CS50-2024

Chapter 1: Week 0 — Scratch, Computational Thinking, and the Foundations of Information

CS50 is Harvard University's introduction to the intellectual enterprises of computer science and the art of programming, and it begins with a deceptively simple promise: you will learn how to solve problems, and you will learn how to express those solutions in a form that a computer can execute. In the first week—often called *week 0*—the course deliberately postpones the syntactic details of traditional programming languages and instead focuses on the underlying ideas that make programming possible at all: how information is represented, how algorithms transform inputs into outputs, and how to reason about correctness and efficiency.

This opening week also sets expectations for what learning computer science tends to feel like in practice. A famous analogy from MIT describes education as "drinking from a fire hose," and that sensation—lots of new terms, many unfamiliar concepts arriving quickly—is normal in an introductory course. The goal is not to absorb everything perfectly on first exposure, but to build comfort over time through repeated practice. A helpful mindset, then, is to get comfortable feeling uncomfortable, especially early on, because "new" is the point of an introduction.

A Course About Learning to Program (Not Just Learning a Language)

Historically, students might have summarized an introductory course by saying, "I learned C." CS50 still teaches C, but it also introduces multiple languages and tools because modern computing problems live in many domains:

- **C**, a lower-level language that reveals how computers manage memory and data in a direct way.
- **Python**, a modern language used widely for data science, scripting, and web development.
- **SQL**, a language for working with databases and structured data.
- **HTML, CSS, and JavaScript**, which together underpin most web applications and many mobile experiences.

The deeper goal is not that you "know C" or "know Python" in isolation, but that you learn the transferable concepts that allow you to teach yourself new languages and tools later. In other words, CS50 aims to teach you *how to program*, which is largely the same as teaching you how to express problem-solving ideas precisely.

Community and Support Structures

CS50 is also structured around community and support. Many students begin with no prior background in computer science, and the course is designed with different "tracks" in problem sets and sections so that students can start from where they are. The course emphasizes progress relative to yourself: what matters is not where you end up compared to classmates, but how far you travel from your own starting point.

Along the way, CS50 culture is punctuated by social and academic events such as CS50 lunches, Puzzle Day (logic puzzles solved in teams), a Hackathon (an overnight build session for final projects), and the CS50 Fair (a showcase of final projects). The course is building toward a capstone final project that you design yourself, with the intention that when you finish CS50, you do not "need CS50" to keep building.

1.1 What Is Computer Science?

At its core, **computer science is the study of information** and the processes by which information can be represented, transformed, stored, and communicated. Since information is central, computer science naturally becomes about **problem solving**: taking some *input* (the given data or situation) and producing some *output* (the desired result).

A simple model captures this:

- **Input → Algorithm → Output**

An **algorithm** is a step-by-step procedure for solving a problem. It can be written in English, described verbally, or illustrated physically. When an algorithm is expressed in a form that a computer can execute, we call that expression **code** (a program). Week 0 is largely about understanding what algorithms and code must operate on: representations of information.

1.2 Representing Information: From Fingers to Bits

Computers ultimately operate using physical phenomena—most commonly electricity. That fact constrains how we represent information inside a machine, but it also provides an elegant foundation.

Unary: Counting With One Symbol

The simplest representation system is **unary**, where each unit is represented by a single mark (or finger). Counting on fingers is a form of unary notation: each raised finger corresponds to "one more."

Unary is straightforward but limited. If you use a single hand and treat each finger as either down or up, you might say you can count to 5. However, if you consider not "how many" fingers are up but *which* fingers are up (each finger being a choice between two states), then five fingers yield:

- $2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 32$ possible configurations.

If one configuration is used for 0, the highest value represented is 31. This observation begins to resemble how computers work: information can be represented by collections of components that are each in one of two possible states.

Binary: Two Symbols, Many Possibilities

Binary is a base-2 representation system that uses only two symbols. By convention, computers use:

- `0` and `1`

These map conveniently to electrical states:

- **0**: no electricity / off
- **1**: electricity present / on

A single `0` or `1` is called a **bit** (short for **binary digit**). Modern computers contain enormous numbers of tiny electrical switches called **transistors**, and each transistor can be treated as storing a bit.

Place Values in Base 10 vs Base 2

To understand binary, it helps to recall how decimal (base 10) works. Consider the decimal number:

`123`

Its meaning depends on *place values*:

- 1×10^2 (hundreds place)
- 2×10^1 (tens place)
- 3×10^0 (ones place)

So:

$$123 = 1 \cdot 100 + 2 \cdot 10 + 3 \cdot 1$$

Binary works the same way, but with powers of 2 instead of powers of 10. With three binary digits, the place values are:

- $2^2 = 4$
- $2^1 = 2$
- $2^0 = 1$

So the pattern `101` in binary means:

$$1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 5$$

Using physical metaphors like light bulbs (each bulb off/on representing 0/1) makes it clear that binary counting is not arbitrary; it is systematic. With three bits, you can represent 0 through 7, and to represent 8 you need a fourth bit (`1000`).

1.3 Bits and Bytes

While a single bit is the fundamental unit, it is often too small to be convenient. A common larger unit is the **byte**, defined as:

- **1 byte = 8 bits**

With 8 bits, you can represent:

- $2^8 = 256$ total patterns

If the smallest pattern is used to represent 0, then the largest number representable with one byte is:

- **255**

This is why you often see the range **0–255** in computing systems: it is the natural range of values representable by one byte.

1.4 Representing Text: ASCII and Unicode

Numbers are only the beginning. Computers must also represent letters, punctuation, and symbols.

ASCII: Mapping Letters to Numbers

One straightforward approach is to **assign each character a number**. This is the idea behind **ASCII**: the **American Standard Code for Information Interchange**.

In ASCII, the capital letter:

- `A` is represented by the number **65**

So, when a computer stores the binary representation of 65 in memory, it can interpret that pattern as the character `A`—provided it is using ASCII (or a compatible encoding) and the context says "treat this as text."

ASCII is sufficient for English letters (upper and lower case), digits, punctuation, and a handful of control characters. But ASCII's common 8-bit storage (256 possible values) is not sufficient for the world's writing systems.

"Hi!" as Numbers Under the Hood

A message like:

- `Hi!`

can be stored as ASCII values:

- `H` = 72
- `I` = 73

- `!` = 33

In binary, each of these is stored as a pattern of 0s and 1s, but humans rarely read those bits directly. The essential idea is that **text is stored as numbers**, and those numbers are stored as bits.

Unicode: A Global Standard for Characters (Including Emoji)

To represent accented characters, non-Latin alphabets, and languages with thousands of symbols, modern systems use **Unicode**, which is best thought of as a superset of ASCII with far more possible characters. Unicode may use:

- 16 bits, 24 bits, or 32 bits per character (often thought of as 2, 3, or 4 bytes)

With 32 bits, a system can represent roughly:

- 2^{32} **4 billion** possible code points

Unicode's mission is broader than convenience: it aims to represent and preserve human languages digitally, past, present, and future, while also enabling symbolic communication such as emoji.

Emoji and Platform Differences Emoji are "characters" in the Unicode sense, but their *appearance* can differ by platform. Apple, Google, Microsoft, Meta, and others may draw the same Unicode character differently, much like different fonts render the same letters differently. For example, the emoji officially named **"face with tears of joy"** is widely popular, but the exact face design varies across devices.

Code Points and Hexadecimal Notation Unicode characters are often referred to using a notation like `U+....`, where `U+` is simply a convention meaning "here comes a Unicode code point." The number after it is frequently written in **hexadecimal** (base 16), which compactly represents large binary values. Hexadecimal uses digits `0-9` and letters `A-F`. The important point in week 0 is not mastering hexadecimal, but recognizing that it is another representation for the same underlying bits.

Skin Tones and Combinations: More Bits, More Meaning Unicode also supports modifiers, such as skin tone variations, using additional code points that adjust how a base character is displayed. Some emoji are also composites formed by joining multiple characters, sometimes using special characters like **zero-width joiners** to indicate that separate symbols should be rendered as one combined glyph (for example, certain family or relationship emoji).

1.5 Context Matters: The Same Bits Can Mean Different Things

A central lesson emerges as soon as we represent text and colors:

> The same pattern of bits can represent different information depending on context.

The ASCII numbers `72 73 33` can mean the text `Hi!` in a messaging context, but if those same values are interpreted as color components, they can represent a shade of yellowish color when treated as **RGB** (red/green/blue) intensities.

In practice:

- A text editor interprets bits as characters.
- A calculator interprets bits as numbers.
- An image editor interprets bits as colors and pixels.

As you begin programming in later weeks, you will often need to tell the computer what *type* of data you intend a sequence of bits to represent, so that it interprets them correctly.

1.6 Representing Images and Color: RGB and Pixels

Most screens create color using **additive color mixing** with three components:

- **R**ed
- **G**reen
- **B**lue

A common representation uses **one byte per component**, so each color channel ranges from 0 to 255. A single dot on the screen is called a **pixel**, and a typical RGB pixel stores:

- 3 bytes = 24 bits total (one byte for red, one for green, one for blue)

This explains, at a high level, why images can be large files: if an image contains about one million pixels and uses three bytes per pixel, then it contains roughly three million bytes (about 3 MB) of color data, even before compression and other file-format details.

When you zoom into a digital image—such as an emoji—you can see the pixel grid, and each pixel corresponds to stored RGB values.

1.7 Representing Sound and Video

Computers do not "understand" music as humans do, but they can represent it numerically.

Sound as Numbers Over Time

One approach to representing music is to encode attributes such as:

- **pitch** (which note)
- **duration** (how long the note lasts)
- **volume** (how loud it is)

More generally, digital audio often represents a sound wave by sampling its amplitude over time, producing sequences of numbers that can be stored as bits.

Video as Many Images

A **video** can be represented as a sequence of still images (frames) displayed quickly, such as 30 frames per second (FPS). When frames are shown rapidly, human perception interprets the changing images as continuous motion.

1.8 Algorithms: Correctness and Efficiency

Once we can represent inputs and outputs, we can focus on the procedures that convert one into the other.

A Phone Book Search as an Algorithm

Consider the problem: find a person's name in a phone book.

A slow but correct algorithm is:

- start at page 1 and flip one page at a time until you find the name (or reach the end)

This is correct, but it can take many steps. Another algorithm flips two pages at a time, but that can miss a name if the target lies on a skipped page—unless you compensate by stepping back when you pass the relevant section.

A much faster strategy is **divide and conquer**, analogous to what searching tools do digitally:

1. open to the middle of the book
2. decide whether the target is earlier or later alphabetically
3. discard half the book
4. repeat on the remaining half

This algorithm is dramatically faster because each step halves the problem size. For a phone book with about 1,000 pages, you can halve roughly 10 times before you narrow to a single page.

Measuring Efficiency With Growth

If you plot:

- problem size on the horizontal axis (how many pages)
- time on the vertical axis (how many steps)

then:

- flipping one page at a time grows linearly: more pages means proportionally more work

- halving the search space grows much more slowly, producing a curve associated with logarithmic growth (specifically log base 2 in this scenario)

Efficiency is not the same as correctness. A "one-step" algorithm that chooses a random page is fast but not reliably correct. In computer science, we care about both:

- **Correctness**: does it always produce the right answer?
- **Efficiency**: how does the required time (or space) grow as inputs get larger?

1.9 Pseudocode and Core Programming Building Blocks

Before writing code in a specific language, it helps to write **pseudocode**: a precise, human-readable description of an algorithm. Pseudocode has no single official format, but it is meant to be clear, structured, and unambiguous enough that someone else could implement it.

A pseudocode version of the phone book algorithm might look like:

```
1. Pick up phone book
2. Open to middle of phone book
3. Look at page
4. If person is on page
5.     Call person
6. Else if person is earlier in book
7.     Open to middle of left half of book
8.     Go back to line 3
9. Else if person is later in book
10.    Open to middle of right half of book
11.    Go back to line 3
12. Else
13.    Quit
```

This example reveals several foundational building blocks that appear across almost all programming languages:

- **Functions**: actions or verbs (e.g., "Pick up," "Open," "Look," "Call").
- **Conditionals**: forks in the road (e.g., "If… else if… else…").
- **Boolean expressions**: yes/no questions that guide conditionals (true/false).
- **Loops**: repeated behavior (e.g., "Go back to line 3").

These ideas are more important than any one language's punctuation or syntax because they transfer directly to C, Python, JavaScript, and beyond.

1.10 Artificial Intelligence, Large Language Models, and the CS50 Duck

Artificial intelligence is a prominent theme in modern computing, and it is relevant to CS50 both intellectually and practically. A simple "chatbot" could be implemented with many explicit conditionals:

```
If the student says "hello"
    say "hello"
Else if the student says "goodbye"
    say "goodbye"
Else if the student asks "how are you"
    say "well"
...
```

But this approach does not scale: it becomes impossible to anticipate every question (for example, every possible question about binary representation). Modern **large language models (LLMs)** instead ingest large amounts of human language (for example, text from the web) and infer statistical patterns that allow them to generate plausible responses.

These systems can be powerful, but they can also make mistakes ("hallucinate"), and their outputs can vary due to randomness and the desire for non-repetitive responses.

Course Guardrails

In CS50's approach for this year, the course emphasizes learning and problem solving rather than outsourcing entire solutions to general-purpose AI systems. For the course's purposes, CS50 provides its own AI-based tool—a chatbot with the personality of a CS50 teaching assistant—available at **cs50.ai**, themed as a **CS50 duck**. The intent is closer to a patient tutor that helps you get unstuck without simply doing the work for you.

This connects to a long-standing programming tradition known as "rubber duck debugging," in which explaining your code aloud—often to an inanimate duck—helps you notice your own errors and clarify your thinking.

1.11 From Bits to Programs: Why Scratch Comes First

If computers "speak" in bits, then any program—no matter how sophisticated—must ultimately be represented as 0s and 1s. For example, even "Hello, world" can be encoded in binary, but humans do not write software at that level. We write software using programming languages that provide abstractions—human-friendly representations that tools translate into machine-level instructions.

CS50 will soon introduce C, where a classic first program looks like:

```c
printf("hello, world\n");
```

Even if you can guess what it does, C also includes syntax that can feel like noise at first: semicolons, braces, `#include`, and other symbols that matter to the compiler but are not the core intellectual ideas.

To focus on ideas rather than punctuation, CS50 begins with **Scratch**, a graphical programming language from MIT. Scratch lets you build algorithms by snapping together blocks, which makes functions, loops, and conditionals tangible and visible.

1.12 Scratch as a Visual Programming Language

Scratch programs are built by combining **blocks** (puzzle-piece shapes) that represent programming constructs. Blocks are grouped by category and color, such as:

- **Motion** (moving sprites)
- **Looks** (speech bubbles, costumes)
- **Sound**
- **Events** (such as "when green flag clicked")
- **Control** (loops and conditionals)
- **Sensing**
- **Operators**
- **Variables**
- **My Blocks** (custom blocks you define)
- **Extensions** (additional capabilities like text-to-speech or video sensing)

Scratch projects typically include one or more **sprites**, which are characters or objects on a stage. The stage uses an (x, y) coordinate system:

- the center is (0, 0)
- y increases upward (top is about 180)
- y decreases downward (bottom is about -180)
- x decreases to the left (about -240)
- x increases to the right (about 240)

This coordinate system makes it possible to describe movement precisely, though many Scratch blocks allow you to work without manual coordinate math.

1.13 A First Scratch Program: "Hello, world"

A minimal Scratch program mirrors the traditional first program in text-based languages:

- **Event**: *when green flag clicked*
- **Action**: *say "hello, world"*

In Scratch terms, the "say" block is a **function-like** action: it performs an operation (displaying a speech bubble). The text inside the block is an **argument** (also called a **parameter**): input that customizes the block's behavior.

This already matches the input–algorithm–output model:

- input: the string `"hello, world"`
- algorithm/function: the "say" operation
- output: a speech bubble appearing on the screen

1.14 Input and Return Values: Asking a Question

Scratch can also request input from the user with an "ask … and wait" block. Conceptually, this block:

1. displays a prompt
2. waits for user input
3. produces a value (the input) that other blocks can use

That produced value is a **return value**, accessible via an "answer" block.

To greet a user by name, a program can:

- ask "What's your name?"
- then say "hello, " combined with the answer

A subtle bug can occur if you try to say "hello" and then immediately say the name in a separate "say" block: the computer may execute the second so quickly that the first is never visible. One fix is to use "say … for 2 seconds," but a cleaner solution is to **compose** the output into a single string using the "join" operator block:

- join `"hello, "` with `answer`
- feed the result into a single "say" block

This illustrates an important programming habit: the output of one operation can become the input of another, and composing smaller pieces carefully often yields clearer behavior.

1.15 Extensions: Text-to-Speech as Another Output Modality

Scratch can be extended with capabilities such as **text-to-speech**. Instead of "say," a program can "speak" the joined greeting aloud. The same underlying structure remains:

- gather input
- combine it with other text
- produce output

Only the output modality changes—from a speech bubble to synthesized audio. This reinforces the idea that programming is often about connecting inputs to outputs through a sequence of transformations, regardless of whether those outputs are visual, textual, or audible.

1.16 Loops, Repetition, and Better Design

If a sprite plays a "meow" sound three times, you could duplicate the "play sound meow" block three times. That works, but it is not a robust design because it repeats code. Repetition becomes error-prone when you later need to change behavior: if you meow ten times and want to adjust the wait time between meows, you may forget to update one copy.

A better approach is to use a **loop**, such as "repeat 3," containing:

- play sound meow
- wait 1 second

This is the same motivation behind efficient algorithms: you want a solution that scales cleanly and minimizes the chance of mistakes.

1.17 Abstraction and Custom Blocks ("My Blocks")

Scratch allows you to define your own blocks under **My Blocks**, which are essentially custom functions. If "meow" is a meaningful action in your program, you can encapsulate its implementation into a single block named "meow," even if internally it plays a sound and waits.

This is **abstraction**: hiding lower-level implementation details so you can think at a higher level. After defining "meow," your main script can become simpler and easier to read.

You can generalize further by giving your block a parameter, such as:

- `meow (n) times`

Internally, the block can include a loop that repeats the sound `n` times. This transforms a specific behavior into a reusable component, which is one of the central practices of programming in any language.

1.18 Conditionals and Event-Driven Interaction

Scratch is naturally **event-driven**, meaning programs often respond to events such as:

- clicking the green flag
- pressing a key
- touching a sprite with the mouse pointer

- detecting motion through a camera

A conditional example is making a cat meow when "petted" (when the mouse pointer touches it). A first attempt might check the condition once, immediately after the green flag is clicked, but that is insufficient because interaction can happen later. The fix is to place the conditional inside a loop such as "forever," so the program keeps checking:

- forever:
  - if touching mouse pointer:
    - play meow

This illustrates a recurring debugging lesson: when a program "does nothing," it is often doing exactly what you asked—just not what you intended. The difference is resolved by being more explicit about timing and repetition.

1.19 Sensing the Physical World: Video Motion

With Scratch extensions such as **video sensing**, a program can react to motion detected by a camera. For instance:

- when video motion > threshold:
  - play meow

Changing the threshold changes sensitivity. Too low a threshold can cause constant triggering; too high may cause no response. This kind of tuning reflects a broader reality in computing: when programs interface with noisy real-world inputs (like camera data), careful design choices are required to produce stable behavior.

1.20 From Blocks to Games: Sprites, Variables, and Incremental Development

Scratch's building blocks are sufficient to create real interactive games. Examples demonstrated in week 0 include:

- a camera-based "whac-a-mole" style game controlled by head movement
- a multi-sprite game involving moving obstacles (Harvard vs Yale vs MIT) with increasing difficulty
- a simple maze-like setup where keyboard input moves a sprite and "walls" block movement

These programs typically involve:

- **multiple sprites**, each with their own scripts
- **variables**, which store values such as score or time remaining
- **randomness**, which keeps gameplay from being identical every time
- **collision detection**, expressed as Boolean expressions like "touching [sprite]?"

Building Big Programs as Many Small Versions: "Oscartime"

A particularly instructive example is a Scratch game developed incrementally, where trash falls from the sky and must be dragged into a trash can. The development process is itself a lesson in problem solving:

1. **Version 1**: build the scene (the stage and major visual elements).
2. **Version 2**: make the trash draggable and animate the trash can lid by switching costumes depending on whether the mouse pointer is touching the can.
3. **Version 3**: make trash fall by repeatedly decreasing its y-coordinate.
4. **Version 4**: detect when trash touches the can and "teleport" it back to the top at a random x-position.
5. **Later versions**: add scoring, timing, sound, and richer interactions.

This approach—reducing a grand idea into achievable steps—is how real software is built. You rarely implement "version 20" directly. Instead, you build a correct version 1, then a correct version 2, and so on, steadily increasing complexity while maintaining control over correctness.

1.21 Where This Chapter Leaves You

By the end of week 0's material, you have the conceptual foundation for the rest of CS50:

- **Information** can be represented with bits.
- **Numbers**, **text**, **colors**, **images**, **audio**, and **video** are all encodings built on top of those bits.
- **Algorithms** are step-by-step procedures that transform inputs into outputs.
- Good solutions require both **correctness** and **efficiency**.
- Programming languages differ in syntax, but most share the same fundamental building blocks:
  - functions
  - conditionals
  - Boolean expressions
  - loops
  - variables
  - abstraction

Scratch serves as the first medium for these ideas because it makes program structure visible. In the next chapter, these same concepts will reappear in a text-based language (C), where you will begin to type programs rather than assemble them from blocks. The intellectual work, however, remains the same: representing information precisely and designing algorithms that reliably and efficiently solve problems.

Chapter 2: Week 1 — C, Compilation, Control Flow, and the Limits of Representation

Week 0 deliberately focused on programming ideas without the distractions of punctuation, file systems, and compilers. In week 1, CS50 keeps the *ideas* from Scratch—functions, conditionals, loops, variables, return values, and abstraction—but now expresses those ideas in a traditional text-based language: **C**. C can look cryptic at first, largely because it makes you write down details that Scratch hides behind blocks, but those details are also what make C a powerful lens for understanding what computers are actually doing underneath the hood.

The goal of learning C in CS50 is not "to memorize C," but to add a new set of tools to your problem-solving toolkit. Along the way, you also begin to see a second major theme of computer science: computers are not magical, and they are not infinitely precise. Their memory is finite, their number representations have limits, and those limits can cause real failures in real systems.

2.1 From Scratch Blocks to C Text: Same Concepts, New Syntax

In Scratch, a first program might be:

- **when green flag clicked**
- **say** `"hello, world"`

In C, the analogous program uses a function named `printf` (pronounced "print f," where the `f` hints at *formatted* printing):

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

At first glance, this is more visually complicated than Scratch. It contains symbols you do not normally use in English writing—parentheses, curly braces, semicolons, and a `#include` line. Early on, it helps to treat much of this as required "boilerplate," while keeping your attention on the central idea:

- `printf("hello, world\n");` is the spiritual equivalent of Scratch's **say** block.

The rest of the file is the surrounding structure that C requires so that the computer knows where your program begins, and what tools (libraries) you intend to use.

2.2 Source Code vs. Machine Code, and the Role of a Compiler

A computer ultimately executes **machine code**: instructions represented as patterns of bits (0s and 1s). Humans, however, write **source code**, which is the human-readable text you type in a language like C.

To get from source code to machine code, you use a **compiler**:

- A **compiler** is a program that translates code written in one language (C source code) into another language (machine code) that the computer can execute.

This translation step is essential. Without it, you would have to write programs directly in 0s and 1s—possible in principle, but painfully slow and error-prone in practice.

2.3 The CS50 Development Environment: `cs50.dev`, VS Code, and the Terminal

Rather than requiring every student to install identical tools on their own laptops (which vary across Windows, macOS, Linux, versions, and configurations), CS50 standardizes your environment using the cloud:

- You work at **https://cs50.dev**, a browser-based environment that includes:
  - **Visual Studio Code (VS Code)**, a widely used professional code editor.
  - A pre-configured system with compilers and tools you need for the course.

2.3.1 GUI vs. CLI

In this environment you use both:

- A **graphical user interface (GUI)**: tabs, file explorer panes, clickable elements.
- A **command-line interface (CLI)** (also called a **terminal** or **console**): a text-based interface where you type commands.

A CLI can feel like a step backward if you are used to clicking icons, but it is often faster once you gain comfort with it, particularly for repetitive developer tasks like compiling, running, and moving files.

2.3.2 Your Prompt and the `$`

In many terminals you will see a symbol such as:

- `$`

This is called a **prompt**. It is not currency; it is simply a convention indicating where you should type commands.

2.4 Your First C Program: Writing, Compiling, Running

CS50 introduces a simple three-step rhythm for C programs:

1. **Write** the code (create/edit a `.c` file)
2. **Compile** the code (convert it to machine code)
3. **Run** the program (execute it)

In CS50's environment, those steps often look like this:

```
code hello.c
make hello
./hello
```

2.4.1 Creating a File: `code hello.c`

The command:

```
code hello.c
```

opens (or creates) a file named `hello.c`. The `.c` extension is the conventional file extension for C source code.

File-naming conventions that save time later When working in a terminal, certain habits prevent avoidable problems:

- Prefer **lowercase** filenames.
- Avoid **spaces** in filenames.
- Keep extensions lowercase as well.

These are not laws of nature, but they reduce friction because spaces and unusual characters complicate command-line typing.

2.4.2 Compiling: `make hello`

After writing code, you compile it:

```
make hello
```

A common beginner mistake is to type `make hello.c`. In CS50's workflow, you typically compile by naming the *program you want to produce* (`hello`), not the source file. The tool `make` will look for `hello.c` and produce an executable program named `hello`.

When compilation succeeds, it often prints **nothing**. Paradoxically, that silence is good news. If compilation fails, you may see many lines of error messages; those messages are trying to point you toward mistakes in your code.

2.4.3 Running: `./hello`

To run the compiled program:

`./hello`

The `./` means "run the program named `hello` located in the current folder."

This is conceptually similar to double-clicking an application icon, but in the CLI you explicitly state where the program is located.

2.5 `printf`, Newlines, and Escape Sequences

If you write:

```
printf("hello, world");
```

your output may appear immediately followed by your next terminal prompt (`$`) on the same line. That is not "wrong" as far as the computer is concerned, but it is usually undesirable formatting for humans.

To print a newline at the end, you include:

- `\n`

```
printf("hello, world\n");
```

2.5.1 What `\n` actually is

`\n` is an **escape sequence**, meaning it is a special two-character pattern inside a string that represents a single special character (a newline). It is "escaped" because it would be hard or inconvenient to type the newline character directly inside the quotes as part of the string.

2.5.2 Printing a literal backslash (escaping the escape)

Sometimes you want to print characters that normally have special meaning. A common pattern in programming is:

- To print a literal backslash, you often need to escape it with another backslash.

For example, printing `\n` literally (as two visible characters) involves using `\\n` in many contexts.

2.6 Libraries, Header Files, and `#include`

C comes with many useful functions, but those functions are organized into **libraries**—bundles of code written by other people that you can reuse rather than reinvent.

To use functions from a library, you often include a **header file**, typically with a `.h` extension:

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

- `stdio.h` is the "standard input/output" header.
- `printf` is associated with this library, so including `stdio.h` tells the compiler that `printf` exists and how it is meant to be used.

A helpful mental model early on is:

- `#include <...>` is like telling the compiler: "Before compiling, bring in the definitions I need from this library."

2.6.1 Documentation: manual pages and `manual.cs50.io`

Professional documentation for C exists in "manual pages," but those are often written for experienced programmers. CS50 provides friendlier documentation at:

- **https://manual.cs50.io**

This documentation is especially useful when you know *what you want to do* ("get input," "print formatted output") but you do not yet remember the exact function names or argument patterns.

2.7 The CS50 Library: Training Wheels for Input (`cs50.h`)

C is an older language, and it does not make user input easy for beginners. To keep your focus on problem-solving rather than low-level input parsing, CS50 provides its own library with helper functions, included via:

```
#include <cs50.h>
```

This gives you functions such as:

- `get_string` — prompt the user for text
- `get_int` — prompt the user for an integer
- `get_float` — prompt the user for a floating-point number
- `get_char` — prompt the user for a single character
- and others

These are "training wheels" in the sense that they simplify early weeks; later in the course, you will learn what these functions are doing and how to work without them.

2.8 Variables and Types: `string`, `int`, `char`, `float`, `double`

In Scratch, you can create a variable without specifying what kind of value it stores. In C, you generally must specify a variable's **type**, which tells the compiler how to represent the value in memory.

2.8.1 A greeting program with input

In Scratch, the "ask … and wait" block returns a value (the user's answer). In C with CS50's library, the analogous program is:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string answer = get_string("What's your name? ");
    printf("hello, %s\n", answer);
}
```

Here:

- `get_string("What's your name? ")` prompts the user and **returns** the typed text.
- `string answer = ...` stores that return value in a variable named `answer`.
- `printf("hello, %s\n", answer);` prints a formatted string where `%s` is replaced by the value of `answer`.

2.8.2 Strings, format codes, and the "formatted" in `printf`

Unlike Scratch's "join" block, C often uses **format codes** inside a format string:

- `%s` means "insert a **string** here"
- `%i` means "insert an **integer** here"
- `%f` means "insert a **floating-point number** here"
- `%li` is commonly used for a **long integer** (more on "long" later)

The important idea is that `printf` can take multiple arguments:

```
printf("hello, %s\n", answer);
```

- The first argument is the format string: `"hello, %s\n"`
- The second argument is the value to plug in where `%s` appears: `answer`

2.8.3 Single characters vs. strings: `char` and quotes

C distinguishes between:

- A **string**: a sequence of characters, written with **double quotes** `"..."`.
- A **char**: a single character, written with **single quotes** `'...'`.

Example:

```
char c = get_char("Do you agree? ");
```

Later, when comparing:

```c
if (c == 'y')
{
    printf("Agreed\n");
}
```

Using `'y'` (single quotes) is essential here, because it is one character, not a whole string.

2.9 Conditionals and Boolean Expressions in C

Scratch's conditionals map directly to C conditionals. In C, the structure is:

```c
if (condition)
{
    // code if condition is true
}
else
{
    // code if condition is false
}
```

The `condition` is a **Boolean expression**: a yes/no question that evaluates to true or false.

2.9.1 Comparing values: <, >, and == vs. =

In C:

- `=` is **assignment** ("put the value on the right into the variable on the left")
- `==` is **equality comparison** ("are these two values equal?")

This difference matters. For example:

```c
if (x == y)
{
    printf("x is equal to y\n");
}
```

Using `=` here would mean something else entirely, and is a classic source of bugs.

2.9.2 A comparison program: `compare.c`

A simple program that compares two integers might look like:

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
```

```
{
    int x = get_int("What's x? ");
    int y = get_int("What's y? ");

    if (x < y)
    {
        printf("x is less than y\n");
    }
    else if (x > y)
    {
        printf("x is greater than y\n");
    }
    else
    {
        printf("x is equal to y\n");
    }
}
```

Notice a design point: the final case does not need to test `x == y`. If `x` is not less than `y` and not greater than `y`, then it must be equal. Using `else` is both simpler and typically more efficient.

2.9.3 Input validation (and why CS50's `get_int` helps)

If the user types something that is not a number (like `"cats"`), CS50's `get_int` will keep prompting until it receives a valid integer. This is *not* C's default behavior; it is part of the CS50 library's beginner-friendly design.

