Lecture 1 — Computers, C and Your First Programs

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 1

Lecturer: Irfan Subakti — yifana@gmail.com

Today we go from nothing to a working program: what a computer does with a program, why we learn C and Python side by side, the free workbench, and the first programs, ending with version 0.1 of the Profitina Laundry cashier. Book: Chapter 1, Sections 1.1–1.14.

What This Meeting Covers

1. Why programs — a laundry counter where arithmetic by hand goes wrong.

2. What a computer does — hardware and software, languages, compiled and interpreted.

3. The workbench — VS Code, GCC and Python 3.14 on Windows, macOS and Linux.

4. First programs — `hello`, variables and memory, `scanf` and `input`, four basic types.

5. Arithmetic and decisions — five operators, integer division, precedence, `if`; then cashier v0.1 and its improvement.

Every program exists in C and in Python, and both versions print exactly the same output. Every output shown is the output the program really printed. Book: Chapter 1.

Where We Begin: a Receipt Written by Hand

Profitina Laundry is a small laundry near a university. Mbak Rina weighs each bag and writes the price on a paper receipt.

Wash, dry and fold costs Rp 8,000 per kilogram; one-day express Rp 12,000; ironing only Rp 6,000.

On a busy Monday she writes Rp 34,000 for 4.3 kg instead of Rp 34,400, forgets the three-kilogram minimum, and the day's total does not match the drawer.

A computer does not get tired and does not forget a rule once it is written down. Writing the rule down is **programming**; the written rule is a **program**.

The laundry stays with us all semester: one receipt today, a queue of customers in Chapter 2, a validated daily report in Chapter 3, and later files, sorting and C++ classes. Book: Section 1.1.

What a Computer Does

Hardware is the machine: a processor that carries out instructions, memory that holds data, storage, input and output devices.

Software is the instructions. Memory is a row of numbered bytes; the number of a byte is its **address**.

The processor understands only **machine language**. A **high-level language** such as C or Python lets us write in terms of the problem, and a translator does the rest.

The address idea looks small today. It comes back in `scanf` this week, in arrays in Chapter 6 and in pointers in Chapter 7. Book: Section 1.2.

C and Python: Two Languages, One Way of Thinking

C (Dennis Ritchie, Bell Labs, 1969–1973) runs the kernels of Linux, Windows and macOS, databases, and the Python interpreter itself. We use the C17 standard, ISO/IEC 9899:2018.

C **hides very little**: you choose the bytes of a number, you pass an address to change a variable, and nothing stops a read outside an array.

Python (Guido van Rossum, 1991) does the bookkeeping: numbers grow as needed, lists grow, memory is released automatically.

The *ideas* (variable, decision, loop, function) are the same in both; the *differences* show what Python does for you.

In the TIOBE index of September 2026 Python is first with 17.76% and C second with 10.28%. The current Python is 3.14, maintenance release 3.14.7. Book: Sections 1.3.1 and 1.3.2.

Compiled and Interpreted

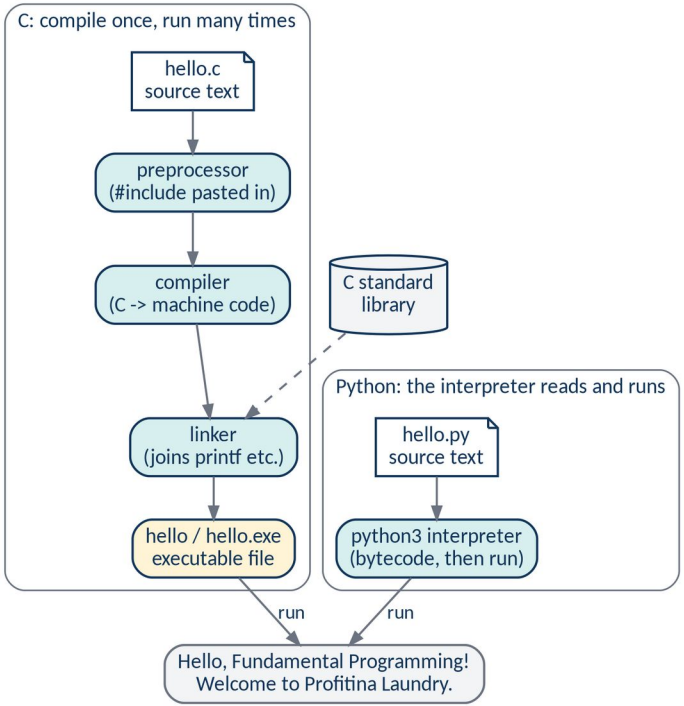


Figure 1.1 Two routes from source text to output: a C program passes through the preprocessor, the compiler and the linker once and becomes an executable file; a Python program is read and run by the interpreter every time.

*Figure 1.1: C is **compiled** once into an **executable** of machine language, which then runs without the compiler. Python is **interpreted**: `python3` reads the source every run, turns it into bytecode and carries it out at once. Book: Section 1.3.3.*

The Workbench

Table 1.1 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	the editor: colors the code, runs it, helps find errors
GCC (Windows, Linux)	16.2 newest; any from 9 on works	the C compiler, command `gcc`
Apple clang (macOS)	with the Xcode Command Line Tools	a C compiler that also answers to `gcc`
Python	3.14.7	the interpreter, `python3` (`python` or `py` on Windows)
VS Code extensions	*C/C++* and *Python*, by Microsoft	colors, error checking, debugging

Table 1.1: on Windows GCC comes from MSYS2, on macOS from the Xcode Command Line Tools, on Linux from `build-essential`. Everyone installs VS Code and its two extensions. Book: Section 1.4.

Installing on Each System

Windows — MSYS2 into `C:\msys64`; in **MSYS2 UCRT64** run `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`; add `C:\msys64\ucrt64\bin` to Path; Python install manager, then `py install 3.14`.

macOS — `xcode-select --install` for clang; the python.org installer for Python 3.14; VS Code into Applications, then *Install 'code' command in PATH*.

Linux — `sudo apt install build-essential gdb python3`. Ubuntu 26.04 LTS ships GCC 15.2 and Python 3.14; Python 3.10 or later runs every program of the book.

Check in a *new* terminal: `gcc --version`, `python3 --version` (or `python`), `code --version`.

> **KEY INSIGHT.** Keep one folder per chapter and open that folder, not a single file, in VS Code. The terminal (**View → Terminal**) then starts in the right place and the commands of the book work as printed.

A terminal reads the Path only when it starts: close every open terminal after changing it. On a Mac `gcc --version` answers "Apple clang", which accepts the same options. Book: Sections 1.4.1–1.4.5.

One Compile Command for the Whole Course

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

`-std=c17` — follow the C17 standard. `-Wall -Wextra` — show the usual warnings and some extra ones.

`-O2` — optimize the machine code. `-o hello` — name the executable `hello` (`hello.exe` on Windows).

Programs that use the mathematics library add `-lm` at the end. Run with `./hello` (Linux, macOS) or `.\hello.exe` (PowerShell).

Every program of the book was also run with `-fsanitize=address,undefined`, which stops a program the moment it touches memory it does not own. Book: Sections 1.4 and 1.6.1.

The First 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 */
}
```

Listing 1.1 is nine lines with six ideas that come back in every C program. We read them on the next page. Book: Section 1.5.

Six Ideas in Nine Lines

`/* ... */` is a **comment**, for people only; `//` comments run to the end of the line.

`#include <stdio.h>` tells the **preprocessor** to paste in the declarations of `printf` and `scanf`.

`int main(void)` — every program starts at the first line of `main`; it gives back an `int` and receives nothing.

Braces enclose the **body**; statements run top to bottom and end with `;`. The escape sequence `\n` means “new line”.

`return 0;` reports success to the operating system; any other number means something went wrong.

Ask the room: what happens if we drop the `\n` from the first `printf`? Both greetings land on one line. Book: Section 1.5.

The Same Program in Python

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

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

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

Listing 1.2 has no `#include`, no `main`, no braces and no semicolons: Python runs the file from top to bottom, and `print` adds the new line itself. Run it with `python3 hello.py`. If you see these two lines, your workbench works. Book: Section 1.5.

Errors, Warnings and Logic Errors

Error — no executable is produced. `hello.c:6:48: error: expected ';' before 'printf'` means file, line 6, column 48.

The compiler often notices a mistake one line *late*: when a message points at a perfect line, look at the line above.

Warning — the program is translated, but it is very probably not what you meant. Treat every warning as an error in disguise.

Logic error — valid and running, but computing the wrong thing. Only testing against answers known in advance finds it.

> **WARNING.** A program that runs is not yet a program that is correct. Never ignore a warning because "the program runs anyway".

Python reports a missing parenthesis as `SyntaxError: '(' was never closed` before any line runs. Book: Section 1.6.2.

A Variable Is a Named Box

DEFINITION. A *variable* is a named place in memory that holds one value. It has a **name**, a **type**, a **value** and an **address**.

`int integer1;` is a **declaration**: a box big enough for one whole number, named `integer1`.

Names use letters, digits and `_`, do not start with a digit, and are case sensitive. `weightKg` beats `w`.

`sum = integer1 + integer2;` is an **assignment**: compute the right side, store it in the box on the left. `=` means "becomes".

`count = count + 1;` *is nonsense as mathematics and perfect sense as an instruction. Book: Section 1.7.1.*

Three Variables in Memory

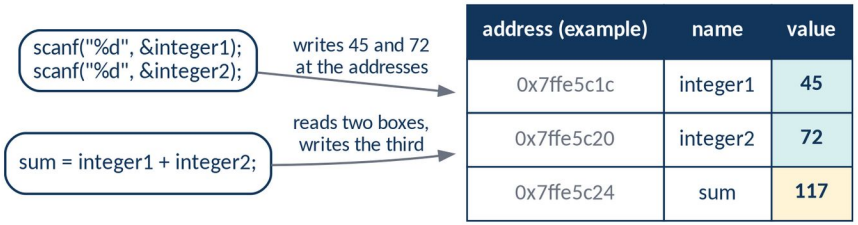


Figure 1.2 Three variables after a run of `add_two`: `scanf` wrote 45 and 72 to the addresses of `integer1` and `integer2`, and the assignment read both and wrote 117 into `sum`.

Figure 1.2 shows name, value and address together. The addresses are examples; they differ on every computer and every run. Book: Section 1.7.1.

Reading Input: `scanf` and `input`

In `scanf("%d", &integer1)` the **format** `"%d"` reads a decimal whole number; `&integer1` is the **address** of the box.

`scanf` needs the address because it *writes* into the box; `printf` only reads the value, so it gets `integer1` without `&`.

`scanf` returns how many values it read. `if (scanf(...) != 1) return 1;` stops the program when the user types `abc`.

Python's `input` returns **text**; `int(...)` or `float(...)` converts it into a number.

The check is also needed for a clean compile: on many Linux systems GCC warns when the result of `scanf` is ignored. Book: Section 1.7.2.

Four Basic Types

Listing 1.3 From `types.c`: an `int`, a `double` and a `char`, and the largest `int`.

```c
    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("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);
```

```
bill  = 40000.00
grade = A, stored as the number 65
largest int  = 2147483647
```

`int`: four bytes, −2147483648 to 2147483647. `double`: eight bytes, about 15 to 16 significant digits. `char`: one byte. `long long`: eight bytes, up to about 9.2×10^{18}.

Listing 1.3: `%d`, `%f` and `%c` are the conversion specifications; printed with `%d`, the letter A shows its ASCII number 65. `INT_MAX` comes from `limits.h`. Python has no `char` and no limit on `int`. Book: Section 1.7.3.

Five Operators and One Trap

`+`, `-`, `*`, `/` and `%` (remainder). When both sides of `/` are whole numbers, C does **integer division**: `7 / 3` is 2. `%` gives what integer division threw away: `7 % 3` is 1. The two always fit together:

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

For $a=7$, $b=3$, Equation (1.1) reads $7 = 2\cdot 3 + 1$. For negatives C cuts toward zero: `-7 / 2` is -3 and `-7 % 2` is -1.

Change for a customer, minutes into hours, "every third turn" with `turn % 3 == 0`: all are integer division and remainder. Book: Section 1.8.1.

Worked Example 1: the Arithmetic Program

Listing 1.4 From `arithmetic.c`: integer division, the remainder, a `double`, and precedence.

```c
    printf("7 / 3 = %d\n", 7 / 3);        /* whole-number division */
    printf("7 %% 3 = %d\n", 7 % 3);       /* the remainder */
    printf("7.0 / 2 = %.2f\n", 7.0 / 2);  /* one double makes it exact */
    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);
```

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

Work Listing 1.4 on the board before showing the output. `%%` prints one percent sign. One `double` on either side keeps the fraction. Book: Section 1.8.1.

Precedence: Who Goes First

Table 1.2 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; left to right
`+ -`	addition, subtraction	next; left to right
`=`	assignment	last

By Table 1.2, `a + b + c / 3` divides only `c`: 85/3=28, plus 170, gives 198, which is not even between the scores.

In Python `/` always keeps the fraction and `//` is integer division, rounding down, not toward zero. Parentheses cost nothing and say what you meant. Book: Sections 1.8.2 and 1.8.3.

Decisions with `if`

`if (condition) statement` runs the statement only when the condition is true.

Six comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`.

`=` assigns; `==` compares. `if (x = 5)` assigns 5 and is always true; `-Wall` warns about it.

In Python the body is the indented block after `if condition:`; indentation is grammar, not decoration.

The book's `compare.c` guards one `printf` with each of the six operators; run it with 7 and 7, then 3 and 9, and ask which lines appear. Book: Section 1.9.

Worked Example 2: Cashier v0.1

Listing 1.5 From `receipt.c`: cashier v0.1 reads a weight and prints the total.

```c
    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("Weight (kg): ");
    if (scanf("%d", &weightKg) != 1) return 1;
    total = weightKg * pricePerKg;
    printf("TOTAL    : Rp %d\n", total);
```

Every variable's comment states its **unit**. Money is a whole number of rupiah: exact, no cents.

Listing 1.5 is the core of the full program in the book, which also prints the service and the price per kilogram. For 5 kg it prints `TOTAL    : Rp 40000`. Book: Section 1.10.

Cashier v0.1 in Python

Listing 1.6 From `receipt.py`: cashier v0.1 in Python.

```
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(f"TOTAL    : Rp {total}")
```

Listing 1.6: `int(input(...))` *turns the typed text into a number, and the f-string puts the value into the text.* `print()` *alone prints an empty line, the counterpart of a leading* `\n`*. Book: Section 1.10.*

Making It Better: Decimals and a Minimum

Listing 1.7 From `receipt_min.c`: decimal weights, a three-kilogram minimum, and rounding to whole rupiah.

```
    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 */
```

`double weight` is read with `%lf` and printed with `%.1f`, one decimal, as the scale shows it.

Charge the real weight; *if* it is below 3.0, charge 3.0 instead. Adding 0.5 before the cast to `int` rounds to the nearest rupiah.

Listing 1.7 is the heart of the improved cashier. Braces around the `if` body make clear which statements belong to it. Book: Section 1.11.

Two Runs, Two Paths

4.3 kg is above the minimum: $4.3 \times 8000 = 34400$.

2.2 kg is below it: the receipt says `Minimum  : 3.0 kg charged` and the customer pays $3 \times 8000 = 24000$.

```
Weight (kg): 2.2

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

Write the expected receipt on paper before the run, then compare. Test the smallest bag, the largest, one exactly at 3.0 kg, and one with a decimal. Book: Sections 1.11 and 1.12.

Using It for Real

The same shape fits a photocopy shop (per page), a water refill depot (per gallon), a parking lot (per hour), a rice seller (per kilogram).

To adapt `receipt_min`: change the service name, the price per unit, and the minimum (or remove it).

A program is only as correct as its rules: put the prices and the date they were valid in a comment at the top.

Chapter 11 moves prices out of the program into a file the owner can edit. Book: Section 1.12.

Common Mistakes

A missing `;` — reported on the next line. **A missing `&` in `scanf`** — a warning, then a crash or garbage.
Wrong conversion — a `double` printed with `%d` or read with `%f` instead of `%lf`. **`=` instead of `==`.**
Running an old executable after editing, without compiling again. **Mixed tabs and spaces** in Python.
Integer division where a fraction was meant — Listing 1.8 compiles without a single warning.

Listing 1.8 From `wrong_average.c`: `1 / 2` is 0, so half of 7 kg becomes 0.00 kg; `kilos / 2.0` gives 3.50 kg.

```c
    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);
```

The cure is to make one side of the division a `double`, or to multiply by `0.5`. Book: Section 1.13.

Summary: Five Things to Remember

1. C is compiled once into an executable; Python is interpreted every run (Figure 1.1). Compile with `gcc -std=c17 -Wall -Wextra -O2`.

2. A C program starts in `main`; statements end with `;`; `return 0;` reports success. Python runs top to bottom, and indentation is grammar.

3. A variable has a name, a type, a value and an address (Figure 1.2); `scanf` needs `&` because it writes into the box.

4. Integer division drops the fraction and `%` keeps the remainder (Equation (1.1)); `*`, `/`, `%` go before `+`, `-`.

5. `if` runs a statement only when its condition is true; `=` assigns, `==` compares. Treat every warning as an error.

Cashier v0.1 reads a weight and prints a receipt; the improved version reads decimals, applies the three-kilogram minimum and rounds to whole rupiah. Book: Section 1.14.

Next Time

Chapter 2 — Structured Program Development. Algorithms, pseudocode, the three control structures, `if`…`else`, and the `while` loop with a counter and with a sentinel.

Cashier v0.2 will serve a whole queue of customers and print an end-of-day summary.

Before then: install the workbench, run `hello` in both languages, and work Exercises 1.2, 1.3, 1.4 and 1.6.

Exercise 1.6, change in as few banknotes as possible, is `/` and `%` at work, and its solution is in the book. Book: Sections 1.15 and 1.16.

Lecture 2 — Structured Program Development

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 2

Lecturer: Irfan Subakti — yifana@gmail.com

Programs are not typed; they are designed. Today: algorithms and pseudocode, the three control structures, `if`…`else` and its chains, the `while` loop with a counter and with a sentinel, top-down refinement, and cashier v0.2. Book: Chapter 2, Sections 2.1–2.14.

What This Meeting Covers

1. **Algorithms and pseudocode** — actions in the right order, written for people.
2. **Three structures are enough** — sequence, selection, repetition; flowcharts and blocks.
3. **Selection** — `if`…`else`, chains, the conditional operator `?:`.
4. **Repetition** — `while` with a counter; with a sentinel, designed by top-down, stepwise refinement; the trace table and the cast.
5. **Putting it together** — nesting, `+=` and `++`, cashier v0.2, and a guard against impossible input.

Every program is again in C and Python with identical output. Book: Chapter 2.

Where We Left Off

Cashier v0.1 prints **one** receipt and stops. The improved version reads decimals and applies a three-kilogram minimum.

At the counter the queue does not stop after one customer. The owner, Pak Hendra, asks at closing time: how many customers, how much money, what was the average bill?

To answer, the program must **repeat** and **choose**, and it must be **designed** before it is typed.

From Lecture 1 we carry `if`, the six comparisons, integer division and `%`. Book: Section 2.1.

Algorithms: Actions in the Right Order

> **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*.

Precise — "add 30 ml of detergent", not "a little". **Finite** — it stops for every input. **Correct** — the right result for every allowed input, including a day with no customers.

Weigh the bag, then put the clothes in the machine. Swap the order and you weigh a wet bag: same actions, wrong order.

Iced tea for a customer: "put a tea bag in a glass, pour water, add sugar, stir" is not correct (cold water), not precise (how much sugar?) and not finite (stir until when?). A computer follows exactly what is written. Book: Section 2.2.

Pseudocode

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
```

Pseudocode 2.1: a name at the top, numbered lines, `←` for assignment, indentation for structure, only executable actions (no declarations).

Pseudocode is never compiled; it is read by people at design time, where changing a line costs seconds. Spend the first ten minutes away from the keyboard. Book: Section 2.3.

Three Structures Are Enough

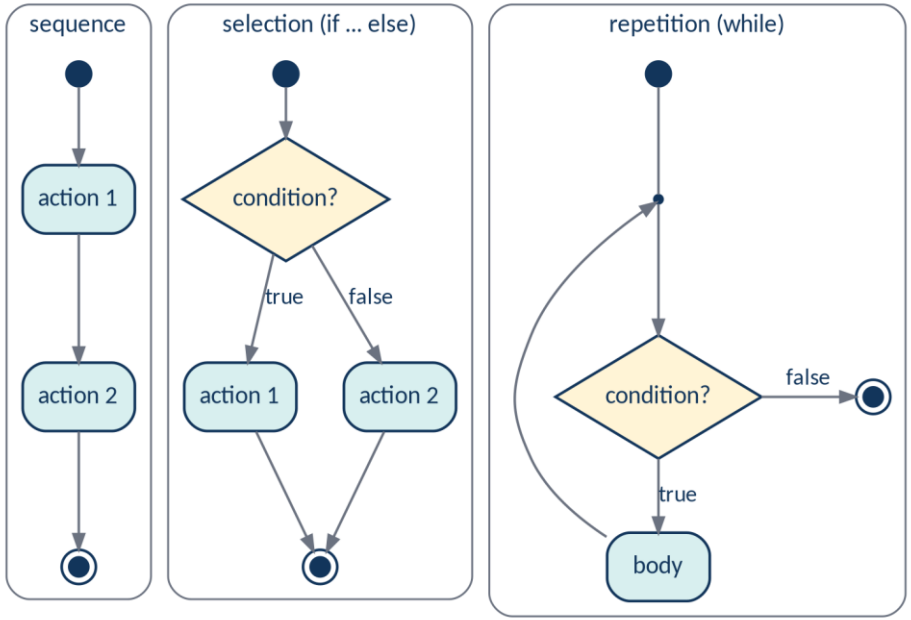


Figure 2.1 The three control structures: sequence, selection and repetition; each has exactly one entry and one exit.

***Figure 2.1**: Böhm and Jacopini (1966) proved that every algorithm can be written with these three, combined by stacking and nesting; Dijkstra's "Go To Statement Considered Harmful" (1968) followed. Book: Section 2.4.1.*

One Entry, One Exit

Sequence — actions one after another. **Selection** — a condition chooses a path: `if`, `if`…`else`, chains, and `switch` in Chapter 3.

Repetition — an action repeats while a condition holds: `while` today; `for` and `do`…`while` in Chapter 3.

Any rectangle can be replaced by a whole structure (**nesting**) or followed by another (**stacking**); the program still reads top to bottom.

A **block**: braces in C, indentation in Python. Without braces, only the *first* statement after `if` or `while` belongs to it.

Flowcharts: a diamond is a decision with exactly two exits, true and false; a rounded rectangle is an action. Put your finger on the entry and follow the arrows. Book: Sections 2.4.2 and 2.4.3.

Chains of Decisions: the Discount

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)
```

***Pseudocode 2.2**: 10% from Rp 100,000, 5% from Rp 50,000, members get five points more. The first true condition wins, so the most demanding test goes first. Book: Section 2.5.2.*

Multiply Before You Divide

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

Equation (2.1): $\lfloor x \rfloor$ is the largest whole number not greater than x. For positive numbers that is C's integer division: `bill * percent / 100`; in Python `bill * percent // 100`.

`bill / 100 * percent` throws away the last two digits of the bill before multiplying.

Worked by hand: $b = 120000$ passes both tests; the 10% test is first, so $p = 10$; a member gets $p = 15$ and $d = 18000$.

Ask: with the 5% test first, what would the Rp 120,000 bill get? Only 5%, and every large bill with it. Book: Section 2.5.2.

The Chain in C

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

```c
    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");
```

*In **Listing 2.1**, `member == 1 ? "member" : "guest"` is the **conditional operator**: an `if`…`else` that produces a value. Python writes the chain with `elif` and the expression as `'member' if member == 1 else 'guest'`. Book: Sections 2.5.2 and 2.5.3.*

The `while` Loop

`while (condition) { statements }` — test first; if false, skip the loop; if true, run the body once and test again.

One run of the body is an **iteration**. Something in the body must eventually make the condition false, or the loop is **infinite**.

Counter-controlled repetition: the number of iterations is known. A counter starts at 1 and runs while it is at most 5.

An **accumulator** such as `total` starts at 0 and gains one value per iteration. Uninitialized, it starts with garbage.

If a program stops responding, press Ctrl+C and look for the statement that should change the loop condition. Book: Sections 2.6.1 and 2.6.2.

Counting to Five

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
```

Pseudocode 2.3 *reads exactly five weights. Line 8 divides two whole numbers, so the average loses its fraction; the sentinel version fixes that with a cast. Book: Section 2.6.2.*

A Harder Problem: How Many Orders?

Pak Hendra wants the average weight of **all** orders of the day. Nobody knows in the morning how many there will be.

A **sentinel value** is a special value that cannot be real data and therefore means “no more data”. No bag weighs 0 kg, so 0 ends the day.

Top-down, stepwise refinement: start from one sentence, the **top**, and split each statement into smaller ones until every line translates directly into C or Python.

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

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

Pseudocode 2.4 *is complete but far too vague to program. Book: Sections 2.7.1 and 2.7.2.*

The First Refinement

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
```

Pseudocode 2.5: three phases that almost every program has.

Refining line 2 reveals a subtle point: the loop must test the weight **before** adding it, so the sentinel is never counted. Read once before the loop, and again at the end of the body.

Refining line 3 reveals another: a day with no orders has $n = 0$, and the mean cannot divide by zero.

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

Equation (2.2) is the ordinary mean we implement. Both key decisions, reading ahead and guarding the division, are found on paper. Book: Section 2.7.3.

The Second Refinement

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."
```

*Every line of **Pseudocode 2.6** now translates directly. Book: Section 2.7.3.*

The Sentinel Loop as a Flowchart

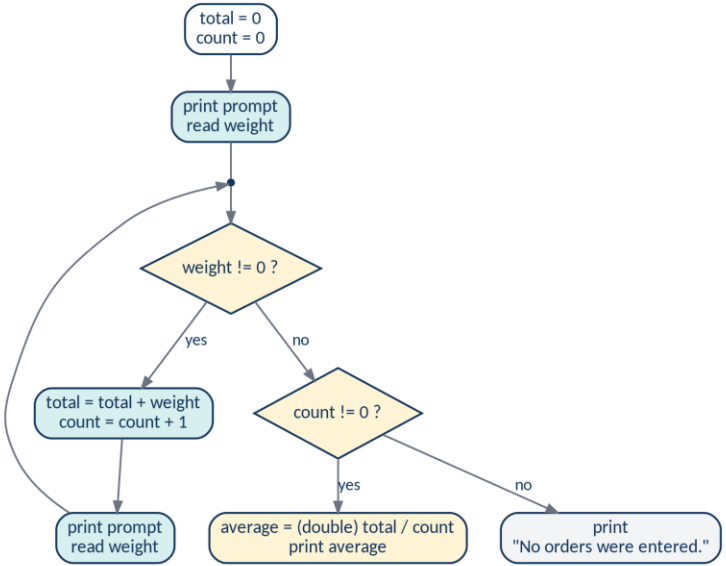


Figure 2.2 The sentinel-controlled average: read before the loop, test at the top, read again at the end of the body, and divide only when there is at least one order.

*Follow **Figure 2.2** with your finger: one read, the diamond, add, count, read, back to the diamond; the second diamond protects the division. Book: Section 2.7.3.*

The Program in C

Listing 2.2 From `average_sentinel.c`: sentinel-controlled repetition, a cast, and a guard against dividing by zero.

```
    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("Average weight: %.2f kg\n", average);
    } else {
        printf("No orders were entered.\n");
    }
```

Listing 2.2 *follows the pseudocode line by line. With 4, 7, 3, 0 it prints* `Orders: 3, total 14 kg` *and* `Average weight: 4.67 kg`*; with 0 at once it prints* `No orders were entered.` *Book: Section 2.7.4.*

Worked Example 1: a Trace Table

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

*Fill **Table 2.1** on the board row by row. The body runs three times, the sentinel is never added, `count` is 3. Two minutes of tracing find most off-by-one errors. Book: Section 2.7.5.*

The Cast

`total` and `count` are `int`s; `total / count` would drop the fraction.

`(double) total` makes a temporary `double` copy; the variable stays an `int`. **Promotion** then converts `count` too, and $14/3 = 4.666\ldots$ survives.

`(double) (total / count)` is too late: the integer division has already lost the fraction, and the cast turns 4 into 4.0. Cast an operand, not a finished result.

Python needs no cast: `/` always gives a fraction, `//` is integer division. Book: Section 2.7.6.

The Same Algorithm in Python

Listing 2.3 From `average_sentinel.py`: the same algorithm in Python.

```
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"Average weight: {average:.2f} kg")
else:
    print("No orders were entered.")
```

*In **Listing 2.3** indentation alone shows which lines belong to the `while`. Book: Section 2.7.6.*

Nested Control Structures

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!"
```

***Pseudocode 2.7**: ten customers answer 1 (satisfied) or 2 (not); more than eight satisfied and Pak Hendra treats the team to lunch. An `if`…`else` sits inside a `while`, and another `if` is stacked after it. Book: Section 2.8.*

Reading a Nested Structure

The `while` runs exactly ten times; inside it, exactly one of the two counters grows on every iteration, so `satisfied + unhappy` is always 10 at the end.

The decision “treat the team” is **stacked after** the loop, not nested in it: it is taken once, when all ten answers are in.

In C the braces show what belongs to the `while`; in Python the indentation alone does. Move the last `if` one level in and it runs ten times.

Worked by hand: answers 1, 1, 2, 1, 1, 1, 1, 1, 1, 1 give 9 satisfied and 1 unhappy, and $9 > 8$, so lunch is on the owner.

Ask the room for an input that gives exactly 8 satisfied: then the condition is false and there is no lunch, a boundary worth testing. Book: Section 2.8.

Compound Assignment and `++`

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

`++c` adds 1 and then uses the new value; `c++` uses the old value and adds 1 afterwards. With `c = 5`: `c++` shows 5, `++c` shows 6; both leave 6.

***Table 2.2**: the variable is named once, so `total = totl + weight` cannot happen. Python has no `++`; `c += 1` is a statement. Never write `c += c++;`. Book: Section 2.9.*

Cashier v0.2: the Design

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)
```

Pseudocode 2.8 *stacks everything of today: a sentinel loop, a chain, accumulators, and a guarded average. Book: Section 2.10.*

Cashier v0.2: the Loop in C

Listing 2.4 From `cashier_v02.c`: the loop of cashier v0.2, with `continue` for an unknown service.

```
    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;
```

*In **Listing 2.4**, `continue` skips the rest of the body and jumps back to the loop test, so no weight is asked for a service that does not exist. Book: Section 2.10.*

Worked Example 2: a Short Day

Input: service 1 with 5 kg, service 2 with 3 kg, a mistyped 7, service 3 with 2 kg, then 0.

By hand: $5 \times 8000 = 40000$; $3 \times 12000 = 36000$; the 7 is rejected and not counted; $2 \times 6000 = 12000$.

Total Rp 88,000 from three customers; average $88000/3 = 29333.33$.

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

Work it on the board first; the program must agree with the hand computation to the last digit. Book: Section 2.10.

Making It Better: Impossible Input

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
```

v0.2 accepts 0 or −3 kg and a negative bill reduces the takings. ***Pseudocode 2.9*** *asks again; 50 kg is a business decision. Chapter 3 gives* `do`…`while` *and* `||` *for exactly this job. Book: Section 2.11.*

Using It for Real

Read a request, check it, handle it by kind, add it to a total: the shape of every point-of-sale program.

A water refill depot: 1 = filtered Rp 6,000, 2 = mineral Rp 8,000, 3 = new gallon Rp 45,000, 0 = close; weight becomes the number of gallons.

Keep money in whole rupiah; choose a sentinel that can never be real data; test the empty day.

0 is safe for a service number, never for a discount amount, which may genuinely be 0. Book: Section 2.12.

Common Mistakes

Uninitialized accumulator; **processing the sentinel**; an **infinite loop** from a missing update.

`(double) (total / count)` — too late; **dividing by zero** without `if (count != 0)`.

`while (weight != 0);` — a stray semicolon; the **wrong order** in an `else if` chain; **indentation without braces** in C.

A sentinel that can be real data; `==` **on fractions**; **testing only the happy path**: try empty, single, typical, and each boundary.

Boundaries for the discount: exactly Rp 50,000 and Rp 100,000. Book: Section 2.13.

