```
add fold 3.5 andi pratama
add wash-iron 2 budi santoso
add iron 4 citra dewi lestari
advance 1
advance 2
advance 2
poweroff
add fold 9 this order is never typed
```

The second session, after the power has come back, lists the orders, hands one over and takes a new one:

```
list
pickup 2
add fold 5 dewi anggraini
quit
```

The runner marks the two runs as `step 1` and `step 2`, and shows `shop.csv` as it is at the end:

```
$ step 1
Profitina Laundry cashier v0.9 - 0 order(s) loaded from shop.csv
> add fold 3.5 andi pratama
  #1 Andi Pratama, fold, 3.5 kg = Rp 24.500 (waiting)
> add wash-iron 2 budi santoso
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000 (waiting)
> add iron 4 citra dewi lestari
  #3 Citra Dewi Lestari, iron, 4.0 kg = Rp 24.000 (waiting)
> advance 1
  #1 Andi Pratama, fold, 3.5 kg = Rp 24.500 (washing)
> advance 2
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000 (washing)
> advance 2
  #2 Budi Santoso, wash-iron, 2.0 kg = Rp 20.000 (ready)
> poweroff
  *** the power goes off ***
$ step 2
Profitina Laundry cashier v0.9 - 3 order(s) loaded from shop.csv
> list
   No  Customer              Service     Weight    Amount  Status
    1  Andi Pratama          fold        3.5 kg    24.500  washing
    2  Budi Santoso          wash-iron   2.0 kg    20.000  ready
    3  Citra Dewi Lestari    iron        4.0 kg    24.000  waiting
  3 order(s) in the shop
> pickup 2
  #2 Budi Santoso picked up, paid Rp 20.000
> add fold 5 dewi anggraini
  #4 Dewi Anggraini, fold, 5.0 kg = Rp 35.000 (waiting)
> quit
Bye. 1 picked up (Rp 20.000), 3 still in the shop.
--- shop.csv ---
id,service,tenths,status,customer
1,fold,35,washing,Andi Pratama
3,iron,40,waiting,Citra Dewi Lestari
4,fold,50,waiting,Dewi Anggraini
```

This is the whole chapter in one screen. The first session was killed without a goodbye and without any save at the end, yet the second session found all three orders, *with their statuses*: Andi's washing, Budi's ready, Citra's waiting. Every change had already been saved at the moment

it was made. After the pickup and the new order, the file holds exactly the three orders in the shop, and the new order was numbered 4, after the largest id in the file.

A second test starts from a damaged `shop.csv`, `data/shop_damaged.csv`, which has a duplicate id on line 4, an unknown service on line 5 and a line without a name on line 6:

```
id,service,tenths,status,customer
1,fold,35,washing,Andi Pratama
3,iron,40,waiting,Citra Dewi Lestari
3,fold,20,ready,Budi Santoso
4,dry-clean,50,waiting,Eko Wibowo
5,wash-iron,15,ready
6,wash-iron,15,ready,Fajar Nugroho
```

```
  shop.csv line 4 is damaged - skipped
  shop.csv line 5 is damaged - skipped
  shop.csv line 6 is damaged - skipped
Profitina Laundry cashier v0.9 - 3 order(s) loaded from shop.csv
> list
   No  Customer              Service     Weight    Amount  Status
    1  Andi Pratama          fold        3.5 kg    24.500  washing
    3  Citra Dewi Lestari    iron        4.0 kg    24.000  waiting
    6  Fajar Nugroho         wash-iron   1.5 kg    15.000  ready
  3 order(s) in the shop
> quit
Bye. 0 picked up (Rp 0), 3 still in the shop.
```

Three lines were reported and skipped, three orders were loaded, and the cashier works. Mbak Rina can open the file in a text editor, look at lines 4 to 6, and decide what they should have been. As in every chapter, the runner checks all of this for C and Python, the C version under AddressSanitizer, whose leak check confirms that even the load-and-quit path frees every order.

11.11 Making It Better

Keep yesterday. The cashier keeps only the orders *in the shop*. When an order is picked up, it disappears from `shop.csv`, so there is no record of the day's business. Appending one line per pickup to a second file, `pickups.log`, in mode `"a"`, with the date, the id, the name and the amount, builds exactly that record; Exercise 11.4 asks for it, and `csv_report` of Section 11.5 can then report on it.

Keep a backup. A file on one laptop is lost with the laptop. Copying `shop.csv` and the log to a memory stick or to cloud storage every evening costs a minute.

Know when a file is not enough. A CSV file rewritten after every change is perfect for one cashier, one computer and a few hundred orders. When two cashiers must work on the same orders at once, when there are hundreds of thousands of records, or when a report must search them in many ways, the right tool is a **database**, a program whose whole job is to store records safely, to let several programs change them at once without mixing up their changes, and to find them quickly. Such systems are built from the same pieces as this chapter, files, fixed-size pages, `fseek`, logs and careful renaming, only many more of them. They are the subject of later courses.

11.12 Using It for Real

The techniques of this chapter apply to any small program that must not forget. A tailor keeps the open orders in a CSV file that the program loads at start and saves after every change; a fruit and durian orchard keeps its harvest log in append mode, one line per crate with the date, the tree block and the weight, and reports it at the end of the season with a program like `csv_report`; a

small shop keeps its price list in a CSV file that the owner edits in a spreadsheet and the cashier program loads every morning.

When you adapt the programs, follow the rules they already obey:

- **Check every `fopen`**, and report failures with the file's name and `perror`.
- **Read with the result of the read**, never with `feof`.
- **Treat every file as input from a stranger.** Check every field, report damaged lines with their number, and keep going with the good ones.
- **Save to a temporary file and rename**, never by overwriting the only copy.
- **Prefer text** for files that people read or that other programs exchange, and state the layout, as the header line of `shop.csv` does.
- **Mind personal data.** A file with names and phone numbers is personal data under Indonesia's Personal Data Protection Law (UU PDP, Law 27 of 2022); keep it on a computer with a password, and delete what the business no longer needs.

11.13 Common Mistakes, and How to Avoid Them

- **Not checking `fopen`.** Every `fopen` can fail. Test for `NULL` and report the name and the reason with `perror`.
- **Opening with `"w"` when you meant `"r"` or `"a"`.** `"w"` empties the file immediately. Double-check the mode of every `fopen` that touches valuable data, and save through a temporary file.
- **Looping with `while (!feof(fp))`.** It runs once too often (**Listing 11.3**). Loop on the result of `fscanf`, `fgets`, `fgetc` or `fread`.
- **Storing `fgetc`'s result in a `char`.** `EOF` does not fit; use an `int`.
- **Reading a name with `%s`.** It stops at the first space. Put fields with spaces last and read them with a bounded scan set such as `%31[^\n]`, or read the line with `fgets` and split it.
- **Forgetting the `'\r'` of Windows files.** A line ending `"\r\n"` read on Linux leaves a `'\r'` at the end of the last field, and `"fold\r"` is not `"fold"`. Remove both with `strcspn(line, "\r\n")`.
- **Splitting CSV with `strtok`.** It skips empty fields, so a missing value shifts every later field one place to the left. Use a splitter that keeps empty fields.
- **Forgetting `fclose`, or ignoring its result.** Buffered data reaches the file only when the stream is flushed or closed, and a full disk is reported by `fclose`.
- **Sharing binary files between machines.** Structure layout and byte order differ; use text for data that travels.
- **Writing a multi-file program without include guards.** Two inclusions of the same header redefine its types. Guard every header.

11.14 Chapter Summary

- RAM forgets when the power goes; files on storage do not. C reaches files through streams of type `FILE *`, opened with `fopen`, which returns `NULL` on failure, and closed with `fclose`, which also flushes the buffer. `stdin`, `stdout` and `stderr` are opened for every program.
- The mode decides what opening does (**Table 11.1**); `"w"` empties the file at once, `"a"` always writes at the end.
- Text files are written with `fprintf` and read with `fscanf` or `fgets`; the reading loop tests the result of each read, never `feof`.

- Real data files are read line by line, split into fields that may be empty, and checked field by field; damaged lines are reported with their number and skipped.
- Logs grow in append mode. Binary files store fixed-size records that `fseek` reaches directly (Equation (11.1), **Figure 11.1**), but they are tied to one platform.
- Safe saving writes a new file, checks `ferror` and `fclose`, and renames it over the old one, so that a power cut leaves a complete file (**Figure 11.2**).
- Larger programs are split into modules with headers, `extern` declarations and include guards.
- Cashier v0.9 loads its orders at start, saves after every change, survives a simulated power cut, and loads a damaged file without losing the good orders, in C and in Python with identical output.

11.15 Exercises

1. What is the difference between running `write_read_text` twice and running `activity_log` twice? Explain it in terms of the modes of **Table 11.1**.
2. Write a file `bad.txt` for which the loop `while (fscanf(fp, "%d %ld %31[^\n]", ...) == 3)` of **Listing 11.1** stops *before* the end of the file. How can the program tell afterwards whether it stopped at the end or at a damaged line?
3. Using Equation (11.1), give the offset of the last record of a binary file of 250 records of 32 bytes, and the size of the file. Write the `fseek` call that goes to the *last* record without knowing how many records the file has.
4. Extend cashier v0.9 so that every pickup appends one line `id,service,tenths,amount,customer` to `pickups.csv` (with a header line when the file is new). Why is mode `"a"` right for this file, although `shop.csv` is saved with write-and-rename?
5. Write a program `find_customer` in C and in Python that searches a CSV file such as `data/week.csv` for customers whose name contains a given text, ignoring case, and prints the matching lines with their line numbers, *without* loading the whole file into memory. When is this better than loading everything first, and when worse?
6. In `save_shop`, the order of the steps is: write the temporary file, close it, check, rename. What could go wrong if the program renamed first and wrote afterwards, that is, renamed the old `shop.csv` to `shop.csv.old`, then wrote a new `shop.csv` with `"w"`? Describe a power cut that loses data in that design and cannot lose data in the design of Section 11.8.

11.16 Solutions to the Exercises

1. `write_read_text` opens `orders.txt` with `"w"`, which empties the file each time, so after two runs the file holds the same three lines as after one. `activity_log` opens `activity.log` with `"a"`, which keeps the old contents and writes at the end, so every run adds one line: after two runs the file has two lines.

2. For example, a file whose second line is `2 twenty Budi`: the first order is read (3 fields), then `%ld` fails on `twenty`, `fscanf` returns 1, and the loop stops with one order although more lines follow. After the loop, `feof(fp)` is false (the end was not reached) and `ferror(fp)` is false (no read error), so the program knows a line was damaged and can report it; had the loop stopped at the end of the file, `feof(fp)` would be true.

3. The last record has index 249, so by Equation (11.1) it starts at $249 \cdot 32 = 7968$, and the file has $250 \cdot 32 = 8000$ bytes. Without knowing the count, move relative to the end: `fseek(fp, -(long)sizeof(Record), SEEK_END);`, which goes 32 bytes back from the end.

4. In `cmd_pickup`, before `free(o)`, open `pickups.csv` with `"a"`, write the line with `fprintf`, and close it; to write the header only for a new file, first try to open it with `"r"`: if that fails, the file does not exist yet. Append mode is right because a log is never rewritten, only extended: a power cut during an append can at worst lose or cut the *last* line, while every earlier line stays intact. `shop.csv`, by contrast, is replaced as a whole on every save, and a replacement needs write-and-rename.

5. The program reads one line at a time with `fgets`, finds the customer field between the first and the second comma with `strchr`, and compares with a case-insensitive search (the `contains_ci` of Chapter 9). Only one line is ever in memory, so it works on a file of any size; this is better for a single search in a large file. Loading everything first is better when the program will answer many questions about the same data, because the file is then read only once. **Listing 11.17** and **Listing 11.18** are the two versions.

Listing 11.17 `find_customer.c`: searching a CSV file line by line.

```c
/* find_customer.c - Exercise 11.5: search a CSV file line by line for a
 * customer, without loading the whole file into memory.
 * Usage: find_customer <file.csv> <part of a name> */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int contains_ci(const char *text, const char *part)
{
    size_t n = strlen(part);
    for (size_t i = 0; text[i] != '\0'; i++) {
        size_t k = 0;
        while (k < n && text[i + k] != '\0' &&
               tolower((unsigned char)text[i + k]) ==
               tolower((unsigned char)part[k]))
            k++;
        if (k == n)
            return 1;
    }
    return 0;
}

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "usage: %s <file.csv> <part of a name>\n", argv[0]);
        return EXIT_FAILURE;
    }
    FILE *fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror(argv[1]);
        return EXIT_FAILURE;
    }
    char line[256];
    int lineno = 0, hits = 0;
    while (fgets(line, sizeof line, fp) != NULL) {  /* one line in memory */
        lineno++;
        line[strcspn(line, "\r\n")] = '\0';
        char *start = strchr(line, ',');            /* customer: 2nd field */
        if (lineno == 1 || start == NULL)
            continue;
        char *end = strchr(start + 1, ',');
        if (end != NULL)
            *end = '\0';
        if (contains_ci(start + 1, argv[2])) {
```

Listing 11.17 (Continued) *find_customer.c: searching a CSV file line by line.*

```c
            if (end != NULL)
                *end = ',';                         /* put the comma back */
            printf("line %2d: %s\n", lineno, line);
            hits++;
        }
    }
    fclose(fp);
    printf("%d line(s) match '%s'\n", hits, argv[2]);
    return 0;
}
```

Listing 11.18 `find_customer.py`: the same search in Python.

```python
# find_customer.py - Exercise 11.5: search a CSV file line by line for a
# customer, without loading the whole file into memory.
# Usage: python3 find_customer.py <file.csv> <part of a name>
import sys


def main():
    if len(sys.argv) != 3:
        print("usage: find_customer.py <file.csv> <part of a name>",
              file=sys.stderr)
        sys.exit(1)
    part = sys.argv[2].lower()
    hits = 0
    with open(sys.argv[1], encoding="utf-8") as fp:
        for lineno, line in enumerate(fp, start=1):   # one line in memory
            line = line.rstrip("\r\n")
            fields = line.split(",")
            if lineno == 1 or len(fields) < 2:
                continue
            if part in fields[1].lower():
                print("line %2d: %s" % (lineno, line))
                hits += 1
    print("%d line(s) match '%s'" % (hits, sys.argv[2]))


if __name__ == "__main__":
    main()
```

`./find_customer ../data/week.csv an` prints:

```
line  2: 2026-09-21,Andi Pratama,fold,3.5
line  3: 2026-09-21,Budi Santoso,wash-iron,2
line  5: 2026-09-22,Dewi Anggraini,fold,5.5
line  9: 2026-09-23,Hadi Susanto,iron
line 15: 2026-09-25,Made Wirawan,fold,3.x
5 line(s) match 'an'
```

Note that “an” matches inside “Santoso” and “Susanto” as well as “Andi”; and that line 9, which `csv_report` rejected for its missing field, is still found, because this program only looks at the name.

6. With rename-then-write, there is a moment after the old file has been renamed away and before the new `shop.csv` is complete. If the power fails then, the next start finds a missing or half-written `shop.csv`; the complete data is in `shop.csv.old`, but the program does not look there, so the day’s orders appear lost unless someone knows to rename the file back. In the design of Section 11.8, the name `shop.csv` refers to a complete file at every moment before the rename and at every moment after it, so a power cut at any point leaves the program something complete to load.

References

[1] ISO/IEC 9899:2018, *Information technology — Programming languages — C* (C17). Geneva, Switzerland: ISO, 2018, §7.21 “Input/output <stdio.h>” and §7.22.4.5 “The _Exit function”.

[2] cppreference.com, “File input/output.” [Online]. Available: https://en.cppreference.com/w/c/io (accessed Sep. 2026).

[3] Python Software Foundation, *The Python Standard Library*, Python 3.14: “Built-in Functions: open,” “struct — Interpret bytes as packed binary data,” “csv — CSV File Reading and Writing,” and “os.replace.” [Online]. Available: https://docs.python.org/3/library/ (accessed Sep. 2026).

[4] Y. Shafranovich, “Common Format and MIME Type for Comma-Separated Values (CSV) Files,” RFC 4180, IETF, Oct. 2005. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4180

[5] The Open Group, “rename,” *The Open Group Base Specifications Issue 8* (IEEE Std 1003.1-2024). [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/rename.html (accessed Sep. 2026).

[6] Microsoft, “rename, _wrename,” *C runtime library reference*. [Online]. Available: https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/rename-wrename (accessed Sep. 2026).

[7] P. J. Deitel and H. M. Deitel, *C How to Program*, 9th ed. Hoboken, NJ, USA: Pearson, 2022, ch. 11 “File Processing” and ch. 13 “The Preprocessor.”

Chapter 12

Evaluation and Discussion III: Lists, Trees and a Knight

This chapter works three complete problems that draw on everything of Chapters 9 to 11 at once: structures that point to other structures, memory taken with `malloc` and given back with `free`, text read from files and cut into commands, and careful testing. The first problem sorts a singly linked list and then inserts at its end. The second grows a binary tree from a file of navigation commands. The third finds the fewest moves a chess knight needs between two squares, with a queue built as a linked list and the method called *breadth-first search*. Each problem is restated in our own words, understood by hand, designed on paper, programmed in C and in Python with identical output, run on prepared tests, and then compared with a tempting wrong solution that looks right and is not. By the end of the chapter you will be able to attack such a problem the same way on your own: draw first, design second, type third, and test until the program has convinced you, not the other way round.

12.1 Where We Left Off

Chapter 9 taught the program to handle text, Chapter 10 to group data into structures and to chain structures into a linked list, and Chapter 11 to read and write files. The three problems of this chapter are a step up from the cashier programs, because they are *data-structure* problems: in each, the whole difficulty lies in keeping a set of pointers right while the structure changes. They are also the first taste of a larger subject. Linked lists, trees and breadth-first search [1], [2] are the opening topics of a data-structures course, and the reader who has solved these three by hand and by program has a head start there.

The three problems are independent, and they suit a small team well: if you work with two friends, each of you can lead one problem, but all of you should read and draw all three first, and each of you should review the others' programs. A program you have only read and never questioned is not one you understand.

12.2 How to Approach a Data-Structure Problem

12.2.1 The steps

Every problem below is worked in the same order, and the order matters more than any trick.

1. **Restate the problem in your own words.** Write down what goes in, what comes out, and every decision the statement leaves open. A decision you do not write down is a decision you will make twice, differently.
2. **Work a small example by hand, with drawings.** Boxes for nodes, arrows for pointers, one drawing per step. For pointer problems this step is not a luxury: if you cannot draw what happens, you cannot program it.
3. **Design on paper.** Choose the structure and write the algorithm as pseudocode, short numbered lines without the details of C. Estimate the cost.

4. **Program it**, one function at a time, testing each function alone before combining them.
5. **Test with prepared cases**: the example of the statement, the edge cases (empty, one element, already in order, in reverse order, duplicates, the largest allowed input) and wrong input. Know the expected answer *before* you run.
6. **Run under the sanitizer** (`-fsanitize=address,undefined`), which reports every stray pointer, every use of freed memory and every leak. A pointer program that has not run under a sanitizer has not been tested.
7. **Take apart a tempting wrong solution.** Knowing *why* the obvious shortcut fails is how you avoid it next time.
8. **Check yourself with a variant.** Change the problem a little and solve it again without looking.

12.2.2 Where the code lives

The programs of the chapter are in the folder `Chapter12`, C in `c/` and Python in `python/`, with every test input in `data/` and the expected outputs in `tests/expected/`. `python3 tests/run_tests.py` compiles every C program with warnings as errors and the sanitizers, runs the C and the Python version on every test, and checks that both print exactly the expected output. All outputs in this chapter are the real outputs of those runs.

12.3 Problem 12.1: Sort the Linked List, Then Insert at the End

12.3.1 The problem

> **PROBLEM 12.1.** A singly linked list of at least five whole numbers is given, built from nodes `struct sNode { int data; struct sNode *next; };`. Sort the list in ascending order with an algorithm of your own (no library sorting function), and then insert one more number at the *end* of the sorted list. Print the list before sorting, after sorting and after the insertion.

In our words: a chain of nodes holds numbers in some order; we must reorder them from small to large, then attach a new node with a given number after the last node, even if that breaks the order. The statement leaves the input open, so we decide: the program reads the numbers of the list from the first line of standard input, and the number to insert from the second line. We also decide that the program must work for *any* list, including an empty list and a list of one number, although the statement promises at least five: programs outlive their statements. Finally we decide to print how many passes and swaps the sort needed, which costs two counters and makes the tests more telling.

12.3.2 Understanding it by hand

Our main example is the list 29, 4, 17, 4, 42, 8, 15 and the number 23. The sorted list is 4, 4, 8, 15, 17, 29, 42, and after the insertion at the end, 4, 4, 8, 15, 17, 29, 42, 23. Note the last number: “insert at the end” means *at the end*, not “in its sorted place”.

Which sort? The one we know from Chapter 6 that works best with a list is **bubble sort**, because it only ever compares and swaps *neighbors*, and in a singly linked list a node’s neighbor is exactly one `->next` away. There is also a choice about *what* to swap. We can swap the nodes, by relinking pointers, or we can leave the nodes where they are and swap the numbers inside them. Swapping the `data` fields is three assignments with an `int` temporary, exactly as in Chapter 6, and no pointer changes at all; for nodes that hold one number it is clearly the simpler choice. (Exercise 12.2 relinks instead.)

Here is the trace. Each pass walks the list from the head, comparing each node with its successor and swapping when they are out of order; the largest number still unsorted "bubbles" to the end of the unsorted part.

```
start   29  4 17  4 42  8 15
pass 1   4 17  4 29  8 15 | 42     5 swaps
pass 2   4  4 17  8 15 | 29 42     3 swaps
pass 3   4  4  8 15 | 17 29 42     2 swaps
pass 4   4  4  8 | 15 17 29 42     0 swaps -> sorted, stop
```

The bar | marks where the sorted tail begins: after pass k, the k largest numbers are in their final places, so pass $k + 1$ can stop one node earlier. And when a pass makes no swap at all, every neighbor pair is in order, so the whole list is sorted and we can stop early. Here that happens in pass 4, so the sort needs 4 passes and $5 + 3 + 2 = 10$ swaps.

Ten is not an accident. An *inversion* is a pair of positions $i < j$ whose numbers are in the wrong order, $a_i > a_j$. Each swap of two neighbors that are out of order removes exactly one inversion and creates none, and a sorted list has no inversions, so

$$\text{swaps of bubble sort} = \#\{(i, j) : i < j \text{ and } a_i > a_j\} \tag{12.1}$$

For our list: 29 is larger than the five numbers after it except 42, which is 5 inversions; the first 4 has none; 17 is larger than 4, 8 and 15, which is 3; the second 4 has none; 42 is larger than 8 and 15, which is 2; 8 has none. The total is $5 + 3 + 2 = 10$, as the trace found. Equation (12.1) gives the expected swap count of every test in advance, which is exactly what step 5 of Section 12.2 asks for.

The insertion at the end: walk from the head while the current node *has* a successor, so that the walk stops *on* the last node, not after it; link the new node there. If the list is empty there is no last node, and the new node becomes the head.

12.3.3 Designing it on paper

Pseudocode 12.1 is bubble sort on a list, with the two improvements of the trace: the pointer `last` marks the start of the sorted tail, and the flag `swapped` stops the sort after a pass without swaps.

Pseudocode 12.1 Bubble sort on a singly linked list, swapping the numbers of neighboring nodes.

```
LIST-BUBBLE-SORT(head)
1  last = NIL                        // everything from last on is sorted
2  swapped = TRUE
3  while head != NIL and swapped
4      swapped = FALSE
5      p = head
6      while p.next != last
7          if p.data > p.next.data
8              exchange p.data with p.next.data
9              swapped = TRUE
10         p = p.next
11     last = p                      // p is the last node this pass looked at
```

Line 6 is the subtle one. In the first pass, `last` is `NIL`, so the inner loop runs while `p.next` is not `NIL`, that is, until `p` is the last node; it never looks at `p.next.data` when `p.next` is `NIL`. In later passes it stops earlier, at the node before the sorted tail. Line 11 records where this pass stopped.

The cost. For n numbers, pass k makes at most $n-k$ comparisons, so all passes together make at most $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$ comparisons, which is $O(n^2)$; a list that is already sorted takes a single pass of $n-1$ comparisons. Insertion at the end walks n nodes, $O(n)$. For lists of a few thousand numbers this is instant.

12.3.4 The C solution

Listing 12.1 follows the design line by line. `insert_end` is used twice: to build the list from the input, and to insert the extra number; it handles the empty list by returning the new node as the new head, which is why it returns the head and the caller writes `head = insert_end(head, value)`. The numbers of the first line are read with `sscanf` and the conversion `%n`, which stores how many characters have been consumed so far, so that the loop can step through the line number by number.

Listing 12.1 `list_sort.c`: bubble sort on a linked list by swapping data, then insertion at the end.

```c
/* list_sort.c - Problem 12.1: sort a singly linked list in ascending order
 * without any library sort, then insert one value at the end.
 * Input (standard input): line 1 = the values, line 2 = the value to insert. */
#include <stdio.h>
#include <stdlib.h>

struct sNode {
    int data;
    struct sNode *next;
};

/* Append value after the last node; works for an empty list too. */
static struct sNode *insert_end(struct sNode *head, int value)
{
    struct sNode *n = malloc(sizeof *n);
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(EXIT_FAILURE);
    }
    n->data = value;
    n->next = NULL;
    if (head == NULL)
        return n;                       /* the new node is the whole list */
    struct sNode *p = head;
    while (p->next != NULL)             /* stop ON the last node */
        p = p->next;
    p->next = n;
    return head;
}

static void print_list(const char *label, const struct sNode *p)
{
    printf("%-15s:", label);
    for (; p != NULL; p = p->next)
        printf(" %d ->", p->data);
    printf(" NULL\n");
}
```

Listing 12.1 (Continued) *list_sort.c: bubble sort on a linked list by swapping data, then insertion at the end.*

```
/* Bubble sort on the list: swap the DATA of neighbours that are out of
 * order. After each pass the largest remaining value has sunk to the end,
 * so the pass stops one node earlier (at 'last') the next time. */
static void bubble_sort(struct sNode *head, int *passes, int *swaps)
{
    struct sNode *last = NULL;          /* everything from last on is sorted */
    int swapped = 1;
    *passes = *swaps = 0;
    while (head != NULL && swapped) {
        swapped = 0;
        struct sNode *p = head;
        while (p->next != last) {
            if (p->data > p->next->data) {
                int t = p->data;
                p->data = p->next->data;
                p->next->data = t;
                swapped = 1;
                (*swaps)++;
            }
            p = p->next;
        }
        last = p;                       /* p is the last node of this pass */
        (*passes)++;
    }
}

static void free_list(struct sNode *p)
{
    while (p != NULL) {
        struct sNode *next = p->next;
        free(p);
        p = next;
    }
}

int main(void)
{
    struct sNode *head = NULL;
    char line[512];
    int value, used, extra, passes, swaps;
```

Listing 12.1 (Continued) *list_sort.c: bubble sort on a linked list by swapping data, then insertion at the end.*

```
    if (fgets(line, sizeof line, stdin) == NULL)
        return EXIT_FAILURE;
    for (const char *s = line; sscanf(s, "%d%n", &value, &used) == 1; s += used)
        head = insert_end(head, value);
    if (scanf("%d", &extra) != 1) {
        fprintf(stderr, "line 2 must hold the value to insert\n");
        free_list(head);
        return EXIT_FAILURE;
    }
    print_list("before sorting", head);
    bubble_sort(head, &passes, &swaps);
    print_list("after sorting", head);
    head = insert_end(head, extra);
    print_list("after insert", head);
    printf("passes: %d, swaps: %d\n", passes, swaps);
    free_list(head);
    return 0;
}
```

12.3.5 The Python solution

Listing 12.2 builds the same chain of `sNode` objects, not a Python list, and swaps with Python's tuple assignment `p.data, p.next.data = p.next.data, p.data`. The comparison `p.next is not last` uses `is`, "the same object", which is the Python counterpart of comparing two C pointers.

Listing 12.2 `list_sort.py`: the same solution in Python.

```
# list_sort.py - Problem 12.1: sort a singly linked list in ascending order
# without any library sort, then insert one value at the end.
# Input (standard input): line 1 = the values, line 2 = the value to insert.
import sys


class sNode:
    def __init__(self, data):
        self.data = data
        self.next = None


def insert_end(head, value):
    n = sNode(value)
    if head is None:
        return n
    p = head
    while p.next is not None:          # stop ON the last node
        p = p.next
    p.next = n
    return head


def print_list(label, p):
    s = "%-15s:" % label
    while p is not None:
        s += " %d ->" % p.data
        p = p.next
    print(s + " NULL")


def bubble_sort(head):
    """Swap the data of neighbours that are out of order; return counts."""
    last, swapped, passes, swaps = None, True, 0, 0
    while head is not None and swapped:
        swapped = False
        p = head
        while p.next is not last:
            if p.data > p.next.data:
                p.data, p.next.data = p.next.data, p.data
                swapped = True
                swaps += 1
            p = p.next
        last = p
        passes += 1
    return passes, swaps
```

Listing 12.2 (Continued) *list_sort.py: the same solution in Python.*

```
def main():
    lines = sys.stdin.read().split("\n", 1)
    head = None
    for word in lines[0].split():
        head = insert_end(head, int(word))
    rest = lines[1].split() if len(lines) > 1 else []
    if not rest:
        print("line 2 must hold the value to insert", file=sys.stderr)
        sys.exit(1)
    print_list("before sorting", head)
    passes, swaps = bubble_sort(head)
    print_list("after sorting", head)
    head = insert_end(head, int(rest[0]))
    print_list("after insert", head)
    print("passes: %d, swaps: %d" % (passes, swaps))


if __name__ == "__main__":
    main()
```

12.3.6 Running and testing it

The main example, `data/list1.txt`:

```
29 4 17 4 42 8 15
23
```

`./list_sort < ../data/list1.txt` prints:

```
before sorting : 29 -> 4 -> 17 -> 4 -> 42 -> 8 -> 15 -> NULL
after sorting  : 4 -> 4 -> 8 -> 15 -> 17 -> 29 -> 42 -> NULL
after insert   : 4 -> 4 -> 8 -> 15 -> 17 -> 29 -> 42 -> 23 -> NULL
passes: 4, swaps: 10
```

Four passes and ten swaps, as the hand trace and Equation (12.1) predicted. The four edge tests give:

```
before sorting : 1 -> 2 -> 3 -> 4 -> 5 -> NULL
after sorting  : 1 -> 2 -> 3 -> 4 -> 5 -> NULL
after insert   : 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL
passes: 1, swaps: 0
before sorting : 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> NULL
after sorting  : 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> NULL
after insert   : 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 0 -> NULL
passes: 6, swaps: 15
before sorting : 7 -> NULL
after sorting  : 7 -> NULL
after insert   : 7 -> 3 -> NULL
passes: 1, swaps: 0
before sorting : NULL
after sorting  : NULL
after insert   : 5 -> NULL
passes: 0, swaps: 0
```

A sorted list takes one pass and no swaps. The reversed list 9, 8, 7, 6, 5, 4 has every pair inverted, $\frac{6 \cdot 5}{2} = 15$ inversions, and takes 15 swaps in 6 passes: the worst case. A single number takes one pass, and inserting 3 after 7 shows again that the insertion ignores order. The empty list prints `NULL` twice, takes no pass, and the insertion creates the first node. All five run clean under the sanitizer, which also confirms that `free_list` returned every node.

12.3.7 A tempting wrong solution

A popular shortcut makes *one* pass of neighbor swaps and stops. **Listing 12.3** tries it on two lists, and **Listing 12.4** does the same in Python.

Listing 12.3 `wrong_list_sort.c`: a single pass of neighbor swaps, which is not a sort.

```c
/* wrong_list_sort.c - a tempting wrong solution to Problem 12.1: ONE pass of
 * neighbour swaps. It moves the largest value to the end, and on a nearly
 * sorted list that is enough to look right. */
#include <stdio.h>
#include <stdlib.h>

struct sNode {
    int data;
    struct sNode *next;
};

static struct sNode *push_front(struct sNode *head, int value)
{
    struct sNode *n = malloc(sizeof *n);
    if (n == NULL)
        exit(EXIT_FAILURE);
    n->data = value;
    n->next = head;
    return n;
}

static void print_list(const char *label, const struct sNode *p)
{
    printf("%-9s:", label);
    for (; p != NULL; p = p->next)
        printf(" %d", p->data);
    printf("\n");
}

static void one_pass_sort(struct sNode *head)
{
    for (struct sNode *p = head; p != NULL && p->next != NULL; p = p->next)
        if (p->data > p->next->data) {
            int t = p->data;
            p->data = p->next->data;
            p->next->data = t;
        }
}
```

Listing 12.3 (Continued) *wrong_list_sort.c: a single pass of neighbor swaps, which is not a sort.*

```
int main(void)
{
    const int tests[2][5] = {{1, 2, 4, 3, 5}, {5, 4, 3, 2, 1}};
    for (int t = 0; t < 2; t++) {
        struct sNode *head = NULL;
        for (int i = 4; i >= 0; i--)
            head = push_front(head, tests[t][i]);
        print_list("input", head);
        one_pass_sort(head);
        print_list("one pass", head);
        while (head != NULL) {
            struct sNode *next = head->next;
            free(head);
            head = next;
        }
    }
    return 0;
}
```

Listing 12.4 `wrong_list_sort.py`: the same mistake in Python.

```
# wrong_list_sort.py - a tempting wrong solution to Problem 12.1: ONE pass of
# neighbour swaps, which only moves the largest value to the end.


class sNode:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next


def print_list(label, p):
    s = "%-9s:" % label
    while p is not None:
        s += " %d" % p.data
        p = p.next
    print(s)


def one_pass_sort(head):
    p = head
    while p is not None and p.next is not None:
        if p.data > p.next.data:
            p.data, p.next.data = p.next.data, p.data
        p = p.next


def main():
    for test in ([1, 2, 4, 3, 5], [5, 4, 3, 2, 1]):
        head = None
        for v in reversed(test):
            head = sNode(v, head)
        print_list("input", head)
        one_pass_sort(head)
        print_list("one pass", head)


if __name__ == "__main__":
    main()
```

```
input    : 1 2 4 3 5
one pass : 1 2 3 4 5
input    : 5 4 3 2 1
one pass : 4 3 2 1 5
```

On the nearly sorted list 1, 2, 4, 3, 5 the one pass happens to be enough, and a programmer who tests only that list believes the program is correct. On the reversed list, one pass carries the 5 to the end and leaves the rest reversed. The general lesson: one pass of bubble sort guarantees only that the *largest* number reaches the end; a list with k inversions needs passes until they are all gone. Test with a reversed list, always.

A second classic mistake is in the insertion: `while (p->next != NULL)` with `p = head` crashes on an empty list, because `head` is `NULL` and `p->next` reads through a null pointer. The test `list_sort_empty` exists to catch exactly that; the sanitizer turns the crash into a clear report of the line.

12.3.8 Check yourself

Sort the list in *descending* order and insert the new number at the *front*. Which single character of **Pseudocode 12.1** changes, and which function replaces `insert_end`? (Answer: the `>` in line 7 becomes `<`; the new function is the two-line `push_front` of Chapter 10.) Then predict the swap count for 29, 4, 17, 4, 42, 8, 15 in descending order: it is the number of pairs $i < j$ with $a_i < a_j$, which is $\frac{7 \cdot 6}{2} = 21$ pairs minus the 10 inversions minus the 1 pair of equal numbers, that is, 10.

12.4 Problem 12.2: A Tree Grown from a File

12.4.1 The problem

> **PROBLEM 12.2.** A text file describes a binary tree. It holds pairs of lines: a command line, then a data line with a whole number. The command `0` creates the root. The command `l` inserts a new node as the *left* child of the current node. A command of the form `u…ur`, some number of letters `u` followed by one `r`, first climbs from the current node one level toward the root for every `u`, and then inserts the new node as the *right* child of the node reached. The node just inserted always becomes the current node. Build the tree from the file and show that it has the right shape.

In our words: the file is a recipe for drawing a tree, like instructions for walking through a building: “start here; go down-left and put a box; go up one floor, then down-right and put a box”. We must follow the recipe and keep track of *where we are*. We decide: the file name is given on the command line; empty lines are ignored; a plain `r` (no `u`) means “right child of the current node”; and the program stops with a clear message, naming the line, if a command is unknown, if a data line is missing or not a number, if the first command is not `0` or a second `0` appears, if the climb would go above the root, or if the place where the new node should go is already taken. To show the shape, we print the tree in three standard orders and also sideways.

A **binary tree** is a structure of nodes in which every node has at most two children, a *left* and a *right* one, and every node except the root has exactly one *parent*. It is the natural next step after the linked list: a list node has one `next`, a tree node has two.

12.4.2 Understanding it by hand

The example file `data/tree5.txt`, one item per line:

```
0
50
l
30
l
20
ur
40
uur
70
```

Figure 12.1 draws the tree after each pair of lines. Line 1 creates the root 50, which becomes current. Line 3, `l`: 30 becomes the left child of 50 and is now current. Line 5, `l`: 20 becomes the left child of 30, current. Line 7, `ur`: from the current node 20, climb *one* level, to 30; 40 becomes the right child of 30, current. Line 9, `uur`: from the current node 40, climb one level to 30 and a second level to 50; 70 becomes the right child of 50.

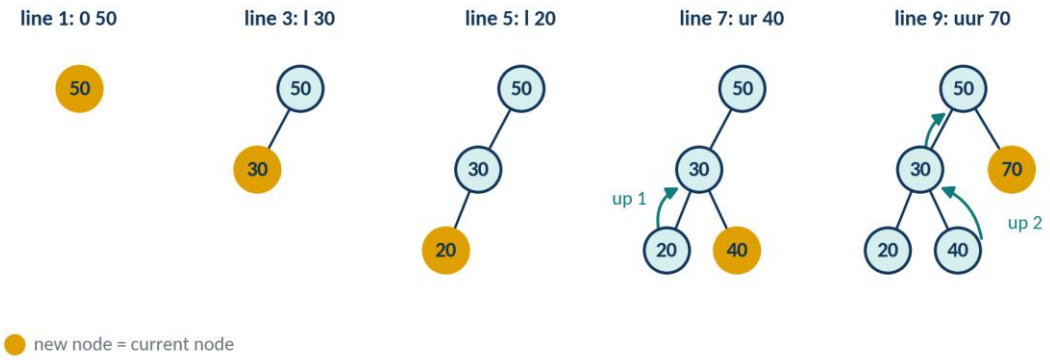


Figure 12.1 Growing the tree of `tree5.txt`: after each command the new node (amber) becomes the current node, and a command `u...ur` first climbs one level per `u` from it.

The key observation is that the climb starts from the *current* node, the one just inserted, and that climbing needs a way up. A tree node normally knows only its children. So each node also stores a pointer to its **parent**, set when the node is created, and "climb one level" is simply `at = at->parent`. The root's parent is `NULL`, which is how a climb above the root is detected.

To *show* the shape, we print the tree in the three classic orders, all defined recursively (Chapter 5): **preorder** visits a node, then its left subtree, then its right subtree; **inorder** visits the left subtree, the node, then the right subtree; **postorder** visits both subtrees and then the node. For our tree the expected results are: preorder 50 30 20 40 70, inorder 20 30 40 50 70, postorder 20 40 30 70 50. Two different trees can have the same inorder, but no two different trees have the same preorder *and* inorder, so printing both proves the shape. The sideways print lies the tree on its left side: right subtree above, left subtree below, indented by depth, so that you can turn the page and see the tree.

12.4.3 Designing it on paper

The node is `struct TNode { int data; struct TNode *left, *right, *parent; }`. **Pseudocode 12.2** builds the tree; `cur` is the current node.

Pseudocode 12.2 Building a binary tree from navigation commands, with a parent pointer in every node.

```
BUILD-TREE(file)
1  root = NIL, cur = NIL
2  for each command line cmd, followed by a data line with number x
3      if cmd = "0"
4          error if root != NIL
5          root = cur = NEW-NODE(x, parent = NIL)
6      else error if root = NIL
7      else if cmd = "l"
8          error if cur.left != NIL
9          cur.left = NEW-NODE(x, parent = cur);  cur = cur.left
10     else if cmd = u^k r   (k >= 0 letters u, then r)
11         at = cur
12         repeat k times: error if at.parent = NIL; at = at.parent
13         error if at.right != NIL
14         at.right = NEW-NODE(x, parent = at);  cur = at.right
15     else error "unknown command"
16 return root
```

The cost. Each node is created once; a climb of k levels costs k steps, and k can never exceed the depth of the tree. The traversals visit each node once, $O(n)$. Freeing the tree is a postorder traversal: both subtrees first, then the node, so that no node is freed while its children still need to be reached through it.

12.4.4 The C solution

Listing 12.5 implements **Pseudocode 12.2** in `build`, which returns the number of nodes created, or -1 after printing an error. `next_line` reads the next non-empty line and counts line numbers, so every message can say where the problem is. `climbs` checks that a command has the form `u…ur` and returns the number of `u`s. The data line is checked with `sscanf(text, "%d %c", &data, &end) == 1`: exactly one conversion must succeed, so a number followed by anything else (the `%c` would then match) is refused, and so is a line with no number at all.

Listing 12.5 `tree_from_file.c`: growing a binary tree from a command file, printing it in three orders and sideways, and freeing it.

```c
/* tree_from_file.c - Problem 12.2: grow a binary tree from a file of
 * command lines and data lines, then print it.
 *   0      create the root (first command only)
 *   l      the new node becomes the LEFT child of the current node
 *   u...ur climb one parent per u from the current node, then the new node
 *          becomes the RIGHT child of the node reached (plain r: no climb)
 * The node just created becomes the current node.
 * Usage: tree_from_file <file> */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct TNode {
    int data;
    struct TNode *left, *right, *parent;
} TNode;

static TNode *new_node(int data, TNode *parent)
{
    TNode *n = malloc(sizeof *n);
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(EXIT_FAILURE);
    }
    n->data = data;
    n->left = n->right = NULL;
    n->parent = parent;
    return n;
}

static void free_tree(TNode *t)
{
    if (t == NULL)
        return;
    free_tree(t->left);
    free_tree(t->right);
    free(t);
}
```

Listing 12.5 (Continued) *tree_from_file.c: growing a binary tree from a command file, printing it in three orders and sideways, and freeing it.*

```c
/* Read the next non-empty line, without its line ending; 0 at the end. */
static int next_line(FILE *fp, char *buf, int size, int *lineno)
{
    while (fgets(buf, size, fp) != NULL) {
        (*lineno)++;
        buf[strcspn(buf, "\r\n")] = '\0';
        if (buf[0] != '\0')
            return 1;
    }
    return 0;
}

/* How many u's before the final r? -1 if cmd is not of the form u...ur. */
static int climbs(const char *cmd)
{
    size_t n = strlen(cmd);
    if (n == 0 || cmd[n - 1] != 'r')
        return -1;
    for (size_t i = 0; i + 1 < n; i++)
        if (cmd[i] != 'u')
            return -1;
    return (int)n - 1;
}

/* Returns the number of nodes created, or -1 after reporting an error. */
static int build(FILE *fp, TNode **root)
{
    char cmd[64], text[64];
    int lineno = 0, count = 0;
    TNode *cur = NULL;
    while (next_line(fp, cmd, sizeof cmd, &lineno)) {
        int cmd_line = lineno, data;
        char end;
        if (!next_line(fp, text, sizeof text, &lineno) ||
            sscanf(text, "%d %c", &data, &end) != 1) {
            printf("line %2d: a whole number must follow command '%s'\n",
                   cmd_line + 1, cmd);
            return -1;
        }
        if (strcmp(cmd, "0") == 0) {
            if (*root != NULL) {
                printf("line %2d: the tree already has a root\n", cmd_line);
                return -1;
            }
            cur = *root = new_node(data, NULL);
            printf("line %2d: %-5s-> %d is the root\n", cmd_line, cmd, data);
```

Listing 12.5 (Continued) *tree_from_file.c: growing a binary tree from a command file, printing it in three orders and sideways, and freeing it.*

```
        } else if (*root == NULL) {
            printf("line %2d: the first command must be 0\n", cmd_line);
            return -1;
        } else if (strcmp(cmd, "l") == 0) {
            if (cur->left != NULL) {
                printf("line %2d: %d already has a left child\n", cmd_line,
                       cur->data);
                return -1;
            }
            cur->left = new_node(data, cur);
            printf("line %2d: %-5s-> %d is the left child of %d\n",
                   cmd_line, cmd, data, cur->data);
            cur = cur->left;
        } else {
            int k = climbs(cmd);
            if (k < 0) {
                printf("line %2d: unknown command '%s'\n", cmd_line, cmd);
                return -1;
            }
            TNode *at = cur;
            for (int i = 0; i < k; i++) {
                if (at->parent == NULL) {
                    printf("line %2d: '%s' climbs above the root\n", cmd_line, cmd);
                    return -1;
                }
                at = at->parent;
            }
            if (at->right != NULL) {
                printf("line %2d: %d already has a right child\n", cmd_line,
                       at->data);
                return -1;
            }
            at->right = new_node(data, at);
            printf("line %2d: %-5s-> up %d to %d, %d is its right child\n",
                   cmd_line, cmd, k, at->data, data);
            cur = at->right;
        }
        count++;
    }
    return count;
}
```

Listing 12.5 (Continued) *tree_from_file.c: growing a binary tree from a command file, printing it in three orders and sideways, and freeing it.*

```c
static void preorder(const TNode *t)
{
    if (t == NULL)
        return;
    printf(" %d", t->data);
    preorder(t->left);
    preorder(t->right);
}

static void inorder(const TNode *t)
{
    if (t == NULL)
        return;
    inorder(t->left);
    printf(" %d", t->data);
    inorder(t->right);
}

static void postorder(const TNode *t)
{
    if (t == NULL)
        return;
    postorder(t->left);
    postorder(t->right);
    printf(" %d", t->data);
}

static int height(const TNode *t)
{
    if (t == NULL)
        return 0;
    int a = height(t->left), b = height(t->right);
    return 1 + (a > b ? a : b);
}

/* The tree lying on its side: right subtree above, left subtree below. */
static void sideways(const TNode *t, int depth)
{
    if (t == NULL)
        return;
    sideways(t->right, depth + 1);
    printf("%*s%d\n", 6 * depth, "", t->data);
    sideways(t->left, depth + 1);
}
```

Listing 12.5 (Continued) *tree_from_file.c: growing a binary tree from a command file, printing it in three orders and sideways, and freeing it.*

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        return EXIT_FAILURE;
    }
    FILE *fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror(argv[1]);
        return EXIT_FAILURE;
    }
    TNode *root = NULL;
    int n = build(fp, &root);
    fclose(fp);
    if (n <= 0) {                   /* an error was reported, or no nodes */
        if (n == 0)
            printf("%s holds no nodes\n", argv[1]);
        free_tree(root);
        return EXIT_FAILURE;
    }
    printf("nodes: %d, height: %d\n", n, height(root));
    printf("preorder :");
    preorder(root);
    printf("\ninorder  :");
    inorder(root);
    printf("\npostorder:");
    postorder(root);
    printf("\nsideways (root on the left, right children above):\n");
    sideways(root, 0);
    free_tree(root);
    return 0;
}
```

12.4.5 The Python solution

Listing 12.6 uses a regular expression for the command (`u*r` means “any number of `u`, then `r`”, from Python’s `re` module [3]) and another for the data line. The traversals collect the values into a list, which is then joined for printing.

Listing 12.6 `tree_from_file.py`: the same tree builder in Python.