2.10 Logical Operators: "or" (`||`) and "and" (`&&`)

When checking multiple conditions, C uses:

- `||` for logical **or**
- `&&` for logical **and**

For example, to accept either lowercase or uppercase yes:

```
if (c == 'y' || c == 'Y')
{
    printf("Agreed\n");
}
else if (c == 'n' || c == 'N')
{
    printf("Not agreed\n");
```

```
}
```

This improves design by avoiding duplicated `printf` statements and reducing the chance you update one branch but forget another.

2.11 Loops in C: `while`, `for`, and `do ... while`

Loops are how you make the computer do something repeatedly without copy/pasting the same line many times.

2.11.1 `while` loops

A `while` loop repeats as long as its condition remains true:

```
int counter = 3;

while (counter > 0)
{
    printf("meow\n");
    counter--;
}
```

This pattern has three key parts:

1. Initialize a variable (`counter = 3`)
2. Check a condition (`counter > 0`)
3. Update the variable (`counter--`) so the loop eventually stops

C supports several equivalent update styles:

```
counter = counter - 1;
counter -= 1;
counter--;
```

Similarly, increments can be written as `counter++`.

2.11.2 `for` loops

A `for` loop packages initialization, condition, and update into one line:

```
for (int i = 0; i < 3; i++)
{
    printf("meow\n");
}
```

You can read this as:

- Start `i` at 0.
- While `i` is less than 3, do the loop body.
- After each iteration, increase `i` by 1.

This is often the most common loop style when you know in advance how many times you want to repeat something.

2.11.3 Infinite loops

Sometimes you want a loop that never ends. One pattern is:

```
while (true)
{
    printf("meow\n");
}
```

The idea is simple: if the condition is always true, the loop never exits. (In standard C, using `true` typically involves including `<stdbool.h>`, though some environments and libraries make it available automatically.)

2.11.4 `do ... while` loops (prompt at least once)

Sometimes you want to do something *at least once* before checking whether it should repeat. That is exactly what a `do ... while` loop expresses:

```
int n;

do
{
    n = get_int("Size: ");
}
while (n < 1);
```

This structure is especially useful for input validation: it guarantees the user is prompted once, and then prompted again only if their input does not meet the constraint.

2.12 Abstraction with Functions: Creating Your Own Building Blocks

Scratch lets you create custom blocks ("My Blocks"). C lets you create your own **functions**, which serve the same purpose: they let you name a behavior and reuse it.

2.12.1 A simple function with no return value (`void`)

A function that only produces a side effect (like printing) and does not return a value is often declared with `void`:

```c
void meow(void)
{
    printf("meow\n");
}
```

- The first `void` means the function returns nothing.
- The `(void)` means the function takes no arguments.

Then you can call it from `main`:

```c
int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
}
```

2.12.2 Function prototypes (so order doesn't trap you)

C is read top-to-bottom. If you call `meow()` before the compiler has seen the definition of `meow`, you will get an error.

One solution is to define `meow` above `main`, but that pushes `main` downward, which is inconvenient because `main` is the entry point you typically want to find quickly.

A better solution is to write a **function prototype** above `main`:

```c
#include <stdio.h>

void meow(void);

int main(void)
{
    meow();
}

void meow(void)
{
    printf("meow\n");
}
```

A prototype is a promise to the compiler: "this function exists later, and it has this signature."

2.12.3 Functions with parameters (inputs)

To generalize `meow` so that it can meow any number of times:

```c
void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");
    }
}
```

Then `main` becomes a high-level description:

```c
int main(void)
{
    meow(3);
}
```

This is abstraction in action: `main` can now express *what* should happen without being cluttered by *how* it happens.

2.13 Return Values and Scope: When Functions Compute Answers

So far, many examples have focused on side effects (printing to the screen). But functions can also compute a value and hand it back to the caller via a **return value**.

2.13.1 A simple calculator and formatted printing

Adding two integers:

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int x = get_int("x: ");
    int y = get_int("y: ");

    printf("%i\n", x + y);
}
```

Here `%i` tells `printf` to print an integer.

2.13.2 Writing an `add` function (and discovering scope)

A first attempt at an `add` function might incorrectly try to use variables from `main`:

```c
int add(void)
{
    return x + y;   // ERROR: x and y are not in scope here
}
```

This fails because of **scope**.

- **Scope** is the region of a program where a variable name is valid and accessible.
- In C, variables declared inside one set of curly braces `{ ... }` are generally not accessible outside those braces.

`x` and `y` exist inside `main`'s braces, not inside `add`'s braces. To fix this, you pass values as parameters:

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

Then call it like:

```c
int z = add(x, y);
printf("%i\n", z);
```

And, if you want, you can even avoid the extra variable and "nest" the call:

```c
printf("%i\n", add(x, y));
```

2.13.3 Why `main` is declared `int main(void)`

You have repeatedly seen:

```c
int main(void)
```

Even if your program does not explicitly return a value, the convention is that `main` returns an integer status code to the operating system:

- `0` typically means "success."
- Nonzero values often indicate some kind of error.

This is one reason why you may sometimes see numeric "error codes" reported by software: they originate from return values used as signals.

2.14 Working with the Linux Command Line: Files and Folders as Commands

When you use `cs50.dev`, you are effectively working on your own Linux-based environment in the cloud (technically, a container). The terminal commands you type are commands executed in that environment.

Some foundational commands include:

- `ls` — list files in the current directory
- `cd` — change directory
- `mv` — move (also used to rename)
- `cp` — copy
- `rm` — remove (delete) files
- `mkdir` — create a directory
- `rmdir` — remove a directory (when empty)
- `clear` — clear the terminal screen

2.14.1 Renaming a file with `mv`

To rename `meow.c` to `woof.c`:

```
mv meow.c woof.c
```

This is the CLI equivalent of right-click → rename in a GUI.

2.14.2 Deleting a file with `rm`

To delete a file:

```
rm hello.c
```

Be careful: deletion is often difficult to undo. Many command-line tools assume that if you type a deletion command, you mean it.

2.14.3 `./program` and the meaning of `.`

In paths:

- `.` means "this directory."
- `..` means "the parent directory."

That is why you run your program with:

```
./hello
```

You are explicitly saying: "run `hello` from here."

2.14.4 Terminal productivity features: history and autocomplete

Two practical features you will naturally start using:

- The **up arrow** cycles through previous commands.
- **Tab completion** auto-completes filenames and commands when the prefix is unambiguous.

These do not change what you can do, but they dramatically reduce typing.

2.15 Building Patterns: Mario, Grids, and Nested Loops

CS50 often uses *Mario* as a playful setting for pattern-printing problems, because they force you to think precisely about rows, columns, and repetition.

2.15.1 Printing a row of question marks

A hardcoded approach:

```
printf("????\n");
```

A loop-based approach prints one character repeatedly:

```
for (int i = 0; i < 4; i++)
{
    printf("?");
}
printf("\n");
```

This illustrates a small but important detail: if you print a newline inside the loop, you get one character per line. If you want one row, print the newline *after* the loop.

2.15.2 Printing a 2D grid: nested loops

To print a 3×3 grid using `#` characters:

```
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        printf("#");
    }
    printf("\n");
}
```

The structure matches how screens are typically written in text:

- The outer loop iterates over **rows**.
- The inner loop iterates over **columns**.
- After each row, print a newline.

2.15.3 Avoiding "magic numbers" with a variable

If the size is used in multiple places, writing `3` repeatedly is fragile. A common improvement is to use a single variable:

```
int n = 3;
```

```c
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        printf("#");
    }
    printf("\n");
}
```

2.15.4 Making the program dynamic with input (and validating it)

With CS50's `get_int`, you can ask the user for the size:

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int n;

    do
    {
        n = get_int("Size: ");
    }
    while (n < 1);

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

The `do ... while` avoids duplicating the prompt, because it naturally expresses: "prompt once, and keep prompting while the input is invalid."

2.16 Comments in C

C supports **comments**, which are notes for humans that the compiler ignores.

A common single-line comment begins with `//`:

```
// prompt user for positive integer
```

Comments do not change what the program does, but they can clarify *why* the program is written the way it is, which matters when you return to your code later or share it with others.

(As a contrast: in Python, `#` starts a comment, but in C `#` is used for directives like `#include`.)

2.17 The Limits of Computers: Memory Is Finite

After learning how to write programs that seem to "do math," it is crucial to confront a deeper reality: a computer has a finite amount of memory, so it cannot represent all numbers perfectly or count forever without consequences.

In practice, limitations often show up in two forms:

1. **Integer overflow** (counting too high with a fixed number of bits)
2. **Floating-point imprecision** (representing real numbers approximately)

2.18 Integer Overflow: When Counting Wraps Around

If you only have a fixed number of digits or bits, you can represent only a fixed range of numbers.

A vivid mental model is a fixed-width counter. If you can store only three digits, you can count:

- 000, 001, 002, …, 999

But then the next count would "wrap" because there is nowhere to store the carry.

Computers store integers with a fixed number of bits. A common size is **32 bits** for an `int` (by modern convention). With 32 bits:

- There are 2^{32} possible bit patterns, roughly **4 billion**.

If the system uses only nonnegative values, the largest representable number is:

- **4,294,967,295**

If the system also represents negative values (as most do), then roughly half the patterns are used for negatives, leaving a maximum around **2 billion** for positive values.

When you exceed the maximum representable value, the number does not become "bigger than maximum." Instead, it typically **wraps around**, producing a value that is suddenly small again (often 0 or a negative number). That phenomenon is **integer overflow**.

2.18.1 Larger integers: `long` and 64 bits

If 32-bit integers are not enough, many systems support 64-bit integers, commonly accessed in C as `long` (or related types depending on platform). With 64 bits, the range is astronomically larger—not merely doubled, but exponentially expanded.

CS50's library even provides `get_long`, and `printf` can print long integers using `%li`.

2.19 Truncation: Integer Division Loses the Fraction

Consider dividing:

- $1 \div 3 = 0.3333...$

If you do integer division in C, the result is truncated (the fractional part is discarded). For example, if `x` and `y` are integers:

```
int x = 1;
int y = 3;
printf("%i\n", x / y);
```

This prints:

- `0`

This is not rounding; it is **truncation**. Everything after the decimal point is thrown away because the computation is being performed in integer arithmetic.

2.20 Floating-Point Values: `float`, `double`, and Imprecision

To preserve fractional values, you use floating-point types:

- `float` (often 32 bits)
- `double` (often 64 bits, more precision)

A first instinct might be to simply print with `%f`, but format codes must match types. If you tell `printf` to expect a floating-point value while passing it an `int`, the compiler may warn you or error.

A common approach is to **cast** integers to floating-point before division.

2.20.1 Casting to avoid truncation

```
int x = get_int("x: ");
int y = get_int("y: ");
```

```
double z = (double) x / (double) y;
printf("%f\n", z);
```

- `(double) x` means "treat `x` as a double for this expression."
- Now the division is performed in floating-point arithmetic, so the fractional part is preserved.

2.20.2 Controlling decimal places in `printf`

`printf` supports formatting precision. For example:

- `%.5f` prints 5 digits after the decimal point.

```
printf("%.5f\n", z);
```

2.20.3 Floating-point imprecision: why you don't get infinite 3's

Even with `double`, you may see:

- `0.33333333333333331483`

instead of a perfect infinite repetition of `3`.

This is **floating-point imprecision**. The computer is not "bad at math"; it is representing a real number using a finite number of bits, and most real numbers cannot be represented exactly in binary floating-point form. The stored value is the closest representable approximation, and printing many digits reveals that approximation.

2.21 Real-World Consequences of Representation Limits

These limitations are not merely academic. They have produced major historical and engineering problems.

2.21.1 Y2K: when two digits weren't enough

Many older systems stored years using two digits (e.g., `99` for 1999). Memory used to be expensive, and saving bytes mattered. But when the year rolled over to 2000, a system storing only two digits could interpret `00` as 1900 rather than 2000, causing date comparisons and time-based logic to fail.

2.21.2 The 2038 Problem: when 32-bit time overflows

Many systems track time as the number of seconds since an "epoch," commonly:

- January 1, 1970

If time is stored in a 32-bit integer, eventually the counter overflows, and the time representation wraps around, potentially making systems think it is the early 1900s again. This is predicted to occur in **2038** for many 32-bit time representations.

A common mitigation is to switch to **64-bit** time counters, which pushes the overflow so far into the future (on the order of hundreds of millions of years) that it is no longer a practical concern—though it is still, fundamentally, finite.

2.21.3 Bugs in games: Pac-Man and Donkey Kong

Representation limits also surfaced in classic games:

- **Pac-Man**: reaching a sufficiently high level (notably around level 256) could trigger overflow-like behavior that corrupted the game state and display.
- **Donkey Kong**: an internal time calculation could wrap in a way that left only a few seconds to complete a level, making progress impossible.

The underlying lesson is the same: if you store a value in a type that cannot represent the range you eventually reach, behavior becomes incorrect in surprising ways.

2.21.4 Boeing 787: overflow and the "turn it off and on" fix

A particularly striking real-world example involved the Boeing 787. Documentation described a scenario in which, after a certain amount of continuous power-on time (248 days), a counter could overflow, leading systems to enter failsafe modes simultaneously—causing loss of power.

The short-term workaround echoed a familiar computing trope:

- power cycle the system (turn it off and back on)

Restarting resets memory, and thus resets counters. Longer-term, the solution required a software update to prevent the overflow condition.

2.22 Where This Chapter Leaves You

By the end of week 1, you have taken the core ideas from Scratch and expressed them in C:

- You can write **source code** and understand that it must be compiled into **machine code**.
- You can use a development environment (`cs50.dev`) that combines a code editor and a terminal.
- You can print output with **`printf`**, including special characters via **escape sequences** like `\n`.
- You can use **libraries** via **header files** (`stdio.h`, `cs50.h`) and consult documentation via **manual pages**.
- You can store data in **variables** with explicit **types** (`int`, `string`, `char`, `float`, `double`).
- You can make decisions with **conditionals**, compare values with `==` (not `=`), and combine logic with `||` and `&&`.
- You can repeat actions with **loops** (`while`, `for`, `do ... while`) and build two-dimensional patterns using **nested loops**.

- You can write your own **functions**, use **prototypes**, pass **arguments**, and return values—while respecting **scope**.
- You understand that computers have **finite memory**, leading to **integer overflow**, **truncation**, and **floating-point imprecision**, and that these constraints have caused real failures in the world.

In the next stage of the course, these foundations become the basis for more complex programs, deeper reasoning about memory, and a clearer understanding of what "dangerous" and "powerful" mean when working close to the machine.

Chapter 3: Week 2 — Arrays, Memory, Debugging, and Strings

Week 1 introduced C as a more explicit, lower-level way to express the same ideas you already used in Scratch: variables, conditionals, loops, functions, and return values. Week 2 keeps using those building blocks, but begins to shift your perspective toward what C is *really* good at teaching: how data is laid out in memory, how you can organize related values, and how seemingly "high-level" things like text are ultimately represented as bytes.

This week also begins to broaden from fundamentals into applications. Two motivating themes appear early:

- **Text analysis**, such as estimating the reading level of a passage by measuring properties like word length and sentence length.
- **Cryptography**, the art and science of scrambling information so that intercepted messages remain unreadable without a key.

To do both well, you need to become comfortable with a core abstraction that sits between "one variable" and "many variables": the **array**.

3.1 From "Make It Work" to "Understand What Happened": Under the Hood of Compilation

In week 1, it was enough to accept a simple story:

- You write C source code.
- You run `make`.
- You get an executable program.

That story is useful, but it hides details that matter when something goes wrong, or when you need to do something slightly unusual. Week 2 peels back part of that abstraction, not to make your life harder, but to make your mental model more reliable.

3.1.1 `make` is not the compiler

The tool you have been using:

```
make hello
```

is not itself the C compiler. Instead, `make` is a program that *runs other programs* for you automatically, using rules that save you from repetitive typing.

In CS50's environment, the underlying compiler `make` typically invokes is **Clang** (short for "C language"), which you can run directly:

```
clang hello.c
```

If you do that, however, you will immediately see why `make` is convenient.

The `a.out` default If you run:

```
clang hello.c
```

Clang will compile successfully, but the output file will not be named `hello`. By default, Clang produces an executable named:

- `a.out`

So you would run it with:

```
./a.out
```

That name is historically rooted in "assembler output," and it is a reminder that compilation involves multiple stages (which you will see soon). But as a human user, `a.out` is a terrible name for a program you intend to run regularly.

Naming the output with `-o` To name the compiled program explicitly, you pass a **command-line argument** (an extra word or symbol after the program name that modifies behavior):

```
clang -o hello hello.c
```

- `-o hello` means "output an executable named `hello`."
- `hello.c` is the input file.

Now you can run:

```
./hello
```

3.1.2 When libraries are involved: "undefined reference" and linking

Consider a program that uses CS50's library:

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string name = get_string("What's your name? ");
```

```
    printf("hello, %s\n", name);
}
```

If you compile this with `make hello`, it works, because `make` adds the right settings automatically.

But if you compile manually with Clang:

```
clang -o hello hello.c
```

you will likely see an error mentioning an **undefined reference** to `get_string`. Including `cs50.h` was necessary, but it was not sufficient. The header file tells the compiler that the function exists (its name, return type, parameter types), but you still need to tell the build process to include the *actual compiled code* for that function.

To do that, you tell Clang to link against the CS50 library:

```
clang -o hello hello.c -lcs50
```

- `-lcs50` means "link with the cs50 library."

This is one of the big reasons `make` is so helpful: it quietly supplies flags like `-o` and `-lcs50` for you.

3.2 "Compiling" is Really Four Steps

In everyday conversation, programmers use "compile" to mean "turn my source code into an executable." Technically, that end-to-end process includes four distinct stages, each representing a layer of tools that have evolved over decades.

3.2.1 Step 1: Preprocessing

Lines that begin with `#` are not "normal C statements." They are **preprocessor directives**, and they are handled before the rest of the code is compiled.

For example:

```
#include <cs50.h>
#include <stdio.h>
```

During **preprocessing**, the compiler effectively does a "copy/paste in memory":

- `#include <cs50.h>` is replaced with the contents of `cs50.h` (notably function prototypes such as `get_string`).
- `#include <stdio.h>` is replaced with the contents of `stdio.h` (notably the prototype for `printf`, among many other things).

This does **not** permanently modify your `.c` file; it is a transformation performed during the build process so that the compiler knows what functions exist and how they should be called.

This also explains why **function prototypes** are necessary in your own code. If you call a function before the compiler has seen its prototype or definition, the compiler does not yet know what that function is.

3.2.2 Step 2: Compiling (in the narrow sense)

After preprocessing, the code is translated from C into **assembly language**, which is a much lower-level, hardware-oriented language. Assembly is still human-readable (unlike raw bits), but it is far closer to what the CPU actually executes.

In assembly, you will still see recognizable names like `printf` and `get_string`, because the compiled code must still refer to those functions. But you will also see low-level instructions for moving values, calling functions, and manipulating memory.

You will not write assembly in CS50, but it is important to know that C is not translated *directly* into 0s and 1s in a single leap.

3.2.3 Step 3: Assembling

The assembly language is then converted into **machine code**: the actual 0s and 1s that the CPU executes.

This is the step that produces "real" binary instructions, but the result is still not necessarily a single finished program file—especially if your program depends on other compiled code.

3.2.4 Step 4: Linking

Even a small program usually relies on other code:

- Your file, like `hello.c`
- The CS50 library implementation (conceptually something like `cs50.c`, compiled)
- The standard I/O library implementation (conceptually something like `stdio.c`, compiled)

Your source file calls functions like `get_string` and `printf`, but the compiled machine code for those functions lives elsewhere. **Linking** is the step that combines multiple compiled pieces into one final executable.

This is why `-lcs50` matters: it tells the linker, "include the compiled CS50 library code so references to `get_string` can be resolved."

3.3 Why "Decompiling" Is Hard (and Why That Matters)

If compilation turns source code into machine code, you might wonder: can you reverse it? Could you take a program's 0s and 1s and recover the original C source?

In practice, reverse engineering is possible only imperfectly, and it is often painful because so much meaning is lost:

- Variable names are usually gone.
- Function names may be missing or obscured.
- Many different C programs can compile into machine code that is functionally similar.

Even if you can recover *some* C-like output, it is often difficult to read, and a skilled programmer might find it easier to re-implement the program than to reconstruct it from compiled output.

This observation becomes more relevant later in the course when you encounter languages whose code is more directly distributed in readable form (for example, web-facing JavaScript, or some forms of Python distribution).

3.4 Debugging: From "Printf Everywhere" to Using a Real Debugger

Writing correct code on the first try is rare, even for experienced programmers. What changes with experience is not the absence of bugs, but the ability to find and fix them systematically.

3.4.1 A brief origin story: why we call them "bugs"

The term "bug" is famously associated with Admiral Grace Hopper and an early computer logbook entry describing a moth found in a relay of the Harvard Mark II. Whether or not it was the first "bug," the story helped popularize the word as a metaphor for defects in programs.

3.4.2 Debugging technique #1: `printf` as a probe

A quick way to understand what your program is doing is to print intermediate values.

Consider this loop intended to print three bricks:

```
for (int i = 0; i <= 3; i++)
{
    printf("#\n");
}
```

The code compiles and runs, but prints **four** bricks. One of the fastest ways to see why is to add a diagnostic print:

```
printf("i is %i\n", i);
```

Then you can observe that `i` takes the values 0, 1, 2, and 3, and that the `<= 3` condition includes four iterations. Changing `<=` to `<` fixes the logic.

This technique is valuable, but it can also become chaotic if you start printing everything everywhere. When that happens, you want a better tool.

3.4.3 Debugging technique #2: `debug50` and breakpoints

A **debugger** lets you pause a program while it runs and execute it step by step. In CS50's VS Code environment, you can start the debugger easily with:

```
debug50 ./buggy
```

A debugger is most useful when you place **breakpoints**, which are markers that tell the debugger where to pause execution. In VS Code, you set a breakpoint by clicking in the left margin (the "gutter") next to a line number, producing a red dot.

Once paused, you can:

- **Step over** a line (execute it without diving into function internals).
- **Step into** a function call (enter the function and debug inside it).
- Observe variable values changing over time.

Garbage values and uninitialized variables While debugging, you may see that a variable has a strange value before you assign anything to it. For example, before you execute a line like:

```c
int h = get_int("Height: ");
```

the debugger might show `h` as some seemingly random number. This is a **garbage value**, meaning the variable's memory contains leftover data from earlier usage.

In C, you should not assume a variable has a meaningful value unless you have explicitly set it.

3.4.4 Debugging technique #3: rubber duck debugging

Sometimes the "tool" is not technical at all. **Rubber duck debugging** is the practice of explaining your code out loud—often to an inanimate object—until the logic error becomes obvious.

The key idea is that your brain often skips steps silently when thinking, but cannot skip steps as easily when speaking. The act of verbalizing the reasoning can surface contradictions.

CS50 also provides a digital version of this idea via the CS50 Duck (accessible at `cs50.ai` and in the `cs50.dev` interface), which you can ask about concepts or snippets of code.

3.5 Types, Bytes, and the Memory "Canvas"

Arrays matter because memory is finite and structured. To use arrays well, you need a practical model of memory.

3.5.1 Common type sizes (rules of thumb)

On modern systems (including CS50's environment), these sizes are typical:

- `bool`: 1 byte (even though 1 bit would theoretically suffice)

- `char`: 1 byte
- `int`: 4 bytes (32 bits)
- `float`: 4 bytes
- `double`: 8 bytes
- `long`: often 8 bytes (platform-dependent, but commonly 64 bits)

A **string** is different: it is not a single fixed-size type. A string can be 0 characters, 5 characters, or 500 characters, so it requires a variable number of bytes.

3.5.2 Memory as addressed bytes

Random-access memory (RAM) can be modeled as a large grid of **bytes**, and each byte has an **address** (a location). You can imagine addresses as numbering the bytes 0, 1, 2, 3, and onward.

When you store values, each value occupies some number of consecutive bytes:

- A `char` uses 1 byte.
- An `int` uses 4 bytes.
- A `long` might use 8 bytes.

This "layout" perspective is the foundation for understanding arrays, strings, and later topics like pointers.

3.6 Arrays: One Name for Many Related Values

3.6.1 The problem: many variables that differ only by a number

Suppose you want three exam scores:

```
int score1 = 72;
int score2 = 73;
int score3 = 33;
```

This works, but it is a design smell: as soon as the number of scores changes, your program becomes an increasingly error-prone pile of near-duplicates (`score4`, `score5`, …). You want a structure that scales.

3.6.2 The idea: an array stores values back-to-back

An **array** is a sequence of values stored contiguously in memory (back-to-back with no intentional gaps), accessed under a single name.

In C, you can declare an array of three integers like this:

```
int scores[3];
```

This means: "allocate enough space for 3 integers."

Then you assign into specific positions using **square bracket indexing**:

```
scores[0] = 72;
scores[1] = 73;
scores[2] = 33;
```

Notice the indexing:

- The first element is at index `0`, not `1`.
- If the array has size 3, the valid indices are `0`, `1`, and `2`.

This "starts at 0" convention is fundamental in C. You can choose to waste a slot and start at 1, but you must then remember that you intentionally left index 0 unused, and you lose one unit of capacity.

3.6.3 Out of bounds: the danger of going past the end

If you write:

```
scores[3] = 100;
```

you are writing to the **fourth** slot, but your array has only three. This is "no man's land": you are accessing memory that does not belong to the array.

In C, the language generally will not stop you at runtime. The result can be:

- Weird behavior
- Corrupted data
- A crash

This lack of automatic safety is one of the reasons C is considered powerful but dangerous.

3.6.4 Making the program interactive: filling an array with a loop

Hardcoding exam scores into source code is not very useful. Instead, you can prompt the user and store the results in the array.

A first attempt might still repeat code:

```
scores[0] = get_int("Score: ");
scores[1] = get_int("Score: ");
scores[2] = get_int("Score: ");
```

But this is exactly the kind of repetition loops eliminate. With a `for` loop:

```
for (int i = 0; i < 3; i++)
{
    scores[i] = get_int("Score: ");
}
```

Now the loop variable `i` acts as the index, and the same line stores into different slots on each iteration.

3.6.5 Avoiding magic numbers: constants and keeping values in sync

A subtle design problem appears when the same number shows up in multiple places:

- The array size
- The loop bound
- The divisor when computing an average

If those values ever get out of sync, your program becomes wrong in a way that can be hard to notice.

A common solution is to define a constant:

```
const int N = 3;
```

By convention, constants are often written in uppercase to visually distinguish them. This is not required by C, but it is a common style.

You can even place such a constant outside of `main`, making it a **global variable** (in scope for the entire file). Global variables can be risky when they are mutable, but as constants they are a reasonable way to share a fixed configuration value.

3.7 Arrays and Functions: Why You Must Pass the Length

In many languages, an array "knows its own length." In C, it generally does not. That means if you write a function that receives an array, you typically must also pass the number of elements.

Consider computing an average. You might design a function like:

- Input: an array of scores and the number of scores
- Output: the average as a floating-point value

Conceptually:

```
float average(int array[], int length);
```

Inside that function, you can accumulate a sum:

- Start at 0
- Add each array element in a loop
- Divide by the length

When dividing, you must avoid integer truncation, so you cast (or otherwise ensure floating-point division), for example:

```
return sum / (float) length;
```

The key point is not the exact code, but the design rule:

- In C, if a function needs to loop over an array, it almost always also needs the array's length as a separate argument.

3.8 From Numbers to Letters: `char`, ASCII, and Interpreting Bytes

Once you understand arrays, you can use the same ideas to work with text. In fact, you have already been working with arrays in week 1—just not explicitly.

3.8.1 `char` values are numbers

A `char` is a one-byte value, and one byte can store a number from 0 to 255. The reason characters "work" is that the world standardized a mapping from numbers to letters.

In ASCII:

- `'A'` is 65
- `'B'` is 66
- …
- `'a'` is 97

This lets you treat characters as characters or as integers, depending on how you print them.

Example:

```
char c1 = 'H';
char c2 = 'I';
char c3 = '!';
```

If you print with `%c`, you see characters:

```
printf("%c%c%c\n", c1, c2, c3);
```

If you print with `%i`, you see the underlying numbers:

```
printf("%i %i %i\n", c1, c2, c3);
```

For `"HI!"`, those numbers are:

- `H` → 72
- `I` → 73
- `!` → 33

The byte in memory did not change; only your interpretation changed.

3.9 Strings in C: An Array of `char` Ending with `\0`

A "string" feels like a single object, but in C it is built from simpler parts.

3.9.1 A string is an array of characters

If you write:

```
string s = "HI!";
```

you can index into it as if it were an array:

- `s[0]` is `'H'`
- `s[1]` is `'I'`
- `s[2]` is `'!'`

This is not just a metaphor. In memory, the characters are stored contiguously, just like an array.

3.9.2 How does the computer know where the string ends?

If memory is just a big grid of bytes, how does the computer know that `"HI!"` stops after three characters rather than continuing into unrelated bytes?

C strings use a special sentinel value at the end: the **NUL terminator**, written:

- `'\0'` (a backslash-zero escape sequence)

This is a single byte whose bits are all 0. It is not the visible character `'0'`. It is a zero byte used as a marker.

So `"HI!"` is stored as **four** bytes:

- `'H'`, `'I'`, `'!'`, `'\0'`

This is why string storage is often described as $\mathbf{n + 1}$ bytes: you need one extra byte for the terminator.

You can even observe this by printing the integer values of characters:

- `s[0]`, `s[1]`, `s[2]` show 72, 73, 33
- `s[3]` shows 0

3.9.3 NUL vs. NULL

The NUL terminator is often written as:

- **NUL** (referring to the zero byte `'\0'`)

Later in the course you will also see:

- **NULL** (spelled N-U-L-L), which refers to a different concept involving pointers

The similar spelling is historically unfortunate, but the ideas are distinct.

3.9.4 Two strings, two terminators

If you have:

```
string s = "HI!";
string t = "BYE!";
```

each string has its own NUL terminator:

- `"HI!\0"`
- `"BYE!\0"`

Those terminators are what prevent `printf("%s", s)` from accidentally continuing into the bytes of `t`.

3.10 Arrays of Strings: Organizing Multiple Words

Just as an array can store multiple integers, it can store multiple strings:

```
string words[2];
words[0] = "HI!";
words[1] = "BYE!";
```

Now:

- `words[0]` is a string (an array of characters ending in `'\0'`)
- `words[1]` is another string

This creates a layered structure:

- `words` is an array of strings
- each string is an array of characters

That is why you can write "two indices":

- `words[0][0]` is the first character of the first word
- `words[1][2]` is the third character of the second word

This resembles a two-dimensional array, and it is often helpful to think of it that way when manipulating text.

3.10.1 Out-of-bounds reads can "leak" adjacent memory

If you accidentally index past the end of one string, you might see bytes that belong to something else stored nearby. For example, if `"HI!"` is followed in memory by `"BYE!"`, reading beyond the `'\0'` of the first string could yield `'B'`—not because the strings are "connected," but because you are reading memory you do not own.

This is a powerful demonstration of why bounds errors in C are so dangerous: you might not crash immediately; you might just silently read or write the wrong bytes.

3.11 Measuring String Length: Writing It Yourself and Using `strlen`

3.11.1 Computing length manually

Because C strings end with `'\0'`, you can compute the length by starting at index 0 and counting until you reach the terminator:

- Initialize a counter to 0
- While the current character is not `'\0'`, increment the counter

This works because the NUL terminator acts like a "stop sign" embedded in memory.