Summary: Five Things to Remember

1. An algorithm is precise, finite and correct; pseudocode writes it for people with `←` and indentation.

2. Sequence, selection and repetition, each with one entry and one exit, stacked and nested, build every program (**Figure 2.1**).

3. In a chain the first true condition wins; `?:` chooses between two values.

4. A sentinel loop reads before the loop and at the end of the body, and guards the division (**Figure 2.2**); stepwise refinement finds these decisions on paper.

5. Cast an operand, `(double) total / count`, not the result; `+=` and `++` abbreviate updates.

Cashier v0.2 serves customers until 0, rejects unknown services with `continue`, and prints customers, takings and the average bill. Book: Section 2.14.

Next Time

Chapter 3 — Program Control. `for`, `do`…`while`, `switch`, `break` and `continue`, and the logical operators `&&`, `||` and `!`. Cashier v0.3 validates every weight, chooses services with `switch`, and prints a formatted daily report.

Before then: Exercises 2.1, 2.2, 2.5 and 2.6.

Exercise 2.5 is a trace table for the input 5, 0: do it by hand before running anything. Book: Sections 2.15 and 2.16.

Lecture 3 — Program Control: for, do…while, switch and Logical Operators

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 3

Lecturer: Irfan Subakti — yifana@gmail.com

Today completes the toolbox of control structures: `for`, `do…while`, `switch`, `break` and `continue`, and the logical operators, each beside its Python form, ending with cashier v0.3 and a price list for the wall. Book: Chapter 3, Sections 3.1–3.13.

What This Meeting Covers

1. `for` — a counting loop in one line; Python's `for` and `range`; compound interest and field widths.

2. `do`…`while` — at least once; "ask until the answer is valid"; Python's `while True` with `break`.

3. `switch` — one decision, many cases; fall-through; Python's `match`.

4. `break`, `continue`, `&&`, `||`, `!` — short-circuit evaluation, De Morgan's laws, business rules as conditions.

5. Choosing the structure — and cashier v0.3 with a formatted daily report.

After today you know every control structure of C; Chapter 4 puts them all to work on five complete problems. Book: Chapter 3.

Where We Left Off

Cashier v0.2 serves a queue until the sentinel 0 and prints customers, takings and the average bill.

Its weak spot: Mbak Rina typed 60 instead of 6, and the takings jumped by Rp 432,000 that nobody paid. The program said nothing.

Pak Hendra wants more: counts per service, the largest bill, a table he can photograph, and a savings plan for a new washing machine.

Each wish maps to one tool: counting to 10 is `for`; asking until sensible is `do`…`while`; choosing among services is `switch`; "at least 1 and at most 50" is a logical operator. Book: Section 3.1.

The `for` Statement

```
for ( int i = 1 ; i <= 5 ; ++i )
        (1)       (2)    (4)
    printf("%d ", i);   (3)
```

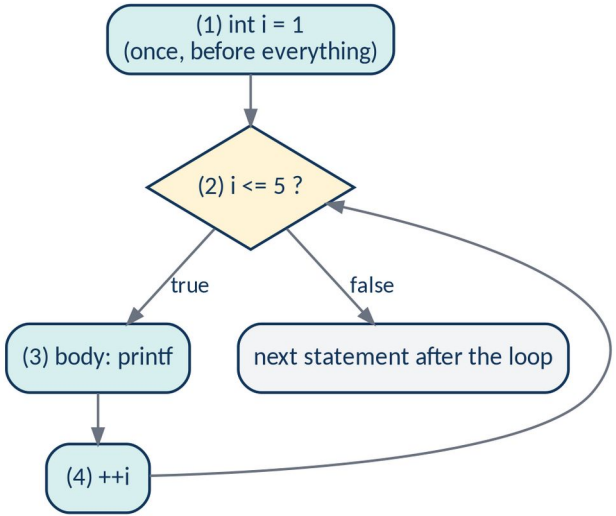


Figure 3.1 The anatomy of a `for` statement: the initialization (1) runs once; then the condition (2), the body (3) and the increment (4) repeat until the condition is false.

Figure 3.1: the order is 1, 2, 3, 4, 2, 3, 4, 2, … until the test in step 2 fails. The three pieces of Chapter 2's counter loop now sit in one header, where none can be forgotten. Book: Section 3.2.1.

`for` Is a `while` in One Line

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

`for (int i = 1; i <= 5; ++i)` counts up; `for (int i = 5; i >= 1; --i)` counts down; `i += 7` counts in steps.

A counter declared in the header exists only inside the loop.

Python: `for i in range(start, stop, step)`; `stop` is **excluded**, so `range(1, 11)` gives 1 to 10.

Prefer `for` whenever a counter controls the loop: the reader sees where it starts, stops and how it moves. Book: Sections 3.2.1 and 3.2.2.

Compound Interest

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

Equation (3.1): principal p, yearly rate r as a fraction, n years. After one year $p + p \cdot r = p(1+r)$; each further year multiplies by $(1+r)$ again.

For $p = 1000000$ and $r = 0.05$: 1,050,000 after one year and 1,102,500 after two, since $1.05^2 = 1.1025$.

The rate is only an example for the arithmetic; real rates and deposit terms come from the bank. Ask the room to compute year two by hand before the program does. Book: Section 3.3.1.

Worked Example 1: the Savings Table

Listing 3.1 From `savings.c`: a deposit growing at compound interest, printed as an aligned table.

```
    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 */
    }
```

```
Year    Amount on deposit
   1           1050000.00
   2           1102500.00
   8           1477455.44
   9           1551328.22
  10           1628894.63
```

Listing 3.1 uses `pow` from `math.h`; on Linux link with `-lm`. Rows 3 to 7 are left out here. Book: Section 3.3.2.

Field Widths

`%4d` — an integer right-aligned in 4 characters, padded with spaces on the left.

`%21.2f` — a `double` with 2 decimals, right-aligned in 21 characters. `%4s` and `%21s` put the headings exactly above their columns.

`%-18s` — a minus sign aligns to the left; the labels of cashier v0.3's report use it.

Python uses the same numbers in f-strings: `{year:4d}`, `{amount:21.2f}`, with `>` for right and `<` for left.

Field widths turn a column of numbers into a table the owner can photograph and send. Book: Section 3.3.3.

How Many Years to the Target?

The machine costs Rp 1,500,000. After eight years the deposit holds Rp 1,477,455.44, not enough; after nine, Rp 1,551,328.22.

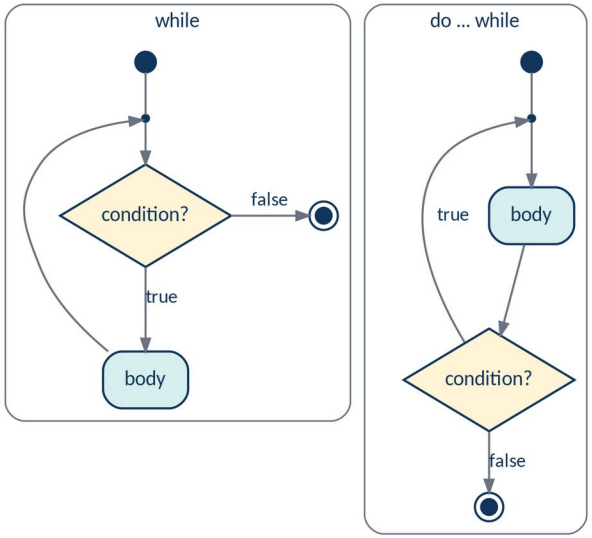
"After how many years?" is not a counting loop: the number of years is the *answer*, not known in advance.

The natural form is `while`: multiply the amount by $1+r$ and count one year *while* the amount is still below the target.

The rule for choosing: do we know the number of repetitions before the loop starts? Yes → `for`. No → `while`.

Nine years is far too slow for a machine that already leaks, which is the owner's real conclusion from the table. Book: Section 3.3.4.

`do`...`while`: At Least Once



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.

Figure 3.2: we cannot check a weight until we have read one, so the test belongs after the body. Note the semicolon after `while (condition);`*, which ends the* `do` *statement. Book: Section 3.4.1.*

Ask Until the Answer Is Valid

Listing 3.2 From `validate.c`: `do`…`while` keeps asking until the weight is in range.

```
    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 */
```

Answers 0, 75, 12: the first two print the message and ask again; 12 makes the condition false and is accepted. The body ran three times.

With a good first answer the body runs once, never zero times. Listing 3.2 is exactly the guard Chapter 2 asked for. Book: Section 3.4.2.

Python: `while True` with `break`

Listing 3.3 From `validate.py`: Python's form of "ask until valid" is `while True` with `break`.

```
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
```

Python may write `1 <= weight <= 50`, which reads like mathematics.

In C, `1 <= weight <= 50` means `(1 <= weight) <= 50`: it compares 0 or 1 with 50 and is **always true**.

Listing 3.3 puts the test where it belongs, after the input; `break` is the only way out. Book: Section 3.4.2.

`switch`: One Decision, Many Cases

Listing 3.4 From `service_count.c`: `switch` with shared cases, `break` and `default`.

```c
        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;
        }
```

In Listing 3.4 the expression is evaluated once, control jumps to the matching `case` or to `default`, and runs until a `break`. The loop around it reads codes with `scanf(" %c", &code)`; the space skips blanks. Book: Section 3.5.1.

Falling Through

Once control jumps to a label, it continues downward through the following labels until a `break`: **fall-through**.

Stacked labels `case 'W': case 'w':` use it on purpose. A *forgotten* `break` uses it by accident; `-Wextra` warns.

`switch` tests only integer values (`int`, `char`, enumerations): no ranges, no fractions, no strings. Use `if`…`else if` for those.

Why `switch` over a chain: it names the tested variable once and lists the cases like a menu.

The chain `if (code == 'W' || code == 'w') ++wash; else if ...` does the same work; the reader must check every condition to know that each one tests `code`. Book: Sections 3.5.2 and 3.5.3.

Python's `match`

Listing 3.5 From `service_count.py`: `match`, where `case "W" | "w"` lists two alternatives and `case _` catches everything else.

```
        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
```

Listing 3.5: Python 3.10 and later. `|` replaces stacked labels, `case _` is `default`, and there is no fall-through and no `break`; a `break` inside a `match` leaves the enclosing loop. Book: Section 3.5.4.

`break` and `continue`

`break` ends the loop at once. `continue` skips the rest of the current iteration; in a `for` the increment still runs.

Breaking at 5 prints `1 2 3 4`; continuing at 5 prints `1 2 3 4 6 7 8 9 10`.

Several `break`s mean several exits: the one-entry, one-exit property is lost. Use `break` for clear cases, such as leaving `while True`.

`break` leaves only the **innermost** loop or `switch`. To stop two nested loops, use a flag such as `found`.

The book's `break_continue.c` and its Python twin print the same two lines. Book: Section 3.6.

Logical Operators

Table 3.1 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

Table 3.1: `&&` needs both, `||` needs at least one, `!` flips. Python writes `and`, `or`, `not`.

Precedence: `!` first, then comparisons, then `&&`, and `||` last. `a || b && c` means `a || (b && c)`; write the parentheses anyway. Book: Section 3.7.1.

Short-Circuit Evaluation

Listing 3.6 From `logic.c`: a division protected by short-circuit evaluation.

```c
    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");
    }
```

`&&` stops when its left side is false; `||` stops when its left side is true. The right side is **not evaluated at all**.

Reversed, `kilos / orders > 5 && orders != 0` divides by zero first. Order the parts so that each protects the next.

Listing 3.6 prints `No verdict: no orders yet.` Python short-circuits too; its `and` and `or` return one of their operands, so the book wraps them in `int(...)`. Book: Section 3.7.2.

De Morgan's Laws

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

Valid weight: `weight >= 1 && weight <= 50`. By Equation (3.2), "not valid" is `!(weight >= 1) || !(weight <= 50)`, that is `weight < 1 || weight > 50`.

The wrong guess `weight < 1 && weight > 50` is never true: no weight is both.

When you must negate a condition, apply the laws on paper instead of guessing. Book: Section 3.7.3.

Worked Example 2: Rules into Conditions

"Express only until 10 in the morning, except for members." → `service != 2 || hour < 10 || member == 1`.

"Free fragrance for at least 5 kg that is not ironing only." → `weight >= 5 && service != 3`.

"Close when the cashier types 0, or past 21:00." The loop *continues* while `!(service == 0 || hour >= 21)`, which by De Morgan is `service != 0 && hour < 21`.

Read each condition back aloud; test it on a small table, one row per combination: four rows for two parts, eight for three.

Work the first rule on the board: a non-express order is always accepted, an express order early in the day, a member's express order any time. Book: Section 3.7.4.

Choosing the Right Structure

Table 3.2 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	`for`	`for … in range(…)`
repeat while a condition holds, possibly zero times	`while`	`while`
at least once, then decide	`do`…`while`	`while True` … `break`
one of two paths	`if`…`else`	`if` … `else`
many paths by ranges or complex conditions	`if`…`else if` chain	`if` … `elif`
many paths by one integer or character value	`switch`	`match`
a choice between two values inside an expression	`?:`	`x if c else y`

Table 3.2: receipt numbers 1 to 40 → `for`; bags until 0 → `while`; a phone number until 10 to 13 digits → `do`…`while`; a menu number → `switch`. The structure that says what you mean is the cheapest documentation. Book: Section 3.8.

Cashier v0.3: the Design

Main loop: `do`…`while`, because the cashier is asked for a service at least once.

`switch` on the service: `case 0: break;` leaves only the `switch`, and the loop ends because `service != 0` is false.

`case 1: case 2: case 3:` share one block by fall-through; a nested `do`…`while` asks for the weight until 1 to 50 kg.

The largest bill: `largest` starts at 0, and every larger bill replaces it.

In Python the main loop is `while True` with `break`, the `switch` becomes `match` with `case 1 | 2 | 3`, and the weight check uses `1 <= weight_kg <= 50`. Book: Section 3.9.

Cashier v0.3: a Day with Mistakes

Input: wash 5 kg; express typed as 0, then 60, then 3 kg; unknown service 4; ironing 2 kg; wash 7 kg; 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
```

Check by hand: 40,000 + 56,000 + 36,000 + 12,000 = 144,000 from four customers; average 36,000; largest 56,000. The 0 and 60 were refused, the 4 reported and not counted. Book: Section 3.9.

Making It Better: a Price List

```
kg      Wash   Express   Ironing
 1      8000     12000      6000
 2     16000     24000     12000
 3     24000     36000     18000
10     80000    120000     60000
```

A table is a loop inside a loop: the outer `for` over kilograms, the inner `for` over the three services; the inner body runs $10 \times 3 = 30$ times.

The price comes from a nested conditional: `service == 1 ? 8000 : service == 2 ? 12000 : 6000`.

Rows 4 to 9 are left out here. Every entry must equal what cashier v0.3 prints for the same weight: two programs computing the same thing check each other. Book: Section 3.10.

Using It for Real

"Ask until valid" belongs wherever people type numbers: an age from 0 to 120, a month from 1 to 12. Take the limits from the business.

A letter typed where a number is expected still ends the program (Chapter 9 fixes that); the report is forgotten at the end (Chapter 11 saves it).

Until then: `./cashier_v03 > report.txt` saves the output of any program to a file.

50 kg is right for this laundry and wrong for a rice wholesaler. Book: Section 3.11.

Common Mistakes

`for (...);` — an empty body; **off-by-one** limits: `i < 10` runs 9 times, and `range(1, 10)` stops at 9.

A missing `break` in a `switch`; **no** `;` after `do`…`while`; `1 <= x <= 50` in C.

`&` **or** `|` instead of `&&` or `||`; **negating without De Morgan**; expecting `break` to leave an `if`.

Changing the counter inside a `for` body; `case 2.5:` or `case "W":`; **forgetting** `-lm` (`undefined reference to 'pow'`).

Count with integers; compute fractions from the counter inside the body. Book: Section 3.12.

Summary: Five Things to Remember

1. `for` gathers initialization, condition and increment in one header (Figure 3.1); Python's `range` excludes its stop.

2. `do`…`while` runs at least once (Figure 3.2) and is the loop for "ask until valid"; Python writes `while True` with `break`.

3. `switch` jumps to the matching `case` and falls through until `break`; Python's `match` has `|`, `case _` and no fall-through.

4. `&&`, `||` and `!` short-circuit, so the left part can protect the right; De Morgan's laws (Equation (3.2)) negate correctly.

5. Choose the structure that says what you mean (Table 3.2); field widths such as `%4d` and `%-18s` align reports.

Cashier v0.3 combines them all: a `do`…`while` main loop, a `switch`, a validated weight, per-service counters, the largest bill, and an aligned report. Book: Section 3.13.

Next Time

Chapter 4 — Evaluation and Discussion I. Five complete problems, each solved from nothing: plane figures, a guessing game, 100 to the power *n*, rock-paper-scissors, and a battle against three goblins.

No new statement: every tool from Chapters 1 to 3, used with a method.

Before then: Exercises 3.2, 3.3, 3.4 and 3.6.

Exercise 3.4 is De Morgan's laws in practice; do it on paper first. Book: Sections 3.14 and 3.15.

Lecture 4 — Evaluation and Discussion I: Five Problems Solved Step by Step

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 4

Lecturer: Irfan Subakti — yifana@gmail.com

No new statement today. We take the tools of Chapters 1 to 3 and attack five complete problems with one method, and for each we take apart a tempting wrong solution. Book: Chapter 4, Sections 4.1–4.11.

What This Meeting Covers

0. The method — seven steps from a short statement to a tested program.

1. Areas of plane figures — a pure sequence with a guard. **2. The guessing game** — a sentinel loop in disguise.

3. 100 to the power n — the limits of numbers in a computer.

4. Rock, paper, scissors — validation and a compact rule. **5. The goblin battle** — state variables, one loop, careful order.

6. Side by side — the five traps, and where the same shapes appear in real software.

Each problem goes the same way: restate, work by hand, design on paper, C, Python with identical output, a tempting wrong solution, and a check-yourself variant. Book: Chapter 4.

Where We Left Off: the Toolkit

Chapter 1 — `gcc -std=c17 -Wall -Wextra -O2`; name, type, value, address; `int` ends at 2,147,483,647; integer division and `%`; `if`.

Chapter 2 — pseudocode and stepwise refinement; counter and sentinel `while` loops, reading ahead; guarded division; `+=`, `++`, `(double)`.

Chapter 3 — `for`, `do`…`while`, `switch` and `match`, `break`, `continue`, `&&`, `||`, `!`, short-circuit, De Morgan.

That is enough for all five problems. Functions, arrays and strings arrive in Chapters 5, 6 and 9, and several of these programs become shorter then; today we solve them the long way and understand every line. Book: Section 4.1.

Seven Steps

1. **Restate** in your own words: what comes in, what goes out, every decision the statement leaves open. **2. Work examples by hand**, including an unusual one.
3. **Design on paper**: structures and pseudocode. **4. Trace** the design on a hand example.
5. **Write** from the pseudocode with warnings on. **6. Test** against the hand examples, character for character.
7. **Attack your own program**: the boundary, the invalid value, the loop that runs zero times or forever.

> **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.

The seven steps are a programmer's version of Pólya: understand, plan, carry out, look back. Code lives in `Chapter04/c`, `python`, *and* `python/tempting` *for the wrong versions. Book: Section 4.2.*

Problem 4.1: Areas of Plane Figures

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.

Our decisions: one prompt per value; the trapezoid shares `height`; two decimals; any negative length prints a message and no areas; 0 is allowed.

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

By hand with Equation (4.1): $b = 10$, $h = 5$ gives 25; $b_1 = 8$, $b_2 = 4$ gives 30; $s = 3$ gives 9; $r = 2$ gives $4\pi \approx 12.57$. A circle of radius 1 printed as 3.00 betrays whole-number arithmetic. Book: Sections 4.3.1 and 4.3.2.

4.1: Design and the Trap

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
```

Listing 4.1 From `geometry_wrong.c`: `1 / 2` is evaluated first, as integer division.

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

Pseudocode 4.1 *is a pure sequence with one guard; C17 has no π, so we define a* `const double`*.* ***Listing 4.1*** *prints* `Triangle area = 0.00` *for every triangle, without a warning. Cure:* `0.5`*,* `1.0 / 2`*, or* `base * height / 2`*. Book: Sections 4.3.3–4.3.6.*

Problem 4.2: the Guessing Game

> **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.

Messages exactly as given; the congratulation reports the secret and the tries, “1 try” or “3 tries”.

By hand, secret 37: 50 too high, 25 too low, 37 right after 3 tries. The unusual case: right at the **first** guess, only the congratulation.

A clever player halving the range needs at most 7 guesses for 1 to 100, but the program does not care how the player guesses. Book: Sections 4.4.1 and 4.4.2.

4.2: a Sentinel Loop in Disguise

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
```

*In **Pseudocode 4.2** the sentinel is not a fixed value but the answer itself. Inside the loop the guess is already known to be wrong, so an `if`…`else` suffices. Book: Section 4.4.3.*

4.2: the Loop in C

Listing 4.2 From `guess.c`: the guessing game as a read-ahead `while` loop.

```
    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");
```

The trap: a `do`…`while` "read, judge, repeat until right" judges the winning guess too, and the `else` prints "too high" for "equal".

*In **Listing 4.2** the body only ever sees wrong guesses. The wrong version works on every wrong guess and fails on the one that matters most. Book: Sections 4.4.4 and 4.4.6.*

> **CHECK YOURSELF.** Limit the game to 5 tries. *Answer:* loop while `guess != answer && tries < 5`; after the loop test `guess == answer`, not `tries == 5`, because a hit on the fifth try also has `tries == 5`.

Problem 4.3: 100 to the Power n

> **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: $100^5 = 10^{10}$ is more than four times the largest `int`.

Our decisions: print 100^n **exactly** for $-100000 \le n \le 100000$; a negative n prints the exact decimal fraction; report whether the value fits `int` and `long long`.

The problem asks whether you notice the limit, and what you do about it. Book: Section 4.5.1.

4.3: Print, Don't Compute

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

By Equation (4.2), 100^n is a 1 followed by $2n$ zeros: 100^3 is a 1 and six zeros. For $n = -m$: "0.", then $2m - 1$ zeros, then 1; $100^{-2} = 0.0001$.

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`?	Fits in `long long`?
4	10^8	yes	yes
5	10^{10}	no	yes
9	10^{18}	no	yes
10	10^{20}	no	no

***Table 4.1**: `int` holds up to about 2.1×10^9, `long long` up to about 9.2×10^{18}. Book: Section 4.5.2.*

4.3: Design and Program

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"
```

Listing 4.3 From `power100.c`: 100 to the power n, printed digit by digit, never overflowing.

```
    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");
    }
```

Pseudocode 4.3 *uses* `for` *because the number of zeros is known before the loop starts;* ***Listing 4.3*** *follows it line by line. Book: Sections 4.5.3 and 4.5.4.*

4.3: the `pow` Trap

Listing 4.4 From `power100_wrong.c`: `pow` returns a `double`, which has only about 16 significant digits.

```
    printf("100^%d = %.0f\n", n, pow(100.0, n));
```

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

For $n = 3$ the answer is right, and a tester who tries only small values believes it. 10^{24} has no exact `double`; `%.0f` faithfully prints the nearest one.

> **CHECK YOURSELF.** Print 1000^n exactly. *Answer:* $1000^n = 10^{3n}$, so $3n$ zeros; a `long long` holds $1000^6 = 10^{18}$ but not 1000^7, so $n \leq 6$.

***Listing 4.4**: Python's `100.0 ** n` prints the same wrong digits, because it uses the same `double`. Book: Section 4.5.6.*

Problem 4.4: Rock, Paper, Scissors

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.

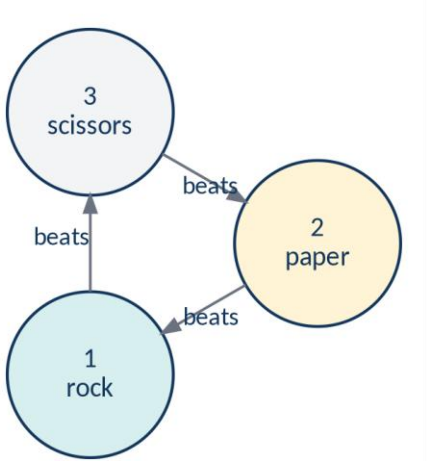


Figure 4.1 The rule of rock, paper, scissors as a cycle: every object beats the next one along the arrows.

Figure 4.1*: nine combinations, three verdicts. Going around the cycle, the winner's number is always one more than the loser's, modulo 3. Book: Sections 4.6.1 and 4.6.2.*

4.4: One Remainder Decides

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

Table 4.2 The combinations checked against Equation (4.3) (the two other draws also give remainder 0): 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
1 rock	1 rock	3	0	draw
1 rock	2 paper	2	2	player 2
1 rock	3 scissors	1	1	player 1
2 paper	1 rock	4	1	player 1
2 paper	3 scissors	2	2	player 2
3 scissors	1 rock	5	2	player 2
3 scissors	2 paper	4	1	player 1

*A formula must be checked, not trusted: **Table 4.2** checks Equation (4.3) case by case. The `+ 3` keeps the left side positive, because C's `%` keeps the sign of the dividend. Book: Section 4.6.2.*

4.4: the Verdict in C

Listing 4.5 From `rps.c`: the winner found with one remainder.

```
    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");
```

Design: two `do`…`while` loops ask each player until 1, 2 or 3; a `switch` prints the names; the verdict is **Listing 4.5**.

The trap: without `+ 3`, rock against scissors gives $1 - 3 = -2$ and `-2 % 3` is -2 in C, so scissors "beat" rock.

Only the combination where the winner has the smaller number fails, which is why a quick test with paper against rock passes. In Python `(p1 - p2) % 3` is 1 and correct. Book: Sections 4.6.3–4.6.6.

Problem 4.5: the Goblin Battle

> **PROBLEM 4.5.** A boy fights three goblins, turn by turn. The boy has 200 HP and an attack of 10; every third turn his skill deals 20. Each goblin has 165 HP; their attacks are 3, 4 and 1. The boy attacks first, then the goblins. Print the battle until one side is wiped out, then the number of turns and the winner's remaining HP.

Our decisions: the boy attacks the first living goblin; only goblins alive **after** his attack hit back; HP never goes below 0; one table line per turn.

Each open decision changes the result, so each is written down. Book: Section 4.7.1.

4.5: Predict the Outcome by Hand

Turns 1, 2, 3: goblin 1 goes $165 \to 155 \to 145 \to 125$; the boy $200 \to 192 \to 184 \to 176$. Every three turns goblin 1 loses 40, the boy 24.

Goblin 1 falls in turn 13, goblin 2 in turn 26, goblin 3 in turn 39.

The boy takes $8 \times 12 + 5 \times 13 + 1 \times 13 = 96 + 65 + 13 = 174$ and wins after 39 turns with $200 - 174 = 26$ HP.

Work this on the board: a prediction made before the program exists is the best possible test. Book: Section 4.7.2.

4.5: One Turn

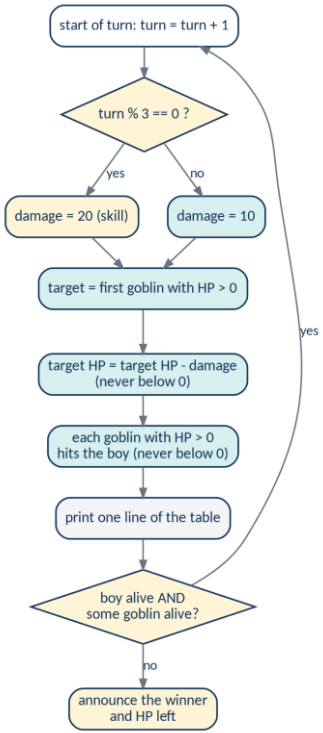

start of turn: turn = turn + 1
turn % 3 == 0 ?
yes
no
damage = 20 (skill)
damage = 10
target = first goblin with HP > 0
target HP = target HP - damage
(never below 0)
each goblin with HP > 0
hits the boy (never below 0)
print one line of the table
boy alive AND
some goblin alive?
yes
no
announce the winner
and HP left

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.

Figure 4.2*: the state is five numbers, the boy's HP, three goblins' HP and the turn. The loop continues while* `boy > 0 && (g1 > 0 || g2 > 0 || g3 > 0)`*, parentheses required. Book: Section 4.7.3.*

4.5: the Design

Pseudocode 4.4 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
```

***Pseudocode 4.4**: $[g_1 > 0]$ is 1 when the goblin lives and 0 otherwise. One iteration is one complete turn. Book: Section 4.7.3.*

4.5: Guarded Updates in C

Listing 4.6 From `battle.c`: only living goblins hit back, and HP never drops below 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;
```

The output ends with `The boy wins after 39 turns with 26 HP left.`, as predicted.

The trap: no "alive" checks. Goblin 1 sits at −5 HP and keeps hitting; the boy falls in turn 25 and the goblins win. Two missing guards flip the verdict.

*In **Listing 4.6** each hit uses the same guarded form, `g1 = g1 > damage ? g1 - damage : 0`. In the wrong version even `g1 + g2 + g3 > 0` fails: with negative HP, a positive sum does not mean anyone is alive. Book: Sections 4.7.4 and 4.7.6.*

> **CHECK YOURSELF.** The smallest starting HP with which the boy still wins? *Answer:* he loses exactly 174 HP in the whole battle, so 175: he wins after 39 turns with 1 HP left.

The Problems Side by Side

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	`1 / 2` is integer division
4.2 Guessing game	read-ahead `while`	a `do`…`while` judges the winning guess
4.3 100^n	`if`…`else`, `for` over digits	`pow` returns a `double`
4.4 Rock, paper, scissors	`do`…`while`, `switch`, one remainder	without `+ 3`, `%` is negative
4.5 Goblin battle	state variables, `while` with AND and OR	no “alive” checks

Table 4.3*: none of the traps is exotic. Integer division, judging after the fact, the limits of a type, the sign of a remainder and a missing guard appear in real programs every day. Book: Section 4.8.*

Using It for Real

4.1 is every calculator page: a tile shop computing boxes for a floor. 4.2 is every retry loop: a PIN pad with a limit of tries.

4.3: money in whole rupiah fits comfortably in a `long long`; know the limits before choosing a type.

4.5 is a simulation: three machines of 60 minutes per load, a bag every 15 minutes, one iteration per minute. How long is the queue by noon?

With the tools of Chapters 1 to 3 the laundry simulation can already be written. Book: Section 4.9.

Common Mistakes

Deciding details silently — write them at the top of the program. **Testing only the typical case**.

Mismatched output text — a missing exclamation mark fails an automatic comparison.

Trusting a formula without checking every case; **overflow the compiler cannot see** — run with `-fsanitize=undefined`.

Printing in the wrong branch; **half-validated input** — check player 2 as carefully as player 1.

Every tempting solution passed an ordinary test and failed an unusual one. Book: Section 4.10.

Summary: Five Things to Remember

1. Seven steps: restate, examples by hand, design, trace, write, test, attack. Most of the time goes before the code.

2. Formulas such as Equation (4.1) need `double` constants: `0.5`, never `1 / 2`.

3. A read-ahead `while` shows the body only wrong guesses; a `do`…`while` judges before testing.

4. Types have limits (**Table 4.1**); printing the digits of 10^{2n} is exact, `pow` is rounded.

5. Check a compact rule in every case (**Table 4.2**); guard every state update (**Figure 4.2**).

The five tempting wrong solutions are the most useful pages of the chapter: each shows a test that separates right from almost right. Book: Section 4.11.

Next Time

Chapter 5 — Functions: Divide, Name and Conquer. Defining and calling functions, prototypes, pass-by-value, the standard library, scope, repeatable randomness and recursion.

Cashier version 4 is split into a header, a file of business rules, a short `main` and a test program.

Before then: rework Problem 4.5 with goblin 2's attack set to 6, predict the result by hand, then run it.

The book's answer: the goblins win after 24 turns with 180 HP left in total. Book: Sections 4.12 and 4.13.

Lecture 5 — Functions: Divide, Name and Conquer

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 5

Lecturer: Irfan Subakti — yifana@gmail.com

Today we cut programs into named pieces: defining and calling functions, prototypes, pass-by-value, the standard library, scope and lifetime, repeatable randomness, recursion, and cashier version 4 split into a module with its own tests. Book: Chapter 5, Sections 5.1–5.14.