```
# tree_from_file.py - Problem 12.2: grow a binary tree from a file of command
# lines and data lines, then print it.
#   0      create the root (first command only)
#   l      the new node becomes the LEFT child of the current node
#   u...ur climb one parent per u from the current node, then the new node
#          becomes the RIGHT child of the node reached (plain r: no climb)
# Usage: python3 tree_from_file.py <file>
import re
import sys


class TNode:
    def __init__(self, data, parent):
        self.data = data
        self.left = self.right = None
        self.parent = parent


def lines_of(path):
    """Yield (line number, text) for every non-empty line."""
    with open(path, encoding="utf-8") as fp:
        for lineno, line in enumerate(fp, start=1):
            line = line.rstrip("\r\n")
            if line:
                yield lineno, line


def climbs(cmd):
    """How many u's before the final r? -1 if cmd is not u...ur."""
    return len(cmd) - 1 if re.fullmatch(r"u*r", cmd) else -1


def build(path):
    """Return (root, number of nodes), or (root, -1) after an error."""
    root = cur = None
    count = 0
    it = lines_of(path)
    for cmd_line, cmd in it:
        nxt = next(it, None)
        if nxt is None or not re.fullmatch(r"\s*[+-]?\d+\s*", nxt[1]):
            print("line %2d: a whole number must follow command '%s'"
                  % (cmd_line + 1, cmd))
            return root, -1
        data = int(nxt[1])
        if cmd == "0":
            if root is not None:
```

Listing 12.6 (Continued) *tree_from_file.py: the same tree builder in Python.*

```
                print("line %2d: the tree already has a root" % cmd_line)
                return root, -1
            root = cur = TNode(data, None)
            print("line %2d: %-5s-> %d is the root" % (cmd_line, cmd, data))
        elif root is None:
            print("line %2d: the first command must be 0" % cmd_line)
            return root, -1
        elif cmd == "l":
            if cur.left is not None:
                print("line %2d: %d already has a left child"
                      % (cmd_line, cur.data))
                return root, -1
            cur.left = TNode(data, cur)
            print("line %2d: %-5s-> %d is the left child of %d"
                  % (cmd_line, cmd, data, cur.data))
            cur = cur.left
        else:
            k = climbs(cmd)
            if k < 0:
                print("line %2d: unknown command '%s'" % (cmd_line, cmd))
                return root, -1
            at = cur
            for _ in range(k):
                if at.parent is None:
                    print("line %2d: '%s' climbs above the root" % (cmd_line, cmd))
                    return root, -1
                at = at.parent
            if at.right is not None:
                print("line %2d: %d already has a right child"
                      % (cmd_line, at.data))
                return root, -1
            at.right = TNode(data, at)
            print("line %2d: %-5s-> up %d to %d, %d is its right child"
                  % (cmd_line, cmd, k, at.data, data))
            cur = at.right
        count += 1
    return root, count


def preorder(t, out):
    if t is not None:
        out.append(t.data)
        preorder(t.left, out)
        preorder(t.right, out)
    return out
```

Listing 12.6 (Continued) *tree_from_file.py: the same tree builder in Python.*

```
def inorder(t, out):
    if t is not None:
        inorder(t.left, out)
        out.append(t.data)
        inorder(t.right, out)
    return out


def postorder(t, out):
    if t is not None:
        postorder(t.left, out)
        postorder(t.right, out)
        out.append(t.data)
    return out


def height(t):
    return 0 if t is None else 1 + max(height(t.left), height(t.right))


def sideways(t, depth=0):
    if t is not None:
        sideways(t.right, depth + 1)
        print("%*s%d" % (6 * depth, "", t.data))
        sideways(t.left, depth + 1)


def main():
    if len(sys.argv) != 2:
        print("usage: tree_from_file.py <file>", file=sys.stderr)
        sys.exit(1)
    try:
        root, n = build(sys.argv[1])
    except OSError as e:
        print("%s: %s" % (sys.argv[1], e.strerror), file=sys.stderr)
        sys.exit(1)
    if n <= 0:
        if n == 0:
            print("%s holds no nodes" % sys.argv[1])
        sys.exit(1)
    print("nodes: %d, height: %d" % (n, height(root)))
    print("preorder :" + "".join(" %d" % v for v in preorder(root, [])))
    print("inorder  :" + "".join(" %d" % v for v in inorder(root, [])))
    print("postorder:" + "".join(" %d" % v for v in postorder(root, [])))
    print("sideways (root on the left, right children above):")
```

Listing 12.6 (Continued) *tree_from_file.py: the same tree builder in Python.*

```
    sideways(root)


if __name__ == "__main__":
    main()
```

12.4.6 Running and testing it

`./tree_from_file ../data/tree5.txt` prints:

```
line  1: 0    -> 50 is the root
line  3: l    -> 30 is the left child of 50
line  5: l    -> 20 is the left child of 30
line  7: ur   -> up 1 to 30, 40 is its right child
line  9: uur  -> up 2 to 50, 70 is its right child
nodes: 5, height: 3
preorder : 50 30 20 40 70
inorder  : 20 30 40 50 70
postorder: 20 40 30 70 50
sideways (root on the left, right children above):
      70
50
            40
      30
            20
```

Each trace line matches a panel of **Figure 12.1**, the three orders match the expectations of Section 12.4.2, and the sideways drawing, turned a quarter to the right, is the last panel of the figure.

A larger test, `data/tree15.txt`, builds the perfectly balanced tree of the numbers 1 to 15, with climbs of one, two and three levels:

```
line  1: 0    -> 8 is the root
line  3: l    -> 4 is the left child of 8
line  5: l    -> 2 is the left child of 4
line  7: l    -> 1 is the left child of 2
line  9: ur   -> up 1 to 2, 3 is its right child
line 11: uur  -> up 2 to 4, 6 is its right child
line 13: l    -> 5 is the left child of 6
line 15: ur   -> up 1 to 6, 7 is its right child
line 17: uuur -> up 3 to 8, 12 is its right child
line 19: l    -> 10 is the left child of 12
line 21: l    -> 9 is the left child of 10
line 23: ur   -> up 1 to 10, 11 is its right child
line 25: uur  -> up 2 to 12, 14 is its right child
line 27: l    -> 13 is the left child of 14
line 29: ur   -> up 1 to 14, 15 is its right child
nodes: 15, height: 4
preorder : 8 4 2 1 3 6 5 7 12 10 9 11 14 13 15
inorder  : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
postorder: 1 3 2 5 7 6 4 9 11 10 13 15 14 12 8
sideways (root on the left, right children above):
                  15
            14
                  13
      12
                  11
            10
                  9
8
                  7
            6
                  5
      4
                  3
            2
                  1
```

The inorder is 1 to 15 in order, because this file happens to build a *binary search tree*, in which every left subtree holds smaller numbers and every right subtree larger ones; the height is 4, the smallest possible for 15 nodes. Three damaged files check the error messages; each prints the nodes built so far and stops:

```
line  1: 0    -> 50 is the root
line  3: l    -> 30 is the left child of 50
line  5: 'uur' climbs above the root
line  1: 0    -> 50 is the root
line  3: l    -> 30 is the left child of 50
line  6: a whole number must follow command 'l'
line  1: 0    -> 50 is the root
line  3: l    -> 30 is the left child of 50
line  5: ur   -> up 1 to 50, 40 is its right child
line  7: 50 already has a right child
```

In the first, `uur` from 30 climbs to 50 and then finds no parent. In the second, `forty` is not a number. In the third, the second `ur` climbs from 40 to 50, which already has a right child. In all three cases the program frees the partial tree before it stops, which the sanitizer confirms.

12.4.7 A tempting wrong solution

The most common mistake is an off-by-one in the climb: starting it from the *parent* of the current node, "because `u` means up, and we are already one level below". It is tempting because it makes `ur` look like "the right child of my parent", which sounds natural. Replay the example with that rule. Lines 1 to 6 are unaffected. Line 7, `ur`: from 20, the rule first goes to the parent 30 and then climbs one more level to 50; 40 becomes the right child of *50*. Line 9, `uur`: from 40, it goes to the parent 50 and then tries to climb two more levels, above the root, and the program reports an error on a file that is perfectly correct. With a slightly different file, the wrong rule produces no error at all, only a tree of the wrong shape, which is worse. The defense is the one of Section 12.2: trace the example of the statement by hand *before* coding, and make the program print a trace line for every command, so that the first line where program and paper disagree points at the mistake.

A second mistake is to forget the parent pointer and climb by searching the tree from the root for the node whose child is `cur`. That works, but it costs a walk of the whole tree for every climb, and it is far more code to get right.

12.4.8 Check yourself

Write the command file for the tree whose preorder is 8 4 2 6 12 10 14 (a balanced tree of seven nodes), run it, and compare the inorder with 2 4 6 8 10 12 14. (One answer: `0 8`, `l 4`, `l 2`, `ur 6`, `uur 12`, `l 10`, `ur 14`, each command followed by its number on the next line.)

12.5 Problem 12.3: The Knight's Shortest Path

12.5.1 The problem

> **PROBLEM 12.3.** On an 8 × 8 chessboard, a knight stands on a start square and must reach a target square. Find the smallest number of moves it needs. Use a queue implemented as a linked list, with the operations *enqueue* (add at the back) and *dequeue* (take from the front). For example, from row 7, column 0 to row 0, column 7 the knight needs 6 moves.

In our words: a knight moves in an "L": two squares in one direction and one square at a right angle, which gives at most eight possible moves from any square, fewer near the edges. Rows and columns are numbered 0 to 7. We must find the length of the shortest sequence of moves from start to target. We decide: the program reads queries from standard input, one per line, `r1 c1 r2 c2`; it prints the number of moves and also one shortest path, since a path can be checked by hand;

a query with a square off the board is answered with a message; and the word `board` after a query prints the distance of every square from the start.

12.5.2 Understanding it by hand

Think of dropping a stone into a pond. The first ring of waves reaches every point at distance 1, the second ring every point at distance 2, and so on. **Breadth-first search** (BFS) explores the board the same way. Start with the start square at distance 0. Every square one knight move away gets distance 1. Every square one move away from those, and not yet reached, gets distance 2. Continue ring by ring until the target is reached. Because each ring is finished before the next begins, the first time a square is reached, it is reached by a shortest path: any shorter path would have reached it in an earlier ring [4].

The tool that keeps the rings in order is a **queue**: squares are taken from the front and their new neighbors added at the back, so all squares of distance d are taken out before any square of distance $d + 1$. When a square u taken from the queue reaches a new square v,

$$\text{dist}(v) = \text{dist}(u) + 1 \tag{12.2}$$

and u is remembered as the square v was reached *from*, so that at the end the path can be followed backwards from the target to the start.

Figure 12.2 shows the result for the example: the distance of every square from (7,0). The target (0,7) has distance 6, as the statement says, and the amber squares are one shortest path, each step one knight move and one ring further.

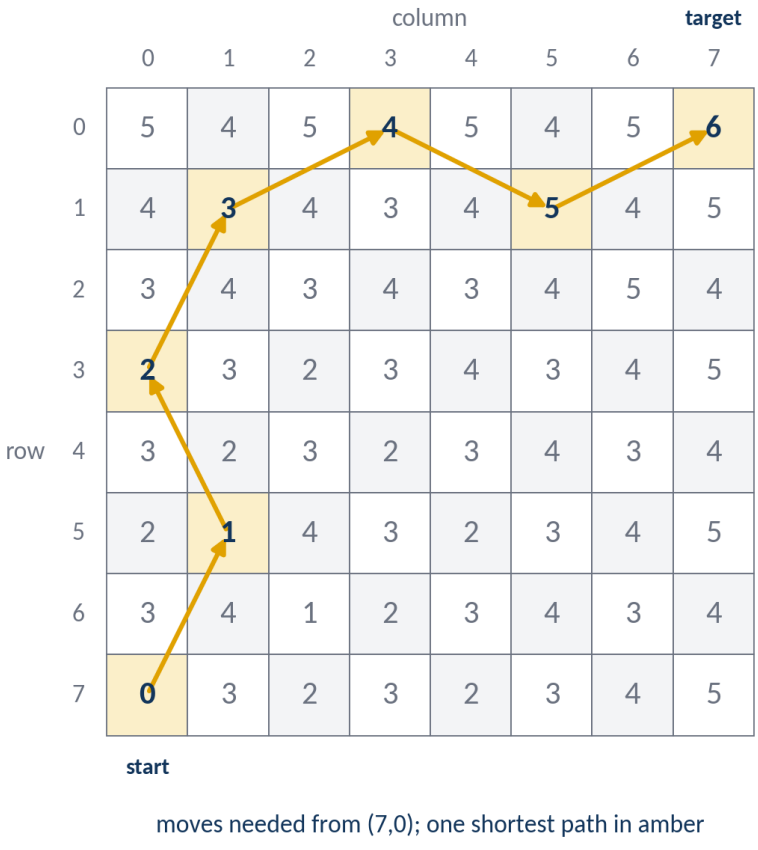


Figure 12.2 The number of knight moves needed from the start (7,0) to every square; the target (0,7) needs 6, and the amber squares show one shortest path.

Look at the numbers near the start: (5,1) and (6,2) have distance 1, as the two moves from a corner square; and (6,1), right next to the start diagonally, needs 4 moves. Distance on a knight's board has little to do with distance on the ruler, which is why the problem needs an algorithm and not a formula.

One more detail decides whether BFS is fast or slow: a square must be marked as *visited* when it is put **into** the queue, not when it comes out. Otherwise a square reachable from several squares of the current ring is put into the queue several times, and each copy later adds its neighbors again.

12.5.3 Designing it on paper

The queue is a linked list with a `head` (the front) and a `tail` (the back), exactly the `List` of Chapter 10: *enqueue* is `append`, *dequeue* is "remove the first node". **Pseudocode 12.3** is the search.

Pseudocode 12.3 Breadth-first search for the fewest knight moves, with the queue as a linked list.

```
KNIGHT-BFS(s, t)
1  for every square v: dist[v] = -1, visited[v] = FALSE
2  dist[s] = 0; visited[s] = TRUE; ENQUEUE(Q, s)
3  while Q is not empty
4      u = DEQUEUE(Q)
5      if u = t: stop                  // the first arrival is the shortest
6      for each of the 8 knight moves from u to a square v on the board
7          if not visited[v]
8              visited[v] = TRUE       // mark when ENQUEUED
9              dist[v] = dist[u] + 1; from[v] = u
10             ENQUEUE(Q, v)
11 return dist[t]
```

The cost. Every square is enqueued at most once because of line 8, so at most 64 squares enter the queue and each tries 8 moves: at most $64 \cdot 8 = 512$ checks per query. On an $N \times N$ board the work grows like N^2, still tiny for any board a person would draw.

12.5.4 The C solution

Listing 12.7 has the pieces of the problem: the type `Cell`, the move tables `DR` and `DC` (the eight moves as row and column changes), `is_valid`, and the queue functions `enqueue`, `dequeue` and `free_queue`. The search `bfs` follows **Pseudocode 12.3**; an extra parameter `late` switches to the tempting variant of Section 12.5.7 for comparison. `print_path` follows `from` backwards from the target, stores the squares in an array, and prints them in reverse. A query with the word `board` runs the search once more with an impossible target, so that it fills the whole board.

Listing 12.7 `knight.c`: breadth-first search with a linked-list queue for the fewest knight moves.

```c
/* knight.c - Problem 12.3: the fewest knight moves from one square of an
 * 8 x 8 board to another, found by breadth-first search (BFS) with a queue
 * built as a linked list.
 * Input: one query per line, "r1 c1 r2 c2" (rows and columns 0..7);
 * add the word "board" to print every square's distance from the start,
 * or "late" to mark squares visited when they leave the queue instead. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 8

typedef struct {
    int r, c;
} Cell;

typedef struct QNode {
    Cell cell;
    struct QNode *next;
} QNode;

typedef struct {
    QNode *head, *tail;      /* take from the head, add at the tail */
} Queue;

static const int DR[8] = {-2, -2, -1, -1, 1, 1, 2, 2};
static const int DC[8] = {-1, 1, -2, 2, -2, 2, -1, 1};

static int is_valid(int r, int c)
{
    return r >= 0 && r < N && c >= 0 && c < N;
}

static void enqueue(Queue *q, Cell cell)
{
    QNode *n = malloc(sizeof *n);
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(EXIT_FAILURE);
    }
    n->cell = cell;
    n->next = NULL;
    if (q->tail != NULL)
        q->tail->next = n;   /* link behind the current last node */
    else
        q->head = n;         /* the queue was empty */
```

Listing 12.7 (Continued) *knight.c: breadth-first search with a linked-list queue for the fewest knight moves.*

```
    q->tail = n;
}

static Cell dequeue(Queue *q)    /* call only when q->head != NULL */
{
    QNode *n = q->head;
    Cell cell = n->cell;
    q->head = n->next;
    if (q->head == NULL)
        q->tail = NULL;      /* the queue is empty again */
    free(n);
    return cell;
}

static void free_queue(Queue *q)
{
    while (q->head != NULL)
        dequeue(q);
}

/* BFS from s. Fills dist (-1 = not reached) and from (the square each
 * square was first reached from). Returns how many squares were enqueued. */
static int bfs(Cell s, Cell t, int dist[N][N], Cell from[N][N], int late)
{
    int visited[N][N] = {{0}}, enqueued = 0;
    Queue q = {NULL, NULL};
    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            dist[r][c] = -1;
    dist[s.r][s.c] = 0;
    if (!late)
        visited[s.r][s.c] = 1;
    enqueue(&q, s);
    enqueued++;
    while (q.head != NULL) {
        Cell cur = dequeue(&q);
        if (late) {                      /* the tempting variant */
            if (visited[cur.r][cur.c])
                continue;
            visited[cur.r][cur.c] = 1;
        }
        if (cur.r == t.r && cur.c == t.c)
            break;                       /* the first arrival is the shortest */
        for (int k = 0; k < 8; k++) {
            int r = cur.r + DR[k], c = cur.c + DC[k];
            if (!is_valid(r, c) || visited[r][c])
```

Listing 12.7 (Continued) *knight.c: breadth-first search with a linked-list queue for the fewest knight moves.*

```
                continue;
            if (!late)
                visited[r][c] = 1;       /* mark it NOW, when it is enqueued */
            if (dist[r][c] < 0) {
                dist[r][c] = dist[cur.r][cur.c] + 1;
                from[r][c] = cur;
            }
            enqueue(&q, (Cell){r, c});
            enqueued++;
        }
    }
    free_queue(&q);
    return enqueued;
}

static void print_path(Cell s, Cell t, Cell from[N][N])
{
    Cell path[N * N];
    int n = 0;
    for (Cell p = t; !(p.r == s.r && p.c == s.c); p = from[p.r][p.c])
        path[n++] = p;
    path[n++] = s;
    printf("  path:");
    for (int i = n - 1; i >= 0; i--)
        printf(" (%d,%d)", path[i].r, path[i].c);
    printf("\n");
}

int main(void)
{
    char line[128], word[16] = "";
    while (fgets(line, sizeof line, stdin) != NULL) {
        Cell s, t;
        word[0] = '\0';
        int got = sscanf(line, "%d %d %d %d %15s", &s.r, &s.c, &t.r, &t.c, word);
        if (got < 4)
            continue;                    /* not a query: skip the line */
        if (!is_valid(s.r, s.c) || !is_valid(t.r, t.c)) {
            printf("(%d,%d) -> (%d,%d): off the board\n", s.r, s.c, t.r, t.c);
            continue;
        }
        int dist[N][N], late = strcmp(word, "late") == 0;
        Cell from[N][N];
        int enq = bfs(s, t, dist, from, late);
        printf("(%d,%d) -> (%d,%d): %d move(s), %d square(s) enqueued%s\n",
               s.r, s.c, t.r, t.c, dist[t.r][t.c], enq,
```

Listing 12.7 (Continued) *knight.c: breadth-first search with a linked-list queue for the fewest knight moves.*

```
               late ? " (marked late)" : "");
        print_path(s, t, from);
        if (strcmp(word, "board") == 0) {
            bfs(s, (Cell){-1, -1}, dist, from, 0);   /* no target: fill all */
            printf("  distance of every square from (%d,%d):\n", s.r, s.c);
            for (int r = 0; r < N; r++) {
                printf("   ");
                for (int c = 0; c < N; c++)
                    printf(" %d", dist[r][c]);
                printf("\n");
            }
        }
    }
    return 0;
}
```

The two branches of `enqueue` are the part that most often goes wrong. If the queue already has a last node, the new node is linked behind it; if the queue is empty, the new node is also the first node. `dequeue` has the mirror image: when it takes out the last node, `tail` must become `NULL` too, or the next `enqueue` would link a new node behind freed memory.

12.5.5 The Python solution

Listing 12.8 builds the queue as a small class with its own linked nodes, to mirror the C program. (In everyday Python one would use `collections.deque`, which does the same job [3].)

Listing 12.8 `knight.py`: the same search in Python.

```
# knight.py - Problem 12.3: the fewest knight moves between two squares of an
# 8 x 8 board, by breadth-first search with a queue built as a linked list.
# Input: one query per line, "r1 c1 r2 c2" (0..7), optionally followed by
# "board" (print all distances) or "late" (mark visited when dequeued).
import sys

N = 8
DR = [-2, -2, -1, -1, 1, 1, 2, 2]
DC = [-1, 1, -2, 2, -2, 2, -1, 1]


class QNode:
    def __init__(self, cell):
        self.cell = cell
        self.next = None


class Queue:
    def __init__(self):
        self.head = self.tail = None       # take from head, add at tail

    def enqueue(self, cell):
        n = QNode(cell)
        if self.tail is not None:
            self.tail.next = n
        else:
            self.head = n
        self.tail = n

    def dequeue(self):
        n = self.head
        self.head = n.next
        if self.head is None:
            self.tail = None
        return n.cell


def is_valid(r, c):
    return 0 <= r < N and 0 <= c < N
```

Listing 12.8 (Continued) *knight.py: the same search in Python.*

```
def bfs(s, t, late=False):
    """Return (dist, from, number of squares enqueued)."""
    dist = [[-1] * N for _ in range(N)]
    frm = [[None] * N for _ in range(N)]
    visited = [[False] * N for _ in range(N)]
    dist[s[0]][s[1]] = 0
    if not late:
        visited[s[0]][s[1]] = True
    q = Queue()
    q.enqueue(s)
    enqueued = 1
    while q.head is not None:
        cur = q.dequeue()
        if late:
            if visited[cur[0]][cur[1]]:
                continue
            visited[cur[0]][cur[1]] = True
        if cur == t:
            break                           # the first arrival is the shortest
        for k in range(8):
            r, c = cur[0] + DR[k], cur[1] + DC[k]
            if not is_valid(r, c) or visited[r][c]:
                continue
            if not late:
                visited[r][c] = True        # mark it NOW, when it is enqueued
            if dist[r][c] < 0:
                dist[r][c] = dist[cur[0]][cur[1]] + 1
                frm[r][c] = cur
            q.enqueue((r, c))
            enqueued += 1
    return dist, frm, enqueued


def main():
    for line in sys.stdin:
        w = line.split()
        try:
            r1, c1, r2, c2 = (int(x) for x in w[:4])
        except ValueError:
            continue                         # not a query: skip the line
        word = w[4][:15] if len(w) > 4 else ""
        s, t = (r1, c1), (r2, c2)
        if not (is_valid(*s) and is_valid(*t)):
            print("(%d,%d) -> (%d,%d): off the board" % (r1, c1, r2, c2))
            continue
        late = word == "late"
```

Listing 12.8 (Continued) *knight.py: the same search in Python.*

```
        dist, frm, enq = bfs(s, t, late)
        print("(%d,%d) -> (%d,%d): %d move(s), %d square(s) enqueued%s"
              % (r1, c1, r2, c2, dist[r2][c2], enq,
                 " (marked late)" if late else ""))
        path, p = [], t
        while p != s:
            path.append(p)
            p = frm[p[0]][p[1]]
        path.append(s)
        print("  path:" + "".join(" (%d,%d)" % x for x in reversed(path)))
        if word == "board":
            dist, _, _ = bfs(s, (-1, -1))
            print("  distance of every square from (%d,%d):" % s)
            for r in range(N):
                print("    " + "".join(" %d" % d for d in dist[r]))


if __name__ == "__main__":
    main()
```

12.5.6 Running and testing it

The test file `data/knight_queries.txt` holds the example with `board`, the edge case start = target, a single move, a short hop that needs three moves, the corner-to-corner query, the example again with the tempting variant, and a square off the board:

```
7 0 0 7 board
0 0 0 0
0 0 1 2
3 3 4 3
0 0 7 7
7 0 0 7 late
9 0 1 1
```

`./knight < ../data/knight_queries.txt` prints:

```
(7,0) -> (0,7): 6 move(s), 64 square(s) enqueued
  path: (7,0) (5,1) (3,0) (1,1) (0,3) (1,5) (0,7)
  distance of every square from (7,0):
    5 4 5 4 5 4 5 6
    4 3 4 3 4 5 4 5
    3 4 3 4 3 4 5 4
    2 3 2 3 4 3 4 5
    3 2 3 2 3 4 3 4
    2 1 4 3 2 3 4 5
    3 4 1 2 3 4 3 4
    0 3 2 3 2 3 4 5
(0,0) -> (0,0): 0 move(s), 1 square(s) enqueued
  path: (0,0)
(0,0) -> (1,2): 1 move(s), 3 square(s) enqueued
  path: (0,0) (1,2)
(3,3) -> (4,3): 3 move(s), 63 square(s) enqueued
  path: (3,3) (1,2) (2,4) (4,3)
(0,0) -> (7,7): 6 move(s), 64 square(s) enqueued
  path: (0,0) (1,2) (0,4) (1,6) (3,5) (5,6) (7,7)
(7,0) -> (0,7): 6 move(s), 169 square(s) enqueued (marked late)
  path: (7,0) (5,1) (3,0) (1,1) (0,3) (1,5) (0,7)
(9,0) -> (1,1): off the board
```

The example needs 6 moves, and the path can be checked with a finger on **Figure 12.2**: every step changes one coordinate by 2 and the other by 1. The printed board is the figure in numbers. Start = target needs 0 moves and enqueues only the start; (0,0) to (1,2) is a single move; (3,3) to (4,3), one square straight down, needs 3 moves; corner to corner needs 6. The off-board query is

refused. And look at the line marked `(marked late)`: the same answer, 6 moves, but 169 squares entered the queue instead of 64, because squares were put in again and again before their first copy came out. On a chessboard that costs microseconds; on a map with millions of places, it is the difference between a quick answer and none.

12.5.7 A tempting wrong solution

Chapter 5's recursion suggests a shorter program: from the current square, try the first move; from there, try the first move again; go as deep as possible, and step back only when stuck. This is **depth-first search** (DFS), and **Listing 12.9** implements it with a `visited` array so that it cannot run in circles. It returns the length of the first path it finds; **Listing 12.10** is the Python version.

Listing 12.9 `knight_dfs.c`: depth-first search reports the first path it finds, not the shortest.

```c
/* knight_dfs.c - a tempting wrong solution to Problem 12.3: depth-first
 * search. It dives along the first move that works and reports the first
 * path it finds, which is a path, but rarely the shortest one. */
#include <stdio.h>

#define N 8

static const int DR[8] = {-2, -2, -1, -1, 1, 1, 2, 2};
static const int DC[8] = {-1, 1, -2, 2, -2, 2, -1, 1};
static int visited[N][N];

/* Returns the length of the first path found from (r,c) to (tr,tc), or -1. */
static int dfs(int r, int c, int tr, int tc, int depth)
{
    if (r == tr && c == tc)
        return depth;
    visited[r][c] = 1;
    for (int k = 0; k < 8; k++) {
        int nr = r + DR[k], nc = c + DC[k];
        if (nr >= 0 && nr < N && nc >= 0 && nc < N && !visited[nr][nc]) {
            int found = dfs(nr, nc, tr, tc, depth + 1);
            if (found >= 0)
                return found;           /* stop at the FIRST path found */
        }
    }
    return -1;
}

int main(void)
{
    const int q[3][4] = {{7, 0, 0, 7}, {0, 0, 7, 7}, {3, 3, 4, 3}};
    for (int i = 0; i < 3; i++) {
        for (int r = 0; r < N; r++)
            for (int c = 0; c < N; c++)
                visited[r][c] = 0;
        int len = dfs(q[i][0], q[i][1], q[i][2], q[i][3], 0);
        printf("(%d,%d) -> (%d,%d): DFS found a path of %d move(s)\n",
               q[i][0], q[i][1], q[i][2], q[i][3], len);
    }
    return 0;
}
```

Listing 12.10 `knight_dfs.py`: the same mistake in Python.

```
# knight_dfs.py - a tempting wrong solution to Problem 12.3: depth-first
# search reports the first path it finds, which is rarely the shortest.
import sys

N = 8
DR = [-2, -2, -1, -1, 1, 1, 2, 2]
DC = [-1, 1, -2, 2, -2, 2, -1, 1]


def dfs(r, c, tr, tc, depth, visited):
    if (r, c) == (tr, tc):
        return depth
    visited[r][c] = True
    for k in range(8):
        nr, nc = r + DR[k], c + DC[k]
        if 0 <= nr < N and 0 <= nc < N and not visited[nr][nc]:
            found = dfs(nr, nc, tr, tc, depth + 1, visited)
            if found >= 0:
                return found            # stop at the FIRST path found
    return -1


def main():
    sys.setrecursionlimit(10000)
    for q in ([7, 0, 0, 7], [0, 0, 7, 7], [3, 3, 4, 3]):
        visited = [[False] * N for _ in range(N)]
        print("(%d,%d) -> (%d,%d): DFS found a path of %d move(s)"
              % (q[0], q[1], q[2], q[3], dfs(q[0], q[1], q[2], q[3], 0, visited)))


if __name__ == "__main__":
    main()
```

```
(7,0) -> (0,7): DFS found a path of 6 move(s)
(0,0) -> (7,7): DFS found a path of 52 move(s)
(3,3) -> (4,3): DFS found a path of 25 move(s)
```

This is the most instructive output of the chapter. On the example of the statement, DFS answers 6, the right number, and a programmer who tests only the example is satisfied. On the corner-to-corner query it answers 52, and on the three-move hop 25. DFS finds *a* path, and with the `visited` array it finds one quickly, but it follows its first choice of move as far as it goes, so the path it finds depends on the order of the move table, not on the distance. BFS finds the shortest path because of how it explores, ring by ring, not because of luck. A correct answer on the given example proves nothing about an algorithm; only an argument (why the first arrival is the shortest) and a variety of tests do.

12.5.8 Check yourself

Change the program for a *king*, which moves one square in any of eight directions. Which lines change? (Only the tables `DR` and `DC`.) Predict the answer for (7,0) to (0,7) before running it: a king needs as many moves as the larger of the row and column differences, max(7,7) = 7.

12.6 The Problems Side by Side

Table 12.1 puts the three problems next to each other. The pattern is worth remembering: in each, the data structure was chosen to make the one operation that matters cheap, and each needed one extra pointer or mark (`last`, `parent`, `visited`) that the statement did not mention.

Table 12.1 The three problems compared: structure, key operation, the idea that makes it work, cost, and the mistake to avoid.

	Problem 12.1	Problem 12.2	Problem 12.3
Structure	singly linked list	binary tree with parent pointers	queue as a linked list, board arrays
Key operation	swap neighbors' data	climb `k` levels, insert a child	enqueue at the back, dequeue at the front
The extra idea	`last` marks the sorted tail	a `parent` pointer in every node	mark `visited` when enqueuing
Cost	$O(n^2)$ sort, $O(n)$ insert	$O(n)$ plus the climbs	at most 64 squares, 512 checks
Tempting mistake	a single pass	climbing from the parent	depth-first search
Tests that caught it	reversed list, empty list	the statement's own example	queries other than the example

12.7 Using It for Real

These three structures appear all around small businesses once you know their shape. A *sorted list with insertion* is the waiting list of a clinic ordered by appointment time, with walk-in patients added at the end. A *tree* is the organization chart of a cooperative, the folder structure on a laptop, or the chart of accounts of a small business, where "Expenses" has children "Rent", "Wages" and "Supplies"; a file that lists such a tree with navigation commands, as in Problem 12.2, is a compact way to store it, and the parent pointer is what lets the program answer "which group does this account belong to?". *Breadth-first search* answers every "fewest steps" question: the fewest transfers between two bus stops in a city's network, the fewest handovers to pass a message through a group of friends, the fewest turns in a warehouse between two shelves. For the laundry, it could answer the fewest stops needed to reach every delivery address by motorbike from a map of which streets connect, with the queue holding the places to visit next.

When the steps are not all equal, for example streets of different lengths, plain BFS is no longer enough, and the next courses teach algorithms that generalize it. But the queue, the visited mark and the "from" pointer for rebuilding the path stay the same.

12.8 Common Mistakes, and How to Avoid Them

- **Coding before drawing.** Every pointer error in this chapter is visible on paper in a minute and invisible on the screen for an hour. Draw each step with boxes and arrows first.
- **Reading `p->next->data` when `p->next` may be `NULL`.** Put the `NULL` test in the loop condition, as in line 6 of **Pseudocode 12.1**.
- **Forgetting the empty structure.** An empty list in `insert_end`, an empty queue in `dequeue`, a missing root in the tree: each needs its own branch, and a test.
- **Updating one end of a queue but not the other.** `enqueue` on an empty queue sets `head` too; `dequeue` of the last node sets `tail` to `NULL`.
- **Off-by-one in navigation.** Climbing from the parent instead of the current node, or stopping the list walk one node too late. Trace the example by hand and print a trace line per step.
- **Marking late in BFS.** Mark squares visited when they enter the queue.
- **Believing the example.** Both wrong solutions of this chapter pass the example of their statement. Test edge cases and cases you have computed by hand.

- **Freeing in the wrong order.** Free a tree's children before the node (postorder), and a list's successor pointer must be saved before the node is freed.
- **Not running the sanitizer.** It turns silent memory errors into precise reports; use it on every pointer program.

12.9 Chapter Summary

- A data-structure problem is solved by restating it, drawing a small example step by step, designing on paper, coding one function at a time, and testing with prepared cases under the sanitizer.
- Problem 12.1: bubble sort on a singly linked list swaps the data of neighbors, stops one node earlier each pass and stops altogether after a pass without swaps (**Pseudocode 12.1**); its swap count equals the number of inversions (Equation (12.1)). Insertion at the end walks to the last node, with a special case for the empty list.
- Problem 12.2: a parent pointer in every node makes "climb k levels" a loop of k steps; the tree is built by **Pseudocode 12.2** with careful error messages, shown by preorder, inorder, postorder and a sideways print (**Figure 12.1**), and freed in postorder.
- Problem 12.3: breadth-first search with a linked-list queue explores ring by ring, so the first arrival is the shortest (Equation (12.2), **Pseudocode 12.3**, **Figure 12.2**); marking squares when they are enqueued keeps the queue small; the "from" array rebuilds the path.
- The tempting wrong solutions, a single bubble pass, a climb from the parent, and depth-first search, all pass the example of their statement, which is why testing beyond the example is not optional (**Table 12.1**).

12.10 Exercises

1. Without running the program, give the number of passes and swaps that `list_sort` needs for the list 3, 1, 2, 5, 4, and the list after inserting 0 at the end.
2. Sort the linked list of Problem 12.1 by *relinking nodes* instead of swapping data, using insertion sort: take the nodes off the list one by one and put each into its place in a new sorted list. Show that equal numbers keep their original order.
3. Draw the tree built by the commands `0 1`, `l 2`, `l 3`, `r 4`, `uuur 5`, `l 6` (each number on its own data line), and give its preorder and inorder. Which command would make the file invalid if `uuur` were replaced by `uuuur`, and what would the program print?
4. Add to `tree_from_file` a function that prints, for every node, the path from the root, such as `50 L 30 R 40`, using the parent pointers. What is its cost for a tree of n nodes and height h?
5. For the knight, which squares of the board need the most moves from (7,0), and how many? Read the answer from **Figure 12.2**, then check it with the program.
6. Change `knight.c` so that the board size N is read from the first line of the input (up to 20), and give the fewest moves from one corner to the opposite corner for $N = 4$, 5 and 10.

12.11 Solutions to the Exercises

1. The inversions of 3, 1, 2, 5, 4 are (3,1), (3,2) and (5,4): three, so 3 swaps by Equation (12.1). Pass 1 swaps 3 with 1, 3 with 2 and 5 with 4, giving 1, 2, 3, 4, 5 with 3 swaps; pass 2 makes no swap and stops. Two passes, three swaps; after the insertion: 1, 2, 3, 4, 5, 0.

2. Listing 12.11 gives each node a second field `born`, its position in the input, so that the output shows *which* node went where. The loop takes the first node off the input list; if it belongs before the first node of the sorted list, it becomes the new first node; otherwise the walk finds the last node whose number is `<=` the new one and links the new node after it, with the two assignments in the safe order of Chapter 10. Because the walk goes past equal numbers (`<=`), a node is placed after the equal nodes already there, so equal numbers keep their input order: the insertion sort is *stable*. **Listing 12.12** is the Python version.

Listing 12.11 `list_insertion_sort.c`: insertion sort by relinking nodes.

```c
/* list_insertion_sort.c - Exercise 12.2: sort the list by RELINKING nodes
 * (insertion sort): take the nodes off one by one and put each into its
 * place in a new, sorted list. No data field is ever changed. */
#include <stdio.h>
#include <stdlib.h>

struct sNode {
    int data;
    int born;                /* position in the input: shows which node is which */
    struct sNode *next;
};

static struct sNode *push_front(struct sNode *head, int value, int born)
{
    struct sNode *n = malloc(sizeof *n);
    if (n == NULL)
        exit(EXIT_FAILURE);
    n->data = value;
    n->born = born;
    n->next = head;
    return n;
}

static void print_list(const char *label, const struct sNode *p)
{
    printf("%-7s:", label);
    for (; p != NULL; p = p->next)
        printf(" %d(#%d)", p->data, p->born);
    printf("\n");
}

static struct sNode *insertion_sort(struct sNode *head)
{
    struct sNode *sorted = NULL;
    while (head != NULL) {
        struct sNode *n = head;          /* take the first node off */
        head = head->next;
        if (sorted == NULL || n->data < sorted->data) {
            n->next = sorted;            /* it goes to the front */
            sorted = n;
        } else {
            struct sNode *p = sorted;    /* find the last node <= n->data */
            while (p->next != NULL && p->next->data <= n->data)
                p = p->next;
            n->next = p->next;           /* 1: n points at p's successor */
            p->next = n;                 /* 2: p points at n */
```

Listing 12.11 (Continued) *list_insertion_sort.c: insertion sort by relinking nodes.*

```
        }
    }
    return sorted;
}

int main(void)
{
    const int values[] = {29, 4, 17, 4, 42, 8, 15};
    struct sNode *head = NULL;
    for (int i = 6; i >= 0; i--)
        head = push_front(head, values[i], i + 1);
    print_list("before", head);
    head = insertion_sort(head);
    print_list("after", head);
    while (head != NULL) {
        struct sNode *next = head->next;
        free(head);
        head = next;
    }
    return 0;
}
```

Listing 12.12 `list_insertion_sort.py`: the same in Python.