3.11.2 The standard library solution: `strlen`

You do not need to keep rewriting length logic. C provides a string library:

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

and inside it is a function:

- `strlen`

which returns the number of characters before the `'\0'` terminator.

3.12 Iterating Over Strings Efficiently: Don't Recompute `strlen` Every Time

A common pattern is to print a string one character at a time:

```
for (int i = 0; i < strlen(s); i++)
{
    printf("%c", s[i]);
}
printf("\n");
```

This is correct, but it has a subtle inefficiency: the loop condition `i < strlen(s)` is evaluated repeatedly, and each evaluation calls `strlen(s)`, which itself must scan the string to find `'\0'`.

If the string length never changes during the loop, you should compute it once and reuse it.

One common idiom is:

```
for (int i = 0, n = strlen(s); i < n; i++)
{
    printf("%c", s[i]);
```

```
}
printf("\n");
```

Here you see a compact C feature:

- `int i = 0, n = strlen(s);` declares **two integers** in one statement, separated by a comma

The result is cleaner and more efficient: the length is calculated once.

3.13 Converting Text Case: ASCII Arithmetic vs. `ctype.h`

Suppose you want to force a string to uppercase.

3.13.1 Doing it "by hand" with ASCII arithmetic

Because ASCII assigns uppercase and lowercase letters consistent numeric values, there is a predictable offset between them. For example:

- `'A'` is 65
- `'a'` is 97
- the difference is 32

So you *can* convert by checking whether a character is between `'a'` and `'z'` and then subtracting 32.

This approach works, but it is fragile and forces you to embed knowledge of numeric encodings into your program.

3.13.2 The better tool: `toupper` from `ctype.h`

C provides a character type library:

```
#include <ctype.h>
```

and within it is:

- `toupper`

You can convert each character with:

```
printf("%c", toupper(s[i]));
```

A particularly convenient property (as documented) is that:

- if the character is already uppercase or not a letter, `toupper` returns it unchanged

So you do not even need to write your own `if` condition for `'a'` to `'z'`; you can simply apply `toupper` to each character and print the result.

This is a recurring theme: C is low-level, but it also comes with libraries that prevent you from reinventing common wheels.

3.14 Command-Line Arguments: Input Without Prompts

So far, your programs have used `get_string` and `get_int` to ask the user for input interactively. Many command-line tools instead accept input as additional words typed at the prompt, which is faster and more scriptable.

You have already been using command-line arguments with tools like:

- `cd foldername`
- `rm filename`
- `clang -o hello hello.c`
- `clang ... -lcs50`

Now you can write programs that behave the same way.

3.14.1 A different `main` signature: `argc` and `argv`

Instead of:

```
int main(void)
```

you can define:

```
int main(int argc, string argv[])
```

By convention:

- `argc` is the **argument count** (how many words were typed)
- `argv` is the **argument vector** (an array of strings containing those words)

The contents of `argv` follow a consistent rule:

- `argv[0]` is the program name (like `./greet`)
- `argv[1]` is the first user-provided word
- `argv[2]` is the second user-provided word
- and so on

So if the user runs:

```
./greet David
```

then:

- `argc` is 2
- `argv[0]` is `"./greet"`
- `argv[1]` is `"David"`

3.14.2 Validating input: don't index what isn't there

If you try to use `argv[1]` without checking that the user actually provided it, your program may misbehave.

A robust approach is:

- If `argc == 2`, greet the user by name.
- Otherwise, print a default message (or an error).

This pattern mirrors the broader rule of C programming: the language gives you power, but expects you to check your assumptions.

3.14.3 Looping over all command-line arguments

Because `argv` is an array, you can use a loop:

```
for (int i = 0; i < argc; i++)
{
    printf("%s\n", argv[i]);
}
```

This prints each word the user typed, one per line, starting with the program name.

3.14.4 A real-world illustration: `cowsay` and flags

Some programs accept not only words but also options, often written with a dash. For instance, `cowsay` can print ASCII art and can be configured with flags like `-f` to choose different "characters" (such as a duck or dragon).

The specific tool is just for fun, but the design pattern is serious and ubiquitous:

- Programs often accept a mix of arguments (data) and flags (configuration).

3.15 Exit Status: How Programs Secretly Signal Success or Failure

Every program you run in a terminal exits with a numeric status code.

- By convention, **0 means success**
- Any **nonzero value** indicates some kind of failure

This idea shows up in real software as "error codes," such as a Zoom error like `1132`. The exact numbers are meaningful mainly to the developers, but the existence of a code is a standard diagnostic mechanism.

3.15.1 Returning a status from `main`

Because `main` returns an `int`, you can explicitly return:

- `return 0;` for success
- `return 1;` (or another nonzero) for failure

This is particularly useful when your program should stop early due to invalid command-line input.

3.15.2 Seeing the last exit status with `echo $?`

In a Unix-like terminal, you can inspect the exit status of the most recently run program with:

```
echo $?
```

This is a low-level but powerful way to understand how tools communicate success and failure to the environment, and it is one reason automated testing tools can work: they can run your program and check whether it exited successfully.

3.16 Cryptography as an Application: Plaintext, Ciphertext, Ciphers, and Keys

Arrays and strings become far more interesting when you use them to transform text. One of the most important real-world motivations for transforming text is **cryptography**.

3.16.1 Vocabulary

- **Plaintext**: the original human-readable message (for example, `"HI!"` or `"I LOVE YOU"`).
- **Ciphertext**: the scrambled output produced by encryption.
- **Cipher**: the algorithm that converts plaintext to ciphertext (and typically allows decryption in reverse).
- **Key**: a secret value (often a number) that configures the cipher so that only someone with the same key can decrypt.

A useful mental model is a lock-and-key system: the cipher is the lock's mechanism, and the key determines how it behaves.

3.16.2 The Caesar cipher: shifting letters

A classic historical cipher is the **Caesar cipher**, associated with Julius Caesar. It works by shifting each letter forward in the alphabet by a fixed amount.

Example with key = 1:

- `H` becomes `I`
- `I` becomes `J`
- `"HI!"` becomes `"IJ!"` (punctuation is typically left unchanged)

If you reach `Z`, you wrap around to `A`.

Decryption is the reverse shift If encryption shifts forward by 1, decryption shifts backward by 1:

- `I` becomes `H`
- `J` becomes `I`

This reversibility is essential; otherwise, encryption would destroy information rather than protect it.

3.16.3 ROT13 and the limits of simple ciphers

A well-known shift is 13, called **ROT13** ("rotate by 13"). It is easy to implement and historically common in informal contexts.

But the Caesar cipher has a small key space: for the English alphabet, there are only 26 possible shifts. That means an attacker can **brute force** the cipher by trying all keys.

This also reveals a humorous "anti-security" example:

- A shift of 26 (sometimes joked about as "ROT26") returns every letter to itself, producing no encryption at all.

Modern cryptography uses far more sophisticated mathematics and far larger keys (often hundreds or thousands of bits), but the Caesar cipher is a useful teaching tool because it forces you to think concretely about:

- characters as numbers (ASCII)
- shifting and wrapping
- iterating over strings
- producing transformed output

3.16.4 A final message

The lecture's encrypted teaser message, once decrypted by shifting letters back by one, resolves to:

> THIS WAS CS50

And with that, the practical arc of week 2 is clear: once you can store sequences of values (arrays), understand how text is stored (strings with `'\0'`), and systematically debug your logic, you can begin writing programs that analyze and transform real data—whether to estimate reading levels or to scramble messages for secure communication.

Chapter 4: Week 3 — Algorithms: Searching, Sorting, Big-O, and Recursion

By week 3, you already have enough C vocabulary to express nontrivial programs: variables, conditionals, loops, functions, arrays, and strings. This week shifts the focus away from new syntax

and toward something more fundamental: **how to think algorithmically**, meaning how to take a real-world problem, describe a step-by-step process to solve it, and then reason about how well that process will scale as the problem grows.

The core theme is that the *same* problem can be solved in multiple ways, and those ways can differ dramatically in performance. The difference is not usually about "micro-optimizations," but about choosing a fundamentally better strategy.

To make that concrete, we will focus on three kinds of tasks that show up everywhere in computing:

- **Searching**: Is the piece of data you want present, and where?
- **Sorting**: Can you rearrange data into an order that makes other tasks faster?
- **Recursion and divide-and-conquer**: Can you solve a big problem by solving smaller versions of the same problem and combining the results?

Along the way, we will introduce the standard language computer scientists use to describe performance: **Big O**, **Omega**, and **Theta** notation.

4.1 Problem Size vs. Time: Why Strategy Matters

In week 0, a phone book illustrated a key insight: when you search a sorted phone book, you do not need to scan one page at a time. You can go to the middle, decide which half you need, then repeat. That "halving, halving, halving" behavior produces a curve that grows very slowly as the phone book grows.

A similar idea appeared in a live "attendance-counting" thought experiment. Instead of counting people one-by-one (or even two-at-a-time), the group paired off, added their counts, sat down, and repeated. In theory, each round halves the number of participants still standing, so the number of rounds grows like a logarithm: even if the room doubled in size, it would add only about one extra round. In practice, the demo produced a wrong number because humans introduced a bug, but the *algorithmic* idea was the point: **divide and conquer** can be fundamentally faster than counting linearly.

That same principle is behind features you use daily, like searching contacts: good implementations do not usually scan top-to-bottom; they typically narrow the search range repeatedly.

To reason about these tradeoffs, we need to describe algorithms precisely, and then analyze them.

4.2 Arrays as the Playground: Contiguous Data Enables Systematic Searching

Week 2 defined an **array** as a sequence of values stored **contiguously** (back-to-back) in memory. That property matters for algorithms because it makes it possible to refer to "the first element," "the middle element," "the last element," and so on in a disciplined way.

We will imagine an array as a row of "lockers" indexed from left to right:

- If the array has `n` elements, the indices are `0` through `n - 1`.
- Access uses square brackets, like `array[i]`.

Even if a human can "see the whole row at once," a computer's model is more constrained: it can read memory locations, but it must do so *procedurally*, step by step. That is exactly why the algorithm matters: the computer will do whatever sequence of inspections you instruct it to do.

4.3 Searching: Linear Search vs. Binary Search

4.3.1 The search problem

A common search problem can be described like this:

- **Input**: an array of values (for example, integers), and a target value (for example, `50`)
- **Output**: a boolean answer—**true** if the target appears in the array, **false** otherwise

There are multiple ways to implement the "black box" that performs the search.

4.3.2 Linear search

Linear search means scanning the data one element at a time, typically from left to right (or right to left). The order does not matter; the defining idea is that you check items one-by-one until you either find the target or exhaust the list.

A clear pseudocode version is:

```
For i from 0 to n - 1:
    If array[i] is the target:
        Return true
Return false
```

A subtle but important detail is the placement of `Return false`. It must happen **after** the loop finishes. If you mistakenly return false the first time you see a mismatch, you would reject the array too early.

Linear search works on any array, sorted or not, but it can be slow when the target is near the end—or not present at all.

4.3.3 Binary search

Binary search is a divide-and-conquer search strategy that applies when the array is **sorted**. The idea is:

1. Look at the middle element.
2. If it is the target, you are done.
3. If the target is smaller, search only the left half.
4. If the target is larger, search only the right half.
5. If there is no data left to search, the target is not present.

A high-level pseudocode version is:

```
If no elements remain:
    Return false
If middle element is the target:
    Return true
Else if target < middle element:
    Search left half
Else if target > middle element:
    Search right half
```

The "no elements remain" rule is crucial. Without it, the algorithm could keep trying to subdivide forever (or, in real code, run into invalid indices). This stopping condition is the algorithm's **base case** (a concept we will return to when discussing recursion).

Binary search is powerful because each step discards half the remaining candidates, so the number of checks grows very slowly as `n` grows. The tradeoff is that binary search requires sorted input; if the data is unsorted, you cannot safely throw away half the array based on one comparison.

4.4 Measuring Efficiency: Big O, Omega, and Theta

When we discuss performance, we usually do **not** mean "how many seconds did it take on my laptop." Hardware varies, and software environments vary. Instead, we measure how the number of steps grows as the input size grows.

Let:

- `n` = the size of the input (number of pages, number of people, number of elements in an array)

4.4.1 Big O: an upper bound (often worst case)

Big O notation, written `O( … )`, describes an **upper bound** on running time. Informally, it answers:

> "In the worst case, how many steps might this take, as `n` grows?"

Examples of common Big O categories include:

- `O(1)` — constant time (a fixed number of steps, regardless of `n`)
- `O(log n)` — logarithmic time (halving repeatedly)

- `O(n)` — linear time (one pass over the data)
- `O(n log n)` — linearithmic time (common in efficient sorting)
- `O(n^2)` — quadratic time (often "nested loop" behavior)

A key practice in Big O analysis is that we typically **ignore constant factors** and lower-order terms, because for large `n` they do not change the overall growth trend. For instance, `O(n)` and `O(n/2)` are both categorized as `O(n)`.

This "throw away constants" habit is not saying constants never matter in reality; rather, it is saying that asymptotically (as `n` becomes very large), the dominant term controls the behavior.

4.4.2 Omega: a lower bound (often best case)

Omega notation, written `Ω( … )`, describes a **lower bound** on running time. Informally, it answers:

> "In the best case, how few steps might this take?"

For searching:

- Linear search has a best case of `Ω(1)` if the target happens to be the first element.
- Binary search also has a best case of `Ω(1)` if the target happens to be the middle element on the first check.

4.4.3 Theta: a tight bound when upper and lower match

Theta notation, written `Θ( … )`, describes a situation where the upper and lower bounds are the same order, meaning the algorithm's growth is tightly characterized.

For example, if an algorithm always takes on the order of `n` steps whether the input is "nice" or "unlucky," then it might be `Θ(n)`.

4.5 Implementing Linear Search in C (Integers)

Algorithms become more concrete when translated to code. Consider implementing linear search over an integer array.

A useful C feature is that if you already know the values you want in an array, you can initialize the array with **curly braces**:

```
int numbers[] = {20, 500, 10, 5, 100, 1, 50};
```

Here, the compiler can infer the array's length from the initializer list, which can prevent mismatches between "how big you said the array is" and "how many values you actually provided."

A simple linear-search program looks like this in spirit:

- Store a small list of integers in an array.

- Prompt the user for a target integer.
- Loop through the array, and if you find it, print `"found"` and exit successfully.
- If the loop ends without finding it, print `"not found"` and exit with a failure status.

A key design detail is **when to print** and **when to exit**. If you print `"not found"` inside the loop every time an element does not match, you will print `"not found"` many times even though the target might appear later. Instead:

- Print `"found"` immediately when found, and return from `main`.
- Only print `"not found"` after the loop finishes.

4.5.1 Exit status and returning from `main`

Recall that `main` returns an `int`. By convention:

- `return 0;` signals success.
- `return 1;` (or any nonzero value) signals failure.

This is why many programs return `0` when they work as expected and nonzero values when something goes wrong. Returning from `main` immediately terminates the program, just like returning from any other function.

4.6 Searching Strings: Why == Is Not Enough, and `strcmp` Matters

If you switch from integers to strings, it is tempting to write:

```
if (strings[i] == s)
{
    ...
}
```

But in C, this does not compare strings the way you expect. For now, the important practical rule is:

> To compare two strings in C, use `strcmp` from `<string.h>`.

4.6.1 `strcmp` and what it returns

The function:

```
int strcmp(const char *s1, const char *s2);
```

returns an integer that indicates how the two strings compare:

- `0` if the strings are equal
- a negative number if `s1` comes "before" `s2`
- a positive number if `s1` comes "after" `s2`

This ordering is often described playfully as **ASCII-betical order**, because the comparison is based on the underlying numeric character codes (ASCII or compatible encodings).

Even if you only care about equality, this richer return value is useful in general because it can support sorting and ordering, not just yes/no equality.

4.6.2 A string-search example

If you have a Monopoly-inspired array:

```
string strings[] = {"Battleship", "Boot", "Cannon", "Iron", "Thimble", "Top hat"};
```

then searching should look conceptually like:

```
if (strcmp(strings[i], s) == 0)
{
    // found
}
```

If you forget to `#include <string.h>`, the compiler will complain because it has not been told that `strcmp` exists or how to call it.

4.7 A Simple Phone Book: From Parallel Arrays to a Better Data Structure

4.7.1 The "code smell" of parallel arrays

A first attempt at a phone book might use two arrays:

- One array of names
- One array of numbers

with the implicit assumption that index `i` in the names array corresponds to index `i` in the numbers array.

This works for tiny examples, but it tends to "smell" wrong: as the program grows, it is easy for the arrays to drift out of sync. Adding, removing, or rearranging entries becomes fragile.

This is a classic motivation for using a richer data structure: we want to keep related fields together.

4.7.2 Phone numbers as strings

Even though we call it a "phone number," it is often best stored as a **string**, not an integer, because:

- Phone numbers can contain `+`, `-`, spaces, or parentheses.
- Phone numbers can be too long for typical integer types.
- You do not do arithmetic on phone numbers.

A good rule of thumb is:

If you will not do math on it, and it may contain formatting, it is often best stored as a string.

4.7.3 Creating your own type with `struct` and `typedef`

C lets you define your own compound data type using a **structure**, or `struct`. A `struct` is a value that contains multiple fields, each with its own type.

A common pattern is:

```
typedef struct
{
    string name;
    string number;
}
person;
```

This does two things:

- `struct { ... }` defines the structure layout: it has a `name` field and a `number` field.
- `typedef ... person;` creates a new type name, `person`, so you can declare variables of that type.

4.7.4 Using the dot operator to access fields

Once you have an array of `person`, you can keep each name with its number:

```
person people[3];
```

To set or read fields inside a `struct`, you use the **dot operator** (`.`):

- `people[0].name` means "the name field of the first person"
- `people[0].number` means "the number field of the first person"

Conceptually, this lets you rewrite a phone book so that each entry is one coherent unit rather than a fragile pairing across two separate arrays.

4.7.5 Initialization and garbage values

If you create a `person` but do not initialize all its fields, the uninitialized fields will contain **garbage values**, just like uninitialized variables in general. You *can* do partial initialization in C, but you should treat it as dangerous unless you are very careful never to read the uninitialized field.

4.8 Sorting: Turning Unsorted Data Into Sorted Data

Searching gets dramatically faster when data is sorted (binary search is the classic example), but sorting itself has a cost. A natural question is:

How expensive is it to sort, and can we sort efficiently enough that it's worth it?

Sorting can be framed as:

- **Input**: an array of values in arbitrary order
- **Output**: the same values in increasing (or alphabetical) order

We will look at three sorting algorithms:

1. **Selection sort**
2. **Bubble sort**
3. **Merge sort**

The first two are simpler to understand but can be slow for large `n`. The third is a standard example of a faster, divide-and-conquer algorithm.

4.9 Selection Sort: Repeatedly Select the Smallest Remaining Element

4.9.1 The idea

Selection sort works by repeatedly selecting the smallest element from the unsorted portion of the array and swapping it into its correct position.

In human terms, if people are holding numbers:

1. Find the smallest number across the whole line.
2. Swap it into position 0.
3. Find the smallest number among positions 1 through the end.
4. Swap it into position 1.
5. Repeat.

A typical pseudocode formulation is:

```
For i from 0 to n - 1:
    Find the smallest element between array[i] and array[n - 1]
    Swap it with array[i]
```

4.9.2 Running time: why it becomes quadratic

Selection sort makes many comparisons. On the first pass, it may compare roughly `n - 1` elements. On the second pass, roughly `n - 2`, then `n - 3`, and so on.

This sum:

```
(n - 1) + (n - 2) + ... + 1
```

is proportional to `n^2`, more precisely `n(n - 1)/2`, which is dominated by the `n^2` term.

So:

- **Big O**: `O(n^2)`
- **Omega**: `Ω(n^2)` (even if the array is already sorted, selection sort still scans to "confirm" the smallest each time)
- Therefore **Theta**: `Θ(n^2)`

Selection sort does not naturally "get lucky" and stop early, because the algorithm as defined does not check whether work is unnecessary; it simply performs its full routine.

4.10 Bubble Sort: Fix Local Out-of-Order Pairs Repeatedly

4.10.1 The idea

Bubble sort repeatedly compares adjacent pairs and swaps them if they are out of order. Over time, large values "bubble" to the right, one swap at a time.

A common pseudocode version is:

```
Repeat the following (n - 1) times:
    For i from 0 to n - 2:
        If array[i] and array[i + 1] are out of order:
            Swap them
```

The loop ends at `n - 2` (not `n - 1`) because the algorithm looks at `array[i + 1]`, and `i + 1` must remain within bounds.

4.10.2 Worst-case running time

This "nested loop" structure leads to roughly `(n - 1)(n - 1)` comparisons, which is still dominated by `n^2`.

So in the worst case:

- **Big O**: `O(n^2)`

4.10.3 A practical improvement: quit early if no swaps occur

Bubble sort has an important optimization: if you make a full pass through the array and perform **no swaps**, then the array must already be sorted, and there is no need to continue.

In pseudocode, you might add the idea:

```
If no swaps occurred during a pass:
    Quit
```

With this improvement, bubble sort can be much faster on already-sorted (or nearly-sorted) data.

- **Best case (already sorted)**: it still must check adjacency across the list at least once, so it takes on the order of `n` steps.

– **Omega**: `Ω(n)`

Because its worst and best cases differ, bubble sort is not naturally described by a single Theta bound in the same way selection sort is.

4.11 Recursion: Solving a Problem by Solving Smaller Versions of Itself

4.11.1 Definition

A function (or algorithm) is **recursive** if it calls itself. Recursion is not magical; it is a disciplined way to express "divide and conquer" by reusing the same logic on smaller inputs.

A recursive algorithm must have:

- A **recursive case**: the step where it calls itself on a smaller problem.
- A **base case**: a stopping condition that prevents infinite recursion.

Binary search is naturally recursive because "search the left half" and "search the right half" are the same operation applied to a smaller range.

4.11.2 Recursion vs. iteration

Earlier in the course, you expressed repetition with loops, such as:

- `for` loops
- `while` loops

Recursion is another way to represent repetition, often when the problem has a "self-similar" structure: smaller subproblems resemble the original problem.

4.12 A Recursive Structure Example: Printing a Pyramid

A pyramid of height 4 can be described recursively:

- A pyramid of height 4 is a pyramid of height 3, plus one more row.
- A pyramid of height 3 is a pyramid of height 2, plus one more row.
- …
- A pyramid of height 1 is a single row (or "nothing plus one row," depending on how you define the base case).

4.12.1 Iterative approach (loop-based)

An iterative solution might use nested loops:

- Outer loop for each row
- Inner loop to print the correct number of `#` characters for that row

The crucial detail is choosing loop bounds so that the first row prints 1 `#`, the second prints 2 `#`, and so on.

4.12.2 Recursive approach (function calls itself)

A recursive `draw(n)` can be defined like this:

1. If `n <= 0`, return (base case).
2. Draw a pyramid of height `n - 1` (recursive case).
3. Print one row containing `n #` characters.

This mirrors the definition "height `n` is height `n - 1` plus one row."

4.12.3 Why the base case matters

If you write `draw(n - 1)` without a base case, `n` will eventually become `0`, then `-1`, then `-2`, and so on. Since integers are signed by default, the recursion would not stop on its own. A compiler may even warn that "all paths through this function will call itself" if it can see that no base case exists.

The base case is what makes recursion safe and finite.

4.13 Merge Sort: A Faster Sorting Algorithm via Divide and Conquer

Selection sort and bubble sort can both be `O(n^2)`, which becomes prohibitively slow as `n` grows. To do better, we return to divide and conquer.

4.13.1 The idea (high level)

Merge sort's pseudocode is concise:

```
If the array has only one element:
    Quit (it is already sorted)
Sort the left half
Sort the right half
Merge the sorted halves
```

This is recursive: "sort the left half" and "sort the right half" are the same sorting problem on smaller arrays.

4.13.2 What "merge" means

The merge step assumes you have two sorted halves, such as:

- Left: `1 3 4 6`
- Right: `0 2 5 7`

To merge them into one sorted list:

- Point at the first element of each half.
- Repeatedly take the smaller of the two pointed-at elements and append it to an output array.
- Advance the pointer in the half you took from.
- Continue until all elements are consumed.

This "stitching together" is efficient because each element is copied exactly once during the merge.

4.13.3 Time–space tradeoff

Merge sort typically uses **extra space** (temporary arrays) while merging. This illustrates a common tradeoff in computing:

- If you are willing to use more **space**, you can often reduce **time**.
- If you restrict yourself to constant extra space, you may be forced into slower methods like selection sort or bubble sort.

Merge sort is fast in large part because it embraces additional memory to make merging simple and systematic.

4.13.4 Running time: `n log n`

Merge sort divides the problem in half repeatedly:

- The number of times you can halve `n` until reaching 1 is on the order of `log n`.

At each "level" of halving, the merge work across all subarrays sums to about `n` operations (you are moving or comparing each element during merging).

So total work is:

- about `log n` levels
- times about `n` work per level

giving:

- **Big O**: `O(n log n)`
- **Omega**: `Ω(n log n)`
- Therefore **Theta**: `Θ(n log n)`

This places merge sort in a fundamentally better category than `n^2` sorts for large inputs.

4.14 Putting It All Together: Choosing Algorithms Intelligently

At this point, you have several key algorithmic tools and a language for comparing them:

- **Linear search**: simple, works on unsorted data, worst case `O(n)`

- **Binary search**: fast, but requires sorted data, worst case `O(log n)`
- **Selection sort**: conceptually straightforward, but `Θ(n^2)`
- **Bubble sort**: also `O(n^2)` worst case, but can be `Ω(n)` best case with an early-exit optimization
- **Merge sort**: uses recursion and extra space to achieve `Θ(n log n)`

You also have an important design lesson from building a phone book:

- When data fields belong together, represent them together.
- In C, you can do this by creating your own types with `struct` and `typedef`, and accessing fields with the dot operator.

These ideas—data organization, algorithm choice, and performance reasoning—are not "extra" topics separate from programming. They are the core of what it means to write software that continues to work well as the size of the world (and your input) grows.

Chapter 5: Week 4 — Memory: Hexadecimal, Pointers, Strings, Dynamic Allocation, and Files

Week 4 is where C stops feeling like "a slightly awkward Python" and starts revealing what it has been all along: a language that lets you talk directly about **memory**, meaning the bytes inside your computer where programs, variables, and files ultimately live. The goal this week is not just to learn new syntax, but to develop a concrete mental model for what your code *does* underneath the hood, so that when something breaks—especially in confusing ways like a segmentation fault—you can reason your way to the cause.

This chapter builds that model in layers:

- We begin with how images can be represented as bits, and why **hexadecimal** is a convenient way to talk about those bits.
- We then introduce **addresses** and **pointers**, the tools C gives you to refer to locations in memory.
- We revisit **strings** and reveal what they really are in C: not a special "string" type, but pointers to characters.
- We learn how to correctly **compare** and **copy** strings by working with memory safely.
- We introduce **dynamic memory allocation** with `malloc` and `free`, and a debugging tool, **valgrind**, that helps detect memory bugs.
- We explore the **stack** and the **heap**, and why passing values to functions sometimes fails to change the original variables.
- Finally, we use pointers to move beyond the keyboard and screen and start working with **files**, including copying binary files byte-by-byte.

5.1 Pixels, Bits, and the Limits of "Enhance"

A digital image looks smooth and detailed on your screen, but it is ultimately just a **grid** of tiny dots called **pixels**. Each pixel stores some information about its color, and the image is simply the collection of all those pixels arranged by row and column.

A useful way to see this is to imagine a very small "pixel art" image made from a limited grid. If each pixel can only be black or white, then one simple representation is:

- `0` means black
- `1` means white

A grid of 0s and 1s can form a recognizable picture (like a smiley face) even though the data is just bits. This observation matters because it highlights a practical limit: if an image contains only so many pixels, then zooming in eventually reveals blocks, not hidden detail. Hollywood-style "enhance, enhance, enhance" is constrained by the fact that the file has a finite amount of information.

Later in the course, you will see that machine learning systems can sometimes *predict* missing detail (by statistical inference), but that is not the same thing as "recovering" information that was never stored in the file to begin with.

5.2 Color as Numbers: RGB and the Rise of Hexadecimal

Black-and-white pixels are a good starting point, but real images need color. A common convention is **RGB**, short for:

- **R**ed
- **G**reen
- **B**lue

A color is represented as a mixture of these three components. Each component is stored as a number—commonly from 0 to 255—where:

- `0` means "none of that color"
- `255` means "as much of that color as possible"

So:

- black is `(0, 0, 0)`
- white is `(255, 255, 255)`
- pure red is `(255, 0, 0)`
- pure green is `(0, 255, 0)`
- pure blue is `(0, 0, 255)`

5.2.1 The notation you see in tools: `#RRGGBB`

In many graphics tools (and web development), RGB colors are written in a compact form that looks like:

- `#000000` for black
- `#FFFFFF` for white
- `#FF0000` for red
- `#00FF00` for green
- `#0000FF` for blue

This is not decimal notation. It is **hexadecimal**.

5.3 Hexadecimal (Base 16)

Hexadecimal is a number system with 16 digits instead of 10. It uses:

- `0 1 2 3 4 5 6 7 8 9`
- `A B C D E F` to represent values 10 through 15

So counting in hexadecimal goes:

- `0, 1, 2, ... 9, A, B, C, D, E, F, 10, 11, 12, ...`

Hexadecimal is also called **base 16**, by analogy with **decimal** being base 10.

5.3.1 Place values in hexadecimal

Just as decimal has place values:

- ones, tens, hundreds, …

hexadecimal has:

- ones, sixteens, two-hundred-fifty-sixes, …

So in a two-digit hexadecimal number:

- the right digit is the **16 (ones)** place
- the left digit is the **16^1 (sixteens)** place

For example:

- `0x00` is 0
- `0x01` is 1
- `0x0A` is 10
- `0x10` is 16
- `0xFF` is 255 (because `15 * 16 + 15 = 255`)

5.3.2 Why programmers like hexadecimal

Hexadecimal is popular not because it is "more correct" than decimal, but because it is **convenient** when working close to the hardware.

A key fact is:

- 1 hexadecimal digit represents **4 bits** (because 16 possibilities = 2)
- 2 hexadecimal digits represent **8 bits**, which is exactly **1 byte**

So when you see something like `FF`, you can think "that is one byte worth of bits," and when you see `#RRGGBB`, you can interpret it as:

- `RR` = 1 byte for red
- `GG` = 1 byte for green
- `BB` = 1 byte for blue

This is why a tool can represent 255 as `FF`—it aligns naturally with bytes.

5.3.3 The `0x` prefix

A practical problem with hexadecimal is ambiguity: `10` could mean ten (decimal) or sixteen (hex). A common convention is to prefix hexadecimal with:

- `0x`

So:

- `0x10` means "hexadecimal 10," which is decimal 16.

This prefix is not mathematical; it is just a label to the reader (and, in some contexts, to the compiler).

5.4 Memory as a Grid of Bytes and the Idea of an Address

To reason about pointers, it helps to picture memory as a large grid of bytes. Each byte has a **location**, called an **address**. Programs store values in memory, and the CPU needs a way to refer to "which byte" holds "which data."

Computers typically represent addresses using **hexadecimal** because addresses are naturally byte-based and hexadecimal matches bytes cleanly.

When you write:

```
int n = 50;
```

your program is asking the computer to reserve enough bytes for an `int` (commonly 4 bytes) and store the bits representing 50 in those bytes. Those bytes live somewhere in memory, at some

address like `0x...` (the exact value is not important, and it will vary each time you run the program).