What This Meeting Covers

1. **The idea** — a function is a named piece of work; what happens during a call; why small pieces win.
2. **Writing functions** — one rule written once, prototypes, `void`, pass-by-value, Python's view.
3. **Borrowing and hiding** — the standard library and `-lm`; scope and lifetime; a random generator with hidden `static` state.
4. **Recursion** — base case and smaller problem, the call stack, rupiah with dots, the price of Fibonacci.
5. **Cashier v4** — a header, a file of rules, a short `main`, and a test program.

Every program again in C and Python, with identical output. Book: Chapter 5.

Where We Left Off

Chapter 4 solved five problems, each inside one long `main`. The cashier grew the same way.

Pak Hendra changed the price list: wash and fold Rp 7,000 per kg, a new wash and iron Rp 10,000, iron only Rp 6,000, express +50%, minimum 2 kg.

Dimas searched for `8000` and changed it in the bill, but the same number also sat in the daily report. Four days of wrong reports followed.

A recipe book writes the sambal once, on page 12, and every dish points there. A program needs the same habit.

You have used functions since the first day: `printf` and `scanf` were written by someone else, and `main` is yours. Today you write the others. Book: Section 5.1.

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 is *defining* it; using it by its name followed by parentheses is *calling* it. The values in the parentheses of a call are the *arguments*.

```
long   bill_for   (int price_per_kg, int kg)   { ... }
^^^^   ^^^^^^^^    ^^^^^^^^^^^^^^^^^^^^^^^^^   ^^^^^^^
type   name        parameters                   body
```

A small machine with a label: the label names it and says what goes in. `void` means nothing comes out; `(void)` means nothing goes in, as in `main`. Book: Section 5.2.1.

What Happens During a Call

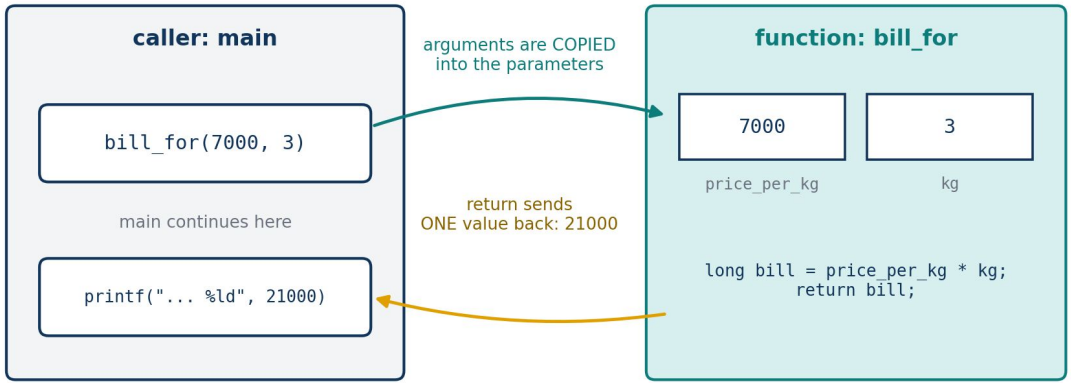


Figure 5.1 A call of `bill_for(7000, 3)`: the arguments are copied into the parameters, the body runs, and `return` sends one value back to the place of the call, where `main` continues.

*Figure 5.1 in four steps: evaluate the arguments; create the parameters and **copy** each argument in; run the body to `return`; send 21000 back, destroy the parameters and locals, and continue in the caller. Book: Section 5.2.2.*

Why Small Pieces Win

One rule, one place — when the price changes, one line changes.

A name explains — `charged_kg(weight)` says what it does; a `main` built from well-named calls reads like the story of the program.

A piece can be tested alone — call it with numbers worked out by hand. **A piece can be reused** — the rupiah printer serves receipts, reports and price lists.

*Cutting a big problem into smaller ones is **divide and conquer**, the stepwise refinement of Wirth applied to code. Book: Section 5.2.3.*

One Rule, Written Once

Listing 5.1 From `first_function.c`: one bill rule, defined once above `main` and called four times.

```c
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;
}
```

Listing 5.1 prints Rp 21000, Rp 50000 and Rp 40000. The cast `(long)` makes the product a `long` before it can overflow. In Python the rule is `def bill_for(price_per_kg, kg): return price_per_kg * kg`. Book: Section 5.3.1.

Prototypes: a Promise Before the Call

Listing 5.2 From `prototypes.c`: four prototypes at the top of the file, before `main`.

```c
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);
```

A **prototype** is the header followed by `;`: name, parameter types, return type. With it, `main` can come first, as the table of contents.

`print_rule` is `void`: it exists for what it does. `max3` is built from `max2`: functions may call functions.

> **WARNING:** Since C99 a call without a prototype is not allowed, and `gcc` reports *implicit declaration of function*. Always give a prototype, or define the function before its first use.

Listing 5.2 is the top of the file; the definitions come after `main`. Book: Section 5.3.2.

Worked Example 1: Pass-by-Value

`main` has `bill = 30000` and calls `with_fee = add_express_fee(bill);`.

The function receives a **copy**, adds half of it and prints its parameter: 45000.

Back in `main`, `bill` is still 30000. The new amount arrives only through the return value, stored in `with_fee`.

```
bill in function : 45000
bill in main     : 30000
with express fee : 45000
```

Work it on the board with two boxes named `bill`, one per function. `add_express_fee(bill);` alone changes nothing; the result is thrown away. Book: Section 5.3.2.

Python's View

Python has no prototypes: a name must exist **when it is called**, not when it is read.

So `main` may be defined first and called at the bottom of the file, after all the definitions.

Parameters behave the same way: rebinding a parameter inside the function does not change the caller's variable.

The book's `prototypes.py` prints the same eight lines as the C program. Book: Section 5.3.3.

Borrowing Functions: the Standard Library

`<stdio.h>` — `printf`, `scanf`, `putchar`. `<math.h>` — `sqrt`, `pow`, `ceil`, `floor`, `round`, `fabs` (link with `-lm` on Linux). `<stdlib.h>`, `<string.h>`, `<time.h>` and others: tested functions, grouped by subject, declared in headers of prototypes. The laundry charges **every started kilogram**, at least 2:

$$k = \max(2, \lceil w \rceil) \tag{5.1}$$

By Equation (5.1): 3.4 kg is charged as 4, 0.8 kg as 2, exactly 2.0 kg as 2. `ceil` *is the ceiling* $\lceil w \rceil$*. Python's* `round(2.5)` *is 2 and C's is 3.0: name the rounding rule you mean. Book: Section 5.4.*

`floor`, `round` and `ceil`

```
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
  7.25     7.0     7.0     8.0      8 kg
```

Row 2.10: `floor` and `round` give 2, but a third kilogram was started, so `ceil` gives 3 and the charge is 3 kg.

Row 0.80: `ceil` gives 1, and `charged_kg` raises it to the minimum of 2.

Python's built-in `round` rounds halves to the **even** neighbor (banker's rounding): `round(2.5)` is 2, `round(3.5)` is 4. C rounds halves away from zero.

The book's `rounding.c` prints this table (one row left out here); `rounding.py` defines `c_round` to print exactly the same. `-lm` goes at the end of the command, because the linker reads libraries in order. Book: Section 5.4.

Scope and Lifetime

Table 5.1 Scope and lifetime of the four kinds of variables in C.

Kind	Declared	Visible (scope)	Exists (lifetime)
local	inside a function	declaration to end of block	entering to leaving the block
parameter	in the function header	the whole body	during one call
global	outside every function	declaration to end of file	the whole run
`static` local	inside, with `static`	like a local	the whole run

A global is visible everywhere, so when it is wrong every function is a suspect. A `static` local keeps its value between calls and stays private.

Table 5.1: the receipt counter `next_receipt_number` is a `static` local, the best of both worlds. A block may declare a second `kg` that hides the outer one. Python has no block scope and no `static` locals; it uses `global` and module variables. Book: Section 5.5.

Randomness You Can Repeat

$$x_{k+1} = (1103515245\, x_k + 12345) \bmod 2^{32} \tag{5.2}$$

Saturday promotion: a bill of at least Rp 50,000 rolls two dice; a double wins a free ironing voucher.

`rand()` differs between systems, so we write Equation (5.2) ourselves: a `static` state, `uint32_t` wrapping at 2^{32}, and bits 16 to 30 returned as 0 to 32767.

`1 + my_rand() % 6` gives a face from 1 to 6. Python imitates the wrap-around with `& 0xFFFFFFFF`.

Pseudo-random means the same seed gives the same sequence: a feature, because such a program can be tested. Faces 1 and 2 are very slightly more likely (5462 against 5461 of 32768), harmless for a voucher, not for a lottery. Book: Section 5.6.

Recursion: a Base Case and a Smaller Problem

DEFINITION. A *recursive* function solves a problem by calling itself on a smaller instance of the same problem. It has a *base case*, small enough to answer directly, and a *recursive step*, which reduces every other instance to a smaller one and combines the answer.

$$n! = \begin{cases} 1 & \text{if } n \leq 1, \\ n \cdot (n-1)! & \text{if } n > 1. \end{cases} \tag{5.3}$$

Listing 5.3 From `recursion.c`: the factorial, translated line by line from its definition.

```c
long factorial(int n)
{
    if (n <= 1)                         /* base case: stop here */
        return 1;
    return n * factorial(n - 1);        /* recursive step: a smaller problem */
}
```

Listing 5.3 is Equation (5.3) in C. Book: Section 5.7.1.

Worked Example 2: Tracing `factorial(4)`

`factorial(4)` needs `4 * factorial(3)` and waits; `factorial(3)` waits for `factorial(2)`, which waits for `factorial(1)`.

`factorial(1)` is the base case and returns 1 at once.

The waiting calls finish in reverse: $2 \times 1 = 2$, $3 \times 2 = 6$, $4 \times 6 = 24$.

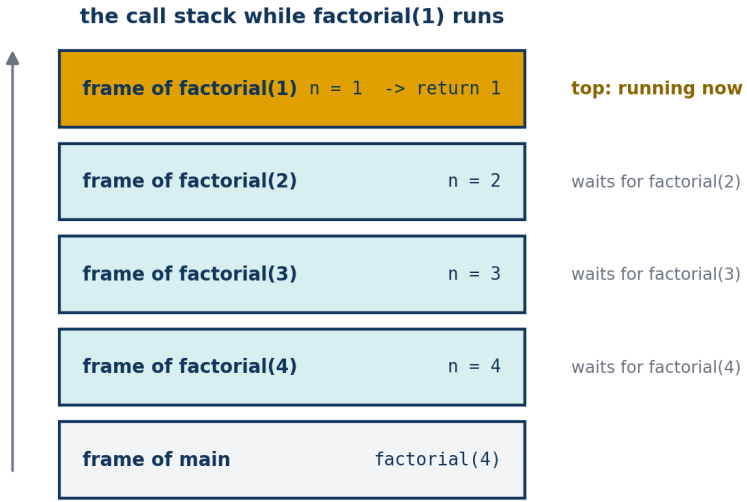


Figure 5.2 The call stack at the deepest point of `factorial(4)`: each call has its own frame with its own `n`, and every frame below the top waits for the answer of the frame above it.

*Figure 5.2: each call has a **frame** on the **call stack**. The stack is a few megabytes; a recursion without a reachable base case ends in a stack overflow; Python stops at depth 1000 with `RecursionError`. Book: Section 5.7.1.*

A Useful Recursion: Rupiah with Dots

Listing 5.4 From `recursion.c`: amounts printed with a dot between groups of three digits.

```c
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 */
}
```

Listing 5.4: to print 1234567, first print 1234 (the same problem, smaller), then `.567`. `%03ld` keeps three digits, so 5007 becomes 5.007, not 5.7. It prints `Rp 7`, `Rp 28.000`, `Rp 187.500`, `Rp 1.234.567`. Book: Section 5.7.2.

The Price of Recursion: Fibonacci

$$F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2) \text{ for } n \geq 2 \tag{5.4}$$

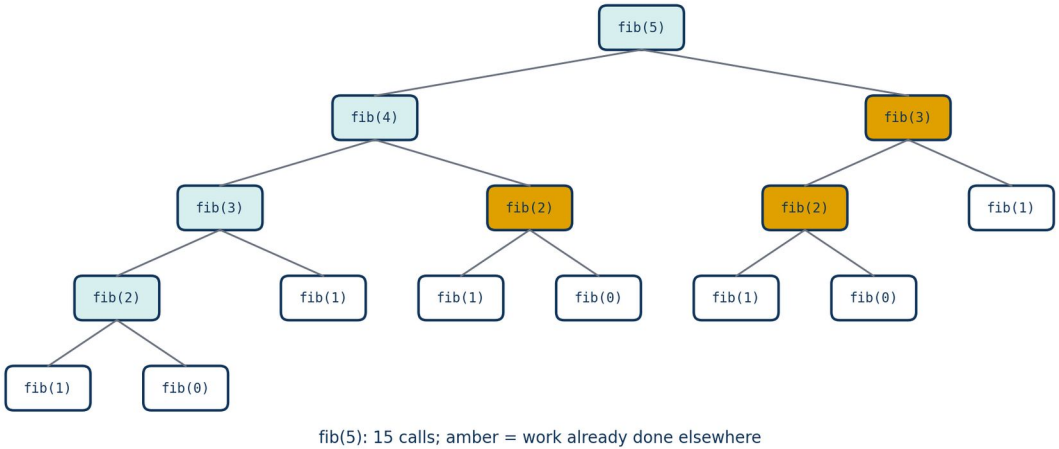


Figure 5.3 Every call made by `fib(5)`: fifteen calls to compute one number, and the amber calls repeat work already done elsewhere in the tree.

Equation (5.4) becomes a three-line function, correct and costly: in Figure 5.3 the whole subtree of `fib(3)` is computed twice. Book: Section 5.7.3.

Counting the Calls

$$C(n)=C(n-1)+C(n-2)+1, C(0)=C(1)=1 \tag{5.5}$$

From Equation (5.5): $C(2)=3$, $C(3)=5$, $C(4)=9$, $C(5)=15$, the fifteen boxes. In general $C(n)=2F(n+1)-1$.

```
fib(10) =     55 after      177 calls
fib(20) =   6765 after    21891 calls
fib(25) =  75025 after   242785 calls
fib(30) = 832040 after  2692537 calls
```

Each step of 5 multiplies the work by about $1.618^5 \approx 11$*. A loop that keeps the last two numbers needs 30 steps for* `fib(30)`*; Chapter 8 measures the two. Choose recursion when the problem is nested, a loop when recursion repeats work. Book: Sections 5.7.3 and 5.7.4.*

Cashier v4: the Header

Listing 5.5 From `laundry.h`: the promises of the business-rules module, with its constants and an include guard.

```
#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
```

Listing 5.5: every promise has a one-line comment, including the answer for bad input. `#define` gives each constant one home; the include guard makes a second `#include` harmless. Book: Section 5.8.1.

Cashier v4: the Rules

Listing 5.6 From `laundry.c`: two of Profitina Laundry's business rules, one function per rule.

```c
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;
}
```

`charged_kg` is Equation (5.1); `bill_for` combines price and kilograms and adds the express fee. It computes and returns; it never prints.

Listing 5.6: `price_per_kg` is a `switch` that returns 0 for an unknown service; the helper `print_thousands` is marked `static`, private to the file. Book: Section 5.8.1.

Compiling Several Files

```
gcc -std=c17 -Wall -Wextra -o cashier cashier.c laundry.c -lm
```

Name every `.c` file; never the header, which `#include` brings in.

`gcc` compiles each file into an *object file*; the **linker** joins them, matching each call in `cashier.c` to its definition in `laundry.c`.

Promised in the header but never defined: the compiler is satisfied, the linker says `undefined reference to ...`, "a promise was not kept".

Python: `python3 cashier.py` finds `laundry.py` in the same folder and imports it.

The same `laundry.h` will serve the day report of Chapter 6 and the closing summary of Chapter 7 unchanged. Book: Section 5.8.2.

A Morning at the Counter

```
#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 by hand: 1.2 kg becomes the minimum 2 kg, $2 \times 10000 = 20000$*, plus half is 30,000; 12.75 kg is shown as 12.8, charged as 13 kg,* $91000 + 45500 = 136500$*. Book: Section 5.9.*

Testing It

Listing 5.7 From `test_laundry.c`: the helper that compares each rule with a hand-worked answer.

```
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);
    }
}
```

The valuable checks are the edges: 0.8 kg → 2, exactly 2.0 kg → 2, 2.1 kg → 3; the receipt counter starts at 1.

When express drops to 40%, `check(bill_for(2, 1.2, 1), 30000, ...)` fails with `got 28000`: good news, the test noticed.

Listing 5.7 gets the line from `__LINE__` and the program returns 1 on any failure, so scripts notice. A fourth service changes exactly two functions by one `case` line each. Book: Sections 5.10 and 5.11.

Using It for Real

A tailor — `price_of(job)`. **A photocopy kiosk** — one more function, `discount_for(pages)`.

A motorcycle workshop — a fee plus parts. **A catering kitchen** — "at least 10 boxes, in steps of 5", the shape of Equation (5.1).

Write the header of promises first, then a test with answers the owner worked out on paper, and only then `main`.

Change the rules, not the story: the owner can read and check a test far more easily than the code. Book: Section 5.12.

Common Mistakes

Expecting a function to change its argument; **a missing** `return` (*control reaches end of non-void function*).

No prototype (*implicit declaration*); **no** `-lm` (*undefined reference to* `ceil`).

Overflow in `price * kg` — cast one factor to `long`; **an unreachable base case**; **tree recursion** for large n.

Python's `round` **is not C's**; **a local that hides a global** — `gcc -Wshadow` warns.

Each mistake in the book comes with its compiler message and its fix. Book: Section 5.13.

Summary: Five Things to Remember

1. A function is a named piece of work; the caller needs only its header (Figure 5.1).

2. Arguments are copied into parameters: the answer comes back through `return`.

3. A prototype is a promise; headers are files of promises; the linker's *undefined reference* means a promise was not kept.

4. Locals live for one call, globals and `static` locals for the whole run (Table 5.1); prefer locals.

5. Recursion needs a base case; each call has its own frame (Figure 5.2); `fib` makes $2F(n+1)-1$ calls.

Cashier v4: rules in a module, the story in `main`, and a test that checks each rule against hand-worked answers. Book: Section 5.14.

Next Time

Chapter 6 — Arrays: Many Values, One Name. Declaring and walking arrays, counters, passing arrays to functions, bubble sort, linear and binary search, two-dimensional tables.

Cashier version 5 remembers the whole day and prints totals per service, the average, the median and the three biggest bills.

Before then: Exercises 5.1, 5.3, 5.4 and 5.7.

Exercise 5.4 is Euclid's algorithm, $\gcd(a, b) = \gcd(b, a \bmod b)$*, a recursion that is shorter than any loop. Book: Sections 5.15 and 5.16.*

Lecture 6 — Arrays: Many Values, One Name

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 6

Lecturer: Irfan Subakti — yifana@gmail.com

Today a program learns to remember many values of one kind: arrays and Python lists, counters, arrays as parameters, bubble sort, linear and binary search, two-dimensional tables, and cashier version 5 with its day report. Book: Chapter 6, Sections 6.1–6.15.

What This Meeting Covers

1. **The idea** — a row of numbered boxes; why the index starts at 0; nobody checks the fence; Python's list.
2. **Using arrays** — declare, fill, walk; arrays as counters; passing arrays to functions with a size and `const`.
3. **Sorting** — bubble sort, its invariant and its cost.
4. **Searching** — linear and binary search, and when to use which.
5. **Tables and the cashier** — two-dimensional arrays in row-major order; version 5 with totals, mean, median and top three.

Every program again in C and Python with identical output. Book: Chapter 6.

Where We Left Off

Cashier v4 prints a correct receipt and forgets it the moment the next customer steps up. It keeps only running totals.

At closing time Mbak Rina asks: the average bill, the biggest order, how many of each service, the busiest day of the week.

A total gives a sum and an average, but not a median, not the top three and not a sorted list: those need **every** bill at the end of the day.

`long bill1, bill2, ..., bill100;` fails: a loop cannot say “the next variable”. We need one name and a number.

The same need appears at a warung’s daily sales, the rainfall at a fruit and durian orchard, the marks of a class. Book: Section 6.1.

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*, selected by an *index* 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 $n - 1$.

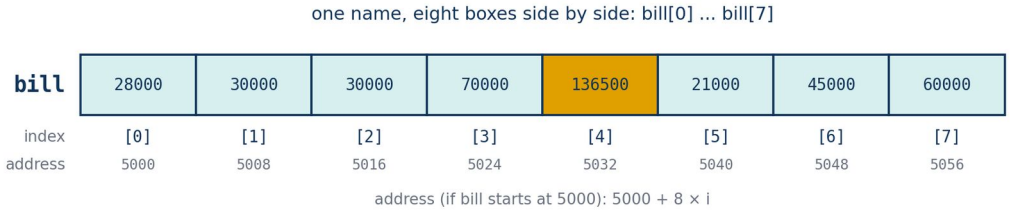


Figure 6.1 The array `bill` with eight elements: one name, indexes 0 to 7, and the elements side by side in memory, eight bytes apart on a computer where a `long` takes eight bytes.

Figure 6.1: `bill[4]` is the fifth box and holds 136500. The power is that the index can be a variable: one loop over `i` visits every box. Book: Section 6.2.1.

Why the Index Starts at 0

$$\text{address}(\text{bill}[i]) = a + i \cdot s \tag{6.1}$$

The index is the **distance** from the start, measured in elements; Equation (6.1) needs no correction for the first one.

With $a = 5000$ and $s = 8$: `bill[4]` is at $5000 + 4 \cdot 8 = 5032$. One multiplication and one addition, however long the array.

The same formula fixes the size: all elements are reserved at once, and the bytes after `bill[7]` may belong to another variable.

A program that must grow reserves more room than it expects to use, or copies into a bigger block, which is what a Python list does quietly. Book: Section 6.2.2.

Nobody Checks the Fence

`bill[8]` in an array of eight: the compiler does not stop it, the program does not check it. Equation (6.1) computes the address past the end, and the program touches whatever lives there.

The C standard calls this **undefined behavior**: anything may happen, including appearing to work until the day it does not.

The defense is a habit: `for (i = 0; i < n; i++)` with `<`, never `<=`; check every index from input; name the size as a constant.

Develop with `-fsanitize=address,undefined`: the first out-of-bounds access stops the program with a clear report.

Python checks: `bill[8]` raises `IndexError: list index out of range`. A list knows its length (`len`), grows with `append`, and holds references to number objects, which costs memory and speed. Book: Sections 6.2.3 and 6.2.4.

C Array and Python List

Table 6.1 The C array and the Python list side by side.

Question	C array	Python list
first index	0	0
knows its length?	no: pass `n` along	yes: `len(bill)`
can grow?	no: fixed when created	yes: `append`, `pop`
index outside?	undefined behavior	`IndexError`
what a box holds	the value itself, packed	a reference to an object

Table 6.1: the list pays for its comfort with memory and speed, several times the memory of a C array for a million numbers; for a few hundred orders a day the difference does not matter. Book: Section 6.2.4.

Declaring, Filling and Walking

Listing 6.1 From `day_bills.c`: eight bills in one array, added up and searched for the biggest.

```
    long bill[N] = {28000, 30000, 30000, 70000, 136500, 21000, 45000, 60000};

    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;
```

Missing values in an initializer become 0: `int count[4] = {0};`. With the size left out, the compiler counts the values.

`best` keeps the **index** of the biggest bill, so both the value and its position are known. `sizeof bill / sizeof bill[0]` gives 8, but only where the declaration is visible.

Listing 6.1 prints total 420500, average 52562.50 and the biggest, 136500 at index 4. An array declared inside a function without an initializer holds garbage. Book: Section 6.3.

Worked Example 1: Tracing the Sum

Before the loop `total` is 0. $i=0$: 28000. $i=1$: 58000. Then 88000, 158000, 294500, 315500, 360500, and 420500 at $i=7$. `i++` makes `i` equal to 8; `i < N` is false; the loop ends without touching `bill[8]`.

$$\acute{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i \tag{6.2}$$

Equation (6.2) is exactly that loop followed by `(double) total / N`: 420500/8=52562.5.

Trace one loop by hand once, the way the computer runs it. The Python version walks with `range(N)`; idiomatic Python would use `sum(bill)` and `max(bill)`, and today we write what they do inside. Book: Section 6.3.

Arrays as Counters

Listing 6.2 From `service_count.c`: an array of counters, indexed by the service code.

```
    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++;
    }
```

The value of the data becomes the index: nothing is compared or searched.

Twenty orders give 11, 5 and 3 for the services and 1 unknown code, the typing slip 9. Without the guard, `count[9]++` writes past the end.

Listing 6.2 wastes `count[0]` so that the code can be the index directly: a good trade. The program draws each count as a bar of stars; Python writes `"*" * count[s]`. Book: Section 6.4.

Passing Arrays to Functions

Listing 6.3 From `array_functions.c`: a function that promises not to change the array, and one that changes the caller's array.

```c
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;
}

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;
}
```

The parameter receives the **address** of the caller's array, not a copy: a function can change it.

The function does not know the length, so the length comes as a parameter `n`; `const` marks an array the function only reads.

In Listing 6.3, `add_to_all(evening, 3, 5000)` adds a delivery fee to every evening bag in `main`'s own array. Chapter 7 explains why: the address is the key. Book: Section 6.5.

Bubble Sort

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
```

Pseudocode 6.1: compare neighbors, swap when out of order; the biggest value rises to the end like a bubble. A sorted list gives the median, the top three and fast searching. Book: Section 6.6.

Worked Example 2: Six Bills, Pass by Pass

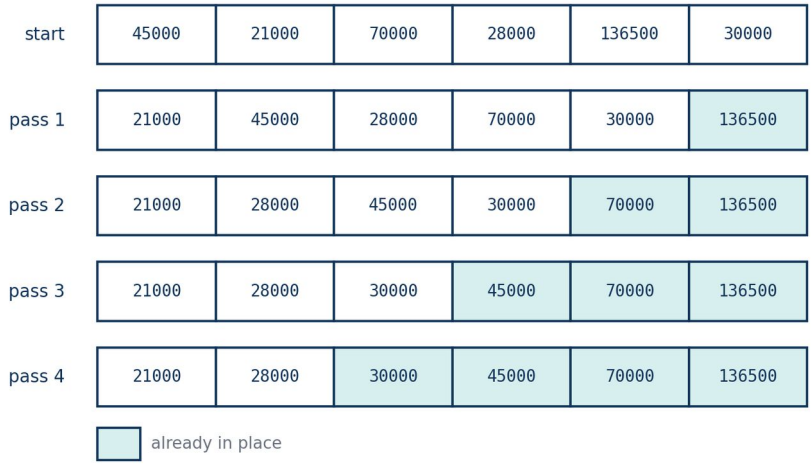


Figure 6.2 Bubble sort on six bills: after each pass one more value (shaded) is in its final place at the right; pass 4 makes no swap, so the sort stops early.

Work pass 1 of Figure 6.2 on the board: 45000 > 21000, swap; 45000 < 70000, no swap; 70000 > 28000, swap; 70000 < 136500, no swap; 136500 > 30000, swap. Now 136500 is at the right end. Book: Section 6.6.

The Invariant and the Cost

> **KEY INSIGHT:** After pass p, the last p positions hold the p biggest values, in their final order. This *loop invariant* explains why the inner loop stops at `n - pass`, and why $n-1$ passes are always enough.

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2} \tag{6.3}$$

By Equation (6.3): 15 comparisons for 6 bills; almost 200 million for 20,000; four times more each time the list doubles.

Bubble sort is **stable**: it swaps only when strictly bigger, so two bills of Rp 30,000 keep their receipt order.

The right method to learn first, and almost never the right one for large data; library sorts are much faster. Book: Section 6.6.

Linear and Binary Search

Listing 6.4 From `sort_search.c`: binary search in a sorted array (the book's version also prints 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;
        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;
}
```

Linear search looks from the front until found or exhausted, and returns -1 for "not there": up to n looks, on any array.

Binary search needs sorted data and halves the part still in question at every look, like opening a dictionary in the middle.

In Listing 6.4, `low + (high - low) / 2` never overflows; `(low + high) / 2` hid in Java's library for nine years before it was found in 2006. Book: Section 6.7.

Tracing Binary Search

```
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
```

$$k_{\max} = \lfloor \log_2 n \rfloor + 1 \tag{6.4}$$

For $n = 6$, $\log_2 6 \approx 2.58$ and Equation (6.4) gives 3 looks, exactly what 50000 needed. For a million receipts: 20 looks against up to a million. Sort once and search many times: binary; search once: linear. Book: Section 6.7.

Two-Dimensional Arrays

Kilograms per day and service: `int kg[7][3];`, row `d`, column `s`, element `kg[d][s]`. The initializer has one pair of braces per row.

Two nested loops visit every cell: the outer over rows, the inner over columns. A row total resets for each row; column totals live across all rows.

$$\text{address}(\text{kg}[r][c]) = a + (r \cdot C + c) \cdot s \tag{6.5}$$

Equation (6.5): r whole rows of C elements come first, then c elements. That is why a parameter must say its number of columns: `int kg[][3]`. Book: Section 6.8.

Row-Major Order

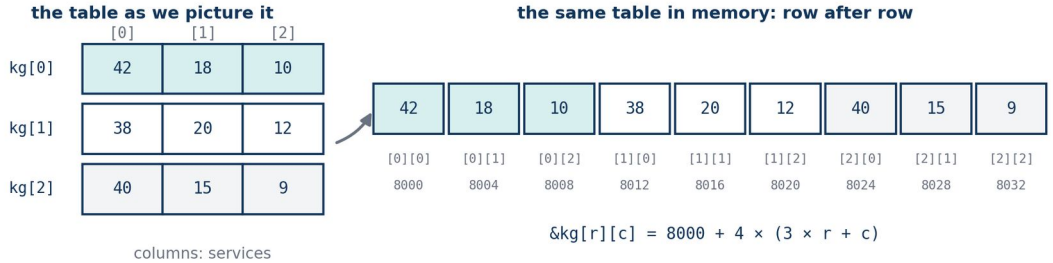


Figure 6.3 The first three rows of the week table, as we picture them and as they lie in memory: row after row, so that `kg[r][c]` lies $3r+c$ elements after the start (four bytes per `int`).

Figure 6.3: the table as we picture it and as memory holds it. Python uses a list of lists; never build it with `[[0] * 3] * 7`, which makes seven names for one row. Book: Section 6.8.

Cashier v5: Parallel Arrays

Listing 6.5 From `day_report.c`: the day read into parallel arrays, at most `MAX_ORDERS` orders.

```
    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++;
    }
```

`service[i]`, `weight[i]` and `bill[i]` together describe order *i*: **parallel arrays**.

`n < MAX_ORDERS` comes first in the condition, so a record day can never write past the end; `n` counts only valid orders.

Per-service totals are arrays indexed by the service code: `count[service[i]]++`, `kg[service[i]] += weight[i]`, the counter pattern three times at once.

Listing 6.5 reuses `laundry.h` of Chapter 5 unchanged: `price_per_kg` says whether a service exists and `bill_for` computes the bill. 100 orders is three times the busiest day and costs 800 bytes. Book: Section 6.9.

Cashier v5: the Median

Listing 6.6 From `bills.c`: the median of a copy, so that the caller's order is untouched.

```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;
}
```

Listing 6.6 belongs to a second module, `bills.h` and `bills.c`, beside the unchanged `laundry.h` of Chapter 5. Version 5 stores the day in parallel arrays: service, weight, express, bill. Book: Section 6.9.

A Saturday Report

```
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 by hand: $3.4+12.75+4.2+7.9+5.5=33.75$ *kg;* $523500/11=47590.9$*, printed 47.590 by whole-number division. The unknown service 7 was skipped. Book: Section 6.10.*

Mean or Median?

The mean, Rp 47,590, is higher than seven of the eleven bills.

One order of Rp 136,500 pulls the mean up but moves the median only one place.

“What does a normal customer spend?” → the median, Rp 35,000. “How much money came in per order?” → the mean.

The tests of this chapter include the empty array and the promise that `median` leaves its input unchanged. Book: Sections 6.10 and 6.11.

Testing and Making It Better

Nine checks on small arrays worked by hand: five values with total 150 and median 30, one element, an even length whose median is the whole-number mean of 2 and 3, that is 2.

Two checks are about behavior: after `median(a, 5)` the array is unchanged, and an empty day gives a total of 0 instead of a crash.

Better: warn when orders were ignored at capacity; sort an array of *indexes* so the data never move; round the average on purpose with `(sum + n / 2) / n`.