```
# list_insertion_sort.py - Exercise 12.2: sort the list by RELINKING nodes
# (insertion sort). No data field is ever changed.


class sNode:
    def __init__(self, data, born, next=None):
        self.data = data
        self.born = born               # position in the input
        self.next = next


def print_list(label, p):
    s = "%-7s:" % label
    while p is not None:
        s += " %d(#%d)" % (p.data, p.born)
        p = p.next
    print(s)


def insertion_sort(head):
    sorted_ = None
    while head is not None:
        n, head = head, head.next           # take the first node off
        if sorted_ is None or n.data < sorted_.data:
            n.next, sorted_ = sorted_, n    # it goes to the front
        else:
            p = sorted_                     # find the last node <= n.data
            while p.next is not None and p.next.data <= n.data:
                p = p.next
            n.next = p.next                 # 1: n points at p's successor
            p.next = n                      # 2: p points at n
    return sorted_


def main():
    values = [29, 4, 17, 4, 42, 8, 15]
    head = None
    for i in range(len(values) - 1, -1, -1):
        head = sNode(values[i], i + 1, head)
    print_list("before", head)
    head = insertion_sort(head)
    print_list("after", head)


if __name__ == "__main__":
    main()
```

```
before : 29(#1) 4(#2) 17(#3) 4(#4) 42(#5) 8(#6) 15(#7)
after  : 4(#2) 4(#4) 8(#6) 15(#7) 17(#3) 29(#1) 42(#5)
```

The node born second (the first 4) still comes before the node born fourth (the second 4), and no `data` field was changed: the nodes moved.

3. 1 is the root; 2 is its left child; 3 is 2's left child; `r 4` makes 4 the right child of the *current* node 3; `uuur 5` climbs from 4 to 3, to 2, to 1 and makes 5 the right child of 1; `l 6` makes 6 the left child of 5. Preorder: 1 2 3 4 5 6; inorder: 3 4 2 1 6 5. With `uuuur` the climb reaches 1 after three steps and has no parent for the fourth, so the program prints `line  9: 'uuuur' climbs above the root` (the command is on line 9 of the file) and stops.

4. For each node, follow `parent` up to the root, storing the nodes in an array, then print them from the root down, writing `L` or `R` according to whether each node is its parent's left or right child. Each node costs at most h steps, so all nodes cost $O(nh)$: $O(n\log n)$ for a balanced tree, $O(n^2)$ for a tree that is one long chain.

5. The largest number in **Figure 12.2** is 6, reached only by (0,7), the opposite corner. The program with the `board` option prints the same board, and no square shows a larger number.

6. Read `N` first and use it instead of the constant 8 in `is_valid`, in the loops and in the array sizes (declare the arrays with the maximum 20 and use only the first `N` rows and columns). Corner to corner needs 2 moves for $N = 4$, for example (0,0), (2,1), (3,3); 4 moves for $N = 5$; and 6 moves for $N = 10$; the program confirms each value. A small board is not always easier: on a 3×3 board the center square cannot be reached at all, and `dist` stays -1, a case the program must report as "unreachable".

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, §2.2 "Linear Lists" and §2.3 "Trees."

[2] P. J. Deitel and H. M. Deitel, *C How to Program*, 9th ed. Hoboken, NJ, USA: Pearson, 2022, ch. 12 "Data Structures."

[3] Python Software Foundation, *The Python Standard Library*, Python 3.14: "re — Regular expression operations" and "collections.deque." [Online]. Available: https://docs.python.org/3/library/ (accessed Sep. 2026).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 10 "Elementary Data Structures" and §20.2 "Breadth-first search."

Chapter 13
C++ as a Better C

For twelve chapters you have written C: a small, honest language in which you manage every byte yourself. This chapter moves you to C++, the language that grew out of C and still accepts almost every C program you have written, but that adds tools which remove whole families of mistakes. By the end of the chapter you will be able to compile C++17 programs with `g++`; read and write with `cin` and `cout` instead of `scanf` and `printf`; store text in `std::string` and lists of values in `std::vector` instead of fixed arrays; pass arguments by *reference* and explain how that differs from passing a value or a pointer; give several functions one name through *overloading* and let callers skip arguments that have *default values*; allocate with `new` and free with the matching `delete`, and know why you will rarely need to; and put all of it together in version 1.0 of the Profitina Laundry cashier, rewritten in C++ and, as always in this book, also in Python with exactly the same output.

13.1 Bu Rina's Receipts

Bu Rina runs Profitina Laundry, a small laundry on a busy street near a university. Students drop off a bag of clothes in the morning, office workers bring shirts to be ironed, and on Sundays families bring bed covers. Every bag is weighed, the weight and the service are typed into a laptop, and a small program prints the price. You have followed that program through the C chapters of this book: it learned to loop over a queue of customers, to keep orders in arrays and structures, and in Chapter 11 to save them to a file.

The C cashier works, but it has a history of small accidents, and each one taught a lesson. Once a customer gave her full name, *Raden Ajeng Kusumawardhani Sulistyaningrum*, and the name overflowed the 30-character array that held it; the letters that did not fit were written over the weight that came next in memory, and the receipt showed a price of several million rupiah. Once a new helper added an order, forgot the `&` in a `scanf` call, and the program crashed on the first customer of the day. Once the program forgot to `free` the memory of a day's orders and, after a week without a restart, the laptop became slow. None of these mistakes was stupid. They are the mistakes *everyone* makes in C, because C trusts the programmer with every detail and checks almost nothing.

C++ was created for exactly this situation. It keeps everything that makes C fast and close to the machine, and it adds a layer of tools that do the dangerous bookkeeping for you: a string type that grows to fit any name, a list type that grows as orders arrive and frees itself when it is no longer needed, input and output that know the type of every variable, and parameters that can change the caller's variable without a single `&` or `*` at the call. The whole point of this chapter is that you can keep thinking the way you learned in the C chapters and let the language take care of the parts that used to bite.

Who does this help? First, Bu Rina: a cashier program that cannot be broken by a long name is a program she does not have to call anyone about. Second, you: every tool in this chapter appears in almost every C++ program ever written, and Chapters 14 and 15 build on all of them. Third,

anyone who will ever maintain the program: C++ code that uses these tools is shorter, and short code has fewer places for mistakes to hide.

13.2 The Idea in Plain Words

13.2.1 One family, two dialects

In 1979, Bjarne Stroustrup, working at Bell Labs where C had been created a few years earlier, began adding *classes* to C so that he could write large simulation programs in a more organized way. He called the result "C with Classes"; in 1983 it was renamed C++, a programmer's joke, because `++` is C's increment operator: C++ is "C, one step further" [1]. The language has been standardized by the International Organization for Standardization (ISO) since 1998, and a new edition of the standard appears every three years: C++11, C++14, C++17, C++20, C++23. The committee finished the technical work on the next edition, C++26, at its meeting in March 2026 [2].

This book uses **C++17** throughout. It is supported completely by every compiler you are likely to meet, it contains every tool this chapter and the next two need, and it is still the version that most existing code is written in. Newer compilers default to newer versions (GCC 16.1, released in April 2026, compiles C++20 by default [3]), which is why we always write `-std=c++17` explicitly: the same command then gives the same result on every machine.

C++ is *almost* a superset of C. Nearly every C program is also a C++ program, and the compiler that you installed in Chapter 1 (`gcc` from MSYS2 on Windows, the Xcode command-line tools on macOS, `build-essential` on Linux) also contains the C++ compiler, called `g++`. There are a few small differences (for example, C++ does not allow assigning the result of `malloc` to an `int*` without a cast, and it has extra keywords such as `class`, `new` and `bool`), but they rarely matter. What matters is that C++ *adds*, and that the additions let you write the same program in a safer way.

13.2.2 What "better C" means

Table 13.1 is the whole chapter in one table: on the left the C way you know, in the middle the C++ way you will learn, and on the right what Python, the second language of this book, does for the same job.

Table 13.1 The migration table: the same jobs in C, in C++ and in Python.

Job	C (Chapters 1–12)	C++ (this chapter)	Python
Print	`printf("%d kg\n", w);`	`std::cout << w << " kg\n";`	`print(f"{w} kg")`
Read	`scanf("%d", &w);`	`std::cin >> w;`	`w = int(input())`
Text	`char name[31];`	`std::string name;`	`name = ""`
A list of values	`int grams[100];`	`std::vector<int> grams;`	`grams = []`
Change the caller's variable	`void f(int* p)`, call `f(&x)`	`void f(int& p)`, call `f(x)`	return a new value
Several functions, one job	`printInt`, `printText`	overloads of `print`	one function, any type
Optional argument	not possible	default argument	default argument
Allocate / free	`malloc` / `free`	`new` / `delete` (or a `vector`)	automatic
True and false	`int` 1 and 0	`bool`: `true`, `false`	`True`, `False`

Both a short tour of the language by its creator [4] and a classic textbook [5] follow the same path from C to C++. Read the table once now, and come back to it at the end of the chapter; by then every row will be familiar.

13.2.3 Compiling a C++ program

C++ source files end in `.cpp`. You compile them with `g++` exactly as you compiled C with `gcc`, and we add the same warning flags as always:

```
g++ -std=c++17 -O2 -Wall -Wextra hello_laundry.cpp -o hello_laundry
./hello_laundry
```

On Windows in the MSYS2 terminal, or in PowerShell once MSYS2's `ucrt64\bin` folder is on the `PATH` (Chapter 1), the commands are the same and the program is called `hello_laundry.exe`. On macOS, `g++` is a second name for Apple's `clang++`, which accepts the same flags. In Visual Studio Code, the C/C++ extension that you installed in Chapter 1 handles `.cpp` files as well; the build task only needs `g++` instead of `gcc`.

The code of this chapter lives in `FP/Code_new/Chapter13`, with the C++ programs in `cpp/`, the Python twins in `python/`, input files in `input/`, the expected output of every program in `expected/`, and a test program in `tests/` (Section 13.11). All commands below are typed inside the `Chapter13` folder.

13.3 Streams: `cout` and `cin`

Our first C++ program asks for a customer's name and the weight of the laundry and prints the price. **Listing 13.1** is the whole program.

Listing 13.1 `hello_laundry.cpp`: reading a name and a weight with `cin` and printing with `cout`.

```cpp
// hello_laundry.cpp - Chapter 13: a first C++ program for Profitina Laundry.
// Reads a customer's name and the weight of the laundry in grams, prints the price.
#include <iostream>
#include <string>

int main() {
    const long long ratePerKg = 7000;          // wash and fold, rupiah per kg
    std::string name;                          // grows to fit any name
    long long grams = 0;

    std::cout << "Customer name: ";
    std::getline(std::cin, name);              // a whole line, spaces included
    std::cout << "Weight in grams: ";
    std::cin >> grams;                         // no format string, no &

    long long price = (grams * ratePerKg + 500) / 1000;   // rounded to 1 rupiah
    std::cout << "\nHello, " << name << "!\n";
    std::cout << grams << " g of laundry costs Rp " << price << ".\n";
    return 0;
}
```

Line by line, the new things are these.

`#include <iostream>` brings in the input/output *streams*. A stream is a flow of characters: `std::cout` ("character output") flows to the screen, `std::cin` ("character input") flows from the keyboard. The header names of C++'s own library have no `.h`; the C headers are still available under names that start with `c`, such as `<cstdio>` and `<cmath>`.

`std::` is a *namespace* prefix. All of the C++ standard library lives in a namespace called `std` ("standard"), so that its names cannot clash with yours: you may write your own function called `sort` without hiding `std::sort`. Many books write `using namespace std;` at the top and then leave the prefix out. That is convenient in a tiny program and a source of confusing clashes in a large one, so in this book we write the prefix every time.

`std::cout << "Customer name: ";` sends a string to the screen. The operator `<<` is called the *insertion* operator here; think of it as an arrow that pushes a value into the stream. Several values can be pushed in one statement, `std::cout << grams << " g of laundry costs Rp " << price << ".\n";`, and each value is printed in the right way *for its type*: `grams` as an integer, the text as text. There is no format string, so there is no way to write `%d` for a `double` or to forget an argument. That is what "type-safe" means: the compiler knows the type of every value and chooses the printing code itself.

`std::cin >> grams;` reads a value from the keyboard into `grams`. The operator `>>` (*extraction*) points the other way: from the stream into the variable. Notice what is missing: there is no `&`. `cin` receives the variable itself, by *reference*, a mechanism you will meet in Section 13.6. Like `scanf("%lld")`, it skips leading spaces and reads one number.

`std::getline(std::cin, name);` reads a whole line, spaces included, into a `std::string`. The `>>` operator would stop at the first space and read only "Bu", which is why names are read with `getline`.

The price is computed with integer arithmetic only. A weight g in grams at a rate r in rupiah per kilogram costs $g \cdot r/1000$ rupiah; to round to the nearest rupiah instead of cutting off, we add half of the divisor before the integer division:

$$P = \left\lfloor \frac{g \cdot r + 500}{1000} \right\rfloor \tag{13.1}$$

Here $\lfloor x \rfloor$ ("floor of x") is the largest whole number not greater than x, which is exactly what C and C++ integer division gives for positive numbers. For $g = 3450$ and $r = 7000$: $3450 \times 7000 = 24{,}150{,}000$; adding 500 and dividing by 1000 gives 24,150 (the fraction .5 is dropped). We avoid `double` on purpose: money that is computed with floating point can come out as 24149.999999 and print as 24149, and a cashier must never be off by a rupiah because of the way a computer stores fractions.

Listing 13.2 is the same program in Python. `input(prompt)` prints the prompt and reads a line, and `int(...)` turns the text into a number; the operator `//` is Python's integer division, which rounds toward minus infinity (the same as C++ for the positive numbers we use).

Listing 13.2 `hello_laundry.py`: the same program in Python.

```python
# hello_laundry.py - Chapter 13: a first program for Profitina Laundry.
# Reads a customer's name and the weight of the laundry in grams, prints the price.

RATE_PER_KG = 7000                          # wash and fold, rupiah per kg

name = input("Customer name: ")
grams = int(input("Weight in grams: "))

price = (grams * RATE_PER_KG + 500) // 1000    # rounded to 1 rupiah
print(f"\nHello, {name}!")
print(f"{grams} g of laundry costs Rp {price}.")
```

To test the program without typing, we put the two input lines in the file `input/hello.txt` ("Bu Rina Wulandari" and "3450") and *redirect* the keyboard to that file with `<`:

```
$ g++ -std=c++17 -O2 -Wall -Wextra cpp/hello_laundry.cpp -o hello_laundry
$ ./hello_laundry < input/hello.txt
$ python3 python/hello_laundry.py < input/hello.txt
```

Both print exactly this:

```
Customer name: Weight in grams:
Hello, Bu Rina Wulandari!
3450 g of laundry costs Rp 24150.
```

The two prompts appear on one line because the typed answers, which would normally end each line, come from the file and are not shown on the screen. Typed by hand, the conversation looks as you would expect.

> **TRAP.** After `std::cin >> grams;` the newline that ended the number is still waiting in the stream. A `std::getline` that follows immediately reads that empty rest of the line and returns an empty string. If you must read a line after a number, first skip the rest of the number's line with `std::cin.ignore(1000, '\n');`. In **Listing 13.1** we avoid the problem by reading the line first.

13.4 `std::string`: Text That Fits

In C, a string is an array of `char` that ends with the character `'\0'`. You must decide the size in advance, copy with `strcpy`, join with `strcat`, compare with `strcmp`, and remember that every one of them will happily write past the end of the array. `std::string` is a *class* (Chapter 14 explains the word) that holds text of any length and manages its own memory [6]. **Listing 13.3** shows the operations you need most.

Listing 13.3 `strings_demo.cpp`: the everyday operations of `std::string`.

```cpp
// strings_demo.cpp - Chapter 13: std::string instead of char arrays.
#include <cctype>
#include <iostream>
#include <string>

int main() {
    std::string first = "Rina";
    std::string last = "Wulandari";
    std::string full = first + " " + last;     // + joins strings, no strcat
    std::cout << "Full name : " << full << '\n';
    std::cout << "Length    : " << full.size() << " characters\n";
    std::cout << "First char: " << full[0] << '\n';

    std::string upper = full;                  // a real copy, not a pointer
    for (char& c : upper)                      // c is a reference into upper
        c = static_cast<char>(std::toupper(static_cast<unsigned char>(c)));
    std::cout << "Upper case: " << upper << '\n';
    std::cout << "Unchanged : " << full << '\n';

    std::size_t space = full.find(' ');
    std::cout << "Space at  : " << space << '\n';
    std::cout << "Surname   : " << full.substr(space + 1) << '\n';

    std::string code = "PL-" + std::to_string(2026) + "-" + first.substr(0, 3);
    std::cout << "Order code: " << code << '\n';

    std::string a = "Ani", b = "Andi";
    std::cout << "Compare   : " << a << (a < b ? " < " : " > ") << b << '\n';
    if (full.find("Sari") == std::string::npos)
        std::cout << "Search    : no 'Sari' in the name\n";
    return 0;
}
```

The output:

```
Full name : Rina Wulandari
Length    : 14 characters
First char: R
Upper case: RINA WULANDARI
Unchanged : Rina Wulandari
Space at  : 4
Surname   : Wulandari
Order code: PL-2026-Rin
Compare   : Ani > Andi
Search    : no 'Sari' in the name
```

Go through it with the C versions in mind.

- `first + " " + last` joins strings with `+`. The result is a new string exactly as long as it needs to be; there is no buffer to overflow. `+=` appends to an existing string.
- `full.size()` is the number of characters (14), computed instantly, because the string remembers its own length; C's `strlen` has to walk to the `'\0'`.
- `full[0]` is the first character, as in C. `full.at(0)` does the same but checks the index and reports an error if it is outside the string.
- `std::string upper = full;` makes a real *copy*. In C, `char* upper = full;` would only copy the address, and changing `upper` would change `full`. The line "Unchanged" proves that the original is untouched.
- `for (char& c : upper)` is a *range-based for loop*: it runs once for each character of `upper`. The `&` makes `c` a reference to the character inside the string, so assigning to `c` changes the string itself (Section 13.6 explains references). `std::toupper` comes from `<cctype>`, the C++ name of C's

`<ctype.h>`; the two `static_cast`s are the careful way to call it, because characters above 127 would otherwise be passed as negative numbers, which `toupper` does not accept.

- `full.find('  ')` returns the position of the first space (4), or the special value `std::string::npos` ("no position") when there is none. `full.substr(4 + 1)` is the part from position 5 to the end; `first.substr(0, 3)` is 3 characters starting at position 0.
- `std::to_string(2026)` turns a number into text, the job `sprintf` did in C.
- `a < b` compares two strings in dictionary order by character codes, like `strcmp(a, b) < 0`. "Ani" is greater than "Andi" because at the third position `'i'` (code 105) is greater than `'d'` (code 100).

Listing 13.4 does the same in Python, where `str` has always behaved like `std::string`. One difference is worth noticing: Python strings are *immutable*, they can never be changed after they are made, so `full.upper()` returns a new string instead of changing characters in place. Slicing, `full[space + 1:]`, plays the role of `substr`, and `find` returns −1 instead of `npos`.

Listing 13.4 `strings_demo.py`: the same operations with Python's `str`.

```
# strings_demo.py - Chapter 13: the same string operations with Python's str.

first = "Rina"
last = "Wulandari"
full = first + " " + last
print(f"Full name : {full}")
print(f"Length    : {len(full)} characters")
print(f"First char: {full[0]}")

upper = full.upper()                          # str is immutable: a new string
print(f"Upper case: {upper}")
print(f"Unchanged : {full}")

space = full.find(" ")
print(f"Space at  : {space}")
print(f"Surname   : {full[space + 1:]}")

code = "PL-" + str(2026) + "-" + first[0:3]
print(f"Order code: {code}")

a, b = "Ani", "Andi"
print(f"Compare   : {a}{' < ' if a < b else ' > '}{b}")
if full.find("Sari") == -1:
    print("Search    : no 'Sari' in the name")
```

13.5 `std::vector`: An Array That Grows

The second everyday tool replaces the fixed-size array. A `std::vector<int>` is a list of `int`s that knows its own size, can grow at the end, frees its memory by itself and can check its indices. The part between the angle brackets is the type of the elements: `std::vector<double>`, `std::vector<std::string>`, and later `std::vector<Order>` all work the same way. **Listing 13.5** keeps the weights of a day's orders.

Listing 13.5 `vector_demo.cpp`: a day's weights in a `std::vector`.

```cpp
// vector_demo.cpp - Chapter 13: std::vector instead of fixed arrays.
#include <algorithm>
#include <iostream>
#include <stdexcept>
#include <vector>

int main() {
    std::vector<int> grams = {3450, 1200, 5000, 2750};
    grams.push_back(4100);                      // the vector grows by itself
    std::cout << "Orders today : " << grams.size() << '\n';

    long long total = 0;
    for (int g : grams)                         // range-based for: every element
        total += g;
    std::cout << "Total weight : " << total << " g\n";
    std::cout << "Heaviest     : "
              << *std::max_element(grams.begin(), grams.end()) << " g\n";

    std::sort(grams.begin(), grams.end());      // the library sorts for us
    std::cout << "Sorted       :";
    for (int g : grams) std::cout << ' ' << g;
    std::cout << '\n';

    std::cout << "grams.at(2)  : " << grams.at(2) << '\n';
    try {
        std::cout << grams.at(9) << '\n';       // at() checks the index
    } catch (const std::out_of_range&) {
        std::cout << "grams.at(9)  : refused, there are only " << grams.size()
                  << " orders\n";
    }
    return 0;
}
```

```
Orders today : 5
Total weight : 16500 g
Heaviest     : 5000 g
Sorted       : 1200 2750 3450 4100 5000
grams.at(2)  : 3450
grams.at(9)  : refused, there are only 5 orders
```

`push_back` appends an element; the vector finds room for it by itself, moving to a larger block of memory when needed (it grows by a factor, so that appending n elements costs time proportional to n in total). `grams.size()` is the current number of elements. The range-based for loop visits every element without an index, so there is no index to get wrong.

Two functions come from `<algorithm>`, the header of ready-made algorithms. `std::max_element` returns a position (an *iterator*, a generalized pointer) of the largest element, and the `*` in front of it reads the value there, just as `*p` reads the value a pointer points to. `std::sort(grams.begin(), grams.end())` sorts the whole vector; `begin()` and `end()` mark the first element and the position *just after* the last, the same "half-open" convention as a C loop `for (i = 0; i < n; i++)`. You wrote bubble sort by hand in Chapter 6, and it is worth having done so once; from now on you may use `std::sort`, which is much faster on large data.

`grams[9]` would behave as in C: no check, and reading past the end is undefined behavior. `grams.at(9)` checks the index and, when it is out of range, *throws an exception* of type `std::out_of_range`. An exception is a way for a function to say "I cannot do this" that the caller cannot ignore: the normal flow stops, and control jumps to the nearest `catch` block that accepts that kind of exception. Here the `catch` prints a polite message. Chapter 14 uses exceptions to refuse invalid orders; for now it is enough to know that `at` turns a silent memory error into a clear, catchable one.

Listing 13.6 is the Python version. A Python `list` has always been a growing array, and it *always* checks indices, so `grams[9]` raises an `IndexError` that is caught in the same way.

Listing 13.6 `vector_demo.py`: Python's `list` does what `std::vector` does.

```
# vector_demo.py - Chapter 13: Python's list does what std::vector does.

grams = [3450, 1200, 5000, 2750]
grams.append(4100)                              # the list grows by itself
print(f"Orders today : {len(grams)}")

total = 0
for g in grams:
    total += g
print(f"Total weight : {total} g")
print(f"Heaviest     : {max(grams)} g")

grams.sort()
print("Sorted       :" + "".join(f" {g}" for g in grams))

print(f"grams.at(2)  : {grams[2]}")
try:
    print(grams[9])                             # Python always checks the index
except IndexError:
    print(f"grams.at(9)  : refused, there are only {len(grams)} orders")
```

13.6 References: Changing the Caller's Variable Without Pointers

In Chapter 7 you learned that a C function receives *copies* of its arguments, and that to let a function change the caller's variable you must pass its address and write `*p` everywhere inside. C++ keeps that and adds a third way. **Figure 13.1** shows all three side by side.

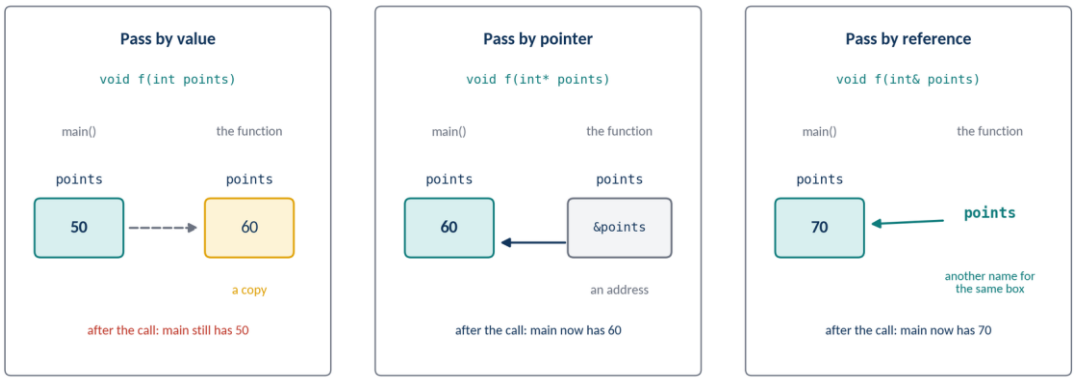


Figure 13.1 Three ways to pass `points`: a value parameter gets a copy, a pointer parameter gets the address, and a reference parameter is another name for the caller's own variable.

> **DEFINITION.** A *reference* is another name for an existing variable. It is declared with `&` after the type, `int& alias = points;`, and from then on `alias` and `points` are the same box in memory: reading or writing one reads or writes the other. A reference must be initialized when it is declared, it can never be made to refer to a different variable later, and there is no "null reference".

A *reference parameter* such as `int& points` becomes, at each call, another name for the argument the caller passed. The function writes `points += 10;` with no `*`, and the caller writes `addBonusByReference(points);` with no `&`. Under the hood the compiler usually passes an address, exactly as with a pointer; the difference is that the language, not the programmer, takes

care of the `&` and the `*`. **Listing 13.7** runs the three ways one after another on Bu Rina's loyalty points.

Listing 13.7 `references.cpp`: value, pointer and reference parameters, and a function with two results.

```
// references.cpp - Chapter 13: pass by value, by pointer and by reference.
#include <iostream>
#include <vector>

void addBonusByValue(int points) { points = points + 10; }   // a copy changes
void addBonusByPointer(int* points) { *points += 10; }       // C style
void addBonusByReference(int& points) { points += 10; }      // C++ style

// Two results at once: the function fills the caller's variables.
void summarize(const std::vector<int>& grams, long long& total, int& heaviest) {
    total = 0;
    heaviest = 0;
    for (int g : grams) {
        total += g;
        if (g > heaviest) heaviest = g;
    }
}

int main() {
    int points = 50;
    addBonusByValue(points);
    std::cout << "by value     : " << points << '\n';
    addBonusByPointer(&points);
    std::cout << "by pointer   : " << points << '\n';
    addBonusByReference(points);
    std::cout << "by reference : " << points << '\n';

    int& alias = points;                        // another name for the same box
    alias = 99;
    std::cout << "alias = 99   : points is now " << points << '\n';

    std::vector<int> grams = {3450, 1200, 5000, 2750};
    long long total = 0;
    int heaviest = 0;
    summarize(grams, total, heaviest);          // const& : no copy, no change
    std::cout << "summary      : " << total << " g in total, heaviest "
              << heaviest << " g\n";
    return 0;
}
```

```
by value     : 50
by pointer   : 60
by reference : 70
alias = 99   : points is now 99
summary      : 12400 g in total, heaviest 5000 g
```

The value version changes only its own copy (50 stays 50); the pointer version and the reference version both change the caller's variable. `int& alias = points;` shows that a reference is not only for parameters: after it, assigning 99 to `alias` puts 99 in `points`.

The function `summarize` shows the two most common uses of references in real code.

- **Output parameters.** `long long& total` and `int& heaviest` let one function return two results by filling the caller's variables. (Chapter 7 did this with pointers.)
- **Const references for input.** `const std::vector<int>& grams` passes the vector *without copying it*, which matters when it holds a million orders, and the `const` promises that the function will not change it. The compiler enforces the promise: `grams.push_back(1);` inside

`summarize` would not compile. Passing large objects as `const&` is the everyday default in C++, and the C++ Core Guidelines recommend it for every parameter that is only read [7]: `const std::string&` in the cashier below is the same idea.

When should you use a pointer and when a reference? Use a reference when there is always exactly one object to refer to, which is almost always the case for parameters. Use a pointer when "no object" is a meaningful answer (a pointer can be `nullptr`, C++'s spelling of `NULL`), or when you must be able to point at different objects over time, for example when walking a linked list as in Chapter 10.

Python has no references in this sense, and **Listing 13.8** shows what it does instead. Every Python variable is a name *bound* to an object. Passing an argument binds the parameter name to the same object; if the function then rebinds the name (`points = points + 10`), only its local name changes. An `int` can never be changed in place, so a function cannot change the caller's integer. A list *can* be changed in place, so a list with one element works like a box that both sides can see. The Pythonic answer to "two results" is simpler still: return a tuple, `return total, heaviest`, and unpack it at the call.

Listing 13.8 `references.py`: what Python does instead of references.

```
# references.py - Chapter 13: what Python does instead of references.
# An int cannot be changed inside a function; a list (a mutable box) can.


def add_bonus_by_value(points):
    points = points + 10                        # rebinds the local name only


def add_bonus_in_box(box):
    box[0] += 10                                # changes the list the caller sees


def summarize(grams):
    """Two results at once: Python returns a tuple."""
    return sum(grams), max(grams)


points = 50
add_bonus_by_value(points)
print(f"by value     : {points}")
box = [points]
add_bonus_in_box(box)
print(f"by pointer   : {box[0]}")
add_bonus_in_box(box)
print(f"by reference : {box[0]}")

alias = box                                     # another name for the same list
alias[0] = 99
print(f"alias = 99   : points is now {box[0]}")

grams = [3450, 1200, 5000, 2750]
total, heaviest = summarize(grams)
print(f"summary      : {total} g in total, heaviest {heaviest} g")
```

13.7 Overloading and Default Arguments

In C every function needs its own name. If you want to print a text, a labeled amount and a ruler line, you write `printText`, `printAmount` and `printRule`. C++ lets all three share one name, because the compiler can tell them apart by their *parameter lists*. This is called *function overloading*.

> **DEFINITION.** Two functions are *overloads* of each other when they have the same name and different parameter lists (a different number of parameters, or different types). At each call, the compiler chooses the overload whose parameters match the arguments best; this is *overload resolution*, and it happens entirely at compile time. The return type alone cannot distinguish overloads.

A *default argument* is a value written in the declaration, `bool express = false`, that the compiler supplies when a caller leaves the argument out. Defaults must be the rightmost parameters, because arguments are matched from the left: `charge(3450, 7000)` means `charge(3450, 7000, false)`. **Listing 13.9** uses both.

Listing 13.9 `overload_default.cpp`: three overloads of `printLine` and a default argument in `charge`.

```
// overload_default.cpp - Chapter 13: one name, several functions, and a default.
#include <iostream>
#include <string>

// Three functions called printLine: the compiler picks one by the arguments.
void printLine(const std::string& text) { std::cout << text << '\n'; }

void printLine(const std::string& label, long long rupiah) {
    std::cout << label << ": Rp " << rupiah << '\n';
}

void printLine(char c, int count) { std::cout << std::string(count, c) << '\n'; }

// A default argument: a caller may leave out 'express'.
long long charge(int grams, long long ratePerKg, bool express = false) {
    long long price = (grams * ratePerKg + 500) / 1000;
    if (express) price += price / 2;          // express costs 50% more
    return price;
}

int main() {
    printLine('=', 30);
    printLine("PROFITINA LAUNDRY");
    printLine('=', 30);
    printLine("Wash 3450 g        ", charge(3450, 7000));
    printLine("Wash 3450 g express", charge(3450, 7000, true));
    printLine("Iron 2000 g        ", charge(2000, 5000));
    printLine('-', 30);
    return 0;
}
```

```
==============================
PROFITINA LAUNDRY
==============================
Wash 3450 g        : Rp 24150
Wash 3450 g express: Rp 36225
Iron 2000 g        : Rp 10000
------------------------------
```

For each call to `printLine` the compiler looks at the arguments. `printLine('=', 30)` has a `char` and an `int`, so only the third overload fits. `printLine("PROFITINA LAUNDRY")` has one argument, so only the first one fits. `printLine("Wash 3450 g        ", charge(3450, 7000))` has a text and a `long long`, which matches the second. The first and third overloads both contain `std::string(count, c)`-style work in other programs; overloading lets the *caller* think of "print a line" as one idea.

The express surcharge is 50% of the regular price, rounded down, so the express price of a regular price P is

$$P_{\text{express}} = P + \left\lfloor \frac{P}{2} \right\rfloor \tag{13.2}$$

For 3450 g of washing, $P = 24{,}150$ by Equation (13.1), and $P_{\text{express}} = 24{,}150 + 12{,}075 = 36{,}225$, which is the second amount in the output.

How can two functions have the same name when the linker needs a unique name for each? The compiler quietly *decorates* each name with its parameter types (this is called *name mangling*): `printLine(char, int)` becomes something like `_Z9printLineci`. That is also why a C++ program that calls a function compiled by a C compiler must declare it `extern "C"`, which switches the decoration off.

Overloading has one danger: the compiler may find two overloads that fit equally well. Then the call is *ambiguous* and the program does not compile. Chapter 16 (Problem 16.2) meets exactly this trap when overloads and default arguments are mixed.

Python has default arguments [8], but no overloading: a second `def printLine` would simply *replace* the first. **Listing 13.10** therefore gives the three printing functions three names. (A Python function can also accept "anything" and look at the types of its arguments with `isinstance`, but separate names are clearer.)

Listing 13.10 `overload_default.py`: default arguments in Python, and three names instead of overloads.

```
# overload_default.py - Chapter 13: Python has default arguments but no
# overloading; a second 'def' with the same name would replace the first.
# So the three C++ printLine functions get three names here.


def print_text(text):
    print(text)


def print_amount(label, rupiah):
    print(f"{label}: Rp {rupiah}")


def print_rule(c, count):
    print(c * count)


def charge(grams, rate_per_kg, express=False):
    price = (grams * rate_per_kg + 500) // 1000
    if express:
        price += price // 2                    # express costs 50% more
    return price


print_rule("=", 30)
print_text("PROFITINA LAUNDRY")
print_rule("=", 30)
print_amount("Wash 3450 g        ", charge(3450, 7000))
print_amount("Wash 3450 g express", charge(3450, 7000, True))
print_amount("Iron 2000 g        ", charge(2000, 5000))
print_rule("-", 30)
```

13.8 `new` and `delete`

C allocates memory at run time with `malloc` and frees it with `free`. C++ has two operators for the same job: `new` creates an object (or an array of objects) and returns a pointer to it, and `delete` destroys it. Compared with `malloc`, `new` knows the type, so there is no `sizeof` to compute and no cast to write, and it can initialize the object at once. **Listing 13.11** shows both forms and then the better alternative.

Listing 13.11 `new_delete.cpp`: `new[]` with `delete[]`, `new` with `delete`, and a `vector` that needs neither.

```
// new_delete.cpp - Chapter 13: new and delete, and why vector is usually better.
#include <iostream>
#include <vector>

int main() {
    int n = 4;
    int* grams = new int[n]{3450, 1200, 5000, 2750};   // typed, no sizeof, no cast
    long long total = 0;
    for (int i = 0; i < n; ++i) total += grams[i];
    std::cout << "new[]  : " << n << " orders, " << total << " g\n";
    delete[] grams;                                    // new[] pairs with delete[]
    grams = nullptr;                                   // no dangling pointer left behind

    int* one = new int(42);                            // one int, set to 42
    std::cout << "new    : *one = " << *one << '\n';
    delete one;                                        // new pairs with delete

    std::vector<int> v = {3450, 1200, 5000, 2750};     // frees itself
    v.push_back(4100);
    total = 0;
    for (int g : v) total += g;
    std::cout << "vector : " << v.size() << " orders, " << total << " g\n";
    return 0;
}
```

```
new[]  : 4 orders, 12400 g
new    : *one = 42
vector : 5 orders, 16500 g
```

The rule you must never break is that the forms are *paired*: memory from `new` is freed with `delete`, memory from `new[]` with `delete[]`, and memory from `malloc` with `free`. Mixing them (`delete` on an array, `free` on something from `new`) is undefined behavior; it may work on one machine and corrupt memory on another. After `delete`, the pointer still holds the old address, a *dangling pointer*; setting it to `nullptr` makes any later mistake visible at once.

The last part of the program is the real lesson: a `std::vector` does everything the `new[]` array did, grows when needed, and frees its memory by itself at the end of `main`. In modern C++ you write `new` and `delete` rarely; you use containers such as `vector` and `string`, and, when you do need a single object on the heap, *smart pointers* such as `std::unique_ptr`, which call `delete` for you. Chapter 14 shows the mechanism behind this: the *destructor*.

Python never asks you to free memory. Every object is created when needed and freed automatically when no name refers to it any more (**Listing 13.12**).

Listing 13.12 `new_delete.py`: Python frees memory by itself.

```python
# new_delete.py - Chapter 13: in Python every object is created on demand and
# freed automatically when nothing refers to it any more; there is no delete.

grams = [3450, 1200, 5000, 2750]
print(f"new[]  : {len(grams)} orders, {sum(grams)} g")
grams = None                                    # the list can now be freed

one = 42
print(f"new    : *one = {one}")

v = [3450, 1200, 5000, 2750]
v.append(4100)
print(f"vector : {len(v)} orders, {sum(v)} g")
```

13.9 Building It: Cashier v1.0

Now we rebuild Bu Rina's cashier with every tool of this chapter. The cashier reads a day's orders from a text file, one order per line, in the form `customer;service;grams;express`:

```
# customer;service;grams;express
Bu Rina Wulandari;wash;3450;no
Pak Budi Santoso;washiron;5200;yes
Mbak Sari Dewi;iron;2000;no
Mas Joko Prasetyo;wash;4100;yes
Bu Rina Wulandari;iron;1250;no
Pak Hendra Gunawan;washiron;2750;no
```

Lines that start with `#` are comments. There are three services, each with a rate per kilogram: `wash` (wash and fold, Rp 7,000), `washiron` (wash and iron, Rp 10,000) and `iron` (iron only, Rp 5,000). The program must check every line, print one row per accepted order, explain every refused line, and finish with totals and a summary per service. **Listing 13.13** is the complete program.

Listing 13.13 `cashier.cpp`: Profitina Laundry cashier v1.0 in C++.

```cpp
// cashier.cpp - Chapter 13: Profitina Laundry cashier v1.0, the C cashier
// rewritten with the tools of C++: string, vector, references, overloading,
// default arguments, ifstream and iomanip.
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

struct Service {
    std::string name;
    long long ratePerKg;
};

struct Order {
    std::string customer;
    std::string service;
    int grams;
    bool express;
    long long price;
};

const std::vector<Service> SERVICES = {
    {"wash", 7000}, {"washiron", 10000}, {"iron", 5000}};

// Splits "a;b;c" into {"a", "b", "c"}.
std::vector<std::string> split(const std::string& line, char sep) {
    std::vector<std::string> parts;
    std::string current;
    for (char c : line) {
        if (c == sep) {
            parts.push_back(current);
            current.clear();
        } else {
            current += c;
        }
    }
    parts.push_back(current);
    return parts;
}
```

Listing 13.13 (Continued) *cashier.cpp: Profitina Laundry cashier v1.0 in C++.*

```
// Finds the rate of a service; the answer comes back through a reference.
bool findRate(const std::string& service, long long& rate) {
    for (const Service& s : SERVICES)
        if (s.name == service) {
            rate = s.ratePerKg;
            return true;
        }
    return false;
}

// Accepts only 1 to 6 digits and a value from 1 to 50000 grams.
bool parseGrams(const std::string& text, int& grams) {
    if (text.empty() || text.size() > 6) return false;
    int value = 0;
    for (char c : text) {
        if (c < '0' || c > '9') return false;
        value = value * 10 + (c - '0');
    }
    if (value < 1 || value > 50000) return false;
    grams = value;
    return true;
}

long long charge(int grams, long long ratePerKg, bool express = false) {
    long long price = (grams * ratePerKg + 500) / 1000;
    if (express) price += price / 2;           // express costs 50% more
    return price;
}

// Two overloads: a weight in grams becomes "3.450 kg".
std::string kg(long long grams) {
    std::string frac = std::to_string(grams % 1000);
    return std::to_string(grams / 1000) + "." + std::string(3 - frac.size(), '0')
           + frac + " kg";
}
std::string kg(int grams) { return kg(static_cast<long long>(grams)); }
```

Listing 13.13 (Continued) *cashier.cpp: Profitina Laundry cashier v1.0 in C++.*

```
// Reads one line; on success fills 'order', otherwise fills 'why'.
bool parseOrder(const std::string& line, Order& order, std::string& why) {
    std::vector<std::string> f = split(line, ';');
    long long rate = 0;
    if (f.size() != 4) {
        why = "expected 4 fields, found " + std::to_string(f.size());
        return false;
    }
    if (f[0].empty()) { why = "customer name is empty"; return false; }
    if (!findRate(f[1], rate)) {
        why = "unknown service '" + f[1] + "'";
        return false;
    }
    if (!parseGrams(f[2], order.grams)) {
        why = "weight '" + f[2] + "' is not 1..50000 grams";
        return false;
    }
    if (f[3] != "yes" && f[3] != "no") {
        why = "express must be yes or no, not '" + f[3] + "'";
        return false;
    }
    order.customer = f[0];
    order.service = f[1];
    order.express = (f[3] == "yes");
    order.price = charge(order.grams, rate, order.express);
    return true;
}

void printOrder(int number, const Order& o) {
    std::cout << std::right << std::setw(2) << number << ' ' << std::left
              << std::setw(20) << o.customer << ' ' << std::setw(9) << o.service
              << std::right << std::setw(10) << kg(o.grams) << "  Rp"
              << std::setw(7) << o.price << (o.express ? "  express" : "") << '\n';
}

int main(int argc, char* argv[]) {
    std::string path = argc > 1 ? argv[1] : "input/orders.txt";
    std::ifstream in(path);
    if (!in) {
        std::cout << "cannot open " << path << '\n';
        return 1;
    }
    std::cout << "PROFITINA LAUNDRY - cashier v1.0 (" << path << ")\n\n";
```

Listing 13.13 (Continued) *cashier.cpp: Profitina Laundry cashier v1.0 in C++.*

```
    std::vector<Order> orders;
    std::string line, why;
    int lineNo = 0, refused = 0;
    while (std::getline(in, line)) {
        ++lineNo;
        if (line.empty() || line[0] == '#') continue;
        Order o;
        if (parseOrder(line, o, why)) {
            orders.push_back(o);
            printOrder(static_cast<int>(orders.size()), o);
        } else {
            ++refused;
            std::cout << "   line " << lineNo << " refused: " << why << '\n';
        }
    }

    long long totalGrams = 0, revenue = 0;
    int express = 0;
    for (const Order& o : orders) {
        totalGrams += o.grams;
        revenue += o.price;
        if (o.express) ++express;
    }
    std::cout << "\nOrders accepted : " << orders.size() << '\n'
              << "Orders refused  : " << refused << '\n'
              << "Express orders  : " << express << '\n'
              << "Total weight    : " << kg(totalGrams) << '\n'
              << "Revenue         : Rp " << revenue << "\n\nPer service\n";
    for (const Service& s : SERVICES) {
        int count = 0;
        long long grams = 0, money = 0;
        for (const Order& o : orders)
            if (o.service == s.name) {
                ++count;
                grams += o.grams;
                money += o.price;
            }
        std::cout << "  " << std::left << std::setw(9) << s.name << std::right
                  << std::setw(2) << count << " orders" << std::setw(11)
                  << kg(grams) << "  Rp" << std::setw(7) << money << '\n';
    }
    return 0;
}
```

Let us walk through it from the top.

The data. `struct Service` and `struct Order` are the structures of Chapter 10, with two C++ differences: a `std::string` member instead of a `char` array, and no `typedef` needed, because in C++ the name of a structure is a type name by itself. `SERVICES` is a `const std::vector<Service>` initialized with a list in braces; `const` means that no part of the program can change the price list by accident.

`split`. This helper turns `"Bu Rina;wash;3450;no"` into a vector of four strings. It walks over the characters with a range-based for loop, collects them in `current`, and whenever it meets the separator it pushes `current` into the result and clears it. The last field is pushed after the loop. Notice what we do *not* need: no array size, no `strtok` (which changes the original string and cannot be used on two strings at once), and no check whether a field is too long.

`findRate` and `parseGrams`. Both return `bool` ("did it work?") and deliver their real result through a reference parameter, the output-parameter pattern of Section 13.6. `parseGrams` is deliberately strict: only the digits `0` to `9`, at most six of them, and a value from 1 to 50,000 grams (50 kg is more

than any bag). We do not use `std::stoi` here, because it would accept `"2,5"` as the number 2 and quietly ignore the rest; a cashier should refuse such a line and say why.

`charge`. This is Equation (13.1) and Equation (13.2) with a default argument, exactly as in Section 13.7.

`kg`. Two overloads turn grams into text such as `3.450 kg`: the whole kilograms, a dot, and the grams padded with zeros to three digits (`std::string(3 - frac.size(), '0')` is a string of that many zeros). One overload takes an `int` (a single order), the other a `long long` (a day's total); the `int` version simply calls the other one.

`parseOrder`. Here the pieces meet. The function receives the line, an `Order&` to fill and a `std::string&` in which to explain a failure. Each check returns at the first problem with a sentence a person can understand. Only when all checks pass are the fields copied into `order` and the price computed.

`printOrder`. Formatting uses *manipulators* from `<iomanip>`. `std::setw(n)` sets the width of the *next* value only; `std::left` and `std::right` choose the alignment and stay in force until changed. So `std::left << std::setw(20) << o.customer` prints the name in a 20-character column, padded on the right. This is the stream version of `printf("%-20s")`.

`main`. `int main(int argc, char* argv[])` is the same as in C; `argv[1]`, if present, is the name of the orders file, and otherwise `input/orders.txt` is used. `std::ifstream in(path);` opens the file for reading: `ifstream` is an input stream connected to a file, it is used exactly like `cin`, and it *closes the file by itself* when `in` goes out of scope at the end of `main`, so there is no `fclose` to forget. `if (!in)` asks whether opening worked. `while (std::getline(in, line))` reads until the end of the file. Accepted orders are pushed into `orders`, a `std::vector<Order>`, so there is no limit on the number of orders in a day. The totals at the end are computed with range-based loops over that vector.

Listing 13.14 is the same cashier in Python. The structure is identical; the differences are Python's own idioms: `line.split(";")` does the job of our `split`, an order is a dictionary, functions that may fail return `None` or a pair `(order, why)` instead of filling reference parameters, and the format specifications `:<20` and `:>7` in f-strings do what `setw`, `left` and `right` do in C++.

Listing 13.14 `cashier.py`: the same cashier in Python.

```python
# cashier.py - Chapter 13: Profitina Laundry cashier v1.0 in Python, printing
# exactly what cashier.cpp prints.
import sys

SERVICES = [("wash", 7000), ("washiron", 10000), ("iron", 5000)]


def find_rate(service):
    """Returns the rate of a service, or None when it is unknown."""
    for name, rate in SERVICES:
        if name == service:
            return rate
    return None


def parse_grams(text):
    """Accepts only 1 to 6 digits and a value from 1 to 50000 grams."""
    if not text or len(text) > 6 or any(c not in "0123456789" for c in text):
        return None
    value = int(text)
    return value if 1 <= value <= 50000 else None


def charge(grams, rate_per_kg, express=False):
    price = (grams * rate_per_kg + 500) // 1000
    if express:
        price += price // 2                    # express costs 50% more
    return price


def kg(grams):
    return f"{grams // 1000}.{grams % 1000:03d} kg"


def parse_order(line):
    """Returns (order, None) on success or (None, why) on failure."""
    f = line.split(";")
    if len(f) != 4:
        return None, f"expected 4 fields, found {len(f)}"
    if f[0] == "":
        return None, "customer name is empty"
    rate = find_rate(f[1])
    if rate is None:
        return None, f"unknown service '{f[1]}'"
    grams = parse_grams(f[2])
    if grams is None:
```

Listing 13.14 (Continued) *cashier.py: the same cashier in Python.*

```
        return None, f"weight '{f[2]}' is not 1..50000 grams"
    if f[3] not in ("yes", "no"):
        return None, f"express must be yes or no, not '{f[3]}'"
    express = f[3] == "yes"
    return {"customer": f[0], "service": f[1], "grams": grams,
            "express": express, "price": charge(grams, rate, express)}, None


def print_order(number, o):
    print(f"{number:>2} {o['customer']:<20} {o['service']:<9}{kg(o['grams']):>10}"
          f"  Rp{o['price']:>7}{'  express' if o['express'] else ''}")


def main():
    path = sys.argv[1] if len(sys.argv) > 1 else "input/orders.txt"
    try:
        lines = open(path, encoding="utf-8").read().splitlines()
    except OSError:
        print(f"cannot open {path}")
        return 1
    print(f"PROFITINA LAUNDRY - cashier v1.0 ({path})\n")

    orders, refused = [], 0
    for line_no, line in enumerate(lines, start=1):
        if line == "" or line.startswith("#"):
            continue
        order, why = parse_order(line)
        if order:
            orders.append(order)
            print_order(len(orders), order)
        else:
            refused += 1
            print(f"   line {line_no} refused: {why}")

    total_grams = sum(o["grams"] for o in orders)
    revenue = sum(o["price"] for o in orders)
    express = sum(1 for o in orders if o["express"])
    print(f"\nOrders accepted : {len(orders)}")
    print(f"Orders refused  : {refused}")
    print(f"Express orders  : {express}")
    print(f"Total weight    : {kg(total_grams)}")
    print(f"Revenue         : Rp {revenue}\n\nPer service")
    for name, _ in SERVICES:
        mine = [o for o in orders if o["service"] == name]
        grams = sum(o["grams"] for o in mine)
        money = sum(o["price"] for o in mine)
```

Listing 13.14 (Continued) *cashier.py: the same cashier in Python.*

```
        print(f"  {name:<9}{len(mine):>2} orders{kg(grams):>11}  Rp{money:>7}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

13.10 Running It

```
$ g++ -std=c++17 -O2 -Wall -Wextra cpp/cashier.cpp -o cashier
$ ./cashier
$ python3 python/cashier.py
```

Both programs print:

```
PROFITINA LAUNDRY - cashier v1.0 (input/orders.txt)

 1 Bu Rina Wulandari    wash       3.450 kg  Rp  24150
 2 Pak Budi Santoso     washiron   5.200 kg  Rp  78000  express
 3 Mbak Sari Dewi       iron       2.000 kg  Rp  10000
 4 Mas Joko Prasetyo    wash       4.100 kg  Rp  43050  express
 5 Bu Rina Wulandari    iron       1.250 kg  Rp   6250
 6 Pak Hendra Gunawan   washiron   2.750 kg  Rp  27500

Orders accepted : 6
Orders refused  : 0
Express orders  : 2
Total weight    : 18.750 kg
Revenue         : Rp 188950

Per service
  wash      2 orders   7.550 kg  Rp  67200
  washiron  2 orders   7.950 kg  Rp 105500
  iron      2 orders   3.250 kg  Rp  16250
```

Check one row by hand. Pak Budi's 5,200 g of wash-and-iron at Rp 10,000 per kg: Equation (13.1) gives $(5200 \times 10000 + 500)/1000 = 52{,}000$ (the $+500$ disappears in the integer division), and express adds half, $52{,}000 + 26{,}000 = 78{,}000$. The per-service summary adds the rows of each service: wash is $24{,}150 + 43{,}050 = 67{,}200$, and the three services together are $67{,}200 + 105{,}500 + 16{,}250 = 188{,}950$, the revenue of the day. Whenever a program prints totals, check at least one of them by hand; it is the cheapest test there is.