5.5 Addresses and Pointers in C

C gives you direct access to these memory locations.

5.5.1 The address-of operator: `&`

The operator `&` means:

> "Give me the address of this variable."

Example:

```
int n = 50;
printf("%p\n", &n);
```

- `%p` is the `printf` format code for a pointer (an address).
- `&n` is the address where `n` is stored.

You will see an output like:

- `0x7ffc...` (some large hexadecimal number)

5.5.2 What is a pointer?

A **pointer** is a variable whose value is an address.

For example:

```
int n = 50;
int *p = &n;
```

Read this slowly:

- `int *p` declares `p` as "a pointer to an int," meaning it can store the address of an integer.
- `&n` is the address of `n`.
- so `p` now stores the location of `n`.

Pointers on modern systems are commonly **8 bytes** (64 bits). Historically, many systems used 4-byte (32-bit) pointers, which limited the total addressable memory. As machines gained more memory, pointer sizes increased to represent larger address spaces.

5.5.3 The dereference operator: `*`

The operator `*` is used in two related but distinct ways:

1. In a declaration, it means "this variable is a pointer":

```
int *p;
```

2. In an expression, it means "go to that address" (dereference it):

```
printf("%i\n", *p);
```

If `p` stores the address of `n`, then `*p` means "the integer located at the address stored in `p`," which is the same as `n`.

So:

```
int n = 50;
int *p = &n;

printf("%p\n", p);   // prints the address
printf("%i\n", *p);  // prints 50
```

Even though the `*` character appears in both places, it is not multiplication here. It is pointer syntax, and context determines whether it is declaring a pointer or dereferencing one.

5.6 Strings Revisited: The Training Wheels Come Off

For weeks, you have used:

```
string s = "HI!";
```

and thought of `s` as "a string." That is a good mental model, but in *C itself*, there is no built-in `string` type. Instead, a C string is an array of characters, and the "string variable" is typically just the address of the first character.

5.6.1 The null terminator

A string like `"HI!"` is not stored as three bytes. It is stored as four:

- `'H'`
- `'I'`
- `'!'`
- `'\0'` (the **null terminator**)

The null terminator is a byte of all zero bits, and it marks the end of the string. Functions like `printf("%s", s)` print characters until they reach `'\0'`.

5.6.2 What `string` really is: `char *`

In CS50, `string` is an abstraction provided by the CS50 library. Underneath, it is:

- `char *`

That is, "a pointer to char," interpreted as "the address of the first character in a sequence of characters."

So these are conceptually the same:

```
string s = "HI!";
```

and (in "raw C"):

```
char *s = "HI!";
```

The double quotes do something important for you: the compiler stores the characters in memory and initializes `s` to the address of the first character. That is why you do **not** write `&"HI!"`—the compiler already gives you the address of the first character automatically.

5.6.3 Printing a string address versus printing a string

Because a string variable is really an address:

- `%p` prints the address itself
- `%s` treats the address as "start of a string" and prints characters until `'\0'`

Example:

```
char *s = "HI!";

printf("%p\n", s);   // address of 'H'
printf("%s\n", s);   // HI!
```

And you can confirm that `s` is the same as `&s[0]`:

```
printf("%p\n", s);
printf("%p\n", &s[0]);
```

5.6.4 Strings as contiguous memory

Because the characters are stored contiguously:

- if `s` points to `'H'` at some address, then `'I'` is at the next address, and so on.

This is exactly the same contiguous-memory idea you used with arrays in week 2—strings are simply arrays of `char` with a special terminator at the end.

5.7 Pointer Arithmetic and "Syntactic Sugar"

Once you accept that `s` is "the address of the first character," you can manipulate strings using pointer operations.

If `s` is a `char *`, then:

- `s + 1` is the address of the next character
- `s + 2` is the address of the character after that

And dereferencing follows the pointer:

```
printf("%c\n", *s);       // first character
printf("%c\n", *(s + 1)); // second character
printf("%c\n", *(s + 2)); // third character
```

This is closely related to array indexing. In fact:

- `s[0]` is equivalent to `*(s + 0)`
- `s[1]` is equivalent to `*(s + 1)`
- `s[2]` is equivalent to `*(s + 2)`

Array indexing is therefore a kind of **syntactic sugar**: a friendlier way to write pointer arithmetic.

5.8 Comparing Strings: Why == Does Not Work

In week 3, you used `strcmp` to compare strings and were warned not to use `==`. Now you can see precisely why.

Consider:

```
char *s = get_string_somehow();
char *t = get_string_somehow();

if (s == t)
{
    printf("same\n");
}
else
{
    printf("different\n");
}
```

Because `s` and `t` are pointers, `s == t` compares their **addresses**, not the characters they point to. Even if the user types the same word twice, those two strings may live in two different places in memory, so their addresses differ, and `==` will report "different."

5.8.1 The correct tool: `strcmp`

To compare the characters, you need a function that walks through both strings and compares them character-by-character:

```c
#include <string.h>

if (strcmp(s, t) == 0)
{
    printf("same\n");
}
else
{
    printf("different\n");
}
```

Recall what `strcmp` returns:

- `0` if equal
- a negative value if `s` comes "before" `t`
- a positive value if `s` comes "after" `t`

The exact positive/negative value is not meaningful; only the sign matters.

5.9 Copying Strings: Assignment Copies the Address, Not the Characters

A second subtle bug appears when you try to "copy a string" the same way you copy an integer:

```c
char *s = get_string_somehow();
char *t = s;  // looks like a copy, but isn't
```

This does not duplicate the characters. It duplicates the pointer value, meaning:

- `t` points to the same memory as `s`

So if you modify the string via `t`, you are also modifying `s`, because they share one underlying array of characters.

A classic demonstration is "capitalize the first letter of the copy." If you do:

```c
t[0] = toupper(t[0]);
```

then `s[0]` changes too, because `s` and `t` are aliases for the same string in memory.

5.10 Dynamic Memory: `malloc`, `free`, and Making a Real Copy

To actually copy a string into a new location, you need to allocate new memory for the copy, then copy the bytes.

5.10.1 `malloc`: memory allocate

`malloc` asks the operating system for a chunk of memory and returns the **address of the first byte** of that chunk.

You pass `malloc` the number of bytes you want.

A typical pattern to copy a string is:

1. Compute the number of bytes needed (string length + 1 for `'\0'`).
2. Allocate that many bytes.
3. Copy characters, including the null terminator.

Example:

```
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

char *s = get_string_somehow();

char *t = malloc(strlen(s) + 1);
if (t == NULL)
{
    // out of memory
    return 1;
}

// Copy the bytes, including '\0'
for (int i = 0; i <= strlen(s); i++)
{
    t[i] = s[i];
}

if (strlen(t) > 0)
{
    t[0] = toupper(t[0]);
}

// use s and t...

free(t);
```

Two details matter here:

- The loop uses `<= strlen(s)` so it copies the null terminator at the end. If you only copy

characters up to `< strlen(s)`, you will forget `'\0'`, and `t` will not be a valid C string.
- `malloc` can fail. When it fails, it returns `NULL`, a special value meaning "address 0," used as a sentinel for "no valid address."

5.10.2 `strcpy`: don't reinvent the wheel

Because copying strings is so common, C provides:

- `strcpy(destination, source)` in `<string.h>`

So instead of a manual loop you can do:

```
char *t = malloc(strlen(s) + 1);
if (t == NULL)
{
    return 1;
}

strcpy(t, s);
```

(You may see the name said informally as "string copy"; the actual function name in standard C is `strcpy`.)

5.10.3 `free`: give memory back

Every `malloc` should eventually be paired with `free`:

```
free(t);
```

If you allocate memory repeatedly and never free it, your program can slowly consume more and more memory, causing a **memory leak**. Over time, leaks can make programs sluggish or unstable, and in large systems they can become serious reliability issues.

5.11 Debugging Memory Bugs with Valgrind

Memory bugs can be deceptive because a program can compile and even appear to run, while still doing something invalid. To help detect common mistakes, you can use **valgrind**, a tool that runs your program and reports memory-related errors such as:

- invalid reads/writes (touching memory you don't own)
- memory leaks (allocating without freeing)

5.11.1 Invalid write example: writing past an allocated block

Suppose you allocate space for 3 integers:

```
int *x = malloc(3 * sizeof(int));
```

Valid indices are:

- `x[0]`, `x[1]`, `x[2]`

If you accidentally write `x[3]`, you are writing beyond the allocated memory. This can cause crashes—or worse, silently corrupt other data.

Valgrind reports errors like "Invalid write of size 4," where 4 is the size of an `int` on that system.

5.11.2 Memory leak example: forgetting to `free`

If you allocate:

```
int *x = malloc(3 * sizeof(int));
```

but never call:

```
free(x);
```

valgrind will report something like "definitely lost," indicating bytes that were allocated and never reclaimed.

5.12 Garbage Values: The Danger of Uninitialized Variables

A **garbage value** is whatever bits happen to already be in a region of memory that you have not initialized. In C, local variables are not automatically set to 0. If you declare:

```
int scores[1024];
```

and immediately print the contents without storing anything first, you will see unpredictable numbers. Those numbers are not random in a mystical sense; they are simply leftover bits from prior memory use.

Garbage values become far more dangerous when they appear in pointers.

5.13 Segmentation Faults and Dereferencing Bad Pointers

A **segmentation fault** occurs when your program tries to access memory it is not allowed to access. A common way to cause this is:

- dereferencing an uninitialized pointer

Consider this pattern:

```
int *x = malloc(sizeof(int));
*x = 42;
```

```c
int *y;      // uninitialized!
*y = 13;     // dereferencing garbage address -> likely crash
```

Here, `y` contains garbage bits. Those bits are treated as an address. `*y = 13` means "go to that address and write 13," which could be anywhere—often an invalid region—so the program crashes.

A popular claymation metaphor ("Binky") dramatizes this: pointers begin as arrows that point nowhere, and dereferencing them before setting up the pointee causes chaos. The core lesson is serious even if the presentation is playful:

> Allocating a pointer variable is not the same as allocating the memory it should point to.

5.14 The Stack and the Heap

To understand why some functions fail to modify variables, it helps to know how memory is typically organized while a program runs.

A simplified model divides memory into regions:

- **Machine code** (the compiled instructions)
- **Global variables** (variables defined outside all functions)
- **Heap** (dynamic memory allocated with `malloc`)
- **Stack** (function call frames: parameters and local variables)

Two regions matter most this week:

5.14.1 The heap

The **heap** is where memory comes from when you call `malloc`. It is used for data that should outlive a single function call, such as dynamically sized arrays or strings whose size is not known until runtime.

5.14.2 The stack

The **stack** stores function call frames (often called **stack frames**). Each time you call a function, a new frame is created containing:

- the function's parameters
- the function's local variables

When the function returns, its stack frame is discarded. That means:

> Local variables inside a function do not survive after the function returns.

5.14.3 Overflows: stack, heap, and buffers

Because the stack and heap both use finite memory, programs can break if they use too much:

- **stack overflow**: too many or too-large stack frames, often from deep recursion
- **heap overflow** (in one sense): using heap memory incorrectly, or allocating so much that memory is exhausted
- **buffer overflow**: writing past the end of an array (a "buffer"), corrupting adjacent memory

These issues are not just academic; they are sources of crashes and, historically, security vulnerabilities.

5.15 Why the "Swap" Function Didn't Work (Passing by Value)

Consider a function intended to swap two integers:

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

And a `main` program:

```
int x = 1;
int y = 2;

swap(x, y);

printf("%i %i\n", x, y);
```

Even though the logic inside `swap` is correct, `x` and `y` do not change. The reason is:

- C passes function arguments **by value**, meaning it copies them.
- `a` and `b` are copies of `x` and `y`.
- Swapping `a` and `b` swaps only the copies inside the `swap` stack frame.
- When `swap` returns, that frame disappears, and `x` and `y` remain unchanged.

5.16 Swapping Correctly: Passing Pointers (Pass by Reference)

To swap the original variables, you must give the function access to their memory addresses:

```
void swap(int *a, int *b)
{
```

```
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Now the call site must pass addresses:

```
swap(&x, &y);
```

This is the same underlying idea as earlier:

- `&x` gives the address of `x`
- `*a` dereferences the pointer to access and modify the integer stored there

This pattern—passing pointers so a function can modify the caller's variables—is one of the most important uses of pointers in C.

5.17 Input Without Training Wheels: `scanf` and Its Risks

So far, CS50's `get_int` and `get_string` have handled many details for you. In standard C, a commonly used function for input is:

- `scanf`

5.17.1 Scanning an integer

This is relatively straightforward:

```
int n;
printf("n: ");
scanf("%i", &n);
printf("n: %i\n", n);
```

The crucial detail is `&n`: `scanf` needs the **address** of `n` so it can store the user's input into that variable. If you passed `n` instead of `&n`, `scanf` would receive only a copy and would not be able to change the original.

5.17.2 Scanning a string: where should the characters go?

Strings are harder because `scanf("%s", ...)` needs a place to store an *unknown number of characters*. If you write:

```
char *s;
scanf("%s", s);
```

you are in trouble because `s` is an uninitialized pointer; it points nowhere valid. `scanf` will try to write to the garbage address stored in `s`, often causing a segmentation fault.

To use `scanf` safely, you need allocated space, such as an array:

```
char s[4];
scanf("%s", s);
```

Now `s` is a valid buffer of 4 bytes, and `scanf` can write into it.

But this is still dangerous: if the user types more than 3 characters (plus `'\0'`), the input will overflow the buffer and may crash or corrupt memory. The deeper point is:

> In C, safe string input is difficult because you rarely know in advance how long the user's input will be.

CS50's `get_string` avoids this by allocating and reallocating memory as the user types, growing the buffer dynamically so it fits the input.

5.18 File I/O: Opening, Writing, Reading, and Closing Files

Once you understand memory and pointers, you can start manipulating **files**: stored data such as text documents, CSVs, and images.

Common file functions in C include:

- `fopen` / `fclose` (open/close a file)
- `fprintf` / `fscanf` (print to / scan from a file, often for text)
- `fread` / `fwrite` (read/write raw bytes, often for binary formats like images)
- `fseek` (jump to a position in a file, like fast-forward/rewind)

5.18.1 Writing a simple phonebook to a CSV file

A CSV file (comma-separated values) is a plain text format where commas separate columns and newlines separate rows. Spreadsheet programs can open it easily.

A program can:

1. open a file in **append mode** (`"a"`) so it adds new entries instead of overwriting,
2. ask the user for a name and number,
3. write a line like `name,number\n`,
4. close the file.

In C, the file handle is a pointer-like type:

- `FILE *`

Example structure:

```
FILE *file = fopen("phonebook.csv", "a");
if (file == NULL)
```

```
{
    return 1;
}

char *name = get_string_somehow();
char *number = get_string_somehow();

fprintf(file, "%s,%s\n", name, number);

fclose(file);
```

A key safety habit appears here: because `fopen` returns a pointer, you should check for `NULL`. A `NULL` file pointer usually indicates the file could not be opened (missing permissions, missing directory, etc.).

5.19 Copying Binary Files Byte-by-Byte

Text output with `fprintf` is useful, but images and other binary files are not plain text. To copy a binary file, you can read raw bytes and write them to a new file.

5.19.1 A `byte` type

C does not have a built-in `byte` type in the simplest sense, but it provides fixed-width integer types in `<stdint.h>`, including:

- `uint8_t`: an unsigned 8-bit integer (exactly one byte)

You can create a friendlier synonym with `typedef`:

```
#include <stdint.h>
typedef uint8_t byte;
```

This is the same technique used by CS50's `string`, which is implemented as a synonym for `char *`.

5.19.2 A small copy program using `fread` and `fwrite`

A minimal file-copy strategy is:

- open source file for reading in binary mode: `"rb"`
- open destination file for writing in binary mode: `"wb"`
- repeatedly read one byte from source and write it to destination until there are no more bytes

Conceptually:

```c
byte b;
while (fread(&b, sizeof(b), 1, src) == 1)
{
    fwrite(&b, sizeof(b), 1, dst);
}
```

This works because:

- `fread` returns the number of "items" successfully read (here we read 1 item of size 1 byte)
- when the file ends, it returns 0, and the loop stops

This style of code makes concrete a powerful idea: a file is ultimately a sequence of bytes, and you can process it at the byte level if you have the right tools.

5.20 From Bytes to Images: Bitmap Files and Filters (A Preview)

A bitmap image format (BMP) stores an image as pixel data in a structured binary file. Because you can now read and write bytes from files, you can begin writing programs that manipulate images by:

- reading pixel values
- modifying them (for example, changing colors)
- writing the modified pixels to a new file

This is the foundation of common "filters":

- **grayscale** (convert RGB to shades of gray)
- **reflection** (mirror the image)
- **blur** (smudge pixels by averaging neighbors)
- **edge detection** (highlight boundaries by comparing neighboring pixels)

At first glance, these effects seem high-level, like features of an app. In reality, they reduce to systematic transformations of pixel data stored as bytes in an image file.

5.21 Summary: What Week 4 Adds to Your Mental Model

After this week, you should be able to think about C programs in a lower-level, more accurate way:

- **Memory** is bytes; variables occupy specific bytes at specific **addresses**.
- **Hexadecimal** is a convenient shorthand for bytes and addresses.
- `&` gets an address; `*` dereferences an address.
- A **pointer** stores an address, and pointers are central to how C works.
- A C **string** is an array of `char` ending with `'\0'`, and a "string variable" is typically a `char *` pointing to the first character.

- Comparing strings requires `strcmp` because `==` compares addresses, not characters.
- Copying strings requires allocating new memory (`malloc`) and copying bytes (`strcpy`), and allocated memory must be released (`free`).
- Tools like **valgrind** help detect memory bugs that compilers often cannot.
- The **stack** holds function frames; the **heap** holds dynamically allocated memory.
- Passing values to functions copies them; passing pointers enables functions to modify the caller's memory.
- File I/O functions (`fopen`, `fprintf`, `fread`, `fwrite`, `fclose`) let you move beyond the keyboard and screen and manipulate persistent data, including binary files like images.

These ideas make C feel more complex, but they also explain behaviors that used to feel mysterious. Once you can "see" the memory model, many bugs become less random and more like solvable puzzles with clear rules.

Chapter 6: Week 5 — Data Structures: Abstract Data Types, Linked Lists, Trees, Hash Tables, and Tries

By week 5, you have enough C in your toolkit to stop thinking of memory as something that merely "holds variables," and to start using memory deliberately as a **design medium**. This week's topic—**data structures**—is really about the question:

> Given the same raw memory (bytes and addresses), how can we organize data so that common operations become faster, simpler, or more scalable?

To make that question precise, we will repeatedly separate two levels of thinking:

- An **abstraction**: a high-level description of what a data structure *does* and what operations it supports.
- An **implementation**: the low-level details of how we represent that structure in memory using arrays, pointers, `struct`, and `malloc`.

This distinction matters because there are often many valid implementations of the same abstraction, and each comes with trade-offs, especially between **time** (speed) and **space** (memory).

6.1 Abstract Data Types (ADTs): What vs. How

An **abstract data type** (ADT) is a "data structure concept" defined primarily by:

- what it stores,
- what operations it supports,
- what guarantees it provides about those operations,

without committing to how it is implemented in memory.

For example, a "queue" is not defined by whether it uses an array or a linked list. A queue is defined by the *rule* it follows: who comes out first.

This week begins with two ADTs that are both familiar from everyday life:

- **Queues** (lines)
- **Stacks** (piles)

These are intentionally simple, because they let you focus on the relationship between an ADT and its implementation.

6.2 Queues: FIFO (First In, First Out)

A **queue** (often called a "line") has a fairness property:

- **FIFO**: **First In, First Out**

If three people line up in order 1, 2, 3, then a queue guarantees they will be served in order 1, 2, 3.

6.2.1 Queue operations: enqueue and dequeue

The standard queue operations are:

- **enqueue**: add an element to the queue (enter the line)
- **dequeue**: remove an element from the queue (leave the line)

The key point is that `dequeue` removes the *oldest* element.

6.2.2 Implementing a queue with an array

A simple implementation is to use an array with a fixed **capacity**, and track how many elements are currently stored (often called the **size**):

- the array stores elements contiguously in memory
- an integer `size` tells you how many slots are in use

This works, but it immediately raises an issue that will drive much of the rest of this chapter:

> Arrays have fixed size, and memory elsewhere in the program may prevent you from simply "extending" an array in place.

6.3 Stacks: LIFO (Last In, First Out)

A **stack** is the opposite ordering policy:

- **LIFO**: **Last In, First Out**

In a stack, the most recently added item is the first one removed.

This is not always "fair" in the same sense as FIFO, but it is often exactly what we want. A common real-world analogy is an email inbox where the newest messages appear at the top; many people naturally read the most recent messages first, which is a stack-like pattern.

6.3.1 Stack operations: push and pop

The standard stack operations are:

- **push**: add an element to the top of the stack
- **pop**: remove the element from the top of the stack

6.3.2 Implementing a stack with an array

Just like a queue, a stack can be implemented with:

- an array of capacity `N`, and
- an integer that tracks the current number of elements.

The difference between a stack and a queue is not "the memory layout," but the **logic** used to decide which element gets removed next.

This is a key ADT lesson:

> The same underlying storage (like an array) can implement different ADTs depending on what operations you perform and how.

6.4 Arrays Revisited: Contiguity and the Cost of Growing

Recall the defining property of an array:

- An array stores elements **contiguously** (back-to-back) in memory.

That contiguity is powerful. It enables direct indexing (`a[i]`) and makes binary search possible when data is sorted. But it becomes a burden the moment you want an array to grow beyond its allocated capacity.

6.4.1 The "add a 4th element" problem

Suppose you have an array of three integers containing:

- `1, 2, 3`

You now want to add `4`.

In an idealized diagram, you might imagine placing `4` immediately after `3`, but real memory is busy. That "next" region might already be in use by something else—perhaps another variable, perhaps a string, perhaps something you never explicitly created but that the program needs.

Even if there are many unused (garbage) bytes elsewhere in memory, you cannot just place `4` anywhere:

- the data must remain contiguous to still be "an array."

6.4.2 The naive growth strategy: allocate a bigger array and copy

A workable strategy is:

1. Allocate a new array that is larger.
2. Copy the old elements into the new array.
3. Add the new element.
4. Free the old array.

This works, but it has costs:

- **Time cost:** copying requires iterating over all old elements → about **Big O(n)** work.
- **Space cost (temporarily):** during copying, you momentarily hold both arrays in memory.

And the problem repeats: if you later add a 5th element, then a 6th, you may keep paying this copying cost again and again.

6.5 Growing an Array Manually with `malloc`: A Worked Example

To see exactly what "copying into a bigger array" looks like in C, consider an example program that begins with space for 3 integers, then "resizes" to space for 4.

6.5.1 Starting with a fixed array (easy, but not resizable)

A fixed array version might look like this:

```c
#include <stdio.h>

int main(void)
{
    int list[3];
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    for (int i = 0; i < 3; i++)
    {
        printf("%i\n", list[i]);
    }
}
```

This prints `1 2 3`, but the array's size is permanently 3 for the life of the variable. You cannot `free` it, and you cannot "make it larger" in place.

6.5.2 Switching to dynamic allocation (resizable in principle)

To make resizing possible, you must allocate the array on the heap:

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

int main(void)
{
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // ... later, we decide we want room for 4 ints ...
}
```

A subtle but important convenience appears here:

- Even though `list` is a pointer, you can still write `list[i]`.
- Array indexing is "syntactic sugar" for pointer arithmetic.

6.5.3 Allocating a bigger block and copying

Now we allocate a second block, copy, and extend:

```
int *tmp = malloc(4 * sizeof(int));
if (tmp == NULL)
{
    free(list);      // avoid leaking the original block
    return 1;
}

for (int i = 0; i < 3; i++)
{
    tmp[i] = list[i];
}
```

```
tmp[3] = 4;

free(list);
list = tmp;
```

Two memory lessons are embedded in this pattern:

1. **Order matters for correctness.**
   You must copy the old elements *before* freeing the old memory.
2. **Error handling must clean up.**
   If allocating `tmp` fails, you must free the original `list` (otherwise you leak memory).

Finally, you would typically free at the end of the program as well:

```
free(list);
```

This works, but the amount of code is disproportionate to the simple goal ("I want to add one more number"). That pain motivates the next idea:

> What if we stop requiring contiguity, and instead connect values with pointers?

6.6 Linked Lists: Escaping Contiguity with Pointers

A **linked list** is a data structure that stores elements in separate "chunks" of memory that can live anywhere, as long as each chunk contains:

- the data you care about, and
- a pointer that tells you where to find the next chunk.

Each chunk is called a **node**.

6.6.1 Nodes and "metadata"

In a linked list node, the "real" information might be the integer `1`, `2`, `3`, etc.

But the pointer to the next node is still data stored in memory. It is useful not because it is part of the problem domain ("what numbers are in the list"), but because it helps organize the structure.

This is a natural place to introduce the term:

- **data**: the values you conceptually care about (like `1, 2, 3`)
- **metadata**: "data about data," such as pointers that help connect nodes together

The pointer is metadata: it is an implementation detail that enables the structure to work.

6.6.2 Terminating the list with `NULL`

The last node in a linked list must not have an uninitialized "next" pointer. If it did, your program might treat that garbage value as an address and try to follow it, causing crashes or corruption.

So the final node's pointer is set to:

- `NULL`

This cleanly marks "end of list."

6.7 Implementing a Linked List Node in C (`struct`)

To represent a linked list node, we define a `struct` with:

- an integer field (the element),
- a pointer to the next node.

Because the node points to another node of the same type, we use the more explicit `struct node` name inside the definition, and then `typedef` it to the shorter alias `node`.

```
typedef struct node
{
    int number;
    struct node *next;
}
node;
```

This pattern is extremely common in C: it solves the "self-referential struct" issue while still giving you a convenient short type name.

6.8 The Arrow Operator (`->`): A Friendlier Syntax for Struct Pointers

When you have a pointer to a struct, you often want to "go to the struct, then access a field."

You can write that in two equivalent ways:

6.8.1 The verbose way: `(*n).field`

```
(*n).number = 1;
```

This means:

- `n` is a pointer,
- `*n` dereferences it (go to the struct),
- `.` accesses a field inside the struct.

6.8.2 The common way: `n->field`

C provides syntactic sugar:

```
n->number = 1;
```

This is the same operation, but it visually resembles the arrows we draw between nodes in diagrams, and it is far easier to read and write.

6.9 Building a Linked List Step by Step

A linked list is usually represented by a single pointer to its first node, often called the **head**. In lecture, this pointer was called `list`.

6.9.1 Start with an empty list

An empty list should be explicitly initialized:

```
node *list = NULL;
```

This avoids a garbage pointer and makes it clear that there are currently no nodes.

6.9.2 Allocate a new node and fill its fields

```
node *n = malloc(sizeof(node));
if (n == NULL)
{
    return 1;
}

n->number = 1;
n->next = NULL;
```

This allocates one node and sets its "next" pointer to `NULL`, because (so far) it is the end of the list.

6.9.3 Make the list point to that node

```
list = n;
```

Now the list has one element.

6.10 Prepending: Constant-Time Insertion at the Front

A very common linked list strategy is to insert new nodes at the front. This is called **prepending**.

The core idea is:

1. Set the new node's `next` pointer to the current head.
2. Update the head pointer to point to the new node.

```
n->next = list;
list = n;
```

6.10.1 Why prepend?

Prepending is fast:

- It does not matter how long the list already is.
- You never traverse the list.
- The number of pointer updates is constant.

So, the running time for insertion via prepending is:

- **Big O(1)** (constant time)

6.10.2 The trade-off: order is reversed

If you insert `1`, then prepend `2`, then prepend `3`, the list becomes:

- `3 -> 2 -> 1`

This is correct as a list, but the order might not match what you want.

6.11 A Complete Example: Building a Linked List from Command-Line Arguments

To avoid repeatedly prompting for input, you can build a list from `argv`.