Chapter 11 keeps the day in a file; the functions of `bills.c` will not change, because they do not care where the array was filled. Book: Sections 6.11 and 6.12.

Using It for Real

Any list of one kind: a warung's sales per day, rainfall per day at the orchard, sacks of rice in a delivery.

Counters by code, sorting for the top three, binary search in a sorted list of receipt numbers, a week table per day and product.

Reserve a named maximum size and check every index that comes from input.

The same five small tools answer almost every question an owner asks at closing time. Book: Section 6.13.

Common Mistakes

`i <= n` — one past the end; **starting at 1**; **uninitialized counters**; **an unchecked index from input**.

`sizeof` **on an array parameter** — it measures an address; pass the length.

Sorting the caller's data by accident; **binary search on unsorted data**; `(low + high) / 2`.

`[[0] * 3] * 7` in Python; `a = b` for arrays: no compile in C, a second name in Python; copy instead.

Each mistake in the book comes with its fix. Book: Section 6.14.

Summary: Five Things to Remember

1. Element i is at distance i from the start (Equation (6.1), Figure 6.1): indexes run from 0 to $n-1$, and any element is one step away.

2. C does not check indexes: loop with `<`, check input, use the sanitizer. Python raises `IndexError`.

3. An array parameter is the caller's array: pass the length, mark read-only arrays `const`.

4. Bubble sort needs up to $n(n-1)/2$ comparisons; binary search at most $\lfloor \log_2 n \rfloor + 1$ looks, on sorted data.

5. A table is stored row after row (Figure 6.3); the median describes the typical customer.

Cashier v5 stores the day in parallel arrays and reports totals per service, the mean, the median and the top three. Book: Section 6.15.

Next Time

Chapter 7 — Pointers: Addresses as Values. `&` and `*`, `swap`, output parameters, pointer arithmetic, `const` and pointers, function pointers and `qsort`.

Why `add_to_all` could change `main`'s array, and why `scanf` needs `&`: the answer is the address.

Before then: Exercises 6.1, 6.2, 6.3 and 6.6.

Exercise 6.2 is Equation (6.1) and Equation (6.5) by hand. Book: Sections 6.16 and 6.17.

Lecture 7 — Pointers: Addresses as Values

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 7

Lecturer: Irfan Subakti — yifana@gmail.com

Today an address becomes a value: `&` and `*`, `swap`, output parameters, arrays and pointer arithmetic, `const` with pointers, pointers to functions and `qsort`, pointer safety, and cashier version 6. We take pointers slowly, with pictures. Book: Chapter 7, Sections 7.1–7.15.

What This Meeting Covers

1. **The idea** — every variable has an address; following the arrow; `NULL`; pointers you have already used.
2. **Changing the caller** — `swap`, pass-by-reference, output parameters with a status.
3. **Arrays and pointers** — decay, pointer arithmetic, `a[i]` is `*(a + i)`, walking with a pointer.
4. **`const` and function pointers** — what `const` protects; behavior as a parameter; `qsort`.
5. **Cashier v6 and safety** — four answers from one call; the classic pointer mistakes and the sanitizer.

Python has no pointers; at every step we say what it does instead. Book: Chapter 7.

Where We Left Off: Two Promises

A function returns **one** value, yet at closing time Mbak Rina wants total, average, smallest and biggest from one walk.

A function receives **copies**, yet `scanf` fills our variables and `add_to_all` of Chapter 6 changed `main`'s array.

A `swap(left, right)` of two bags on a shelf cannot work with values: Chapter 5 proved it.

The laundry already has the answer: a **tag** with a shelf number. Small, easy to copy and pass on, and whoever holds it reaches the same bag.

The receipt number leads to exactly one bag on one shelf. A lost receipt: the bag exists but cannot be found quickly. Two receipts with one number: two customers claim one bag. These are exactly the pointer mistakes of today. Book: Section 7.1.

Every Variable Has an Address

> **DEFINITION.** A *pointer* is a variable whose value is the address of another variable. If `p` holds the address of `kg`, `p` *points to* `kg`. `int *p;` makes `p` a pointer to an `int`. The operator `&` gives the address of a variable; `*`, applied to a pointer, gives the variable it points to, a step called *dereferencing* or *following* the pointer.

Memory is a long row of numbered lockers; an `int` takes four, a `double` eight. The number of the first locker is the **address**.

You have written `&` since your first `scanf("%d", &kg)`. Book: Section 7.2.1.

Following the Arrow

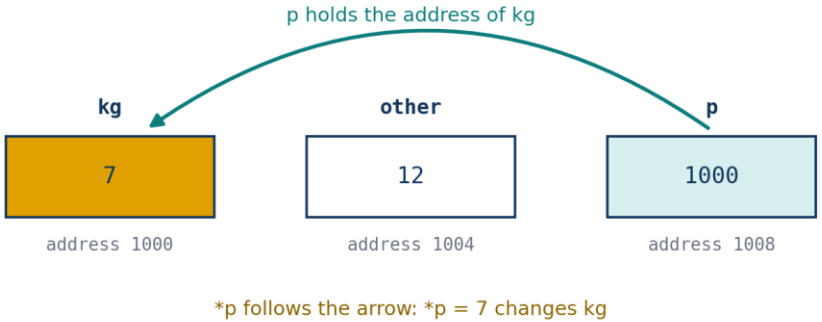


Figure 7.1 Three variables in memory: `p` holds the address of `kg`, drawn as an arrow; `*p` follows the arrow, so `*p = 7` changes `kg`.

$$*(\&x) \equiv x \tag{7.1}$$

***Figure 7.1**: `p` at 1008 holds 1000, the address of `kg`, drawn as an arrow. By Equation (7.1), `&` and `*` undo each other; ≡ means "is the same thing as", so `*(&x)` can be assigned to. The addresses are small invented numbers. Book: Section 7.2.2.*

Two Stars, Two Jobs

In a **declaration**, `int *p;` says “p is a pointer to int”. In an **expression**, `*p = 7;` says “follow p”.

Read `int *p` as “`*p` is an `int`”. Then `double *price;` says `*price` is a `double`.

Trap: `long *a, b;` makes only `a` a pointer. Write one declaration per line.

A pointer has a type: it says how many bytes to read and how. `NULL` points nowhere; following it is undefined behavior and usually a crash.

`int **pp;` says `**pp` is an `int`: a pointer to a pointer, needed for linked lists in Chapter 10. Book: Sections 7.2.2 and 7.2.3.

Your First Pointer

Listing 7.1 From `first_pointer.c`: taking an address, writing through it, re-aiming the pointer, and copying it.

```c
int kg = 5;
int *p = &kg;                   /* p holds the address of kg */
*p = 7;                         /* write THROUGH the pointer */

int other = 12;
p = &other;                     /* aim p at another variable */
*p += 1;

int *q = p;                     /* copying a pointer: two arrows */
*q = 20;
```

After `*p = 7;` the value in `kg` is 7, although the statement never names `kg`: that is the whole point.

`p = &other;` without a star moves the arrow; `*p += 1` makes `other` 13. `int *q = p;` gives two arrows to one box: `*q = 20` makes `*p` 20 as well.

*Draw **Listing 7.1** on the board as boxes and arrows, one statement at a time. The program prints no addresses, because real addresses change on every run. Python has no pointers, only references: a one-element list plays a box that two names share. Book: Section 7.3.*

Pointers You Have Already Used

`scanf("%d", &kg)` receives a pointer to `kg` and writes through it.

The text in `printf("Hello\n")` is an array of characters; `printf` receives a pointer to its first character.

The parameter `const long a[]` of Chapter 6 is a pointer, which is why those functions could change the caller's array and did not know its length.

Forgetting the `&` hands `scanf` the value of `kg`, a random number that it then uses as an address. Book: Sections 7.2.4 and 7.4.

Changing the Caller's Variables

Listing 7.2 From `swap.c`: swapping copies changes nothing in `main`; swapping through pointers changes `main`'s variables.

```c
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;
}
```

```
after swap_copies: left 9, right 4
after swap:        left 4, right 9
```

Listing 7.2 *is called as* `swap(&shelf_left, &shelf_right)`*. The next page draws both calls. Book: Section 7.4.*

Worked Example 1: Two Swaps in Memory

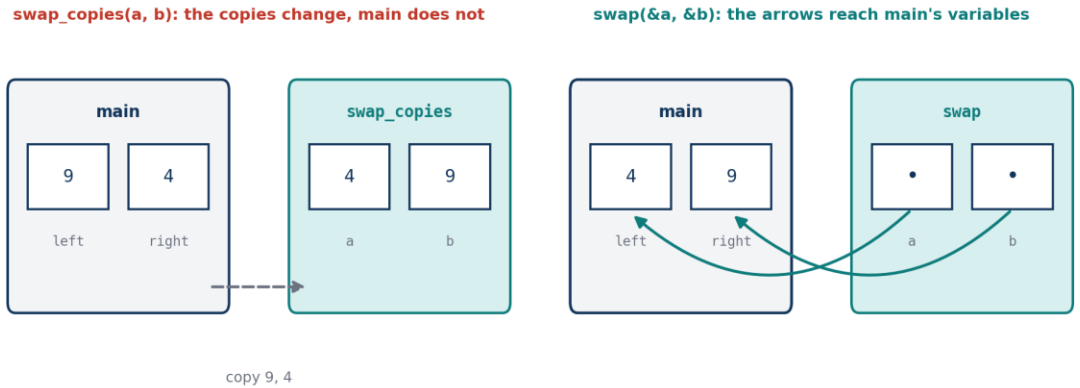


Figure 7.2 Left: `swap_copies` works on copies, and `main` keeps 9 and 4. Right: `swap` receives the addresses of `main`'s variables and exchanges the values there.

*Step through **Figure 7.2** on the board: `int t = *a;` reads `main`'s left (9); `*a = *b;` copies `main`'s right (4) into left; `*b = t;` puts 9 into right. Strictly, C still passes by value: a copy of the address, which leads to the same variable, like a photocopy of a shelf tag. Book: Section 7.4.*

Sorting Three Bags, and Python

`if (bag[0] > bag[1]) swap(&bag[0], &bag[1]);` then `bag[1]`, `bag[2]`, then `bag[0]`, `bag[1]` again: a *sorting network*.

On `{7, 3, 5}`: swap to `{3, 7, 5}`; swap to `{3, 5, 7}`; no swap. The output says `3 5 7`.

Python cannot pass the address of a variable. It swaps in the caller with `left, right = right, left`, or changes a shared list inside a function.

Try the other five orders of three numbers on paper; each ends sorted, a small proof that three steps suffice. Book: Section 7.4.

Output Parameters: One Call, Several Answers

Listing 7.3 From `out_params.c`: three answers through pointers, and a status that says whether they are valid.

```c
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;
}
```

*In **Listing 7.3** the caller passes the addresses of three variables that receive the answers; the return value says whether they are valid. An empty day returns 0 and does not touch them. `scanf` follows the same pattern. Book: Section 7.5.*

Worked Example 2: 3.45 kg in kg and Grams

$$g = \text{round}(1000\,w), \qquad \text{kg} = \lfloor g/1000 \rfloor, \qquad \text{grams} = g \bmod 1000 \tag{7.2}$$

$w = 3.45$: $g = 3450$, kg $= 3$, grams $= 450$. The program prints `3.45 kg is 3 kg and 450 g`.

The rounding matters: 0.29 kg times 1000 is 289.99999999999994 in binary, which a plain conversion turns into 289 grams. `lround` gives 290.

`split_weight(3.45, &kg, &grams)` writes both answers through `*kg` and `*grams`.

Equation (7.2) on the board, then the call. Python returns a tuple, `kg, grams = split_weight(3.45)`, and `None` for "no valid answer". Book: Section 7.5.

The Name of an Array Is an Address

In almost every expression, `bill` means `&bill[0]`: *array-to-pointer decay*. So `long *p = bill;` needs no `&`.

The exception is `sizeof`: `sizeof bill` is 40 bytes (five `long`s), `sizeof p` is 8, the size of an address.

An array *is* its elements; a pointer only *knows where* they are.

This is why an array parameter cannot tell its length. Book: Section 7.6.1.

Pointer Arithmetic

$$\text{address}(p + k) = \text{address}(p) + k \cdot \text{sizeof}(* p) \tag{7.3}$$

$$a[i] \equiv * (a + i) \tag{7.4}$$

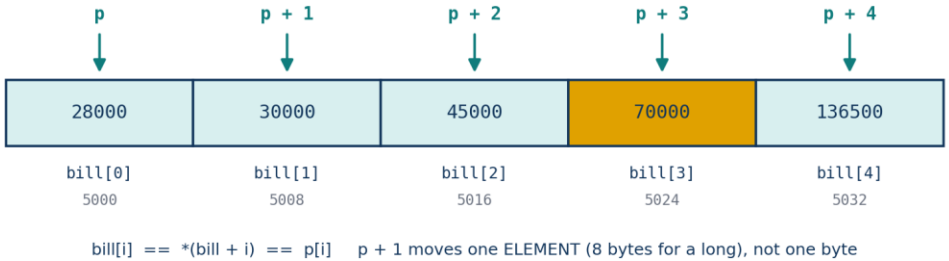


Figure 7.3 Pointer arithmetic: `p + k` points `k` elements further, whatever the size of an element, so `bill[i]`, `*(bill + i)` and `p[i]` name the same box.

Equation (7.3) moves by elements, not bytes; the C standard defines the brackets by Equation (7.4). In ***Figure 7.3*** *the array starts at 5000:* `p + 3` *is* $5000 + 3 \cdot 8 = 5024$*, and* `*(p + 3)` *is 70000. With* `int` *it would be 5012, still element 3. Book: Section 7.6.2.*

Walking with a Pointer

Listing 7.4 From `pointer_walk.c`: a pointer walk, pointer difference, and a negative index from a moved pointer.

```
    long bill[5] = {28000, 30000, 45000, 70000, 136500};
    long *p = bill;                       /* the same as &bill[0] */

    long sum = 0;
    for (long *q = bill; q < bill + 5; q++)   /* walk the array with a pointer */
        sum += *q;

    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]);
```

`bill + 5` is one past the end: a pointer may point there, never be followed there.

Listing 7.4 *prints `sum by walking: 309500`, `last - p = 4 elements apart`, and `*p = 70000`, `p[-1] = 45000`. Pointer difference has type `ptrdiff_t`, printed with `%td`. The book's program also reverses "laundry" with two pointers walking toward each other. Book: Section 7.6.3.*

Why Pointers and Not Only Indexes?

Speed is no longer the reason: today's compilers produce the same code for `a[i]` and a pointer walk.

A pointer can point into the **middle** of an array: `total(bill + 3, 2)` adds `bill[3]` and `bill[4]` with an unchanged function.

Pointers are how C passes arrays, strings and output parameters, and later links structures together.

Python expresses the same walks with indexes and slices. Book: Section 7.6.4.

`const` and Pointers

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

*Read **Table 7.1** right to left: `const long *p` is "p is a pointer to a long that is const". The second row is the one you will write most; `const long a[]` in a parameter list means the same. Book: Section 7.7.*

Pointers to Functions

Listing 7.5 From `sort_by.c`: one bubble sort that asks a function which of two values comes first.

```c
int cheaper_first(long a, long b) { return a < b; }
int dearer_first(long a, long b) { return a > b; }

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;
            }
}
```

*In **Listing 7.5**, `int (*before)(long, long)` is a pointer to a function; the parentheses are required. The sort is written once; the rule is a parameter. An array `rule[3]` of such pointers lets a menu choose the rule with one checked index. Book: Section 7.8.*

Three Rules, One Sort, and `qsort`

```
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
```

$$\mathrm{cmp}(a, b) = [a > b] - [a < b] \in \{-1,\ 0,\ 1\} \tag{7.5}$$

*"Round first" keeps the original order within each group: the sort is **stable**. `qsort` takes the array, the count, the element size and a comparison on `const void *`; write it as Equation (7.5), never `a - b`, which overflows. `qsort` is not required to be stable. Book: Section 7.8.*

Cashier v6: Four Answers from One Call

Listing 7.6 From `stats.c`: the four-answer `day_stats` with a `const` pointer walk.

```c
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;
}
```

Listing 7.6 *refuses a null pointer and an empty day, and writes the answers only at the end, so a refused call leaves them untouched. Python's* `stats.py` *returns a tuple or* `None`*. Book: Section 7.9.*

The Closing Summary

```
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)
```

Same Saturday as Chapter 6: $33.75 + 10.7 + 8.5 = 52.95$ *kg, shown by Equation (7.2) as 52 kg 950 g. The average is now 47.591, rounded to the nearest rupiah, where Chapter 6 cut off the fraction: choose one rule and say it. Book: Section 7.10.*

Testing It

`split_weight(0.9996, ...)` must round up to 1 kg and 0 g, never report 0 kg and 1000 g.

`day_stats` must give the right four answers for four small bills, and refuse an empty day and a null pointer.

A refused call must leave the answers untouched: the test first puts the right value in `total`, calls with `n = 0`, and checks that `total` still holds it.

Fourteen checks in all, ending with `qsort` and `compare_long`. Testing behavior as well as results is what makes output parameters safe to use. Book: Section 7.11.

Pointer Safety

Initialize every pointer, to a real address or `NULL`. **Test before following** a pointer from outside (SEI CERT EXP34-C).

Never return the address of a local: it dies at the return and the pointer *dangles*; `gcc` warns *function returns address of local variable*.

Know who owns the memory: a function borrows it for the call and must not keep the pointer for later.

Develop with `-fsanitize=address,undefined`: a silent corruption becomes a loud, precise message.

```
return &best;          /* WRONG: best disappears at the return */
```

Return the value, a pointer into the caller's array (`return &bill[k];` is fine), or write through an output parameter. Book: Section 7.12.

Using It for Real

Output parameters with a status — a market scale: `int read_weight(double *kg)` returns 0 when the scale sends nothing.

Several statistics from one pass — a fruit and durian orchard's week: total, average per tree, weakest and strongest tree.

A rule chosen at run time — a photocopy kiosk sorts by price, by time or by name: one sort, three comparisons.

A pointer into the middle — `total(sales + 7, 7)` is the second week of a warung's month, without copying.

`split_weight` becomes `split_duration(int minutes, int *hours, int *mins)` for a motorcycle rental, and the two-pointer walk becomes the palindrome checker of Chapter 8. Book: Section 7.13.

Common Mistakes

Forgetting `&` in the call; **forgetting `*`** in the function (`a = b;` swaps the local addresses).

An uninitialized pointer; **following `NULL`**; **returning the address of a local**.

Arithmetic in bytes; **walking past the end**; **`sizeof` on a pointer**; **`a - b` in a comparison**.

`long *a, b;`; **`p == q` versus `*p == *q`**; **expecting Python to change a caller's number**.

`swap(left, right)` draws makes pointer from integer without a cast: treat that warning as an error. Book: Section 7.14.

Summary: Five Things to Remember

1. A pointer stores an address; `*p` follows it; `*(&x)` is `x` (Equation (7.1), **Figure 7.1**); `NULL` is never followed.

2. Passing addresses lets a function change its caller's variables (**Figure 7.2**); output parameters plus a status give several answers.

3. An array name decays to a pointer; `p + k` moves *k* elements; `a[i]` is `*(a + i)` (Equation (7.4)).

4. `const` before the type protects the value, after the star the pointer (**Table 7.1**); a function pointer passes behavior; `qsort` needs Equation (7.5).

5. Initialize, test for `NULL`, never return a local's address, develop with the sanitizer.

Cashier v6 gets four statistics from one call, splits the day's weight with Equation (7.2), and sorts with `qsort`. Book: Section 7.15.

Next Time

Chapter 8 — Evaluation and Discussion II. Six problems with functions, arrays and pointers: statistics functions, Fibonacci two ways, palindromes, a 3×3 determinant, a receipt by recursive binary search, and merging two shifts.

Each with a hand solution, a design, C and Python, a tempting wrong solution and a check-yourself question.

Before then: Exercises 7.1, 7.2, 7.5 and 7.7.

Exercise 7.1: draw the boxes and arrows after every statement; it is the best preparation for Chapter 8. Book: Sections 7.16 and 7.17.

Lecture 8 — Evaluation and Discussion II: Functions, Arrays and Pointers, Solved Step by Step

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 8

Lecturer: Irfan Subakti — yifana@gmail.com

No new feature of C today. We take functions, arrays and pointers and solve six complete problems slowly, each with a hand solution, a design, a program, and a tempting wrong solution taken apart. Book: Chapter 8, Sections 8.1–8.11.

What This Meeting Covers

0. The method — eight steps, and why the hand-worked example decides what “correct” means.

1. Statistics functions in a fixed skeleton. **2. Fibonacci** two ways, measured.

3. Palindromes three ways. **4. A** 3×3 **determinant** and invertibility.

5. Finding a receipt by recursive binary search. **6. Merging two shifts** with walking pointers.

7. Side by side — the six traps and the edge tests that expose them.

Every program was compiled with `-Wall -Wextra` *and again with* `-fsanitize=address,undefined`*; the Python versions print exactly the same text. Book: Chapter 8.*

Where We Left Off: the Toolkit

Chapter 5 — functions, prototypes, pass-by-value, `<math.h>` with `-lm`, scope, recursion with a base case.

Chapter 6 — arrays from 0 to $n-1$, loops with `i < n`, arrays passed with their length, bubble sort, linear and binary search, tables.

Chapter 7 — `&` and `*`, output parameters, `a[i]` is `*(a + i)`, two pointers walking toward each other, `const`, functions as parameters.

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	output parameter, recursion, counting calls
8.3	Is a word a palindrome?	strings as arrays, two pointers, recursion, XOR
8.4	3×3 determinant	two-dimensional array, one formula
8.5	Finding a receipt number	recursive binary search
8.6	Merging two shifts	two walking pointers, an output array

Table 8.1: naming the tool a problem needs is already half of its solution. Book: Section 8.1.

Eight Steps

1. Restate, and write down every open decision. **2.** Solve a small example by hand, to the last digit. **3.** Design on paper: functions, arrays, loop or recursion.

4. Estimate the cost: 2^n steps are useless for $n = 60$. **5.** Code one function at a time, warnings on.

6. Test the example and the edges. **7.** Find the tempting wrong solution and the input it fails on. **8.** Look back.

> **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*.

Pólya's four phases, understand, plan, carry out, look back, unfolded for programs. Try steps 1 to 3 yourself before reading each solution. Book: Section 8.2.

Problem 8.1: Statistics Functions

> **PROBLEM 8.1.** You are given a complete `main`, 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 bodies, without changing `main`, the names or the prototypes, so that the program prints the largest value, the smallest, the sum, the average and the standard deviation, the last two with three decimals.

The given code is an agreement: renaming `max` or changing a prototype breaks it, even if the program still runs.

Our data: the kilograms that a small boarding house brought on ten visits: 12, 7, 15, 9, 20, 11, 8, 14, 10, 18.

In real teams prototypes are agreements with other people's code. Book: Section 8.3.1.

8.1: by Hand

$$\acute{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i = \frac{124}{10} = 12.4 \tag{8.1}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} (x_i - \acute{x})^2} \tag{8.2}$$

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	12	7	15	9	20	11	8	14	10	18
$x_i - \acute{x}$	−0.4	−5.4	2.6	−3.4	7.6	−1.4	−4.4	1.6	−2.4	5.6
$(x_i - \acute{x})^2$	0.16	29.16	6.76	11.56	57.76	1.96	19.36	2.56	5.76	31.36

Equation (8.1) gives 12.4. For Equation (8.2), Table 8.2: the deviations add up to 0, a check of the arithmetic; $166.40/10 = 16.64$ *and* $\sqrt{16.64} \approx 4.079$*. So the program must print 20, 7, 124, 12.400 and 4.079. Book: Section 8.3.2.*

8.1: Design and Solution

Listing 8.1 From `stats.c`: the average, with the cast before the division.

```c
double average(int *a, unsigned int aSize)
{
    return (double) sum(a, aSize) / aSize;   /* cast BEFORE dividing */
}
```

`max` starts at `INT_MIN`, `min` at `INT_MAX`, so the first element always replaces them.

`sDeviation` reuses `average`, which reuses `sum`, then takes `sqrt` of the mean squared deviation: link with `-lm`.

The size is `unsigned int`, so the counter is `unsigned int i` too; mixing them draws *comparison of integer expressions of different signedness*.

Listing 8.1 is one of the five completed bodies; the output matches the hand computation line by line. Book: Sections 8.3.3 and 8.3.4.

8.1: the Tempting Wrong Solution

`min` started at 0 “because a counter starts at 0”; the average written `sum(a, aSize) / aSize`, two whole numbers.

```
Minimum value      : 0
Average            : 12.000
Standard deviation : 4.099
```

The minimum is wrong only for arrays of positive values, exactly the test many people forget. The whole-number mean also spoils the deviation.

> **CHECK YOURSELF.** Another program prints 4.300 instead of 4.079. *Answer:* it divided by $n-1=9$: $\sqrt{166.40/9}\approx 4.300$, the *sample* standard deviation. Neither is a bug; the bug is not saying which one a program computes.

Python’s `statistics.stdev` divides by $n-1$, `statistics.pstdev` by n. Book: Section 8.3.6.

Problem 8.2: Fibonacci Two Ways

PROBLEM 8.2. Read n. Compute $F(n)$, starting with $F(0)=0$ and $F(1)=1$, and the sum from $F(0)$ to $F(n)$, with a loop and with recursion. Then compare the work of the two ways for n above 30. For $n=9$: 0 1 1 2 3 5 8 13 21 34, so $F(9)=34$ and the sum is 88.

$$\sum_{i=0}^{n} F(i) = F(n+2) - 1 \tag{8.3}$$

By hand: $F(10)=55$, $F(11)=89$, *and* $89-1=88$. *Equation (8.3) is proved by induction, the mathematical twin of recursion. Values above 90 are refused: the sum up to* $F(90)$ *is the largest that fits a* `long long`. *Book: Sections 8.4.1 and 8.4.2.*

8.2: the Loop

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
```

Trace Pseudocode 8.1 for $n=2$: the sum collects 0, then 1, and after the loop `prev` is $F(2)=1$, giving $0+1+1=2=F(4)-1$. The C function returns $F(n)$ and gives the sum through `long long *sum`. The recursion is Chapter 5's `fib` with a call counter. Book: Section 8.4.3.

8.2: the Work of Two Methods

```
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
```

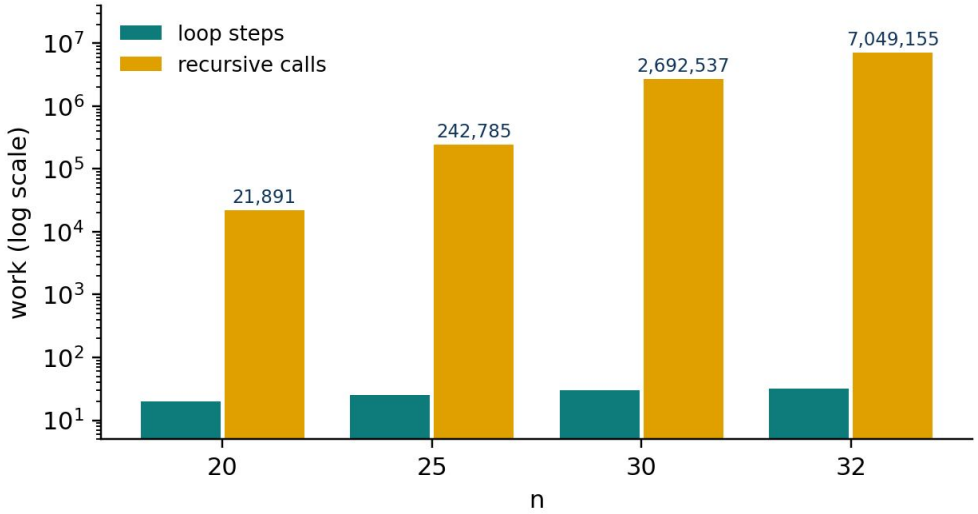


Figure 8.1 The work of the two methods, as counted by `fibonacci.c`: loop steps grow like n, recursive calls like 1.618^n (logarithmic scale).

Figure 8.1: $C(32) = 2 \cdot 3524578 - 1 = 7049155$ *calls against 32 steps. Every increase of 5 multiplies the calls by about 11. Book: Section 8.4.4.*

8.2: Measured, and the Trap

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`	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

The trap: starting at 1, 1 prints ten true Fibonacci numbers, shifted: $F(9) = 55$, sum 143. The given sample exists to catch it.

Table 8.3: the optimizer makes C five times faster but cannot change the growth; Python is about seventy times slower; the loop beats them all. A better algorithm beats a faster language. With `int` the sum overflows at $n = 45$. Book: Sections 8.4.6 and 8.4.7.

Problem 8.3: Palindromes

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`. The words are in lowercase letters without spaces. Optionally, decide it also with recursion or bit operations.

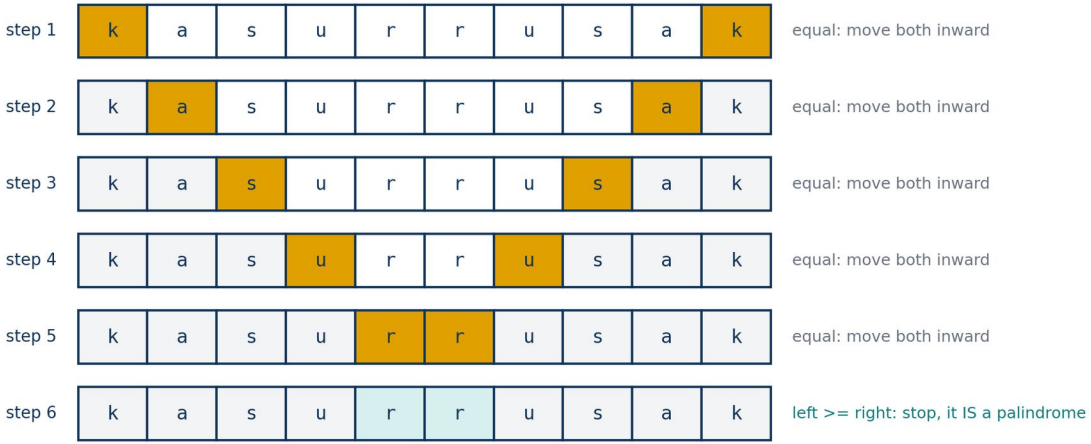


Figure 8.2 The two-pointer scan on `kasurrusak`: the amber letters are compared, the gray ones are done; after five equal pairs the pointers cross and the word is a palindrome.

Figure 8.2: two fingers on `kasurrusak`; five equal pairs, the fingers cross, a palindrome. For `laundry`, `l` against `y` decides at once. One letter has no pair and is a palindrome. At most $\lfloor n/2 \rfloor$ comparisons. Book: Sections 8.5.1 and 8.5.2.

8.3: Three Ways That Must Agree

Listing 8.2 From `palindrome.c`: the two-pointer scan and its recursive twin.

```c
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);
}
```

Listing 8.2: the scan is Chapter 7's reversal walk with a comparison instead of a swap. The third way ORs together `x ^ y` of every pair: 0 at the end means a palindrome, with no `if` in the loop. Read with `scanf("%99s", word)`: no `&`, and a width. Book: Sections 8.5.3 and 8.5.4.

8.3: the Trap `n - i`

Listing 8.3 From `wrong_palindrome.c`: the partner index is `n - i` instead of `n - 1 - i`.