13.11 Testing It

Every program in this book comes with the output it must print, stored in the folder `expected/`. The test program `tests/run_tests.py` compiles each C++ program with `-Wall -Wextra -Werror` (a warning counts as a failure), runs it and its Python twin with the same input, and compares both outputs with the expected text character by character. The list of programs is a small text file, `tests/cases.txt`, one line per program:

```
# name | C++ sources | Python script | stdin | arguments
hello_laundry | hello_laundry.cpp | hello_laundry.py | input/hello.txt | -
strings_demo | strings_demo.cpp | strings_demo.py | - | -
vector_demo | vector_demo.cpp | vector_demo.py | - | -
references | references.cpp | references.py | - | -
overload_default | overload_default.cpp | overload_default.py | - | -
new_delete | new_delete.cpp | new_delete.py | - | -
cashier | cashier.cpp | cashier.py | - | -
cashier_bad | cashier.cpp | cashier.py | - | input/orders_bad.txt
```

Each line gives a name, the C++ sources, the Python script, a file for the standard input (`-` for none) and the command-line arguments. The last line runs the cashier on a second file of orders, which Section 13.12 looks at. **Listing 13.15** is the test program itself; the same file is used in Chapters 14, 15 and 16.

Listing 13.15 `tests/run_tests.py`: compiles, runs and compares every program of a chapter.

```
#!/usr/bin/env python3
"""run_tests.py - build every C++ program of this chapter, run it and its
Python twin, and check that both print exactly the expected output in
expected/NAME.txt.

usage (from the chapter folder):  python3 tests/run_tests.py [--sanitize] [--update]
Each line of tests/cases.txt is:
    name | C++ sources | Python script | stdin | arguments
('-' means none). A C++ source list of the form '!file.cpp' must FAIL to compile.
"""
import os, shutil, subprocess, sys, tempfile

sys.dont_write_bytecode = True

HERE = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
FLAGS = ["-std=c++17", "-O2", "-Wall", "-Wextra", "-Werror"]
SAN = ["-fsanitize=address,undefined", "-fno-sanitize-recover=all"]


def run(cmd, stdin_path):
    stdin = open(os.path.join(HERE, stdin_path)) if stdin_path != "-" else None
    try:
        r = subprocess.run(cmd, cwd=HERE, stdin=stdin, capture_output=True,
                           text=True, timeout=60)
    finally:
        if stdin:
            stdin.close()
    return r.stdout + r.stderr


def main():
    sanitize = "--sanitize" in sys.argv
    update = "--update" in sys.argv
    gxx = shutil.which("g++") or "g++"
    py = sys.executable
    tmp = tempfile.mkdtemp()
    failures = 0
    for line in open(os.path.join(HERE, "tests", "cases.txt")):
        if not line.strip() or line.startswith("#"):
            continue
        name, srcs, script, stdin, args = [p.strip() for p in line.split("|")]
        args = [] if args == "-" else args.split()
        if srcs.startswith("!"):                    # a program that must not compile
            src = os.path.join(HERE, "cpp", srcs[1:])
            r = subprocess.run([gxx] + FLAGS + [src, "-o", os.path.join(tmp, name)],
                               capture_output=True, text=True)
```

Listing 13.15 (Continued) *tests/run_tests.py: compiles, runs and compares every program of a chapter.*

```
            ok = r.returncode != 0 and "ambiguous" in r.stderr
            verdict = "refused by g++ as expected" if ok else "FAIL"
            print("%-26s %s" % (name, verdict))
            failures += not ok
            continue
        exe = os.path.join(tmp, name)
        sources = [os.path.join(HERE, "cpp", s) for s in srcs.split()]
        r = subprocess.run([gxx] + FLAGS + (SAN if sanitize else []) + sources
                           + ["-o", exe], capture_output=True, text=True)
        if r.returncode != 0:
            print("%-26s COMPILE ERROR\n%s" % (name, r.stderr))
            failures += 1
            continue
        out_cpp = run([exe] + args, stdin)
        out_py = run([py, "-B", os.path.join("python", script)] + args, stdin)
        exp_path = os.path.join(HERE, "expected", name + ".txt")
        if update:
            with open(exp_path, "w") as f:
                f.write(out_cpp)
        expected = open(exp_path).read() if os.path.exists(exp_path) else None
        ok_cpp, ok_py = out_cpp == expected, out_py == expected
        print("%-26s C++ %-4s Python %s" % (name, "ok" if ok_cpp else "FAIL",
                                            "ok" if ok_py else "FAIL"))
        failures += (not ok_cpp) + (not ok_py)
    shutil.rmtree(tmp)
    print("all programs agree" if failures == 0 else "%d FAILURE(S)" % failures)
    sys.exit(1 if failures else 0)


if __name__ == "__main__":
    main()
```

With `--sanitize`, the C++ programs are also compiled with the *address sanitizer* and the *undefined-behavior sanitizer*, which add checks for out-of-range memory access, use of freed memory, integer overflow and similar errors, and stop the program at the first one. A run on this chapter:

```
$ python3 tests/run_tests.py --sanitize
```

```
hello_laundry              C++ ok   Python ok
strings_demo               C++ ok   Python ok
vector_demo                C++ ok   Python ok
references                 C++ ok   Python ok
overload_default           C++ ok   Python ok
new_delete                 C++ ok   Python ok
cashier                    C++ ok   Python ok
cashier_bad                C++ ok   Python ok
all programs agree
```

Two languages that print exactly the same thing are a strong check on each other: if one of them had a rounding difference or a different order of rows, the comparison would fail. After any change to a program, run the tests again.

13.12 Making It Better: Refusing Bad Lines Politely

A cashier program meets real input, and real input is messy. The file `input/orders_bad.txt` collects the mistakes that actually happen at a counter: a service that does not exist, a weight written with a comma, a line with a field missing, an empty name, a misspelled “yes”, and a weight typed with one zero too many.

```
# customer;service;grams;express
Bu Rina Wulandari;wash;3450;no
Pak Budi Santoso;dryclean;5200;yes
Mbak Sari Dewi;iron;2,5;no
Mas Joko Prasetyo;wash;4100
;wash;1000;no
Pak Hendra Gunawan;washiron;2750;maybe
Bu Lestari;iron;60000;no
Bu Lestari;iron;1800;yes
```

```
$ ./cashier input/orders_bad.txt
```

```
PROFITINA LAUNDRY - cashier v1.0 (input/orders_bad.txt)

 1 Bu Rina Wulandari    wash       3.450 kg  Rp  24150
   line 3 refused: unknown service 'dryclean'
   line 4 refused: weight '2,5' is not 1..50000 grams
   line 5 refused: expected 4 fields, found 3
   line 6 refused: customer name is empty
   line 7 refused: express must be yes or no, not 'maybe'
   line 8 refused: weight '60000' is not 1..50000 grams
 2 Bu Lestari           iron       1.800 kg  Rp  13500  express

Orders accepted : 2
Orders refused  : 6
Express orders  : 1
Total weight    : 5.250 kg
Revenue         : Rp 37650

Per service
  wash      1 orders   3.450 kg  Rp  24150
  washiron  0 orders   0.000 kg  Rp      0
  iron      1 orders   1.800 kg  Rp  13500
```

Every bad line is refused with its line number and a reason in plain words, and the good lines around them are still accepted. This is the behavior of a trustworthy program: it never guesses what a wrong line meant, it never crashes, and it tells the person what to fix. Compare it with the C cashier of Chapter 11, where a comma in the weight made `scanf` stop reading and the rest of the file was silently lost.

Three further improvements are worth making and are left as exercises: accept the weight also in kilograms with a decimal comma, as Indonesians write it (`3,45`), converting it carefully to grams (Exercise 3); read the price list from a file instead of the source code, so that Bu Rina can change prices without a programmer (Exercise 4); and group the receipt by customer, so that Bu Rina Wulandari, who has two orders today, gets one total (Exercise 5).

13.13 Using It for Real

The cashier is small, but its shape fits many small businesses: a price list, a file of transactions, a strict check of each line, a printed row per transaction and a summary. To adapt it to your own business, change three things.

- **The price list.** A tailor charges per piece and per kind of work; a photocopy shop charges per page with a lower price above 100 pages; a car wash charges per vehicle class. Replace `SERVICES` and the rate calculation. If the price is per piece, the weight column becomes a count, and `parseGrams` becomes `parseCount` with its own limits.
- **The checks.** Ask the person at the counter which mistakes really happen, and add one check with one clear message for each. A check that is never needed costs nothing; a missing check can cost a day’s accounts.

- **The summary.** Ask the owner which numbers she looks at in the evening. For Bu Rina it was revenue per service; a photocopy shop might want pages per hour, to know when to open a second machine.

Keep money in whole rupiah and integer arithmetic, as Equation (13.1) does. If a price involves a percentage, round once, at a documented place, in the same way in every program that computes it. Chapter 15 turns this rule into a `Rupiah` class that makes it hard to break.

13.14 Common Mistakes, and How to Avoid Them

- **Forgetting `-std=c++17`.** Without it, a different compiler version may choose a different language version, and code that compiled yesterday gives warnings or errors today. Always write the flag.
- **Mixing `>>` and `getline`.** A `getline` directly after `cin >> number` reads the empty rest of the line. Read lines first, or call `std::cin.ignore(1000, '\n')` in between.
- **Comparing C strings with `==`.** For `std::string`, `a == b` compares the text. For two `char*`, it compares *addresses* and is almost always false. When a C++ program still has C strings, convert them: `std::string(argv[1]) == "yes"`.
- **Indexing a vector past its end.** `v[i]` does not check. Use range-based loops where you can, `v.at(i)` while developing, and compile tests with `-fsanitize=address`.
- **A reference to something that no longer exists.** A function must never return a reference to one of its own local variables: the variable dies when the function returns, and the reference points at nothing (a *dangling reference*). Return by value instead; C++ moves the value out cheaply.
- **Mismatched `new` and `delete`.** `new[]` must be freed with `delete[]`, `new` with `delete`. Better still, use a `vector` and write neither.
- **Money in `double`.** Rupiah amounts computed in floating point can print as one rupiah less than expected. Keep whole rupiah in `long long` and round explicitly, as Equation (13.1) does.
- **Ambiguous overloads.** Two overloads that fit a call equally well stop the compilation. Keep overloads clearly different, and do not add default arguments that make one overload look like another.

13.15 Chapter Summary

- C++ grew out of C and compiles almost every C program; `g++ -std=c++17 -Wall -Wextra` compiles the programs of this book on Windows (MSYS2), macOS and Linux. The ISO committee finished C++26 in March 2026, but C++17 is all this book needs.
- `std::cout << x` and `std::cin >> x` print and read values of any type without format strings; `std::getline` reads a whole line. Everything in the standard library lives in the namespace `std`.
- `std::string` holds text of any length, copies and compares with ordinary operators, and offers `size`, `find`, `substr` and `+`. `std::vector<T>` is a growing array with `push_back`, `size`, `at` and range-based loops, and works with `std::sort`.
- A reference is another name for an existing variable. Reference parameters (`int&`) let a function change the caller's variable without pointers; `const` references pass large objects without copying and without the right to change them.
- Overloading gives several functions one name, distinguished by their parameter lists; default arguments let callers leave out trailing arguments. Both are resolved at compile time.
- `new`/`delete` and `new[]`/`delete[]` replace `malloc`/`free` and must be paired correctly; in practice, containers such as `vector` make them rarely necessary.

- Python offers the same conveniences in its own way: `str`, `list`, names bound to objects, default arguments, automatic memory, and one function per name. The cashier v1.0 was written in both languages with identical output and tested by comparing both with the expected text.

13.16 Exercises

1. Without running it, say what `std::cout << 7 / 2 << ' ' << 7.0 / 2 << ' ' << (7 > 2) << '\n';` prints, and why the last value is not `true`. Then change the line so that it prints `true`.
2. Write a C++ function `swapOrders` that exchanges two `std::string` values through reference parameters, and the Python equivalent. Why does the Python version have to return something?
3. Extend `parseGrams` in the cashier so that it also accepts kilograms with a decimal comma, such as `3,45` (meaning 3,450 g) or `0,8` (800 g), with at most three digits after the comma. Keep refusing `2,5,1`, `,5` and `3,4567`. Do it in C++ and in Python and keep the outputs equal.
4. Move the price list of the cashier into a file `input/services.txt` with lines such as `wash;7000`, read it at the start of `main` into the `std::vector<Service>`, and refuse to start if the file is missing or a rate is not a positive number.
5. Add a third summary, “Per customer”, to the cashier: the number of orders and the total amount of each customer, in the order in which each customer first appears.
6. The function `kg(int)` in **Listing 13.13** only calls `kg(long long)`. What would happen if we deleted it? What would happen if we deleted `kg(long long)` instead and kept `kg(int)`? Try both and explain the compiler’s behavior.

13.17 Solutions to the Exercises

1. `7 / 2` is integer division, 3; `7.0 / 2` is floating-point division, 3.5; and `(7 > 2)` is the `bool` value `true`, which a stream prints as `1` by default. The output is `3 3.5 1`. Sending the manipulator `std::boolalpha` into the stream first switches `bool` printing to words: `std::cout << std::boolalpha << (7 > 2)` prints `true`. (Chapter 15’s `rupiah_demo.cpp` uses exactly this manipulator.)

2. In C++:

```
void swapOrders(std::string& a, std::string& b) {
    std::string t = a;
    a = b;
    b = t;
}
```

The call is `swapOrders(first, second);` and both variables are changed. (The standard library already has `std::swap`, which does the same for any type.) In Python a function cannot rebind the caller’s names, so it returns the pair in the new order and the caller unpacks it: `def swap_orders(a, b): return b, a` and `first, second = swap_orders(first, second)`; Python programmers usually write `first, second = second, first` directly.

3. Split the text at the comma. With no comma, keep the old rules. With one comma, both parts must be digits, the left part at most two digits and the right part one to three digits; pad the right part with zeros to three digits and compute $1000 \cdot \text{left} + \text{right}$. So `3,45` becomes $3000 + 450 = 3450$ and `0,8` becomes $0 + 800 = 800$. Then apply the same 1..50000 limit. `2,5,1` has two commas, `,5` an empty left part and `3,4567` four digits on the right, and all three are refused. In

Python, `text.split(",")` gives the parts, and the same checks apply; never use `float("3.45") * 1000`, which gives `3450.0000000000005` and shows why money and measures are safer in integers.

4. Change `SERVICES` from a constant into a `std::vector<Service>` that `main` fills: open `input/services.txt` with `std::ifstream`, skip comment lines, split each line at `;`, check that there are two fields and that the rate consists only of digits and is at least 1, and push a `Service`. If the file cannot be opened or any line is wrong, print the problem and return 1 before reading any orders. `findRate` and the summary loop need one change each: they receive the vector as a `const std::vector<Service>&` parameter instead of using the global constant. The Python version reads the file into a list of pairs in the same way.

5. Keep a `std::vector<std::string>` of customer names in first-appearance order: for each accepted order, search the vector (a linear search, Chapter 6) and push the name if it is not there yet. Then, for each name, loop over `orders` and add up the count and the amounts, exactly as the per-service loop does. For `input/orders.txt`, Bu Rina Wulandari appears first with 2 orders and $24{,}150 + 6{,}250 = 30{,}400$ rupiah. In Python, a `dict` keeps insertion order since Python 3.7, so `totals.setdefault(name, [0, 0])` does the bookkeeping in one line.

6. Without `kg(int)`, the call `kg(o.grams)` with an `int` argument still compiles: the only candidate is `kg(long long)`, and the `int` is converted to `long long` automatically (a *promotion* that loses nothing). Without `kg(long long)`, the call `kg(totalGrams)` with a `long long` argument also compiles, because a `long long` can be converted to `int`; with `-Wall -Wextra` there is no warning by default, but the conversion could lose information for totals above about 2.1 billion grams. This is why we kept the `long long` version as the real one: conversions toward the wider type are always safe, conversions toward the narrower one are not.

References

[1] B. Stroustrup, *The C++ Programming Language*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.

[2] H. Sutter, “C++26 is done! Trip report: March 2026 ISO C++ standards meeting (London Croydon, UK),” Sutter’s Mill, Mar. 29, 2026. [Online]. Available: https://herbsutter.com/2026/03/29/c26-is-done-trip-report-march-2026-iso-c-standards-meeting-london-croydon-uk/

[3] GNU Project, “GCC 16 Release Series: Changes, New Features, and Fixes,” Apr. 2026. [Online]. Available: https://gcc.gnu.org/gcc-16/

[4] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[5] P. Deitel and H. Deitel, *C++ How to Program*, 10th ed. Hoboken, NJ, USA: Pearson, 2017.

[6] cppreference.com, “C++ reference: Strings library, Containers library (std::vector), Input/output library,” accessed Sep. 2026. [Online]. Available: https://en.cppreference.com/w/cpp

[7] B. Stroustrup and H. Sutter, Eds., “C++ Core Guidelines,” accessed Sep. 2026. [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[8] Python Software Foundation, “The Python Tutorial, Section 4.9: More on Defining Functions,” Python 3.14 documentation, accessed Sep. 2026. [Online]. Available: https://docs.python.org/3.14/tutorial/controlflow.html

Chapter 14
Classes and Data Abstraction

A structure groups values that belong together, but it cannot protect them: any line anywhere in the program may set the weight of an order to zero or the name of its customer to nothing. This chapter introduces the *class*, the central idea of C++, which joins data with the functions that are allowed to touch it and hides everything else. By the end of the chapter you will be able to explain what data abstraction and an invariant are; write a class with private data, public member functions, constructors that refuse invalid values and `const` functions that promise not to change anything; split a class into a header and an implementation file and compile them together; use static members that belong to the class as a whole; rely on destructors that run by themselves at the end of a block; read and write a first small class hierarchy with inheritance and virtual functions; and build the order book of Profitina Laundry as two cooperating classes, `Order` and `Shop`, in C++ and in Python, with identical output.

14.1 Free Laundry, by Accident

At the end of one busy Saturday, Bu Rina's helper counted the money in the drawer and found it Rp 24,150 short of what the cashier program said. The next day the reason came out. One order had been entered with the wrong weight, and to "fix" it the helper had used a small menu option that a student had once added to the program: it set the weight of the last order directly. He typed 0 by mistake, the price became zero, and Bu Rina Wulandari took her laundry home for free. Nobody had done anything wicked. A number had been written where it should not have been, by code far away from the code that computes prices, and nothing in the program objected.

In the C cashier, an order is a `struct`. Every function that can see an order can change any of its fields to any value: a weight of 0 or −500 grams, a rate of 700 instead of 7,000, an empty name. The rules that make an order *sensible* (a real customer, a known service, a weight between 1 gram and 50 kilograms) live in the programmer's head and in a few `if` statements in the input function. Every new function is a new chance to forget them.

The idea of this chapter is to put those rules *inside* the order itself. An order will check its values when it is created and refuse to exist if they are wrong. Its data will be hidden, so that the only way to change it is through a small set of functions that keep the rules. An order that has been picked up cannot be sent back to the washing machine; an order that is still being washed cannot be picked up. Once the rules are inside, no helper, student or future programmer can break them by accident, because the program will not compile or will refuse the operation with a clear message.

This helps Bu Rina, whose accounts will add up. It helps every programmer who ever touches the cashier, because they can use an `Order` without knowing its internals. And it prepares you for the rest of your programming life: the class is the basic building block of C++, Java, C#, Python and almost every language you will use after this book.

14.2 The Idea in Plain Words

14.2.1 Data abstraction

Think about how you use a rice cooker. You put in rice and water, press one button, and wait for the light to switch from “cook” to “warm”. You do not open it to adjust the heating element, and you could not do so safely if you tried: the element is sealed inside, and the thermostat that decides when the rice is done is not something you can touch. What you get is a simple *interface* (a button and a light) and a guarantee (the rice will not burn), and the *implementation* (element, thermostat, timing) is hidden. The makers can even change the implementation in next year’s model without changing how you use it.

A class does the same for data. Its *interface* is a set of functions that other code may call: create an order, ask for its price, move it to the next step, hand it over to the customer. Its *implementation* is the data inside and the code of those functions, and it is hidden. This separation is called *data abstraction*, and the thing that results, a type defined by what you can do with it rather than by how it is stored, is called an *abstract data type* [1].

14.2.2 The invariant

What does hiding buy us? It buys an *invariant*.

> **DEFINITION.** An *invariant* of a class is a condition on its data that is true for every object of the class at every moment when code outside the class can look at it. The constructor establishes the invariant; every public member function may assume it at its start and must restore it before it returns.

For a Profitina Laundry order, the invariant is: the customer’s name is not empty, the service is one of `wash`, `washiron` and `iron`, and the weight g in grams satisfies

$$1 \le g \le 50\,000 \tag{14.1}$$

and the status is one of four values that change only in the order Received, Washing, Ready, Picked up. If the data is private, only the class’s own functions can change it, so if each of them keeps the invariant, the invariant holds forever. That is a remarkable guarantee from a small amount of code: we check the rules in *one* place, and they then hold in *every* place, including code that will be written years from now [2].

14.2.3 From `struct` to `class`

In C++ a class is written like a structure, with two additions. The members are divided into sections by *access specifiers*, and the functions that work on the data are declared *inside* the class, as *member functions*. **Table 14.1** shows who may use a member in each section.

Table 14.1 The three access specifiers and who may use a member in each.

Access	The class’s own member functions	Derived classes (Section 14.6)	Everyone else
`public`	yes	yes	yes
`protected`	yes	yes	no
`private`	yes	no	no

In C++, `struct` and `class` are almost the same thing; the only difference is the default: members of a `struct` are public unless you say otherwise, members of a `class` are private. By convention we

use `struct` for plain bundles of data with no rules to keep (as the cashier of Chapter 13 did) and `class` for types with an invariant.

14.3 The Problem, Measured

Before we fix the problem, let us see it. **Listing 14.1** is the order of the C chapters, written as a C++ structure, and a `main` that does to it what the accident did.

Listing 14.1 `struct_problem.cpp`: an open structure accepts any value in any field.

```
// struct_problem.cpp - Chapter 14: why an open struct is not enough.
#include <iostream>
#include <string>

struct Order {                                  // every field is public
    std::string customer;
    int grams;
    long long ratePerKg;
};

long long price(const Order& o) { return (o.grams * o.ratePerKg + 500) / 1000; }

void show(const Order& o) {
    std::cout << o.customer << ", " << o.grams << " g at Rp " << o.ratePerKg
              << "/kg: Rp " << price(o) << '\n';
}

int main() {
    Order o{"Bu Rina", 3450, 7000};
    show(o);
    o.grams = 0;                                // a slip somewhere far away...
    show(o);                                    // ...and the laundry is free
    o.grams = 3450;
    o.ratePerKg = 700;                          // one zero missing
    show(o);
    o.customer = "";                            // an order for nobody
    show(o);
    return 0;
}
```

```
Bu Rina, 3450 g at Rp 7000/kg: Rp 24150
Bu Rina, 0 g at Rp 7000/kg: Rp 0
Bu Rina, 3450 g at Rp 700/kg: Rp 2415
, 3450 g at Rp 700/kg: Rp 2415
```

Three assignments, three nonsense orders: free laundry, a price ten times too low, and an order for nobody. The compiler cannot help, because nothing in the program says these values are wrong. **Listing 14.2** shows that Python behaves in the same way with a plain record (a *dataclass*, Python's quick way to declare a structure).

Listing 14.2 `struct_problem.py`: a Python dataclass is just as open.

```python
# struct_problem.py - Chapter 14: a plain record lets anyone write anything.
from dataclasses import dataclass


@dataclass
class Order:                                    # every field is open
    customer: str
    grams: int
    rate_per_kg: int


def price(o):
    return (o.grams * o.rate_per_kg + 500) // 1000


def show(o):
    print(f"{o.customer}, {o.grams} g at Rp {o.rate_per_kg}/kg: Rp {price(o)}")


o = Order("Bu Rina", 3450, 7000)
show(o)
o.grams = 0                                     # a slip somewhere far away...
show(o)                                         # ...and the laundry is free
o.grams = 3450
o.rate_per_kg = 700                             # one zero missing
show(o)
o.customer = ""                                 # an order for nobody
show(o)
```

14.4 Building the `Order` Class

Figure 14.1 is the plan. The public part is the interface: a constructor, functions that *read* the order (`id`, `customer`, `grams`, `price`, `status` and so on), two functions that *change* it in allowed ways (`advance` and `pickUp`), and a counter of all valid orders. The private part holds the data, and code outside the class cannot reach it.

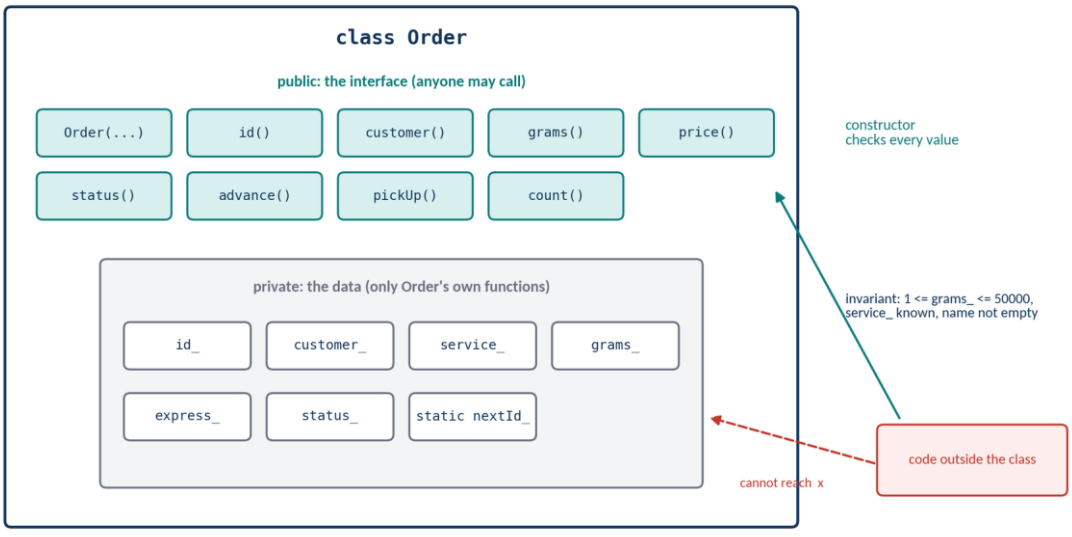


Figure 14.1 The `Order` class: a public interface around private data; outside code can only call the public functions, and the constructor checks every value.

14.4.1 The header: the interface

In C++ a class is usually split into two files. The *header* `order.h` says *what* the class offers; the *implementation* `order.cpp` says *how*. Other files include only the header. **Listing 14.3** is the header.

Listing 14.3 `order.h`: the interface of the `Order` class.

```
// order.h - Chapter 14: the Order class of Profitina Laundry (the interface).
#ifndef ORDER_H
#define ORDER_H

#include <string>

enum class Status { Received, Washing, Ready, PickedUp };

std::string toString(Status s);

class Order {
public:
    Order(const std::string& customer, const std::string& service, int grams,
          bool express = false);

    int id() const { return id_; }
    const std::string& customer() const { return customer_; }
    const std::string& service() const { return service_; }
    int grams() const { return grams_; }
    bool express() const { return express_; }
    Status status() const { return status_; }
    long long price() const;                    // computed, never stored

    bool advance();                             // Received -> Washing -> Ready
    bool pickUp();                              // only from Ready
    static int count() { return nextId_ - 1; } // valid orders made so far

private:
    int id_;
    std::string customer_;
    std::string service_;
    int grams_;
    bool express_;
    Status status_;
    static int nextId_;                         // one counter shared by all orders
    static long long rateFor(const std::string& service);
};

#endif
```

Read it from the top.

- `#ifndef ORDER_H`, `#define ORDER_H` and `#endif` are an *include guard*, the preprocessor trick of Chapter 11: if a file includes `order.h` twice (directly and through another header), the second copy is skipped.
- `enum class Status { Received, Washing, Ready, PickedUp };` is a *scoped enumeration*. Like C's `enum` from Chapter 10 it gives names to a few values, but the names must be written with their type, `Status::Ready`, so they cannot clash with other names, and a `Status` does not silently turn into an `int`. `toString` turns a status into text for printing.
- The constructor `Order(customer, service, grams, express = false)` has the same name as the class and no return type. It runs automatically whenever an `Order` is created, and its job is to establish the invariant.

- The small functions such as `int id() const { return id_; }` are *getters*: they give read access to one field. Their bodies are written right in the class, which makes them *inline*: the compiler replaces each call with the body, so a getter costs nothing at run time.
- The word `const` after the parameter list is a promise: *this function does not change the object*. The compiler checks the promise (assigning to a member inside a `const` function does not compile), and only `const` functions may be called on a `const Order`, such as the `const Order& o` parameter of `show` below. Every function that only reads should be `const`.
- `price()` is declared but not stored. The price is always *computed* from the weight, the service and the express flag, so it can never disagree with them.
- `static int count()` and `static int nextId_` are *static members*: they belong to the class as a whole, not to one order. There is exactly one `nextId_`, shared by all orders, and `Order::count()` can be called without any order at all.
- The private data members end with an underscore (`id_`, `grams_`), a common convention that makes it clear, inside member functions, which names are members and which are parameters or local variables.

14.4.2 The implementation

Listing 14.4 fills in the bodies.

Listing 14.4 `order.cpp`: the implementation of the `Order` class.

```
// order.cpp - Chapter 14: the Order class of Profitina Laundry (implementation).
#include "order.h"

#include <stdexcept>

int Order::nextId_ = 1;

std::string toString(Status s) {
    switch (s) {
    case Status::Received: return "received";
    case Status::Washing:  return "washing";
    case Status::Ready:    return "ready";
    case Status::PickedUp: return "picked up";
    }
    return "?";
}

long long Order::rateFor(const std::string& service) {
    if (service == "wash") return 7000;
    if (service == "washiron") return 10000;
    if (service == "iron") return 5000;
    return -1;
}

Order::Order(const std::string& customer, const std::string& service, int grams,
             bool express)
    : id_(0), customer_(customer), service_(service), grams_(grams),
      express_(express), status_(Status::Received) {
    if (customer_.empty())
        throw std::invalid_argument("customer name is empty");
    if (rateFor(service_) < 0)
        throw std::invalid_argument("unknown service '" + service_ + "'");
    if (grams_ < 1 || grams_ > 50000)
        throw std::invalid_argument("weight must be 1..50000 g, not "
                                    + std::to_string(grams_));
    id_ = nextId_++;                            // only a valid order gets a number
}

long long Order::price() const {
    long long p = (grams_ * rateFor(service_) + 500) / 1000;
    if (express_) p += p / 2;
    return p;
}
```

Listing 14.4 (Continued) *order.cpp: the implementation of the Order class.*

```
bool Order::advance() {
    if (status_ == Status::Received) { status_ = Status::Washing; return true; }
    if (status_ == Status::Washing)  { status_ = Status::Ready;   return true; }
    return false;                               // Ready and PickedUp stay put
}

bool Order::pickUp() {
    if (status_ != Status::Ready) return false;
    status_ = Status::PickedUp;
    return true;
}
```

`int Order::nextId_ = 1;` *defines* the static member: the header only declared it, and exactly one source file must create it. The prefix `Order::` says "the `nextId_` that belongs to `Order`". The same prefix appears in front of every member function defined outside the class: `long long Order::price() const` is the `price` function of `Order`.

The constructor begins with a *member initializer list*, the part after the colon: `: id_(0), customer_(customer), service_(service), ...`. It initializes each data member directly with a value, before the body of the constructor runs. This is the preferred way to set members in C++: it works for members that cannot be assigned later (constants and references), and it avoids first creating an empty string and then assigning to it.

Then the body checks the invariant of Equation (14.1). When a value is wrong, the constructor *throws an exception*: `throw std::invalid_argument("unknown service '" + service_ + "'");`. Section 13.5 showed an exception thrown by the library; here we throw one ourselves. Throwing from a constructor is the only honest thing a constructor can do when it cannot build a valid object: the object is simply never created, and the code that tried to create it jumps to its `catch` block. There is no half-built order with a bad weight lying around.

Only when all checks pass does the order get its number: `id_ = nextId_++;` copies the counter and then increases it. Because this line comes *after* the checks, refused orders do not use up numbers, and the numbers of valid orders are 1, 2, 3, … without gaps. So the number of valid orders is always

$$\text{count} = \text{nextId} - 1 \tag{14.2}$$

which is exactly what `count()` returns.

`advance` and `pickUp` are the only two functions that change an order after it has been created, and they implement the life cycle of **Figure 14.2**. `advance` moves Received to Washing and Washing to Ready, and refuses anything else by returning `false`; `pickUp` works only from Ready. Returning `bool` lets the caller decide what to tell the person at the counter.

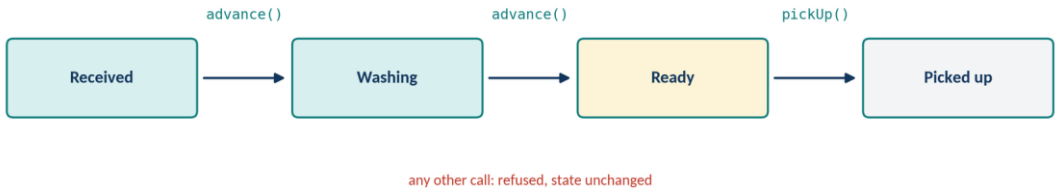


Figure 14.2 The life cycle of an order: `advance()` moves it forward twice, `pickUp()` hands it over, and every other call is refused without changing the state.

14.4.3 Using the class

Listing 14.5 creates orders, tries to create two invalid ones, and walks the first order through its life.

Listing 14.5 `order_main.cpp`: creating, refusing and advancing orders.

```cpp
// order_main.cpp - Chapter 14: using the Order class.
// Build: g++ -std=c++17 -Wall -Wextra order.cpp order_main.cpp -o order_main
#include <iostream>
#include <stdexcept>

#include "order.h"

void show(const Order& o) {
    std::cout << "#" << o.id() << " " << o.customer() << ", " << o.service() << ", "
              << o.grams() << " g" << (o.express() ? ", express" : "") << ": Rp "
              << o.price() << " [" << toString(o.status()) << "]\n";
}

int main() {
    Order a("Bu Rina", "wash", 3450);
    Order b("Pak Budi", "iron", 2000, true);
    show(a);
    show(b);

    try {
        Order bad("Mbak Sari", "wash", -500);
        show(bad);                              // never reached
    } catch (const std::invalid_argument& e) {
        std::cout << "refused: " << e.what() << '\n';
    }
    try {
        Order bad("Mas Joko", "dryclean", 1000);
        show(bad);
    } catch (const std::invalid_argument& e) {
        std::cout << "refused: " << e.what() << '\n';
    }

    std::cout << "pick up #1 now? " << (a.pickUp() ? "yes" : "no") << '\n';
    while (a.advance())
        std::cout << "#1 is now " << toString(a.status()) << '\n';
    std::cout << "pick up #1 now? " << (a.pickUp() ? "yes" : "no") << '\n';
    show(a);
    // a.grams_ = 0;   would not compile: 'int Order::grams_' is private
    std::cout << "valid orders created: " << Order::count() << '\n';
    return 0;
}
```

A program made of several source files is compiled by naming all of them; each `.cpp` file is compiled separately and the linker joins the results:

```
$ g++ -std=c++17 -O2 -Wall -Wextra cpp/order.cpp cpp/order_main.cpp -o order_main
$ ./order_main
```

```
#1 Bu Rina, wash, 3450 g: Rp 24150 [received]
#2 Pak Budi, iron, 2000 g, express: Rp 15000 [received]
refused: weight must be 1..50000 g, not -500
refused: unknown service 'dryclean'
pick up #1 now? no
#1 is now washing
#1 is now ready
pick up #1 now? yes
#1 Bu Rina, wash, 3450 g: Rp 24150 [picked up]
valid orders created: 2
```

The two invalid orders never came into existence: each constructor threw, the `show(bad)` line was skipped, and the `catch` printed the reason. Order #1 could not be picked up while it was still “received”, then advanced twice, stopped advancing at “ready” (the `while` loop ended when `advance` returned `false`), and was picked up. The counter says 2: the refused orders did not use numbers.

Notice the comment near the end of `main`. If you remove the `//` in front of `a.grams_ = 0;`, the compiler stops with the message that `grams_` is private. The accident of Section 14.1 has become impossible: not unlikely, *impossible*, checked by the compiler before the program ever runs.

14.4.4 The same class in Python

Python has classes too, and **Listing 14.6** is the same `Order`. The constructor is the method `__init__`; `self` is the object being built, written explicitly where C++ has the hidden pointer `this`. Python has no `private` keyword. Instead, a name that starts with an underscore (`_grams`) is private *by agreement*: the language does not stop you from touching it, but every Python programmer reads the underscore as "do not use from outside" [3]. The read-only `property` objects give the same interface as the C++ getters (`o.grams` instead of `o.grams()`), and `raise ValueError(...)` plays the role of `throw std::invalid_argument(...)`. `Enum` gives the scoped enumeration, and a class attribute `_next_id` is the static member.

Listing 14.6 `order.py`: the `Order` class in Python.

```
# order.py - Chapter 14: the Order class of Profitina Laundry in Python.
# A leading underscore marks a name as private by agreement; the read-only
# properties give the same public interface as order.h.
from enum import Enum


class Status(Enum):
    RECEIVED = "received"
    WASHING = "washing"
    READY = "ready"
    PICKED_UP = "picked up"


RATES = {"wash": 7000, "washiron": 10000, "iron": 5000}


class Order:
    _next_id = 1                                # one counter shared by all orders

    def __init__(self, customer, service, grams, express=False):
        if customer == "":
            raise ValueError("customer name is empty")
        if service not in RATES:
            raise ValueError(f"unknown service '{service}'")
        if not 1 <= grams <= 50000:
            raise ValueError(f"weight must be 1..50000 g, not {grams}")
        self._customer = customer
        self._service = service
        self._grams = grams
        self._express = express
        self._status = Status.RECEIVED
        self._id = Order._next_id               # only a valid order gets a number
        Order._next_id += 1

    id = property(lambda self: self._id)
    customer = property(lambda self: self._customer)
    service = property(lambda self: self._service)
    grams = property(lambda self: self._grams)
    express = property(lambda self: self._express)
    status = property(lambda self: self._status)
```

Listing 14.6 (Continued) *order.py: the Order class in Python.*

```
    def price(self):
        p = (self._grams * RATES[self._service] + 500) // 1000
        if self._express:
            p += p // 2
        return p

    def advance(self):
        if self._status == Status.RECEIVED:
            self._status = Status.WASHING
            return True
        if self._status == Status.WASHING:
            self._status = Status.READY
            return True
        return False                            # READY and PICKED_UP stay put

    def pick_up(self):
        if self._status != Status.READY:
            return False
        self._status = Status.PICKED_UP
        return True

    @staticmethod
    def count():
        return Order._next_id - 1
```

Listing 14.7 `order_main.py`: using the Python `Order`; it prints exactly what `order_main.cpp` prints.

```
# order_main.py - Chapter 14: using the Order class.
from order import Order


def show(o):
    print(f"#{o.id} {o.customer}, {o.service}, {o.grams} g"
          f"{', express' if o.express else ''}: Rp {o.price()} [{o.status.value}]")


a = Order("Bu Rina", "wash", 3450)
b = Order("Pak Budi", "iron", 2000, True)
show(a)
show(b)

for args in [("Mbak Sari", "wash", -500), ("Mas Joko", "dryclean", 1000)]:
    try:
        bad = Order(*args)
        show(bad)                               # never reached
    except ValueError as e:
        print(f"refused: {e}")

print(f"pick up #1 now? {'yes' if a.pick_up() else 'no'}")
while a.advance():
    print(f"#1 is now {a.status.value}")
print(f"pick up #1 now? {'yes' if a.pick_up() else 'no'}")
show(a)
print(f"valid orders created: {Order.count()}")
```

Listing 14.7 differs from its C++ twin in one visible way: the two invalid orders are tried in a loop over a list of argument tuples, and `Order(*args)` *unpacks* each tuple into the three arguments of the constructor. The output is character for character the same as before, which the test program of Section 14.9 checks.

14.4.5 Two questions beginners ask

Why write getters at all? Why not make the data public and `const`? Because the interface should promise *behavior*, not *storage*. Today `price()` computes the price from the weight; tomorrow it may look up a price list that changes on Sundays. Callers who wrote `o.price()` do not notice the change; callers who read a public member would all have to be rewritten. Keeping the data private also leaves the class free to change how it stores things, for example keeping the weight in tenths of a kilogram, without breaking a single line elsewhere. Experienced C++ programmers put it as a rule: make data members private, and let the interface consist of functions [4].

Inside a member function, how does `grams_` know *which* order's weight it means? Every non-static member function receives a hidden extra parameter, a pointer called `this`, that points at the object on which the function was called. When `main` calls `a.price()`, `this` points at `a`, and `grams_` inside `price` means `this->grams_`, the weight of `a`. You may write `this->grams_` yourself, and sometimes you must, for example when a parameter has the same name as a member. A `const` member function receives `this` as a pointer to a `const Order`, which is precisely how the compiler knows that the function may not change the object. Static member functions have no `this`, because they are not called on any particular object; that is why `count()` can only use static members such as `nextId_`. The Python parameter `self` is the same idea made visible: Python passes the object explicitly as the first argument, and you must write `self._grams` every time. Textbooks on C++ describe this mechanism in detail with pictures of memory [5]; for everyday programming it is enough to know that every member function works on "its own" object.

14.4.6 Tracing a refused order step by step

It helps to follow one refused order through the machine once, slowly. `main` executes `Order bad("Mbak Sari", "wash", -500);` inside a `try` block.

1. Memory for `bad` is reserved on the stack, and the constructor is called with `customer = "Mbak Sari"`, `service = "wash"`, `grams = -500` and, from the default argument, `express = false`.
2. The member initializer list runs: `id_` becomes 0, `customer_` a copy of the name, `service_` a copy of `"wash"`, `grams_` becomes −500, `express_` false, `status_ Status::Received`. At this moment the object *breaks* its invariant, but nobody outside can see it yet: the constructor has not finished.
3. The first check, `customer_.empty()`, is false. The second, `rateFor(service_) < 0`, is false, because `wash` costs 7,000. The third, `grams_ < 1`, is true.
4. `throw` creates a `std::invalid_argument` object holding the text `weight must be 1..50000 g, not -500` and leaves the constructor at once. The line `id_ = nextId_++;` never runs, so `nextId_` stays 3.
5. Because the constructor did not finish, the object is considered never to have existed. The members that *were* initialized (the two strings) are destroyed properly, so no memory leaks; there is no destructor call for `bad` itself.
6. The exception travels outward to the nearest matching `catch`, the one in `main` for `const std::invalid_argument&`. The line `show(bad);` is skipped. `e.what()` returns the stored text, and `main` prints `refused: weight must be 1..50000 g, not -500`.

The important point is step 5: in C++ there is no such thing as a half-constructed object that the rest of the program can touch. Either the constructor finishes and the invariant holds, or the object does not exist. Python gives the same guarantee in practice: if `__init__` raises, the expression `Order(...)` produces no value, and the name `bad` is never bound.

14.5 Constructors and Destructors: Born Valid, Cleaned Up Automatically

A constructor runs when an object is born. Its partner, the *destructor*, runs when the object dies. A destructor has the class name with a tilde, `~ShiftLog()`, no parameters and no return type. For an ordinary local variable, "dies" means "reaches the closing brace of the block in which it was declared", and this happens no matter how the block is left: at the brace, by `return`, or because an exception is passing through. **Listing 14.8** makes the moments visible with a shift log that prints a line when a cashier's shift opens and when it closes.