A full program outline looks like this:

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

typedef struct node
{
    int number;
    struct node *next;
}
node;

int main(int argc, char *argv[])
{
    node *list = NULL;
```

```
    for (int i = 1; i < argc; i++)
    {
        int number = atoi(argv[i]);

        node *n = malloc(sizeof(node));
        if (n == NULL)
        {
            // free memory thus far (not shown here)
            return 1;
        }

        n->number = number;
        n->next = list;
        list = n;
    }

    // Print the list
    node *ptr = list;
    while (ptr != NULL)
    {
        printf("%i\n", ptr->number);
        ptr = ptr->next;
    }

    // Free the list (not shown here)
}
```

Key ideas shown here:

- `atoi` converts a string to an integer (ASCII-to-integer).
- Each new node is prepended, so output will be reversed relative to input.
- Traversal is performed by a "walking pointer" (`ptr`) that follows `next` until `NULL`.

6.12 Memory Leaks and Orphaned Nodes: The Danger of Losing the Only Pointer

Linked lists are flexible, but they add a new kind of bug that arrays do not have.

Suppose:

- `list` points to the first node (containing `1`).
- You allocate a new node `n` (containing `2`).

If you do this incorrectly:

```
list = n;
```

without first connecting `n` to the old list, you have a problem:

- nothing points to the node containing `1` anymore.

That node still exists in memory, but your program has lost the only address that could reach it. This situation is often described as:

- the node has been **orphaned**

and it creates:

- a **memory leak**, because the program can no longer `free` that node later.

The correct order for prepending is therefore essential:

```
n->next = list;
list = n;
```

6.13 Appending: Preserving Order, but Paying Linear Time

If you want the list to preserve the input order (`1 -> 2 -> 3`), you can **append** each new node to the end instead of prepending to the front.

6.13.1 The main idea: find the tail

If `list` is empty, appending is easy:

- `list = n`

But if the list already has nodes, you must traverse until you find the last node (the one whose `next` is `NULL`), then attach the new node:

- `tail->next = n`

6.13.2 The trade-off: insertion becomes Big O(n)

Appending costs time because to append you must:

- walk through the list to reach the end.

If there are `n` nodes already, finding the end is **Big O(n)**, which means insertion is no longer constant-time.

This is an early example of a broader theme:

> Improving one property (like preserving order) may worsen another (like insertion speed).

6.14 Inserting into a Sorted Linked List: Correctness Requires More Cases

If you want the list to be kept in **sorted order** regardless of input order, insertion becomes more intricate, because you must decide where the new node belongs:

- at the beginning (if it is smaller than the current head),
- in the middle (between two nodes),
- at the end (if it is larger than everything currently present).

The key structural idea is **splicing**:

- connect the new node to the "right-hand" neighbor,
- connect the "left-hand" neighbor to the new node.

The code to do this is longer and case-heavy, but conceptually it is still just pointer manipulation with careful order of operations.

The trade-off remains:

- insertion is still **Big O(n)** in the worst case, because you might need to walk to the end.

6.15 Linked Lists vs. Arrays: The Time–Space Trade-Off

At this point, we can summarize why linked lists are appealing and what they cost.

6.15.1 What linked lists solve

Linked lists are **dynamic**:

- they can grow one node at a time,
- they do not require contiguity,
- they do not require copying all elements just to add one more.

6.15.2 What linked lists cost

Linked lists require extra memory:

- each node stores at least one extra pointer (`next`)

and they lose the key advantage of arrays:

- you cannot jump directly to the middle
- therefore, **binary search does not work** on a plain linked list

Searching a linked list is typically:

- **Big O(n)** in the worst case, because you may have to traverse the whole chain.

So linked lists trade:

- less wasted space and less expensive resizing
- for slower searching (and extra per-element metadata)

6.16 Trees: Recovering "Divide and Conquer" Without Contiguity

Arrays made binary search possible because sorted elements were contiguous. Linked lists removed contiguity, but they also removed the ability to divide and conquer efficiently.

Trees are a way to bring back "divide and conquer" structure, even when nodes are scattered throughout memory, by adding more pointers.

6.16.1 Binary Search Trees (BSTs): the rule

A **binary search tree** is a tree where each node has:

- a value (like an integer), and
- two pointers:
  - a **left** child
  - a **right** child

The BST property is:

- everything in the left subtree is **smaller** than the node's value
- everything in the right subtree is **larger** than the node's value

Because this property holds at every node, a BST is naturally **recursive**: each subtree is itself a BST.

6.16.2 A BST node in C

A node might look like this:

```
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
}
node;
```

6.17 Searching a BST with Recursion (and Why It's Elegant)

Searching a BST mirrors binary search conceptually:

- compare the target number to the current node
- decide to go left or right
- stop when you find the value or hit `NULL`

A recursive search function can be remarkably compact:

```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else
    {
        return true;
    }
}
```

This function has a clear base case:

- if the subtree pointer is `NULL`, the number is not present

and two recursive cases:

- search left subtree
- search right subtree

The code "divides the problem" implicitly by passing a smaller subtree pointer.

6.17.1 Expected running time: Big O(log n) ... sometimes

If the tree is **balanced** (roughly equal height on left and right), then searching behaves like binary search:

- **Big O(log n)**

This is attractive because it combines:

- the dynamism of linked structures (nodes can be allocated anywhere),
- with the fast search behavior of divide and conquer.

6.18 The Catch: Unbalanced Trees Can Devolve into Linked Lists

A BST does not automatically stay balanced.

If values are inserted in an unfortunate order (for example, `1`, then `2`, then `3`), the tree can become a straight chain:

- every node has only a right child

At that point, the structure is effectively a linked list in disguise, and searching becomes:

- **Big O(n)**

It is possible to fix this by using specialized "self-balancing" trees that perform rotations and rebalancing during insertion and deletion, but that is beyond the scope of this week's implementation focus. The key lesson is the conceptual one:

> A BST can be fast, but only if it remains balanced; otherwise, its performance can collapse to linear time.

6.19 Dictionaries: An ADT for Key–Value Pairs

A **dictionary** is another ADT, defined by storing **key–value pairs**:

- key: a lookup identifier (like a word, or a person's name)
- value: the associated information (like a definition, or a phone number)

A real dictionary pairs:

- words → definitions

A contacts app pairs:

- names → phone numbers (and possibly more data)

Like stacks and queues, a dictionary can be implemented in multiple ways:

- arrays (limited size, contiguity)
- linked lists (dynamic, but slower search)
- trees (can be fast, but may need balancing)
- hash tables (very fast in practice, with collisions)
- tries (very fast lookups by character, but memory-heavy)

6.20 Hashing: Mapping Large Inputs to a Finite Set of Buckets

A key idea behind hash tables is **hashing**.

A **hash function** takes an input from a potentially large (even "infinite") set and maps it to an output in a finite range.

- Domain: all possible names (huge)
- Range: bucket indices like `0..25` (small)

A helpful physical analogy is sorting playing cards into a small number of piles by suit:

- many cards $\rightarrow$ 4 buckets (hearts, spades, clubs, diamonds)

You are "hashing" each card to a bucket based on some attribute (its suit). Once bucketed, each pile is smaller and easier to work with.

6.21 Hash Tables: Arrays of Linked Lists (and Collisions)

A **hash table** is a common dictionary implementation that combines:

- an **array** of buckets, and
- **linked lists** (or other structures) to handle collisions inside each bucket.

6.21.1 The basic design (26 buckets for A–Z)

A simple approach for storing contacts is:

- create an array of size 26
- bucket each name by its first letter

Conceptually:

- bucket 0: names starting with A
- bucket 1: names starting with B
- …
- bucket 25: names starting with Z

Each array location stores a pointer:

- `NULL` if there are no names in that bucket
- otherwise, a pointer to the first node in a linked list of that bucket's entries

6.21.2 Collisions

A **collision** occurs when two different inputs hash to the same bucket.

For example:

- "Luigi", "Link", and "Lyu" all start with `L`

- they all hash to the same bucket

Collisions are not a bug in hash tables; they are an expected consequence of mapping many inputs into a small number of outputs.

The implementation response is:

- store multiple entries in the bucket using a linked list (or similar)

6.22 Hash Table Performance: Theoretical Worst Case vs. Practical Reality

In an ideal world, hashing would give constant-time lookup:

- **Big O(1)**

But collisions complicate the analysis.

6.22.1 Worst case: Big O(n)

In the worst case, all keys hash to the same bucket. Then the hash table becomes:

- one linked list containing all `n` elements

and searching becomes:

- **Big O(n)**

6.22.2 Typical case: much better than linear

If keys are distributed reasonably uniformly across buckets, then each bucket contains about:

- `n / k` elements, where `k` is the number of buckets (for example, 26)

In that scenario, searching is closer to:

- **O(n / 26)**

and while Big O notation simplifies that to O(n) (because constant factors are ignored), in real systems:

- “26 times faster” is still a meaningful improvement.

This is one of the places where it becomes important to distinguish:

- asymptotic theory (Big O),
- from real-world wall-clock time and performance engineering.

6.23 Hash Function Example in C (First Letter Only)

A simple hash function for 26 buckets can use the first letter of the word.

In C, you can compute a bucket index by converting the first letter to uppercase and subtracting `'A'`:

```c
#include <ctype.h>

unsigned int hash(const char *word)
{
    return toupper(word[0]) - 'A';
}
```

A few implementation details are worth noticing:

- `const char *word` indicates that the function does not intend to modify the string.
- `unsigned int` emphasizes that the result should not be negative.
- `toupper(word[0]) - 'A'` gives a number in the range 0–25 for alphabetic input.

This is intentionally simple, and it also highlights a limitation:

- if the first character is not a letter (punctuation, etc.), this function needs additional checks to avoid incorrect results.

In practice, real hash functions are typically more complex than "use the first letter," specifically to reduce the chance of collisions.

6.24 The Memory Trade-Off: More Buckets Reduce Collisions, But Increase Space

If collisions are a problem, you can reduce them by increasing the number of buckets.

For example, instead of hashing only the first letter, you might hash the first **three** letters. That creates buckets like:

- LAA, LAB, LAC, …, LZZ
- (and similarly for all other starting letters)

But this explodes the number of buckets:

- `26^3 = 26 * 26 * 26`

Most of those buckets will be unused (because most three-letter combinations are not the start of any real name), but you still have to allocate space for the array if you want direct indexing into it.

This is the time–space trade-off again:

- more buckets can make lookups faster by reducing collisions,
- but more buckets require more memory.

6.25 Hash Table Structures in C (Nodes and the Table)

A hash table bucket that uses linked lists needs a node structure that stores:

- the key (name),
- the value (phone number),
- a pointer to the next node (for collisions).

Conceptually:

```
typedef struct node
{
    char *name;
    char *number;
    struct node *next;
}
node;
```

And the table is typically:

```
node *table[26];
```

Each `table[i]` is:

- `NULL` (no entries),
- or a pointer to the first node of a linked list for that bucket.

6.26 Tries: A Tree of Arrays (Retrieval Tree)

A **trie** (from "retrieval") is another dictionary implementation, and it is best described as:

- a **tree of arrays**

Where a hash table is an *array of linked lists*, a trie is an *array (node) whose entries point to arrays (nodes), recursively.*

6.26.1 The core idea: one step per character

In a trie:

- each node contains an array of pointers (often size 26 for letters A–Z)
- following a pointer corresponds to consuming the next letter of a word

To store the name “TOAD”:

- go to the `T` pointer from the root
- then to `O`
- then to `A`
- then to `D`

The end of a valid word is marked explicitly (for example, by storing a value like the phone number at that node, or by having a boolean that indicates “this node ends a word”).

This explicit end-marker matters because one name can be a prefix of another:

- “TOAD” is a prefix of “TOADETTE”

A trie must represent both as valid keys, even though one continues beyond the other.

6.26.2 A trie node conceptually

A trie node might have:

- `children[26]`: pointers to child nodes
- `number`: the associated value (or `NULL` if no word ends here)

Conceptually:

```
typedef struct node
{
    struct node *children[26];
    char *number;   // non-NULL only if a word ends here
}
node;
```

6.27 Trie Performance: Effectively Constant Time (with a Big Caveat)

Searching for a name in a trie takes time proportional to the length of the name:

- “TOM” takes 3 steps
- “TOADETTE” takes 8 steps

If we assume there is a fixed maximum length for keys (names are not infinitely long), then lookup time is bounded by a constant `K`, and we often describe trie operations as:

- **Big O(1)** (constant time), in the sense that it does not grow with the number of stored keys `n`

This is a powerful property: whether you store 3 names or 3 million names, you still only traverse characters of the query key.

6.27.1 The downside: enormous memory usage

The practical cost is space.

Even if only one child pointer is used at a node, the node still contains an entire array of 26 pointers, most of which are `NULL`. Across many names, this can create a massive amount of unused pointer space.

So, tries offer extremely fast lookup behavior at the cost of substantial memory overhead, and in many real-world systems hash tables are often favored because they provide excellent performance with less memory consumption.

6.28 Data Structures Are Everywhere: A Real-World Hash Table Example

A useful way to cement this week's ideas is to recognize data structures "outside the code."

Consider a pickup shelf organized alphabetically by first name (for example, at a restaurant where mobile orders are placed under lettered sections). That system is functioning like a hash table:

- the **hash function**: "take the first letter of the name"
- the **buckets**: labeled A through Z
- the **collision handling**: multiple orders under the same letter

It performs well when names are distributed across letters, and performs worse during busy periods when many orders accumulate under the same few letters—exactly the same collision behavior we discussed in theory.

6.29 Summary: What Week 5 Adds to Your Mental Model

This week's new power is not just "more syntax," but a new way of thinking:

- Data structures can be described as **ADTs** (queues, stacks, dictionaries), where the key question is what operations they support and what guarantees they provide.
- The implementation is a separate question, and in C it ultimately comes down to how you use **memory**, **pointers**, and **structs**.

The major structures and their central trade-offs are:

- **Arrays**: contiguous, fast indexing, binary search possible; fixed-size and expensive to grow (copying).
- **Linked lists**: dynamic and easy to grow; require extra pointers and generally require linear-time searching.
- **Binary search trees**: can recover logarithmic search time if balanced; can devolve into linked-list behavior if unbalanced.

- **Hash tables**: often very fast in practice; collisions can degrade performance; quality depends heavily on the hash function and bucket count.
- **Tries**: lookups take time proportional to key length, effectively constant with respect to number of stored keys; can use enormous memory due to many null pointers.

Across all of them is the unifying theme:

> You can often save time by spending space, or save space by spending time, and good design is choosing the trade-off that fits your constraints.

Chapter 7: Week 6 — Python: From Low-Level Control to High-Level Abstraction

By week 5, you had learned to treat memory as something you can design with, using pointers, `struct`, and dynamic allocation to build your own data structures. In week 6, the course makes a deliberate shift in perspective: instead of asking you to implement everything "from scratch" in C, we introduce **Python**, a more modern, higher-level language that provides many of the same capabilities through built-in abstractions.

This shift is not a retreat from what you learned in C. Rather, it is a payoff: once you understand what is happening "under the hood," you are in a much better position to appreciate what higher-level languages are doing for you, what trade-offs they introduce, and how to use them effectively.

7.1 From C to Python: The Same Problems, Less Machinery

The most immediate difference you notice in Python is that many of the syntactic requirements of C simply disappear. Consider the simplest program you wrote at the start of the course:

7.1.1 "Hello, world" in C vs. Python

In C, printing text required several pieces of structure:

- including a library (`stdio.h`)
- defining `main`
- using `printf`
- ending lines with semicolons
- explicitly printing `\n` if you wanted a newline

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

In Python, the same output is:

```
print("hello, world")
```

Even in this tiny example, Python is already doing several things for you:

- You do not need to `#include` a standard I/O library for `print` to work.
- You do not need to define `main`.
- You do not need semicolons.
- You get a newline automatically after `print(...)`.

Python also allows either double quotes (`"..."`) or single quotes (`'...'`) for strings, as long as you are consistent. Many programmers prefer double quotes so that apostrophes inside a string are less likely to require escaping.

7.2 Compilation vs. Interpretation

In C, your workflow typically looked like this:

1. compile (often via `make`, which calls a compiler like `clang`)
2. run the compiled program (e.g., `./hello`)

Python removes the explicit compile step.

7.2.1 The Python interpreter

Python is typically run through a program called `python`, which is an **interpreter**. An interpreter reads your source code and executes it directly, top to bottom, left to right, without you manually compiling it into a separate executable first.

Python programs are conventionally stored in files ending with:

- `.py`

and run like:

```
python hello.py
```

This does not mean there is no translation happening at all (Python does compile internally to bytecode), but the key practical point remains: you do not explicitly compile and link the program each time you change it.

7.3 Libraries in Python: `import` Instead of `#include`

C uses header files (`.h`) and preprocessing directives like:

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

Python uses **imports**.

7.3.1 Importing an entire module

You can import a whole module (a library) like this:

```
import cs50
```

and then refer to functions inside it using dot notation:

```
x = cs50.get_int("x: ")
```

This style is useful when you want to avoid name conflicts (for example, if you wrote your own `get_int` function).

7.3.2 Importing specific functions

If you only want a specific function, you can import it directly:

```
from cs50 import get_int
```

and then call it without a prefix:

```
x = get_int("x: ")
```

This "fine-grained" importing can also be more efficient conceptually: you are being explicit about what you need.

7.4 Variables and Types: Fewer Declarations, Same Concepts

Python still has familiar types like integers and strings, but it does not require you to declare a variable's type in advance.

7.4.1 Assignment (still not equality)

Just like in C, the equals sign assigns right-to-left:

```
counter = 0
```

Python also supports:

```
counter = counter + 1
counter += 1
```

but **Python does not have** `++` or `--`. The designers intentionally omitted them, expecting `+= 1` and `-= 1` to be sufficient and clearer.

7.4.2 Core built-in types you will use immediately

Python includes:

- `bool` (`True` / `False` — note the capitalization)
- `int`
- `float`
- `str` (Python's name for string)

Some C types disappear from view (like `long` and `double`) because Python tries to simplify the numeric model for you. For integers in particular, Python's `int` is not limited to a fixed number of bits in the same way C's `int` typically is.

7.5 Input in Python: `input()` and Conversion with `int(...)`

Python has a built-in function:

```
input("Prompt: ")
```

It always returns a **string** (`str`), because everything typed on a keyboard is fundamentally text.

7.5.1 A greeting program

In Python, you can write a greeting program using `input()`:

```
answer = input("What's your name? ")
print(f"hello, {answer}")
```

This uses an **f-string** (format string), discussed more below.

7.5.2 Why `1 + 2` can become `"12"`

If you do:

```
x = input("x: ")
y = input("y: ")
print(x + y)
```

and type `1` then `2`, you will see:

- `12`

because `x` and `y` are strings, and `+` concatenates strings.

To add numerically, you must convert:

```
x = int(input("x: "))
y = int(input("y: "))
print(x + y)
```

Here, `int(...)` is not "casting" in the C sense; it is conversion using a function.

7.5.3 What happens if conversion fails?

If the user types something like `cat`:

```
x = int(input("x: "))
```

Python will raise a runtime error (an exception), such as a `ValueError`, because `"cat"` is not a valid integer literal.

This is one reason the CS50 Python library's `get_int()` can still be useful early on: it repeatedly prompts until the user provides an integer.

7.6 Printing and Formatting Output

Python's `print()` is more flexible than C's `printf()`. It can print strings, integers, floats, and more without format codes.

7.6.1 Concatenation with +

```
name = input("Name: ")
print("hello, " + name)
```

This works, but it requires you to manage spacing carefully.

7.6.2 Multiple arguments to `print()`

`print()` can take multiple arguments separated by commas:

```
name = input("Name: ")
print("hello,", name)
```

By default, Python inserts a single space between arguments.

7.6.3 f-strings (format strings)

An f-string is a string prefixed with `f`, which allows expressions inside `{...}`:

```
name = input("Name: ")
print(f"hello, {name}")
```

If you forget the `f`, Python will print the braces literally instead of interpolating the variable.

7.6.4 Controlling the newline: named parameters (e.g., `end=`)

By default, `print()` ends with a newline. You can override that with the **named parameter** `end`:

```
print("?", end="")
print("?", end="")
print("?")
```

or more commonly:

```
for _ in range(4):
    print("?", end="")
print()
```

Here:

- `end=""` suppresses the newline each iteration
- the final `print()` (with no arguments) prints just a newline

Named parameters like `end=` are different from the positional arguments you used in C: they explicitly name which parameter you are setting.

7.7 Conditionals in Python: Colons and Indentation

Python removes curly braces and instead uses:

- a colon (`:`) to begin a block
- indentation (typically 4 spaces) to define the block

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

A key stylistic and technical difference from C is that indentation is not just for human readability in Python. It is part of the language's syntax. Poor indentation can break your program.

Python also uses the keyword `elif` (not `else if`).

7.8 String Comparison: More Intuitive Than C

In C, comparing strings with `==` compares pointers (addresses), not characters. That is why two identical typed strings could still compare as "different."

In Python, strings are higher-level objects, and `==` compares their contents:

```
s = input("s: ")
t = input("t: ")
```

```
if s == t:
    print("Same")
else:
    print("Different")
```

Typing `cat` and `cat` yields `Same`, as you would naturally expect.

7.9 Methods and Object-Oriented Ideas: `s.lower()` Instead of `tolower(s)`

Python embraces **object-oriented programming (OOP)**. One practical consequence is that data types often come with built-in functionality.

A **method** is a function associated with an object. For strings (`str`), methods include:

- `.lower()`
- `.upper()`

Instead of writing something like `tolower(s)` (and having to apply it character-by-character), you write:

```
s = input("Do you agree? ")
s = s.lower()
```

You can also chain calls:

```
s = input("Do you agree? ").lower()
```

7.9.1 Improving "agree" with normalization and membership testing

A direct translation of a yes/no check might be:

```
s = input("Do you agree? ")

if s == "Y" or s == "y":
    print("Agreed")
elif s == "N" or s == "n":
    print("Not agreed")
```

This fails for reasonable inputs like `yes`, `YES`, or `no`.

A more robust approach:

1. convert the input to lowercase
2. check membership in a collection of allowed values

```
s = input("Do you agree? ").lower()
```

```
if s in ["y", "yes"]:
    print("Agreed")
elif s in ["n", "no"]:
    print("Not agreed")
```

Here, the keyword `in` checks whether a value is a member of a list.

7.10 Loops in Python: `while`, `for`, and `range()`

Python supports loops similar in spirit to C, but with simpler syntax.

7.10.1 `while` loops

```
i = 0
while i < 3:
    print("meow")
    i += 1
```

7.10.2 `for` loops and `range()`

A more Pythonic approach:

```
for i in range(3):
    print("meow")
```

`range(3)` yields the sequence:

- `0, 1, 2`

7.10.3 The underscore convention: "I won't use this variable"

If you do not actually use the loop variable, you can signal that with `_`:

```
for _ in range(3):
    print("meow")
```

This has no special runtime meaning; it is a readability convention.

7.10.4 Infinite loops

In Python:

```
while True:
    print("meow")
```

Note the capitalization: `True` and `False` are capitalized in Python.

7.11 Iterating Over Strings

In Python, strings are **iterable**, meaning you can loop over them character-by-character:

```
before = input("Before: ")
print("After: ", end="")
for c in before:
    print(c.upper(), end="")
print()
```

This shows how `end=""` can be used to prevent `print()` from moving to a new line each time.

However, Python strings also support an `.upper()` method that applies to the whole string at once:

```
before = input("Before: ")
after = before.upper()
print(f"After: {after}")
```

You can even place expressions directly inside an f-string:

```
before = input("Before: ")
print(f"After: {before.upper()}")
```

This is convenient when the expression is short and still readable.

7.12 Defining Your Own Functions: `def`, `main`, and Order of Definitions

Python uses the keyword `def` to define a function:

```
def meow():
    print("meow")
```

7.12.1 Why order matters

Python reads top to bottom. If you call a function before Python has seen its definition, you will get a runtime error such as:

- `NameError: name 'meow' is not defined`

This leads to a common convention: define a `main()` function, define helper functions beneath it, and then call `main()` at the bottom.

7.12.2 A conventional Python structure

```
def main():
    for _ in range(3):
        meow()
```

```
def meow():
    print("meow")

main()
```

Unlike C, Python does not automatically call `main()`. If you define it, you must call it.

7.12.3 Parameters

You can make functions more flexible by adding parameters:

```
def main():
    meow(3)

def meow(n):
    for _ in range(n):
        print("meow")

main()
```

7.13 Division, Precision, and Integer Overflow

Python changes the behavior of some arithmetic in ways that reduce common beginner mistakes, but not all numeric issues disappear.

7.13.1 Division: no integer truncation by default

In C, `1 / 3` with integers truncates to `0`.

In Python:

```
x = int(input("x: "))
y = int(input("y: "))
z = x / y
print(z)
```

If `x = 1` and `y = 3`, `z` becomes approximately:

- `0.3333333333333333`

Python returns a float when needed.

7.13.2 Floating-point imprecision still exists

Python still uses floating-point representations, so precision issues remain. You can display many digits using formatting syntax in an f-string:

```
x = int(input("x: "))
y = int(input("y: "))
z = x / y
print(f"{z:.50f}")
```

This prints 50 digits after the decimal, revealing the underlying approximation.

7.13.3 Integer overflow: not the same problem in Python

In C, integers are typically fixed-size, so they can overflow.

In Python, integers grow as needed: Python allocates more memory to store larger and larger integers. This removes the classic "wraparound" overflow behavior for `int` in typical modern Python implementations.

7.14 Exceptions: Handling Errors Without Special Return Values

In C, one of the main ways to signal errors is to return "special" values like `NULL` or `-1`. That approach can consume legitimate return values and forces programmers to check results constantly.

Python supports **exceptions**, which are "out-of-band" error signals. When something goes wrong, Python raises an exception such as:

- `ValueError`
- `NameError`

7.14.1 Catching a `ValueError` with `try` / `except`

If you attempt:

```
x = int(input("x: "))
```

and the user types `cat`, Python raises a `ValueError`.

You can handle it:

```
try:
    x = int(input("x: "))
except ValueError:
    print("Not an integer")
```

7.14.2 Implementing a robust `get_int()` with a loop

A CS50-style `get_int()` can be written by repeatedly trying until success:

```
def get_int(prompt):
    while True:
```

```
        try:
            return int(input(prompt))
        except ValueError:
            pass
```

Here:

- `while True` creates an intentional infinite loop
- `return` ends the function (and therefore stops the loop)
- `pass` does nothing, allowing the loop to try again silently

This demonstrates both exception handling and a common Python pattern: deliberately loop forever, then `break` or `return` once you have valid input.

7.15 Rewriting "Mario" Patterns in Python

Many of the looping patterns you wrote in C become smaller and clearer in Python.

7.15.1 A column of bricks

```
for _ in range(3):
    print("#")
```

7.15.2 Prompting for a positive height (a Pythonic substitute for do-while)

Python has no `do ... while` loop. A common approach is:

```
from cs50 import get_int

while True:
    n = get_int("Height: ")
    if n > 0:
        break

for _ in range(n):
    print("#")
```

This forces repeated prompting until `n` is positive.

7.15.3 A row of characters

Using `end=""`:

```
for _ in range(4):
    print("?", end="")
print()
```

Or using string "multiplication":

```
print("?" * 4)
```

7.15.4 A 3×3 grid

Nested loops:

```
for _ in range(3):
    for _ in range(3):
        print("#", end="")
    print()
```

Or using repetition for each row:

```
for _ in range(3):
    print("#" * 3)
```

7.16 Lists: Dynamic Sequences (Like Arrays, But Managed for You)

Python's **list** type is written with square brackets:

```
scores = [72, 73, 33]
```

Python manages memory for lists automatically: they can grow and shrink without you calling `malloc` or `free`.

7.16.1 Averaging scores with `sum()` and `len()`

```
scores = [72, 73, 33]
average = sum(scores) / len(scores)
print(f"Average: {average}")
```

In C, you would have needed to loop and sum manually. Python provides these building blocks directly.

7.16.2 Building a list by appending user input

```
from cs50 import get_int

scores = []

for _ in range(3):
    score = get_int("Score: ")
    scores.append(score)
```

```
average = sum(scores) / len(scores)
print(f"Average: {average}")
```

`append()` is a list method that adds to the end of the list, handling resizing for you.

7.17 Looping, Searching, and a Python Quirk: `for ... else`

Python supports a construct that looks unusual at first: a `for` loop can have an `else` clause. The `else` runs only if the loop finishes **without** hitting `break`.

A linear search might be written like this:

```
names = ["Carter", "David", "John Harvard"]
name = input("Name: ")

for n in names:
    if n == name:
        print("Found")
        break
else:
    print("Not found")
```

However, Python often lets you avoid writing the loop at all by using membership testing:

```
names = ["Carter", "David", "John Harvard"]
name = input("Name: ")

if name in names:
    print("Found")
else:
    print("Not found")
```

Python performs the linear search for you.

7.18 Dictionaries (`dict`) and Sets (`set`): Hash Tables and Uniqueness Built In

In week 5, a dictionary (in the CS sense) was an ADT, often implemented using a hash table. In Python, dictionaries and sets are built into the language.

7.18.1 Sets: collections without duplicates

A **set** is a collection that automatically removes duplicates. Python uses sets in the spell-checker example because it is convenient to test membership quickly.

In Python, a global set can store all dictionary words:

```
words = set()
```

Then checking membership becomes simple:

```
def check(word):
    return word.lower() in words
```

7.18.2 Loading a dictionary file into a set

A Pythonic way to load all lines from a dictionary file:

```
words = set()

def load(dictionary):
    with open(dictionary) as file:
        words.update(file.read().splitlines())
    return True
```

Key ideas here:

- `with open(...) as file` ensures the file is managed cleanly
- `file.read()` reads the whole file into a string
- `.splitlines()` turns it into a list of lines (words)
- `words.update(...)` inserts them into the set

Then:

```
def size():
    return len(words)
```

And `unload()` is effectively unnecessary in Python for this program, because memory is managed automatically:

```
def unload():
    return True
```

In contrast to C, Python hides `malloc`, `free`, pointers, and manual cleanup, which is a major reason programs can become dramatically shorter.

7.18.3 Dictionaries: key–value pairs (hash tables for free)

A Python dictionary associates keys with values.

One approach is a list of dictionaries (each person is a dictionary with a `"name"` and `"number"` key):

```
people = [
    {"name": "Carter", "number": "+1-617-495-1000"},
```

```
    {"name": "David",  "number": "+1-617-495-1000"},
    {"name": "John",   "number": "+1-949-468-2750"},
]
```

You can then search through `people`:

```
name = input("Name: ")

for person in people:
    if person["name"] == name:
        print(f"Found {person['number']}")
        break
else:
    print("Not found")
```

Notice the syntax:

- `person["name"]` indexes a dictionary using a string key
- it looks like array indexing, but the "index" is a key, not a number

If you only need a simple mapping from names to numbers, you can simplify further into one dictionary:

```
people = {
    "Carter": "+1-617-495-1000",
    "David":  "+1-617-495-1000",
    "John":   "+1-949-468-2750",
}
```

Then lookup becomes very direct:

```
name = input("Name: ")

if name in people:
    print(f"Found {people[name]}")
else:
    print("Not found")
```

This illustrates why dictionaries are sometimes called a "Swiss Army knife" data structure in practice: any time you want to associate one thing with another, a dictionary is a natural tool.

7.19 Command-Line Arguments in Python: `sys.argv`

In C, command-line arguments arrive via `argc` and `argv` in `main`.

In Python, they are available through the `sys` module.

7.19.1 Reading `argv`

```
from sys import argv

if len(argv) == 2:
    print("hello,", argv[1])
else:
    print("hello, world")
```

Here:

- `argv[0]` is the script name (e.g., `greet.py`)
- `argv[1]` is the first command-line argument after the script name

When you run:

```
python greet.py David
```

the word `python` is not included in `argv`; the list begins at the script name.

7.19.2 Exiting with status codes: `sys.exit`

You can import `sys` and exit with specific status codes (as you did in C by returning from `main`):

```
import sys

if len(sys.argv) != 2:
    print("Missing command-line argument")
    sys.exit(1)

print("hello,", sys.argv[1])
sys.exit(0)
```

7.20 Why Not Always Use Python? Trade-Offs: Speed, Memory, and Development Time

Python can make programs dramatically shorter, more readable, and faster to write, especially because it provides:

- powerful built-in data structures (`list`, `dict`, `set`)
- high-level I/O (`input`, `print`)
- a large ecosystem of libraries

But Python often comes with costs:

- It is frequently **slower** than C for the same algorithm, because the interpreter and runtime do more work.

- It can use **more memory**, because Python stores richer objects and manages memory for you with additional overhead.

The key trade-off is similar to the one you saw with data structures: Python often saves **human time** (developer effort) by spending **computer resources** (CPU time and RAM).

7.21 Python's Ecosystem: Doing Powerful Things with Libraries

One of Python's biggest strengths is the ecosystem of third-party libraries. Instead of implementing everything yourself, you can often "stand on the shoulders" of others.

7.21.1 Image processing with PIL (Python Imaging Library)

Using PIL, you can blur an image in only a few lines:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.BoxBlur(10))
after.save("out.bmp")
```

Similarly, edge detection can be performed with a built-in filter:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.FIND_EDGES)
after.save("out.bmp")
```

These examples mirror the kinds of computations you implemented manually in C by iterating over pixels, but they are now accessible through library abstractions.

7.21.2 Installing libraries with `pip`

Python commonly uses a tool called `pip` to install third-party packages, such as:

```
pip install face_recognition
```

With such a library, tasks like face detection and recognition can be performed with relatively little code compared to implementing computer vision yourself.

7.21.3 Fun (but real) libraries: `cowsay`

After installing:

```
pip install cowsay
```

you can write:

```
import cowsay

name = input("What's your name? ")
cowsay.cow(f"Hello, {name}")
```

This demonstrates both Python's ease of expression and the idea that importing a package can add entirely new capabilities to your programs quickly.

7.21.4 Generating QR codes with `qrcode`

After installing:

```
pip install qrcode
```

you can generate a QR code image:

```
import qrcode

img = qrcode.make("https://youtu.be/xvFZjo5PgG0")
img.save("qr.png")
```

In just a few lines, your program produces an image file (`qr.png`) that encodes a URL—an example of how Python can connect code to real-world formats and tools with minimal friction.

7.22 Summary: What Changes in Week 6 (and What Doesn't)

Week 6 introduces Python as a language that deliberately provides higher-level abstractions over the low-level mechanisms you used in C.

What changes:

- You run programs with an interpreter (`python`), not by manually compiling and running.
- Much C syntax disappears: headers, semicolons, curly braces, `main`, manual newlines.
- Indentation becomes part of program correctness.
- Many data structures are built in (`list`, `dict`, `set`) and dynamically managed.
- Strings behave more intuitively (e.g., `==` compares contents).
- Memory management is automatic: no `malloc`, no `free`.
- Exceptions provide a structured way to handle errors without special return values.
- A rich ecosystem of libraries makes advanced tasks (image filtering, face detection, QR generation) accessible quickly.

What does not change:

- The fundamental ideas of programming still apply: variables, conditionals, loops, functions, and abstractions.

- You still must think carefully about correctness, input validation, and edge cases.
- Trade-offs still exist, especially between performance and convenience.
- The skills you built in C remain valuable, because they explain what Python is doing behind the scenes and why those abstractions have costs.

Python makes many problems easier to express, but the course's deeper goal remains the same: learning how to program in a way that transfers across languages, tools, and years.

Chapter 8: Artificial Intelligence — From Decision Trees to Large Language Models

Artificial intelligence (AI) can feel like it "arrived" suddenly, because the tools that the public now uses—image generators, chatbots, and deepfakes—are unusually visible and unusually accessible. But the ideas beneath those tools connect back to foundational computer science themes you have already practiced: representing problems precisely, choosing algorithms, reasoning about efficiency, and using abstraction to manage complexity.

This chapter builds those ideas in a deliberate progression. We begin with the human-facing phenomenon—AI that can generate convincing images, video, and text—and then work backward toward the underlying mechanisms: decision trees, game-playing with minimax, the limits of brute-force search, and the shift toward machine learning, reinforcement learning, and neural networks. We end where today's popular tools live: large language models (LLMs), transformers, attention, embeddings, and the practical realities of systems that can still "hallucinate."

8.1 AI in Everyday Life (and Why It Suddenly Feels Everywhere)

AI is not only the newest generation of chatbots. It has been present for years in systems most people now take for granted:

- **Spam detection** in services like Gmail and Outlook has become so effective that many people rarely need to check their spam folder at all.
- **Handwriting recognition** has steadily improved as systems learn to adapt to many different writing styles.
- **Recommendation systems** (for movies, music, shopping, and more) often outperform any hand-written rule set, because no human can explicitly encode rules for the preferences of millions of people across millions of items.
- **Voice assistants** like Siri, Google Assistant, and Alexa rely on AI for speech recognition and natural-language understanding.

These are not "magic," but they are systems that successfully detect patterns in data and make useful predictions. What has changed recently is that AI has become dramatically better at producing *human-like outputs*—especially images, video, and text—which makes it more emotionally

compelling and, in some cases, more socially dangerous.

8.2 Generative AI and the New Problem of Believability

8.2.1 Images that no longer have obvious "tells"

In the early days of modern image generation, you could often spot obvious mistakes. A famous example was **hands**: generated images frequently produced the wrong number of fingers, unnatural joints, or inconsistent anatomy. Those details served as "tells" that revealed the image was synthetic.

But those tells are disappearing. AI-generated images can now look like plausible photographs, and sometimes even careful observers cannot reliably distinguish real images from generated ones. In demonstrations of side-by-side comparisons, a viewer may correctly identify one generated image, only to discover that a later pair of images are *both* generated.

The practical implication is that the burden of proof shifts. If "seeing is believing" becomes unreliable for images, it becomes harder to treat visual evidence as inherently trustworthy.

8.2.2 Deepfakes: when video becomes editable reality

Image generation is only part of the story. **Deepfakes** extend the same idea to video: a real person may perform motions and speak words, while software modifies the resulting footage so that it appears to be someone else. A convincing deepfake is not merely a filter; it is a synthesis of visual identity.

This matters beyond novelty, because disinformation becomes easier to create and harder to refute when it can be produced not only as text, but also as realistic images—and eventually as realistic video.

8.2.3 Text generation: the rise of ChatGPT

If image generators made AI visible, conversational text generators made AI *interactive*. Tools like **ChatGPT** create the feeling that you are talking with an assistant that can explain, summarize, rewrite, and answer questions in natural language. This interactivity has pushed AI into classrooms, workplaces, and daily routines.

One consequence is that education must decide not only what AI can do, but what students *should* do with it while learning.

8.3 AI in Education: Rubber Duck Debugging Becomes a "Duck Debugger"

8.3.1 Rubber duck debugging (the human version)

Within programming culture, there is a long-standing technique called **rubber duck debugging** (often just "rubber ducking"). The idea is simple: when your code is not working and you have no one nearby to ask, you explain your program out loud to an inanimate object—often, literally, a rubber duck.

The duck does not respond, but the act of turning vague thoughts into precise explanations often reveals the mistake. The debugging "assistant" is not the duck's intelligence; it is your own reasoning becoming clearer as you articulate it.

8.3.2 From a quacking duck to an AI teaching assistant

CS50 historically provided a digital duck that would respond with a small number of "quacks." Even that minimal interaction was sometimes enough to prompt students to realize what they had overlooked.

More recently, however, the duck has effectively "come to life" as an AI-based teaching assistant—often framed as a rubber duck persona—designed to be helpful without undermining learning.

8.3.3 Academic honesty and the need to temper helpfulness

Modern AI tools can be *too helpful*. In a course setting, a system that simply produces final answers can function like an "overzealous friend" who completes your work rather than helping you learn.

As a result, CS50's policy (as described in the course materials) prohibits using third-party AI tools that suggest or complete answers to questions or lines of code (for example, ChatGPT in its general form, GitHub Copilot, Bing Chat, and similar systems), because they can bypass the learning process.

At the same time, it would be reactionary to ignore the genuine potential for AI to support learning. CS50 therefore allows students to use **CS50's own AI-based tools**, designed to behave more like a good teacher or teaching fellow: guiding students toward answers rather than handing over complete solutions.

8.4 How CS50's AI Duck Works: Systems, Prompts, and APIs

8.4.1 APIs: borrowing capabilities from other companies

Many modern AI systems are accessed through **APIs** (Application Programming Interfaces). An API is a service boundary: another company provides a capability (like generating text), and your software sends requests and receives results.

In this model, the "heavy lifting"—the large language model computations—often happens on infrastructure operated by organizations such as OpenAI or Microsoft. A course or company can then build its own product on top of those services.

8.4.2 The problem of "recent" knowledge and the role of a vector database

Large language models are trained on data up to a certain point in time. If a model's training data ends at a cutoff date, it may not "know" about:

- last week's lecture,
- this semester's assignments,
- course-specific conventions,
- newly updated documentation.

To address this, CS50 augments the AI with course-specific information stored in a local **vector database**. The purpose of this database is to enable searching for relevant, more recent information and then providing that information to the model as additional context, so the responses can be grounded in the course's actual materials rather than only in the model's older training data.

This pattern—retrieving current or local information and injecting it into the model's context—is a practical way to improve accuracy and course relevance.

8.4.3 Prompt engineering: shaping the AI's behavior with language

In current AI practice, a common technique is **prompt engineering**, which is less like "engineering" in the traditional mathematical sense and more like carefully written instructions in English.

A particularly important concept is the **system prompt**, which is a piece of text given to the model that describes how it should behave. CS50 uses a system prompt to encourage a teaching-assistant persona and to enforce constraints such as staying on-topic and not providing full problem set solutions.

Alongside the system prompt is the **user prompt**, which is the student's actual question. Conceptually:

- The system prompt sets the role and boundaries ("how you should behave").
- The user prompt supplies the immediate task ("what the user wants right now").

This framing helps explain why the same underlying model can behave quite differently depending on how it is instructed.

8.5 Where Students Encounter the Duck: Integrated Help in the Programming Environment

CS50 students use **Visual Studio Code** through a browser-based environment at `cs50.dev`. Within that environment, the duck can provide several types of support that are particularly

useful when learning:

8.5.1 Explaining highlighted code

Students can highlight one or more lines of code and request an explanation. The duck then produces a natural-language description of what those lines do.

This is especially valuable when the code looks "arcane," as early C code often can, because the system can translate syntactic details into a narrative explanation.

8.5.2 Advising on code style and formatting

The duck can also propose improvements to code formatting and style—for instance, correcting indentation and structure so the code becomes more readable. Students can then ask the system to explain the changes, turning a purely aesthetic transformation into a learning moment about conventions and clarity.

8.5.3 Explaining error messages in plain English

Programming frequently produces error messages that are precise but unfriendly, especially to beginners. Another form of support is translating terminal or compiler errors into more approachable explanations, so students can focus on the underlying mistake instead of being blocked by unfamiliar wording.

8.5.4 AI-assisted Q&A with disclaimers and human endorsement

CS50 has also integrated the duck into an asynchronous Q&A platform used across courses. In this setting, the duck can answer questions quickly—often in seconds—providing what amounts to "virtual office hours."

Because AI is not perfect, CS50 includes disclaimers such as an explicit reminder that the duck is experimental and that students should not assume correctness unless a human has endorsed the reply. Course staff can then mark responses as correct or add additional guidance.

This hybrid approach acknowledges a key reality: even when AI is highly useful, it can still be wrong in ways that sound confident.

8.6 From "AI as Magic" to "AI as Algorithms": Starting with Games

To understand what makes AI work, it helps to begin with problems where the world can be described clearly, the possible actions are limited, and success can be measured. **Games** are ideal for this, because you can often reduce them to:

- a finite set of states (board positions, paddle positions),
- a finite set of actions (move left, move right),

- a clear objective (win, score points, survive).

8.6.1 Breakout and decision trees: turning intuition into rules

Consider *Breakout*, an early arcade game in which a paddle deflects a ball toward bricks. Humans can usually predict where the ball will go and move the paddle accordingly.

That intuitive reasoning can be represented as a **decision tree**, which is a branching structure of yes/no questions leading to actions:

- If the ball is left of the paddle, move the paddle left.
- Else if the ball is right of the paddle, move the paddle right.
- Else, do not move.

In pseudocode, that logic looks like:

- While the game is ongoing:
  - If the ball is left of the paddle: move left.
  - Else if the ball is right of the paddle: move right.
  - Else: do nothing.

This is a form of "AI" in the broad sense: it is a system that behaves intelligently in a limited domain. But it is also completely **deterministic** and **hand-authored**. A human has anticipated the relevant scenarios and encoded the response.

In simple games with predictable physics, this can be effective—but it does not scale well to domains where the number of situations explodes.

8.7 Tic-Tac-Toe, Minimax, and Scoring Game Outcomes

8.7.1 Strategy as a decision tree (win, block, otherwise…)

Tic-tac-toe is a useful next step because the state space is bigger than Breakout's immediate "left/right" choice, and because planning ahead matters.

A natural decision-tree strategy begins with two core questions:

1. Can I win on this move (get three in a row)?
   - If yes, take the winning move.
2. If not, can my opponent win on their next move?
   - If yes, block that move.
3. Otherwise, the "best" move may require thinking more than one step ahead.

This is where human intuition often becomes inconsistent: many people play reasonably but not optimally, because the consequences of a move may be several turns away.

8.7.2 Assigning numeric values to outcomes

To make reasoning more systematic, we can assign **scores** to final board outcomes. One simple scoring scheme is:

- If **O** wins: score = **-1**
- If **X** wins: score = **+1**
- If the game is a draw: score = **0**

Now the players have opposite goals:

- **X** wants to **maximize** the score (prefer +1, tolerate 0, avoid -1).
- **O** wants to **minimize** the score (prefer -1, tolerate 0, avoid +1).

This converts "winning" and "losing" into a mathematical objective.

8.7.3 Minimax: choosing moves by looking ahead

The **minimax** algorithm is built on this idea. In words:

- If it's X's turn, evaluate each possible move and choose the one with the **highest** resulting score.
- If it's O's turn, evaluate each possible move and choose the one with the **lowest** resulting score.

A crucial detail is what "evaluate" means. In tic-tac-toe, you can expand a decision tree of future moves until you reach end states (wins, losses, draws). Those leaves have known values (+1, 0, -1). You then propagate those values backward through the tree:

- At an X node (X's choice), you take the **maximum** child value.
- At an O node (O's choice), you take the **minimum** child value.

Even when a player cannot force a win, minimax can still choose the move that avoids a loss—often producing a draw if that is the best achievable outcome.

8.7.4 Why minimax becomes hard: the tree gets enormous

Tic-tac-toe is small enough that brute-force lookahead is feasible. In fact, there are **255,168** possible games of tic-tac-toe—large for a human to enumerate, but manageable for a computer.

But the same approach scales poorly:

- In chess, considering only the first **four moves** back and forth yields about **288 million** possible sequences.
- In Go, the first four moves yield about **266 quintillion** possibilities.

At that scale, even modern computers cannot simply "search everything" to guarantee optimal play, because:

- the number of branches is too large,

- the required memory is too large,
- and the required time is too large.

This limitation motivates a shift from hand-authored rules and exhaustive search toward systems that can **learn patterns** from data and experience.

8.8 Machine Learning: Letting the Computer Learn Instead of Being Told

Machine learning is a subset of AI that focuses on enabling systems to improve performance without being explicitly programmed with step-by-step rules for every scenario. Instead of writing the full decision logic ourselves, we provide:

- data,
- feedback,
- and an objective,

and the system adjusts itself to perform better over time.

One particularly intuitive form is reinforcement learning.

8.9 Reinforcement Learning: Rewards, Punishments, and Learned Behavior

8.9.1 The core idea: maximize reward

In **reinforcement learning**, an agent (a program or robot) learns by trial and error. The system performs actions, observes outcomes, and receives feedback as a numeric reward:

- **+1** for desirable behavior,
- **-1** for undesirable behavior (or "punishment").

If the agent is designed to maximize total reward, it will tend to repeat behaviors that lead to positive outcomes and reduce behaviors that lead to negative outcomes.

A key feature of reinforcement learning is that a human does not need to specify the correct action in every situation. Instead, the human (or environment) defines what counts as "good" or "bad," and the agent discovers a strategy.

8.9.2 Pancake flipping as reinforcement learning

A vivid demonstration of this concept is training a robot to flip a pancake. Initially, the robot may fail repeatedly—dropping the pancake or flipping it incorrectly. Over many trials, with feedback indicating success or failure, the robot can learn the motion pattern that reliably produces a flip.

What makes this example especially instructive is that there is no single "line of code" that describes the perfect motion. The behavior emerges from repeated attempts and feedback.

8.9.3 The "Floor Is Lava" grid: learning paths through trial and error

Another reinforcement learning scenario can be represented as a grid:

- A starting position (the agent).
- A goal square (success).
- "Lava" squares (failure).

The agent can move up, down, left, or right. It does not begin with knowledge of where the hazards are. It must explore, suffer failures, and gradually learn which moves to avoid and which moves to prefer.

Over time, the agent can learn a safe path to the goal.

8.9.4 A key trade-off: exploring vs. exploiting

A learned solution is not always the best solution. If an agent discovers *a* safe path, it can "exploit" that knowledge by repeating it, but that path might be longer than necessary.

This introduces the fundamental tension between:

- **Exploiting**: using what you already know works.
- **Exploring**: trying something new to see if there is a better option.

A common way to formalize this is to add a small probability of making a random move. That probability is often denoted **epsilon ( )**:

- With probability (for example, 10%), choose a random action (explore).
- Otherwise, choose the action with the highest known value (exploit).

The point is not randomness for its own sake; it is a disciplined way to avoid getting stuck in a solution that is locally good but globally suboptimal.

8.10 Deep Learning and Neural Networks: Learning Patterns Through Layers of Computation

Some problems are too complex for simple trial-and-error reward signals, or too high-dimensional to be handled with small tables of values. This is where **deep learning** becomes especially important.

Deep learning is closely associated with **neural networks**, which are computational structures inspired (loosely) by biological neurons. The inspiration is not that computers replicate biology exactly, but that networks of simple units can produce complex behavior when connected and tuned.

8.10.1 Neural networks as inputs, outputs, and learned parameters

A neural network can be described at a high level as:

- **Inputs**: numeric features that describe the situation.
- **Outputs**: a prediction or classification.
- **Internal structure**: layers of nodes ("neurons") connected by weighted edges.
- **Parameters**: numbers (weights and biases) that determine how inputs are transformed into outputs.

Training a neural network means adjusting those parameters so that the network's outputs match desired outcomes on many examples.

8.10.2 A simple classification example: red vs. blue points

Imagine a coordinate plane with:

- an x-axis and y-axis,
- points scattered in the plane,
- each point labeled either **red** or **blue**.

The computational task is: given a new point's (x, y), predict whether it should be labeled red or blue.

A simple approach is to find a line that separates the plane into two regions:

- points on one side are predicted blue,
- points on the other side are predicted red.

Initially, a vertical line might separate many points correctly, but as more data appears, a diagonal line might perform better. The model is trying to find the boundary that best fits the observed labels.

8.10.3 The math beneath the boundary: a linear function

This kind of separation can be expressed as a function like:

$$ax + by + c$$

If the result is greater than 0, you predict one class (say, red). Otherwise, you predict the other class (say, blue).

In this framing:

- x and y are inputs,
- a, b, and c are parameters the model must learn.

The deep-learning idea generalizes this far beyond two inputs and one line. Real neural networks can have many layers, many nodes, and enormous numbers of parameters. Even the engineers building them do not necessarily have a human-interpretable meaning for each parameter; what matters is that the system performs well on prediction tasks.

8.11 From Predicting Labels to Generating Content: Large Language Models

Deep learning can be used not only to classify ("is this red or blue?") but also to **generate** ("produce the next word," "produce an image," "produce a continuation of this sentence").

8.11.1 Large language models: predicting the next token at scale

Systems like ChatGPT are built on **large language models (LLMs)**: very large neural networks trained on massive amounts of text. You can think of them as pattern-learning machines that have absorbed statistical regularities from enormous corpora—often described informally as "a lot of the internet," plus other sources.

A practical way to understand their behavior is:

- The model reads a prompt (your input).
- It predicts what should come next.
- It repeats this process to generate a sequence of words.

Even though this process is based on prediction, the resulting text can look like reasoning, explanation, or conversation, because many examples of those forms exist in the data the model learned from.

8.11.2 Transformers and attention: handling long-range relationships in text

A major leap in language modeling came from the **transformer architecture**, introduced in a widely influential 2017 paper. One of the key ideas in transformers is **attention**.

Attention allows the model to assign different "importance" to different words in a sentence when interpreting or generating text. Intuitively:

- Some words strongly relate to other specific words.
- Some words (like articles and common prepositions) may contribute less meaningfully in many contexts.

By representing relationships between words with learned numeric values, the model can keep track of context more effectively—especially relationships between words that are far apart in the sentence.

This matters because older approaches struggled with long-distance dependencies. For example, in the sentence:

> "Massachusetts is a state in the New England region of the Northeastern United States. It borders on the Atlantic Ocean to the east. The state's capital is …"

A system must connect "The state's capital" back to "Massachusetts," despite many intervening words. Modern models are substantially better at this kind of contextual linkage.

8.12 Embeddings: Turning Words into Numbers

Computers ultimately operate on numbers. For a language model to work with text, it must represent words (and pieces of words) numerically.

An **embedding** is a numeric representation of a word (or token) that captures aspects of its meaning and usage. In modern systems, an embedding is often a high-dimensional vector—meaning a long list of floating-point numbers.

For example, one commonly used embedding representation has **1,536** floating-point values for a single word. This is not meant to be human-readable; rather, it is a mathematical object that can be compared to other embeddings to measure similarity or relationship in a way that supports learning and prediction.

Embeddings help make "semantic proximity" computational: words and phrases used in similar contexts tend to end up with vectors that are "near" each other in the embedding space, allowing the model to generalize patterns rather than memorize only exact strings.

8.13 What Can Go Wrong: Hallucinations and Overconfidence

Despite their capabilities, LLMs can produce incorrect statements that sound plausible and confident. This phenomenon is commonly called a **hallucination**.

A useful way to understand hallucinations is to remember what the model is optimizing for: producing a likely continuation of text. Sometimes the most statistically "likely" continuation is not actually true.

In practice, this is why systems like CS50's duck include disclaimers and human oversight. It is also why users must learn a new literacy: not only how to ask good questions, but how to verify claims, cross-check sources, and treat fluent output as *not automatically authoritative*.

8.14 A Closing Metaphor: The "Homework Machine" and the Limits of Apparent Perfection

A playful, cautionary metaphor for AI's usefulness—and fallibility—appears in Shel Silverstein's 1981 poem "Homework Machine," in which a "perfect contraption" produces homework answers instantly, only to reveal a ridiculous mistake:

> "Here it is, 9 plus 4,
> and the answer is 3.
> 3?
> Oh, me.

I guess it's not as perfect
as I thought it would be."

The poem captures something surprisingly modern: output can look polished and fast while still being wrong in ways that matter. As AI systems improve, the mistakes may become rarer and subtler, but the underlying lesson remains important—especially in education and in any setting where correctness matters.

8.15 Summary: A Mental Map of Modern AI

By connecting the visible world of generative AI back to foundational algorithms and learning paradigms, you now have a structured way to think about what "AI" means in practice:

- Some "AI" is simply **hand-written logic**, such as decision trees that translate human intuition into deterministic rules.
- Algorithms like **minimax** show how game-playing can be formalized with scoring and exhaustive lookahead—until the state space becomes too large.
- **Machine learning** shifts from explicit rules to systems that learn from data and feedback.
- **Reinforcement learning** uses reward signals to shape behavior through trial and error, while balancing **exploration vs. exploitation**, often with an epsilon-style randomness mechanism.
- **Deep learning** uses neural networks with many parameters to detect patterns and make predictions, even when the internal representations are not easily interpretable.
- **Large language models** apply deep learning to language generation, enabled by transformer architectures, attention mechanisms, and embeddings that represent words as high-dimensional numeric vectors.
- Today's systems can still produce confident errors, making **verification** and **careful use** essential skills alongside programming itself.

This progression—from deterministic rules, to search, to learning, to generative models—reflects the broader arc of AI: when problems become too complex for humans to encode fully, we increasingly build systems that learn patterns from the world, and we then manage the trade-offs that come with that power.

Chapter 9: Data at Scale — From CSV Files to Relational Databases with SQL

Software projects rarely stay small for long, and the moment your program has to store and analyze more than a handful of values, the question becomes less about *whether* you can write code to process data and more about *which tool is appropriate for the job*. You have already seen this pattern: using C for everything becomes painful once you want higher-level expressiveness, and Python often makes many tasks dramatically more pleasant. This chapter continues that progres-

sion by introducing a tool that is specifically designed for a particular kind of problem: storing, searching, and aggregating structured data efficiently.

That tool is **SQL**—*Structured Query Language*—a language designed not for general-purpose programming, but for asking questions of data that is stored in a **database**. The key idea is not "learn another language to collect languages," but rather to recognize a theme that appears throughout computing: different problems call for different abstractions, different runtimes, and different ways of thinking.

We will begin with a familiar data format—**CSV files**—and use Python to read, summarize, and query them. Then we will deliberately hit a point where the Python approach starts to feel tedious and "too manual," and we will use that discomfort as motivation to introduce SQL. From there we will build up relational database concepts, including **tables**, **schemas**, **primary keys**, **foreign keys**, and several common relationship patterns (one-to-one, one-to-many, and many-to-many). We will apply these ideas to a real-world dataset derived from IMDb, and we will end with two practical concerns that matter anytime databases are used in real systems: **performance** (via indexes) and **correctness/security** (via transactions and defenses against SQL injection).

9.1 Collecting and Exporting Data: Spreadsheets and "Flat File" Databases

A convenient way to gather structured data is to use a form-based interface, such as Google Forms, which automatically records responses into a spreadsheet. Conceptually, a spreadsheet is already a database-like structure: it stores information in **rows** and **columns**, where each row is typically one "record" (one form submission, one customer, one event) and each column represents one attribute (a timestamp, a name, a selection).

However, spreadsheets are primarily designed for human interaction—clicking, sorting, filtering, and using formulas—whereas programming often requires **automated** analysis and repeatable queries. The simplest bridge between those worlds is to export spreadsheet data into a plain text format that programs can read easily.

9.1.1 CSV: Comma-Separated Values

A **CSV file** (Comma-Separated Values) is a plain text representation of a table:

- Each **row** is written on its own line, separated by newline characters (`\n`).
- Each **column** within a row is separated by a delimiter, most commonly a comma (`,`).

Because CSV is plain text, it is widely portable: it can be produced by Google Sheets, Excel, Apple Numbers, and countless other systems, and it can be consumed by programs written in almost any language.

A CSV file often begins with a **header row**, which is a special first row that names the columns. For example, a file might begin with something like:

- `Timestamp,language,problem`

followed by many rows of actual data.

9.1.2 A subtlety: commas *inside* data

CSV seems simple until you notice a problem: what if a field contains a comma as part of its content? For instance, a value like `"Hello, World"` includes a comma, which would normally be interpreted as a delimiter between columns.

The conventional solution is **quoting**: if a field contains a comma (or certain other special characters), it is wrapped in double quotes so that CSV readers treat the comma as literal content rather than as a separator. Spreadsheet software typically handles this automatically when exporting.

This detail matters because it explains why "just splitting each line on commas" is not a robust way to parse CSV; you generally want to use a CSV parser library that properly handles quoting and escaping rules.

9.2 Reading CSV Data with Python

Python includes a standard module called `csv` that understands the CSV format and provides tools for reading it safely and conveniently.

Suppose you have a file named `favorites.csv`, exported from a spreadsheet. It contains responses to questions like "favorite language" and "favorite problem," along with an automatically generated timestamp. That means each row has three columns:

1. `Timestamp`
2. `language`
3. `problem`

9.2.1 Opening files: `open` vs. `with open(...)`

You can open a file in Python using:

```
file = open("favorites.csv", "r")
# use file
file.close()
```

This resembles the pattern you may recognize from C (`fopen` / `fclose`), but Python commonly encourages a more robust pattern using `with`, which automatically closes the file for you when the block ends:

```
with open("favorites.csv", "r") as file:
    # use file
# file is now closed automatically
```

The advantage is not just convenience: automatic closing is safer, especially when errors occur.

9.2.2 Reading rows with `csv.reader`

A straightforward approach is to create a **reader** that yields each row as a list of strings:

```
import csv

with open("favorites.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)  # skip the header row
    for row in reader:
        print(row[1])  # language is column 1 (0-indexed)
```

Here, `row` is a list with three elements. Because Python lists are zero-indexed:

- `row[0]` is the timestamp
- `row[1]` is the language
- `row[2]` is the problem

This works, but it quietly assumes that column positions never change. If someone rearranges the spreadsheet columns and re-exports the CSV, code like `row[1]` can suddenly refer to the wrong column.

9.2.3 Making CSV code more robust with `csv.DictReader`

To avoid fragile "magic indices," Python also provides `csv.DictReader`, which uses the header row as dictionary keys. Each row becomes a dictionary mapping column names to values:

```
import csv

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row["language"])
```

This approach is more resilient: even if column order changes, as long as the header names remain consistent, the code continues to work.

9.3 Counting Categories in Python: From Manual Counters to Dictionaries and `Counter`

Once you can read rows, a natural next step is to *summarize* the data: for example, count how many people chose each favorite language.

9.3.1 The manual approach: separate variables

One way is to create one variable per category:

```
scratch = 0
c = 0
python = 0

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        favorite = row["language"]
        if favorite == "Scratch":
            scratch += 1
        elif favorite == "C":
            c += 1
        elif favorite == "Python":
            python += 1

print(f"Scratch: {scratch}")
print(f"C: {c}")
print(f"Python: {python}")
```

This is not "wrong," but it is rigid:

- It assumes you know all categories in advance.
- It grows awkward if more categories are added (for example, if "SQL" becomes an option later).

9.3.2 A more general approach: a dictionary of counts

A **dictionary** is a natural fit for counting: use the category as the key and the running total as the value.