```
    for (size_t i = 0; i < n / 2; i++)
        if (s[i] != s[n - i])            /* WRONG: s[n] is the '\0' */
            return 0;
```

Every palindrome of two or more letters is rejected: `s[0]` is compared with `s[n]`, the terminating `'\0'`. Only `x` slips through.

In C it does not crash, because `s[n]` is inside the array: even the sanitizer is silent. Python without the explicit zero would raise `IndexError`.

> **CHECK YOURSELF.** Accept “Kasur rusak” and “Never odd or even”. *Answer:* skip non-letters with `isalpha`, compare `tolower` of both; and read by lines, not with `%99s`, which stops at the first space.

Listing 8.3: the partner of `i` is `n - 1 - i`. Book: Section 8.5.6.

Problem 8.4: the Determinant

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.

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 0 & 0 \end{pmatrix} \tag{8.4}$$

Why a laundry cares: three receipts for detergent, softener and starch, Rp 60,000, Rp 72,000 and Rp 20,000, with the quantities of Equation (8.4). The prices can be recovered exactly when A is invertible. A repeated receipt gives two equal rows and determinant 0. Book: Section 8.6.1.

8.4: Expansion Along the First Row

$$\det A = a(ei - fh) - b(di - fg) + c(dh - eg) \tag{8.5}$$

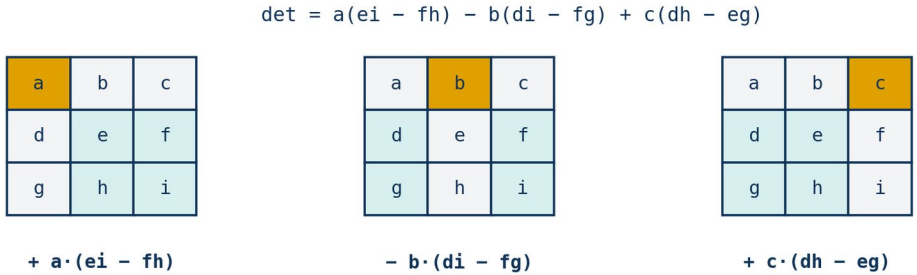


Figure 8.3 Expansion along the first row: each first-row entry (amber) multiplies the 2×2 determinant of the cells left when its row and column are removed (teal), with the signs $+, -, +$.

Figure 8.3: the entries are a, b, c; d, e, f; g, h, i row by row. The minus on the middle term is the part that is easy to forget. Book: Section 8.6.2.

8.4: Worked by Hand, and in C

Rows $(2,-3,1)$, $(2,0,-1)$, $(1,4,5)$: $2\cdot4+3\cdot11+1\cdot8=8+33+8=49$. Invertible.

Equation (8.4): $2\cdot0-1\cdot(-2)+1\cdot(-3)=-1$. Not zero: the prices can be recovered.

Listing 8.4 From `determinant.c`: Equation (8.5) transcribed with `m[0][0]` for a, `m[0][1]` for b, and so on.

```c
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]);
}
```

Listing 8.4 uses `long long`, because products of three entries grow quickly; `read_matrix` reads with two nested loops and returns 0 when the input runs out. Book: Sections 8.6.3 and 8.6.4.

8.4: the Sign Trap

The trap: all three terms added. For the identity matrix it still gives 1, because the middle term is multiplied by $b = 0$.

Two equal rows must give 0; the wrong program says 24. The hand example gives -17 instead of 49; the receipts -5 instead of -1.

> **CHECK YOURSELF.** Rows $(1, 2, 3)$, $(2, 4, 6)$, $(0, 1, 5)$? *Answer:* zero, because the second row is twice the first: $1 \cdot 14 - 2 \cdot 10 + 3 \cdot 2 = 0$.

The test everybody tries first, the identity, cannot catch this bug; choose tests in which every entry matters. Book: Section 8.6.6.

Problem 8.5: Finding a Receipt

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.

Listing 8.5 From `find_receipt.c`: recursive binary search (the book's version also prints each call, indented by its depth).

```c
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;
    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);
}
```

Listing 8.5: two base cases, an empty part and a hit; every recursive call works on a strictly smaller part. Book: Sections 8.7.1–8.7.3.

8.5: Tracing the Calls

```
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
```

Twelve receipts from 1003 to 1066. 1047 is found in two calls; 1003 in three; 1050, absent, after four.

The trap: recursing on `mid` instead of `mid - 1` or `mid + 1`. With a two-element part and the key on the right, the part never shrinks: stack overflow.

The stack is as deep as the number of calls, at most $\lfloor \log_2 n \rfloor + 1$*: 20 frames for a million receipts.* `fib` *was dangerous for its total calls, not its depth. Book: Sections 8.7.3 and 8.7.4.*

Problem 8.6: Merging Two Shifts

> **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.

Two piles face up, smallest on top: take the smaller top receipt, repeat; when one pile is empty, put the rest of the other on the result.

Morning 12, 28, 30, 45, 136.5 and evening 18, 30, 56, 70 (thousands) give 12, 18, 28, 30, 30, 45, 56, 70, 136.5.

Work the merge on the board with two piles of cards before showing the code. Book: Sections 8.8.1 and 8.8.2.

8.6: Three Walking Pointers

Listing 8.6 From `merge_shifts.c`: merging two sorted arrays with three walking pointers.

```c
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);
}
```

$$n_a + n_b - 1 \tag{8.6}$$

Listing 8.6: `<=` *takes the morning bill on a tie, so the merge is stable;* `po - out` *counts the result.* $n_a + n_b$ *steps and at most Equation (8.6) comparisons. This step is the heart of merge sort.* ***The trap:*** *only the first loop; bills are lost unless both piles end together. Book: Sections 8.8.2–8.8.4.*

The Problems Side by Side

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` at 0; whole-number average	positive values only
8.2 Fibonacci	loop n; recursion $2F(n+1)-1$	starting at 1, 1	the given sample
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
8.5 Receipt	at most $\lfloor \log_2 n \rfloor + 1$ calls	recursing on `mid`	a two-element part
8.6 Merge	$n_a + n_b$ steps	forgetting the leftovers	an empty array

Table 8.4: in every case the exposing test is an edge: a boundary, an empty input, a repeated value, a sample worked by hand. Kernighan and Pike call it testing at the boundaries. Book: Section 8.9.

Common Mistakes

Changing given code; **casting the result** instead of an operand; **a minimum starting at 0**; **signed and unsigned mixed**.

Skipping the sample; **recursion on large inputs without counting the calls**.

An off-by-one partner index; **`&` on an array in `scanf`**, and no width; **weak test matrices**.

A recursion that does not shrink; **forgetting the rest after a merge**.

Every item is one of the six traps in general form. Book: Section 8.10.

Summary: Five Things to Remember

1. Eight steps; the hand-worked example decides what “correct” means (Table 8.2).

2. A better algorithm beats a faster language: n loop steps against $2\,F(n+1)-1$ calls (Figure 8.1).

3. Two pointers from both ends decide a palindrome; the partner of `i` is `n - 1 - i` (Figure 8.2).

4. Transcribe a formula exactly (Equation (8.5)); the identity matrix cannot catch a sign error (Figure 8.3).

5. Every recursive call must shrink its part; a merge must copy the leftovers. Test the edges.

Functions, arrays and pointers together solve problems that none of them solves alone. Book: Section 8.11.

Next Time

Chapter 9 — Characters, Strings and Formatted I/O. Text as arrays of `char`, reading whole lines, and the full power of `printf` and `scanf`.

The palindrome checker learns to read “Kasur rusak” as a line.

Before then: Exercises 8.1, 8.3, 8.5 and 8.6.

Exercise 8.5 uses Cramer’s rule to recover the three prices from Mbak Rina’s receipts; the book’s answer gives detergent Rp 20,000 and softener Rp 12,000. Book: Sections 8.12 and 8.13.

Lecture 9 — Characters, Strings and Formatted I/O

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 9

Lecturer: Irfan Subakti — yifana@gmail.com

Today the cashier learns to handle names: how C stores text, how to classify and change characters, how to line up output, how to read a line safely, and cashier v0.7. Book: Chapter 9, Sections 9.1–9.13.

What This Meeting Covers

1. Text is numbers — a character is a small integer; a string is a `char` array that ends with `'\0'`.

2. One character at a time — `<ctype.h>`: classifying, cleaning a phone number, tidying a name.

3. The string library and its sharp edges — copy, join, compare, search, cut; one safety rule for each.

4. The full dial of `printf` — flags, width, precision, type; a receipt that lines up.

5. Reading text safely, and cashier v0.7 — `fgets`, a line too long, words cut from a line, a read–evaluate–print loop.

Every program exists in C and in Python and both print exactly the same output. Book: Chapter 9.

Where We Left Off: Mbak Rina's Names

By Chapter 7 the cashier computes bills, keeps a day's orders in arrays and returns several results through pointers; Chapter 8 used those tools on complete problems.

But orders are numbered 1, 2, 3, and Mbak Rina keeps a paper notebook to remember which number is which customer.

The notebook shows "andi", "ANDI PRATAMA", "Andi Pratama" and "andi p." for one person, and phone numbers in three different shapes.

Receipts have wandering columns: a long name pushes the price to the right.

Every one of these is a problem of text: tidying letters, keeping digits, finding one text inside another, telling `printf` how many columns to use, cutting a typed line into words. Book: Section 9.1.

A Character Is a Small Number

C stores the letter `A` as the number agreed for it: in **ASCII**, `'A'` is 65, `'a'` is 97, `'0'` is 48, the space is 32.

The digits are consecutive, so a digit's value is `c - '0'`: `'7' - '0'` is $55 - 48 = 7$.

Each small letter lies 32 places above its capital: `'a' - 'A'` is $97 - 65 = 32$.

Letters outside English, such as "Chloé", take more than one byte in UTF-8; this chapter works with ASCII, which covers what the laundry types.

A `char` can be added, compared and subtracted like any other integer. That is all the "magic" behind case conversion. Book: Section 9.2.1.

A String Ends With a Zero

> **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*, such as `"Andi"`, is stored by the compiler with the terminator already added. The *length* of a string is the number of characters before the terminator.

The terminator is the whole trick: it marks where the text ends, so no length is stored anywhere.

C has no built-in string type that grows by itself as Python's does, but the idea behind C strings is small enough to understand completely today. Book: Section 9.2.2.

Two Strings in Memory

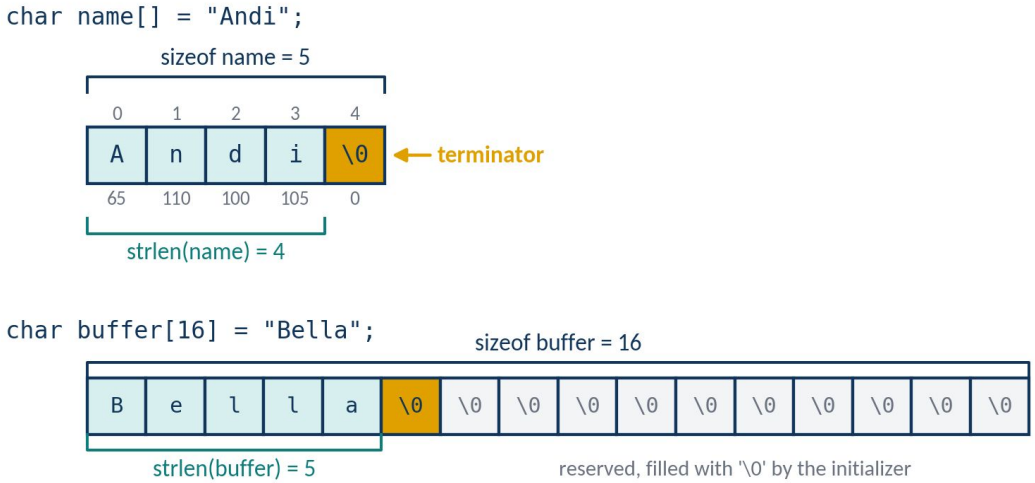


Figure 9.1 Two strings in memory: `name` holds four letters and the terminator in exactly five bytes, while `buffer` reserves sixteen bytes of which the string uses six. The grey numbers under `name` are the ASCII codes.

Figure 9.1: `char name[] = "Andi";` makes an array just large enough, five bytes; `char buffer[16] = "Bella";` reserves sixteen and fills the rest with zeros. Book: Section 9.2.2.

Bytes Versus Characters

`sizeof` — how many bytes the array occupies; the compiler knows it from the declaration.

`strlen` — how long the text is; a loop that counts characters until it meets `'\0'`.

$$\text{sizeof name} = n + 1 \,, \text{strlen}\backslash(\text{name}\backslash) = n \tag{9.1}$$

Equation (9.1) holds for an array declared from a literal of n characters. Forgetting the "+ 1" is the most common way to make an array one byte too small.

For `name` the answers are 5 and 4; for `buffer` they are 16 and 5. Book: Section 9.2.2.

Every String Function Is a Loop

Listing 9.1 From `string_basics.c`: `strlen` written by hand; the rest of the program prints the arrays of Figure 9.1 byte by byte.

```c
static size_t my_strlen(const char *s)
{
    size_t n = 0;
    while (s[n] != '\0')
        n++;
    return n;
}
```

Listing 9.1 is the pattern of every function in `<string.h>`: walk until the terminator. Book: Section 9.2.2.

What `string_basics` 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
```

`name[0] = 'a'` is allowed: the array holds its **own copy** of the letters.

`buffer[3] = '\0'` erases nothing, yet every function now stops at index 3 and sees `Bel`.

The byte dump `65 110 100 105 0` is the figure in numbers. Book: Section 9.2.2.

Arrays Versus Pointers to Literals

> **WARNING.** `const char *shop = "Profitina Laundry";` does not make an array; it makes a *pointer* to a literal stored in memory that may not be written. `shop[0] = 'p';` is undefined behavior and usually crashes. Declare such pointers `const`, and the compiler refuses the assignment. Text you will change goes in an array.

In Python, strings know their length and cannot be changed in place; `len("Andi")` is 4 and a “change” builds a new string (`string_basics.py` in the book).

Chapter 7’s pointers meet Chapter 9’s text here: an array name and a pointer look alike in an expression but are different things. Book: Sections 9.2.2–9.2.3.

One Character at a Time: `<ctype.h>`

Classify — `isalpha`, `isdigit`, `isspace`, `isupper`, `islower`, `isalnum`, `ispunct`.

Convert — `toupper`, `tolower`; a character that is not a letter comes back unchanged.

The one rule — pass `(unsigned char)c`; a negative `char` is undefined behavior.

Three tasks for the laundry: count kinds of characters, keep the digits of a phone number (`digits_only`), and write a name in title case (`tidy_name`).

`valid_mobile` then checks that the cleaned number starts with `08` and has 10 to 13 digits. Book: Section 9.3.

Worked Example 1: Tidying a Name

Listing 9.2 From `char_classes.c`: `tidy_name` trims, squeezes inner spaces to one and writes each word in title case.

```c
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';
}
```

Listing 9.2: the state variable `new_word` remembers whether the previous character was a space. Book: Section 9.3.

Tracing `"  aNDi  pratama "`

Two leading spaces — each sets `new_word = 1` and copies nothing; `n` is still 0.

`a` — a word starts, but `n` is 0, so no space is written; `A` is written in capitals. `n` becomes 1.

`NDi` — inside a word: written as `ndi`. Result so far `Andi`.

Three spaces — only `new_word = 1`. Then **`p`**: one space is written, then `P`; the rest gives `ratama`.

The trailing space — no character follows it, so it produces nothing. Result: `Andi Pratama`.

"What have I just seen, and what does that mean for the current character?" is the heart of all text processing. In Python the whole function is `" ".join(w.capitalize() for w in src.split())`, up to the size limit. Book: Section 9.3.

The Same Ideas in Python

Characters — `ord("A")` is 65 and `chr(97)` is `"a"`; a one-letter string plays the role of a `char`.

Classifying — `c.isalpha()`, `c.isdigit()`, `c.isspace()` behave like their C cousins for ASCII text.

Cleaning a phone number — `"".join(c for c in phone if c.isdigit())` keeps only the digits.

Tidying a name — `src.split()` with no argument splits at any run of white space and drops empty pieces: the trimming and squeezing of the C loop in one call.

Searching — `part.lower() in name.lower()` is the case-insensitive `strstr` of the cashier.

Python builds new strings instead of filling arrays, so there is no terminator to forget and no size to pass; the price is that every change makes a copy. The outputs of the two languages are identical, line for line. Book: Section 9.3.

The String Library: Six Families

Length — `strlen`. **Copy** — `strcpy`, `strncpy`; prefer `snprintf(dst, sizeof dst, "%s", src)`.

Join — `strcat`; bounded joining with `snprintf` at the current end.

Compare — `strcmp(a, b)` is negative, zero or positive; *equal* means `== 0`.

Search — `strchr` for a character, `strstr` for a text; both return `NULL` when nothing is found.

Cut — `strtok` splits in place, changes its input and remembers its position in a hidden variable.

None of these functions checks anything. The rule for each: give every copy a size, test searches for `NULL`, compare with `== 0`, and remember what `strtok` does to its input. `string_library.c` and `string_library.py` show them side by side. Book: Section 9.4.

The Sharp Edge: One Byte Too Many

> **WARNING.** `char code[7]; strcpy(code, "PL-0042");` writes 8 bytes into 7: seven letters and the terminator. `strcpy` does not know the size of `code`, so it writes past the end into whatever lies next in memory. The program may even appear to work.

The fix: size by Equation (9.1), or copy with `snprintf(code, sizeof code, "%s", text)`, which never writes more than `sizeof code` bytes and always terminates.

Compiling with `-fsanitize=address,undefined` turns the silent overflow into a clear report.

This is the most famous source of bugs in the history of C; one habit removes it. Book: Section 9.4.

The Full Dial of `printf`

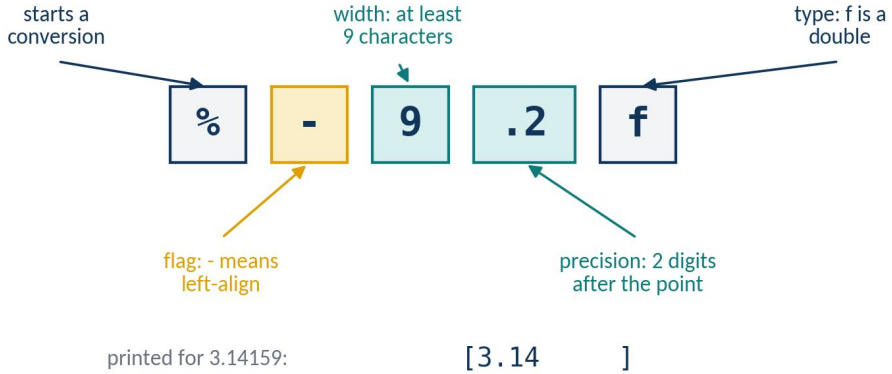


Figure 9.2 The parts of the conversion specification `%-9.2f`: the flag `-` left-aligns, the width 9 is the minimum number of characters, the precision 2 gives the digits after the decimal point, and the type `f` says that a `double` is printed.

Figure 9.2 takes one specification apart. Book: Section 9.5.

The Conversion Specification

$$\%\,[\,\text{flags}\,][\,\text{width}\,][\,.\,\text{precision}\,]\,\text{type} \tag{9.2}$$

Width — a minimum: short values are padded (on the left unless `-`); long values are never cut.

Precision — digits after the point for `f`; significant digits for `g`; the *maximum* characters for `s`.

Flags — `-` left-align, `0` pad with zeros, `+` always a sign, space for a blank before positives.

`*` — the width comes from the argument list: `printf("%*s", w, text)`.

Equation (9.2) is the whole grammar. `format_dial.c` (`format_dial.c`) turns one knob at a time and prints every field between brackets; Python's `%` operator reads the same specifications (`format_dial.py`). Book: Section 9.5.

Worked Example 2: Read the Dial

EXAMPLE. What does `printf("[%-6.3s][%06.1f][%+5d]\n", "Profitina", 3.14159, 42);` print?

`%-6.3s` — at most 3 characters, `Pro`, left-aligned in 6: `[Pro   ]`.

`%06.1f` — one decimal, `3.1`, padded with zeros to width 6: `[0003.1]`.

`%+5d` — a sign always, `+42`, right-aligned in 5: `[  +42]`.

```
[Pro   ][0003.1][  +42]
```

Work it on the board before running it; this is Exercise 9.1 of the book. Book: Sections 9.5 and 9.14.

A Receipt That Lines Up

Fixed widths per column: `%-20s` for the name, `%-10s` for the service, `%6.1f` for the weight, a right-aligned amount.

`rupiah` writes 1250000 as `1.250.000`: `snprintf` makes the digits, then a dot goes before digit `i` whenever `len - i` is a multiple of three and `i` is not 0.

`%*s` centers the shop's name: the width is computed from the text's length.

Money stays in **integers**. `%.2f` of 2.675 prints 2.67, because 2.675 is not exactly representable in binary.

`receipt.c` and `receipt.py` print the same receipt; `rupiah` and the centering helper are used for the rest of the book. Book: Section 9.6.

Reading Text Safely

`scanf("%s", ...)` stops at the first space and knows nothing of the array's size; `%31s` fixes only the second problem.

Step 1 — read one whole line with `fgets(buf, size, stdin)`: never more than `size` bytes, always terminated, `NULL` at the end of input.

Step 2 — look at what arrived: a final `'\n'` means the line fit; remove it (and a `'\r'` from Windows files). A full array without newline means the line was too long; read and discard the rest.

Step 3 — only then cut the line into words and decide what they mean.

Separate reading from understanding. The one-liner `buf[strcspn(buf, "\r\n")] = '\0';` *removes the line ending. Book: Section 9.7.*

`read_line` and `next_word`

`read_line` returns 1 for a good line, 0 at the end of input, −1 for a line that was too long.

With 80 bytes, a line may have 79 characters: when the array is full, `getchar()` peeks; a newline means the line fit exactly.

`next_word(&p, out, size)` skips spaces, copies one word (cut to `size - 1`, but always consuming the whole word) and moves `p` past it.

The parameter `const char **p` is Chapter 7's technique for a function that changes the caller's pointer.

Why not `sscanf` with `%15s`? It reads 15 letters of a 20-letter word and leaves 5 to be taken as the next word, silently changing the line's meaning. Book: Section 9.7.

Cashier v0.7: a Read–Evaluate–Print Loop

Prompt `>`, read a line, carry out the command, print the result, start again.

`add <service> <kg> <name>` — service `fold`, `wash-iron` or `iron`; weight like `3` or `3.5`; a name of several words in any case.

`find <part of a name>` — a case-insensitive `strstr` over tidy names. **`list`**, **`help`**, **`quit`**.

Every bad line gets a polite message, never a crash.

`cashier_v7.c` and `cashier_v7.py` are the full programs, with the helpers of this chapter. Book: Section 9.8.

A Morning at the Counter

```
> 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 iron 4.25 citra
  weight must look like 3 or 3.5 (0.1 to 50.0 kg)
> add fold 1.5
  the customer's name is missing
> find ari
  #3 Citra Dewi Lestari, iron, 4.0 kg = Rp 24.000
```

From the run on `data/session9.txt` with `--echo`; C and Python print the same. `find ari` finds "Lestari": search by part of a name. Book: Section 9.9.

The Table and the Long Line

```
> 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
> 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.
```

The columns no longer wander, and the long line is refused as a whole instead of being read as two commands. The test runner compares both programs' output and runs C under the sanitizers. Book: Section 9.9.

Making It Better, Using It for Real

Three parallel arrays — a new field means a new array everywhere; Chapter 10 groups an order into one `struct`.

Fixed size, lost at power-off — Chapter 10 grows with `malloc`; Chapter 11 saves to a file.

For real — a tailor, a clinic or a fruit and durian orchard selling by chat can reuse `tidy_name`, `digits_only` and the receipt layout.

Names and phone numbers are personal data (UU PDP, Law 27 of 2022): keep only what the business needs, and print only the last four digits.

State the limits (`NAME_LEN` 32, lines of 79 characters) in the messages. UTF-8 names pass through safely, but `toupper` leaves é alone and lengths count bytes. Book: Sections 9.10–9.11.

Common Mistakes

The terminator's byte forgotten; `==` **on strings** (compares addresses); `if (strcmp(a, b))` read as "equal".

`sizeof` **on a pointer** parameter gives 8, not the caller's array size: pass the size separately.

A plain `char` **to** `<ctype.h>`; **writing into a string literal**; **a bare** `%s` without width.

`fgets` **keeps the newline**, so `"quit\n"` is not `"quit"`; `strtok` on empty fields; **money in** `double`.

Each mistake in the book comes with its fix. Book: Section 9.12.

Summary: Five Things to Remember

1. A character is a small integer; a digit's value is `c - '0'`.

2. A C string is a `char` array ending in `'\0'`; `sizeof` counts bytes, `strlen` characters (Equation (9.1)).

3. `<ctype.h>` takes `unsigned char`; `<string.h>` checks nothing, so give every copy a size and test every search for `NULL`.

4. `%[flags][width][.precision]type` (Equation (9.2)) lines up every column; money stays in integers.

5. Read a whole line with `fgets`, handle a line too long, strip the line ending, then cut it into words.

Cashier v0.7 puts all five together, in C and Python with identical output. Book: Section 9.13.

Next Time

Chapter 10 — Structures, Enums and a First Linked List. One order, one name: `struct`, `enum`, `typedef`, `malloc` and `free`.

The three parallel arrays of v0.7 become one structure, and the order book grows one order at a time.

Before then: work Exercises 9.1–9.4, especially the phone column of Exercise 9.4, and list which lines a `struct` would make unnecessary.

The `word_stats` program of the solutions (`word_stats.c`, `word_stats.py`) is a good warm-up. Book: Sections 9.14–9.15.

Lecture 10 — Structures, Enums and a First Linked List

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 10

Lecturer: Irfan Subakti — yifana@gmail.com

Today an order becomes one thing with one name, memory is requested while the program runs, and the orders in the shop are chained into a linked list. Book: Chapter 10, Sections 10.1–10.12.

What This Meeting Covers

1. **One record, one name** — declaring a `struct`, the dot operator, initializers, the layout in memory with padding.
2. **Structures and functions** — by value and by pointer, the `->` operator, an array of structures.
3. **`enum` and `typedef`** — names instead of numbers, and a service table indexed by name.
4. **Memory on demand** — `malloc`, `free` and their three rules.
5. **The linked list, and cashier v0.8** — walking, adding, removing, freeing; orders that move from waiting to washing to ready.

Every program is in C and Python with identical output, and the C runs clean under AddressSanitizer. Book: Chapter 10.

Where We Left Off: Three Arrays That Must Agree

Cashier v0.7 keeps an order in three **parallel arrays**: `names[i]`, `service_of[i]`, `tenths_of[i]`. Only the index `i` ties them together.

A hurried sorting loop swaps the names but not the weights: Andi's name stands next to Budi's bill. Nothing crashes; the table is simply wrong.

The arrays have a fixed size of 50: space wasted on Tuesday, the 51st customer turned away after a long weekend.

An order leaving from the middle means shifting every later order, in all three arrays.

C's answers: the structure for the first problem; memory requested one order at a time and chained with pointers, the linked list, for the second and third. Book: Section 10.1.

Declaring a Structure

Listing 10.1 The declaration of `struct Order`: four members of different types under one name.

```
struct Order {
    int id;
    char customer[22];
    int tenths;
    long long amount;
};
```

Listing 10.1 declares a *type*; no variable exists yet. `struct Order a;` makes one, and `a.id`, `a.customer`, `a.amount` reach its members.

Initialize in order, `{2, "Budi Santoso", 20, 20000}`, or by name, `{.customer = "Citra", .id = 3}`; missing members become zero.

The semicolon after the closing brace is required; forgetting it gives a confusing error on the next line. Book: Section 10.2.1.

What a Structure Is

> **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`.

Assignment `b = a;` copies the whole structure, arrays inside included.

There is no `==` for structures: compare member by member.

In Python the same record is a data class, `@dataclass class Order`, whose objects are always handled by reference. Book: Sections 10.2.1 and 10.2.3.

A Structure in Memory

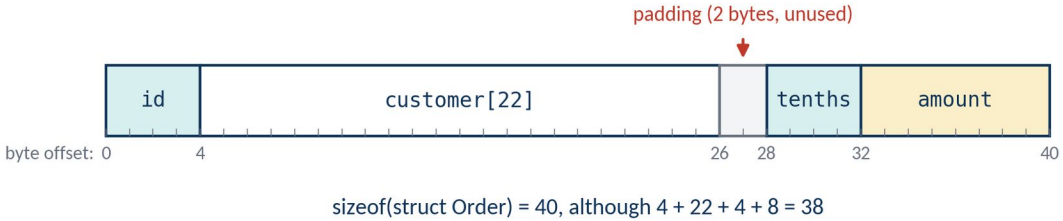


Figure 10.1 The layout of `struct Order` in memory: four members at offsets 0, 4, 28 and 32, and two bytes of padding after `customer` so that `tenths` starts at a multiple of four.

Figure 10.1: the members add up to 38 bytes, yet `sizeof(struct Order)` is 40. Book: Section 10.2.2.

Worked Example 1: Where Each Member Starts

$$o' = a \cdot \left\lceil \frac{o}{a} \right\rceil \tag{10.1}$$

Equation (10.1): with the next free byte at o and a member of alignment a, the member starts at the first multiple of a not below o.

`id` — offset 0, bytes 0–3. `customer` — 22 bytes, offsets 4–25.

`tenths` — $o = 26$, $a = 4$: $o' = 4 \cdot \lceil 6.5 \rceil = 28$. Two bytes of padding.

`amount` — $o = 32$, $a = 8$: $o' = 32$, no padding. Ends at byte 39; the size, 40, is a multiple of 8.

The whole size is rounded up to a multiple of the largest alignment, so every element of an array of structures is aligned too. Measure with `sizeof` and `offsetof`; never assume. Book: Section 10.2.2.

The Same Record in Python

`@dataclass` writes the constructor, the printing and the `==` of a record class for us: `Order(1, "Andi Pratama", 35, 24500)`.

`b = a` in Python does **not** copy: both names refer to one object. A copy is `dataclasses.replace(a)` or `copy.copy(a)`.

The `==` of a data class compares field by field, which is what C makes us write by hand.

`ctypes.Structure` describes the C layout, so `order_struct.py` can print the same sizes and offsets, 40 bytes with `tenths` at 28.

The difference in copying is the one to remember: in C, assignment copies a structure; in Python, assignment shares an object. Book: Section 10.2.3.

Structures and Functions

By value — the function receives a copy; the caller's structure is never changed. A 1,000-byte structure costs 1,000 bytes per call.

By pointer — the function receives 8 bytes, whatever the size, and can change the caller's structure through it.

`o->amount` means exactly `(*o).amount`; `*o.amount` means `*(o.amount)` and does not compile.

The normal way — `const struct Order *o` to read: no copy, and the compiler enforces the promise not to change.

`order_functions.c` uses `struct Order day[4]`, an array of structures: `day[i]` is the whole order, and `qsort` moves each order as one piece, so the mix-up of Section 10.1 cannot happen. Book: Section 10.3.

`enum` and `typedef`

Listing 10.2 The service enumeration: `FOLD` is 0, `WASH_IRON` 1, `IRON_ONLY` 2, and `N_SERVICES` counts them.

```
typedef enum { FOLD, WASH_IRON, IRON_ONLY, N_SERVICES } Service;
```

Listing 10.2: `if (service == WASH_IRON)` reads better than `if (service == 2)`; `N_SERVICES` sizes tables and bounds loops.

`typedef` gives a type a second, shorter name; it creates no new type.

Designated initializers for arrays — `[WASH_IRON] = {"wash-iron", 10000, 48}` puts that entry at index 1, in whatever order the lines are written.

> **WARNING.** An `enum` variable can still hold any `int`. Check values from input or files before using them as an index.

`service_enum.c` builds the service table this way, and a second enumeration `Status` names the life of an order; Python uses `IntEnum` and a dictionary. Book: Section 10.4.

Memory on Demand: `malloc` and `free`

Globals live for the whole run; locals live while their function runs (the stack). Neither lets the program decide *while running* how many orders to keep.

`malloc(n)` reserves `n` bytes on the **heap** and returns a pointer, or `NULL`. The contents are not initialized.

`free(p)` gives the block back; `free(NULL)` does nothing.

The idiom: `Node *n = malloc(sizeof *n);` then test `n == NULL`. `sizeof *n` stays right if the type of `n` changes.

The block lives until `free`, independently of the function that created it: a function can create an order, return a pointer, and the order survives. Book: Section 10.5.

The Three Rules of `malloc`

1. **Check for `NULL`.** Writing through a null pointer crashes the program.
2. **Free every block exactly once.** Never freed is a *memory leak*; freed twice corrupts the heap.
3. **Never use a block after freeing it.** The pointer still holds the old address, but the memory may belong to something else.

> **KEY INSIGHT.** *Use after free* and *NULL pointer dereference* are both on MITRE's 2025 list of the most dangerous software weaknesses, and both are exactly these rules broken. `-fsanitize=address` catches all three.

Python has no `malloc` and no `free`: the interpreter frees an object when nothing refers to it any more (garbage collection, in CPython mostly by reference counting). Book: Section 10.5.

A Structure That Points to Itself

Listing 10.3 A self-referential structure: each node holds one order and the address of the next node.

```
typedef struct Node {
    int id;
    char customer[24];
    long amount;
    struct Node *next;
} Node;
```

Listing 10.3: following `next` from node to node is a treasure hunt where every clue says where the next is hidden. The last node's `next` is `NULL`; the first node's address is kept in `head`, which is `NULL` for an empty list.