Listing 14.8 `raii_demo.cpp`: constructors and destructors announce themselves.

```
// raii_demo.cpp - Chapter 14: constructors and destructors run by themselves.
#include <iostream>
#include <string>

class ShiftLog {
public:
    explicit ShiftLog(const std::string& cashier) : cashier_(cashier) {
        std::cout << "[open ] shift of " << cashier_ << '\n';
    }
    ~ShiftLog() {                                // runs when the object dies
        std::cout << "[close] shift of " << cashier_ << ", " << entries_
                  << " entries\n";
    }
    void record(const std::string& what) {
        ++entries_;
        std::cout << "        " << cashier_ << " #" << entries_ << ": " << what
                  << '\n';
    }

private:
    std::string cashier_;
    int entries_ = 0;
};

int main() {
    std::cout << "shop opens\n";
    {
        ShiftLog morning("Dewi");
        morning.record("order #1 received");
        morning.record("order #2 received");
    }                                            // morning's destructor runs here
    ShiftLog evening("Eko");
    evening.record("order #1 picked up");
    {
        ShiftLog helper("Fajar");                // two logs alive at once
        helper.record("ironing for order #2");
    }
    std::cout << "shop closes\n";
    return 0;                                    // evening's destructor runs now
}
```

```
shop opens
[open ] shift of Dewi
        Dewi #1: order #1 received
        Dewi #2: order #2 received
[close] shift of Dewi, 2 entries
[open ] shift of Eko
        Eko #1: order #1 picked up
[open ] shift of Fajar
        Fajar #1: ironing for order #2
[close] shift of Fajar, 1 entries
shop closes
[close] shift of Eko, 1 entries
```

Dewi's log closes at the brace that ends her block, before Eko's opens. Fajar's log opens and closes inside Eko's shift. Eko's log closes last, after "shop closes", when `main` returns. Objects in the same scope are destroyed in the *reverse* order of their creation, like plates taken off a stack.

This behavior has a name that looks strange at first: *RAII*, "resource acquisition is initialization" [6]. The idea is that any resource (memory, an open file, a lock, a cashier's shift) is taken in a constructor and given back in a destructor. Then it is impossible to forget to give it back, because the language calls the destructor for you. You have already used RAII without knowing it: `std::string` and `std::vector` free their memory in their destructors, and `std::ifstream` closes its file in its destructor. That is why the cashier of Chapter 13 has no `free`, no `delete` and no `fclose`. When you write your own classes, follow the same rule: if a class takes something, its destructor gives it back.

Python frees memory by itself, but it does not promise *when* an object is destroyed, so a Python "destructor" (`__del__`) is not a reliable place to close a shift. Python's tool for guaranteed clean-up is the `with` statement: an object with the methods `__enter__` and `__exit__` is a *context manager*, and `__exit__` runs when the `with` block ends, however it ends. **Listing 14.9** prints exactly the same lines.

Listing 14.9 `raii_demo.py`: the same guarantee with Python's `with` statement.

```python
# raii_demo.py - Chapter 14: Python's 'with' gives the same guaranteed clean-up
# that a C++ destructor gives at the end of a block.


class ShiftLog:
    def __init__(self, cashier):
        self._cashier = cashier
        self._entries = 0
        print(f"[open ] shift of {cashier}")

    def __enter__(self):
        return self

    def __exit__(self, *exc):                  # runs when the 'with' block ends
        print(f"[close] shift of {self._cashier}, {self._entries} entries")
        return False

    def record(self, what):
        self._entries += 1
        print(f"        {self._cashier} #{self._entries}: {what}")


print("shop opens")
with ShiftLog("Dewi") as morning:
    morning.record("order #1 received")
    morning.record("order #2 received")
with ShiftLog("Eko") as evening:
    evening.record("order #1 picked up")
    with ShiftLog("Fajar") as helper:          # two logs alive at once
        helper.record("ironing for order #2")
    print("shop closes")
```

14.6 A First Look at Inheritance

Profitina Laundry does not only sell work by the kilogram. Bed covers and shoes are priced per piece, and any service can be ordered express at 50% more. All of them are *services*: each has a name and a way to compute a price for an amount. What differs is *how* the price is computed. C++ expresses "several kinds of one thing" with *inheritance* [7].

> **DEFINITION.** A class `D` *derived* from a *base class* `B`, written `class D : public B`, contains everything that `B` contains and may add members of its own; a `D` object *is a* `B` object and can be used wherever a `B` is expected. A member function declared `virtual` in the base class may be *overridden* in a derived class; a call through a pointer or reference to the base class then runs the version of the object's real class. A virtual function marked `= 0` is *pure*: the base class does not define it, and every concrete derived class must.

Listing 14.10 defines a base class `Service` and three derived classes.

Listing 14.10 `services.cpp`: a base class `Service` and three kinds of service.

```cpp
// services.cpp - Chapter 14: a first look at inheritance and virtual functions.
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

class Service {                                 // the base class
public:
    explicit Service(const std::string& name) : name_(name) {}
    virtual ~Service() = default;
    const std::string& name() const { return name_; }
    virtual long long price(int amount) const = 0;   // each kind decides
    virtual std::string unit() const = 0;

protected:
    std::string name_;                          // visible to derived classes
};

class PerKilo : public Service {                // a PerKilo IS-A Service
public:
    PerKilo(const std::string& name, long long ratePerKg)
        : Service(name), rate_(ratePerKg) {}
    long long price(int grams) const override {
        return (grams * rate_ + 500) / 1000;
    }
    std::string unit() const override { return "g"; }

private:
    long long rate_;
};

class PerPiece : public Service {
public:
    PerPiece(const std::string& name, long long ratePerPiece)
        : Service(name), rate_(ratePerPiece) {}
    long long price(int pieces) const override { return pieces * rate_; }
    std::string unit() const override { return "pcs"; }

private:
    long long rate_;
};
```

Listing 14.10 (Continued) *services.cpp: a base class Service and three kinds of service.*

```
class Express : public Service {               // wraps any service, adds 50%
public:
    explicit Express(const Service& base)
        : Service(base.name() + " express"), base_(base) {}
    long long price(int amount) const override {
        long long p = base_.price(amount);
        return p + p / 2;
    }
    std::string unit() const override { return base_.unit(); }

private:
    const Service& base_;
};

int main() {
    PerKilo wash("wash", 7000);
    PerKilo washIron("washiron", 10000);
    PerPiece bedCover("bed cover", 25000);
    Express fastWash(wash);
    Express fastCover(bedCover);

    std::vector<const Service*> menu = {&wash, &washIron, &bedCover, &fastWash,
                                        &fastCover};
    std::vector<int> amounts = {3450, 3450, 2, 3450, 2};
    for (std::size_t i = 0; i < menu.size(); ++i)   // one loop, five behaviors
        std::cout << std::left << std::setw(18) << menu[i]->name() << std::right
                  << std::setw(5) << amounts[i] << ' ' << std::left << std::setw(4)
                  << menu[i]->unit() << "Rp " << menu[i]->price(amounts[i]) << '\n';
    return 0;
}
```

```
wash               3450 g   Rp 24150
washiron           3450 g   Rp 34500
bed cover             2 pcs Rp 50000
wash express       3450 g   Rp 36225
bed cover express     2 pcs Rp 75000
```

The pieces, one by one:

- `Service` keeps the name in a **`protected`** member. According to **Table 14.1**, derived classes may use it but the rest of the program may not. `price` and `unit` are pure virtual, so `Service` is an *abstract* class: you cannot create a plain `Service`, only one of its concrete kinds.
- `virtual ~Service() = default;` gives the base class a virtual destructor. The rule is simple: a class that is meant to be a base class gets a virtual destructor, so that deleting a derived object through a base pointer runs the right destructor.
- `PerKilo` and `PerPiece` are derived with `: public Service`. **Constructors are not inherited**: each derived class writes its own, and it builds its base part first by calling the base constructor in its initializer list, `: Service(name), rate_(ratePerKg)`. Each overrides `price` and `unit`; the keyword `override` asks the compiler to check that a matching virtual function really exists in the base class, which catches misspelled names and wrong parameter types.
- `Express` is a service that *wraps* another service: its price is the other service's price plus half, the express rule of Section 13.7 once more. It holds a `const Service&`, a reference to any service at all, even another `Express`.
- In `main`, `menu` is a `std::vector<const Service*>`: pointers to the base class, pointing at objects of four different classes. The loop calls `menu[i]->price(amounts[i])` without knowing which class it is talking to, and each object answers in its own way. This is *polymorphism* ("many forms"): one call, several behaviors, chosen at run time by the real class of the object.

How does the program know, at run time, which `price` to call? The usual mechanism is simple enough to picture. For every class with virtual functions, the compiler builds one small table, the *virtual table*, holding the addresses of that class's versions of the virtual functions: the table of `PerKilo` points at `PerKilo::price` and `PerKilo::unit`, the table of `Express` at `Express::price` and `Express::unit`. Every object of such a class carries one hidden pointer to its class's table, set by the constructor. A call `menu[i]->price(amounts[i])` then does three small steps: follow the object's hidden pointer to its table, read the address stored in the `price` slot, and call the function at that address. Trace the fourth line of the output: `menu[3]` points at `fastWash`, an `Express`; its table leads to `Express::price`, which calls `base_.price(3450)`; `base_` refers to `wash`, a `PerKilo`, whose table leads to `PerKilo::price`, which returns $(3450 \times 7000 + 500)/1000 = 24{,}150$; back in `Express::price`, $24{,}150 + 12{,}075 = 36{,}225$ is returned and printed. The cost of a virtual call is one extra memory look-up, which is why C++ lets you choose, function by function, whether to pay it: a function that is not `virtual` is called directly, as in C.

This is also why the base class needs `virtual` on its destructor. Without it, `delete` through a `Service*` would call `Service`'s destructor directly, never looking in the table, and the derived part of the object would not be cleaned up.

Inheritance is a large subject, and a course on object-oriented programming usually follows this one. For this book you need exactly what this section showed: reading `class D : public B`, writing a constructor that calls the base constructor, using `protected` members, and overriding a virtual function. Chapter 16 (Problem 16.1) applies these four skills to chess pieces.

Listing 14.11 is the Python version. `class PerKilo(Service):` derives a class; `super().__init__(name)` calls the base constructor; the `abc` module (abstract base classes) with `@abstractmethod` makes `Service` abstract. In Python *every* method behaves like a C++ virtual function, so no keyword is needed.

Listing 14.11 `services.py`: the same hierarchy in Python.

```python
# services.py - Chapter 14: inheritance in Python; every method is "virtual".
from abc import ABC, abstractmethod


class Service(ABC):                             # the base class
    def __init__(self, name):
        self._name = name

    def name(self):
        return self._name

    @abstractmethod
    def price(self, amount): ...

    @abstractmethod
    def unit(self): ...


class PerKilo(Service):                         # a PerKilo IS-A Service
    def __init__(self, name, rate_per_kg):
        super().__init__(name)
        self._rate = rate_per_kg

    def price(self, grams):
        return (grams * self._rate + 500) // 1000

    def unit(self):
        return "g"


class PerPiece(Service):
    def __init__(self, name, rate_per_piece):
        super().__init__(name)
        self._rate = rate_per_piece

    def price(self, pieces):
        return pieces * self._rate

    def unit(self):
        return "pcs"
```

Listing 14.11 (Continued) *services.py: the same hierarchy in Python.*

```python
class Express(Service):                         # wraps any service, adds 50%
    def __init__(self, base):
        super().__init__(base.name() + " express")
        self._base = base

    def price(self, amount):
        p = self._base.price(amount)
        return p + p // 2

    def unit(self):
        return self._base.unit()


wash = PerKilo("wash", 7000)
wash_iron = PerKilo("washiron", 10000)
bed_cover = PerPiece("bed cover", 25000)
menu = [wash, wash_iron, bed_cover, Express(wash), Express(bed_cover)]
amounts = [3450, 3450, 2, 3450, 2]
for service, amount in zip(menu, amounts):      # one loop, five behaviors
    print(f"{service.name():<18}{amount:>5} {service.unit():<4}"
          f"Rp {service.price(amount)}")
```

14.7 Building It: The Shop

An order on its own is only half of a cashier. The other half is the book of all orders of the day: receive a new order, find an order by its number, move it forward, hand it over, list everything, and report the money. We make that a second class, `Shop`, which *has* a vector of orders. Where inheritance expresses "is a" (a `PerKilo` is a `Service`), a data member expresses "has a" (a `Shop` has orders), and most classes in real programs are built this second way, from other classes.

14.7.1 The `Shop` class

Listing 14.12 is the interface. Every operation returns one line of text that the program shows the person at the counter; this keeps `Shop` free of any decision about *where* text is printed.

Listing 14.12 `shop.h`: the interface of the `Shop` class.

```
// shop.h - Chapter 14: the Shop class keeps all orders of one day.
#ifndef SHOP_H
#define SHOP_H

#include <string>
#include <vector>

#include "order.h"

class Shop {
public:
    explicit Shop(const std::string& name) : name_(name) {}

    // Each operation returns one line of text for the cashier's screen.
    std::string receive(const std::string& customer, const std::string& service,
                        int grams, bool express);
    std::string advance(int id);
    std::string pickUp(int id);
    std::string list() const;
    std::string report() const;

private:
    Order* find(int id);                        // nullptr when there is no such id
    std::string name_;
    std::vector<Order> orders_;
};

#endif
```

The helper `find` is private: other code has no business holding a pointer into the shop's vector. It returns `nullptr` when no order has the given number, which is exactly the situation where a pointer is better than a reference (Section 13.6): "no such order" is a normal answer. `list` and `report` are `const`: looking at the shop does not change it.

Listing 14.13 is the implementation.

Listing 14.13 `shop.cpp`: the implementation of the `Shop` class.

```cpp
// shop.cpp - Chapter 14: the Shop class (implementation).
#include "shop.h"

#include <stdexcept>

Order* Shop::find(int id) {
    for (Order& o : orders_)
        if (o.id() == id) return &o;
    return nullptr;
}

std::string Shop::receive(const std::string& customer, const std::string& service,
                          int grams, bool express) {
    try {
        orders_.push_back(Order(customer, service, grams, express));
    } catch (const std::invalid_argument& e) {
        return std::string("refused: ") + e.what();
    }
    const Order& o = orders_.back();
    return "order #" + std::to_string(o.id()) + " for " + o.customer() + ", Rp "
           + std::to_string(o.price());
}

std::string Shop::advance(int id) {
    Order* o = find(id);
    if (o == nullptr) return "order #" + std::to_string(id) + " does not exist";
    std::string before = toString(o->status());
    if (!o->advance())
        return "order #" + std::to_string(id) + " cannot advance (" + before + ")";
    return "order #" + std::to_string(id) + ": " + before + " -> "
           + toString(o->status());
}

std::string Shop::pickUp(int id) {
    Order* o = find(id);
    if (o == nullptr) return "order #" + std::to_string(id) + " does not exist";
    if (!o->pickUp())
        return "order #" + std::to_string(id) + " is not ready ("
               + toString(o->status()) + ")";
    return "order #" + std::to_string(id) + " picked up by " + o->customer()
           + ", pay Rp " + std::to_string(o->price());
}
```

Listing 14.13 (Continued) *shop.cpp: the implementation of the Shop class.*

```
std::string Shop::list() const {
    std::string text = name_ + ", " + std::to_string(orders_.size()) + " orders";
    for (const Order& o : orders_)
        text += "\n  #" + std::to_string(o.id()) + " " + o.customer() + " ("
                + o.service() + ", " + std::to_string(o.grams()) + " g): "
                + toString(o.status());
    return text;
}

std::string Shop::report() const {
    long long paid = 0, open = 0;
    int done = 0;
    for (const Order& o : orders_) {
        if (o.status() == Status::PickedUp) {
            paid += o.price();
            ++done;
        } else {
            open += o.price();
        }
    }
    return "picked up " + std::to_string(done) + " of "
           + std::to_string(orders_.size()) + ", paid Rp " + std::to_string(paid)
           + ", still in the shop Rp " + std::to_string(open);
}
```

Look at how little `Shop` has to check. `receive` does not examine the weight or the service at all: it simply tries to construct an `Order`, and if the order's constructor throws, `receive` catches the exception and turns its message into a "refused" line. The rules live in one place, `Order`, and `Shop` trusts them. `advance` and `pickUp` find the order, remember its status as text for the message, and call the order's own function; if that returns `false`, the order has refused, and the shop reports why. `report` walks over the orders and separates the money already paid (orders picked up) from the money still in the shop.

14.7.2 The program around it

The program in **Listing 14.14** reads commands, one per line, and prints each command with the shop's answer. The commands are `new customer;service;grams;express`, `advance N`, `pickup N`, `list` and `report`.

Listing 14.14 `shop_main.cpp`: reads commands and passes them to the shop.

```
// shop_main.cpp - Chapter 14: runs a day of commands against a Shop.
// Build: g++ -std=c++17 -Wall -Wextra order.cpp shop.cpp shop_main.cpp -o shop
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include <vector>

#include "shop.h"

std::vector<std::string> split(const std::string& line, char sep) {
    std::vector<std::string> parts;
    std::string part;
    std::istringstream in(line);
    while (std::getline(in, part, sep)) parts.push_back(part);
    return parts;
}

std::string execute(Shop& shop, const std::string& line) {
    std::size_t space = line.find(' ');
    std::string command = line.substr(0, space);
    std::string rest = space == std::string::npos ? "" : line.substr(space + 1);
    try {
        if (command == "new") {
            std::vector<std::string> f = split(rest, ';');
            if (f.size() != 4) return "new needs customer;service;grams;express";
            return shop.receive(f[0], f[1], std::stoi(f[2]), f[3] == "yes");
        }
        if (command == "advance") return shop.advance(std::stoi(rest));
        if (command == "pickup") return shop.pickUp(std::stoi(rest));
    } catch (const std::logic_error&) {        // stoi failed: not a number
        return "not a number in '" + line + "'";
    }
    if (command == "list") return shop.list();
    if (command == "report") return shop.report();
    return "unknown command '" + command + "'";
}
```

Listing 14.14 (Continued) *shop_main.cpp: reads commands and passes them to the shop.*

```
int main() {
    Shop shop("Profitina Laundry");
    std::string line;
    while (std::getline(std::cin, line)) {
        if (line.empty() || line[0] == '#') continue;
        std::cout << "> " << line << '\n' << execute(shop, line) << '\n';
    }
    return 0;
}
```

Two details are new. `split` uses a `std::istringstream`, a stream that reads from a string instead of from the keyboard, together with the three-argument `std::getline(in, part, sep)`, which reads up to a separator of your choice. And `std::stoi` ("string to int") converts text to a number, throwing `std::invalid_argument` if the text does not start with a number; since that exception class is derived from `std::logic_error`, the single `catch (const std::logic_error&)` also catches the out-of-range error `stoi` throws for huge numbers. Here, inheritance helps us catch a whole family of errors at once.

The Python versions (**Listing 14.15** and **Listing 14.16**) have the same structure, method for method.

Listing 14.15 `shop.py`: the `Shop` class in Python.

```python
# shop.py - Chapter 14: the Shop class keeps all orders of one day.
from order import Order, Status


class Shop:
    def __init__(self, name):
        self._name = name
        self._orders = []

    def _find(self, order_id):
        for o in self._orders:
            if o.id == order_id:
                return o
        return None

    def receive(self, customer, service, grams, express):
        try:
            o = Order(customer, service, grams, express)
        except ValueError as e:
            return f"refused: {e}"
        self._orders.append(o)
        return f"order #{o.id} for {o.customer}, Rp {o.price()}"

    def advance(self, order_id):
        o = self._find(order_id)
        if o is None:
            return f"order #{order_id} does not exist"
        before = o.status.value
        if not o.advance():
            return f"order #{order_id} cannot advance ({before})"
        return f"order #{order_id}: {before} -> {o.status.value}"

    def pick_up(self, order_id):
        o = self._find(order_id)
        if o is None:
            return f"order #{order_id} does not exist"
        if not o.pick_up():
            return f"order #{order_id} is not ready ({o.status.value})"
        return f"order #{order_id} picked up by {o.customer}, pay Rp {o.price()}"
```

Listing 14.15 (Continued) *shop.py: the Shop class in Python.*

```python
    def list(self):
        text = f"{self._name}, {len(self._orders)} orders"
        for o in self._orders:
            text += (f"\n  #{o.id} {o.customer} ({o.service}, {o.grams} g): "
                     f"{o.status.value}")
        return text

    def report(self):
        done = [o for o in self._orders if o.status == Status.PICKED_UP]
        paid = sum(o.price() for o in done)
        still = sum(o.price() for o in self._orders) - paid
        return (f"picked up {len(done)} of {len(self._orders)}, paid Rp {paid}, "
                f"still in the shop Rp {still}")
```

Listing 14.16 `shop_main.py`: the command loop in Python.

```python
# shop_main.py - Chapter 14: runs a day of commands against a Shop.
import sys

from shop import Shop


def execute(shop, line):
    command, _, rest = line.partition(" ")
    try:
        if command == "new":
            f = rest.split(";")
            if len(f) != 4:
                return "new needs customer;service;grams;express"
            return shop.receive(f[0], f[1], int(f[2]), f[3] == "yes")
        if command == "advance":
            return shop.advance(int(rest))
        if command == "pickup":
            return shop.pick_up(int(rest))
    except ValueError:                          # int() failed: not a number
        return f"not a number in '{line}'"
    if command == "list":
        return shop.list()
    if command == "report":
        return shop.report()
    return f"unknown command '{command}'"


shop = Shop("Profitina Laundry")
for line in sys.stdin.read().splitlines():
    if line == "" or line.startswith("#"):
        continue
    print(f"> {line}\n{execute(shop, line)}")
```

14.8 Running It

The file `input/day.txt` is one day at the counter, including mistakes: an order with a negative weight, a pick-up before the laundry is ready, an order number that does not exist, a word where a number should be, and an unknown command.

```
$ g++ -std=c++17 -O2 -Wall -Wextra cpp/order.cpp cpp/shop.cpp cpp/shop_main.cpp -o shop
$ ./shop < input/day.txt
$ python3 python/shop_main.py < input/day.txt
```

```
> new Bu Rina;wash;3450;no
order #1 for Bu Rina, Rp 24150
> new Pak Budi;iron;2000;yes
order #2 for Pak Budi, Rp 15000
> new Mbak Sari;washiron;-500;no
refused: weight must be 1..50000 g, not -500
> new Mas Joko;washiron;5200;no
order #3 for Mas Joko, Rp 52000
> advance 1
order #1: received -> washing
> advance 1
order #1: washing -> ready
> pickup 2
order #2 is not ready (received)
> advance 2
order #2: received -> washing
> advance 2
order #2: washing -> ready
> pickup 2
order #2 picked up by Pak Budi, pay Rp 15000
> advance 2
order #2 cannot advance (picked up)
> pickup 1
order #1 picked up by Bu Rina, pay Rp 24150
> advance 9
order #9 does not exist
> pickup two
not a number in 'pickup two'
> refund 1
unknown command 'refund'
> list
Profitina Laundry, 3 orders
  #1 Bu Rina (wash, 3450 g): picked up
  #2 Pak Budi (iron, 2000 g): picked up
  #3 Mas Joko (washiron, 5200 g): received
> report
picked up 2 of 3, paid Rp 39150, still in the shop Rp 52000
```

Follow the life of order #2 in the output. Pak Budi's order is received; an early pick-up is refused because it is still "received"; two `advance` commands take it to "ready"; the pick-up succeeds and asks for Rp 15,000; one more `advance` is refused because a picked-up order goes nowhere. Every step is exactly the diagram of **Figure 14.2**. Mbak Sari's order with -500 grams never got a number, so Mas Joko's order is #3, as Equation (14.2) promised. The report splits the Rp 91,150 of the three orders into Rp 39,150 already paid and Rp 52,000 for the laundry still on the shelves.

14.9 Testing It

The test program of Section 13.11 runs this chapter as well; its list of cases names the source files of the multi-file programs:

```
# name | C++ sources | Python script | stdin | arguments
struct_problem | struct_problem.cpp | struct_problem.py | - | -
order_main | order.cpp order_main.cpp | order_main.py | - | -
raii_demo | raii_demo.cpp | raii_demo.py | - | -
services | services.cpp | services.py | - | -
shop | order.cpp shop.cpp shop_main.cpp | shop_main.py | input/day.txt | -
$ python3 tests/run_tests.py --sanitize
```

```
struct_problem             C++ ok   Python ok
order_main                 C++ ok   Python ok
raii_demo                  C++ ok   Python ok
services                   C++ ok   Python ok
shop                       C++ ok   Python ok
all programs agree
```

A class invites a second, finer kind of test: *testing the invariant directly*. For `Order`, write down the boundary values of Equation (14.1) and try each one: 0 g and 50,001 g must be refused, 1 g and 50,000 g must be accepted. Try every transition of **Figure 14.2**, allowed and forbidden: there are 4 states and 2 operations, so 8 cases, of which exactly 3 succeed. Exercise 2 asks you to write these tests as a program.

14.10 Making It Better

The two classes can grow in several useful directions without changing how they are used.

- **Money as its own type.** Prices are still `long long` values that could be added to weights by mistake. Chapter 15 introduces a `Rupiah` class with its own operators, so that `price + grams` no longer compiles.
- **Printing with `<<`.** `show` in `order_main.cpp` is a free function with a made-up name. Chapter 15 shows how to teach `std::cout << order` to print an order, so that orders print like numbers.
- **Keeping the day.** At closing time the orders disappear. With the file processing of Chapter 11, `Shop` could save its orders to a file in its destructor and load them in its constructor, a perfect use of RAII. Exercise 5 asks for the saving half.
- **Finding faster.** `find` is a linear search, fine for a hundred orders a day. For a large shop, a `std::map<int, Order>` finds an order by number in $O(\log n)$ steps; Chapter 15 uses `std::map` for customers and services.

14.11 Using It for Real

The pattern of this chapter, a small class that guards its own rules plus a larger class that manages many of them, appears in every business system you will ever see. A clinic's queue has patients whose status goes Registered, Waiting, With the doctor, Done; a warung's delivery has orders that go Ordered, Cooking, On the way, Delivered; a library has books that are On the shelf, Borrowed, Overdue, Returned. For each of them, ask three questions, and the class almost writes itself:

1. *What makes one item valid?* Those conditions go into the constructor, as in Equation (14.1).
2. *In which order may its state change?* Draw the diagram, as in **Figure 14.2**, and write one member function per arrow; every function refuses what the diagram does not allow.
3. *What does the owner want to know at the end of the day?* That is the `report` of the managing class.

Take Mbak Wulan's catering warung as a worked example. An order is valid when it has a customer, at least one portion and at most 200 (her kitchen cannot cook more in one morning), and a delivery time between 06:00 and 20:00: three conditions for the constructor. Its states are Ordered, Cooking, On the way, Delivered, and one extra arrow, Ordered to Cancelled, because customers cancel only before cooking starts: five member functions at most, each refusing what the diagram does not show. At the end of the day she wants the portions cooked, the money received and the money still owed by office customers who pay at the end of the month: the

`report`. Written this way, her program cannot deliver an order that was never cooked or cook an order that was cancelled, however hurried the person at the phone.

For a clinic, a program like this only keeps the queue in order. Deciding who is urgent is a matter for the nurse and the doctor; a queue program should make it easy for them to move a patient forward, never decide it for them.

14.12 Common Mistakes, and How to Avoid Them

- **Public data "just for now".** Making a member public to fix a problem quickly lets every line of the program break the invariant again. Add a member function that does the needed change *and keeps the rules*.
- **Forgetting `const` on reading functions.** A getter without `const` cannot be called through a `const Order&`, and the compiler error appears far from the real cause. Mark every function that does not change the object `const` from the start.
- **Checking the invariant outside the class.** If `Shop::receive` also checked the weight, the two checks would drift apart the first time someone changed one of them. Check once, in the constructor, and let other code rely on it.
- **Assigning in the constructor body instead of initializing.** A reference member or a `const` member *must* be set in the initializer list; for the others, the list is still the clean and efficient way. Initialize the members in the same order in which they are declared in the class, because that is the order in which C++ initializes them regardless of how the list is written (`-Wall` warns when the orders differ).
- **Defining a static member twice or not at all.** `static int nextId_;` in the class only declares it; exactly one `.cpp` file must contain `int Order::nextId_ = 1;`. Without it the linker reports an "undefined reference"; with two definitions it reports a "multiple definition".
- **Compiling only one file of a multi-file program.** `g++ order_main.cpp` alone ends with "undefined reference to `Order::Order(...)`". Name every `.cpp` file on the command line (or let a build tool do it).
- **A base class without a virtual destructor.** Deleting a derived object through a base pointer then skips the derived part's destructor. Give every base class `virtual ~Base() = default;`.
- **Relying on Python's `__del__`.** It is not called at a guaranteed moment. Use `with` for anything that must be closed or released.

14.13 Chapter Summary

- Data abstraction separates a type's interface (what you may do with it) from its implementation (how it is stored and computed); a type defined by its operations is an abstract data type.
- A class joins data and member functions. `private` data can be changed only by the class's own functions, so a rule that those functions keep, the invariant, holds for every object at all times; `public` members form the interface; `protected` members are shared with derived classes.
- Constructors establish the invariant and may throw an exception to refuse invalid values, so that no invalid object ever exists; member initializer lists set members directly. Destructors run automatically at the end of an object's life, which makes RAII possible: resources taken in a constructor are always released in the destructor.
- `const` member functions promise not to change the object and are the only ones callable on `const` objects. Static members belong to the class as a whole.

- A class is usually split into a header (interface) and a source file (implementation); a program made of several source files is compiled by naming all of them.
- Inheritance (`class D : public B`) expresses "is a"; virtual functions let one call behave differently for each derived class; pure virtual functions make a base class abstract. Constructors are not inherited; each derived constructor calls its base constructor.
- In Python, `__init__` is the constructor, a leading underscore marks private data by agreement, properties give read-only access, `with` gives guaranteed clean-up, and every method is virtual. The `Order` and `Shop` classes behave identically in both languages.

14.14 Exercises

1. Add a member function `bool changeWeight(int grams)` to `Order` that corrects the weight of an order, but only while it is still `Received`, and only to a value that keeps the invariant. Why is this safe where the old "set the weight" menu option of Section 14.1 was not?
2. Write a test program `order_test.cpp` (and `order_test.py`) that checks the four boundary weights 0, 1, 50,000 and 50,001 and all 8 combinations of state and operation of **Figure 14.2**, printing one line per check and a final count of passed checks.
3. Add a fifth status, `Cancelled`, to `Order`: an order may be cancelled only while it is `Received`, and a cancelled order cannot advance or be picked up. Update the diagram, `toString`, and `Shop::report` (cancelled orders count neither as paid nor as in the shop).
4. Add a derived class `Minimum` to `services.cpp` that wraps a per-kilo service and charges at least the price of 3 kg: a customer with 1.2 kg of washing pays for 3 kg. What does `Minimum(wash)` charge for 1,200 g and for 3,450 g?
5. Give `Shop` a member function `save(const std::string& path) const` that writes one line per order (`id;customer;service;grams;express;status`) to a file with `std::ofstream`. Why should `save` be `const`?
6. In `order.cpp`, what would go wrong if the line `id_ = nextId_++;` were moved to the start of the constructor body, before the checks? Describe the output of `shop_main` for `input/day.txt` in that case.

14.15 Solutions to the Exercises

1. Declare `bool changeWeight(int grams);` in the public section and define it as: if `status_ != Status::Received` or `grams < 1` or `grams > 50000`, return `false`; otherwise set `grams_ = grams` and return `true`. It is safe because it is a member function: it can see the private weight, and it keeps both parts of the invariant, the range of Equation (14.1) and the rule that a weight is fixed once washing has begun. The old menu option wrote any number directly into the structure, with no rule at all. In Python, `def change_weight(self, grams)` does the same with `self._grams`.

2. For the weights, a small helper `bool accepted(int g)` tries `Order("Test", "wash", g)` inside `try` and returns `false` in the `catch`; the expected answers are `false, true, true, false`. For the transitions, write a function that creates a fresh order and brings it into each of the four states (0, 1 or 2 calls of `advance`, plus `pickUp` for the fourth), then tries `advance` or `pickUp` and compares the result with the table: `advance` succeeds from Received and Washing, `pickUp` only from Ready, so exactly 3 of the 8 cases return `true`. The program prints, for example, `advance from ready: false (expected false) ok` and at the end `12 of 12 checks passed`.

3. Add `Cancelled` to the `enum class` and to `toString`. Add `bool cancel()` that succeeds only from `Received`. `advance` and `pickUp` already refuse every state they do not name, so they need no

change: this is the benefit of writing "allow these" rather than "forbid those". In `Shop`, add a `cancel N` command and, in `report`, skip cancelled orders when adding the money; count them separately in the report line.

4. `class Minimum : public Service` holds a `const Service& base_` like `Express`, and its `price(int grams)` returns `base_.price(grams < 3000 ? 3000 : grams)`. For 1,200 g of `wash` it charges the price of 3,000 g: $(3000 \times 7000 + 500)/1000 = 21{,}000$. For 3,450 g the minimum does not apply, and the price is the normal 24,150.

5. `std::ofstream out(path);`, then one `out << o.id() << ';' << o.customer() << ...` line per order, and the file is closed by the destructor of `out` at the end of the function. `save` must be `const` because saving does not change the shop; declaring it `const` lets it be called on a `const Shop&` and documents that promise. Note that it can only call `const` member functions of the orders, which all the getters are.

6. Every constructor call, valid or not, would take a number, because the number is taken before the checks run. Mbak Sari's refused order would silently consume number 3, and Mas Joko's order would become #4. The `list` would show orders #1, #2 and #4, and `Order::count()` would say 4 although only 3 orders exist, breaking Equation (14.2). The lesson: change shared state only after all checks have passed.

References

[1] B. Liskov and S. Zilles, "Programming with abstract data types," *ACM SIGPLAN Notices*, vol. 9, no. 4, pp. 50–59, 1974.

[2] B. Stroustrup and H. Sutter, Eds., "C++ Core Guidelines, Section C: Classes and class hierarchies (C.2: Use class if the class has an invariant)," accessed Sep. 2026. [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[3] Python Software Foundation, "The Python Tutorial, Chapter 9: Classes," Python 3.14 documentation, accessed Sep. 2026. [Online]. Available: https://docs.python.org/3.14/tutorial/classes.html

[4] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2005.

[5] P. Deitel and H. Deitel, *C++ How to Program*, 10th ed. Hoboken, NJ, USA: Pearson, 2017.

[6] B. Stroustrup, *The C++ Programming Language*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.

[7] cppreference.com, "Derived classes" and "virtual function specifier," accessed Sep. 2026. [Online]. Available: https://en.cppreference.com/w/cpp/language/derived_class

Chapter 15

Operator Overloading and a Complete Case Study

You have written `a + b` since Chapter 1, and `std::cout << x` since Chapter 13, without asking what the `+` and the `<<` really are. In C++ they are functions with unusual names, and you can write such functions for your own classes. This chapter shows how and when to do it, and then uses everything from Chapters 13 to 15 in one complete program. By the end of the chapter you will be able to explain what the compiler does with an expression such as `wash + iron` when the operands are objects of your class; decide whether an operator should be a member or a non-member function; write arithmetic, compound-assignment, comparison and output operators for a money type and a fraction type; compute greatest common divisors and least common multiples with Euclid's algorithm; and build, run and test the complete Profitina Laundry program, which reads its price list, customers and orders from files, prints receipts with express surcharges and member discounts, and writes a sorted daily report, in C++ and in Python with identical output.

15.1 Receipts That Add Up

At the end of the month Bu Rina adds up her receipts, and every month the total is a few hundred rupiah away from what the cashier program says. The difference is never large enough to worry about and never small enough to ignore. When she finally asked her two helpers to compute the same receipt by hand, the reason appeared at once: one of them rounded the 10% member discount *down*, the other *to the nearest* rupiah, and the program did something else again. The receipts also looked different: one helper wrote "Rp 24150", the other "Rp 24,150", and the old program printed "Rp 24150".

The problem is not arithmetic; everyone can compute 10% of a number. The problem is that *money* is treated as a plain number, and every piece of code that handles it makes its own small decisions: how to round, how to print, whether it is fine to add a weight to a price. In a program of a few hundred lines there are dozens of such places, and they drift apart.

The cure is the idea of Chapter 14 applied to a number: make money a *class*. A `Rupiah` object holds a whole number of rupiah; it knows how to add and subtract itself, how to take a percentage with one fixed rounding rule, how to compare itself, and how to print itself in the Indonesian way, with dots between groups of three digits. And thanks to *operator overloading*, code that uses it still looks like ordinary arithmetic: `base + express - discount`.

15.2 The Idea in Plain Words

15.2.1 An operator is a function with a special name

When the compiler sees `wash + iron` and both operands are `int`, it emits a machine addition. When the operands are objects of a class, there is no machine instruction for them, so the compiler looks for a *function* named `operator+` that accepts them. **Figure 15.1** shows the two places where it looks.

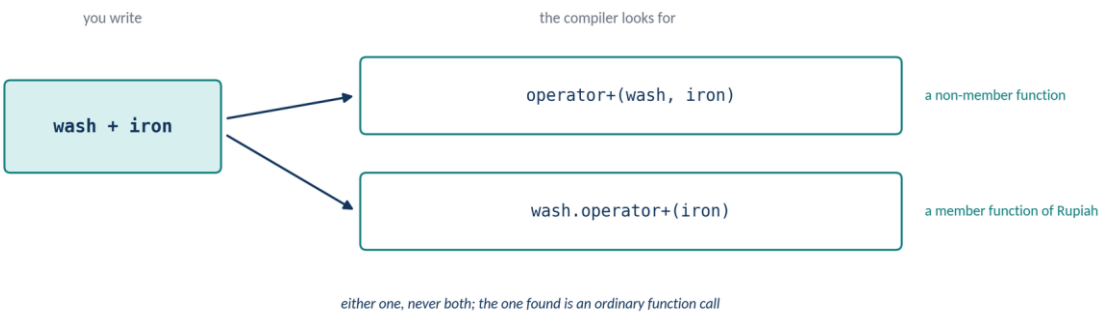


Figure 15.1 What the compiler does with `wash + iron` for class operands: it looks for a non-member `operator+(wash, iron)` or a member `wash.operator+(iron)` and calls the one it finds.

That is really all there is to it [1]. `operator+` is an ordinary function whose name happens to be `operator+`. You could even call it by that name, `operator+(wash, iron)`, and get the same result. The special syntax only lets you *use* it as `wash + iron`. Likewise, `std::cout << x` calls a function `operator<<(std::cout, x)`, which the standard library defines for every built-in type and which you may define for your own. You have been using overloaded operators since Chapter 13: `+` on two `std::string`s is one, and so are `==` and `<` on strings, and `[]` on a `std::vector`.

> **DEFINITION.** *Operator overloading* is defining the meaning of a C++ operator for operands of a user-defined type by writing a function named `operator` followed by the operator symbol. A binary operator is either a non-member function with two parameters or a member function with one parameter (the left operand is the object itself); a unary operator is a non-member with one parameter or a member with none.

15.2.2 The rules

C++ puts sensible limits on overloading [2]. You cannot invent new operators (there is no `**` for powers); you cannot change how many operands an operator takes, its precedence or its associativity (`a + b * c` always multiplies first); you cannot redefine operators on built-in types (`1 + 1` is always 2); and a few operators cannot be overloaded at all: `.` (member access), `::` (scope), `?:` and `sizeof`. **Table 15.1** lists the operators this chapter overloads, where they are usually written, and the Python method that plays the same role.

Table 15.1 The operators overloaded in this chapter, their usual form in C++ and their Python counterparts.

Operator	Typical form in C++	Returns	Python method
`a += b`, `a -= b`	member	`*this`, by reference	`__iadd__`, `__isub__` (optional)
`a + b`, `a - b`, `a * k`	non-member	a new object	`__add__`, `__sub__`, `__mul__`, `__rmul__`
`-a`	member	a new object	`__neg__`
`a == b`, `a != b`	non-member	`bool`	`__eq__`, `__ne__`
`a < b` and friends	non-member	`bool`	`__lt__` (+ `functools.total_ordering`)
`os << a`	non-member	`os`, by reference	`__str__`, `__format__`

Three guidelines decide the column “typical form” [3], [4].

- **Operators that change their left operand are members.** `a += b` changes `a`, so it needs access to `a`'s private data; it returns `*this` (the object itself) by reference so that it can be chained like the built-in version.
- **Symmetric operators are non-members, built on the members.** `a + b` does not change either operand, so it is written outside the class as "copy `a`, then `+= b`". This keeps the class small and treats both operands alike, which matters when a conversion is involved (Section 15.4 shows `2 + half`).
- **`<<` must be a non-member**, because its left operand is a stream, not your object, and you cannot add members to `std::ostream`.

Two more operators appear often in the standard library, and you have already used both. `operator[]` gives indexing: `grams[2]` on a `std::vector` and `services_[code]` on a `std::map` (Section 15.5) are calls of member functions named `operator[]`, which is why a map can decide to *create* a missing entry while a vector does not check its index at all; the behavior is whatever the function was written to do. `operator()` makes an object callable like a function; such objects are called *function objects*, and the small comparison functions that Chapter 16 passes to `std::stable_sort` are compiled into objects of exactly this kind. Neither is needed for the classes of this chapter, but recognizing them removes the last bit of magic from the standard library: every piece of special syntax you meet on a class is a function you could read.

The most important rule is not in the language at all: *an overloaded operator should do what a reader expects*. `+` on money adds money. Using `+` to mean "add an order to the shop" or `<<` to mean "send an e-mail" makes code shorter and much harder to read. When in doubt, write a named function.

15.3 A Money Class: `Rupiah`

15.3.1 The class

Listing 15.1 is the complete class. It is written entirely in a header, with every function `inline`, so that any program can use it by including one file.

Listing 15.1 `rupiah.h`: an amount of money in whole rupiah, with its operators.

```
// rupiah.h - Chapter 15: an amount of money in whole rupiah, with operators.
#ifndef RUPIAH_H
#define RUPIAH_H

#include <ostream>
#include <string>

class Rupiah {
public:
    Rupiah() = default;                          // Rp 0
    explicit Rupiah(long long amount) : amount_(amount) {}
    long long value() const { return amount_; }

    Rupiah& operator+=(const Rupiah& other) {  // members change *this
        amount_ += other.amount_;
        return *this;
    }
    Rupiah& operator-=(const Rupiah& other) {
        amount_ -= other.amount_;
        return *this;
    }
    Rupiah operator-() const { return Rupiah(-amount_); }

    // p percent of this amount, rounded half away from zero.
    Rupiah percent(int p) const {
        long long a = amount_ < 0 ? -amount_ : amount_;
        long long r = (a * p + 50) / 100;
        return Rupiah(amount_ < 0 ? -r : r);
    }

private:
    long long amount_ = 0;
};

// Non-members, built on the members above.
inline Rupiah operator+(Rupiah a, const Rupiah& b) { return a += b; }
inline Rupiah operator-(Rupiah a, const Rupiah& b) { return a -= b; }
inline Rupiah operator*(const Rupiah& a, long long k) {
    return Rupiah(a.value() * k);
}
inline Rupiah operator*(long long k, const Rupiah& a) { return a * k; }
```

Listing 15.1 (Continued) *rupiah.h: an amount of money in whole rupiah, with its operators.*

```
inline bool operator==(const Rupiah& a, const Rupiah& b) {
    return a.value() == b.value();
}
inline bool operator!=(const Rupiah& a, const Rupiah& b) { return !(a == b); }
inline bool operator<(const Rupiah& a, const Rupiah& b) {
    return a.value() < b.value();
}
inline bool operator>(const Rupiah& a, const Rupiah& b) { return b < a; }
inline bool operator<=(const Rupiah& a, const Rupiah& b) { return !(b < a); }
inline bool operator>=(const Rupiah& a, const Rupiah& b) { return !(a < b); }

// "Rp 1.234.500": Indonesian style, a dot between groups of three digits.
inline std::string toText(const Rupiah& r) {
    long long v = r.value();
    std::string digits = std::to_string(v < 0 ? -v : v);
    std::string out;
    int count = 0;
    for (int i = static_cast<int>(digits.size()) - 1; i >= 0; --i) {
        out.insert(out.begin(), digits[i]);
        if (++count % 3 == 0 && i > 0) out.insert(out.begin(), '.');
    }
    return (v < 0 ? "-Rp " : "Rp ") + out;
}

inline std::ostream& operator<<(std::ostream& os, const Rupiah& r) {
    return os << toText(r);                    // honors setw, left and right
}

#endif
```

Walk through it.

The data and the constructors. The only data member is `long long amount_`, whole rupiah. A `long long` holds amounts up to about 9.2×10^{18}, far more than any laundry needs, and whole numbers make every rounding decision explicit. `Rupiah() = default;` asks for the ordinary default constructor, which uses the member initializer `= 0`, so a new `Rupiah` is Rp 0. The second constructor is marked **`explicit`**: `Rupiah r(5000);` is allowed, but `Rupiah r = 5000;` and passing a bare `5000` where a `Rupiah` is expected are not. That is deliberate. With `explicit`, the compiler will never turn a weight or a count into money behind your back; you must say `Rupiah(5000)`.

Compound assignment as members. `operator+=` adds the other amount to this one and returns `*this` by reference. `this` is the pointer to the current object (Section 14.4.5), so `*this` is the object itself. Returning a reference is what makes `till += wash;` behave like the built-in `+=`.

Unary minus. `Rupiah operator-() const` returns the negated amount without changing the object. The receipt uses it to print a discount as a negative amount.