```
counts = {}

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        favorite = row["language"]
        if favorite in counts:
            counts[favorite] += 1
        else:
            counts[favorite] = 1
```

```
for favorite in counts:
    print(f"{favorite}: {counts[favorite]}")
```

This version adapts automatically to new categories because it does not hard-code them into variables.

9.3.3 Sorting results in Python

If you want output in a specific order, Python's `sorted` can help.

Sorting by key (alphabetical):

```
for favorite in sorted(counts):
    print(f"{favorite}: {counts[favorite]}")
```

Sorting by value (count), using `counts.get` as the key function:

```
for favorite in sorted(counts, key=counts.get):
    print(f"{favorite}: {counts[favorite]}")
```

Reversing to get largest-first:

```
for favorite in sorted(counts, key=counts.get, reverse=True):
    print(f"{favorite}: {counts[favorite]}")
```

This highlights an important Python idea: objects (like dictionaries) come with associated methods (like `.get`), and many built-in tools accept functions as parameters to control behavior.

9.3.4 The library approach: `collections.Counter`

Counting categories is so common that Python provides a dedicated tool:

```
import csv
from collections import Counter

counts = Counter()

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        counts[row["language"]] += 1

for favorite, count in counts.most_common():
    print(f"{favorite}: {count}")
```

A `Counter` automatically treats missing keys as having count 0, so you can increment without writing `if/else` initialization logic.

9.3.5 Making analysis interactive

Once you have counts, you can also build a small interactive tool:

```
favorite = input("Favorite problem: ")
print(f"{favorite}: {counts[favorite]}")
```

This is a useful pattern in data analysis: build code that loads and summarizes data once, and then allows fast queries.

9.3.6 The motivation for SQL: "I am writing too much code to ask simple questions"

Even with `Counter`, you are still writing code, loops, and bookkeeping. The moment you find yourself repeatedly writing "read every row, filter it, count it, sort it," you are experiencing the reason SQL exists: databases are built to do this kind of work directly.

9.4 Databases and SQL: A Tool Designed for Structured Queries

A **database** is software designed to store data, manage it over time, and answer questions about it efficiently. Instead of keeping data in a plain text file, you store it in a database file or on a database server, and you query it with SQL.

9.4.1 Relational databases: tables and relations

A **relational database** stores data in one or more **tables**, where each table resembles a spreadsheet sheet: rows and columns. The word "relational" emphasizes that you can have **multiple tables** with **relationships** between them, rather than one giant sheet that tries to contain everything.

9.4.2 SQL as a declarative language

SQL is typically a **declarative** language:

- In Python, you often say *how* to compute something: loop over rows, increment counters, sort a list.
- In SQL, you usually say *what* you want: "count rows where language is C," or "group by language," and the database decides how to execute that request efficiently.

9.4.3 CRUD: the basic operations

In most database systems, you can describe the main capabilities with **CRUD**:

- **Create**: add new data
- **Read**: retrieve data
- **Update**: change existing data
- **Delete**: remove data

SQL's common keywords align with these ideas:

- **INSERT**: add rows (create data)
- **SELECT**: retrieve rows (read data)
- **UPDATE**: modify rows (update data)
- **DELETE**: remove rows (delete data)

Some systems also include **DROP**, which deletes entire tables—powerful and dangerous.

9.5 SQLite: A Lightweight SQL Database

There are many database systems (with slightly different dialects of SQL), including Oracle, MySQL, PostgreSQL, and Microsoft SQL Server. In this chapter's examples, we use **SQLite**, a lightweight and widely used SQL implementation.

SQLite is common on desktops and mobile devices because it stores a database in a single local file (often with extension `.db`) and does not require running a separate server process.

9.5.1 Creating a database from a CSV file

Suppose you have `favorites.csv`. You can create a SQLite database `favorites.db` and import the CSV into a table.

From the terminal:

1. Start SQLite and create/open the database file:

```
sqlite3 favorites.db
```

2. Switch SQLite into CSV import mode:

```
.mode csv
```

3. Import the CSV as a table (here named `favorites`):

```
.import favorites.csv favorites
```

4. Quit SQLite:

```
.quit
```

You now have a binary database file `favorites.db` that contains the CSV's contents in table form.

9.5.2 Inspecting the database: `.schema`

When you open the database again:

```
sqlite3 favorites.db
```

You can ask SQLite to show the **schema**, meaning the structural definition of tables:

```
.schema
```

or for a specific table:

```
.schema favorites
```

In SQLite, commands beginning with a dot (like `.schema`) are SQLite-specific "meta commands," not standard SQL. By contrast, keywords like `SELECT`, `FROM`, `WHERE`, `GROUP BY`, and `ORDER BY` are standard SQL concepts.

9.6 Querying Data with SQL: SELECT, WHERE, LIMIT, COUNT, DISTINCT

9.6.1 Selecting rows and columns

To retrieve data, SQL uses `SELECT ... FROM ...`.

Select every column (`*`) from a table:

```
SELECT * FROM favorites;
```

Select one column:

```
SELECT language FROM favorites;
```

Because databases can contain many rows, it is often useful to limit output:

```
SELECT language FROM favorites LIMIT 10;
```

9.6.2 Counting rows

If you want the number of rows:

```
SELECT COUNT(*) FROM favorites;
```

This returns a small result table with a single value.

9.6.3 DISTINCT values

To find which distinct languages appear:

```
SELECT DISTINCT(language) FROM favorites;
```

To count how many distinct languages appear:

```
SELECT COUNT(DISTINCT(language)) FROM favorites;
```

This demonstrates a common pattern: SQL functions can be **nested**, passing the output of one function into another.

9.6.4 Filtering with WHERE

SQL uses `WHERE` to filter rows without writing loops and `if` statements.

Count how many rows have language `"C"`:

```
SELECT COUNT(*) FROM favorites WHERE language = 'C';
```

Filter by multiple conditions using boolean operators like `AND`:

```
SELECT COUNT(*)
FROM favorites
WHERE language = 'C' AND problem = 'Hello, World';
```

Notice an important difference from C and Python:

- SQL uses `=` for equality comparison (similar to Scratch), not `==`.

9.7 Aggregation with GROUP BY and ORDER BY: Doing in One Line What Took Many Lines in Python

The counting task that required dictionaries or `Counter` in Python becomes very concise in SQL.

9.7.1 Grouping and counting

Count how many times each language appears:

```
SELECT language, COUNT(*)
FROM favorites
GROUP BY language;
```

Here, `GROUP BY language` "smushes together" all rows with the same language and lets `COUNT(*)` compute how many were in each group.

9.7.2 Ordering by count

To sort these groups by how frequent they are:

```
SELECT language, COUNT(*)
FROM favorites
GROUP BY language
ORDER BY COUNT(*);
```

This sorts in ascending order by default. To reverse it:

```
SELECT language, COUNT(*)
FROM favorites
GROUP BY language
ORDER BY COUNT(*) DESC;
```

9.7.3 Aliases with AS

If you want to refer to `COUNT(*)` more easily, you can give it an alias:

```
SELECT language, COUNT(*) AS n
FROM favorites
GROUP BY language
ORDER BY n DESC;
```

9.7.4 Top results with LIMIT

To get the single most popular language:

```
SELECT language, COUNT(*) AS n
FROM favorites
GROUP BY language
ORDER BY n DESC
LIMIT 1;
```

The combination of `GROUP BY`, `ORDER BY`, and `LIMIT` is one of the most common patterns in real SQL usage.

9.8 INSERT, DELETE, and UPDATE: Changing Data (and Why It Can Be Dangerous)

SQL is not only for reading; it can modify data as well.

9.8.1 Inserting rows

To insert a new row, use `INSERT INTO ...`.

For example, you might insert a new favorite language and problem while leaving timestamp unspecified:

```
INSERT INTO favorites (language, problem)
VALUES ('SQL', 'Fiftyville');
```

If a column is not provided, it may become `NULL`, which represents the absence of a value. In SQL, `NULL` is not a pointer address; it is simply the database's way of saying "no value here."

9.8.2 Deleting rows

A dangerous command is:

```
DELETE FROM favorites;
```

This deletes **all rows**. Without a `WHERE` clause, it is catastrophic.

A safer pattern is to delete only rows matching a condition, such as removing entries with missing timestamps:

```
DELETE FROM favorites
WHERE Timestamp IS NULL;
```

The presence of `WHERE` is the difference between a targeted change and wiping a table.

9.8.3 Updating rows

Similarly, `UPDATE` can change many rows. Without a `WHERE` clause, it changes *all* rows:

```
UPDATE favorites
SET language = 'SQL', problem = 'Fiftyville';
```

This kind of command illustrates why production systems rely on careful access control, backups, review processes, and often separate "staging" databases. SQL is powerful enough to make irreversible mistakes quickly.

9.9 Designing Data Well: Normalization and Relational Structure

So far, our `favorites` data lived comfortably in a single table. Many real-world data sets do not.

To see why, consider modeling TV shows and the people who star in them.

9.9.1 The "too many columns" problem

A naïve spreadsheet design might try to represent a show like *The Office* as:

- `title`
- `star1`
- `star2`
- `star3`
- …

But shows have different numbers of stars. Some rows would need 2 star columns; others might need 20. The table becomes jagged and sparse, with many empty cells.

9.9.2 The "too much duplication" problem

A second attempt might place each (show, star) pair on its own row:

- `title`, `star`
- `The Office`, `Steve Carell`
- `The Office`, `Rainn Wilson`
- …

This avoids the jagged edge, but it repeats the string `"The Office"` many times. Repeating long strings wastes space and creates opportunities for inconsistencies (typos, formatting differences, accidental edits).

9.9.3 Normalization: store facts once, relate them with IDs

A common relational approach is to split data into multiple tables, each with a **unique identifier**:

- A **shows** table: one row per show (with a unique show ID)
- A **people** table: one row per person (with a unique person ID)
- A **stars** table: a linking table that associates show IDs with person IDs

This design:

- stores each show title once,
- stores each person name once,
- uses numeric IDs to represent relationships.

Although the `stars` table repeats the show ID multiple times, repeating a small integer is typically far cheaper than repeating the entire show title string, and it is better suited for fast indexing and joins.

9.10 IMDb as a Case Study: Multiple Tables and Real Queries

To practice SQL on large, realistic data, consider a database derived from IMDb data, stored in a file such as `shows.db`. This database includes multiple tables, including:

- `shows` (show metadata)
- `ratings` (rating and vote counts)
- `genres` (genre labels per show)
- `people` (people involved)
- `stars` (show–person starring relationships)
- `writers` (show–person writing relationships)

9.10.1 Exploring tables with small samples

A practical first step with any unfamiliar database is to inspect a few rows:

```
SELECT * FROM shows LIMIT 10;
SELECT * FROM ratings LIMIT 10;
```

You can also count scale:

```
SELECT COUNT(*) FROM shows;
```

A database like this can contain hundreds of thousands of rows, far beyond what you would want to process by hand in a spreadsheet.

9.10.2 Understanding schemas and types

Viewing schemas reveals column types and constraints.

For example, `shows` might have:

- `id` (INTEGER, primary key)
- `title` (TEXT, NOT NULL)
- `year` (NUMERIC)
- `episodes` (INTEGER)

And `ratings` might have:

- `show_id` (INTEGER, NOT NULL, foreign key)
- `rating` (REAL, NOT NULL)
- `votes` (INTEGER, NOT NULL)

SQLite uses a small family of types such as:

- **INTEGER**: whole numbers
- **REAL**: floating-point values
- **TEXT**: strings
- **NUMERIC**: often used for date/time-like values or numeric data not strictly typed as integer/real
- **BLOB**: binary large objects (files or raw bytes), though it is often better to store large files outside the database and store file paths in the database instead

Additionally, schemas can impose constraints like:

- **NOT NULL**: the value must be present
- **UNIQUE**: no duplicates allowed
- **PRIMARY KEY**: the unique identifier for rows in a table
- **FOREIGN KEY**: a value that references a primary key in another table

---

9.11 Primary Keys, Foreign Keys, and Relationship Patterns

9.11.1 Primary keys

A **primary key** is a column (or set of columns) whose value uniquely identifies a row in a table. For a `shows` table, `id` is typically the primary key: two different shows should not share the same `id`.

9.11.2 Foreign keys

A **foreign key** is a column whose values are primary keys in another table.

For example, `ratings.show_id` refers to `shows.id`. The show ID is "foreign" to the `ratings` table in the sense that the ID originates as the primary key of a different table.

9.11.3 One-to-one, one-to-many, many-to-many

Different real-world relationships map naturally to different table patterns:

- **One-to-one**: each show has exactly one rating record (as a design decision). This could be stored in the same table, but keeping it separate can be convenient or reflect data sources.
- **One-to-many**: a show can have multiple genres (comedy, adventure, family). This means one show ID appears in multiple genre rows.
- **Many-to-many**: a show can have many stars, and a person can star in many shows. This typically requires a linking table (like `stars`) containing pairs of IDs.

9.12 Subqueries and JOINs: Combining Tables to Answer Real Questions

Once data is split across tables, you need tools to recombine it when querying. SQL provides two common approaches:

1. **Subqueries** (nested `SELECT` statements)
2. **JOINs** (explicitly combining tables on matching keys)

Both are widely used.

9.12.1 A first question: shows with rating 6.0

If you query `ratings` alone, you can find show IDs:

```
SELECT show_id
FROM ratings
WHERE rating >= 6.0
LIMIT 10;
```

But those IDs are not meaningful to humans. To get titles, you can use a subquery:

```
SELECT title
FROM shows
WHERE id IN (
    SELECT show_id
    FROM ratings
    WHERE rating >= 6.0
)
LIMIT 10;
```

This reads naturally if you think inside-out:

1. Find the show IDs with rating 6.0.
2. Select titles for shows whose IDs are in that set.

However, this still does not show the ratings next to the titles, because the rating lives in a different table.

9.12.2 Joining shows and ratings to get title + rating together

A `JOIN` combines tables by aligning matching rows on a condition, typically involving a primary key and foreign key:

```
SELECT title, rating
FROM shows
JOIN ratings ON shows.id = ratings.show_id
WHERE rating >= 6.0
LIMIT 10;
```

Here, the dot notation `shows.id` means "the `id` column in the `shows` table," which is especially helpful when multiple tables have columns with similar names.

This produces a "wider" result that contains fields from both tables.

9.12.3 One-to-many joins: why duplication appears in results

When you join a one-to-many relationship, you often see repeated data in the result set.

For example, a show with multiple genres will appear multiple times when joined with the `genres` table—once per genre—because a SQL result table cannot have "gaps." Each row in the result represents one particular match between show and genre.

A query like:

```
SELECT title, genre
FROM shows
JOIN genres ON shows.id = genres.show_id
WHERE shows.id = 63881;
```

might yield:

- `Catweazle | Adventure`
- `Catweazle | Comedy`
- `Catweazle | Family`

This duplication is not necessarily "waste" in the database; it is simply a consequence of representing multiple related rows in a flat result table.

9.13 Many-to-Many Relationships: Shows, People, and the Stars Linking Table

The `stars` table exists to connect shows and people. It typically contains:

- `show_id` (foreign key to `shows.id`)
- `person_id` (foreign key to `people.id`)

This structure supports the idea that:

- one show has many people,
- one person appears in many shows.

9.13.1 Example: finding the right "The Office"

A title might not be unique in a database. If you query:

```
SELECT * FROM shows WHERE title = 'The Office';
```

you may get multiple results from different years. Adding a year disambiguates:

```
SELECT * FROM shows
WHERE title = 'The Office' AND year = 2005;
```

9.13.2 Subquery approach: who starred in The Office (2005)?

To get the stars' names, you can walk through the linking table:

1. Find the show ID for The Office (2005).
2. Find all person IDs in `stars` for that show ID.
3. Find names in `people` for those person IDs.

In SQL, that can be written as nested queries:

```
SELECT name
FROM people
WHERE id IN (
    SELECT person_id
    FROM stars
    WHERE show_id = (
        SELECT id
        FROM shows
        WHERE title = 'The Office' AND year = 2005
    )
);
```

This structure is conceptually straightforward: each layer produces the identifiers needed by the next layer.

9.13.3 Reversing the query: what shows has Steve Carell starred in?

You can also start from a person's name and walk outward:

```
SELECT title
FROM shows
WHERE id IN (
    SELECT show_id
    FROM stars
    WHERE person_id = (
        SELECT id
        FROM people
        WHERE name = 'Steve Carell'
    )
);
```

9.13.4 Joining three tables instead of nesting subqueries

You can express the same idea using explicit joins across three tables:

```
SELECT title
FROM shows
JOIN stars  ON shows.id = stars.show_id
JOIN people ON stars.person_id = people.id
WHERE name = 'Steve Carell';
```

And there is also a style that lists multiple tables and uses `WHERE` clauses to connect them:

```
SELECT title
FROM shows, stars, people
WHERE shows.id = stars.show_id
  AND people.id = stars.person_id
  AND name = 'Steve Carell';
```

These are different ways to express the same relational logic. In practice, codebases tend to standardize on a style for readability, but you will encounter all of these forms.

9.13.5 Writers as another relationship table

A `writers` table often mirrors `stars` structurally: it links shows and people, but the relationship represents "writing credits" rather than "starring credits." The distinction is carried by the table name and the data it contains, even if the schema looks similar.

9.14 Performance: Indexes and the Time–Space Trade-off

A database is not only a storage format; it is an execution engine for queries. Some queries are fast, others slow, and real systems care deeply about the difference.

9.14.1 Timing queries with `.timer`

SQLite can report how long queries take:

```
.timer on
```

Then you can run a query such as:

```
SELECT * FROM shows WHERE title = 'The Office';
```

Even a fraction of a second matters when multiplied by millions of users.

9.14.2 Indexes: making searches faster

An **index** is a data structure that accelerates searches on a column. You can create one with:

```
CREATE INDEX title_index ON shows(title);
```

This can dramatically reduce query time for searches that filter by `title`.

Under the hood, databases often implement indexes using a **B-tree** (not a binary tree). A B-tree is "short and wide": nodes can have many children, which reduces the tree's height and reduces how many steps are needed to locate a value. This is closely related to the general idea from data structures: better organization can turn slow linear scans into much faster logarithmic-time searches.

9.14.3 Indexes for foreign keys and frequently filtered columns

Primary keys are commonly indexed automatically, but foreign keys are not necessarily indexed by default. If you repeatedly filter or join on foreign keys (like `stars.show_id` or `stars.person_id`) or on frequently searched text fields (like `people.name`), indexes can help.

For example:

```
CREATE INDEX person_index ON stars(person_id);
CREATE INDEX show_index   ON stars(show_id);
CREATE INDEX name_index   ON people(name);
```

Adding these indexes can reduce multi-table queries from seconds to milliseconds.

9.14.4 The trade-off: why not index everything?

Indexes improve read performance, but they cost resources:

- **Space**: indexes take additional storage.

- **Write overhead**: inserts, updates, and deletes become slower because the database must also update index structures.

This is a classic time–space trade-off: you spend memory/disk space and maintenance work to gain speed on queries that matter.

9.15 Using SQL from Python: Combining Tools

Real applications rarely ask users to type SQL into a database prompt. Instead:

- a program (Python, Java, JavaScript, etc.) receives user input,
- the program queries the database using SQL,
- the program formats the result as a webpage, app screen, or report.

9.15.1 A Python interface to SQLite via the CS50 library

One convenient interface uses the CS50 Python library:

```
from cs50 import SQL

db = SQL("sqlite:///favorites.db")

favorite = input("Favorite problem: ")

rows = db.execute(
    "SELECT COUNT(*) AS n FROM favorites WHERE problem = ?",
    favorite
)

row = rows[0]
print(row["n"])
```

Several details matter here:

- `sqlite:///favorites.db` is a common connection-string style. The triple slash is part of the convention for a local SQLite file.
- `db.execute(...)` returns a list of rows (a result set).
- `COUNT(*) AS n` names the count column `n` so it is easy to retrieve.
- The `?` is a **placeholder** for user input.

That last point is not just syntactic convenience—it is a major security feature.

9.16 Correctness at Scale: Transactions and Race Conditions

When you have many users, many servers, and many concurrent actions, correctness becomes subtle. A database is often the shared state that everyone is trying to update, and if updates are not handled carefully, you can get results that are *plausible but wrong*.

9.16.1 The "likes" problem and race conditions

Consider a system that tracks likes on a post. A naïve approach might do:

1. `SELECT likes FROM posts WHERE id = ?`
2. In Python, compute `likes + 1`
3. `UPDATE posts SET likes = ? WHERE id = ?`

The problem is concurrency: if multiple users like the same post at nearly the same time, multiple servers might read the same old value (say, 1,000,000), then all update it to 1,000,001—effectively losing likes.

This is a form of **race condition**: the system's outcome depends on the timing and interleaving of operations.

A memorable analogy is the "milk in the fridge" problem:

- Person A opens the fridge, sees no milk, goes to buy milk.
- Person B opens the fridge before A returns, sees no milk, also buys milk.
- Now you have too much milk because each person acted on stale state.

In a database, you can get the opposite problem (lost increments) because multiple updates overwrite each other.

9.16.2 Transactions: grouping operations atomically

A common solution is a **transaction**, which groups multiple operations so they behave as one **atomic** unit: either all succeed together, or none do, and the database prevents dangerous interleavings that would corrupt the logic.

Conceptually, a program can:

- `BEGIN TRANSACTION`
- perform a sequence of reads/writes
- `COMMIT`

The idea is: do not allow another concurrent process to slip in between "read the old value" and "write the new value" in a way that causes the math to break.

Transactions are a large topic (with different isolation levels and trade-offs), but the core takeaway is simple: concurrency makes "obvious" code incorrect unless you deliberately protect multi-step updates.

9.17 Security: SQL Injection and the Importance of Placeholders

A second major real-world issue is not about speed or concurrency, but about attackers (or curious users) providing input that changes the meaning of your SQL.

9.17.1 The dangerous pattern: building SQL with string formatting

Suppose a developer writes:

```
query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
rows = db.execute(query)
```

This feels natural if you have recently learned Python f-strings. Unfortunately, it is unsafe.

In SQL (including SQLite):

- strings are quoted with single quotes `'...'`
- comments can begin with `--`, meaning "ignore the rest of the line"

An attacker could type a username like:

```
mail@harvard.edu'--
```

This closes the developer's quote and turns the remainder (including the password check) into a comment. The resulting query effectively becomes "select user where username matches," with the password condition ignored.

This is a **SQL injection attack**: user input is injected into SQL syntax, altering the query itself.

9.17.2 The safe pattern: parameterized queries with placeholders

The correct defense is to use placeholders (`?` in SQLite-style parameterization) and pass parameters separately:

```
SELECT * FROM users WHERE username = ? AND password = ?;
```

When you do this through a proper library call (as with `db.execute(..., username, password)`), the library ensures that dangerous characters are escaped or treated as literal data rather than executable SQL syntax.

The essential habit is:

- never build SQL by concatenating user input into query strings,
- always use parameterized queries (placeholders).

9.18 Summary: What SQL Adds to Your Toolkit

By moving from CSV processing in Python to relational databases with SQL, you gain a new way to think about data and a new set of tools that dramatically reduce the amount of code required for common analytics tasks.

Key concepts from this chapter include:

- **CSV files** as a portable, plain-text "flat file" representation of tabular data, including the importance of quoting to handle commas inside fields.
- Python's `csv.reader` and `csv.DictReader`, and why dictionaries keyed by header names are more robust than numeric indices.
- Counting patterns in Python, progressing from manual counters → dictionaries → `collections.Counter`, and using sorting with `sorted(...)`, including `key=counts.get` and `reverse=True`.
- **SQL** as a declarative language designed for asking questions of data, organized around CRUD operations: **INSERT**, **SELECT**, **UPDATE**, and **DELETE**.
- SQLite workflows for importing CSV data, viewing schemas (`.schema`), and querying with `SELECT`, `WHERE`, `LIMIT`, `COUNT`, `DISTINCT`, `GROUP BY`, and `ORDER BY`.
- Relational database design principles, including **normalization**, **primary keys**, **foreign keys**, and relationship patterns: one-to-one, one-to-many, and many-to-many.
- How to combine tables with **subqueries** and **JOINs**, including why one-to-many joins naturally produce repeated rows in result sets.
- Performance improvements using **indexes**, commonly implemented with B-trees, and the associated **time–space trade-off**.
- Practical systems concerns:
  - **transactions** for correctness under concurrency (avoiding race conditions),
  - **parameterized queries** to prevent **SQL injection**.

The larger lesson is consistent with the broader arc of programming: as data grows and systems become multi-user and long-lived, the "right tool for the job" becomes less optional and more fundamental. SQL and relational databases are one of the most enduring examples of that principle.

Chapter 10: The Web — Building on the Internet with HTML, CSS, and JavaScript

Modern software rarely lives in isolation on a single machine. Even when you are "just using an app," that app is usually exchanging data with services across the network, retrieving information from servers, and updating interfaces dynamically based on what it receives. This chapter introduces the foundational ideas and tools that make that possible: the Internet as a packet-switched network, the Web as a protocol layered on top of it, and the three core languages that most browsers understand—**HTML**, **CSS**, and **JavaScript**.

Two of these languages, HTML and CSS, are primarily about *presentation*: describing the structure

of a page and how it should look. The third, JavaScript, is a real programming language, used to make web pages interactive by allowing code to react to user events and modify the page dynamically.

To build confidently on top of the Web, it helps to begin with the underlying plumbing. Many of the details you have been taking for granted—addresses, routers, ports, status codes, and URLs—turn out to be practical tools for debugging and for reasoning about how your own software behaves.

10.1 The Internet: Packets, Routers, and "Protocols"

At a high level, the Internet is a collection of interconnected networks that allows computers to exchange data. Historically, it began as a small experimental network called **ARPANET** in the late 1960s and 1970s. The key technical idea that scaled from "a few computers" to "the entire world" is that information is broken into small chunks called **packets**, and those packets are routed independently across a mesh of intermediate machines.

10.1.1 Routers: the computers whose job is to forward traffic

A **router** is a device (in practice, a specialized computer) whose purpose is to forward packets toward their destinations. When you send data to a friend across the country, or request a web page from a server, your computer typically does **not** have a direct connection to the other computer. Instead, your data passes through a sequence of routers—often more than a handful, but typically fewer than a few dozen.

A crucial design feature of the Internet is that packets can take **different paths**. Even if two computers communicate repeatedly, the packets might not all travel the same route, because routers can adapt to failures, congestion, or changes in network conditions. This ability to "route around" problems is not an accident; it is a core robustness property.

10.1.2 Packets: digital "envelopes"

A **packet** is a unit of transmitted information, and it is helpful to think of it as analogous to a physical envelope:

- The **outside** of the envelope contains addressing information so the network knows where to send it.
- The **inside** contains the content (the message, part of a file, part of a web page, and so on).

The Internet is largely built out of agreed-upon rules about what goes on the outside and inside of these envelopes. Those agreed-upon rules are called **protocols**.

10.1.3 Protocols: standardized rules for communication

A **protocol** is a shared convention that both sides understand. In everyday life, a handshake is a protocol: one person initiates, the other responds, and both agree on what the motions mean.

Computer protocols work the same way: a **client** initiates, a **server** responds, and both follow a standard format so they can interpret each other correctly.

Two especially important Internet protocols are commonly used together as **TCP/IP**:

- **IP** (Internet Protocol): addressing and routing
- **TCP** (Transmission Control Protocol): reliability and ports (service identification)

10.2 IP: Addressing Computers on the Internet

10.2.1 IP addresses as "street addresses" for computers

To deliver a packet, the Internet needs addresses. An **IP address** identifies a destination machine (or, more precisely, a network interface).

A commonly seen format is:

```
number.number.number.number
```

For example:

```
1.2.3.4
```

This is an **IPv4** address (Internet Protocol version 4). Each of the four numbers is between **0 and 255**, which means each number fits in **8 bits** (one byte). Four bytes total means IPv4 addresses are **32 bits**, which yields about 2^{32} possible addresses—roughly **4 billion**.

In modern life, with billions of humans and multiple devices per human (phones, laptops, tablets, servers, "internet of things" devices), 4 billion addresses is not enough. The Internet is therefore gradually transitioning to **IPv6**, which uses **128 bits**, yielding 2^{128} possible addresses. IPv6 addresses are much longer and less human-friendly, so IPv4 remains a useful mental model even as the world migrates.

10.2.2 Source and destination: what IP contributes to packets

When a packet is sent, the outside must at least include:

- the **source address** (where the packet came from), and
- the **destination address** (where it is going).

This is what IP standardizes: how those addresses are represented, and how routers should forward packets to move them closer to their destination.

10.2.3 IP alone does not guarantee delivery

Even though IP can route packets, it does not guarantee that a packet arrives. Routers can drop packets if they are overloaded or if they run out of memory to buffer incoming traffic. In other words, the network can lose envelopes.

This is where TCP enters the picture.

10.3 TCP: Reliability and Port Numbers

TCP is layered on top of IP and provides two crucial services for many kinds of Internet communication.

10.3.1 Sequence numbers: supporting "guaranteed" delivery

If a message is large, it may be split across many packets. TCP adds **sequence numbers** so the receiver can detect missing packets and request retransmission.

You can think of this as writing "1 of 2" and "2 of 2" on envelopes. Real TCP sequence numbers are much larger than this toy example because connections may involve thousands or millions of packets, but the core idea is the same: number the pieces so they can be reassembled reliably.

TCP therefore aims to provide *reliable* communication, even though the underlying network might drop packets.

10.3.2 Ports: one computer, many services

A single machine might run multiple services:

- a web server,
- an email server,
- a chat server,
- a video conferencing service,
- and so on.

Even if packets arrive at the correct **IP address**, the machine still needs to know *which program* should receive them. TCP solves this with **port numbers**.

A **port** is a numeric identifier for a service on a machine. Two especially common ports are:

- **80** for HTTP (web traffic)
- **443** for HTTPS (web traffic over encryption)

Port numbers are not mathematically special; they are conventions that humans standardized long ago.

When you access a web server, your packets are addressed not only to an IP address, but also to a port—conceptually "IP address : port".

10.4 DNS: Translating Domain Names to IP Addresses

Humans do not like typing IP addresses. We prefer names like:

- `harvard.edu`
- `yale.edu`
- `google.com`

These are **domain names**, and computers must translate them into numeric IP addresses before sending packets. That translation is performed by **DNS**: the **Domain Name System**.

10.4.1 DNS servers as a global dictionary

A **DNS server** is a server whose job is to answer questions of the form:

> What is the IP address for this domain name?

You can think of DNS as a globally distributed dictionary (similar in spirit to a hash table):

Domain name	IP address
harvard.edu	…
google.com	…

DNS is **hierarchical**. Your computer typically asks a nearby DNS server (often operated by your ISP, your university, or your company). If that server does not know the answer, it asks other DNS servers "higher up" in the hierarchy, eventually reaching **root servers** that coordinate knowledge about top-level domains like `.com`, `.edu`, and country-code domains.

10.4.2 Hostnames, top-level domains, and fully qualified domain names

In a URL like:

- `www.example.com`

you can distinguish:

- `www` as a **hostname** (a conventional label for a particular server or role),
- `example` as the organization's domain, and
- `.com` as the **top-level domain (TLD)**.

Historically, `www.` was used to signal "this is a web address," but it is increasingly common for sites to omit it (e.g., `cs50.dev`, `cs50.ai`). Whether a site uses `www` is largely a configuration and branding choice; either can be redirected to the other.

10.5 DHCP: Automatically Configuring Devices

In early networking, a person might manually configure a computer's network settings: which IP address to use, which DNS server to ask, and which router to send packets to. Today, most configuration happens automatically through **DHCP**: the **Dynamic Host Configuration Protocol**.

A **DHCP server** answers questions like:

- "What IP address should I use on this local network?"
- "What is my DNS server?"
- "What is my default router (gateway)?"

When you open your laptop or connect your phone to a network, your device broadcasts a request, and a nearby DHCP server replies with the configuration it should use.

10.6 HTTP and HTTPS: The Web as a Protocol

The **World Wide Web** is built on **HTTP**: the **Hypertext Transfer Protocol**. HTTP is one more protocol layered on top of TCP/IP, but this time it standardizes what goes *inside* the packets to allow a browser (the client) and a web server to communicate.

10.6.1 HTTP vs. HTTPS

- **HTTP** is plain text: if someone can intercept your packets, they can read the contents.
- **HTTPS** is HTTP over encryption: traffic is scrambled using cryptography so that interception does not reveal the message contents in any practical way.

This is why HTTPS is used for logins, payments, and increasingly for essentially all web traffic.

10.6.2 URLs: the parts of a web address

A canonical URL looks like:

`https://www.example.com/folder/file.html`

Key components include:

- **Scheme / protocol**: `https` (or `http`)
- **Hostname and domain**: `www.example.com`
- **Path**: `/folder/file.html`

The path often corresponds conceptually to a file or folder on the server, though modern servers often hide file extensions (`.html`) for cleaner URLs.

A URL can also include a **query string** (discussed later), which begins with `?` and contains key–value pairs.

10.7 GET and POST: Two Common Ways to Make Requests

HTTP supports multiple request "methods," but two of the most important are:

- **GET**: request information (the common method when clicking links and visiting pages)
- **POST**: submit information (common for forms that send data such as passwords, uploads, or payment details)

For much of this chapter, GET is sufficient, because we are learning the mechanics of requesting pages and sending simple form inputs.

10.8 HTTP Messages: Requests, Responses, Headers, and Status Codes

When you type a URL into a browser and press Enter, your browser sends an HTTP **request** message to the server. The server sends back an HTTP **response** message.

10.8.1 A simple HTTP request

A request resembles:

```
GET / HTTP/2
Host: www.harvard.edu
...
```

Important parts:

- `GET` is the method.
- `/` is the path (the "default" page at the root).
- `HTTP/2` is the protocol version (you may also see `HTTP/1.1` and increasingly HTTP/3).
- `Host: ...` is a **header**, a key–value line that helps the server understand what the client wants.

The `Host` header matters because a single physical server can host multiple websites. The host header lets the server know which site's content to return.

10.8.2 A simple HTTP response

A response resembles:

```
HTTP/2 200 OK
Content-Type: text/html
...
```

Important parts:

- `200` is a **status code** indicating success.
- `Content-Type: text/html` tells the browser what kind of content it received (here, HTML).

10.8.3 Inspecting HTTP with `curl`

A convenient command-line tool for observing HTTP responses is `curl`. For example:

```
curl -I https://www.harvard.edu/
```

The `-I` flag requests only headers (not the full page content), making it easier to focus on status codes and metadata.

10.8.4 Browser Developer Tools: Network inspection

Modern browsers include **developer tools**, often with a **Network** tab that shows every HTTP request made while loading a page. This is useful because loading a single page typically triggers dozens of requests: the initial HTML plus images, CSS files, JavaScript files, fonts, and more.

10.8.5 Status codes: what the numbers mean

HTTP status codes communicate what happened. Common examples include:

- **200 OK**: the request succeeded.
- **301 Moved Permanently**: the server is redirecting you to a new URL.
  - For example, visiting `https://harvard.edu/` might redirect to `https://www.harvard.edu/`.
- **404 Not Found**: the requested path does not exist on the server.
- **403 Forbidden**: you are not allowed to access the resource (often due to missing authentication).
- **500 Internal Server Error**: the server encountered a bug or misconfiguration (a "server-side failure," often analogous in spirit to "something crashed").
- **418 I'm a teapot**: an April Fool's joke status code that exists as a piece of internet folklore; some servers may still recognize it.

Status codes beginning with:

- **3xx** often relate to redirection,
- **4xx** generally indicate a client-side problem (bad request, forbidden, not found),
- **5xx** generally indicate a server-side problem.

10.9 HTML: Structuring Web Pages with Tags and Attributes

HTML (Hypertext Markup Language) is a **markup language**, not a general-purpose programming language. It describes the *structure* of a document: headings, paragraphs, lists, tables, links, forms, and so on.

HTML is built from two core ideas:

- **Tags** (like `<p>` or `<title>`)
- **Attributes** (like `lang="en"` or `href="https://..."`)

10.9.1 A minimal HTML page

A simple example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>hello, title</title>
  </head>
  <body>
    hello, body
  </body>
</html>
```

Key concepts:

- `<!DOCTYPE html>` is the **document type declaration**, typically copied and pasted; it signals modern HTML (HTML5).
- `<html lang="en">` begins the HTML document and declares the language of the page content.
- `<head>` contains metadata such as the `<title>`.
- `<body>` contains the main visible content in the viewport.