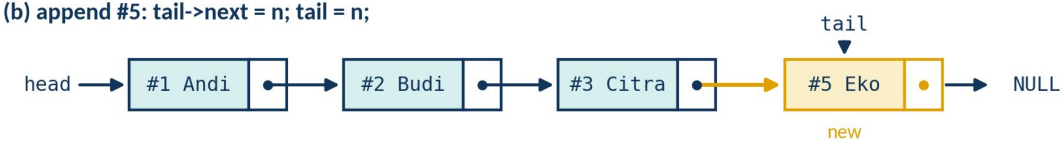
Inside the braces the short name `Node` does not exist yet, so the member is written `struct Node *next`. Book: Section 10.6.1.

A Linked List of Orders

(a) the list



(b) append #5: tail->next = n; tail = n;



(c) remove #2: prev->next = cur->next; free(cur);

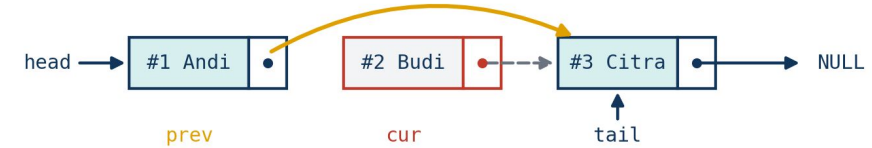


Figure 10.2 A singly linked list of orders: (a) `head` points at the first node and every `next` at the following one, the last at `NULL`; (b) appending changes the last node's `next` and the `tail` pointer; (c) removing a node from the middle bridges over it with one assignment.

Figure 10.2 is the picture to keep in mind for every list operation today. Book: Section 10.6.1.

Array Versus Linked List

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

Table 10.1: the list gives up direct access and gains growth, cheap insertion and cheap removal. For a laundry queue, where orders arrive at the end and leave from anywhere, the list wins. Book: Section 10.6.1.

Walking and Adding

The traversal — `for (const Node *p = head; p != NULL; p = p->next)`; for an empty list the body runs zero times, no special case.

Printing, counting, adding up and searching are all this loop with a different body.

Add at the front — `n->next = head;` *then* `head = n;`. The other order loses the whole list.

Add at the end — keep a `tail` pointer: `tail->next = n; tail = n;`; an empty list makes `n` the head as well.

`head`, `tail` and `count` live together in a structure `List`, so a function receives the whole list through one pointer. Book: Sections 10.6.2–10.6.3.

Removing a Node

Listing 10.4 From `order_list.c`: `remove_id`, the two-pointer walk with its three cases.

```c
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 10.4: `prev` stays one step behind `cur`. First node: `head` moves. Last node: `tail` moves back. Only after unlinking is the node freed. Book: Section 10.6.4.

Worked Example 2: Removing #2 by Hand

EXAMPLE. The list is Dewi (#4), Andi (#1), Budi (#2), Citra (#3). Remove #2.

Start — `prev` is `NULL`, `cur` is Dewi. Id 4, not 2: move on.

Step 2 — `prev` is Dewi, `cur` is Andi. Id 1: move on.

Step 3 — `prev` is Andi, `cur` is Budi. Id 2: stop.

Unlink — `prev` is not `NULL`, so `prev->next = cur->next`: Andi now points at Citra. `tail` is Citra, not Budi, so it stays.

Free — Budi's node is freed; `count` drops to 3. The list reads Dewi, Andi, Citra.

Removing a found node costs one assignment however long the list; finding it may walk all n nodes, and an array would then still shift the rest. Book: Section 10.6.4.

Freeing the Whole List

> **WARNING.** `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.

Listing 10.5 From `order_list.c`: `free_all` saves the successor before freeing.

```c
Node *p = l->head;
while (p != NULL) {
    Node *next = p->next;            /* read next BEFORE free */
    free(p);
    p = next;
}
```

Listing 10.5 is the loop to write every time; the leak check of the test runner confirms nothing was left behind. Book: Section 10.6.5.

The List at Work

```
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
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
head -> [#1 Andi] -> [#5 Eko] -> NULL   count=2
```

Selected lines of `order_list`: first node, middle node, last node removed. Appending Eko afterwards lands right after Andi, the proof that `tail` moved back; had it not, the sanitizer would report a heap-use-after-free. Book: Section 10.6.6.

The Same List in Python

A `Node` class with `id`, `customer`, `amount` and `next`; `None` plays the role of `NULL`.

The walk is the same loop: `p = head`, then `while p is not None: ...; p = p.next`.

`remove_id` keeps `prev` and `cur` exactly as in C, including the three cases for the first node, a middle node and the last node.

There is no `free`: once no variable refers to the removed node, Python reclaims it. There is no `free_all` either; setting `head` and `tail` to `None` is enough.

In everyday Python a `list` or `collections.deque` would do this job; we build the list by hand to see the pointers that C makes visible. `order_list.py` prints exactly what `order_list.c` prints. Book: Section 10.6.6.

Cashier v0.8: One Structure per Order

An order holds `id`, `customer`, `service` (a `Service`), `tenths`, `amount`, `status` (a `Status`) and `next`.

The orders in the shop form one linked list inside a structure `Shop`, with the next order number and the day's takings.

`add` — `malloc`, fill, append; no limit on the number of orders.

`advance <id>` — waiting → washing → ready. **`pickup <id>`** — a ready order is unlinked, paid and freed. **`list`**, **`quit`**.

The text helpers of Chapter 9 are copied in unchanged; Chapter 11 moves them into a file of their own. Book: Section 10.7.

An Order's Life at the Counter

```
> 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 7
  no order #7 in the shop
```

From the run on `data/session10.txt`; C and Python print the same. An order cannot be picked up before it is ready, cannot advance past ready, and an unknown number gets a clear message. Book: Section 10.8.

Making It Better, Using It for Real

Keep a useful order — `insert_sorted` links express orders in before later ones, with the two-pointer walk.

All of a customer's orders — a traversal that does not return early.

Do not lose the day — a power cut still loses everything; Chapter 11 saves after every change.

For real — a motorcycle repair shop: plate, owner, complaint, a `BikeStatus` enumeration; `advance` becomes the mechanic's next step.

A small leak is harmless in a program restarted each morning and fatal in one that runs for months: check every adaptation under the sanitizer. Book: Sections 10.9–10.10.

Common Mistakes

No semicolon after `}` of a structure; **`.` with a pointer or `->` with a structure**.

Large structures by value by accident; **`memcmp` on structures** (padding differs); **assuming 38 is the size**.

Unchecked `malloc`; **freeing, then using**; **losing the list** by assigning `head = n` first.

Forgetting `tail` when removing the last node or adding to an empty list; **trusting an `enum`** from input.

Test removing the last node and then appending: it catches the forgotten `tail` at once. Book: Section 10.11.

Summary: Five Things to Remember

1. A `struct` groups named members; `.` on a structure, `->` through a pointer; assignment copies, comparison is by member.
2. Members are aligned with padding (Equation (10.1), Figure 10.1); `sizeof` and `offsetof` tell the truth.
3. Pass structures by `const` pointer to read and by pointer to change; an array of structures replaces parallel arrays.
4. `malloc` memory lives until `free`: check `NULL`, free once, never use after freeing.
5. A linked list with `head` and `tail` adds at either end in one step and removes with a walk and one assignment (Table 10.1, Figure 10.2).

`enum` names codes, `typedef` names types, and cashier v0.8 uses all of it, clean under AddressSanitizer. Book: Section 10.12.

Next Time

Chapter 11 — File Processing: Data That Survives Power-Off. Streams, text and binary files, CSV, logs, and saving without losing.

Cashier v0.9 loads the list at start and saves it after every change, and the program is split into several files.

Before then: Exercises 10.1–10.6, especially the size of `struct Item` by hand and the in-place reversal of a list in C and Python.

At four o'clock the power goes off; next time the orders survive it. Book: Sections 10.13–10.14.

Lecture 11 — File Processing: Data That Survives Power-Off

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 11

Lecturer: Irfan Subakti — yifana@gmail.com

Today the cashier stops forgetting: streams and files, the modes of `fopen`, the `feof` trap, a real CSV file with mistakes in it, logs, binary records, saving without losing, and cashier v0.9 split into modules. Book: Chapter 11, Sections 11.1–11.14.

What This Meeting Covers

1. **Files and streams** — `FILE *`, `fopen`, `fclose`, buffering, and the six modes.
2. **Text files and the `feof` trap** — `fprintf`, `fscanf`, and why the reading loop tests the read itself.
3. **Real data** — a CSV file read line by line, damaged lines reported with their number.
4. **Logs and binary files** — append mode; fixed-size records reached with `fseek`.
5. **Saving without losing, and cashier v0.9** — write, close, rename; headers and include guards; a simulated power cut.

As always, every program exists in C and in Python with identical output. Book: Chapter 11.

Where We Left Off: Four O’Clock

Cashier v0.8 has tidy names, orders that move from *waiting* to *washing* to *ready*, and a list that grows with the day.

At four o’clock on a busy Saturday it holds thirty orders. Then the power goes off, and all thirty are gone.

RAM keeps its contents only while it has power; variables, arrays and linked lists all live there.

Data that must outlive the program goes to **storage** (SSD, disk, memory card) through **files**.

Writing to a file is easy; writing safely takes more thought. Saving only at `quit` still loses the day, and rewriting after every order can leave a half-written file if the power fails during the rewrite. Book: Section 11.1.

Streams and the `FILE` Pointer

> **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`.

`fopen(name, mode)` returns `NULL` when the file cannot be opened; report it with `perror(name)`.

Every program starts with three open streams: `stdin`, `stdout`, `stderr`.

Streams are **buffered**: output reaches the file in blocks, and the last block only on `fflush` or `fclose`.

The program never looks inside the `FILE` object; it passes the pointer around, the way you show a ticket without reading the fine print. Book: Section 11.2.1.

The Modes of `fopen`

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

Table 11.1: the dangerous mode is `"w"`, which empties an existing file at the moment of opening. Python's `open` uses the same letters, raises `FileNotFoundError` instead of returning `NULL`, and in a `with` block closes the file automatically. Book: Section 11.2.2.

Writing and Reading a Text File

Write — `fprintf(fp, "%d %ld %s\n", id, amount, name)`: exactly `printf`, with a stream in front.

Read — `fscanf(fp, "%d %ld %31[^\n]", &id, &amount, name)`; the scan set reads the rest of the line, spaces included, at most 31 characters.

Put the field that may contain spaces **last** on the line.

The loop — `while (fscanf(...) == 3)`: continue exactly as long as a whole record was read.

`write_read_text.c` writes three orders to `orders.txt` and reads them back, with a total of 68500. Book: Section 11.3.

The `feof` Trap

Listing 11.1 From `feof_trap.c`: the loop `while (!feof(fp))` runs once too often.

```
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);
}
```

Listing 11.1 looks right to almost every beginner. Predict its output for a file of three orders before turning the page. Book: Section 11.4.

Worked Example 1: Four Rounds for Three Orders

```
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)
```

After round 3 the stream stands just before the end, but no read has *failed* yet, so `feof` is still false.

Round 4's `fscanf` fails and returns -1 (`EOF`); the variables keep Citra's values, and her 24000 is added twice.

Only now is `feof` true. The fix: loop on the result of the read, `fscanf`, `fgets`, `fgetc` or `fread`.

`feof` does not look ahead; it reports the end only after a read has tried to go past it. It is useful after the loop, to tell the end of the file from a read error. Book: Section 11.4.

Real Data: a CSV File

CSV, comma-separated values: one record per line, fields separated by commas, a header line naming the columns. Every spreadsheet reads and writes it.

`data/week.csv` has 13 orders typed by the owner, and several mistakes: an unknown service, a missing field, an extra field, an empty line, a weight that is not a number.

A good reader does not crash, does not stop at the first mistake, and does not accept nonsense silently. It reports every damaged line **with its line number** and keeps the good ones.

Treat every file as input from a stranger. Book: Section 11.5.

Reading a Line in Three Steps

1. `fgets` reads the whole line, up to 255 characters.
2. `line[strcspn(line, "\r\n")] = '\0';` cuts off `"\n"` or the Windows `"\r\n"`.
3. `split_csv` writes a terminator over each comma and remembers where each field starts; empty fields are **kept**, and the number of fields is counted.

Each field is then checked: a known service, a weight that `parse_tenths` accepts completely (digits, at most one decimal) and that lies in range.

> **WARNING.** `strtok` skips empty fields, so a missing value shifts every later field one place left and the error goes unnoticed. Use a splitter that keeps empty fields.

`csv_report.c` totals by service and by day; `csv_report.py` does the same with `line.split(",")`. Book: Section 11.5.

Worked Example 2: What the Report Says

```
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
```

Four messages for exactly the four damaged lines; the empty line 11 is simply skipped. Line 13's extra field is `4,5`, a weight typed with a decimal comma, the Indonesian habit, which is why counting fields matters. A missing file prints `No such file or directory` on `stderr` and exits with status 1. Book: Section 11.5.

Append Mode: How Logs Grow

Mode `"a"` sends every write to the end and never empties the file: exactly what a **log** needs.

When something goes wrong, the first question is “what happened, and when?”; the log answers it.

`activity_log.c` counts the lines already there (a missing file has none), appends one numbered entry, and prints the whole log; every run adds one line.

Every server, cash register and bank keeps logs. The Python version opens with `"a"` in the same way. Book: Section 11.6.

Binary Files and Random Access

A text file must be read from the front: line 5,000 starts wherever the 4,999 newlines before it end.

A **binary** file stores the bytes of the data as they are in memory: `fwrite(ptr, size, count, fp)` and `fread(...)` return how many whole objects they moved.

$$\text{offset}(i) = i \cdot s \tag{11.1}$$

With records of s bytes, record i starts at byte Equation (11.1), and `fseek(fp, offset, SEEK_SET)` goes there without reading anything on the way: **random access**.

`fseek(fp, 0, SEEK_END)` followed by `ftell(fp)` gives the file's size in bytes. Book: Section 11.7.

Four Records, One Jump

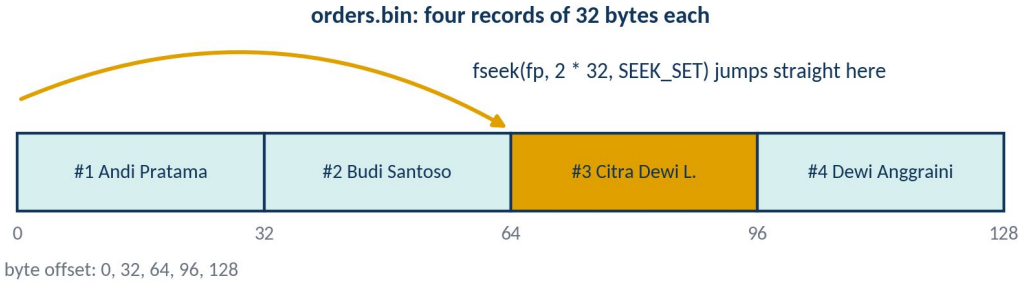


Figure 11.1 A binary file of four 32-byte records: by Equation (11.1), record 2 starts at byte $2 \cdot 32 = 64$, and `fseek` goes there directly.

Figure 11.1: `binary_records.c` writes four records, jumps to record 2, changes it in place with `r+b`, and reads it back; Python does the same with the `struct` module. Book: Section 11.7.

The Price of Binary Files

A binary file is tied to one platform: structure padding and byte order differ between compilers and machines.

Nobody can read it in a text editor or open it in a spreadsheet.

Rule of thumb — text for data that people read or that travels between programs; binary only for fixed-size records on one machine that must be reached fast.

> **WARNING.** A binary file must be opened with `b` in the mode on Windows, or every byte 10 written becomes the two bytes 13 10 and the records shift.

The cashier chooses text: `shop.csv` can be opened in any spreadsheet when something looks wrong. Book: Section 11.7.

Saving Without Losing

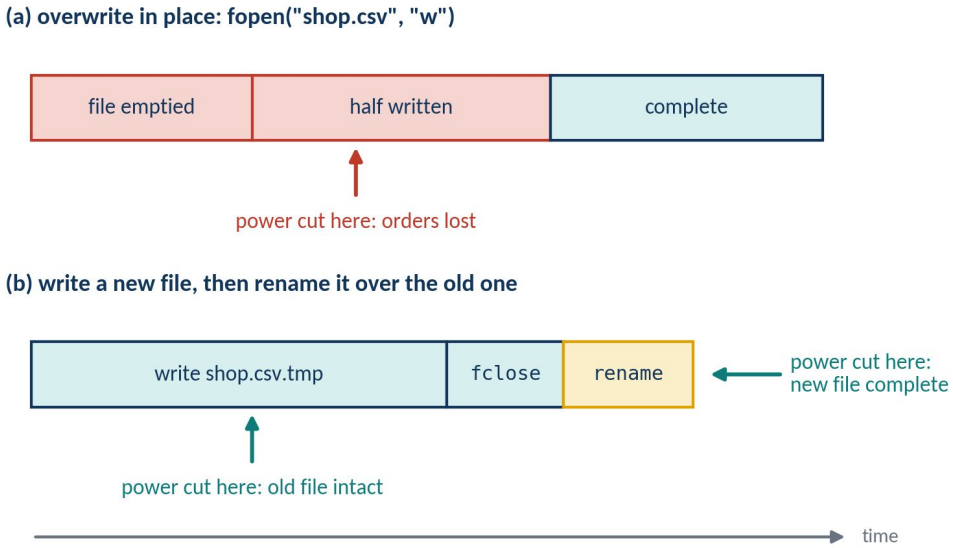


Figure 11.2 Two ways to save: (a) overwriting the file in place leaves it empty or half written during the save; (b) writing a new file and then renaming it over the old one leaves, at every moment, one complete file.

Figure 11.2: in (a), `"w"` empties the file at once, so for a few milliseconds the only copy is empty or half written. Book: Section 11.8.

Write, Close, Rename

1. Write all orders to a *new* file, `shop.csv.tmp`.

2. Close it and check: `ferror(fp)` for any failed write, and the result of `fclose`, which reports the last buffered bytes (a full disk, for example).

3. Only if everything succeeded, `rename` the new file over `shop.csv`.

A power cut during 1 or 2 leaves the old file untouched. On Linux and macOS, POSIX makes `rename` replace the target **atomically**.

Even this trusts the operating system's cache; databases additionally call `fsync` (POSIX) or `FlushFileBuffers` (Windows) and wait for the disk. Book: Section 11.8.

`save_shop` in C

Listing 11.2 From `storage.c`: the end of `save_shop`, which checks the writes and renames the temporary file.

```
    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;
```

Listing 11.2 is Section 11.8 in code, with a second attempt for Windows. Python's `os.replace` renames onto an existing file on every system. Book: Section 11.9.2.

One Program, Several Files

`shop.h` / `shop.c` — orders, text helpers, the linked list and the commands: v0.8 without `main`.

`storage.h` / `storage.c` — loading the CSV, checking every field of every line, and saving safely.

`cashier_v9.c` — `main` and the command loop, calling `save_shop` after every change.

A **header** holds declarations only; `#include "shop.h"` with quotes for your own headers; `extern` declares data defined in another file.

An **include guard**, `#ifndef SHOP_H` / `#define SHOP_H` / `#endif`, makes a header safe to include twice.

Compile all sources together: `gcc -std=c17 -O2 -Wall -Wextra -o cashier_v9 cashier_v9.c shop.c storage.c`. *Book: Section 11.9.1.*

A Simulated Power Cut

```
$ step 1
Profitina Laundry cashier v0.9 - 0 order(s) loaded from shop.csv
> 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
```

Selected lines. The first session is killed with no goodbye and no final save, yet the second finds all three orders with their statuses. That is the whole chapter in one screen. Book: Section 11.10.

A Damaged `shop.csv`

```
  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
```

Line 4 repeats id 3; line 5 has the unknown service `dry-clean`; line 6 has no name.

The good orders, #1, #3 and #6, load; the cashier keeps working.

A loader that refused the whole file because of one bad line would leave Mbak Rina with nothing. Book: Section 11.10.

Making It Better, Using It for Real

Keep yesterday — append one line per pickup to `pickups.log` in mode `"a"`.

Keep a backup — a memory stick or cloud copy every evening. **Know when a file is not enough** — many users or many records call for a database.

For real — a tailor's open orders in CSV; a fruit and durian orchard's harvest log in append mode, one line per crate.

Files of names and phone numbers are personal data (UU PDP): a password on the computer, and delete what is no longer needed.

Check every `fopen`, read with the result of the read, treat files as input from a stranger, save through a temporary file, prefer text. Book: Sections 11.11–11.12.

Common Mistakes

`fopen` unchecked; `"w"` when `"r"` or `"a"` was meant; `while (!feof(fp))`.

`fgetc` into a `char` (`EOF` does not fit); **a name read with `%s`; the `'\r'` of Windows files.**

CSV split with `strtok`; `fclose` forgotten or its result ignored.

Binary files shared between machines; headers without include guards.

Each of these has a line in the book's programs that avoids it. Book: Section 11.13.

Summary: Five Things to Remember

1. RAM forgets; files do not. `fopen` can return `NULL`; `fclose` flushes the buffer. `"w"` empties at once (Table 11.1).

2. The reading loop tests the result of each read, never `feof`.

3. Real files are read line by line, split into fields that may be empty, checked field by field; damaged lines are reported with their number.

4. Logs grow in append mode; binary records are reached by `fseek` (Equation (11.1), Figure 11.1) but tied to one platform.

5. Save by writing a new file, checking `ferror` and `fclose`, and renaming (Figure 11.2).

Cashier v0.9, in modules with guarded headers, survives the power cut in C and Python with identical output. Book: Section 11.14.

Next Time

Chapter 12 — Evaluation and Discussion III: Lists, Trees and a Knight. Three complete problems solved step by step.

Sorting a linked list, growing a binary tree from a command file, and the knight's shortest path with breadth-first search.

Before then: Exercises 11.1–11.6, especially `find_customer`, which searches a CSV file line by line.

Next time we use everything from Chapters 9 to 11 on problems that are larger than one idea. Book: Sections 11.15–11.16.

Lecture 12 — Evaluation and Discussion III: Lists, Trees and a Knight

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 12

Lecturer: Irfan Subakti — yifana@gmail.com

Today we walk through three complete data-structure problems: how each is understood by hand, designed on paper, programmed, tested, and how a tempting wrong solution fails. Book: Chapter 12, Sections 12.1–12.9.

What This Meeting Covers

1. The method — eight steps for a data-structure problem, from restating it to checking yourself with a variant.

2. Problem 12.1 — sort a singly linked list with bubble sort, then insert at the end.

3. Problem 12.2 — grow a binary tree from a file of navigation commands.

4. Problem 12.3 — the knight's shortest path with breadth-first search and a linked-list queue.

5. Side by side — the extra idea each problem needed, and the tests that caught each wrong solution.

Each program exists in C and Python with identical output, and every C program ran under the sanitizers. Book: Chapter 12.

Where We Left Off

Chapter 9 taught text, Chapter 10 structures and linked lists, Chapter 11 files.

These three problems are a step up: the whole difficulty lies in keeping a set of pointers right while the structure changes.

They are independent and suit a small team: each person leads one, but everyone draws all three and reviews the others' programs.

A program you have only read and never questioned is not one you understand. Book: Section 12.1.

The Eight Steps

1. Restate in your own words: input, output, every decision left open. **2. Work a small example by hand**, one drawing per step.

3. Design on paper — pseudocode and cost. **4. Program** one function at a time.

5. Test with prepared cases — empty, one element, sorted, reversed, duplicates, largest input, wrong input; know the answer *before* running.

6. Run under the sanitizer. 7. Take apart a tempting wrong solution. 8. Check yourself with a variant.

For pointer problems step 2 is not a luxury: if you cannot draw what happens, you cannot program it. `python3 tests/run_tests.py` *compiles, runs both languages and compares with the expected output. Book: Section 12.2.*

Problem 12.1: Sort, Then Insert at the End

> **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.

Our decisions: numbers from the first input line, the new number from the second; the program must work for any list, empty and one-element included.

"Insert at the end" means at the end, not in its sorted place. Book: Section 12.3.1.

Worked Example 1: Bubble Sort by Hand

```
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
```

Bubble sort compares only **neighbors**, and a node's neighbor is one `->next` away. We swap the `data`, not the nodes: three assignments, no pointer changes.

The bar marks the sorted tail: after pass k the k largest are in place. A pass with no swap ends the sort.

Four passes, $5+3+2=10$ *swaps. Then 23 is attached after 42. Book: Section 12.3.2.*

Ten Is Not an Accident

$$\text{swaps of bubble sort} = \#\left\{(i, j) : i < j \text{ and } a_i > a_j\right\} \tag{12.1}$$

An **inversion** is a pair of positions $i < j$ with $a_i > a_j$. Each neighbor swap removes exactly one and creates none.

For 29, 4, 17, 4, 42, 8, 15: 29 gives 5, 17 gives 3, 42 gives 2, the rest none. Total 10.

Equation (12.1) gives the expected swap count of every test *in advance*, which is exactly what step 5 asks for.

The program prints its passes and swaps, so the prediction can be checked. Book: Section 12.3.2.

Bubble Sort on a List, on Paper

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
```

Pseudocode 12.1, line 6 is the subtle one: in pass 1, `last` is `NIL`, so the loop stops on the last node and never reads `p.next.data` through `NIL`. Cost: at most $\frac{n(n-1)}{2}$ *comparisons,* $O(n^2)$*; one pass if already sorted; insertion at the end* $O(n)$*. Book: Section 12.3.3.*

Running Problem 12.1

```
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.

Edge tests: an already sorted list (one pass, zero swaps), a reversed list, a single number, an empty list.

`list_sort.c` *and* `list_sort.py` *print the same lines on all five tests. Book: Section 12.3.6.*

A Tempting Wrong Solution: One Pass

```
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 a nearly sorted list one pass happens to be enough, and the programmer believes the program is correct.

One pass guarantees only that the *largest* number reaches the end. **Test with a reversed list, always.**

Second classic: `p = head; while (p->next != NULL)` crashes on an empty list; the test `list_sort_empty` and the sanitizer catch it.

From `wrong_list_sort.c`. Check yourself: for descending order only the `>` of line 7 changes; inserting at the front uses `push_front` of Chapter 10. Book: Sections 12.3.7–12.3.8.

Problem 12.2: A Tree Grown from a File

> **PROBLEM 12.2.** A text file describes a binary tree in pairs of lines: a command, then a whole number. `0` creates the root. `l` inserts a new node as the *left* child of the current node. A command `u…ur` first climbs one level toward the root for every `u`, then inserts the new node as the *right* child. The new node becomes the current node.

A **binary tree**: every node has at most two children, and every node except the root has exactly one parent. A list node has one `next`; a tree node has two.

In our words: a recipe for walking through a building and placing boxes. Our decisions: the file name on the command line, empty lines ignored, a plain `r` means the right child of the current node, and errors named with their line. Book: Section 12.4.1.

Worked Example 2: Following `tree5.txt`

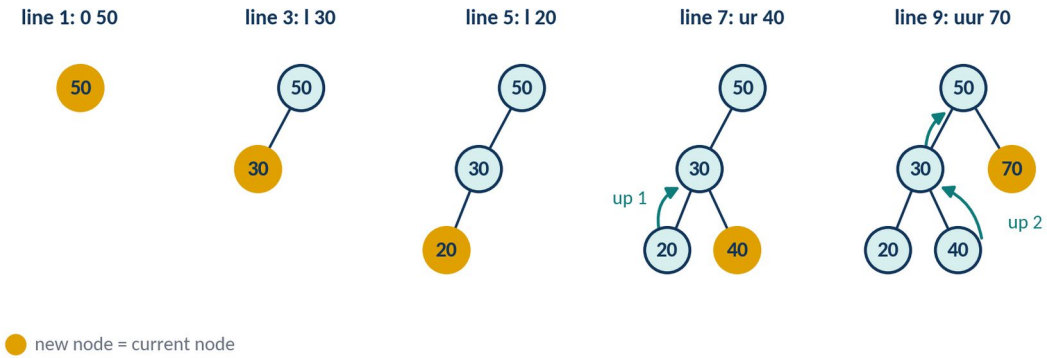


Figure 12.1 Growing the tree of `tree5.txt`: after each command the new node (amber) becomes the current node, and a command `u…ur` first climbs one level per `u` from it.

Figure 12.1: `0 50` *makes the root.* `l 30`, `l 20` *go down-left.* `ur 40`*: from 20 climb one level to 30; 40 is 30's right child.* `uur 70`*: from 40 climb to 30, then to 50; 70 is 50's right child. Book: Section 12.4.2.*

The Key Observation: a Way Up

The climb starts from the **current** node, the one just inserted.

A tree node normally knows only its children, so each node also stores a `parent` pointer, set at creation. "Climb one level" is `at = at->parent`.

The root's parent is `NULL`: that is how a climb above the root is detected.

To show the shape: **preorder** 50 30 20 40 70, **inorder** 20 30 40 50 70, **postorder** 20 40 30 70 50, and a sideways print.

The three orders are recursive (Chapter 5). Two different trees can share one order, which is why the program prints all three and the picture. Book: Section 12.4.2.

Building the Tree, on Paper

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
```

Pseudocode 12.2: each node created once; a climb of k costs k steps; the traversals are $O(n)$. Freeing is a postorder traversal: children first, then the node. Book: Section 12.4.3.

Running Problem 12.2

```
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
```

One trace line per command: the first line where program and paper disagree points at the mistake. Further tests cover a climb above the root, a second root, an occupied child and an unknown command. Book: Section 12.4.6.

A Tempting Wrong Solution: Climbing from the Parent

The off-by-one: start the climb from the *parent* of the current node, “because we are already one level below”.

Replay: line 7 `ur` goes from 20 to 30, then to 50; 40 becomes the right child of **50**.

Line 9 `uur` goes from 40 to 50 and tries to climb two more, above the root: an error on a perfectly correct file.

With a slightly different file there is no error at all, only a tree of the wrong shape, which is worse.

The defense: trace the statement’s own example by hand before coding. Check yourself: write the file for the balanced tree with preorder 8 4 2 6 12 10 14 and compare its inorder with 2 4 6 8 10 12 14. Book: Sections 12.4.7–12.4.8.

Problem 12.3: The Knight's Shortest Path

> **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.

A knight moves in an "L": two squares one way, one at a right angle; at most eight moves from a square, fewer near the edges.

Our decisions: queries `r1 c1 r2 c2` *from standard input, one per line; print the number of moves and one shortest path; reject squares off the board. Book: Section 12.5.1.*

Breadth-First Search: Rings on a Pond

The start has distance 0; every square one move away gets 1; every new square one move from those gets 2; ring by ring until the target.

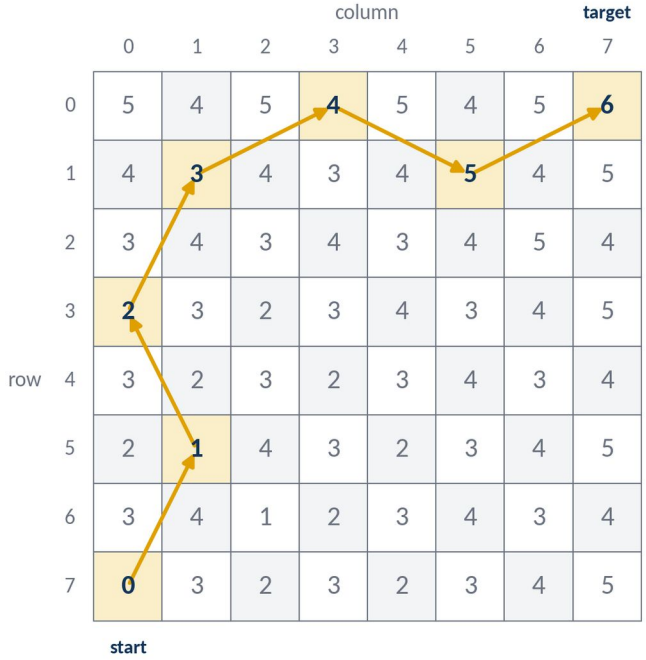
A **queue** keeps the rings in order: all squares of distance d leave before any of distance $d+1$.

$$\text{dist}(v) = \text{dist}(u) + 1 \tag{12.2}$$

Equation (12.2): when u reaches a new square v; and u is remembered as where v came *from*, to rebuild the path backwards.

Because the rings grow one move at a time, the first arrival at the target is the shortest. Book: Section 12.5.2.

Distances from (7,0)



row \ column	0	1	2	3	4	5	6	7
0	5	4	5	**4**	5	4	5	**6**
1	4	**3**	4	3	4	**5**	4	5
2	3	4	3	4	3	4	5	4
3	**2**	3	2	3	4	3	4	5
4	3	2	3	2	3	4	3	4
5	2	**1**	4	3	2	3	4	5
6	3	4	1	2	3	4	3	4
7	**0**	3	2	3	2	3	4	5

moves needed from (7,0); one shortest path in amber

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.

Figure 12.2: (6,1), diagonally next to the start, needs 4 moves. Knight distance has little to do with ruler distance, which is why this needs an algorithm, not a formula. Book: Section 12.5.2.

The Search, on Paper

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]
```

Pseudocode 12.3: the queue is Chapter 10's `List`; enqueue is `append`, dequeue removes the first node. Line 8 marks squares when they enter the queue, so at most 64 enter and at most $64 \cdot 8 = 512$ *moves are tried. Book: Section 12.5.3.*

Running Problem 12.3

```
(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
```

The first lines of `knight` on `data/knight_queries.txt`: the path has six steps, each one knight move and one ring further. Other queries: start equal to target, one move, a three-move hop, corner to corner, and a square off the board. Book: Section 12.5.6.