Percentages, one rule for everyone. `percent(p)` computes $p\%$ of the amount, rounded *half away from zero*. For a non-negative amount a the formula is

$$\text{percent}(a, p) = \left\lfloor \frac{a \cdot p + 50}{100} \right\rfloor \tag{15.1}$$

and for a negative amount the same is computed for $|a|$ and the sign put back, so that -10% of a refund rounds in the same way as 10% of a sale. Adding 50 before dividing by 100 is the same trick as adding 500 before dividing by 1000 in Chapter 13: it turns "cut off the fraction" into "round to the nearest", with halves going up. For example, 10% of Rp 24,150 is $\lfloor(241{,}500 + 50)/100\rfloor = \lfloor 2415.5 \rfloor = 2415$: the exact value 2,415.0 stays 2,415. And 50% of Rp 35,001 is

$\lfloor (1{,}750{,}050 + 50)/100 \rfloor = 17{,}501$, the half rupiah rounded up. Now every receipt in the shop rounds in exactly one way, because there is exactly one function that does it.

Arithmetic as non-members. `operator+(Rupiah a, const Rupiah& b)` takes its left operand *by value*, which makes a copy; it adds `b` to the copy with `+=` and returns it. One line, and all the real work stays in `+=`. `operator*` exists in two versions, `Rupiah * long long` and `long long * Rupiah`, so that both `rate * amount` and `3 * iron` work. There is no `Rupiah * Rupiah`, because money times money means nothing, and a mistake such as `price * price` does not compile.

Comparisons. `==` and `<` compare the amounts; the other four are written in terms of these two (`a > b` is `b < a`, `a <= b` is `!(b < a)`), so that they can never disagree. (C++20 can generate all six from one `operator<=>`; in C++17 we write them.)

Printing. `toText` builds the text from the right, inserting a dot after every three digits, and puts `Rp` or `-Rp` in front: 1234567890 becomes `Rp 1.234.567.890`, the way amounts are written in Indonesia. `operator<<` sends that text into the stream. Because it inserts a single string, the manipulators `std::setw`, `std::left` and `std::right` work on a `Rupiah` exactly as on a number, which is what lines up the receipts below.

15.3.2 The class at work

Listing 15.2 exercises every operator.

Listing 15.2 `rupiah_demo.cpp`: every operator of `Rupiah` in use.

```
// rupiah_demo.cpp - Chapter 15: the Rupiah class at work.
#include <iomanip>
#include <iostream>

#include "rupiah.h"

int main() {
    Rupiah wash(24150), iron(10000);
    Rupiah sum = wash + iron;                  // operator+(wash, iron)
    std::cout << "wash + iron      = " << sum << '\n';
    std::cout << "wash - iron      = " << wash - iron << '\n';
    std::cout << "iron - wash      = " << iron - wash << '\n';
    std::cout << "3 * iron         = " << 3 * iron << '\n';
    std::cout << "10% of the sum   = " << sum.percent(10) << '\n';
    std::cout << "big amount       = " << Rupiah(1234567890) << '\n';

    Rupiah till;                               // Rp 0
    till += wash;
    till += iron;
    till -= Rupiah(5000);
    std::cout << "till             = " << till << '\n';

    std::cout << std::boolalpha;
    std::cout << "wash == iron     : " << (wash == iron) << '\n';
    std::cout << "iron <  wash     : " << (iron < wash) << '\n';
    std::cout << "till >= sum      : " << (till >= sum) << '\n';

    std::cout << '[' << std::setw(14) << wash << "]\n";   // right aligned
    std::cout << '[' << std::left << std::setw(14) << wash << "]\n";
    // Rupiah x = 5000;   would not compile: the constructor is explicit
    // wash * wash;       would not compile: no operator for Rupiah * Rupiah
    return 0;
}
```

```
wash + iron      = Rp 34.150
wash - iron      = Rp 14.150
iron - wash      = -Rp 14.150
3 * iron         = Rp 30.000
10% of the sum   = Rp 3.415
big amount       = Rp 1.234.567.890
till             = Rp 29.150
wash == iron     : false
iron <  wash     : true
till >= sum      : false
[     Rp 24.150]
[Rp 24.150     ]
```

Each line is ordinary arithmetic to read, and each is a function call underneath. `3 * iron` found the non-member `operator*(long long, const Rupiah&)`. `iron - wash` produced a negative amount, printed with its sign in front of `Rp`. The two lines at the end show `setw(14)` right-aligning and left-aligning an amount inside brackets. The two commented lines show what the compiler refuses: `Rupiah x = 5000;` because the constructor is `explicit`, and `wash * wash` because no such operator exists. A type that refuses nonsense at compile time is worth more than any number of careful comments.

15.3.3 Tracing one expression by hand

It is worth following one expression all the way down once, because afterwards operators hold no mystery. Take the total of a receipt, `base + express - discount`, with `base` = Rp 27,500, `express` = Rp 13,750 and `discount` = Rp 4,125. Like the built-in operators, `+` and `-` of the same precedence are evaluated from left to right, so the expression means `(base + express) - discount`.

1. The compiler needs `operator+` for two `Rupiah`s. It finds the non-member `operator+(Rupiah a, const Rupiah& b)`. The parameter `a` is a *copy* of `base` (27,500), and `b` is another name for `express`.
2. Inside, `a += b` calls the member `operator+=` on the copy: `amount_` becomes $27{,}500 + 13{,}750 = 41{,}250$, and the function returns a reference to the copy.
3. `operator+` returns that copy as its result, a temporary `Rupiah` holding 41,250. `base` itself was never touched.
4. The compiler now needs `operator-` for the temporary and `discount`. It finds the non-member `operator-`, which copies the temporary, calls `-=` with 4,125, and returns a new temporary holding $41{,}250 - 4{,}125 = 37{,}125$.
5. The temporaries die at the end of the full statement; for a class whose only member is a `long long`, the compiler optimizes all of this into the same few machine instructions as the plain integer version.

So the expression costs no more than integer arithmetic, and yet it cannot be written with a weight or a count by mistake. That combination, safety at no run-time cost, is the reason C++ programmers use small classes like `Rupiah` so freely.

15.3.4 The same class in Python

Python has operator overloading too; it simply uses fixed method names, the *special* or "dunder" (double underscore) methods of its data model [5]. `a + b` calls `a.__add__(b)`; `3 * a` first tries `int`'s multiplication, which does not know `Rupiah`, and then calls `a.__rmul__(3)` ("reflected multiply"); `a < b` calls `a.__lt__(b)`; `str(a)` calls `a.__str__()`; and an f-string field such as `{wash:>14}` calls `wash.__format__(">14")`. The decorator `@functools.total_ordering` derives

`<=`, `>` and `>=` from `__eq__` and `__lt__`, exactly as the C++ header does by hand. **Listing 15.3** is the class, and **Listing 15.4** prints the same output as the C++ demo.

Listing 15.3 `rupiah.py`: the `Rupiah` class in Python.

```
# rupiah.py - Chapter 15: an amount of money in whole rupiah, with operators.
from functools import total_ordering


@total_ordering                                 # derives <=, >, >= from == and <
class Rupiah:
    __slots__ = ("_amount",)

    def __init__(self, amount=0):
        self._amount = int(amount)

    def value(self):
        return self._amount

    def __add__(self, other):
        return Rupiah(self._amount + other._amount)

    def __sub__(self, other):
        return Rupiah(self._amount - other._amount)

    def __neg__(self):
        return Rupiah(-self._amount)

    def __mul__(self, k):                       # Rupiah * int
        return Rupiah(self._amount * k)

    __rmul__ = __mul__                          # int * Rupiah

    def __eq__(self, other):
        return isinstance(other, Rupiah) and self._amount == other._amount

    def __lt__(self, other):
        return self._amount < other._amount

    __hash__ = None                             # changeable, so not a dict key

    def percent(self, p):
        """p percent of this amount, rounded half away from zero."""
        a = abs(self._amount)
        r = (a * p + 50) // 100
        return Rupiah(-r if self._amount < 0 else r)

    def __str__(self):                          # "Rp 1.234.500"
        text = f"{abs(self._amount):,}".replace(",", ".")
        return ("-Rp " if self._amount < 0 else "Rp ") + text
```

Listing 15.3 (Continued) *rupiah.py: the Rupiah class in Python.*

```
    def __format__(self, spec):                 # lets f"{r:>12}" align it
        return format(str(self), spec)
```

Listing 15.4 `rupiah_demo.py`: the Python demo prints exactly what `rupiah_demo.cpp` prints.

```
# rupiah_demo.py - Chapter 15: the Rupiah class at work.
from rupiah import Rupiah

wash, iron = Rupiah(24150), Rupiah(10000)
total = wash + iron                          # wash.__add__(iron)
print(f"wash + iron      = {total}")
print(f"wash - iron      = {wash - iron}")
print(f"iron - wash      = {iron - wash}")
print(f"3 * iron         = {3 * iron}")
print(f"10% of the sum   = {total.percent(10)}")
print(f"big amount       = {Rupiah(1234567890)}")

till = Rupiah()                              # Rp 0
till += wash                                 # Python falls back on __add__
till += iron
till -= Rupiah(5000)
print(f"till             = {till}")

print(f"wash == iron     : {str(wash == iron).lower()}")
print(f"iron <  wash     : {str(iron < wash).lower()}")
print(f"till >= sum      : {str(till >= total).lower()}")

print(f"[{wash:>14}]")                       # right aligned
print(f"[{wash:<14}]")
```

Two Python details deserve a word. `__hash__ = None` says that a `Rupiah` cannot be used as a dictionary key; the rule in Python is that objects that define `__eq__` must decide about hashing, and money that can be compared is not something we want as a key anyway. And the thousands separator comes from Python's own format mini-language: `f"{24150:,}"` gives `24,150`, and replacing the comma with a dot gives the Indonesian form.

15.4 A Second Example: Fractions and Euclid's Algorithm

Money needs only whole numbers. Some quantities are naturally fractions: half a scoop of detergent plus a third of a scoop of softener, three quarters of a bottle of perfume. A `Fraction` class is the classic exercise in operator overloading, and it brings back one of the oldest algorithms in the world.

A fraction $\frac{a}{b}$ is kept in *lowest terms*: $\frac{6}{8}$ is stored as $\frac{3}{4}$. To reduce a fraction we divide the numerator and the denominator by their *greatest common divisor* (gcd), the largest number that divides both. Euclid described how to find it around 300 BC [6]: the gcd does not change if the larger number is replaced by its remainder when divided by the smaller one,

$$\gcd(a, b) = \gcd(b,\ a \bmod b), \qquad \gcd(a, 0) = a \tag{15.2}$$

For 270 and 192: $\gcd(270,192) = \gcd(192,78) = \gcd(78,36) = \gcd(36,6) = \gcd(6,0) = 6$. Each step makes the numbers smaller, quickly: the number of steps grows only with the number of *digits*, not with the size of the numbers. Why is it true? Any number that divides both a and b also divides $a - qb$ for any whole q, in particular the remainder $a \bmod b$; and any number that divides b and the remainder also divides $a = qb + (a \bmod b)$. So the pairs (a, b) and $(b, a \bmod b)$ have exactly the same common divisors, and therefore the same greatest one.

The *least common multiple* (lcm), the smallest number that both divide, follows from the gcd:

$$\operatorname{lcm}(a, b) = \frac{a}{\gcd(a, b)} \cdot b \tag{15.3}$$

Dividing *first* matters in a program. $\frac{a \cdot b}{\gcd(a,b)}$ is the same number mathematically, but the product $a \cdot b$ can overflow even when the lcm itself fits; $a/\gcd(a,b)$ is always a whole number, and multiplying it by b gives the answer with a smaller intermediate value. Adding fractions uses the lcm of the denominators as the common denominator:

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot \frac{m}{b} + c \cdot \frac{m}{d}}{m}, \qquad m = \operatorname{lcm}(b,d) \tag{15.4}$$

For $\frac{1}{2} + \frac{1}{3}$: $m = 6$, and the sum is $\frac{1 \cdot 3 + 1 \cdot 2}{6} = \frac{5}{6}$. **Listing 15.5** is the class with its operators.

Listing 15.5 `fraction.cpp`: a `Fraction` class built on Euclid's gcd, with arithmetic, comparison and output operators.

```
// fraction.cpp - Chapter 15: a Fraction class built on Euclid's gcd.
#include <iostream>
#include <stdexcept>

long long gcd(long long a, long long b) {     // Euclid, about 300 BC
    if (a < 0) a = -a;
    if (b < 0) b = -b;
    while (b != 0) {
        long long r = a % b;
        a = b;
        b = r;
    }
    return a;
}

long long lcm(long long a, long long b) {
    return a / gcd(a, b) * b;                  // divide first: no overflow
}

class Fraction {
public:
    Fraction(long long num = 0, long long den = 1) : num_(num), den_(den) {
        if (den_ == 0) throw std::invalid_argument("denominator is zero");
        if (den_ < 0) { num_ = -num_; den_ = -den_; }   // sign lives on top
        long long g = gcd(num_, den_);
        if (g > 1) { num_ /= g; den_ /= g; }            // always in lowest terms
    }
    long long num() const { return num_; }
    long long den() const { return den_; }

private:
    long long num_, den_;
};

Fraction operator+(const Fraction& a, const Fraction& b) {
    long long d = lcm(a.den(), b.den());       // the least common denominator
    return Fraction(a.num() * (d / a.den()) + b.num() * (d / b.den()), d);
}
Fraction operator-(const Fraction& a, const Fraction& b) {
    return a + Fraction(-b.num(), b.den());
}
Fraction operator*(const Fraction& a, const Fraction& b) {
    return Fraction(a.num() * b.num(), a.den() * b.den());
}
Fraction operator/(const Fraction& a, const Fraction& b) {
    return Fraction(a.num() * b.den(), a.den() * b.num());
```

Listing 15.5 (Continued) *fraction.cpp: a Fraction class built on Euclid's gcd, with arithmetic, comparison and output operators.*

```
}
bool operator==(const Fraction& a, const Fraction& b) {
    return a.num() == b.num() && a.den() == b.den();   // lowest terms: unique
}
bool operator<(const Fraction& a, const Fraction& b) {
    return a.num() * b.den() < b.num() * a.den();      // both dens positive
}
std::ostream& operator<<(std::ostream& os, const Fraction& f) {
    if (f.den() == 1) return os << f.num();
    return os << f.num() << '/' << f.den();
}

int main() {
    long long pairs[][2] = {{12, 18}, {7, 13}, {100, 75}, {270, 192}};
    for (auto& p : pairs)
        std::cout << "gcd(" << p[0] << ", " << p[1] << ") = " << gcd(p[0], p[1])
                  << ", lcm = " << lcm(p[0], p[1]) << '\n';

    Fraction half(1, 2), third(1, 3), soap(3, 4);
    std::cout << "1/2 + 1/3   = " << half + third << '\n';
    std::cout << "1/2 - 1/3   = " << half - third << '\n';
    std::cout << "3/4 * 2/3   = " << soap * Fraction(2, 3) << '\n';
    std::cout << "3/4 / 1/2   = " << soap / half << '\n';
    std::cout << "6/8 == 3/4  : " << (Fraction(6, 8) == soap ? "yes" : "no")
              << '\n';
    std::cout << "1/3 < 1/2   : " << (third < half ? "yes" : "no") << '\n';
    std::cout << "2 + 1/2     = " << 2 + half << '\n';   // 2 becomes Fraction(2)
    std::cout << "4/-6        = " << Fraction(4, -6) << '\n';
    try {
        std::cout << half / Fraction(0) << '\n';
    } catch (const std::invalid_argument& e) {
        std::cout << "1/2 / 0     : refused, " << e.what() << '\n';
    }
    return 0;
}
```

```
gcd(12, 18) = 6, lcm = 36
gcd(7, 13) = 1, lcm = 91
gcd(100, 75) = 25, lcm = 300
gcd(270, 192) = 6, lcm = 8640
1/2 + 1/3   = 5/6
1/2 - 1/3   = 1/6
3/4 * 2/3   = 1/2
3/4 / 1/2   = 3/2
6/8 == 3/4  : yes
1/3 < 1/2   : yes
2 + 1/2     = 5/2
4/-6        = -2/3
1/2 / 0     : refused, denominator is zero
```

Three design decisions make the class easy to trust.

- **The invariant is enforced in the constructor.** A denominator of zero throws; a negative denominator moves the sign to the numerator; and every fraction is reduced by the gcd. Because *every* fraction passes through the constructor, including the results of `+`, `-`, `*` and `/`, every fraction in the program is always in lowest terms with a positive denominator.
- **That invariant makes `==` trivial.** Two fractions in lowest terms are equal exactly when their numerators and denominators are equal, so $\frac{6}{8} = \frac{3}{4}$ is decided by comparing $3 = 3$ and $4 = 4$. Without the invariant, `==` would need cross-multiplication. And `<` can cross-multiply safely, $\frac{a}{b} <$

$\frac{c}{d} \Leftrightarrow ad < cb$, only because both denominators are positive; with a negative denominator the inequality would flip.

- **This constructor is *not* `explicit`, on purpose.** `Fraction(long long num = 0, long long den = 1)` can be called with one argument, so it also works as a conversion from a whole number: in `2 + half`, the compiler turns `2` into `Fraction(2)` (that is, $\frac{2}{1}$) and calls the non-member `operator+`. This works only because `operator+` is a non-member: a member `operator+` would need a `Fraction` on the left, and `2` is not one. For fractions, "every whole number is a fraction" is true, so the conversion is welcome. For money it was not ("every whole number is an amount of rupiah" is how weights become prices), which is why `Rupiah`'s constructor is `explicit`. Choosing between the two is a design decision you make for every one-argument constructor.

Listing 15.6 is the Python version. Python needs `__radd__` for `2 + half`, since `int.__add__` does not know our class, and a small helper `_of` converts a whole number on the right into a `Fraction`. (The standard library has a complete `fractions.Fraction` [7]; building our own shows what is inside it.)

Listing 15.6 `fraction.py`: the `Fraction` class in Python.

```python
# fraction.py - Chapter 15: a Fraction class built on Euclid's gcd.
# (Python's standard library has fractions.Fraction; we build our own to learn.)


def gcd(a, b):                                  # Euclid, about 300 BC
    a, b = abs(a), abs(b)
    while b != 0:
        a, b = b, a % b
    return a


def lcm(a, b):
    return a // gcd(a, b) * b


class Fraction:
    def __init__(self, num=0, den=1):
        if den == 0:
            raise ValueError("denominator is zero")
        if den < 0:                             # sign lives on top
            num, den = -num, -den
        g = gcd(num, den)
        if g > 1:                               # always in lowest terms
            num, den = num // g, den // g
        self.num, self.den = num, den

    @staticmethod
    def _of(x):
        return x if isinstance(x, Fraction) else Fraction(x)

    def __add__(self, other):
        other = Fraction._of(other)
        d = lcm(self.den, other.den)
        top = self.num * (d // self.den) + other.num * (d // other.den)
        return Fraction(top, d)

    __radd__ = __add__                          # 2 + half

    def __sub__(self, other):
        other = Fraction._of(other)
        return self + Fraction(-other.num, other.den)

    def __mul__(self, other):
        other = Fraction._of(other)
        return Fraction(self.num * other.num, self.den * other.den)
```

Listing 15.6 (Continued) *fraction.py: the Fraction class in Python.*

```
    def __truediv__(self, other):
        other = Fraction._of(other)
        return Fraction(self.num * other.den, self.den * other.num)

    def __eq__(self, other):
        other = Fraction._of(other)
        return self.num == other.num and self.den == other.den

    def __lt__(self, other):
        other = Fraction._of(other)
        return self.num * other.den < other.num * self.den

    def __str__(self):
        return str(self.num) if self.den == 1 else f"{self.num}/{self.den}"


for a, b in [(12, 18), (7, 13), (100, 75), (270, 192)]:
    print(f"gcd({a}, {b}) = {gcd(a, b)}, lcm = {lcm(a, b)}")

half, third, soap = Fraction(1, 2), Fraction(1, 3), Fraction(3, 4)
print(f"1/2 + 1/3   = {half + third}")
print(f"1/2 - 1/3   = {half - third}")
print(f"3/4 * 2/3   = {soap * Fraction(2, 3)}")
print(f"3/4 / 1/2   = {soap / half}")
print(f"6/8 == 3/4  : {'yes' if Fraction(6, 8) == soap else 'no'}")
print(f"1/3 < 1/2   : {'yes' if third < half else 'no'}")
print(f"2 + 1/2     = {2 + half}")
print(f"4/-6        = {Fraction(4, -6)}")
try:
    print(half / Fraction(0))
except ValueError as e:
    print(f"1/2 / 0     : refused, {e}")
```

15.5 The Complete Case Study: Profitina Laundry

Now we put Chapters 13, 14 and 15 together into one program that Bu Rina could use at the end of every day. **Figure 15.2** is its plan.

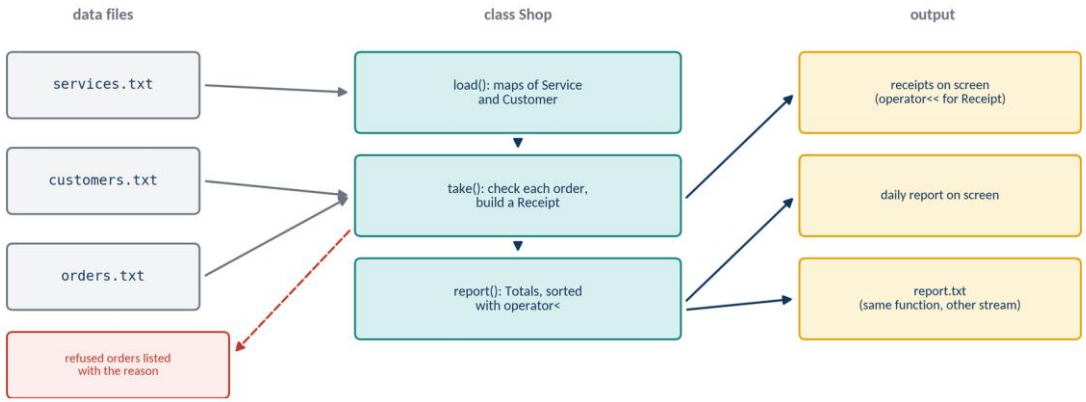


Figure 15.2 The case study: three data files are loaded into the `Shop`, each order is checked and turned into a `Receipt`, and the receipts are printed and summarized in a report written both to the screen and to `report.txt`.

15.5.1 The data

The price list is no longer inside the program. `data/services.txt` has one line per service: a short code, a name, a unit (`kg` or `pcs`) and a rate in rupiah.

```
# code;name;unit;rate in rupiah per kg or per piece
W;Wash & fold;kg;7000
WI;Wash & iron;kg;10000
I;Iron only;kg;5000
BC;Bed cover;pcs;25000
SH;Shoes;pcs;35000
```

`data/customers.txt` lists the customers and whether they are members, who receive a 10% discount:

```
# id;name;member (yes/no)
C01;Bu Rina Wulandari;yes
C02;Pak Budi Santoso;no
C03;Mbak Sari Dewi;yes
C04;Mas Joko Prasetyo;no
```

`data/orders.txt` is the day's orders: the order number, the customer's id, the service code, the amount (grams for `kg` services, pieces for `pcs` services) and whether it is express. Two lines are wrong on purpose: order 1006 names a customer who does not exist, and order 1009 a service that does not exist.

```
# number;customer;service;amount (grams or pieces);express (yes/no)
1001;C01;W;3450;no
1002;C02;WI;5200;yes
1003;C03;BC;2;no
1004;C01;I;2000;no
1005;C04;SH;1;yes
1006;C05;W;3000;no
1007;C02;W;4100;no
1008;C03;WI;2750;yes
1009;C04;DC;1000;no
```

15.5.2 The rules

The price of one order is computed in three steps. The *base price* is the rate times the amount, with the rounding of Chapter 13 for services priced per kilogram. The *express surcharge* is 50% of the base price, by Equation (15.1), for express orders, and zero otherwise. The *member discount* is 10% of base price plus surcharge, again by Equation (15.1), for members, and zero otherwise. The total is

$$\text{total} = \text{base} + \text{express} - \text{discount} \tag{15.5}$$

and because all four quantities are `Rupiah` objects, this line appears in the program literally as `base + express - discount`.

15.5.3 The program

Listing 15.7 is the complete program. It includes `rupiah.h` from Section 15.3 unchanged.

Listing 15.7 `laundry.cpp`: the complete Profitina Laundry program in C++.

```
// laundry.cpp - Chapter 15: the complete Profitina Laundry case study.
// Reads services, customers and orders from data/, prints one receipt per
// order and the daily report, and saves the report to report.txt.
// Build: g++ -std=c++17 -O2 -Wall -Wextra laundry.cpp -o laundry
#include <algorithm>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <vector>

#include "rupiah.h"

std::vector<std::string> split(const std::string& line, char sep) {
    std::vector<std::string> parts;
    std::string part;
    std::istringstream in(line);
    while (std::getline(in, part, sep)) parts.push_back(part);
    if (!line.empty() && line.back() == sep) parts.push_back("");
    return parts;
}

// Reads the non-comment lines of a data file, each split at ';'.
std::vector<std::vector<std::string>> readTable(const std::string& path) {
    std::vector<std::vector<std::string>> rows;
    std::ifstream in(path);
    std::string line;
    while (std::getline(in, line))
        if (!line.empty() && line[0] != '#') rows.push_back(split(line, ';'));
    return rows;
}

bool isNumber(const std::string& s) {
    if (s.empty() || s.size() > 6) return false;
    for (char c : s)
        if (c < '0' || c > '9') return false;
    return true;
}
```

Listing 15.7 (Continued) *laundry.cpp: the complete Profitina Laundry program in C++.*

```
struct Service {
    std::string code, name, unit;              // unit is "kg" or "pcs"
    Rupiah rate;
    Rupiah price(int amount) const {           // amount: grams or pieces
        if (unit == "kg") return Rupiah((amount * rate.value() + 500) / 1000);
        return rate * amount;
    }
    std::string describe(int amount) const {   // "3.450 kg" or "2 pcs"
        if (unit != "kg") return std::to_string(amount) + " pcs";
        std::string g = std::to_string(amount % 1000);
        return std::to_string(amount / 1000) + "." + std::string(3 - g.size(), '0')
               + g + " kg";
    }
};

struct Customer {
    std::string id, name;
    bool member;
};

struct Receipt {
    std::string number;
    const Customer* customer;
    const Service* service;
    int amount;
    Rupiah base, express, discount;
    Rupiah total() const { return base + express - discount; }
};

std::string line40(const std::string& label, const Rupiah& money) {
    std::ostringstream out;
    out << "  " << std::left << std::setw(24) << label << std::right
        << std::setw(14) << money;
    return out.str();
}
```

Listing 15.7 (Continued) *laundry.cpp: the complete Profitina Laundry program in C++.*

```
std::ostream& operator<<(std::ostream& os, const Receipt& r) {
    os << std::string(40, '=') << '\n'
       << std::left << std::setw(31) << "PROFITINA LAUNDRY" << "No. " << r.number
       << '\n'
       << r.customer->name << (r.customer->member ? " (member)" : "") << '\n'
       << r.service->name << ", " << r.service->describe(r.amount) << " x "
       << r.service->rate << '\n'
       << line40("Base price", r.base) << '\n';
    if (r.express.value() != 0) os << line40("Express +50%", r.express) << '\n';
    if (r.discount.value() != 0)
        os << line40("Member discount 10%", -r.discount) << '\n';
    return os << line40("TOTAL", r.total()) << '\n';
}

struct Total {                                  // one line of the daily report
    std::string key, name;
    int count = 0;
    Rupiah money;
};

bool operator<(const Total& a, const Total& b) {   // largest money first
    if (a.money != b.money) return a.money > b.money;
    return a.key < b.key;
}

class Shop {
public:
    void load(const std::string& dir) {
        for (auto& f : readTable(dir + "/services.txt"))
            if (f.size() == 4 && isNumber(f[3]))
                services_[f[0]] =
                    Service{f[0], f[1], f[2], Rupiah(std::stoll(f[3]))};
        for (auto& f : readTable(dir + "/customers.txt"))
            if (f.size() == 3)
                customers_[f[0]] = Customer{f[0], f[1], f[2] == "yes"};
        for (auto& f : readTable(dir + "/orders.txt")) take(f);
    }

    void printReceipts(std::ostream& os) const {
        for (const Receipt& r : receipts_) os << r;
        os << std::string(40, '=') << '\n';
        for (const std::string& why : refused_) os << "refused: " << why << '\n';
    }
```

Listing 15.7 (Continued) *laundry.cpp: the complete Profitina Laundry program in C++.*

```
    void report(std::ostream& os) const {
        std::map<std::string, Total> byService, byCustomer;
        Rupiah grand, discounts;
        for (const Receipt& r : receipts_) {
            add(byService[r.service->code], r.service->code, r.service->name, r);
            add(byCustomer[r.customer->id], r.customer->id, r.customer->name, r);
            grand += r.total();
            discounts += r.discount;
        }
        os << "DAILY REPORT - " << receipts_.size() << " receipts, "
           << refused_.size() << " orders refused\n";
        section(os, "Revenue by service (largest first)", byService);
        section(os, "Revenue by customer (largest first)", byCustomer);
        os << std::left << std::setw(26) << "Member discounts given" << std::right
           << std::setw(14) << discounts << '\n'
           << std::left << std::setw(26) << "GRAND TOTAL" << std::right
           << std::setw(14) << grand << '\n';
    }

private:
    void take(const std::vector<std::string>& f) {
        std::string no = f.empty() ? "?" : f[0];
        if (f.size() != 5) return refuse(no, "expected 5 fields");
        auto c = customers_.find(f[1]);
        if (c == customers_.end()) return refuse(no, "unknown customer " + f[1]);
        auto s = services_.find(f[2]);
        if (s == services_.end()) return refuse(no, "unknown service " + f[2]);
        if (!isNumber(f[3]) || std::stoi(f[3]) < 1) return refuse(no, "bad amount");
        if (f[4] != "yes" && f[4] != "no") return refuse(no, "express is yes/no");

        Receipt r{no, &c->second, &s->second, std::stoi(f[3]), {}, {}, {}};
        r.base = r.service->price(r.amount);
        if (f[4] == "yes") r.express = r.base.percent(50);
        if (r.customer->member) r.discount = (r.base + r.express).percent(10);
        receipts_.push_back(r);
    }

    void refuse(const std::string& no, const std::string& why) {
        refused_.push_back("order " + no + ", " + why);
    }
```

Listing 15.7 (Continued) *laundry.cpp: the complete Profitina Laundry program in C++.*

```
    static void add(Total& t, const std::string& key, const std::string& name,
                    const Receipt& r) {
        t.key = key;
        t.name = name;
        ++t.count;
        t.money += r.total();
    }

    static void section(std::ostream& os, const std::string& title,
                        const std::map<std::string, Total>& totals) {
        std::vector<Total> rows;
        for (const auto& kv : totals) rows.push_back(kv.second);
        std::sort(rows.begin(), rows.end());   // uses our operator<
        os << title << '\n';
        for (const Total& t : rows)
            os << "  " << std::left << std::setw(4) << t.key << std::setw(20)
               << t.name << std::right << std::setw(2) << t.count << std::setw(12)
               << t.money << '\n';
    }

    std::map<std::string, Service> services_;
    std::map<std::string, Customer> customers_;
    std::vector<Receipt> receipts_;
    std::vector<std::string> refused_;
};

int main(int argc, char* argv[]) {
    std::string dir = argc > 1 ? argv[1] : "data";
    Shop shop;
    shop.load(dir);
    shop.printReceipts(std::cout);
    std::cout << '\n';
    shop.report(std::cout);                    // the same function writes...
    std::ofstream file("report.txt");
    shop.report(file);                         // ...to the screen and to a file
    std::cout << "\nreport saved to report.txt\n";
    return 0;
}
```

The program is about 200 lines. Here is how the pieces work together.

Reading the files. `readTable` opens a file with `std::ifstream`, skips empty lines and comments, and splits every other line at `;` with the `split` of Chapter 14. The result is a vector of rows, each row a vector of strings. One small improvement over Chapter 14: if a line ends with `;`, `split` adds an empty last field, so that a line with an empty last column still has the right number of fields and is checked properly. If a file is missing, the stream simply delivers no lines, and the shop starts with an empty price list, which makes every order be refused with a clear reason.

`Service` and `Customer`. Two small structures with no invariant to guard: after loading, nobody changes them. `Service::price` computes the base price, as a `Rupiah`, for grams or pieces, and `describe` prints the amount in its unit.

`Receipt`. One accepted order with all its money. The receipt holds *pointers* to its customer and service rather than copies, because the customer and service already exist in the shop's maps and must not be duplicated. `total()` is Equation (15.5).

Printing a receipt with `<<`. `operator<<(std::ostream& os, const Receipt& r)` prints the whole receipt block: a ruler, the shop name with the order number, the customer (with "(member)" when it applies), the service line, and then one line for each amount, with the express and discount lines only when they are not zero. Each amount line is formatted by `line40`, which

uses a `std::ostringstream`, a stream that writes into a string, to right-align a `Rupiah` with `setw(14)`. Because the operator takes any `std::ostream&`, the same code could print a receipt into a file or a string.

`Total` and its `operator<`. One line of the report: a key (service code or customer id), a name, a count and an amount. The report must list the largest amounts first, and for equal amounts in the order of the key. We express that order *once*, as `operator<` for `Total`: "`a` comes before `b` if `a` has more money, or the same money and a smaller key". Then `std::sort(rows.begin(), rows.end())`, which uses `<` whenever it compares two elements, sorts the report without any further instructions. This is overloading at its most useful: we taught a standard algorithm what "before" means for our type.

`Shop`. The class that owns everything. Its private data are two `std::map`s and two vectors. A `std::map<std::string, Service>` is a *dictionary*: it stores pairs of a key (the service code) and a value (the `Service`), keeps them sorted by key, and finds a key in $O(\log n)$ steps. `services_[code] = service` inserts or replaces; `services_.find(code)` returns an iterator to the pair, or `services_.end()` when the key is absent; `it->second` is the value of the pair. `load` fills the maps and then calls `take` for every order line. `take` checks the fields one by one, returning early with `refuse` at the first problem, exactly like `parseOrder` in Chapter 13; when all is well, it builds a `Receipt` and computes its three amounts with the rules of Section 15.5.2. Notice how the checks read: `if (c == customers_.end()) return refuse(no, "unknown customer " + f[1]);`, one line per rule.

The report. `report` groups the receipts twice, by service and by customer, with a `std::map<std::string, Total>`: `byService[r.service->code]` finds the total for that code, *creating an empty one the first time*, which is the one surprising property of `operator[]` on a map, and here a convenient one. The static helper `section` copies the totals into a vector, sorts it with our `operator<` and prints it. At the end come the discounts given and the grand total.

One report, two destinations. `report` takes a `std::ostream&`. `main` calls it twice, once with `std::cout` and once with a `std::ofstream` opened on `report.txt`. Both `std::cout` and `std::ofstream` *are* `std::ostream`s (the standard library uses the inheritance of Section 14.6 for exactly this), so the same function writes to the screen and to the file, and the two can never differ. The file is closed by the `ofstream`'s destructor when `main` ends.

15.5.4 Following two orders through `take`

Follow order 1006 first. `readTable` has already split its line into the five strings `1006`, `C05`, `W`, `3000` and `no`, so the field count is right. `customers_.find("C05")` walks down the map's sorted keys, C01, C02, C03, C04, finds no C05 and returns `customers_.end()`. The test `c == customers_.end()` is true, so `take` calls `refuse("1006", "unknown customer C05")`, which appends the text to `refused_`, and returns. Nothing else happens: no receipt, no partial totals. The service, the amount and the express flag of the line are never examined, because the first problem is enough to refuse the order.

Now order 1002, `1002;C02;WI;5200;yes`. The customer C02 is found (Pak Budi Santoso, not a member), the service WI is found (wash and iron, 10,000 per kg), `5200` is a number of at most six digits and at least 1, and `yes` is a valid express flag. `take` builds the receipt with pointers to the customer and the service, computes the base price $(5200 \times 10000 + 500)/1000 = 52{,}000$, the express surcharge `r.base.percent(50)`, which is 26,000, and no discount, because `r.customer->member` is false. The receipt goes to the end of `receipts_`, and later `operator<<` prints it with the

lines “Base price”, “Express +50%” and “TOTAL”, skipping the discount line because the discount is zero.

Listing 15.8 is the same program in Python. `@dataclass` generates the constructors of the plain record classes; `Receipt.__str__` plays the role of `operator<<`; `Total.__lt__` is the sort order, used by `sorted`; `dict` replaces `std::map` (and `setdefault` replaces the create-on-first-use of `[]`); and the report function takes any object with a `write` method, so it writes to `sys.stdout` and to a file just as the C++ version writes to `std::cout` and to an `ofstream`.

Listing 15.8 `laundry.py`: the complete Profitina Laundry program in Python.

```
# laundry.py - Chapter 15: the complete Profitina Laundry case study in Python.
# Reads services, customers and orders from data/, prints one receipt per
# order and the daily report, and saves the report to report.txt.
import sys
from dataclasses import dataclass, field

from rupiah import Rupiah


def read_table(path):
    """The non-comment lines of a data file, each split at ';'."""
    with open(path, encoding="utf-8") as f:
        return [line.split(";") for line in f.read().splitlines()
                if line and not line.startswith("#")]


def is_number(s):
    return 0 < len(s) <= 6 and all(c in "0123456789" for c in s)


@dataclass
class Service:
    code: str
    name: str
    unit: str                                   # "kg" or "pcs"
    rate: Rupiah

    def price(self, amount):                    # amount: grams or pieces
        if self.unit == "kg":
            return Rupiah((amount * self.rate.value() + 500) // 1000)
        return self.rate * amount

    def describe(self, amount):                 # "3.450 kg" or "2 pcs"
        if self.unit != "kg":
            return f"{amount} pcs"
        return f"{amount // 1000}.{amount % 1000:03d} kg"


@dataclass
class Customer:
    id: str
    name: str
    member: bool
```

Listing 15.8 (Continued) *laundry.py: the complete Profitina Laundry program in Python.*

```
@dataclass
class Receipt:
    number: str
    customer: Customer
    service: Service
    amount: int
    base: Rupiah = field(default_factory=Rupiah)
    express: Rupiah = field(default_factory=Rupiah)
    discount: Rupiah = field(default_factory=Rupiah)

    def total(self):
        return self.base + self.express - self.discount

    def __str__(self):                          # plays the role of operator<<
        lines = ["=" * 40, f"{'PROFITINA LAUNDRY':<31}No. {self.number}",
                 self.customer.name + (" (member)" if self.customer.member else ""),
                 f"{self.service.name}, {self.service.describe(self.amount)} x "
                 f"{self.service.rate}", line40("Base price", self.base)]
        if self.express.value() != 0:
            lines.append(line40("Express +50%", self.express))
        if self.discount.value() != 0:
            lines.append(line40("Member discount 10%", -self.discount))
        lines.append(line40("TOTAL", self.total()))
        return "\n".join(lines) + "\n"


def line40(label, money):
    return f"  {label:<24}{money:>14}"


@dataclass
class Total:                                    # one line of the daily report
    key: str = ""
    name: str = ""
    count: int = 0
    money: Rupiah = field(default_factory=Rupiah)

    def __lt__(self, other):                    # largest money first
        if self.money != other.money:
            return self.money > other.money
        return self.key < other.key
```

Listing 15.8 (Continued) *laundry.py: the complete Profitina Laundry program in Python.*

```
class Shop:
    def __init__(self):
        self._services, self._customers = {}, {}
        self._receipts, self._refused = [], []

    def load(self, folder):
        for f in read_table(folder + "/services.txt"):
            if len(f) == 4 and is_number(f[3]):
                self._services[f[0]] = Service(f[0], f[1], f[2], Rupiah(int(f[3])))
        for f in read_table(folder + "/customers.txt"):
            if len(f) == 3:
                self._customers[f[0]] = Customer(f[0], f[1], f[2] == "yes")
        for f in read_table(folder + "/orders.txt"):
            self._take(f)

    def _take(self, f):
        no = f[0] if f else "?"
        if len(f) != 5:
            return self._refuse(no, "expected 5 fields")
        c = self._customers.get(f[1])
        if c is None:
            return self._refuse(no, "unknown customer " + f[1])
        s = self._services.get(f[2])
        if s is None:
            return self._refuse(no, "unknown service " + f[2])
        if not is_number(f[3]) or int(f[3]) < 1:
            return self._refuse(no, "bad amount")
        if f[4] not in ("yes", "no"):
            return self._refuse(no, "express is yes/no")
        r = Receipt(no, c, s, int(f[3]))
        r.base = s.price(r.amount)
        if f[4] == "yes":
            r.express = r.base.percent(50)
        if c.member:
            r.discount = (r.base + r.express).percent(10)
        self._receipts.append(r)

    def _refuse(self, no, why):
        self._refused.append(f"order {no}, {why}")
```

Listing 15.8 (Continued) *laundry.py: the complete Profitina Laundry program in Python.*

```
    def print_receipts(self, out):
        for r in self._receipts:
            out.write(str(r))
        out.write("=" * 40 + "\n")
        for why in self._refused:
            out.write(f"refused: {why}\n")

    def report(self, out):
        by_service, by_customer = {}, {}
        grand, discounts = Rupiah(), Rupiah()
        for r in self._receipts:
            _add(by_service.setdefault(r.service.code, Total()),
                 r.service.code, r.service.name, r)
            _add(by_customer.setdefault(r.customer.id, Total()),
                 r.customer.id, r.customer.name, r)
            grand += r.total()
            discounts += r.discount
        out.write(f"DAILY REPORT - {len(self._receipts)} receipts, "
                  f"{len(self._refused)} orders refused\n")
        _section(out, "Revenue by service (largest first)", by_service)
        _section(out, "Revenue by customer (largest first)", by_customer)
        out.write(f"{'Member discounts given':<26}{discounts:>14}\n")
        out.write(f"{'GRAND TOTAL':<26}{grand:>14}\n")


def _add(t, key, name, r):
    t.key, t.name = key, name
    t.count += 1
    t.money += r.total()


def _section(out, title, totals):
    out.write(title + "\n")
    for t in sorted(totals.values()):          # uses our __lt__
        out.write(f"  {t.key:<4}{t.name:<20}{t.count:>2}{t.money:>12}\n")
```

Listing 15.8 (Continued) *laundry.py: the complete Profitina Laundry program in Python.*

```
def main():
    folder = sys.argv[1] if len(sys.argv) > 1 else "data"
    shop = Shop()
    shop.load(folder)
    shop.print_receipts(sys.stdout)
    print()
    shop.report(sys.stdout)                    # the same function writes...
    with open("report.txt", "w", encoding="utf-8") as f:
        shop.report(f)                         # ...to the screen and to a file
    print("\nreport saved to report.txt")


if __name__ == "__main__":
    main()
```

15.6 Running It

```
$ g++ -std=c++17 -O2 -Wall -Wextra cpp/laundry.cpp -o laundry
$ ./laundry
$ python3 python/laundry.py
```

Both programs print the following and write the report part to `report.txt`:

```
======================================
PROFITINA LAUNDRY             No. 1001
Bu Rina Wulandari (member)
Wash & fold, 3.450 kg x Rp 7.000
  Base price                  Rp 24.150
  Member discount 10%         -Rp 2.415
  TOTAL                       Rp 21.735
======================================
PROFITINA LAUNDRY             No. 1002
Pak Budi Santoso
Wash & iron, 5.200 kg x Rp 10.000
  Base price                  Rp 52.000
  Express +50%                Rp 26.000
  TOTAL                       Rp 78.000
======================================
PROFITINA LAUNDRY             No. 1003
Mbak Sari Dewi (member)
Bed cover, 2 pcs x Rp 25.000
  Base price                  Rp 50.000
  Member discount 10%         -Rp 5.000
  TOTAL                       Rp 45.000
======================================
PROFITINA LAUNDRY             No. 1004
Bu Rina Wulandari (member)
Iron only, 2.000 kg x Rp 5.000
  Base price                  Rp 10.000
  Member discount 10%         -Rp 1.000
  TOTAL                        Rp 9.000
======================================
PROFITINA LAUNDRY             No. 1005
Mas Joko Prasetyo
Shoes, 1 pcs x Rp 35.000
  Base price                  Rp 35.000
  Express +50%                Rp 17.500
  TOTAL                       Rp 52.500
======================================
PROFITINA LAUNDRY             No. 1007
Pak Budi Santoso
Wash & fold, 4.100 kg x Rp 7.000
  Base price                  Rp 28.700
  TOTAL                       Rp 28.700
======================================
PROFITINA LAUNDRY             No. 1008
Mbak Sari Dewi (member)
Wash & iron, 2.750 kg x Rp 10.000
  Base price                  Rp 27.500
  Express +50%                Rp 13.750
  Member discount 10%         -Rp 4.125
  TOTAL                       Rp 37.125
======================================
refused: order 1006, unknown customer C05
refused: order 1009, unknown service DC
```

```
DAILY REPORT - 7 receipts, 2 orders refused
Revenue by service (largest first)
  WI  Wash & iron          2  Rp 115.125
  SH  Shoes                1   Rp 52.500
  W   Wash & fold          2   Rp 50.435
  BC  Bed cover            1   Rp 45.000
  I   Iron only            1    Rp 9.000
Revenue by customer (largest first)
  C02 Pak Budi Santoso     2  Rp 106.700
  C03 Mbak Sari Dewi       2   Rp 82.125
  C04 Mas Joko Prasetyo    1   Rp 52.500
  C01 Bu Rina Wulandari    2   Rp 30.735
Member discounts given          Rp 12.540
GRAND TOTAL                    Rp 272.060

report saved to report.txt
```

Check the most complicated receipt by hand, order 1008 for Mbak Sari Dewi, a member, 2,750 g of wash and iron, express. The base price is $(2750 \times 10000 + 500)/1000 = 27{,}500$. The surcharge is 50% of that by Equation (15.1): $(27{,}500 \times 50 + 50)/100 = 13{,}750$. The discount is 10% of $27{,}500 + 13{,}750 = 41{,}250$, that is $(412{,}500 + 50)/100 = 4{,}125$. By Equation (15.5), the total is $27{,}500 + 13{,}750 - 4{,}125 = 37{,}125$. The receipt agrees.

Then check the report. Wash and iron has two receipts, $78{,}000 + 37{,}125 = 115{,}125$, the largest amount, so it comes first. Customer C01 has $21{,}735 + 9{,}000 = 30{,}735$ and comes last among the customers. The four member discounts are $2{,}415 + 5{,}000 + 1{,}000 + 4{,}125 = 12{,}540$. And the two refused orders are listed with their reasons instead of disappearing.

15.7 Testing It

The chapter's test list names the three programs; `laundry` reads its data files from the `data/` folder.

```
# name | C++ sources | Python script | stdin | arguments
rupiah_demo | rupiah_demo.cpp | rupiah_demo.py | - | -
fraction | fraction.cpp | fraction.py | - | -
laundry | laundry.cpp | laundry.py | - | -
```

```
$ python3 tests/run_tests.py --sanitize
```

```
rupiah_demo                C++ ok   Python ok
fraction                   C++ ok   Python ok
laundry                    C++ ok   Python ok
all programs agree
```

A class with operators deserves tests of the *laws* its operators should obey, because readers of `a + b` assume them. For `Rupiah`: $a + b = b + a$; $(a + b) - b = a$; $a + (-a) = 0$; $3 \times a = a + a + a$; and exactly one of $a < b$, $a = b$, $b < a$ holds. For `Fraction`: $x + 0 = x$, $x \times 1 = x$, $x/x = 1$ for $x \neq 0$, and $\frac{a}{b} \times \frac{b}{a} = 1$. Exercise 3 turns these into a small program that tries them on many values. A law that fails points straight at the operator that is wrong.

15.8 Making It Better

- **A better sort key for customers.** The customer report is sorted by money; the owner may prefer names. Because the order lives in one `operator<`, a second order is a second comparison function passed to `std::sort` as its third argument, the way Chapter 16 sorts pick-up times.

- **Refused orders into a separate file.** The helper at the counter needs the list of refused orders to call the customers. `printReceipts` already takes a `std::ostream&`, so writing the refused lines to `refused.txt` needs only a second stream.
- **Weights as their own type.** A `Grams` class, like `Rupiah`, would make `Rupiah + Grams` impossible and give `describe` a natural home as `operator<<` for `Grams`.
- **Several days.** A date column in `orders.txt` and a `std::map` from date to `Shop` would give a report per day and a month's summary. Chapter 16 (Problem 16.4) shows how a matrix product turns daily kilograms into daily revenue in one step.

15.9 Using It for Real

This program is complete enough to be adapted to a real small business, and the adaptation follows the files. A tailor puts "hem trousers;pcs;15000" and "take in a dress;pcs;45000" into `services.txt`; a car wash puts vehicle classes; a catering business prices per portion. The customer file gains a column for the discount percentage if members do not all get the same. Everything else, the checks, the receipts and the report, stays.

Keep three habits from this chapter when you adapt it. Keep money in a type of its own, so that it is always rounded and printed in one way. Keep the rules of the business, such as surcharges and discounts, in one visible place, like Section 15.5.2 and the few lines of `take` that implement it, so that the owner can read them and confirm them. And check the program by hand against one real receipt from the shop before trusting it with a month of accounts. The program helps the owner count; decisions about prices and discounts remain the owner's.

15.10 Common Mistakes, and How to Avoid Them

- **Returning a copy from `+=`.** `Rupiah operator+=(...)` (without `&`) compiles, but `(a += b) += c` then changes a temporary copy, not `a`. Compound assignments return `*this` by reference.
- **Making `operator+` a member.** It then needs your object on the left, so `3 * iron` or `2 + half` fails to compile. Write symmetric operators as non-members built on `+=`.
- **Defining `==` and `!=` (or `<` and `>`) independently.** Sooner or later they disagree. Define `==` and `<`, and write the others in terms of them.
- **A `<` that is not a strict order.** `std::sort` needs a comparison where `a < a` is false and where "neither `a < b` nor `b < a`" means "equivalent". Writing `<=` instead of `<` in `Total`'s `operator<` breaks this rule and is undefined behavior for `std::sort`.
- **Forgetting `explicit` on a converting constructor.** Every one-argument constructor is a silent conversion unless it is `explicit`. Decide for each: is every value of the parameter type really a value of your type?
- **Overflow in `a * b / gcd`.** Divide first, as Equation (15.3) does. The same reasoning applies to percentages of very large amounts.
- **Surprising meanings.** An operator that does something a reader would not guess from its symbol makes the code a puzzle. Use a named function.
- **Python: returning `NotImplemented` versus raising.** When `__add__` cannot handle the other operand, a careful Python class returns the special value `NotImplemented` so that Python can try the reflected method of the other operand; our small classes assume correct operands, which is fine for a program that controls all its own calls.

15.11 Chapter Summary

- An overloaded operator is a function named `operator` followed by the symbol; `a + b` on class objects becomes `operator+(a, b)` or `a.operator+(b)`. Precedence, arity and the built-in meanings cannot change, and a few operators cannot be overloaded at all.
- Operators that change their left operand (`+=`, `-=`) are members returning `*this` by reference; symmetric operators (`+`, `-`, `==`, `<`) are non-members built on them; `<<` for output is a non-member that returns the stream.
- The `Rupiah` class keeps money in whole rupiah, rounds percentages in one documented way (Equation (15.1)), prints amounts in the Indonesian style, and refuses nonsense such as `Rupiah * Rupiah` and silent conversions from numbers (an `explicit` constructor).
- The `Fraction` class keeps its values in lowest terms with Euclid's algorithm (Equation (15.2)) and adds with the lcm (Equation (15.3), Equation (15.4)); its non-explicit constructor makes `2 + half` work.
- The complete Profitina Laundry program reads its price list, customers and orders from files, checks every order, prints receipts through `operator<<`, sorts its report with `operator<`, uses `std::map` for look-ups, and writes one report function to both the screen and a file.
- Python offers the same through special methods (`__add__`, `__rmul__`, `__lt__`, `__str__`, `__format__`) and `functools.total_ordering`; both versions of every program print identical output.

15.12 Exercises

1. Add `operator/` to `Rupiah` for dividing an amount into k equal parts, rounded half up, and a member function `split(k)` that returns a `std::vector<Rupiah>` of k parts which *add up exactly* to the original amount. Split Rp 100,000 into 3 parts.
2. Write a factorial function in two versions, iterative and recursive, both returning `long long`, and print $n!$ for $n = 0,5,10,15,20,21$. What happens at 21, and how can the program detect it before it happens?
3. Write a program that checks the laws of Section 15.7 for `Rupiah` on all pairs of the amounts $-1000, -1,0,1,999,24150$ and for `Fraction` on all pairs of $\frac{1}{2}, -\frac{2}{3}, \frac{5}{4}, 3, 0$ (skipping division by zero), and prints how many checks passed.
4. The sum of squares $1^2 + 2^2 + \cdots + n^2$ can be computed with a loop or with the formula $\frac{n(n+1)(2n+1)}{6}$. Write both, compare them for $n = 1, \ldots, 1000$, and explain why the formula must be computed as `n * (n + 1) / 2 * (2 * n + 1) / 3` or with another careful order to avoid both overflow and wrong rounding.
5. Change `laundry.cpp` so that a member's discount is read from the customer file (a fourth column with a percentage, for example 10 or 15), and non-members have 0. What changes in the output if Mbak Sari Dewi's discount is 15%?
6. Add `operator<<` for `Fraction` that prints mixed numbers when the fraction is larger than 1 in absolute value: $\frac{7}{2}$ as `3 1/2` and $-\frac{7}{2}$ as `-3 1/2`. Why is this a questionable change to the standard output operator, and what would you do instead?

15.13 Solutions to the Exercises

1. `Rupiah operator/(const Rupiah& a, long long k)` returns `Rupiah((a.value() * 2 + k) / (2 * k))` for non-negative amounts, which rounds half up (it is Equation (15.1) with the divisor k instead of 100). Rounded parts do not always add up: 100,000/3 gives 33,333 three times, which is 99,999. `split(k)` fixes this: compute the whole part $q = \lfloor a/k \rfloor = 33{,}333$ and the remainder $r = a \bmod k = 1$, and give $q + 1$ to the first r parts and q to the rest: $33{,}334 + 33{,}333 + 33{,}333 = 100{,}000$. This is how a bill is split fairly between friends without losing a rupiah.

2. The iterative version multiplies `f *= i` for $i = 2, \ldots, n$; the recursive one returns `n <= 1 ? 1 : n * factorial(n - 1)`. They agree up to $20! = 2{,}432{,}902{,}008{,}176{,}640{,}000$, the largest factorial that fits in a `long long`. 21! is about 5.1×10^{19}, larger than the maximum of about 9.2×10^{18}: the multiplication overflows, which for signed integers is undefined behavior, and in practice a wrong, often negative, number is printed (the sanitizer of Section 13.11 stops the program instead). To detect it in advance, check before multiplying: if `f > LLONG_MAX / i`, the product would not fit. In Python, integers grow without limit, so `math.factorial(21)` is simply correct, which is a real difference between the two languages worth remembering.

3. Loop over all pairs with two nested range-based loops, and for each pair test each law with `==`, counting successes and printing any failure with its values. With the six amounts there are 36 pairs; with the four pair-laws of `Rupiah` that are about pairs ($a + b = b + a$, $(a + b) - b = a$, the trichotomy, and $3 \times a = a + a + a$ applied to the first element), every check passes. For fractions, $x/x = 1$ is skipped for $x = 0$, and all other checks pass. The value of the program is what happens when you later change an operator: a wrong change shows up at once as a failed law.

4. The loop adds `1LL * i * i`; the formula gives the same value, for example 385 for $n = 10$ and 333,833,500 for $n = 1000$. The product $n(n + 1)(2n + 1)$ is always divisible by 6, but a program that divides a *part* of it too early may round: $n/6$ alone is wrong whenever 6 does not divide n. The safe way is to divide by a factor that is guaranteed to divide what you have: $n(n + 1)$ is always even, so `n * (n + 1) / 2` is exact; then multiplying by $2n + 1$ gives a number divisible by 3, because $n(n + 1)(2n + 1)$ is divisible by 3; so `/ 3` is exact as well. Dividing early also keeps the intermediate value smaller, which postpones overflow.

5. Add a fourth field to each line of `customers.txt`, store it as `int discount` in `Customer`, check it in `load` with `isNumber` and a limit of 0 to 100, and in `take` write `r.discount = (r.base + r.express).percent(r.customer->discount);`, removing the `member` test (a discount of 0 gives Rp 0). With 15% for Mbak Sari Dewi, order 1003 gets a discount of $(50{,}000 \times 15 + 50)/100 = 7{,}500$ and a total of 42,500; order 1008 gets $(41{,}250 \times 15 + 50)/100 = 6{,}188$ (the exact value 6,187.5 rounds up) and a total of 35,062. The discount line should then say “Member discount 15%”, so `line40` receives a label built from the percentage.

6. Compute the whole part `w = num / den` and the remainder `r = num % den` (C++ division truncates toward zero, so for $-\frac{7}{2}$, $w = -3$ and $r = -1$); print `w`, a space, `std::abs(r)`, `/` and `den` when both are non-zero. It is questionable because `<<` is expected to print a value in a form that can be read back and that everyone understands; mixed numbers are a *presentation* choice. Better: keep `<<` as it is and add a named function `toMixed(const Fraction&)` that returns the mixed form as a string, used only where it helps a reader.

References

[1] B. Stroustrup, *The C++ Programming Language*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.

[2] cppreference.com, "operator overloading," accessed Sep. 2026. [Online]. Available: https://en.cppreference.com/w/cpp/language/operators

[3] B. Stroustrup and H. Sutter, Eds., "C++ Core Guidelines, Section C.over: Overloading and overloaded operators," accessed Sep. 2026. [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[4] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2005.

[5] Python Software Foundation, "The Python Language Reference, Section 3.3: Special method names," Python 3.14 documentation, accessed Sep. 2026. [Online]. Available: https://docs.python.org/3.14/reference/datamodel.html

[6] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[7] Python Software Foundation, "fractions — Rational numbers," Python 3.14 documentation, accessed Sep. 2026. [Online]. Available: https://docs.python.org/3.14/library/fractions.html

Chapter 16

Evaluation and Discussion IV: Classes, Overloading and the Whole Book, Solved Step by Step

This last chapter adds no new language feature. It takes the C++ of Chapters 13 to 15, and with it everything the book has built since Chapter 1, and puts it to work on five complete problems. Four are the classic closing problems of a first programming course: a family of chess pieces that must be completed by imitating one given class; a function `add` overloaded for two to five arguments; finding every anagram of a word inside a longer text; and multiplying two matrices after checking that they fit together. The fifth is new and belongs to Profitina Laundry: a `Time` class with operators that tells each customer when the laundry can be picked up. Every problem is solved in the same order as in Chapters 4, 8 and 12: the problem restated in our own words; examples worked by hand; a design on paper; a complete C++ program and its Python twin, with the real output of both; a *tempting wrong solution* and the exact way it fails; and a *check yourself* variant with its answer. By the end of the chapter you will have a tested method for any small problem that involves classes, overloading, strings and arrays, five fully worked examples of it, and a clear view of how far you have come since your first program printed one line.

16.1 Where We Left Off

Here is the toolkit these problems draw on, chapter by chapter.

From **Chapters 1 to 3** you have the basics that never go away: variables and types, integer division and its rounding, `if`, `switch`, the three loops, and the habit of computing a result by hand before trusting a program. From **Chapters 5 to 7** you have functions with parameters and return values, recursion, arrays and two-dimensional arrays, sorting and searching, and pointers, the addresses behind every array. From **Chapters 9 to 11** you have strings, structures and file processing.

From **Chapter 13** you have C++ itself [1]: `cin` and `cout`, `std::string`, `std::vector`, references, overloading and default arguments. From **Chapter 14** you have classes: private data guarded by an invariant, constructors that refuse invalid values, `const` member functions, and a first class hierarchy with `protected` members, derived classes and virtual functions. From **Chapter 15** you have operator overloading: arithmetic and comparison operators, `operator<<` for printing, and the rule that each operator should mean what a reader expects.

Problem 16.1 needs inheritance (Chapter 14); Problem 16.2 needs overloading (Chapter 13); Problem 16.3 needs strings (Chapters 9 and 13) and a loop whose bounds must be exactly right (Chapter 3); Problem 16.4 needs two-dimensional arrays (Chapter 6), now as vectors of vectors; and Problem 16.5 needs a class with an invariant and operators (Chapters 14 and 15). Naming the tool is already half of the solution.

16.2 How to Approach a Problem

16.2.1 The seven steps

Every problem in this chapter goes through the same seven steps. They are an old idea, the problem-solving method of Pólya [2] adapted to programs, and worth learning as a routine, because a routine is what carries you through a problem when you are tired or unsure.

1. **Restate the problem in your own words.** Write down what goes in, what comes out, and every decision the statement leaves open. A decision you make silently is a decision a reader cannot check.
2. **Work examples by hand.** Take the example of the statement, if there is one, and at least one of your own, including the edge cases: empty input, the smallest and largest values, input that must be refused.
3. **Design on paper.** Choose the data (a class, a vector, a string) and write the algorithm as pseudocode or as a short list of steps. Estimate its cost.
4. **Write the program.** Complete, compiling without warnings, with the design visible in the code.
5. **Run it on the hand examples.** The output must agree with step 2, character for character where the format is given.
6. **Try to break it.** Look for the input that a hurried programmer would get wrong. The "tempting wrong solution" of each problem is exactly such a program.
7. **Change one thing and predict the result.** The "check yourself" box asks you to change the problem slightly and to predict the new output *before* running anything. If your prediction is right, you understand the solution; if not, you have found what to study.

16.2.2 How each problem is laid out, and where the code lives

Each problem has six parts: the problem, understanding it by hand, designing it on paper, the C++ solution, the Python solution, and a tempting wrong solution with a "check yourself" box. The code is in `FP/Code_new/Chapter16`: the solutions in `cpp/` and `python/`, the tempting wrong solutions in `cpp/tempting/` and `python/tempting/`, input files in `input/`, and the expected output of every program in `expected/`. The test program of Section 13.11 runs every solution in both languages and compares the outputs; its list of cases also checks that the wrong solution of Problem 16.2 is *refused* by the compiler.

16.3 Problem 16.1: Chess Pieces

16.3.1 The problem

> **PROBLEM 16.1.** A program represents chess pieces as classes. A base class `Piece` stores the piece's name, its side (white or black) and its position on the board, and it offers a function that prints the position. A complete class `Pawn`, derived from `Piece`, is given: its constructor sets the name and the side, and its move function checks the pawn's rule, moves the piece and prints the new position, or prints `Invalid movement!!`. Add the classes `Bishop`, `Knight`, `Rook` and `Queen` in the same way, each starting at (0, 0), and test each of them with a legal move, an illegal move, and a legal move from the new position.

We make four decisions. *The board:* coordinates (x, y) with $0 \le x, y \le 7$, x to the right and y upward, as on a sheet of graph paper; a move to a square outside the board is invalid. *The side:* a member `faction_` is $+1$ for white and -1 for black; a white pawn moves up (its y grows by 1)

and a black pawn moves down. *Alone on the board:* each piece is tested alone, so there are no other pieces to block a path or to capture; a real chess program would need a board object that knows every piece. *Standing still is not a move:* moving to the square you are on is invalid for every piece.

16.3.2 Understanding it by hand

Write $d_x = x' - x$ and $d_y = y' - y$ for the change from the current square (x, y) to the destination (x', y'), and $|d_x|$, $|d_y|$ for their sizes. The five rules of chess movement, for a piece alone on the board, become five conditions:

- **Pawn** (white): same column and one step up: $d_x = 0$ and $d_y = 1$; for black, $d_y = -1$. Both are covered by $d_y = 1 \cdot \text{faction}$.
- **Bishop:** diagonally, any distance: $|d_x| = |d_y|$ and $d_x \neq 0$.
- **Knight:** an "L", two squares one way and one the other: $\{|d_x|, |d_y|\} = \{1,2\}$.
- **Rook:** along a row or a column, any distance: *exactly one* of x and y changes.
- **Queen:** like a rook or like a bishop.

The rook's "exactly one of two things" is the logical operation *exclusive or* (XOR), written $\oplus$: true when exactly one of its two operands is true. The rook's rule is therefore

$$\text{rook}(x, y \to x', y') = (x' \neq x) \oplus (y' \neq y) \tag{16.1}$$

In C++ and in Python, `^` is the bitwise XOR operator; applied to two `bool` values, it is exactly $\oplus$. **Figure 16.1** shows, for each piece standing on (3,3), the squares it may move to according to these conditions; check a few squares against the formulas with a pencil.

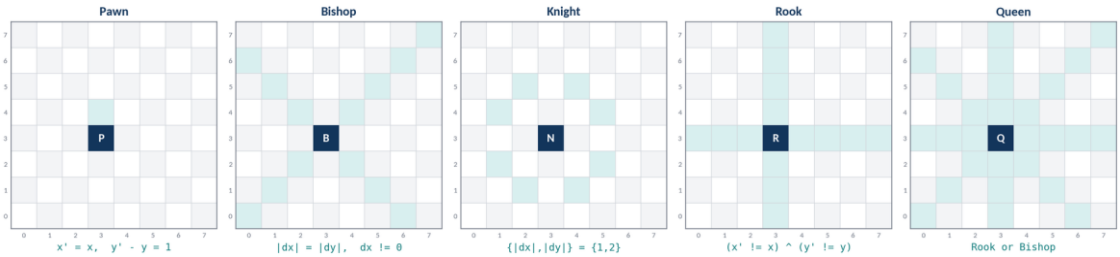


Figure 16.1 The squares each piece may reach from (3, 3) on an empty board, with the condition that decides them.

Now the tests, worked by hand. The bishop starts at (0,0). To (3,3): $|d_x| = |d_y| = 3$, $d_x \neq 0$, valid. From (3,3) to (3,5): $|d_x| = 0 \neq 2 = |d_y|$, invalid, and the bishop stays at (3,3). To (5,1): $|d_x| = |d_y| = 2$, valid. The rook from (0,0) to (0,6): only y changes, valid; from (0,6) to (3,3): both change, invalid; to (7,6): only x changes, valid; to (8,6): off the board, invalid.

16.3.3 Designing it on paper

The given `Pawn` shows the pattern: a constructor that passes the name, side and start square to the `Piece` constructor, and a function with the pawn's condition. Every new class repeats the pattern with its own condition, so there is almost nothing to invent. One improvement makes the imitation even simpler: we put the steps that are the same for every piece (check the board, check the condition, move and print, or complain) into *one* function `move` in `Piece`, and let it call a pure virtual function `moveCondition` that each piece overrides. This is the polymorphism of Section 14.6: the base class decides *when* to ask, each derived class decides *what* to answer.

Pseudocode 16.1 The move shared by all pieces; MOVE-CONDITION is different for each kind of piece.

```
PIECE-MOVE(piece, x', y')
1  if 0 <= x' <= 7 and 0 <= y' <= 7 and MOVE-CONDITION(piece, x', y')
2      piece.x = x'
3      piece.y = y'
4      print "moved, " and the new location
5  else
6      print "Invalid movement!!"
```

Pseudocode 16.1 costs a constant amount of work per move. The whole difficulty of the problem lies in writing the five conditions *exactly*, including the parentheses.

16.3.4 The C++ solution

Listing 16.1 is the complete program. The part marked “given” is the model; the four classes after it are our answer.

Listing 16.1 `chess.cpp`: the given `Piece` and `Pawn`, and the four added pieces.

```cpp
// chess.cpp - Problem 16.1: chess pieces as a small class hierarchy.
// Piece and Pawn are the given part; Bishop, Knight, Rook and Queen are added
// "like the Pawn class". One piece alone on an 8 x 8 board, no captures.
#include <cstdlib>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

class Piece {
public:
    Piece(const std::string& name, int faction, int x, int y)
        : name_(name), faction_(faction), x_(x), y_(y) {}
    virtual ~Piece() = default;

    std::string getLocation() const {
        return name_ + (faction_ == 1 ? " (white)" : " (black)") + " at ("
               + std::to_string(x_) + ", " + std::to_string(y_) + ")";
    }

    void move(int xDest, int yDest) {          // the same steps for every piece
        std::cout << "  to (" << xDest << ", " << yDest << "): ";
        if (onBoard(xDest, yDest) && moveCondition(xDest, yDest)) {
            x_ = xDest;
            y_ = yDest;
            std::cout << "moved, " << getLocation() << '\n';
        } else {
            std::cout << "Invalid movement!!\n";
        }
    }

protected:
    virtual bool moveCondition(int xDest, int yDest) const = 0;
    std::string name_;
    int faction_;                              // +1 white (moves up), -1 black
    int x_, y_;

private:
    static bool onBoard(int x, int y) { return x >= 0 && x < 8 && y >= 0 && y < 8; }
};

// ---- given: the model to imitate ------------------------------------------
class Pawn : public Piece {
public:
    Pawn(int faction, int x = 0, int y = 0) : Piece("Pawn", faction, x, y) {}
```

Listing 16.1 (Continued) *chess.cpp: the given Piece and Pawn, and the four added pieces.*

```
protected:
    bool moveCondition(int xDest, int yDest) const override {
        return (xDest == x_) && ((yDest - y_) == 1 * faction_);
    }
};

// ---- added: four more pieces, built exactly like Pawn ----------------------
class Bishop : public Piece {
public:
    Bishop(int faction, int x = 0, int y = 0) : Piece("Bishop", faction, x, y) {}

protected:
    bool moveCondition(int xDest, int yDest) const override {
        return (xDest != x_) && (std::abs(xDest - x_) == std::abs(yDest - y_));
    }
};

class Knight : public Piece {
public:
    Knight(int faction, int x = 0, int y = 0) : Piece("Knight", faction, x, y) {}

protected:
    bool moveCondition(int xDest, int yDest) const override {
        int dx = std::abs(xDest - x_), dy = std::abs(yDest - y_);
        return (dx == 1 && dy == 2) || (dx == 2 && dy == 1);
    }
};

class Rook : public Piece {
public:
    Rook(int faction, int x = 0, int y = 0) : Piece("Rook", faction, x, y) {}

protected:
    bool moveCondition(int xDest, int yDest) const override {
        return (xDest != x_) ^ (yDest != y_);  // exactly one coordinate changes
    }
};

class Queen : public Piece {
public:
    Queen(int faction, int x = 0, int y = 0) : Piece("Queen", faction, x, y) {}
```

Listing 16.1 (Continued) *chess.cpp: the given Piece and Pawn, and the four added pieces.*

```
protected:
    bool moveCondition(int xDest, int yDest) const override {
        bool straight = (xDest != x_) ^ (yDest != y_);
        bool diagonal =
            (xDest != x_) && (std::abs(xDest - x_) == std::abs(yDest - y_));
        return straight || diagonal;
    }
};

int main() {
    Pawn whitePawn(1), blackPawn(-1, 3, 6);
    Bishop bishop(1);
    Knight knight(1);
    Rook rook(1);
    Queen queen(1);

    using Moves = std::vector<std::pair<int, int>>;
    std::vector<std::pair<Piece*, Moves>> tests = {
        {&whitePawn, {{0, 1}, {1, 2}, {0, 2}}},
        {&blackPawn, {{3, 5}, {3, 6}, {3, 4}}},
        {&bishop, {{3, 3}, {3, 5}, {5, 1}}},
        {&knight, {{1, 2}, {2, 2}, {3, 1}}},
        {&rook, {{0, 6}, {3, 3}, {7, 6}, {8, 6}}},
        {&queen, {{4, 4}, {5, 6}, {4, 0}, {4, 0}}},
    };
    for (auto& [piece, moves] : tests) {       // C++17 structured binding
        std::cout << piece->getLocation() << '\n';
        for (auto& [x, y] : moves) piece->move(x, y);
    }
    return 0;
}
```

```
Pawn (white) at (0, 0)
  to (0, 1): moved, Pawn (white) at (0, 1)
  to (1, 2): Invalid movement!!
  to (0, 2): moved, Pawn (white) at (0, 2)
Pawn (black) at (3, 6)
  to (3, 5): moved, Pawn (black) at (3, 5)
  to (3, 6): Invalid movement!!
  to (3, 4): moved, Pawn (black) at (3, 4)
Bishop (white) at (0, 0)
  to (3, 3): moved, Bishop (white) at (3, 3)
  to (3, 5): Invalid movement!!
  to (5, 1): moved, Bishop (white) at (5, 1)
Knight (white) at (0, 0)
  to (1, 2): moved, Knight (white) at (1, 2)
  to (2, 2): Invalid movement!!
  to (3, 1): moved, Knight (white) at (3, 1)
Rook (white) at (0, 0)
  to (0, 6): moved, Rook (white) at (0, 6)
  to (3, 3): Invalid movement!!
  to (7, 6): moved, Rook (white) at (7, 6)
  to (8, 6): Invalid movement!!
Queen (white) at (0, 0)
  to (4, 4): moved, Queen (white) at (4, 4)
  to (5, 6): Invalid movement!!
  to (4, 0): moved, Queen (white) at (4, 0)
  to (4, 0): Invalid movement!!
```

Every line agrees with the hand work of Section 16.3.2. Look at the details that make the imitation correct. Each class has its own constructor, because constructors are not inherited, and each passes its own name. Each condition is written with `std::abs` from `<cstdlib>`, as the problem's own conditions are. The `Queen` reuses the two formulas of the rook and the bishop under names,

`straight` and `diagonal`, which makes the code read like the rule. The test table in `main` is a vector of pairs, each pair holding a pointer to a piece and its list of moves; the loop uses C++17 *structured bindings*, `auto& [piece, moves]`, to give the two halves of a pair their own names.

16.3.5 The Python solution

Listing 16.2 is the same hierarchy in Python, with an abstract `_move_condition`.

Listing 16.2 `chess.py`: the chess pieces in Python; the output is identical.

```
# chess.py - Problem 16.1: chess pieces as a small class hierarchy.
# Piece and Pawn are the given part; Bishop, Knight, Rook and Queen are added
# "like the Pawn class". One piece alone on an 8 x 8 board, no captures.
from abc import ABC, abstractmethod


class Piece(ABC):
    def __init__(self, name, faction, x, y):
        self._name, self._faction = name, faction   # +1 white, -1 black
        self._x, self._y = x, y

    def get_location(self):
        side = " (white)" if self._faction == 1 else " (black)"
        return f"{self._name}{side} at ({self._x}, {self._y})"

    def move(self, x_dest, y_dest):            # the same steps for every piece
        print(f"  to ({x_dest}, {y_dest}): ", end="")
        on_board = 0 <= x_dest < 8 and 0 <= y_dest < 8
        if on_board and self._move_condition(x_dest, y_dest):
            self._x, self._y = x_dest, y_dest
            print(f"moved, {self.get_location()}")
        else:
            print("Invalid movement!!")

    @abstractmethod
    def _move_condition(self, x_dest, y_dest): ...


class Pawn(Piece):                             # given: the model to imitate
    def __init__(self, faction, x=0, y=0):
        super().__init__("Pawn", faction, x, y)

    def _move_condition(self, x_dest, y_dest):
        return x_dest == self._x and (y_dest - self._y) == 1 * self._faction


class Bishop(Piece):
    def __init__(self, faction, x=0, y=0):
        super().__init__("Bishop", faction, x, y)

    def _move_condition(self, x_dest, y_dest):
        return x_dest != self._x and abs(x_dest - self._x) == abs(y_dest - self._y)
```

Listing 16.2 (Continued) *chess.py: the chess pieces in Python; the output is identical.*

```
class Knight(Piece):
    def __init__(self, faction, x=0, y=0):
        super().__init__("Knight", faction, x, y)

    def _move_condition(self, x_dest, y_dest):
        dx, dy = abs(x_dest - self._x), abs(y_dest - self._y)
        return (dx == 1 and dy == 2) or (dx == 2 and dy == 1)


class Rook(Piece):
    def __init__(self, faction, x=0, y=0):
        super().__init__("Rook", faction, x, y)

    def _move_condition(self, x_dest, y_dest):
        return (x_dest != self._x) ^ (y_dest != self._y)   # exactly one changes


class Queen(Piece):
    def __init__(self, faction, x=0, y=0):
        super().__init__("Queen", faction, x, y)

    def _move_condition(self, x_dest, y_dest):
        straight = (x_dest != self._x) ^ (y_dest != self._y)
        diagonal = (x_dest != self._x
                    and abs(x_dest - self._x) == abs(y_dest - self._y))
        return straight or diagonal


tests = [
    (Pawn(1), [(0, 1), (1, 2), (0, 2)]),
    (Pawn(-1, 3, 6), [(3, 5), (3, 6), (3, 4)]),
    (Bishop(1), [(3, 3), (3, 5), (5, 1)]),
    (Knight(1), [(1, 2), (2, 2), (3, 1)]),
    (Rook(1), [(0, 6), (3, 3), (7, 6), (8, 6)]),
    (Queen(1), [(4, 4), (5, 6), (4, 0), (4, 0)]),
]
for piece, moves in tests:
    print(piece.get_location())
    for x, y in moves:
        piece.move(x, y)
```

Python's `^` on two `bool`s is also exclusive or, and its chained comparison `0 <= x_dest < 8` reads exactly like mathematics.

16.3.6 A tempting wrong solution

The rook's rule sounds like "the rook moves if it moves along a row or a column", and the tempting translation of "or" is `||`. **Listing 16.3** writes it that way, together with a bishop that forgets $d_x \neq 0$.

Listing 16.3 `wrong_rook.cpp`: the rook's condition with `||` and the bishop's without $d_x \neq 0$.

```cpp
// wrong_rook.cpp - a tempting wrong solution to Problem 16.1: the Rook's
// condition written with || instead of ^ (and the bishop's without x != x0).
#include <cstdlib>
#include <iostream>

bool rookWrong(int x, int y, int xDest, int yDest) {
    return (xDest != x) || (yDest != y);       // "something changed"
}

bool bishopWrong(int x, int y, int xDest, int yDest) {
    return std::abs(xDest - x) == std::abs(yDest - y);   // allows standing still
}

int main() {
    std::cout << "rook   (0,6) -> (3,3): " << rookWrong(0, 6, 3, 3) << '\n';
    std::cout << "rook   (0,6) -> (5,1): " << rookWrong(0, 6, 5, 1) << '\n';
    std::cout << "bishop (3,3) -> (3,3): " << bishopWrong(3, 3, 3, 3) << '\n';
    return 0;
}
```

```
rook   (0,6) -> (3,3): 1
rook   (0,6) -> (5,1): 1
bishop (3,3) -> (3,3): 1
```

A stream prints `true` as `1`. The wrong rook accepts (0,6) → (3,3) and (0,6) → (5,1), two diagonal moves, because `||` is true when *at least one* coordinate changes, and on a diagonal both do. `||` would be right only if it meant "one or the other but not both", which is what $\oplus$ in Equation (16.1) says. The wrong bishop accepts "moving" from (3,3) to (3,3), because $0 = 0$. The two mistakes have the same root: the rule was translated from a loose sentence instead of from a precise condition checked on examples, including the example "stay where you are".

> **CHECK YOURSELF.** A *king* moves one square in any direction. Write its `moveCondition` with $|d_x|$ and $|d_y|$, and predict the output for a white king at (0,0) moved to (1,1), then (1,3), then (1,1).
>
> *Answer.* `return std::max(std::abs(xDest - x_), std::abs(yDest - y_)) == 1;`: the larger of the two distances must be exactly 1, which also rules out standing still. (0,0) → (1,1): both distances 1, moved. (1,1) → (1,3): $|d_y| = 2$, `Invalid movement!!`, and the king stays at (1,1). (1,1) → (1,1): the larger distance is 0, `Invalid movement!!`.

16.4 Problem 16.2: Overloading `add`

16.4.1 The problem

> **PROBLEM 16.2.** A function `int add(int a, int b)` that returns $a + b$ is given. Write overloads of `add` for three, four and five integer arguments, and print the result of one call of each.

Our decisions: the arguments are the weights of laundry bags in grams, from Profitina Laundry's scale; the program prints each call and its result in aligned columns; and we add, as extras, an overload for two `double`s and a C++17 template that accepts any number of arguments, to show where overloading ends.

16.4.2 Understanding it by hand

The bags weigh 3,450, 1,200, 5,000, 2,750 and 4,100 grams. The sums of the first two, three, four and five bags are 4,650, 9,650, 12,400 and 16,500 grams. For the `double` overload, $2.5 + 1.25 = 3.75$.

16.4.3 Designing it on paper

Overload resolution chooses by the *number* of arguments (Section 13.7), so the four functions can coexist. The elegant design builds each overload on the one below it:

$$\text{add}(a_1, \ldots, a_k) = \text{add}(\text{add}(a_1, \ldots, a_{k-1}),\ a_k) \tag{16.2}$$

This is the same idea as a recursive definition (Chapter 5), unrolled by hand for $k = 3,4,5$: each version is one line, and a fix in the two-argument version automatically reaches all the others.

16.4.4 The C++ solution

Listing 16.4 `add_overload.cpp`: `add` for two to five integers, for two `double`s, and a variadic template.

```
// add_overload.cpp - Problem 16.2: add() overloaded for 2, 3, 4 and 5 arguments.
#include <iostream>

int add(int a, int b) { return a + b; }        // given
int add(int a, int b, int c) { return add(add(a, b), c); }
int add(int a, int b, int c, int d) { return add(add(a, b, c), d); }
int add(int a, int b, int c, int d, int e) { return add(add(a, b, c, d), e); }

double add(double a, double b) { return a + b; }   // same count, other types

// Going further (C++17): one template for any number of arguments.
template <typename... Ts>
long long addAll(Ts... xs) {
    return (0LL + ... + xs);                   // a fold expression
}

int main() {
    std::cout << "add(3450, 1200)                  = " << add(3450, 1200) << '\n';
    std::cout << "add(3450, 1200, 5000)            = " << add(3450, 1200, 5000)
              << '\n';
    std::cout << "add(3450, 1200, 5000, 2750)      = "
              << add(3450, 1200, 5000, 2750) << '\n';
    std::cout << "add(3450, 1200, 5000, 2750, 4100) = "
              << add(3450, 1200, 5000, 2750, 4100) << '\n';
    std::cout << "add(2.5, 1.25)                   = " << add(2.5, 1.25) << '\n';
    std::cout << "addAll(1, 2, 3, 4, 5, 6, 7)      = "
              << addAll(1, 2, 3, 4, 5, 6, 7) << '\n';
    // add(2, 1.5);  would not compile: int->double and double->int tie
    return 0;
}
```

```
add(3450, 1200)                  = 4650
add(3450, 1200, 5000)            = 9650
add(3450, 1200, 5000, 2750)      = 12400
add(3450, 1200, 5000, 2750, 4100) = 16500
add(2.5, 1.25)                   = 3.75
addAll(1, 2, 3, 4, 5, 6, 7)      = 28
```

Listing 16.4 follows Equation (16.2) literally. The `double` overload has the same number of parameters as the given `add`, but different types, so `add(2.5, 1.25)` chooses it.

The template at the end answers the obvious question: why write four functions when the only difference is the count? Because overloading is resolved at compile time, one signature at a time, and a C function cannot take "any number of" typed arguments safely. C++11 added *variadic templates* for exactly this: `template <typename... Ts>` accepts any number of arguments of any types, and C++17's *fold expression* `(0LL + ... + xs)` expands to `0LL + x1 + x2 + ... + xn` at

compile time. Templates are beyond this book, but it is good to know that the four overloads are the "by hand" version of a general tool.

The commented line at the end of `main` is a trap worth understanding: `add(2, 1.5)` does not compile. The first argument fits `add(int, int)` exactly and `add(double, double)` only after a conversion; the second argument fits `add(double, double)` exactly and `add(int, int)` only after a conversion. Neither overload is better for *both* arguments, so the call is ambiguous: the rules of overload resolution require one candidate to be at least as good for every argument and better for one [3].

16.4.5 The Python solution

Python has no overloading; a second `def add` would replace the first. It has something simpler: a parameter written `*numbers` collects any number of arguments into a tuple, so one function covers every count (**Listing 16.5**).

Listing 16.5 `add_overload.py`: one Python function for any number of arguments.

```python
# add_overload.py - Problem 16.2 in Python: no overloading, so one function
# with *numbers accepts any count (a second 'def add' would replace the first).


def add(*numbers):
    total = 0
    for x in numbers:
        total += x
    return total


print(f"add(3450, 1200)                   = {add(3450, 1200)}")
print(f"add(3450, 1200, 5000)             = {add(3450, 1200, 5000)}")
print(f"add(3450, 1200, 5000, 2750)       = {add(3450, 1200, 5000, 2750)}")
print(f"add(3450, 1200, 5000, 2750, 4100) = {add(3450, 1200, 5000, 2750, 4100)}")
print(f"add(2.5, 1.25)                    = {add(2.5, 1.25)}")
print(f"addAll(1, 2, 3, 4, 5, 6, 7)       = {add(1, 2, 3, 4, 5, 6, 7)}")
```

The output is identical, line for line; the last label keeps the C++ name `addAll` so that the two outputs can be compared automatically.

16.4.6 A tempting wrong solution

Default arguments look like a shortcut: one function `add(int a, int b, int c = 0, int d = 0, int e = 0)` covers two to five arguments at once. Used *instead of* the given function, that is actually a neat solution. The tempting mistake is to *add* it next to the given `add(int, int)`, which the problem says must stay (**Listing 16.6**).

Listing 16.6 `wrong_add.cpp`: a defaulted `add` next to the given one; the call with two arguments is ambiguous.

```
// wrong_add.cpp - a tempting wrong solution to Problem 16.2: default arguments
// ON TOP of the given two-argument add(). This file does not compile.
#include <iostream>

int add(int a, int b) { return a + b; }        // given
int add(int a, int b, int c = 0, int d = 0, int e = 0) {   // defaults added
    return a + b + c + d + e;
}

int main() {
    std::cout << add(1, 2, 3) << '\n';         // fine: only the second fits
    std::cout << add(1, 2) << '\n';            // both fit equally well
    return 0;
}
```

`g++` refuses the program:

```
tempting/wrong_add.cpp: In function 'int main()':
tempting/wrong_add.cpp:12:21: error: call of overloaded 'add(int, int)' is ambiguous
   12 |     std::cout << add(1, 2) << '\n';            // both fit equally well
      |                  ~~~^~~~~
tempting/wrong_add.cpp:5:5: note: candidate: 'int add(int, int)'
    5 | int add(int a, int b) { return a + b; }        // given
      |     ^~~
tempting/wrong_add.cpp:6:5: note: candidate: 'int add(int, int, int, int, int)'
    6 | int add(int a, int b, int c = 0, int d = 0, int e = 0) {   // defaults added
```

For `add(1, 2)`, both functions fit perfectly: the given one with its two parameters, the new one with three defaults filled in. Overload resolution ranks candidates only by how well the *arguments* match the parameters, and here both match exactly, so there is no winner. The message names both candidates, which is how you recognize this mistake: whenever `g++` says "ambiguous", look for two overloads that fit the same call equally well.

> **CHECK YOURSELF.** Keep **Listing 16.4** and add `long long add(long long a, long long b)`. Which function does `add(3450, 1200)` call now, and which does `add(3450LL, 1200)` call?
>
> *Answer.* `add(3450, 1200)` still calls `add(int, int)`: both arguments are `int`, an exact match, better than any conversion. `add(3450LL, 1200)` has a `long long` and an `int`: the new overload matches the first exactly and the second after a promotion; `add(int, int)` would need a narrowing conversion for the first; `add(double, double)` a conversion for both. The `long long` overload is at least as good for each argument and better for one, so it wins and returns 4650 as a `long long`.

16.5 Problem 16.3: Finding Anagrams

16.5.1 The problem

> **PROBLEM 16.3.** Given a pattern and a text, find every position in the text where a substring that is an anagram of the pattern begins, and print each such substring with its index. Use `std::is_permutation`. For example, for the pattern `aaba` and the text `aabaabaa` the anagrams start at the indices 0, 1, 3 and 4.

An *anagram* of a word uses the same letters the same number of times, in any order: *stressed* and *desserts*, or *listen* and *silent*. Our decisions: the input consists of pairs `pattern text`, words without spaces, one pair per line, read until the end of the input; the output shows each hit and the number of hits; letters are compared exactly (capitals are different letters); and a pattern that is empty or longer than the text has no anagrams.

16.5.2 Understanding it by hand

Let n be the length of the text and m the length of the pattern. An anagram has the same length as the pattern, so we look at every *window* of m consecutive characters. The first window starts at index 0 and the last one at index $n - m$ (it then ends at index $n - 1$, the last character). So there are

$$W = n - m + 1 \tag{16.3}$$

windows, provided $m \leq n$. **Figure 16.2** shows the five windows for $n = 8$ and $m = 4$. The pattern `aaba` has three `a`s and one `b`, and so do the windows at 0, 1, 3 and 4; the window at 2, `baab`, has two `b`s. The answer of the statement is confirmed.

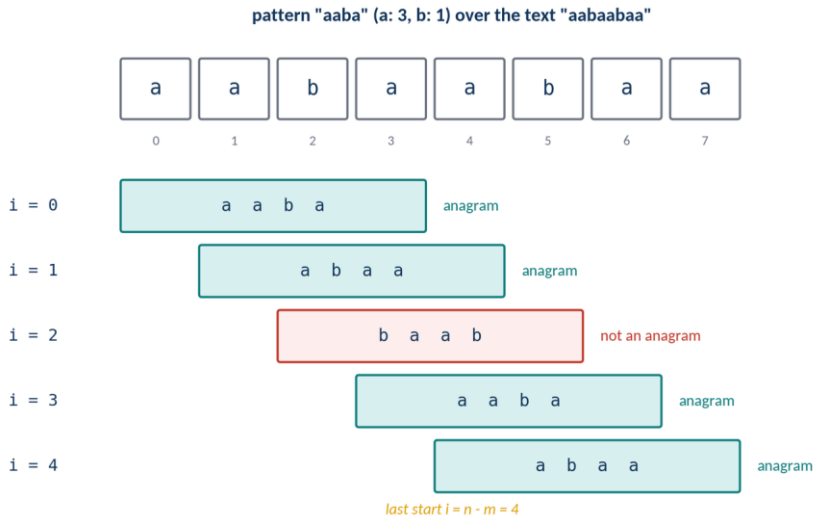


Figure 16.2 The five windows of length 4 in `aabaabaa`: four are anagrams of `aaba`; the last window starts at $n - m = 4$.

Our own examples: `abc` in `cbabcacab` (hits at 0, 2, 3 and 6, including overlapping ones); `wash` in `washwash`, where *every* window is an anagram, including the last one at index 4, which is exactly the one a careless loop misses; and `aab` in `ab`, a pattern longer than the text, which must give no hits and, above all, must not crash.

16.5.3 Designing it on paper

The straightforward design follows the statement: for each start i from 0 to $n - m$, ask `std::is_permutation` whether the window `text[i .. i+m-1]` is a permutation of the pattern. `is_permutation(first1, last1, first2)` compares the range from `first1` to `last1` with an equally long range starting at `first2` and returns `true` when both contain the same elements with the same counts [4].