A Tempting Wrong Solution: Depth-First Search

```
(0,0) -> (7,7): DFS found a path of 52 move(s)
(3,3) -> (4,3): DFS found a path of 25 move(s)
```

Recursion suggests diving along the first move that works, stepping back only when stuck: **depth-first search**.

On the statement's example DFS answers 6, the right number, and a programmer who tests only the example is satisfied.

Corner to corner it answers 52; the three-move hop, 25. DFS finds *a* path, rarely the shortest.

From `knight_dfs.c`. Check yourself: for a king only the tables `DR` and `DC` change, and (7,0) to (0,7) needs $\max(7,7)=7$ *moves. Book: Sections 12.5.7–12.5.8.*

The Problems Side by Side

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

Table 12.1: in each, the structure makes the one operation that matters cheap, and each needed one extra pointer or mark the statement did not mention. Book: Section 12.6.

Using It for Real

A sorted list with insertion — a clinic's waiting list by appointment time, walk-in patients added at the end.

A tree — a cooperative's organization chart, the folders on a laptop, a chart of accounts where "Expenses" has "Rent" and "Wages".

BFS — the fewest transfers between two bus stops, the fewest turns in a warehouse between two shelves, the fewest stops to reach every delivery address by motorbike.

When steps are not equal, streets of different lengths for example, plain BFS is not enough; later courses generalize it, but the queue, the visited mark and the "from" pointer stay. Book: Section 12.7.

Common Mistakes

Coding before drawing; `p->next->data` **when** `p->next` **may be** `NULL`; **the empty structure forgotten**.

One end of a queue updated but not the other: enqueue on empty sets `head`; dequeue of the last sets `tail` to `NULL`.

Off-by-one in navigation; **marking late in BFS**; **believing the example**.

Freeing in the wrong order (a tree in postorder); **not running the sanitizer**.

Every pointer error here is visible on paper in a minute and invisible on screen for an hour. Book: Section 12.8.

Summary: Five Things to Remember

1. Restate, draw, design on paper, code one function at a time, test prepared cases under the sanitizer.

2. Bubble sort on a list swaps neighbors' data and stops after a pass without swaps (Pseudocode 12.1); swaps equal inversions (Equation (12.1)).

3. A `parent` pointer turns "climb k levels" into k steps (Pseudocode 12.2); free a tree in postorder.

4. BFS with a queue explores ring by ring, so the first arrival is the shortest (Pseudocode 12.3); mark when enqueuing.

5. All three wrong solutions pass the statement's example: test beyond it (Table 12.1).

Book: Section 12.9.

Next Time

Chapter 13 — C++ as a Better C. Streams, `std::string`, `std::vector`, references, overloading, default arguments, `new` and `delete`.

Cashier v1.0 is rewritten in C++, and many of the sharp edges of Chapters 9–11 disappear.

Before then: Exercises 12.1–12.6, especially insertion sort by relinking nodes (`list_insertion_sort`).

We leave C with every tool of the first twelve chapters in hand. Book: Sections 12.10–12.11.

Lecture 13 — C++ as a Better C

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 13

Lecturer: Irfan Subakti — yifana@gmail.com

Today we move from C to C++ and keep everything that makes C fast, while handing the dangerous bookkeeping to the language: streams, strings, vectors, references, overloading, default arguments, `new` and `delete`, and cashier v1.0. Book: Chapter 13, Sections 13.1–13.15.

What This Meeting Covers

1. **One family, two dialects** — where C++ came from, C++17, compiling with `g++`.
2. **Streams, `std::string`, `std::vector`** — input and output without format strings; text and lists that grow by themselves.
3. **References** — changing the caller's variable without pointers; `const` references for large inputs.
4. **Overloading and default arguments** — one name, several functions; arguments that may be left out.
5. **`new` and `delete`, and cashier v1.0** — typed allocation, and why a `vector` usually makes it unnecessary.

From this chapter on the programs are in C++17 and Python, still with identical output. Book: Chapter 13.

Where We Left Off: Bu Rina's Receipts

Through Chapters 1–12 the cashier grew in C: loops, arrays, functions, strings, structures, a linked list, files.

It works, but it has a history of accidents. A customer named *Raden Ajeng Kusumawardhani Sulistyaningrum* overflowed a 30-character array; the extra letters overwrote the weight, and the receipt showed millions of rupiah.

Every accident was a piece of bookkeeping the programmer had to do by hand: sizes, terminators, `free`, `&` and `*`.

C++ keeps what makes C fast and close to the machine, and adds tools that do that bookkeeping for you. Book: Section 13.1.

One Family, Two Dialects

1979: Bjarne Stroustrup at Bell Labs adds *classes* to C; “C with Classes” becomes **C++** in 1983, “C, one step further”.

Standardized by ISO; C++26 was finished in March 2026. This book uses **C++17**: complete in every compiler, enough for Chapters 13–16.

C++ is *almost* a superset of C; the `gcc` installed in Chapter 1 includes `g++`.

`g++ -std=c++17 -O2 -Wall -Wextra hello_laundry.cpp -o hello_laundry`. Always write `-std=c++17`: GCC 16.1 defaults to C++20.

On macOS, `g++` is a second name for `clang++` with the same flags; on Windows the MSYS2 commands are the same. Book: Section 13.2.

The Whole Chapter in One Table

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`

Read Table 13.1 once now and again at the end; by then every row will be familiar. Book: Section 13.2.2.

Streams: `cout` and `cin`

Listing 13.1 From `hello_laundry.cpp`: reading a name and a weight with `cin` and printing with `cout`.

```
    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";
```

`<<` (insertion) sends values to the stream; `>>` (extraction) takes them out. The type decides the format: no `%d`, no `&`.

`std::getline` reads a whole line; `>>` would stop at the space and read only “Bu”.

Listing 13.1: everything in the standard library lives in the namespace `std`. `hello_laundry.py` uses `input` and `int`. Book: Section 13.3.

Worked Example 1: The Price in Whole Rupiah

$$P = \left\lfloor \frac{g \cdot r + 500}{1000} \right\rfloor \tag{13.1}$$

Equation (13.1): weight g in grams, rate r in rupiah per kilogram; adding half the divisor rounds to the nearest rupiah.

$g = 3450$, $r = 7000$: $3450 \times 7000 = 24{,}150{,}000$.

Add 500 and divide by 1000: $24{,}150{,}500 / 1000 = 24{,}150$; the fraction .5 is dropped.

Integer division is exactly the floor for positive numbers, in C++ and with Python's `//`. No `double`: money must not lose a rupiah.

The same formula prices every row of the cashier. Book: Section 13.3.

A Trap Between `>>` and `getline`

> **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. Skip it first with `std::cin.ignore(1000, '\n')`, or read every line with `getline` and convert.

This is why `hello_laundry.cpp` reads the name before the number. Book: Section 13.3.

`std::string`: Text That Fits

A class that holds text of **any** length and manages its own memory; no terminator to forget, no size to choose.

Join with `+`; **compare** with `==` and `<`; **copy** with `=`.

`size()`, **`find(text)`** (returns `std::string::npos` when absent), **`substr(pos, len)`**, and `s[i]` for one character.

The long name of Section 13.1 now simply fits.

`strings_demo.cpp` builds "Rina Wulandari" with `+`, searches, cuts and converts case; `strings_demo.py` does the same with Python's `str`. Book: Section 13.4.

`std::vector`: an Array That Grows

`std::vector<int> grams = {3450, 1200, 5000, 2750};` then `grams.push_back(4100);`: the vector grows by itself.

It knows its `size()`, frees its memory by itself, and works with range-based loops: `for (int g : grams)`.

`v[i]` does not check; `v.at(i)` does, and throws `std::out_of_range`.

`std::sort(grams.begin(), grams.end())` sorts it; the element type goes in the angle brackets: `std::vector<std::string>`, later `std::vector<Order>`.

`vector_demo.cpp` and `vector_demo.py`: Python's `list` does what `std::vector` does. Book: Section 13.5.

Three Ways to Pass a Variable

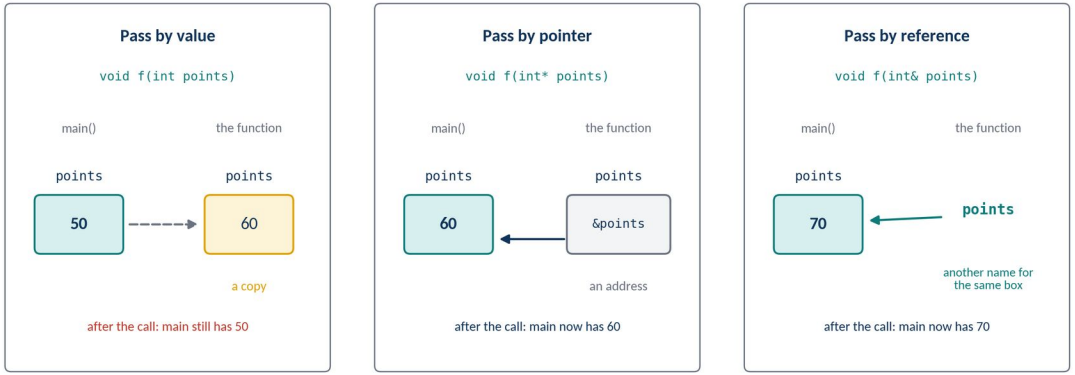


Figure 13.1 Three ways to pass `points`: a value parameter gets a copy, a pointer parameter gets the address, and a reference parameter is another name for the caller's own variable.

Figure 13.1: by value (Chapter 5), by pointer (Chapter 7), and new in C++, by reference. Book: Section 13.6.

References

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, and it can never be made to refer to a different variable.

A reference parameter `int& points` writes `points += 10;` with no `*`; the caller writes `f(points)` with no `&`.

Under the hood the compiler usually passes an address; the difference is that a reference cannot be null and cannot be re-pointed. Book: Section 13.6.

Worked Example 2: Value, Pointer, Reference

Listing 13.2 From `references.cpp`: value, pointer and reference parameters.

```
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
```

Start with `points = 50`. **By value**: the copy becomes 60; `points` stays **50**.

By pointer, `addBonusByPointer(&points)`: `*points` is the caller's box; **60**.

By reference, `addBonusByReference(points)`: `points` inside *is* the caller's box; **70**.

`int& alias = points; alias = 99;` puts **99** in `points`.

```
by value     : 50
by pointer   : 60
by reference : 70
alias = 99   : points is now 99
summary      : 12400 g in total, heaviest 5000 g
```

Listing 13.2: work the four lines on the board before showing the output. Book: Section 13.6.

Two Common Uses of References

Listing 13.3 From `references.cpp`: a function with two results and a `const` reference input.

```
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;
    }
}
```

Output parameters — `total` and `heaviest` let one function return two results (Chapter 7 did it with pointers).

`const` references for input — the vector is passed without copying, and `grams.push_back(1)` inside would not compile.

Listing 13.3. Python has no references: a function returns a tuple, `return total, heaviest`, and the caller unpacks it. Book: Section 13.6.

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.

`printLine(text)`, `printLine(label, rupiah)`, `printLine(ch, count)`: three functions, one name.

`printLine('=', 30)` fits only the third; `printLine("PROFITINA LAUNDRY")` only the first.

The linker still needs unique names, so the compiler decorates each with its parameter types (name mangling): `printLine(char, int)` *becomes something like* `_Z9printLineci`*. Book: Section 13.7.*

Default Arguments and the Express Price

`long long charge(int grams, long long ratePerKg, bool express = false)`: `charge(3450, 7000)` means `charge(3450, 7000, false)`.

Defaults must be the rightmost parameters, because arguments are matched from the left.

$$P_{\text{express}} = P + \lfloor \frac{P}{2} \rfloor \tag{13.2}$$

Equation (13.2): for 3450 g of washing, $P = 24{,}150$ by Equation (13.1), so $P_{\text{express}} = 24{,}150 + 12{,}075 = 36{,}225$.

Python has default arguments too, but no overloading: one name, one function; `overload_default.py` *uses three names instead. Book: Section 13.7.*

`new` and `delete`

Listing 13.4 From `new_delete.cpp`: `new[]` with `delete[]`.

```
    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
```

`new` knows the type: no `sizeof`, no cast. `new` pairs with `delete`, `new[]` with `delete[]`.

In practice a `std::vector` does the same job and needs neither.

Listing 13.4: `nullptr` is C++'s typed null pointer. Python frees memory by itself. Book: Section 13.8.

The Same Tools in Python

`str` has always behaved like `std::string`, but it is *immutable*: `full.upper()` returns a new string; slicing `full[space + 1:]` plays the role of `substr`, and `find` returns -1 instead of `npos`.

`list` is Python's `std::vector`: `append`, `len`, `sorted`, and an `IndexError` for a bad index.

No references — names are bound to objects; a function that must give back two results returns a tuple.

No overloading — one name, one function; default arguments work as in C++.

The conveniences are the same; only the spelling differs. That is why the Python twins stay as short as the C++ programs. Book: Sections 13.4–13.8.

Cashier v1.0

Reads a day's orders from `input/orders.txt`: `customer;service;grams;express`, `#` for comments.

Three services: `wash` Rp 7,000/kg, `washiron` Rp 10,000/kg, `iron` Rp 5,000/kg.

Checks every line, prints one row per accepted order, explains every refused line, finishes with totals and a summary per service.

Uses everything: `std::string`, `std::vector`, references, overloads, a default argument, `std::ifstream` and `<iomanip>` for columns.

`cashier.cpp` and `cashier.py` print the same text; `tests/run_tests.py` compiles, runs and compares every program of the chapter with the expected output. Book: Sections 13.9 and 13.11.

Testing It: Two Languages Check Each Other

Every program has the output it must print in `expected/`; the list of programs, inputs and arguments is a small text file in `tests/`.

`tests/run_tests.py` compiles each C++ program with `-Wall -Wextra -Werror`, so a warning counts as a failure.

It runs the C++ program and its Python twin on the same input and compares both with the expected text, character by character.

With `--sanitize`, the address and undefined-behavior sanitizers stop a program at the first bad memory access or overflow.

```
references                C++ ok   Python ok
cashier                   C++ ok   Python ok
cashier_bad               C++ ok   Python ok
all programs agree
```

A rounding difference or a different order of rows in either language would fail the comparison. After any change, run the tests again. Book: Section 13.11.

Running Cashier v1.0

```
 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
Revenue         : Rp 188950
```

Check one row by hand: Pak Budi's 5,200 g at Rp 10,000 gives $52,000$ *by Equation (13.1), and express adds half:* $78,000$*. Book: Section 13.10.*

Refusing Bad Lines Politely

```
 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
```

From `input/orders_bad.txt`: every bad line refused with its number and a reason in plain words, the good lines around it still accepted. A trustworthy program never guesses, never crashes, and says what it did. Book: Section 13.12.

Using It for Real

The price list — a tailor per piece, a photocopy shop per page with a lower price above 100, a car wash per vehicle class.

The checks — ask the person at the counter which mistakes really happen; one check, one clear message each.

The summary — ask the owner which numbers she looks at in the evening.

Money in whole rupiah, rounded once at a documented place; Chapter 15 makes that a `Rupiah` class.

A check that is never needed costs nothing; a missing check can cost a day's accounts. Book: Section 13.13.

Common Mistakes

Forgetting `-std=c++17`; mixing `>>` and `getline`; comparing C strings with `==`.

Indexing a vector past its end (`v[i]` does not check); **returning a reference to a local variable** (a dangling reference).

`new[]` freed with `delete`; money in `double`; ambiguous overloads.

Return by value instead of by reference to a local; C++ moves the value out cheaply. Book: Section 13.14.

Summary: Five Things to Remember

1. C++ compiles almost every C program; `g++ -std=c++17 -Wall -Wextra` on every system (Table 13.1).

2. `cout <<` and `cin >>` need no format strings; `getline` reads a line; `std::string` and `std::vector` grow by themselves.

3. A reference is another name for a variable (Figure 13.1); `const&` passes large objects without copying.

4. Overloads share a name and differ in parameters; default arguments fill in trailing parameters; both at compile time.

5. `new`/`delete` must be paired; a container usually makes them unnecessary; money stays in integers (Equation (13.1), Equation (13.2)).

Python gives the same conveniences its own way, and cashier v1.0 prints identically in both. Book: Section 13.15.

Next Time

Chapter 14 — Classes and Data Abstraction. An `Order` that cannot be wrong: `private` data, constructors, invariants, destructors, and a first look at inheritance.

The cashier becomes a `Shop` class that the rest of the program can only use through its public functions.

Before then: Exercises 13.1–13.6, especially accepting a weight in kilograms with a decimal comma.

Next time begins with free laundry given away by accident, and the class that prevents it. Book: Sections 13.16–13.17.

Lecture 14 — Classes and Data Abstraction

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 14

Lecturer: Irfan Subakti — yifana@gmail.com

Today an order learns to guard its own rules: private data, a constructor that refuses bad values, a life cycle that cannot be broken, destructors that clean up by themselves, a first look at inheritance, and a `Shop` class for the whole day. Book: Chapter 14, Sections 14.1–14.13.

What This Meeting Covers

1. **Data abstraction and the invariant** — interface versus implementation; `public`, `protected`, `private`.
2. **The `Order` class** — header and implementation, getters, `const`, static members, a constructor that throws.
3. **Constructors and destructors** — born valid, cleaned up automatically: RAII.
4. **A first look at inheritance** — a base class `Service`, virtual functions, polymorphism.
5. **The `Shop`** — a class that has many orders, and one day at the counter.

The classes exist in C++ and Python and behave identically. Book: Chapter 14.

Where We Left Off: Free Laundry, by Accident

One Saturday the drawer was Rp 24,150 short. A helper had “fixed” an order with a menu option that set the weight directly, typed 0, and the price became zero.

In the C cashier an order is a `struct`: any function can set any field to any value, −500 grams, a rate of 700, an empty name.

The rules of a sensible order live in the programmer’s head and in a few `if` statements in the input function.

The idea of today: put the rules inside the order. It checks its values when created, refuses to exist if they are wrong, and hides its data behind functions that keep the rules. Book: Section 14.1.

Data Abstraction: the 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 adjust the heating element; it is sealed inside. You get a simple **interface** and a guaranteed result.

A class does the same for data: its **interface** is the functions others may call; its **implementation**, the data and code inside, is hidden.

A type defined by what you can do with it, not by how it is stored, is an *abstract data type*.

The class is the building block of C++, Java, C#, Python and almost every language you will use after this book. Book: Section 14.2.1.

The 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.

$$1 \le g \le 50\,000 \tag{14.1}$$

For an order: a non-empty name, a known service, a weight in grams as in Equation (14.1), and a status that moves only Received → Washing → Ready → Picked up.

If the data is private and every member function keeps the invariant, it holds forever: checked in one place, true in every place, including code written years from now. Book: Section 14.2.2.

From `struct` to `class`

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

Table 14.1: in C++, `struct` and `class` differ only in the default, public for `struct`, private for `class`.

Convention: `struct` for plain bundles of data with no rules; `class` for types with an invariant. Book: Section 14.2.3.

The Problem, Measured

```
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
```

`struct_problem.cpp` makes one correct order, then three assignments: free laundry, a price ten times too low, an order for nobody.

The compiler cannot help, because nothing in the program says these values are wrong. A Python dataclass is just as open.

Before fixing a problem, see it. Book: Section 14.3.

The Plan of the `Order` Class

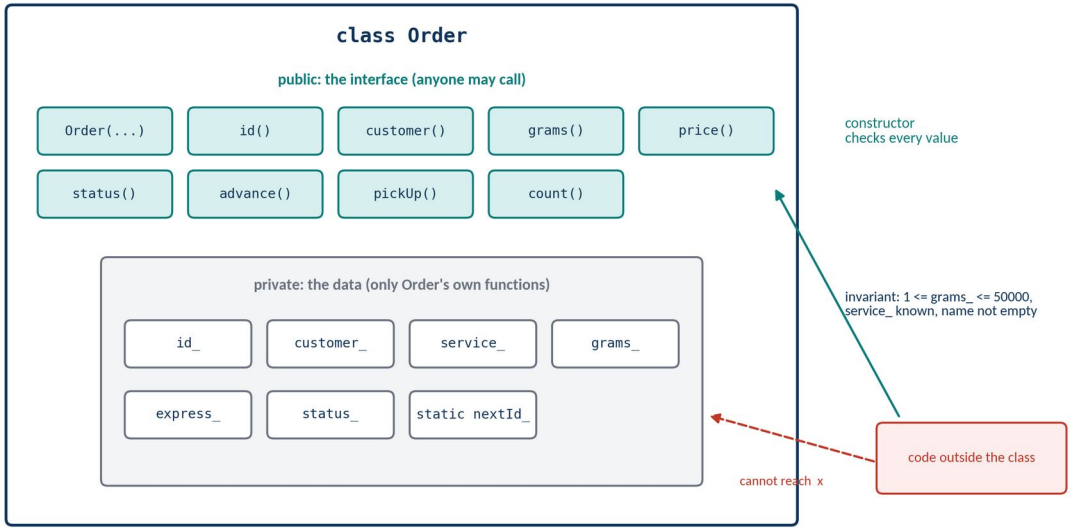


Figure 14.1 The `Order` class: a public interface around private data; outside code can only call the public functions, and the constructor checks every value.

Figure 14.1: a constructor, functions that read, two functions that change in allowed ways (`advance`, `pickUp`), and a counter; the data is private. Book: Section 14.4.

The Interface: `order.h`

Listing 14.1 From `order.h`: the interface of the `Order` class (some getters omitted).

```
enum class Status { Received, Washing, Ready, PickedUp };

class Order {
public:
    Order(const std::string& customer, const std::string& service, int grams,
          bool express = false);

    int id() const { return id_; }
    int grams() const { return grams_; }
    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);
};
```

Listing 14.1 sits inside an include guard, as in Chapter 11. `enum class` *names must be written* `Status::Ready` *and do not silently turn into integers. Book: Section 14.4.1.*

Reading the Interface

The constructor has the class's name and no return type; it runs whenever an `Order` is created and establishes the invariant.

Getters such as `id()` give read access; written in the class they are *inline* and cost nothing at run time.

`const` after the parameters promises *this function does not change the object*; only `const` functions may be called on a `const Order&`.

`price()` is computed, never stored, so it can never disagree with the weight. `static` members belong to the class: one `nextId_` for all orders.

The trailing underscore (`grams_`) marks data members, so inside a member function it is clear which names are members. Book: Section 14.4.1.

The Constructor: `order.cpp`

Listing 14.2 From `order.cpp`: the constructor, with its member initializer list and its three checks.

```
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
}
```

Listing 14.2: the member initializer list after the colon sets each member directly; the body checks the invariant and throws when a value is wrong. `int Order::nextId_ = 1;` *defines the static member in exactly one file. Book: Section 14.4.2.*

Numbers Without Gaps

`id_ = nextId_++;` comes *after* the checks, so refused orders use no number; valid orders are 1, 2, 3, …

$$\text{count} = \text{nextId} - 1 \tag{14.2}$$

Equation (14.2) is exactly what `count()` returns.

`price()` applies Equation (14.3) and adds half for express, as in Chapter 13.

$$P = \left\lfloor \frac{g \cdot r + 500}{1000} \right\rfloor \tag{14.3}$$

The pricing rule of Chapter 13 has moved inside the class, where no caller can compute it differently. Book: Section 14.4.2.

The Life Cycle of an Order

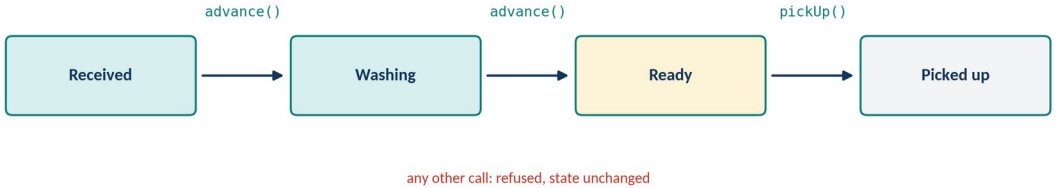


Figure 14.2 The life cycle of an order: `advance()` moves it forward twice, `pickUp()` hands it over, and every other call is refused without changing the state.

Figure 14.2: `advance` and `pickUp` are the only functions that change an order after creation. They return `false` for anything the diagram does not allow, leaving the state unchanged. Book: Section 14.4.2.

Using the Class

```
#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. `a.grams_ = 0;` does not compile: `grams_` is private.

`order_main.cpp`, compiled with `g++ -std=c++17 -O2 -Wall -Wextra cpp/order.cpp cpp/order_main.cpp -o order_main`: every `.cpp` of the program is named. The accident of Section 14.1 is now not unlikely but impossible. Book: Section 14.4.3.

Worked Example 1: A Refused Order, Step by Step

EXAMPLE. `main` executes `Order bad("Mbak Sari", "wash", -500);` inside a `try` block.

1. Memory for `bad` is reserved; the constructor is called, `express = false` from the default.

2. The initializer list runs: `grams_` becomes −500. The object breaks its invariant, but nobody outside can see it yet.

3. Name not empty; `wash` costs 7,000; `grams_ < 1` is **true**.

4. `throw` creates `std::invalid_argument` and leaves at once; `id_ = nextId_++;` never runs, so `nextId_` stays 3.

5. The object never existed; its two strings are destroyed properly. **6.** The nearest `catch` prints `refused: weight must be 1..50000 g, not -500`.

In C++ there is no half-constructed object the program can touch: either the constructor finishes and the invariant holds, or the object does not exist. Python gives the same guarantee when `__init__` raises. Book: Section 14.4.6.

The Same Class in Python

The constructor is `__init__`; `self` is written explicitly where C++ has the hidden pointer `this`.

No `private` keyword: a leading underscore (`_grams`) marks a name as private *by agreement*.

Read-only **properties** give the same public interface as `order.h`; `raise ValueError(...)` refuses a bad value.

`order_main.py` prints exactly what `order_main.cpp` prints.

Why getters at all? The interface should promise behavior, not storage: tomorrow `price()` may read a price list, and callers will not notice. Book: Sections 14.4.4–14.4.5.

Destructors and RAII

A **destructor**, `~ShiftLog()`, runs when an object dies: for a local variable, at the closing brace of its block, whatever the way out.

RAII, "resource acquisition is initialization": take a resource in a constructor, give it back in the destructor, and it can never be forgotten.

Objects in one scope die in the *reverse* order of their creation, like plates taken off a stack.

```
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
```

`raii_demo.cpp`: Dewi's log closes at her block's brace; Eko's closes last, after "shop closes". Python does not promise when `__del__` runs; its tool for guaranteed clean-up is `with`, whose `__exit__` runs at the end of the block. Book: Section 14.5.

A First Look at Inheritance

> **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.

`Service` is the base: a `protected` name, and pure virtual `price` and `unit`, so it is *abstract*.

`PerKilo`, `PerPiece`, and `Express`, which *wraps* another service and adds half.

Constructors are not inherited: each derived constructor calls its base constructor, `: Service(name), rate_(ratePerKg)`. A base class gets `virtual ~Service() = default;`. Book: Section 14.6.

Worked Example 2: One Call, Several Behaviors

```
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
```

`menu` is a `std::vector<const Service*>`; the loop calls `menu[i]->price(amounts[i])` without knowing the class.

wash: $(3450 \cdot 7000 + 500)/1000 = 24,150$. **wash express**: $24,150 + 12,075 = 36,225$.

bed cover, 2 pieces: $2 \cdot 25,000 = 50,000$; express: $75,000$.

This is polymorphism. At run time each object carries a pointer to its class's virtual table, which holds the addresses of that class's versions of the virtual functions. In Python every method behaves like a virtual one. Book: Section 14.6.

The `Shop`: a Class That Has Orders

Inheritance says “is a”; `Shop` **has** a `std::vector<Order>`.

Operations: receive a new order, find by number, advance, pick up, list, report the money.

Every operation returns one line of text; `Shop` does not decide *where* text is printed.

`Shop` never checks a weight itself: it relies on the `Order` constructor.

`shop.h`, `shop.cpp` and `shop_main.cpp`, and `shop.py` with `shop_main.py`. Book: Section 14.7.

One Day at the Counter

```
> new Mbak Sari;washiron;-500;no
refused: weight must be 1..50000 g, not -500
> pickup 2
order #2 is not ready (received)
> advance 2
order #2 cannot advance (picked up)
> advance 9
order #9 does not exist
> pickup two
not a number in 'pickup two'
> report
picked up 2 of 3, paid Rp 39150, still in the shop Rp 52000
```

Selected lines of `./shop < input/day.txt`: every mistake is refused with a reason, and the report adds up. The test runner compares C++ and Python on this day and on the demos. Book: Sections 14.8–14.9.

Making It Better, Using It for Real

Money as a type and **printing with** `<<` — Chapter 15. **Keeping the day** — save in the destructor, load in the constructor. **Finding faster** — `std::map`.

Three questions: *What makes one item valid?* (constructor) *In which order may its state change?* (one function per arrow) *What does the owner want at the end of the day?* (`report`).

Mbak Wulan's catering warung: a customer, 1 to 200 portions, delivery between 06:00 and 20:00; Ordered, Cooking, On the way, Delivered.

For a clinic, a queue program only keeps the queue in order; deciding who is urgent is for the nurse and the doctor. Book: Sections 14.10–14.11.

Common Mistakes

Public data “just for now”; **forgetting `const` on reading functions**; **checking the invariant outside the class**.

Assigning in the constructor body instead of the initializer list; initialize in declaration order.

A static member defined twice or not at all; **compiling only one file** (“undefined reference”).

A base class without a virtual destructor; **relying on Python’s `__del__`**.

Book: Section 14.12.

Summary: Five Things to Remember

1. Data abstraction separates interface from implementation (Figure 14.1); access specifiers enforce it (Table 14.1).

2. Private data plus member functions that keep the rules make an invariant hold forever (Equation (14.1)).

3. Constructors establish the invariant and throw to refuse; no invalid object ever exists; destructors make RAII possible.

4. `const` members promise not to change; static members belong to the class (Equation (14.2)); a state diagram becomes one function per arrow (Figure 14.2).

5. Inheritance is “is a”, composition is “has a”; virtual functions give one call several behaviors.

In Python: `__init__`, underscore by agreement, properties, `with`, and every method virtual. Book: Section 14.13.

Next Time

Chapter 15 — Operator Overloading and a Complete Case Study. A `Rupiah` class with `+`, `*`, `==`, `<` and `<<`; fractions with Euclid's algorithm.

Then the complete Profitina Laundry program: customers, services, orders, discounts, a daily report.

Before then: Exercises 14.1–14.6, especially saving the shop's orders in its destructor.

Next time receipts will add up by construction. Book: Sections 14.14–14.15.

Lecture 15 — Operator Overloading and a Complete Case Study

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 15

Lecturer: Irfan Subakti — yifana@gmail.com

Today money becomes a class with its own operators, fractions meet Euclid's algorithm, and Chapters 13, 14 and 15 come together in the complete Profitina Laundry program. Book: Chapter 15, Sections 15.1–15.11.

What This Meeting Covers

1. **An operator is a function** — how the compiler reads `wash + iron`, and the rules of overloading.
2. **The `Rupiah` class** — compound assignment as members, arithmetic and comparison as non-members, one rounding rule, printing with `<<`.
3. **The `Fraction` class** — lowest terms with Euclid's gcd, addition with the lcm, a converting constructor.
4. **The complete case study** — three data files, receipts, a sorted daily report on screen and in a file.
5. **Checking it by hand** — the most complicated receipt, and the report totals.

Every program in C++ and Python, with identical output. Book: Chapter 15.

Where We Left Off: Receipts That Add Up

Every month Bu Rina's receipts are a few hundred rupiah away from the program's total.

Two helpers computed the same receipt by hand, and each rounded the 10% member discount in a different way.

The problem is not arithmetic. *Money* is a plain number, and every piece of code makes its own small decisions: how to round, how to print, whether a weight may be added to a price.

The cure is Chapter 14's idea applied to a number: make money a class that adds, takes percentages with one rule, compares and prints itself, and let operator overloading keep the code readable. Book: Section 15.1.

An Operator Is a Function

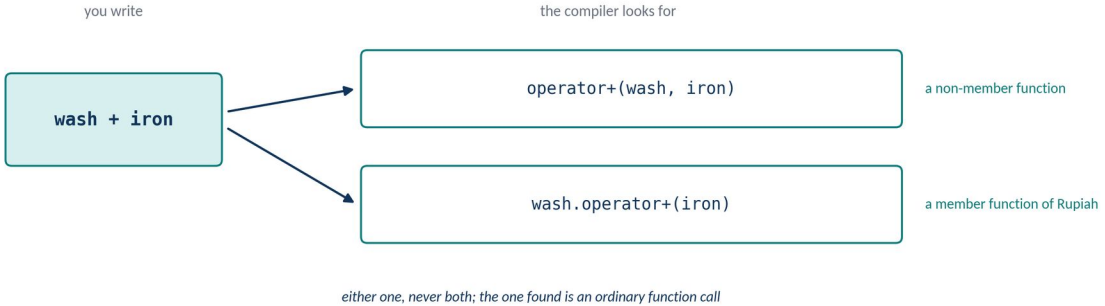


Figure 15.1 What the compiler does with `wash + iron` for class operands: it looks for a non-member `operator+(wash, iron)` or a member `wash.operator+(iron)` and calls the one it finds.

Figure 15.1: for two `int`s the compiler emits a machine addition; for class objects it looks for a function named `operator+`. `std::cout << x` is likewise a call of `operator<<(std::cout, x)`. Book: Section 15.2.1.

Operator Overloading

> **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).

The limits — no new operators (no `**`); precedence, arity and associativity never change; `1 + 1` is always 2; `.`, `::`, `?:` and `sizeof` cannot be overloaded.

The unwritten rule — an operator should do what a reader expects. `+` on money adds money; when in doubt, write a named function.

`operator[]` is also an overloaded operator: that is why `std::map` can create a missing entry while `std::vector` does not check its index. Book: Section 15.2.2.

Which Form for Which Operator

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__`

Table 15.1: operators that change their left operand are members; symmetric operators are non-members built on them; `<<` must be a non-member because its left operand is a stream. Book: Section 15.2.2.

`Rupiah`: the Members

Listing 15.1 From `rupiah.h`: the class `Rupiah` (`operator-=` omitted).