Pseudocode 16.2 Finding anagrams by testing every window with a permutation check.

```
FIND-ANAGRAMS(text, pattern)
1  n = length(text), m = length(pattern)
2  if m == 0 or m > n
3      return an empty list                 // guard BEFORE computing n - m
4  hits = empty list
5  for i = 0 to n - m                       // inclusive: W = n - m + 1 windows
6      if IS-PERMUTATION(text[i .. i+m-1], pattern)
7          append i to hits
8  return hits
```

The cost. **Pseudocode 16.2** tests W windows. The library's `is_permutation` needs up to about m^2 comparisons for one window in the worst case, so the total is $O((n-m+1)\,m^2)$ in the worst case. For short patterns this is nothing. For long ones there is a faster idea, the *sliding count*: keep the number of each character in the current window in a table of 256 counters; moving the window one step to the right adds one character and removes one, two table updates; and a window is an anagram when its table equals the pattern's table. That costs $O(n+m)$ table updates plus one comparison of 256 counters per window, a constant. Our program implements both and checks that they agree on every input: two independent methods that give the same answer are the cheapest correctness proof there is.

16.5.4 The C++ solution

Listing 16.7 `anagram.cpp`: anagram search with `std::is_permutation` and with a sliding count.

```
// anagram.cpp - Problem 16.3: every place in a text where an anagram of a
// pattern starts, with std::is_permutation and again with a sliding count.
#include <algorithm>
#include <array>
#include <iostream>
#include <string>
#include <vector>

// Straightforward: test every window of length m.  O((n - m + 1) * m^2) worst.
std::vector<std::size_t> findAnagrams(const std::string& text,
                                      const std::string& pattern) {
    std::vector<std::size_t> hits;
    const std::size_t n = text.size(), m = pattern.size();
    if (m == 0 || m > n) return hits;          // guard BEFORE computing n - m
    for (std::size_t i = 0; i + m <= n; ++i)   // i runs 0 .. n - m inclusive
        if (std::is_permutation(text.begin() + i, text.begin() + i + m,
                                pattern.begin()))
            hits.push_back(i);
    return hits;
}

// Faster: keep letter counts of the window and slide it.  O(n + m) steps.
std::vector<std::size_t> findAnagramsFast(const std::string& text,
                                          const std::string& pattern) {
    std::vector<std::size_t> hits;
    const std::size_t n = text.size(), m = pattern.size();
    if (m == 0 || m > n) return hits;
    std::array<int, 256> need{}, have{};
    for (unsigned char c : pattern) ++need[c];
    for (std::size_t i = 0; i < n; ++i) {
        ++have[static_cast<unsigned char>(text[i])];          // enters on the right
        if (i >= m) --have[static_cast<unsigned char>(text[i - m])];   // leaves
        if (i + 1 >= m && have == need) hits.push_back(i + 1 - m);
    }
    return hits;
}
```

Listing 16.7 (Continued) *anagram.cpp: anagram search with std::is_permutation and with a sliding count.*

```
int main() {
    std::string pattern, text;
    while (std::cin >> pattern >> text) {
        std::vector<std::size_t> hits = findAnagrams(text, pattern);
        std::cout << "pattern \"" << pattern << "\" in \"" << text << "\"\n";
        for (std::size_t i : hits)
            std::cout << "  index " << i << ": " << text.substr(i, pattern.size())
                      << '\n';
        std::cout << "  " << hits.size() << " found; sliding count agrees: "
                  << (hits == findAnagramsFast(text, pattern) ? "yes" : "no")
                  << '\n';
    }
    return 0;
}
```

The input file `input/anagram.txt` holds the five pairs of Section 16.5.2:

```
aaba aabaabaa
abc cbabcacab
cat actcatac
aab ab
wash washwash
```

```
$ ./anagram < input/anagram.txt
```

```
pattern "aaba" in "aabaabaa"
  index 0: aaba
  index 1: abaa
  index 3: aaba
  index 4: abaa
  4 found; sliding count agrees: yes
pattern "abc" in "cbabcacab"
  index 0: cba
  index 2: abc
  index 3: bca
  index 6: cab
  4 found; sliding count agrees: yes
pattern "cat" in "actcatac"
  index 0: act
  index 2: tca
  index 3: cat
  index 5: tac
  4 found; sliding count agrees: yes
pattern "aab" in "ab"
  0 found; sliding count agrees: yes
pattern "wash" in "washwash"
  index 0: wash
  index 1: ashw
  index 2: shwa
  index 3: hwas
  index 4: wash
  5 found; sliding count agrees: yes
```

Three details of **Listing 16.7** deserve attention. First, the loop is written `for (std::size_t i = 0; i + m <= n; ++i)`, which is "$i \le n - m$" rearranged so that no subtraction appears; Section 16.5.6 shows why that matters. Second, the sliding version casts each character to `unsigned char` before using it as an index, because a `char` may be negative for characters outside plain ASCII, and a negative index is out of the array. Third, `have == need` compares two `std::array`s element by element; `std::array` is the C++ wrapper around a fixed C array, and unlike a C array it can be compared and copied with ordinary operators.

16.5.5 The Python solution

Python has no `is_permutation`, but two strings are permutations of each other exactly when their sorted letters are equal, and `collections.Counter` [5] gives the letter counts for the sliding version (**Listing 16.8**).

Listing 16.8 `anagram.py`: the same two methods in Python.

```
# anagram.py - Problem 16.3: every place in a text where an anagram of a
# pattern starts, by sorting each window and again with a sliding count.
import sys
from collections import Counter


def is_permutation(a, b):
    return sorted(a) == sorted(b)              # same letters, same counts


def find_anagrams(text, pattern):
    n, m = len(text), len(pattern)
    if m == 0 or m > n:
        return []
    return [i for i in range(n - m + 1)        # i runs 0 .. n - m inclusive
            if is_permutation(text[i:i + m], pattern)]


def find_anagrams_fast(text, pattern):
    n, m = len(text), len(pattern)
    if m == 0 or m > n:
        return []
    need, have, hits = Counter(pattern), Counter(), []
    for i, c in enumerate(text):
        have[c] += 1                           # enters on the right
        if i >= m:
            have[text[i - m]] -= 1             # leaves on the left
            if have[text[i - m]] == 0:
                del have[text[i - m]]
        if i + 1 >= m and have == need:
            hits.append(i + 1 - m)
    return hits


words = sys.stdin.read().split()
for pattern, text in zip(words[0::2], words[1::2]):
    hits = find_anagrams(text, pattern)
    print(f'pattern "{pattern}" in "{text}"')
    for i in hits:
        print(f"  index {i}: {text[i:i + len(pattern)]}")
    agree = "yes" if hits == find_anagrams_fast(text, pattern) else "no"
    print(f"  {len(hits)} found; sliding count agrees: {agree}")
```

`range(n - m + 1)` is the Python spelling of "0 to $n - m$ inclusive": `range` stops *before* its argument. When a count drops to zero, the sliding version deletes the letter from the `Counter`, so that two counters with the same non-zero counts compare as equal.

16.5.6 A tempting wrong solution

The classic mistake is the loop bound. `range(n - m)` in Python, or `i < n - m` in C++, looks natural and stops one window too early (**Listing 16.9**).

Listing 16.9 `wrong_anagram.py`: the loop stops one window too early.

```python
# wrong_anagram.py - a tempting wrong solution to Problem 16.3: the loop stops
# one window too early, because range(n - m) ends at n - m - 1.


def find_anagrams(text, pattern):
    n, m = len(text), len(pattern)
    return [i for i in range(n - m) if sorted(text[i:i + m]) == sorted(pattern)]


print(find_anagrams("aabaabaa", "aaba"))       # should be [0, 1, 3, 4]
print(find_anagrams("washwash", "wash"))       # should be [0, 1, 2, 3, 4]
```

```
[0, 1, 3]
[0, 1, 2, 3]
```

The hit at index 4 is lost in both texts, and nothing warns us: the program runs and prints a plausible list. Only the hand example of the statement, which lists index 4, exposes the error; this is why step 5 of Section 16.2.1 asks you to run the example of the statement first.

In C++ the same mistake has a second, worse form. `text.size()` has the *unsigned* type `std::size_t`, which cannot hold negative numbers. If the pattern is longer than the text, `n - m` does not become -1; it *wraps around* to the largest possible `size_t`, about 1.8×10^{19}, and a loop `for (i = 0; i < n - m; ++i)` then runs far past the end of the string and reads memory that does not belong to it. Our solution avoids both forms: it returns early when $m > n$, *before* any subtraction, and the loop condition `i + m <= n` contains no subtraction at all.

CHECK YOURSELF. Predict the output of the correct program for the pair `aa aaaa`, and the number of windows by Equation (16.3).

Answer. $n = 4$, $m = 2$, so $W = 4 - 2 + 1 = 3$ windows, at 0, 1 and 2, and each of them is `aa`. The program prints three hits, `index 0: aa`, `index 1: aa`, `index 2: aa`, then `3 found; sliding count agrees: yes`. The wrong loop would print only the first two.

16.6 Problem 16.4: Matrix Multiplication

16.6.1 The problem

PROBLEM 16.4. Read two matrices A and B. If the number of columns of A equals the number of rows of B, compute and print $C = A \times B$; otherwise print a message that the matrices cannot be multiplied. A partial program is provided that does not yet give correct results; correct it.

Our decisions: the input holds pairs of matrices, each written as the number of rows and columns followed by the values row by row; the program handles every pair in the file; the values are integers, and the products are computed in `long long`.

16.6.2 Understanding it by hand

DEFINITION. If A has n rows and m columns and B has m rows and p columns, their product $C = A \times B$ has n rows and p columns, and each element is the sum of the products of row i of A with column j of B:

$$C_{ij} = \sum_{k=0}^{m-1} A_{ik}\, B_{kj}, \qquad 0 \le i < n,\ 0 \le j < p \tag{16.4}$$

The dimensions work like dominoes: $(n \times m) \times (m \times p) = (n \times p)$; the two inner numbers must be equal and disappear. That is the check the problem asks for:

$$\text{cols}(A) = \text{rows}(B) \tag{16.5}$$

The first example belongs to Profitina Laundry. A holds the kilograms washed on Monday, Tuesday and Wednesday (rows) for the three services wash, wash-and-iron and iron (columns). B holds the price per kilogram of each service (rows) as a regular order and as an express order (columns). Then $C = A \times B$ has one row per day and one column per price type: C_{00} is Monday's revenue if every order was regular. **Figure 16.3** computes it by Equation (16.4): $12 \times 7{,}000 + 8 \times 10{,}000 + 5 \times 5{,}000 = 189{,}000$. For the second column, $12 \times 10{,}500 + 8 \times 15{,}000 + 5 \times 7{,}500 = 126{,}000 + 120{,}000 + 37{,}500 = 283{,}500$.

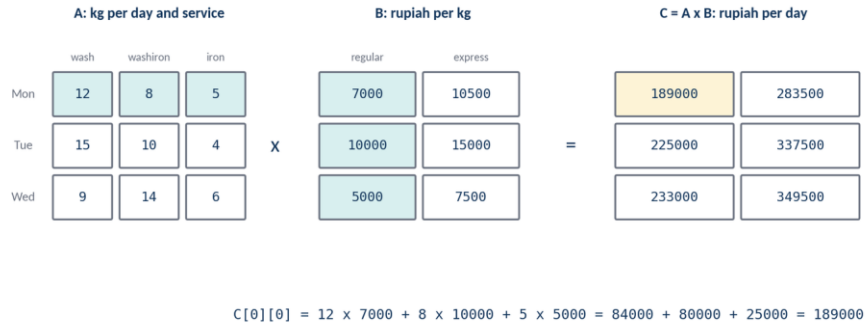


Figure 16.3 Daily kilograms per service times the price per kilogram gives the daily revenue; C_{00} combines row 0 of A with column 0 of B.

Our other examples: a small product computed completely by hand, $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix} = \begin{pmatrix} 58 & 64 \\ 139 & 154 \end{pmatrix}$ (for example $C_{00} = 1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 58$); a product with the identity matrix I, which must return A unchanged; and a pair that does not fit, a 2×3 times a 2×2, which Equation (16.5) refuses because $3 \neq 2$.

16.6.3 Designing it on paper

A matrix is a `std::vector<std::vector<long long>>`, a vector of rows; `a[i][k]` is row i, column k, as with the two-dimensional arrays of Chapter 6, but the size is chosen at run time and each vector knows its own size. The product is the triple loop of Equation (16.4):

Pseudocode 16.3 Matrix multiplication with the dimension check first.

```
MULTIPLY(A, B)                          // A is n x m, B is m' x p
1  if n == 0 or m != m'
2      return "cannot multiply"
3  C = an n x p matrix of zeros          // every sum starts at 0
4  for i = 0 to n - 1
5      for j = 0 to p - 1
6          for k = 0 to m - 1
7              C[i][j] = C[i][j] + A[i][k] * B[k][j]
8  return C
```

The innermost line runs $n \cdot p \cdot m$ times, so the cost is $O(nmp)$, or $O(n^3)$ for square matrices; faster algorithms exist for very large matrices, but the triple loop is the standard starting point [6]. When the partial program "does not give correct results", **Pseudocode 16.3** tells us where to

look: a missing line 1 (no check), a missing line 3 (sums that start from garbage), wrong loop bounds, or wrong indices in line 7.

16.6.4 The C++ solution

Listing 16.10 implements **Pseudocode 16.3** line for line.

Listing 16.10 `matrix.cpp`: matrix multiplication with a dimension check.

```
// matrix.cpp - Problem 16.4: matrix multiplication with a dimension check.
// Input: pairs of matrices, each written as "rows cols" followed by the values.
#include <iomanip>
#include <iostream>
#include <vector>

using Matrix = std::vector<std::vector<long long>>;

bool readMatrix(std::istream& in, Matrix& m) {
    std::size_t rows = 0, cols = 0;
    if (!(in >> rows >> cols)) return false;
    m.assign(rows, std::vector<long long>(cols, 0));
    for (auto& row : m)
        for (long long& x : row)
            if (!(in >> x)) return false;
    return true;
}

// C = A * B, allowed only when A's columns equal B's rows.
bool multiply(const Matrix& a, const Matrix& b, Matrix& c) {
    std::size_t n = a.size(), m = b.size(), p = b.empty() ? 0 : b[0].size();
    if (n == 0 || a[0].size() != m) return false;
    c.assign(n, std::vector<long long>(p, 0));          // every sum starts at 0
    for (std::size_t i = 0; i < n; ++i)
        for (std::size_t j = 0; j < p; ++j)
            for (std::size_t k = 0; k < m; ++k)         // sum of A[i][k] * B[k][j]
                c[i][j] += a[i][k] * b[k][j];
    return true;
}

void print(const Matrix& m) {
    for (const auto& row : m) {
        for (long long x : row) std::cout << std::setw(10) << x;
        std::cout << '\n';
    }
}
```

Listing 16.10 (Continued) *matrix.cpp: matrix multiplication with a dimension check.*

```
int main() {
    Matrix a, b, c;
    int number = 0;
    while (readMatrix(std::cin, a) && readMatrix(std::cin, b)) {
        std::cout << "Case " << ++number << ": A is " << a.size() << " x "
                  << a[0].size() << ", B is " << b.size() << " x " << b[0].size()
                  << '\n';
        if (!multiply(a, b, c)) {
            std::cout << "  cannot multiply: A has " << a[0].size()
                      << " columns but B has " << b.size() << " rows\n";
            continue;
        }
        std::cout << "  A x B is " << c.size() << " x " << c[0].size() << ":\n";
        print(c);
    }
    return 0;
}
```

The input file holds the four pairs of Section 16.6.2:

```
3 3
12  8 5
15 10 4
 9 14 6
3 2
 7000 10500
10000 15000
 5000  7500

2 3
1 2 3
4 5 6
3 2
 7  8
 9 10
11 12

2 2
2 3
5 7
2 2
1 0
0 1

2 3
1 2 3
4 5 6
2 2
1 2
3 4
```

```
Case 1: A is 3 x 3, B is 3 x 2
  A x B is 3 x 2:
    189000    283500
    225000    337500
    233000    349500
Case 2: A is 2 x 3, B is 3 x 2
  A x B is 2 x 2:
        58        64
       139       154
Case 3: A is 2 x 2, B is 2 x 2
  A x B is 2 x 2:
         2         3
         5         7
Case 4: A is 2 x 3, B is 2 x 2
  cannot multiply: A has 3 columns but B has 2 rows
```

Every number agrees with the hand computations. `m.assign(rows, std::vector<long long>(cols, 0))` makes a matrix of the right size filled with zeros in one line, which is line 3 of **Pseudocode 16.3**; `readMatrix` returns `false` when the input ends, which ends the loop in `main`. `multiply` returns `false` for an incompatible pair *before* touching `c`, and `main` prints the reason with both dimensions, so that the person who typed the file knows what to fix.

16.6.5 The Python solution

Listing 16.11 is the same, with lists of lists. `[[0] * p for _ in range(n)]` builds n separate rows; the shorter `[[0] * p] * n` would build *one* row and n references to it, so that writing to `c[0][0]` would change every row, a classic Python trap.

Listing 16.11 `matrix.py`: matrix multiplication in Python.

```
# matrix.py - Problem 16.4: matrix multiplication with a dimension check.
# Input: pairs of matrices, each written as "rows cols" followed by the values.
import sys


def read_matrix(tokens):
    """Takes one matrix from the front of the token list, or returns None."""
    if len(tokens) < 2:
        return None
    rows, cols = int(tokens.pop(0)), int(tokens.pop(0))
    if len(tokens) < rows * cols:
        return None
    return [[int(tokens.pop(0)) for _ in range(cols)] for _ in range(rows)]


def multiply(a, b):
    """C = A * B, or None when A's columns differ from B's rows."""
    n, m, p = len(a), len(b), len(b[0]) if b else 0
    if n == 0 or len(a[0]) != m:
        return None
    c = [[0] * p for _ in range(n)]            # every sum starts at 0
    for i in range(n):
        for j in range(p):
            for k in range(m):                 # C[i][j] = sum A[i][k] B[k][j]
                c[i][j] += a[i][k] * b[k][j]
    return c


tokens = sys.stdin.read().split()
number = 0
while True:
    a = read_matrix(tokens)
    b = read_matrix(tokens) if a is not None else None
    if b is None:
        break
    number += 1
    print(f"Case {number}: A is {len(a)} x {len(a[0])}, "
          f"B is {len(b)} x {len(b[0])}")
    c = multiply(a, b)
    if c is None:
        print(f"  cannot multiply: A has {len(a[0])} columns "
              f"but B has {len(b)} rows")
        continue
    print(f"  A x B is {len(c)} x {len(c[0])}:")
    for row in c:
        print("".join(f"{x:>10}" for x in row))
```

16.6.6 A tempting wrong solution

When a program that multiplies matrices is “almost right”, the most frequent planted bug is a swapped index: `b[j][k]` instead of `b[k][j]`. **Listing 16.12** makes that mistake and then tests itself the way many people do, with the identity matrix.

Listing 16.12 `wrong_matrix.cpp`: *B*’s indices swapped, tested with the identity matrix and with an ordinary matrix.

```cpp
// wrong_matrix.cpp - a tempting wrong solution to Problem 16.4: B's indices
// swapped, B[j][k] instead of B[k][j]. The identity test does not notice.
#include <iostream>
#include <vector>

using Matrix = std::vector<std::vector<long long>>;

Matrix multiplyWrong(const Matrix& a, const Matrix& b) {
    Matrix c(a.size(), std::vector<long long>(b[0].size(), 0));
    for (std::size_t i = 0; i < a.size(); ++i)
        for (std::size_t j = 0; j < b[0].size(); ++j)
            for (std::size_t k = 0; k < b.size(); ++k)
                c[i][j] += a[i][k] * b[j][k];  // should be b[k][j]
    return c;
}

void print(const char* title, const Matrix& m) {
    std::cout << title;
    for (const auto& row : m) std::cout << " [" << row[0] << ' ' << row[1] << ']';
    std::cout << '\n';
}

int main() {
    Matrix a = {{2, 3}, {5, 7}}, identity = {{1, 0}, {0, 1}}, b = {{1, 2}, {3, 4}};
    print("A x I =", multiplyWrong(a, identity));    // looks right
    print("A x B =", multiplyWrong(a, b));           // should be [11 16] [26 38]
    return 0;
}
```

```
A x I = [2 3] [5 7]
A x B = [8 18] [19 43]
```

The identity test *passes*: $A \times I$ comes out as A. It passes because swapping the indices of B multiplies by B's *transpose* (rows and columns exchanged), and the identity matrix is its own transpose, so the bug is invisible. The second test exposes it: the correct product of $\begin{pmatrix} 2 & 3 \\ 5 & 7 \end{pmatrix}$ and $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ is $\begin{pmatrix} 11 & 16 \\ 26 & 38 \end{pmatrix}$ ($C_{00} = 2 \cdot 1 + 3 \cdot 3 = 11$), but the program prints $\begin{pmatrix} 8 & 18 \\ 19 & 43 \end{pmatrix}$. The lesson reaches beyond matrices: a test is only as good as its power to fail. Test with *asymmetric* data, where every mistake changes the result, and compare with a value computed by hand. And with non-square matrices the swapped index is worse than wrong: `b[j][k]` can run past the end of a row, which is undefined behavior.

> **CHECK YOURSELF.** Without running the program, compute Wednesday's two revenues (C_{20} and C_{21}) in the laundry example, and say what the program prints for a 3×3 matrix A times a 2×3 matrix B.
>
> *Answer.* Row 2 of A is (9,14,6). $C_{20} = 9 \times 7{,}000 + 14 \times 10{,}000 + 6 \times 5{,}000 = 63{,}000 + 140{,}000 + 30{,}000 = 233{,}000$, and $C_{21} = 9 \times 10{,}500 + 14 \times 15{,}000 + 6 \times 7{,}500 = 94{,}500 + 210{,}000 + 45{,}000 = 349{,}500$, the last line of Case 1. For 3×3 times 2×3, Equation (16.5) fails ($3 \neq 2$), and the program prints `cannot multiply: A has 3 columns but B has 2 rows`.

16.7 Problem 16.5: When Can I Pick It Up?

16.7.1 The problem

> **PROBLEM 16.5.** Profitina Laundry is open from 08:00 to 20:00 every day. Washing takes 6 hours, washing and ironing 12 hours, ironing alone 3 hours; express orders take half the time. Laundry that becomes ready while the shop is closed can be picked up at 08:00 the next

> opening. Write a class `Time` for a moment in the week, with `+` (add minutes), `-` (minutes between two times), `<` and `==`, and printing in the form `Mon 09:15`. For each order, print the drop-off time, the time the laundry is ready, the pick-up time and the waiting time; then list the orders in pick-up order.

Our decisions: day 0 is a Monday, and days are counted on without limit (day 7 is the next Monday); a time is printed with the weekday name only; laundry ready at exactly 20:00 can still be picked up at 20:00; a `Time` with an hour outside 0–23 or a minute outside 0–59 is refused.

16.7.2 Understanding it by hand

The key to the class is its *invariant*: we store a time as a day number and the minutes after midnight, $0 \le \text{minute} < 1440$ (there are $24 \times 60 = 1440$ minutes in a day). Adding Δ minutes to (day, minute) gives

$$\text{day}' = \text{day} + \left\lfloor \frac{\text{minute} + \Delta}{1440} \right\rfloor, \qquad \text{minute}' = (\text{minute} + \Delta) \bmod 1440 \tag{16.6}$$

which restores the invariant after every addition. With it, the difference of two times is simply $1440 \cdot (\text{day}_a - \text{day}_b) + (\text{minute}_a - \text{minute}_b)$, and comparing two times is comparing that difference with zero.

Now the orders, by hand. Pak Budi drops off wash-and-iron, express, on Monday at 14:40: 720 minutes halved is 360; $880 + 360 = 1240$ minutes is 20:40, after closing, so the pick-up time is Tuesday 08:00, and the wait is $1440 + 480 - 880 = 1040$ minutes, 17 hours 20 minutes. Pak Hendra drops off wash-and-iron on Sunday (day 6) at 18:00: $1080 + 720 = 1800$, so by Equation (16.6) the day becomes $6 + \lfloor 1800/1440 \rfloor = 7$ and the minute $1800 \bmod 1440 = 360$, 06:00 on the next Monday; that is before opening, so he picks up at 08:00, after 14 hours.

16.7.3 Designing it on paper

Pseudocode 16.4 Computing the pick-up time from the drop-off time and the service.

```
PICKUP-TIME(order)
1  duration = 360 for wash, 720 for washiron, 180 for iron
2  if order is express
3      duration = duration / 2
4  ready = order.in + duration              // Time + minutes, normalized
5  if ready.hour < 8
6      return (ready.day, 08:00)            // wait for opening, same day
7  if ready.hour * 60 + ready.minute > 20 * 60
8      return (ready.day + 1, 08:00)        // closed: next morning
9  return ready
```

The operators follow Chapter 15: `+=` is a member that applies Equation (16.6); `+` is a non-member built on it; `-`, `<` and `==` compare through the difference. To sort the orders by pick-up time we use `std::stable_sort`, which keeps orders with the same pick-up time in their original order.

16.7.4 The C++ solution

Listing 16.13 `pickup_time.cpp`: the `Time` class with its operators, and the pick-up times of five orders.

```cpp
// pickup_time.cpp - Problem 16.5: a Time class with operators, used to tell
// customers when their laundry can be picked up (shop open 08:00 to 20:00).
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

class Time {
public:
    Time(int day, int hour, int minute) : day_(day), minute_(hour * 60 + minute) {
        if (day < 0) throw std::invalid_argument("day must be 0 or more");
        if (hour < 0 || hour > 23)
            throw std::invalid_argument("hour must be 0..23");
        if (minute < 0 || minute > 59)
            throw std::invalid_argument("minute must be 0..59");
    }
    int day() const { return day_; }
    int hour() const { return minute_ / 60; }
    int minute() const { return minute_ % 60; }

    Time& operator+=(int minutes) {            // keeps 0 <= minute_ < 1440
        int total = minute_ + minutes;
        day_ += total / 1440;
        minute_ = total % 1440;
        return *this;
    }
    // Minutes from b to a; the invariant makes this one line.
    friend int operator-(const Time& a, const Time& b) {
        return (a.day_ - b.day_) * 1440 + (a.minute_ - b.minute_);
    }
    friend bool operator<(const Time& a, const Time& b) { return (a - b) < 0; }
    friend bool operator==(const Time& a, const Time& b) { return (a - b) == 0; }

private:
    int day_;                                  // 0 = the first Monday
    int minute_;                               // minutes after midnight
};

Time operator+(Time t, int minutes) { return t += minutes; }
```

Listing 16.13 (Continued) *pickup_time.cpp: the Time class with its operators, and the pick-up times of five orders.*

```
std::ostream& operator<<(std::ostream& os, const Time& t) {
    static const char* names[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
    char oldFill = os.fill('0');
    auto oldFlags = os.flags();
    os << names[t.day() % 7] << ' ' << std::right << std::setw(2) << t.hour()
       << ':' << std::setw(2) << t.minute();
    os.fill(oldFill);                          // leave the stream as we found it
    os.flags(oldFlags);
    return os;
}

// Laundry that is ready while the shop is closed waits for 08:00.
Time pickupTime(const Time& ready) {
    if (ready.hour() < 8) return Time(ready.day(), 8, 0);
    if (ready.hour() * 60 + ready.minute() > 20 * 60)
        return Time(ready.day() + 1, 8, 0);
    return ready;
}

int processingMinutes(const std::string& service, bool express) {
    int m = service == "wash" ? 360 : service == "washiron" ? 720 : 180;
    return express ? m / 2 : m;                // express takes half the time
}

struct Ticket {
    int number;
    std::string customer, service;
    bool express;
    Time in, pickup;
};

int main() {
    std::vector<Ticket> tickets = {
        {1001, "Bu Rina", "wash", false, Time(0, 9, 15), Time(0, 0, 0)},
        {1002, "Pak Budi", "washiron", true, Time(0, 14, 40), Time(0, 0, 0)},
        {1003, "Mbak Sari", "iron", false, Time(0, 19, 30), Time(0, 0, 0)},
        {1004, "Mas Joko", "wash", true, Time(0, 10, 0), Time(0, 0, 0)},
        {1005, "Pak Hendra", "washiron", false, Time(6, 18, 0), Time(0, 0, 0)},
    };
    for (Ticket& t : tickets) {
        Time ready = t.in + processingMinutes(t.service, t.express);
        t.pickup = pickupTime(ready);
        int wait = t.pickup - t.in;
        std::cout << t.number << ' ' << std::left << std::setw(11) << t.customer
                  << "in " << t.in << "  ready " << ready << "  pick up "
                  << t.pickup << "  (" << wait / 60 << " h " << wait % 60
```

Listing 16.13 (Continued) *pickup_time.cpp: the Time class with its operators, and the pick-up times of five orders.*

```
                  << " min)\n";
    }
    std::cout << "1002 and 1003 together: "
              << (tickets[1].pickup == tickets[2].pickup ? "yes" : "no") << '\n';
    std::stable_sort(tickets.begin(), tickets.end(),   // earliest pick-up first
                     [](const Ticket& a, const Ticket& b) {
                         return a.pickup < b.pickup;
                     });
    std::cout << "Pick-up order:";
    for (const Ticket& t : tickets) std::cout << ' ' << t.number;
    std::cout << '\n';
    try {
        Time bad(0, 24, 0);
        std::cout << bad << '\n';
    } catch (const std::invalid_argument& e) {
        std::cout << "Time(0, 24, 0) refused: " << e.what() << '\n';
    }
    return 0;
}
```

```
1001 Bu Rina    in Mon 09:15  ready Mon 15:15  pick up Mon 15:15  (6 h 0 min)
1002 Pak Budi   in Mon 14:40  ready Mon 20:40  pick up Tue 08:00  (17 h 20 min)
1003 Mbak Sari  in Mon 19:30  ready Mon 22:30  pick up Tue 08:00  (12 h 30 min)
1004 Mas Joko   in Mon 10:00  ready Mon 13:00  pick up Mon 13:00  (3 h 0 min)
1005 Pak Hendra in Sun 18:00  ready Mon 06:00  pick up Mon 08:00  (14 h 0 min)
1002 and 1003 together: yes
Pick-up order: 1004 1001 1002 1003 1005
Time(0, 24, 0) refused: hour must be 0..23
```

The results match the hand work. Three features of **Listing 16.13** are new. `operator-`, `operator<` and `operator==` are declared `friend` inside the class: a *friend* function is not a member but may read the private data, which is convenient for symmetric operators that need both objects' internals. `operator<<` saves and restores the stream's fill character and flags, because it switches to `'0'` padding for the hours and minutes and must not leave the stream in that state for the next output (an early version of this program printed `Mon 90:15` for 09:15, because the `std::left` set for the customer's name was still active; restoring the flags and setting `std::right` locally fixed it). And the comparison passed to `std::stable_sort` is a *lambda*, a small unnamed function written in place, `[](const Ticket& a, const Ticket& b) { return a.pickup < b.pickup; }`, which uses our `operator<` for `Time`.

16.7.5 The Python solution

Listing 16.14 is the same class in Python; `list.sort` is always stable, and `key=lambda t: t[5]` sorts by the pick-up time using `Time.__lt__`.

Listing 16.14 `pickup_time.py`: the `Time` class in Python.

```python
# pickup_time.py - Problem 16.5: a Time class with operators, used to tell
# customers when their laundry can be picked up (shop open 08:00 to 20:00).
from functools import total_ordering

NAMES = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]


@total_ordering
class Time:
    def __init__(self, day, hour, minute):
        if day < 0:
            raise ValueError("day must be 0 or more")
        if not 0 <= hour <= 23:
            raise ValueError("hour must be 0..23")
        if not 0 <= minute <= 59:
            raise ValueError("minute must be 0..59")
        self._day, self._minute = day, hour * 60 + minute

    day = property(lambda self: self._day)
    hour = property(lambda self: self._minute // 60)
    minute = property(lambda self: self._minute % 60)

    def __add__(self, minutes):                # keeps 0 <= _minute < 1440
        total = self._minute + minutes
        t = Time(self._day, 0, 0)
        t._day += total // 1440
        t._minute = total % 1440
        return t

    def __sub__(self, other):                  # minutes from other to self
        return (self._day - other._day) * 1440 + (self._minute - other._minute)

    def __lt__(self, other):
        return self - other < 0

    def __eq__(self, other):
        return self - other == 0

    def __str__(self):
        return f"{NAMES[self.day % 7]} {self.hour:02d}:{self.minute:02d}"
```

Listing 16.14 (Continued) *pickup_time.py: the Time class in Python.*

```
def pickup_time(ready):
    """Laundry that is ready while the shop is closed waits for 08:00."""
    if ready.hour < 8:
        return Time(ready.day, 8, 0)
    if ready.hour * 60 + ready.minute > 20 * 60:
        return Time(ready.day + 1, 8, 0)
    return ready


def processing_minutes(service, express):
    m = {"wash": 360, "washiron": 720}.get(service, 180)
    return m // 2 if express else m            # express takes half the time


tickets = [
    [1001, "Bu Rina", "wash", False, Time(0, 9, 15)],
    [1002, "Pak Budi", "washiron", True, Time(0, 14, 40)],
    [1003, "Mbak Sari", "iron", False, Time(0, 19, 30)],
    [1004, "Mas Joko", "wash", True, Time(0, 10, 0)],
    [1005, "Pak Hendra", "washiron", False, Time(6, 18, 0)],
]
for t in tickets:
    number, customer, service, express, time_in = t
    ready = time_in + processing_minutes(service, express)
    pickup = pickup_time(ready)
    t.append(pickup)
    wait = pickup - time_in
    print(f"{number} {customer:<11}in {time_in}  ready {ready}  pick up {pickup}"
          f"  ({wait // 60} h {wait % 60} min)")
together = tickets[1][5] == tickets[2][5]
print(f"1002 and 1003 together: {'yes' if together else 'no'}")
tickets.sort(key=lambda t: t[5])               # Python's sort is stable
print("Pick-up order:" + "".join(f" {t[0]}" for t in tickets))
try:
    bad = Time(0, 24, 0)
    print(bad)
except ValueError as e:
    print(f"Time(0, 24, 0) refused: {e}")
```

16.7.6 A tempting wrong solution

The tempting design stores hours and minutes separately, as they are printed, and adds minutes to the minute field (**Listing 16.15**).

Listing 16.15 `wrong_time.py`: hours and minutes kept apart, without carrying.

```
# wrong_time.py - a tempting wrong solution to Problem 16.5: hours and minutes
# kept apart, and minutes added without carrying into hours and days.


class WrongTime:
    def __init__(self, hour, minute):
        self.hour, self.minute = hour, minute

    def add(self, minutes):
        return WrongTime(self.hour, self.minute + minutes)   # no carry!

    def __lt__(self, other):
        return (self.hour, self.minute) < (other.hour, other.minute)

    def __str__(self):
        return f"{self.hour:02d}:{self.minute:02d}"


a = WrongTime(14, 40).add(360)                  # should be 20:40
b = WrongTime(19, 30).add(180)                  # should be 22:30
closing = WrongTime(20, 0)
print(a, b)
print("b ready before closing?", b < closing)  # 22:30 is after 20:00: False
```

```
14:400 19:210
b ready before closing? True
```

Pak Budi's laundry is "ready at 14:400", and Mbak Sari's at "19:210", which the program then believes is before closing time, because the comparison looks at the hour first and 19 is less than 20. The invariant of Section 16.7.2 was never established, so every operation that relies on it is wrong. The fix is not a smarter comparison; it is a representation in which invalid values cannot exist, which is exactly what Equation (16.6) provides.

> **CHECK YOURSELF.** Mbak Sari brings ironing, express, on Friday (day 4) at 18:30. When can she pick it up, and how long does she wait? What if she comes at 18:31?
>
> *Answer.* Ironing takes 180 minutes, express 90. 18:30 plus 90 minutes is 20:00; that is not *after* 20:00, so she picks up on Friday at 20:00 after 1 hour 30 minutes. At 18:31 the laundry is ready at 20:01, after closing, so she picks up on Saturday at 08:00, after $1440 + 480 - 1111 = 809$ minutes, 13 hours 29 minutes. One minute at the counter changes the answer by almost half a day, which is why boundary cases belong in every test.

16.8 The Problems Side by Side

Table 16.1 collects the five problems: the main tool, the cost, the trap, and the test that catches the trap.

Table 16.1 The five problems of this chapter side by side.

Problem	Main tool	Cost	Trap	Test that catches it
16.1 Chess pieces	inheritance, virtual function	$O(1)$ per move	logical *or* instead of `^`; standing still	a diagonal move for the rook; a move to the same square
16.2 Overloading `add`	overloading, reuse	$O(k)$ for k arguments	defaults next to an overload	compiling a call with two arguments
16.3 Anagrams	strings, `is_permutation`	$O((n-m+1)m^2)$, or $O(n+m)$ sliding	last window missed; unsigned wrap-around	the statement's example; pattern longer than text
16.4 Matrices	vectors of vectors	$O(nmp)$	`b[j][k]`; no dimension check	an asymmetric hand example, not only I
16.5 Pick-up time	class invariant, operators	$O(1)$ per operation	minutes without carrying	a time that crosses an hour and a day

Read down the last column: every trap is caught by a *specific* example chosen in step 2 of the method. That is the most important lesson of the four evaluation chapters of this book. The examples you choose before you write code decide which mistakes you will find.

16.9 Using It for Real

The five problems are small, but each is the core of a real task at Profitina Laundry or any small business. The chess hierarchy is the shape of every "several kinds of one thing" in a program: services priced per kilogram or per piece, customers who are members or not, payments in cash or by QRIS transfer. The overloads of `add` are the everyday use of one name for one idea across types. The sliding window is how a program finds a pattern in a long stream: the busiest three-hour window of the day, or the seven-day period with the most express orders. The matrix product turns kilograms into money for every day and price list at once; with the price matrix changed, the same multiplication answers "what if we raise the ironing price by 10%?". And the `Time` class, extended with the shop's real opening hours (shorter on Sundays, closed on Idul Fitri), is what prints the pick-up time on every receipt.

When you adapt them, keep the method. Restate the problem with its decisions written down, work examples by hand, including the boundary ones, design before coding, and keep the hand examples as tests that run after every change.

16.10 Common Mistakes, and How to Avoid Them

- **Translating a rule from a loose sentence.** "Moves along a row or a column" is not `||`. Write the rule as a condition on d_x and d_y and check it on examples, including "stay put".
- **Changing given code instead of imitating it.** When a problem gives a model class, the new classes should follow it exactly: the same constructor pattern, the same output format, the same names. Creativity belongs in the conditions, not in the structure.
- **Mixing default arguments and overloads.** Two functions that can both accept the same call make every such call ambiguous. Choose one mechanism for one family of functions.
- **Off-by-one in window loops.** There are $n-m+1$ windows (Equation (16.3)); the last start is $n-m$, inclusive. In C++, guard $m > n$ before any subtraction of unsigned sizes.
- **Testing only with symmetric data.** The identity matrix hides index mistakes. Use a hand-computed asymmetric example.

- **Forgetting to initialize sums.** Every C_{ij} must start at 0. `assign(n, std::vector<long long>(p, 0))` does it in one line.
- **Storing a value in a form that allows invalid states.** Hours and minutes kept apart allow 14:400. Choose a representation with an invariant, such as minutes since midnight, and restore the invariant in every operation.
- **Leaving a stream in a changed state.** An `operator<<` that sets `fill('0')` or `std::left` must restore the old state, or the next unrelated output is corrupted.

16.11 Chapter Summary

- A seven-step method (restate, examples by hand, design, code, run the examples, try to break it, change one thing and predict) solved five problems that use the C++ of Chapters 13 to 15 and the foundations of the whole book.
- Problem 16.1 completed a class hierarchy by imitation; each piece's rule became an exact condition, with exclusive or for the rook (Equation (16.1)) and a shared `move` that calls a virtual `moveCondition`.
- Problem 16.2 overloaded `add` for two to five arguments, each built on the one below (Equation (16.2)); default arguments added beside an existing overload make calls ambiguous.
- Problem 16.3 found anagrams in $W = n - m + 1$ windows (Equation (16.3)) with `std::is_permutation`, and again with a linear-time sliding count; the loop bound and the unsigned subtraction are the traps.
- Problem 16.4 multiplied matrices by Equation (16.4) after checking Equation (16.5); the identity matrix does not reveal swapped indices, an asymmetric hand example does.
- Problem 16.5 built a `Time` class whose invariant (Equation (16.6)) makes addition, difference and comparison one line each, and used it to compute pick-up times around opening hours.
- Every solution exists in C++ and Python with identical output, checked automatically; every tempting wrong solution fails in a way that a well-chosen example reveals.

16.12 Exercises

1. Add a class `King` to `chess.cpp` and `chess.py` (the condition of the first "check yourself" box) and test it from (4,4) with a legal, an illegal and a second legal move.
2. Extend Problem 16.1 with a `Board` class that holds several pieces and refuses a move of a rook, bishop or queen if another piece stands on a square strictly between the start and the destination. Describe the data structure and the check; the code is optional.
3. Write `add` overloads for two to five `Rupiah` arguments (Chapter 15), each built on `operator+`. Is overloading `add` still useful when `+` is available?
4. Change `anagram.cpp` so that capital and small letters count as the same letter (`Wash` is an anagram of `shaw`). Which of the two methods needs which change?
5. Add a function `transpose(const Matrix&)` to `matrix.cpp` and use it to show, on the example of Section 16.6.6, that the wrong program computes $A \times B^{\mathsf{T}}$.
6. Give the `Time` class of Problem 16.5 real opening hours per weekday: 08:00 to 20:00 on Monday to Saturday, 09:00 to 15:00 on Sunday. What is Pak Hendra's pick-up time now?

16.13 Solutions to the Exercises

1. `class King : public Piece` with the constructor pattern of the others and `moveCondition` returning `std::max(std::abs(xDest - x_), std::abs(yDest - y_)) == 1` (`std::max` needs `<algorithm>`); in Python, `max(abs(x_dest - self._x), abs(y_dest - self._y)) == 1`. Starting it at (4,4) needs the optional start coordinates: `King(1, 4, 4)`. Moves (5,5), (5,7), (4,4) give: moved to (5,5); invalid, because $|d_y| = 2$; moved back to (4,4).

2. A `Board` holds a `std::vector<Piece*>` (or an 8×8 array of `Piece*` with `nullptr` for empty squares, which makes "is this square occupied?" one look-up). For a straight or diagonal move, the squares strictly between start and destination are found by stepping with $s_x = \text{sign}(d_x)$ and $s_y = \text{sign}(d_y)$ (each -1, 0 or $+1$) from the start, $\max(|d_x|, |d_y|) - 1$ times; if any of them is occupied, the move is invalid. Knights jump and need no such check. The pieces must then be able to ask the board, so `moveCondition` gets a `const Board&` parameter: a small design change that shows why "a piece alone on the board" was an important decision in Section 16.3.1.

3. `Rupiah add(const Rupiah& a, const Rupiah& b) { return a + b; }` and the three larger overloads built on it as in Equation (16.2). With `operator+` available, `a + b + c` is already shorter than `add(a, b, c)`, so these overloads add nothing; they are useful only when a function name is required, for example to pass "adding" to another function. Choosing the natural notation is part of good design.

4. Lower-case both strings first with `std::tolower` (and the `unsigned char` cast of Section 13.4), once, before searching. Neither method then needs any change, and the printed substrings should be taken from the *original* text so that the output shows the real letters. In Python, `text.lower()` and `pattern.lower()` do the same.

5. `transpose` builds a $p \times n$ matrix `t` with `t[j][i] = m[i][j]`. `multiply(a, transpose(b), c)` on $A = \begin{pmatrix} 2 & 3 \\ 5 & 7 \end{pmatrix}$ and $B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, with $B^{\mathsf{T}} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$, gives $\begin{pmatrix} 8 & 18 \\ 19 & 43 \end{pmatrix}$, exactly the wrong program's output ($C_{00} = 2 \cdot 1 + 3 \cdot 2 = 8$). And for the identity, $I^{\mathsf{T}} = I$, which is why that test passed.

6. Replace the two constants 8 and 20 by a small table of opening and closing minutes per weekday, looked up with `day % 7`. The rule of **Pseudocode 16.4** becomes: if the ready time is before that day's opening, pick up at the opening; if after that day's closing, pick up at the *next day's* opening. Pak Hendra's laundry is ready at 06:00 on day 7, a Monday, which opens at 08:00, so nothing changes for him: Monday 08:00. A Sunday drop-off of ironing at 13:00 would change, though: ready at 16:00, after the Sunday closing time of 15:00, so the pick-up moves to Monday at 08:00.

Sixteen chapters ago, your first program printed one line. Since then you have learned to make a computer decide, repeat, remember, read files, and now to model the things of the real world as classes that guard their own rules. The problems of this chapter are small, but the method you used on them scales to programs of any size: say exactly what you want, try it by hand, design before you type, and let well-chosen examples test every step.

References

[1] B. Stroustrup, *The C++ Programming Language*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.

[2] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ, USA: Princeton University Press, 1957.

[3] cppreference.com, “Overload resolution,” accessed Sep. 2026. [Online]. Available: https://en.cppreference.com/w/cpp/language/overload_resolution

[4] cppreference.com, “std::is_permutation,” accessed Sep. 2026. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/is_permutation

[5] Python Software Foundation, “collections — Container datatypes: Counter,” Python 3.14 documentation, accessed Sep. 2026. [Online]. Available: https://docs.python.org/3.14/library/collections.html

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.