```
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 { 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;
};
```

Listing 15.1: one data member, whole rupiah in a `long long`. The constructor is `explicit`, so a bare number never silently becomes money. `+=` returns `*this` by reference, which makes `till += wash;` behave like the built-in `+=`. Book: Section 15.3.1.

One Rounding Rule for Everyone

$$\text{percent}(a, p) = \left\lfloor \frac{a \cdot p + 50}{100} \right\rfloor \tag{15.1}$$

Equation (15.1) rounds p % of a non-negative amount a *half away from zero*; a negative amount is rounded as its absolute value and the sign put back.

Adding 50 before dividing by 100 is Chapter 13's trick of adding 500 before dividing by 1000.

10% of Rp 24,150: $\lfloor (241{,}500 + 50)/100 \rfloor = 2{,}415$.

The two helpers of Section 15.1 can no longer disagree: there is only one place where a percentage is rounded. Book: Section 15.3.1.

`Rupiah`: the Non-Members

Listing 15.2 From `rupiah.h`: arithmetic, comparison and output as non-members.

```
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; }
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.value() < b.value();
}
inline std::ostream& operator<<(std::ostream& os, const Rupiah& r) {
    return os << toText(r);                    // honors setw, left and right
}
```

Listing 15.2: `+` copies its left operand and uses `+=`. Two versions of `*` make both `rate * amount` and `3 * iron` work; there is no `Rupiah * Rupiah`. The other four comparisons are written from `==` and `<`, so they can never disagree. `toText` prints `Rp 1.234.567.890`. Book: Section 15.3.1.

The Class at Work

```
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
[     Rp 24.150]
[Rp 24.150     ]
```

From `rupiah_demo.cpp`: each line is ordinary arithmetic to read and a function call underneath. `setw(14)` aligns an amount right or left like a number. `Rupiah x = 5000;` and `wash * wash` do not compile. Book: Section 15.3.2.

Worked Example 1: `base + express - discount`

`base` Rp 27,500, `express` Rp 13,750, `discount` Rp 4,125; `+` and `-` go left to right: `(base + express) - discount`.

1. Non-member `operator+`: `a` is a *copy* of `base`, `b` another name for `express`.

2. `a += b`: $27,500 + 13,750 = 41,250$; a temporary `Rupiah` is returned; `base` is untouched.

3. `operator-` copies the temporary and calls `-=`: $41,250 - 4,125 = 37,125$.

4. The temporaries die at the end of the statement; the optimizer makes it as cheap as plain integers.

Safety at no run-time cost: the expression cannot be written with a weight or a count by mistake. Book: Section 15.3.3.

The Same Class in Python

Special methods play the role of operators: `__add__`, `__sub__`, `__mul__`, and `__rmul__` for `3 * iron`.

`__eq__` and `__lt__`, with `functools.total_ordering` generating the other comparisons.

`__str__` gives the text; `__format__` lets an f-string align an amount like a number.

`rupiah_demo.py` prints exactly what `rupiah_demo.cpp` prints.

A careful Python class returns `NotImplemented` when it cannot handle the other operand, so Python can try the other operand's reflected method. Book: Section 15.3.4.

Fractions and Euclid's Algorithm

$$\gcd(a,b)=\gcd(b,a \bmod b),\gcd(a,0)=a \tag{15.2}$$

A fraction is kept in **lowest terms**: $\frac{6}{8}$ is stored as $\frac{3}{4}$, dividing by the greatest common divisor, found by Equation (15.2) (Euclid, around 300 BC).

Worked: $\gcd(270,192)=\gcd(192,78)=\gcd(78,36)=\gcd(36,6)=\gcd(6,0)=6$.

Why: any divisor of a and b divides $a \bmod b$, and back again; the steps grow only with the number of *digits*.

Detergent and softener measured in scoops are natural fractions: half a scoop plus a third. Book: Section 15.4.

The lcm and Adding Fractions

$$\operatorname{lcm}(a,b) = \frac{a}{\gcd(a,b)} \cdot b \tag{15.3}$$

Equation (15.3): divide *first*; $a \cdot b$ can overflow even when the lcm fits.

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot \frac{m}{b} + c \cdot \frac{m}{d}}{m}, m = \operatorname{lcm}(b,d) \tag{15.4}$$

Equation (15.4): for $\frac{1}{2} + \frac{1}{3}$, $m = 6$, and the sum is $\frac{1 \cdot 3 + 1 \cdot 2}{6} = \frac{5}{6}$.

`fraction.cpp` *implements* `+`, `-`, `*`, `/`, `==`, `<` *and* `<<` *on these formulas. Book: Section 15.4.*

`Fraction` at Work

```
gcd(270, 192) = 6, lcm = 8640
1/2 + 1/3   = 5/6
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
```

The invariant in the constructor — a zero denominator throws; a negative denominator moves its sign up; every fraction is reduced.

That makes `==` trivial — equal fractions in lowest terms have equal numerators and denominators.

Not `explicit`, on purpose — `Fraction(long long num = 0, long long den = 1)` converts `2` into $\frac{2}{1}$, so `2 + half` works.

Selected lines. `Rupiah` refuses conversions; `Fraction` welcomes them. Decide for each one-argument constructor: is every value of the parameter type really a value of your type? Book: Section 15.4.

The Complete Case Study

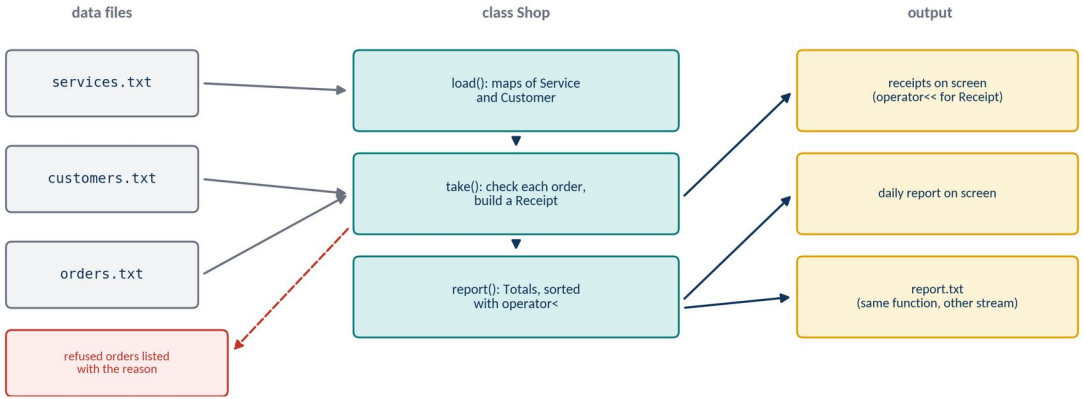


Figure 15.2 The case study: three data files are loaded into the `Shop`, each order is checked and turned into a `Receipt`, and the receipts are printed and summarized in a report written both to the screen and to `report.txt`.

Figure 15.2: the plan of the program Bu Rina could use at the end of every day. Book: Section 15.5.

The Data

`data/services.txt` — code, name, unit (`kg` or `pcs`), rate: `W` Wash & fold 7,000/kg, `WI` Wash & iron 10,000/kg, `I` Iron only 5,000/kg, `BC` Bed cover 25,000/pc, `SH` Shoes 35,000/pc.

`data/customers.txt` — id, name, member; members get 10% off.

`data/orders.txt` — number, customer, service, amount (grams or pieces), express.

Two lines are wrong on purpose: order 1006 names customer `C05`, who does not exist; order 1009 names service `DC`.

The price list is no longer inside the program: a tailor or a car wash changes a file, not the code. Book: Section 15.5.1.

The Rules

Base price — rate times amount, with Chapter 13's rounding for services per kilogram.

Express surcharge — 50% of the base by Equation (15.1), for express orders.

Member discount — 10% of base plus surcharge, for members.

$$\text{total} = \text{base} + \text{express} - \text{discount} \tag{15.5}$$

Because all four are `Rupiah` objects, Equation (15.5) appears in the program literally as `base + express - discount`. Book: Section 15.5.2.

The Program

`std::map` from code to `Service` and from id to `Customer`: `find` returns `end()` for an unknown key.

`take` checks each order line, builds a `Receipt` or records a refusal with its reason.

`operator<<` prints a receipt; `operator<` on a `Total` sorts the report, largest first.

One report function takes a `std::ostream&`, so the same code writes to the screen and to `report.txt`.

`laundry.cpp` includes `rupiah.h` unchanged. `laundry.py` uses `@dataclass`, `__str__`, `__lt__`, `dict` with `setdefault`, and a report function that writes to any object with a `write` method. Book: Section 15.5.3.

Worked Example 2: Two Orders Through `take`

Order 1006 — five fields, correct count. `customers_.find("C05")` passes C01 to C04, finds nothing, returns `end()`: refused, “unknown customer C05”.

Order 1002 — `1002;C02;WI;5200;yes`. Pak Budi, not a member; wash and iron at 10,000 per kg.

Base: $(5200 \times 10000 + 500)/1000 = 52,000$. Express: $(52,000 \times 50 + 50)/100 = 26,000$. No discount.

Total by Equation (15.5): $52,000 + 26,000 - 0 = 78,000$.

The refused order never becomes a receipt; the good one carries pointers to its customer and service. Book: Section 15.5.4.

The Receipt to Check by Hand

```
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
```

Base $(2750 \times 10000 + 500)/1000 = 27{,}500$; surcharge $(27{,}500 \times 50 + 50)/100 = 13{,}750$.

Discount: 10% of $41{,}250$ is $(412{,}500 + 50)/100 = 4{,}125$. Total $37{,}125$.

The most complicated receipt of the day, with all three rules. Book: Section 15.6.

The Daily Report

```
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
Member discounts given          Rp 12.540
GRAND TOTAL                    Rp 272.060
```

Check: wash and iron is $78{,}000+37{,}125=115{,}125$*, the largest, so first. The four member discounts are* $2{,}415+5{,}000+1{,}000+4{,}125=12{,}540$*. C++ and Python print the same, and both write* `report.txt`*. Book: Sections 15.6–15.7.*

Making It Better, Using It for Real

A second sort order as a comparison function for `std::sort`; **refused orders** to `refused.txt` through a second stream.

A `Grams` class so that `Rupiah + Grams` cannot compile; **several days** with a date column and a map from date to `Shop`.

For real — a tailor puts “hem trousers;pcs;15000” into `services.txt`; a car wash, vehicle classes; a caterer, portions.

Three habits: money in a type of its own; business rules in one visible place; every input line checked and refused with a reason. Book: Sections 15.8–15.9.

Common Mistakes

Returning a copy from `+=`; `operator+` as a member (then `3 * iron` fails).

`==` and `!=` defined independently; a `<` that is not strict (`<=` breaks `std::sort`).

`explicit` forgotten; overflow in `a * b / gcd`; operators with surprising meanings.

Python: return `NotImplemented` rather than guess when an operand is of an unknown type.

Book: Section 15.10.

Summary: Five Things to Remember

1. `a + b` on objects is a call of `operator+` (Figure 15.1); precedence and arity never change.

2. `+=` is a member returning `*this`; `+`, `==`, `<` are non-members built on it; `<<` returns the stream (Table 15.1).

3. `Rupiah` rounds percentages one way (Equation (15.1)) and refuses `Rupiah * Rupiah` and silent conversions.

4. `Fraction` stays in lowest terms with Euclid's gcd (Equation (15.2)) and adds with the lcm (Equation (15.3), Equation (15.4)).

5. The case study reads files, checks every order, and computes Equation (15.5) literally (Figure 15.2).

Python does the same with special methods and `functools.total_ordering`. Book: Section 15.11.

Next Time

Chapter 16 — Evaluation and Discussion IV. Five problems solved step by step: chess pieces, overloading `add`, anagrams, matrix multiplication, pick-up times.

Each with a hand example, a design, C++ and Python, and a tempting wrong solution.

Before then: Exercises 15.1–15.6.

The last meeting uses the whole book. Book: Sections 15.12–15.13.

Lecture 16 — Evaluation and Discussion IV: Classes, Overloading and the Whole Book

Course: Fundamental Programming (EF234101)

Book: Irfan Subakti, *Fundamental Programming with C, C++ and Python*, Chapter 16

Lecturer: Irfan Subakti — yifana@gmail.com

Today we walk through five complete problems that use the C++ of Chapters 13–15 and the foundations of the whole book, each with a hand example, a design, a program, and a tempting wrong solution taken apart. Book: Chapter 16, Sections 16.1–16.11.

What This Meeting Covers

The method — seven steps, from restating the problem to changing one thing and predicting the result.

16.1 Chess pieces — completing a class hierarchy by imitation; exclusive or for the rook.

16.2 Overloading `add` — two to five arguments, and why default arguments beside an overload fail.

16.3 Anagrams — every window of the text, and the loop bound that loses the last one.

16.4 Matrix multiplication and **16.5 Pick-up times** — the dimension check; a `Time` class with an invariant.

Every solution in C++ and Python, identical output. Book: Chapter 16.

Where We Left Off: the Toolkit

Chapters 1–12 — types, loops, functions, arrays, pointers, strings, structures, lists, files, and computing by hand first.

Chapters 13–15 — C++ strings, vectors, references and overloading; classes with invariants; operators.

16.1 needs inheritance, 16.2 overloading, 16.3 strings and exact loop bounds, 16.4 two-dimensional data, 16.5 a class with operators.

Book: Section 16.1.

The Seven Steps

1. Restate — input, output, every decision the statement leaves open. **2. Examples by hand** — the statement's, and your own edge cases.

3. Design on paper — data, pseudocode, cost. **4. Write the program** — complete, no warnings.

5. Run the hand examples — character for character. **6. Try to break it** — the tempting wrong solution.

7. Change one thing and predict — the "check yourself" box, answered *before* running.

Pólya's problem-solving method adapted to programs. Book: Section 16.2.

Problem 16.1: Chess Pieces

> **PROBLEM 16.1.** A base class `Piece` stores a piece's name, side and position and prints the position; a complete class `Pawn` derived from it is given. Add the classes `Bishop`, `Knight`, `Rook` and `Queen` in the same way, each starting at (0, 0), and test each with a legal move, an illegal move, and a legal move from the new position.

Our decisions: coordinates $0 \le x, y \le 7$; `faction_` is $+1$ for white, -1 for black; each piece is tested alone on the board; a refused move prints `Invalid movement!!` and leaves the piece where it was.

Creativity belongs in the conditions, not in the structure: the new classes imitate `Pawn` exactly. Book: Section 16.3.1.

Five Rules as Five Conditions

With $d_x = x' - x$ and $d_y = y' - y$: **Pawn** $d_x = 0$ and $d_y = \text{faction}$. **Bishop** $|d_x| = |d_y|$ and $d_x \neq 0$.

Knight $\{|d_x|, |d_y|\} = \{1, 2\}$. **Queen** like a rook or like a bishop.

Rook — *exactly one* of x and y changes: exclusive or.

$$\text{rook}(x, y \to x', y') = (x' \neq x) \oplus (y' \neq y) \tag{16.1}$$

Equation (16.1): in C++ and Python, `^` *applied to two* `bool` *values is exactly* $\oplus$*. Book: Section 16.3.2.*

The Squares from (3, 3)

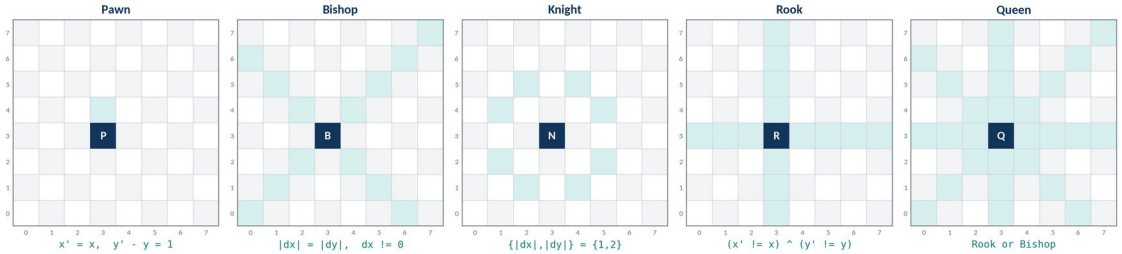


Figure 16.1 The squares each piece may reach from (3, 3) on an empty board, with the condition that decides them.

Figure 16.1: check a few squares against the conditions with a pencil before writing any code. Book: Section 16.3.2.

One Move for Every Piece

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 lives in `Piece` and calls a virtual `moveCondition` that each derived class overrides (Chapter 14). Each class has its own constructor, because constructors are not inherited. Book: Section 16.3.3.

Worked Example 1: the Bishop and the Rook

Bishop from $(0,0)$ to $(3,3)$: $|d_x|=|d_y|=3$, $d_x \neq 0$: moved.

$(3,3) \to (3,5)$: $0 \neq 2$: invalid, it stays. $(3,3) \to (5,1)$: $2=2$: moved.

Rook from $(0,0)$ to $(0,6)$: only y changes: moved. $(0,6) \to (3,3)$: both change, true $\oplus$ true is false: invalid.

$(0,6) \to (7,6)$: moved. $(7,6) \to (8,6)$: off the board: invalid.

`chess.cpp` *prints exactly these verdicts, and* `chess.py` *the same. Book: Sections 16.3.2 and 16.3.4.*

A Tempting Wrong Solution: `||` for the Rook

Listing 16.1 From `wrong_rook.cpp`: the rook's condition with `||` and the bishop's without $d_x \neq 0$.

```
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
}
```

```
rook   (0,6) -> (3,3): 1
rook   (0,6) -> (5,1): 1
bishop (3,3) -> (3,3): 1
```

Listing 16.1: "a row or a column" is not `||`; on a diagonal both coordinates change and `||` accepts it. The wrong bishop accepts standing still. Check yourself: a king's condition is `max(|dx|, |dy|) == 1`. Book: Section 16.3.6.

Problem 16.2: Overloading `add`

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.

$$\text{add}(a_1,\ldots,a_k)=\text{add}(\text{add}(a_1,\ldots,a_{k-1}),a_k) \tag{16.2}$$

Equation (16.2): each overload is built on the one below it. The bags weigh 3,450, 1,200, 5,000, 2,750 and 4,100 g; the sums are 4,650, 9,650, 12,400 and 16,500.

Overload resolution chooses by the number of arguments, so the four functions coexist (Section 13.7). Book: Sections 16.4.1–16.4.3.

The Solution, and Beyond

Listing 16.2 From `add_overload.cpp`: `add` for two to five integers, for two `double`s, and a variadic template.

```
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
}
```

Listing 16.2: the `double` overload has the same count but other types; the fold expression shows where overloading ends. Python needs one function, `def add(*values)`. Book: Sections 16.4.4–16.4.5.

A Tempting Wrong Solution: Defaults Beside an Overload

Listing 16.3 From `wrong_add.cpp`: a defaulted `add` next to the given one; the call with two arguments is ambiguous.

```
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;
}
```

`add(1, 2, 3)` is fine: only the second fits. `add(1, 2)`: both fit exactly, and `g++` stops with **call of overloaded 'add(int, int)' is ambiguous**, naming both candidates.

Listing 16.3: instead of the given function, the defaulted version is neat; beside it, it is an error. Book: Section 16.4.6.

Problem 16.3: Finding Anagrams

> **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* uses the same letters the same number of times: *listen* and *silent*.

$$W = n - m + 1 \tag{16.3}$$

Equation (16.3): a text of length n has W windows of length m, the last starting at $n - m$, provided $m \le n$.

Our own examples: `wash` in `washwash`, where every window is a hit, and `aab` in `ab`, which must give nothing and must not crash. Book: Sections 16.5.1–16.5.2.

Worked Example 2: the Five Windows

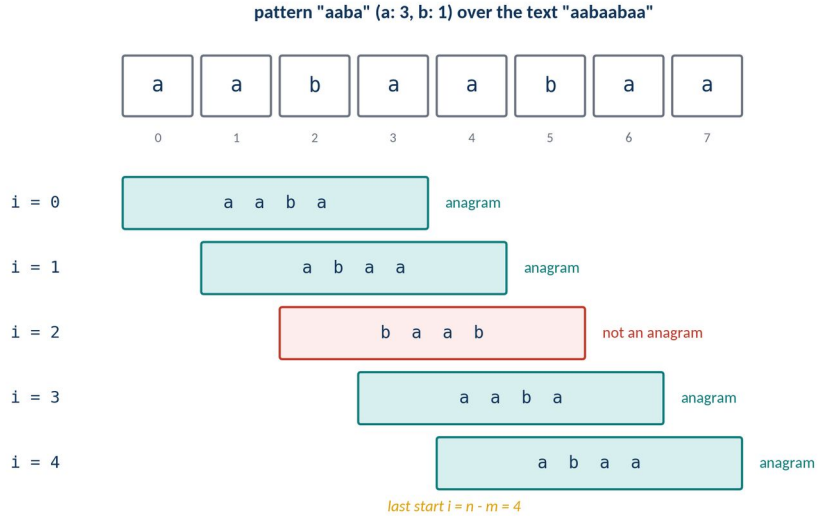


Figure 16.2 The five windows of length 4 in `aabaabaa`: four are anagrams of `aaba`; the last window starts at $n - m = 4$.

Figure 16.2: $n = 8$, $m = 4$, $W = 5$. `aaba` has three `a`s and one `b`; so do the windows at 0, 1, 3 and 4; the window at 2, `baab`, has two `b`s. The statement's answer is confirmed. Book: Section 16.5.2.

Finding Anagrams, on Paper

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
```

Pseudocode 16.2: up to $O((n-m+1)m^2)$ with `is_permutation`. The sliding count keeps 256 counters for the current window and updates two of them per step, $O(n+m)$; the program runs both methods and checks that they agree. Book: Section 16.5.3.

A Tempting Wrong Solution: One Window Too Few

```
[0, 1, 3]
[0, 1, 2, 3]
```

`range(n - m)` in Python, or `i < n - m` in C++, stops at $n - m - 1$: the hit at index 4 is lost in `aabaabaa` and in `washwash`. Nothing warns us; only the statement's own example, which lists index 4, exposes it.

> **WARNING.** In C++, `text.size()` is an unsigned `std::size_t`. If the pattern is longer than the text, `n - m` does not become -1; it wraps around to about 1.8×10^{19}, and the loop reads far past the end. Guard $m > n$ before subtracting.

From `wrong_anagram.py`. Check yourself: `aa` in `aaaa` has $W = 3$ windows, three hits. Book: Section 16.5.6.

Problem 16.4: Matrix Multiplication

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.

$$C_{ij} = \sum_{k=0}^{m-1} A_{ik} B_{kj}, 0 \le i < n, 0 \le j < p \tag{16.4}$$

$$\text{cols}(A) = \text{rows}(B) \tag{16.5}$$

Equation (16.4) for each element; Equation (16.5) is the check: $(n \times m) \times (m \times p) = (n \times p)$, the inner numbers equal and disappear like dominoes. Book: Sections 16.6.1–16.6.2.

Daily Kilograms Times Prices

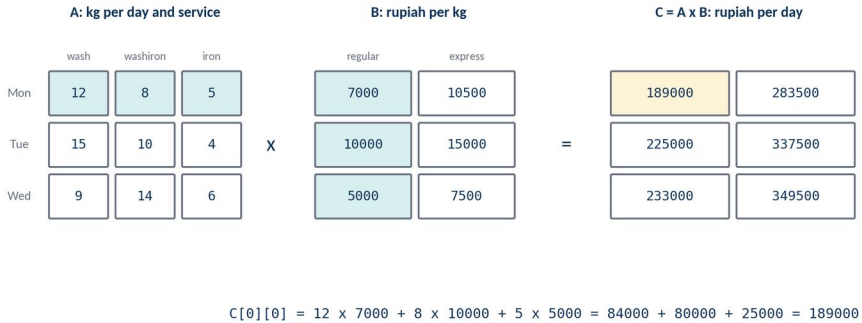


Figure 16.3 Daily kilograms per service times the price per kilogram gives the daily revenue; C_{00} combines row 0 of A with column 0 of B.

Figure 16.3: A holds the kilograms per day (rows) and service (columns); B the price per kilogram, regular and express. Monday's row is $(12, 8, 5)$: $C_{00} = 12 \cdot 7{,}000 + 8 \cdot 10{,}000 + 5 \cdot 5{,}000 = 189{,}000$. Book: Section 16.6.2.

The Triple Loop

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
```

Pseudocode 16.3: $O(nmp)$. *A matrix is a* `std::vector<std::vector<long long>>`; `a[i][k]` *is row* i, *column* k. *Look for a missing check, sums not starting at 0, or swapped indices. By hand:* $\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}\times\begin{pmatrix}7 & 8\\ 9 & 10\\ 11 & 12\end{pmatrix}=\begin{pmatrix}58 & 64\\ 139 & 154\end{pmatrix}$. *Book: Section 16.6.3.*

A Tempting Wrong Solution: `b[j][k]`

```
A x I = [2 3] [5 7]
A x B = [8 18] [19 43]
```

The swapped index multiplies by the *transpose* of B. The identity matrix is its own transpose, so the identity test **passes**.

An asymmetric hand example exposes it: $\begin{pmatrix} 2 & 3 \\ 5 & 7 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ should start with $2+9=11$, not 8.

From `wrong_matrix.cpp`. Check yourself: Wednesday's row $(9, 14, 6)$ gives $C_{20}=233,000$ and $C_{21}=349,500$; a 3×3 times a 2×3 prints "cannot multiply". Book: Section 16.6.6.

Problem 16.5: When Can I Pick It Up?

> **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` with `+`, `-`, `<`, `==` and printing as `Mon 09:15`; for each order print the drop-off, ready, pick-up and waiting times, then list the orders in pick-up order.

Our decisions: day 0 is a Monday; laundry ready at exactly 20:00 can be picked up at 20:00; an hour outside 0–23 or a minute outside 0–59 is refused.

Book: Section 16.7.1.

The Invariant of `Time`

$$\text{day}' = \text{day} + \left\lfloor \frac{\text{minute} + \Delta}{1440} \right\rfloor, \quad \text{minute}' = (\text{minute} + \Delta) \bmod 1440 \tag{16.6}$$

Store a day number and the minutes after midnight, $0 \le \text{minute} < 1440$; Equation (16.6) restores it after every addition.

Difference: $1440 \cdot (\text{day}_a - \text{day}_b) + (\text{minute}_a - \text{minute}_b)$; comparison is that difference against zero.

Pak Budi, wash-and-iron express, Monday 14:40: $720/2 = 360$; $880 + 360 = 1240$ is 20:40, after closing: Tuesday 08:00, a wait of 17 h 20 min.

`+=` is a member applying Equation (16.6); `+` a non-member built on it; `std::stable_sort` sorts the orders by pick-up time and keeps ties in their original order. Book: Sections 16.7.2–16.7.3.

Computing the Pick-up Time

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
```

```
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)
1005 Pak Hendra in Sun 18:00  ready Mon 06:00  pick up Mon 08:00  (14 h 0 min)
Pick-up order: 1004 1001 1002 1003 1005
Time(0, 24, 0) refused: hour must be 0..23
```

Pseudocode 16.4 and selected lines of `pickup_time.cpp`: Pak Hendra's Sunday order crosses into day 7, a Monday, by Equation (16.6), and waits for opening. Book: Sections 16.7.3–16.7.4.

A Tempting Wrong Solution: No Carry

```
14:400 19:210
b ready before closing? True
```

Hours and minutes kept apart, minutes added without carrying: Pak Budi's laundry is "ready at 14:400".

"19:210" is believed to be before closing, because the comparison looks at the hour first.

The fix is not a smarter comparison; it is a representation with an invariant.

From `wrong_time.py`. Check yourself: ironing express on Friday at 18:30 is ready at 20:00 and picked up at once; at 18:31 it waits until Saturday 08:00, 13 h 29 min. Book: Section 16.7.6.

The Problems Side by Side

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

Table 16.1, last column: every trap is caught by a specific example chosen in step 2. The examples you choose before you write code decide which mistakes you will find. Book: Section 16.8.

Common Mistakes

A rule translated from a loose sentence (“or” is not `||`); **changing given code instead of imitating it**.

Default arguments mixed with overloads; **off-by-one in window loops**, and unsigned subtraction.

Testing only with symmetric data; **sums not initialized to 0**.

A representation that allows invalid states; **a stream left with `fill('0')` or `std::left` set**.

Book: Section 16.10.

Summary: Five Things to Remember

1. Seven steps: restate, examples by hand, design, code, run the examples, try to break it, change one thing and predict.

2. Imitate a given hierarchy; write each rule as an exact condition, exclusive or for the rook (Equation (16.1)).

3. Build overloads on each other (Equation (16.2)); never put defaults beside an overload that accepts the same call.

4. Count windows exactly (Equation (16.3)); check dimensions before multiplying (Equation (16.5)); test with asymmetric data.

5. Choose a representation with an invariant (Equation (16.6)), and every operation becomes one line.

Book: Section 16.11.

After This Course

Exercises 16.1–16.6 — a `King`, a `Board` that blocks paths, `add` for `Rupiah`, case-blind anagrams, `transpose`, real Sunday opening hours.

The cashier went from a first `printf` to a complete program with classes, files and a daily report, in three languages.

Next course: Data Structures — the lists, trees and queues met in Chapters 10 and 12, studied in depth, with their costs.

The method outlasts every language: restate, work examples by hand, design before coding, and keep the examples as tests. Book: Sections 16.12–16.13.