An **attribute** is written as `key="value"`. The language of HTML heavily uses key–value pairs, but with syntax different from other languages.

Many tags come in pairs:

- an opening tag: `<title>`
- a closing tag: `</title>`

The content between them forms an **element**.

10.9.2 Running a local web server in a development environment

If your HTML file lives locally on your computer, you can often open it directly by double-clicking it. In a cloud development environment (such as a hosted code space), you typically run a web server to serve files to your browser.

A simple development server is:

```
http-server
```

This starts a server that listens for HTTP requests. In many setups, it uses port **8080** by default (instead of 80 or 443), because those standard ports are often already in use or restricted.

When you visit the server's URL, you may see a **directory listing**, which is simply a web page showing files in the current folder. Clicking a file like `hello.html` causes the browser to request it, and the server responds with that file's contents.

10.9.3 HTML ignores most whitespace

A common surprise is that extra newlines and spaces in HTML do not necessarily appear on the page. HTML generally **collapses sequences of whitespace** into a single space. This is why writing three "paragraphs" separated only by blank lines still looks like one continuous block.

To create actual paragraphs, you use the paragraph tag:

```
<p>First paragraph...</p>
<p>Second paragraph...</p>
<p>Third paragraph...</p>
```

10.10 Common HTML Elements: Headings, Lists, Tables, Images, Video, Links

HTML provides many tags. A few appear constantly in real pages.

10.10.1 Headings: `<h1>` through `<h6>`

Headings provide structure and default styling:

```
<h1>Heading 1</h1>
<h2>Heading 2</h2>
<h3>Heading 3</h3>
```

`<h1>` is the largest by default, and sizes decrease through `<h6>`.

10.10.2 Lists: unordered and ordered

An **unordered list** uses `<ul>` with list items `<li>`:

```
<ul>
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ul>
```

An **ordered list** uses `<ol>`:

```
<ol>
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
```

The browser handles numbering automatically, which is useful if you insert items in the middle.

10.10.3 Tables: rows and cells

A basic table uses:

- `<table>` as the container
- `<tr>` for table rows
- `<td>` for data cells

For example, a numeric keypad layout can be represented by rows and cells. For more structured tables (like a phonebook), HTML often distinguishes:

- `<thead>` for the header section
- `<tbody>` for the data section

This is especially useful for styling and semantics.

10.10.4 Images: `<img>` with `src` and accessibility via `alt`

An image tag includes a source file:

```
<img src="bridge.png" alt="Photo of bridge">
```

Two key attributes:

- `src`: which image file to load
- `alt`: **alternative text** used by screen readers and displayed if the image fails to load

Including `alt` text is a best practice for accessibility.

10.10.5 Video: `<video>` and `<source>`

Embedding video looks like:

```
<video controls muted>
  <source src="video.mp4" type="video/mp4">
</video>
```

Notable details:

- Some attributes (like `controls` and `muted`) are **boolean attributes**: they do not need `= "true"`; the presence of the word enables the feature.
- The `type` attribute uses a MIME/content-type style string such as `video/mp4`.

10.10.6 Links: the anchor tag `<a>`

A hyperlink uses the **anchor** tag:

```
<a href="https://www.harvard.edu/">Harvard</a>
```

The human-visible text does not have to match the `href`. This flexibility powers ordinary web design, but it also enables **phishing**: displaying a trustworthy-looking URL while linking somewhere else.

10.11 Metadata: `<meta>` Tags for Mobile and Link Previews

The `<head>` of a page can contain metadata beyond the title.

10.11.1 Mobile-friendly viewport settings

A common meta tag for mobile friendliness is:

```
<meta name="viewport" content="initial-scale=1, width=device-width">
```

This helps the browser scale content appropriately on phones and tablets.

10.11.2 Social media link previews ("Open Graph" style metadata)

Web pages can include metadata that social platforms use to generate rich previews when a link is pasted. Such tags specify:

- the title shown in the preview,
- a short description,
- and an image.

This gives the page author control over how the page appears when shared.

10.12 Forms and Query Strings: Sending Input with GET

A **form** is the basic HTML mechanism for collecting user input and sending it to a server.

10.12.1 A search form that sends data to Google

A simplified version:

```
<form action="https://www.google.com/search" method="get">
  <input name="q" type="search" placeholder="query" autocomplete="off" autofocus>
  <input type="submit" value="Google Search">
</form>
```

Important ideas:

- `action` is the URL that will receive the form submission.
- `method="get"` means the browser will encode the form input into the URL's query string.
- The input's `name` is the **key** used in the query string. Google expects the key `q`.

If the user types `cats`, the resulting request URL becomes:

`https://www.google.com/search?q=cats`

10.12.2 Query strings as key–value pairs

A query string starts with `?` and contains `key=value` pairs. Multiple pairs are separated with `&`, such as:

`/search?q=cats&lang=en`

This is a simple but powerful web convention: it is one of the primary ways data is passed from a browser to a server using GET.

10.12.3 Useful input attributes: `placeholder`, `autocomplete`, `autofocus`, and `type`

- `placeholder` shows hint text in the input before typing.
- `autocomplete="off"` disables the browser's suggestions (useful sometimes, though often autocomplete is user-friendly).
- `autofocus` places the cursor in the field automatically.
- `type` influences validation and user experience:
  - `type="text"` (default)
  - `type="search"` (may add features like a clear "x" in some browsers)
  - `type="email"` (asks the browser to enforce an email-like format)

10.13 Client-Side Validation and Regular Expressions (Regex)

Browsers can validate input before sending it to a server, which can improve user experience by catching mistakes early.

10.13.1 Email validation with `type="email"`

If you use:

```
<input type="email">
```

most browsers will reject submissions that lack an `@` or otherwise clearly fail to resemble an email address.

10.13.2 Restricting formats with `pattern=...`

You can add a `pattern` attribute containing a **regular expression** (regex). For example, to require an address ending in `.edu`:

```
<input
  name="email"
```

```
  type="email"
  placeholder="email"
  pattern=".+@.+\.edu"
  autocomplete="off"
  autofocus
>
```

A few regex building blocks used here:

- `.` matches (almost) any single character
- `+` means "one or more occurrences"
- `\.` escapes a dot so it means a literal period instead of "any character"

So `.+@.+\.edu` roughly means:

- one or more characters,
- then an `@`,
- then one or more characters,
- then a literal `.edu`.

Real email validation is more complicated than this; full email regexes can become long and intimidating. The important takeaway is that regex is a powerful pattern language for matching and validating strings.

10.13.3 Why client-side validation is not security

Client-side validation is **not trustworthy** as a security measure, because the user controls their browser. Browser developer tools allow a user to:

- inspect the HTML and CSS,
- edit the page locally,
- remove attributes like `pattern`,
- and then submit values that the original HTML tried to reject.

Client-side validation is best understood as:

- good for user experience,
- not sufficient for enforcing rules or protecting systems.

Server-side checks are still required to ensure correctness and security.

10.13.4 Validating HTML syntax with the W3C validator

To check whether your HTML is syntactically valid, a widely used tool is the W3C validator:

- `https://validator.w3.org`

It can analyze HTML you paste, upload, or point to by URL, and report errors and warnings.

10.14 CSS: Making Pages Look Better (and More Consistent)

CSS stands for **Cascading Style Sheets**. CSS controls presentation: font sizes, colors, spacing, alignment, borders, layout, and more.

In CSS you again see key–value pairs, but in a different syntax:

```
property: value;
```

CSS is applied by selecting HTML elements (or groups of elements) and attaching properties to them.

10.14.1 Inline styles: the `style="..."` attribute

You can attach CSS directly to an element:

```
<p style="font-size: large; text-align: center;">John Harvard</p>
```

This works, but it tends to produce repetition and clutter in larger projects.

10.14.2 Cascading: applying styles to parents affects children

Rather than centering each element individually, you can center everything inside `<body>`:

```
<body style="text-align: center;">
  ...
</body>
```

This illustrates the "cascading" idea: styles applied to a parent often flow down to its children.

10.14.3 Semantic HTML tags: `<header>`, `<main>`, `<footer>`

HTML5 includes tags that communicate meaning:

```
<header>John Harvard</header>
<main>Welcome to my homepage!</main>
<footer>&copy; John Harvard</footer>
```

These tags may not change appearance by themselves, but they improve semantics for:

- search engines (SEO),
- screen readers (accessibility),
- and developers reading the code.

The copyright symbol can be written using an **HTML entity**, such as:

```
&copy;
```

(One way to represent it is by numeric entities as well, e.g., a code corresponding to the copyright symbol.)

10.14.4 Style blocks: using `<style>` in the head

Instead of inline styles, you can put CSS in a `<style>` tag:

```
<head>
  <style>
    body { text-align: center; }
    header { font-size: large; }
    main { font-size: medium; }
    footer { font-size: small; }
  </style>
</head>
```

This separates content (HTML) from presentation (CSS), improving readability and maintainability.

10.14.5 Classes: reusable groups of properties

You can define reusable class-based styles:

```
.centered { text-align: center; }
.large { font-size: large; }
.medium { font-size: medium; }
.small { font-size: small; }
```

…and apply them in HTML with the `class` attribute:

```
<body class="centered">
  <header class="large">John Harvard</header>
  <main class="medium">Welcome to my homepage!</main>
  <footer class="small">&copy; John Harvard</footer>
</body>
```

A **class selector** begins with a dot (`.`) in CSS.

10.14.6 IDs: selecting a unique element

Sometimes you want to style one specific element. You can give it an `id`:

```
<a id="harvard" href="https://www.harvard.edu/">Harvard</a>
<a id="yale" href="https://www.yale.edu/">Yale</a>
```

Then style by ID:

```
#harvard { color: red; }
#yale { color: blue; }
```

An **ID selector** begins with a hash (`#`) in CSS.

10.14.7 Pseudo-classes: styling states like `:hover`

CSS can select not just elements but *states*. For example, to remove underlining by default but show it on hover:

```
a { text-decoration: none; }
a:hover { text-decoration: underline; }
```

This is common on desktop sites, where a mouse cursor can hover. It is less meaningful on mobile devices.

10.14.8 External style sheets: linking a `.css` file

For larger projects, CSS is commonly placed in a separate file, such as `home.css`, and linked:

```
<link href="home.css" rel="stylesheet">
```

This enables reusing a single style sheet across many pages and collaborating more cleanly (one person can focus on HTML content while another focuses on CSS design).

10.14.9 Using third-party CSS frameworks: Bootstrap

Writing good CSS can take time, and many common interface patterns have already been solved by others. **Bootstrap** is a popular CSS framework that provides a large set of predefined classes.

A page can include Bootstrap by linking to its CSS:

```
<link
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
  rel="stylesheet"
>
```

Then, HTML elements can be given Bootstrap classes, such as:

- `btn` and `btn-light` for buttons
- table-related classes for styled tables

For example, a search page can look significantly more polished by applying Bootstrap classes rather than writing custom CSS from scratch. The conceptual pattern is the same as your own `.centered` or `.large` classes—Bootstrap simply provides many more of them, along with careful design choices.

10.14.10 Learning from real sites via developer tools

Browser developer tools often show the CSS rules currently applied to an element and where they came from. This makes it possible to:

- experiment with style changes live (locally in your browser),
- discover which CSS rules affect a specific element,
- and trace a computed value back to the relevant style sheet rule.

10.15 JavaScript: Programming in the Browser

JavaScript is a full programming language that runs in the browser (and can also run on servers). In the browser, JavaScript's most prominent role is to make pages interactive by:

- responding to user actions (clicks, typing, submissions),
- reading and modifying the HTML structure of the page, and
- changing styles dynamically.

10.15.1 The DOM: the page as a tree in memory

When the browser parses HTML, it builds an internal tree representation called the **DOM**: the **Document Object Model**. You can think of it as the in-memory version of the HTML structure (a tree of nodes: elements and text).

JavaScript can query and manipulate this tree, which is why many modern web apps can update dynamically without a full page reload.

10.15.2 Script tags and functions

JavaScript is embedded using a `<script>` tag:

```
<script>
  function greet() {
    alert("hello, " + document.querySelector("#name").value);
  }
</script>
```

A few key ideas appear here:

- JavaScript defines functions with the keyword `function`.
- The browser provides an `alert(...)` function that shows a pop-up.
- `document.querySelector(...)` selects an element from the DOM using a CSS-like selector.
  - `#name` refers to an element with `id="name"`.
- `.value` retrieves the value of an input element.

10.15.3 Responding to form submission: `onsubmit` (inline) vs event listeners

One way to handle a form submit is to put JavaScript directly in HTML:

```
<form onsubmit="greet(); return false;">
  <input id="name" autocomplete="off" autofocus>
  <input type="submit">
</form>
```

Here, `return false` prevents the browser from performing the form's default submission behavior (which would usually send a request and reload/navigate).

However, mixing JavaScript into HTML attributes is often avoided in larger codebases. A more typical approach is to attach an event listener in JavaScript:

```
<script>
  document.querySelector("form").addEventListener("submit", function (event) {
    alert("hello, " + document.querySelector("#name").value);
    event.preventDefault();
  });
</script>
```

Key ideas:

- `addEventListener("submit", ...)` tells the browser what code to run when the event happens.
- `event.preventDefault()` prevents the default form submission.

10.15.4 Waiting for the DOM to load: `DOMContentLoaded`

Sometimes you want to ensure the page has been fully parsed into the DOM before running code. Browsers provide a `DOMContentLoaded` event:

```
<script>
  document.addEventListener("DOMContentLoaded", function () {
    document.querySelector("form").addEventListener("submit", function (event) {
      alert("hello, " + document.querySelector("#name").value);
      event.preventDefault();
    });
  });
</script>
```

This pattern ensures the relevant elements exist before JavaScript tries to select them.

10.16 JavaScript and CSS Together: Dynamic Styling

JavaScript can change CSS properties by modifying an element's `style` object. A simple demonstration is changing the page's background color in response to button clicks.

10.16.1 Buttons that change the background color

HTML might include:

```
<button id="red">R</button>
<button id="green">G</button>
<button id="blue">B</button>
```

JavaScript might do:

```
let body = document.querySelector("body");

document.querySelector("#red").addEventListener("click", function () {
  body.style.backgroundColor = "red";
});

document.querySelector("#green").addEventListener("click", function () {
  body.style.backgroundColor = "green";
});

document.querySelector("#blue").addEventListener("click", function () {
  body.style.backgroundColor = "blue";
});
```

Notice the CSS property name changes form:

- In CSS: `background-color`
- In JavaScript: `backgroundColor`

The hyphen would otherwise be interpreted as subtraction in JavaScript, so the convention is to use **camelCase** for such property names.

Developer tools can show these style changes occurring live, which is valuable for understanding and debugging dynamic behavior.

10.17 Timers and Effects: Recreating "Blink" with `setInterval`

Even if a feature is not built into HTML as a tag, JavaScript can recreate effects by manipulating styles over time.

A "blink" effect can be created by toggling `visibility` repeatedly:

- `visible`
- `hidden`

A timer can be scheduled with:

```
window.setInterval(blink, 500);
```

This calls the function `blink` every 500 milliseconds. A subtle but important point is that you pass the *function name* (a reference to the function), not the result of calling it. That is why you write `blink`, not `blink()`.

10.18 Autocomplete: Updating the DOM in Response to Typing

Many modern interfaces react immediately to user input. Autocomplete is one example: as the user types, a list of suggestions updates.

A common pattern involves:

- listening for keyboard events such as `keyup`,
- filtering a list of possible words,
- and dynamically generating list items (`<li>`) inside an existing `<ul>`.

A JavaScript program can build a string of HTML and then insert it into the DOM using something like an element's `innerHTML`. In effect, JavaScript is generating new HTML "on the fly," updating the in-memory DOM tree so the page changes without a reload.

This technique is powerful, but it also demands careful design in real systems, because generating HTML dynamically has security implications if user-provided data is inserted unsafely. (The broader theme is the same as earlier: input is tricky, and trust boundaries matter.)

10.19 Geolocation: Browser APIs and Privacy Implications

Browsers provide powerful APIs beyond HTML and CSS. One striking example is **geolocation**. If a user grants permission (and has location services enabled), JavaScript can obtain the user's approximate latitude and longitude with a browser-provided object:

- `navigator.geolocation.getCurrentPosition(...)`

This enables applications like ride-sharing and maps, but it also illustrates an important reality: client-side code can access sensitive information if users allow it, and developers have a responsibility to understand both the functionality and the privacy implications of what they build.

10.20 Summary: From Packets to Pages

This chapter connected the Web's surface-level experience—typing URLs, clicking links, submitting forms—to the deeper mechanisms that make it work.

Key ideas include:

- **Packet routing** through **routers**, enabling robust communication even when paths change.
- **TCP/IP** as the foundational pair of protocols:
  - **IP** for addressing and routing via IP addresses,
  - **TCP** for reliability (sequence numbers) and service targeting (ports like 80 and 443).
- **DNS** as a distributed naming system translating domain names into IP addresses.
- **DHCP** as the protocol that configures devices automatically with network settings.
- **HTTP/HTTPS** as the web's application protocol, including:
  - requests and responses,
  - headers,
  - status codes like 200, 301, 404, and 500,
  - diagnostic tooling via `curl` and browser developer tools.
- **HTML** as a markup language for document structure, including tags, attributes, and common elements like paragraphs, headings, lists, tables, images, video, and links.
- **Forms** and **query strings** as a simple but central mechanism for sending input to servers via GET.
- **Regular expressions** via the `pattern` attribute for client-side validation, paired with the crucial warning that **client-side checks are not security** because users can modify HTML locally.
- **CSS** as a styling language with selectors, properties, cascading behavior, classes, IDs, pseudo-classes like `:hover`, external style sheets, and frameworks like **Bootstrap**.
- **JavaScript** as the browser's programming language, enabling event-driven interactivity, DOM manipulation, dynamic styling, timers, autocomplete behavior, and access to browser APIs like geolocation.

With these foundations in place, you are prepared to start building software that feels like "real" modern applications: interfaces that are networked, interactive, and layered—where the browser becomes a runtime environment, and the Internet becomes the medium through which your programs communicate.

Chapter 11: Flask — Building Dynamic Web Applications with Python

In the previous chapter, we treated the browser primarily as a viewer for **static files**: HTML documents, CSS style sheets, JavaScript programs, and media assets that already existed on disk before any user ever visited the site. A simple web server such as `http-server` is perfectly suited to that world, because its job is essentially to map a URL path to a file on the server and then

return that file's bytes as an HTTP response.

Modern web experiences, however, are rarely that static. When you search on Google, no engineer has prewritten an HTML file for every possible query. When you open Gmail, no one has pre-generated a web page for the precise set of emails that will be relevant to you at this moment. Instead, a modern web application takes **input**, performs **logic**, often consults **data storage** (such as a database), and then produces **output**—frequently generating HTML dynamically, and increasingly exchanging data in machine-readable formats such as JSON.

This chapter introduces **Flask**, a popular Python "microframework" for building web applications. The term *framework* here means two things at once:

1. **Code** you can reuse (functions and libraries that solve common problems, like parsing query strings or issuing redirects), and

2. **Conventions** you adopt (standard file names, folder structures, and patterns for organizing your application).

Flask is called a *microframework* because it is comparatively small and lightweight; it solves a core set of web-application problems without forcing you into an unusually large or complex system. There are larger frameworks (for example, Django), but Flask provides a clean way to learn concepts that transfer well to other environments.

11.1 From URLs-as-Filenames to URLs-as-Routes

When you serve static content, it is natural to think of URLs as mapping directly to files:

- `/` often maps to a default file such as `index.html`.
- `/folder/` might map to `folder/index.html`.
- `/folder/file.html` maps to a specific file.

Once you write your own server logic, you can treat everything after the domain name more abstractly. Instead of thinking "this path must correspond to a real file," you can think:

- Everything after the domain is a **path**, and in web-application terminology that path is often called a **route**.
- A route is simply a string such as `/`, `/greet`, `/register`, or `/search`.
- A route is associated with **code**, not necessarily with a file.

This shift is one of the core mental changes when moving from static sites to web applications: a path is no longer just a location on disk; it is an entry point into your program.

11.2 Flask's Typical Project Layout

Flask does not require one particular layout, but it strongly encourages conventions that make projects predictable. A common structure is:

- `app.py`
  The main Python program (the "controller" logic) that defines routes and what they do.
- `templates/`
  A folder containing HTML templates (often with placeholders and control flow). Flask expects templates here by convention.
- `static/`
  A folder for files served "as-is," such as `.css`, `.js`, and images. These are called *static* not because they never change during development, but because the server does not dynamically generate them per-request.
- `requirements.txt`
  A list (one per line) of third-party libraries your project depends on, so that installation is repeatable.

These conventions matter because they reduce "where did we put that file?" friction, especially when collaborating, debugging, or returning to a project later.

11.3 Your First Flask Application: A Route That Returns Text

A minimal Flask program begins by importing Flask and creating an "app" object:

```
from flask import Flask

app = Flask(__name__)
```

The argument `__name__` is a special Python variable that evaluates to the current module's name, and it is a conventional way to let Flask know where your application code lives without hard-coding filenames.

To associate a URL route with code, Flask uses a *decorator* syntax: `@app.route(...)`. Conceptually, you are telling Flask, "When someone requests this path, call this function."

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "hello, world"
```

If you run this application with Flask's development server (commonly via `flask run`), visiting the server's `/` path produces the response body `hello, world`.

At this stage, notice what is happening: you are not serving an HTML file at all. You are producing a response body directly from Python. The browser will display it, but "View Page Source" will show that this is not HTML—just plain text.

This is already an application in the sense that **code** is generating output, but it is not yet producing the kind of structured web pages we typically want.

11.4 Returning HTML with `render_template`

Flask's `render_template` function solves a common need: "Take an HTML file from my templates folder and return it as the HTTP response."

A typical `app.py` might import three frequently used pieces from Flask:

```
from flask import Flask, render_template, request
```

- `Flask` creates the app object.
- `render_template` loads an HTML template from `templates/` and returns it to the browser.
- `request` represents the incoming HTTP request (headers, parameters, method, and more).

If your project has:

- `app.py`
- `templates/index.html`

then you can do:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")
```

Now the browser receives a full HTML document (whatever is in `templates/index.html`), and "View Page Source" will show that HTML.

At this point you have still built something that could have been served statically, but you have placed yourself in a position to make it dynamic, because a Python function is now in control of the response.

11.5 Query Strings and `request.args`: Input via GET

One of the simplest ways a browser sends input to a server is the **query string**, appended to the URL after a `?`, with key–value pairs separated by `&`:

- `/greet?name=David`
- `/search?q=cats`
- `/search?q=cats&lang=en`

When the browser uses the HTTP method **GET**, these key–value pairs appear in the URL. Flask parses them for you and exposes them as a dictionary-like object:

- `request.args`

For example, if you want to greet the user by name, you might place a placeholder in your HTML template:

```
<!-- templates/index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>hello</title>
    <meta name="viewport" content="initial-scale=1, width=device-width">
  </head>
  <body>
    hello, {{ name }}
  </body>
</html>
```

The double curly brace syntax `{{ ... }}` is not Python and not HTML; it comes from a templating language called **Jinja**, which Flask uses for templates. In Jinja, `{{ name }}` means: "Insert the value of the variable `name` here."

In Python, you can obtain the query parameter and pass it into the template:

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/")
def index():
    name = request.args["name"]
    return render_template("index.html", name=name)
```

Now, visiting:

- `/?name=David`

will render:

- `hello, David`

11.5.1 Defensive coding: missing keys and HTTP 400

The line:

```
name = request.args["name"]
```

assumes that the `name` parameter exists. If it does not, the request can fail, and you may see an HTTP error such as **400** (a client-side error indicating a "bad request" situation for your application's expectations).

Two common strategies avoid this failure:

Strategy A: Explicit conditional logic

```
if "name" in request.args:
    name = request.args["name"]
else:
    name = "world"
```

Strategy B: `.get()` with a default Python dictionaries support `.get(key, default)`. Flask's `request.args` supports a similar pattern:

```
name = request.args.get("name", "world")
```

This yields `"world"` if the key does not exist.

11.5.2 Passing variables into templates

Flask's `render_template` accepts named parameters:

```
return render_template("index.html", name=name)
```

It is common to see `name=name`, which can look redundant, but it is simply "parameter name" on the left and "Python variable value" on the right.

11.6 Forms as a User Interface for Parameters

Typing query strings into the URL bar is not a realistic user interface. The web's standard input mechanism is an HTML **form**.

A simple greeting form might look like this:

```
<form action="/greet" method="get">
  <input name="name" type="text" placeholder="Name" autocomplete="off" autofocus>
```

```
  <button type="submit">Greet</button>
</form>
```

When the user submits this form, the browser navigates to:

- `/greet?name=David`

But this introduces a key design point: **if your HTML form's action points to `/greet`, then your Flask app must implement a `/greet` route**. Otherwise, the browser will show **404 Not Found**, because your server has no code associated with that path.

A common pattern is therefore:

- `/` displays a page containing the form (input)
- `/greet` processes input and displays results (output)

For example:

```
@app.route("/")
def index():
    return render_template("index.html")

@app.route("/greet")
def greet():
    name = request.args.get("name", "world")
    return render_template("greet.html", name=name)
```

Where `templates/greet.html` might contain:

```
hello, {{ name }}
```

11.7 Avoiding Duplication with Template Inheritance (`layout.html`)

If `index.html` and `greet.html` are both full HTML documents, they will likely share a large amount of boilerplate: doctype, `<html>`, `<head>`, `<title>`, meta viewport, and so on.

Duplicating that boilerplate is a design smell: if you change the title or add CSS, you must update multiple files.

Jinja supports **template inheritance**, commonly organized around a `layout.html` file that defines the shared structure and a named block where each page inserts custom content.

A conventional `templates/layout.html` might include:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>hello</title>
```

```
    <meta name="viewport" content="initial-scale=1, width=device-width">
  </head>
  <body>
    {% block body %}{% endblock %}
  </body>
</html>
```

Here, `{% block body %}{% endblock %}` is a Jinja construct that defines a placeholder for a *block* of HTML, not just a single variable.

Then `templates/greet.html` can "extend" the layout:

```
{% extends "layout.html" %}

{% block body %}
  hello, {{ name }}
{% endblock %}
```

And `templates/index.html` can do the same:

```
{% extends "layout.html" %}

{% block body %}
  <form action="/greet" method="get">
    <input name="name" placeholder="Name" autocomplete="off" autofocus>
    <button type="submit">Greet</button>
  </form>
{% endblock %}
```

This looks "uglier" at first because it introduces template syntax, but it scales far better: shared page structure lives in one place, while each page contributes only what is unique.

11.8 POST, Privacy, and `request.form`

Using GET has a property that is sometimes a feature and sometimes a problem:

- The URL becomes **stateful**: you can copy it, paste it, and someone else can reproduce the same page.
- But input appears in the URL, browser history, auto-complete, logs, and screenshots.

For non-sensitive inputs (like a search query), GET is often fine. For sensitive inputs (like passwords or credit card numbers), GET is inappropriate.

The alternative is **POST**, which sends form data in the HTTP request body rather than in the URL.

Changing:

```
<form method="get">
```

to:

```
<form method="post">
```

prevents the parameter from appearing in the URL. However, it also changes what your Flask route must support:

- If your route only accepts GET (the default), but the browser sends POST, the server will respond with **405 Method Not Allowed**.

11.8.1 Allowing POST with `methods=[...]`

Flask routes accept a `methods` parameter:

```
@app.route("/greet", methods=["POST"])
def greet():
    ...
```

If a route needs both behaviors—displaying a form (GET) and processing it (POST)—you can support both:

```
@app.route("/login", methods=["GET", "POST"])
def login():
    ...
```

11.8.2 Reading POST data with `request.form`

When using GET, form data is available in `request.args`. When using POST, Flask makes it available in `request.form`.

So a POST-based greeting handler might look like:

```
@app.route("/greet", methods=["POST"])
def greet():
    name = request.form.get("name", "world")
    return render_template("greet.html", name=name)
```

A subtle point arises here: `request.args` and `request.form` are both tied to "form submission" in the human sense, but Flask uses different properties depending on whether the browser used GET or POST.

11.9 Consolidating Related Functionality into One Route

A small app can afford separate routes for "show form" and "process form," but as applications grow, doubling the number of routes can become noisy.

A common alternative is to use a single route that behaves differently depending on the request method:

```
@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        name = request.form.get("name")
        return render_template("greet.html", name=name)

    return render_template("index.html")
```

Here:

- GET `/` renders the form.
- POST `/` processes the form submission.

If you make this change, you must also adjust your form's `action`. If the route is now `/`, then the form should submit to `/` (or omit `action` entirely, which causes the browser to submit to the current page's route).

11.10 A Template Bug: Empty Strings and Jinja Conditionals

Even when you provide a default value in `.get("name", "world")`, a form can still submit a key with an **empty value** (the empty string). In that case, the key exists, so your default is not used, and you might display:

- `hello,` (with nothing after the comma)

One approach is to handle this in the template using Jinja's control flow:

```
hello,
{% if name %}
  {{ name }}
{% else %}
  world
{% endif %}
```

In Jinja:

- `{% if ... %}` introduces logic.
- `endif` closes the block.

- A non-empty string is "truthy," while the empty string is treated as "falsey," which makes this conditional a natural way to choose between a provided name and a fallback.

This illustrates an important idea: templates are not only for substitution; they can contain lightweight logic such as conditionals (and, as we will see next, loops).

11.11 Case Study: Frosh IMs Registration (Forms, Validation, and Templates)

A simple "registration" web application is a useful way to synthesize HTML forms, Flask routes, validation, and templates.

11.11.1 The registration form: text input + select menu

A registration page might include:

- A name text field
- A sport selector
- A submit button

In HTML, a dropdown menu is a `<select>` element containing `<option>` elements:

```
<select name="sport">
  <option value="basketball">basketball</option>
  <option value="soccer">soccer</option>
  <option value="ultimate frisbee">ultimate frisbee</option>
</select>
```

A user-experience issue is that the first option is selected by default, which can cause accidental registrations (or bias users toward the first option).

One workaround is to add a placeholder-like option that is selected by default but cannot be chosen afterward:

```
<select name="sport">
  <option disabled selected value="">Sport</option>
  ...
</select>
```

This is essentially a "hack" to emulate placeholder behavior in a control that does not have a true `placeholder` attribute.

11.11.2 Client-side validation with `required` (and why it's not security)

HTML allows client-side validation, such as requiring a field:

```
<input name="name" required>
```

This improves user experience, but it is not a security feature. A user can open developer tools, remove `required`, and submit anyway. Therefore:

- Client-side checks are helpful for usability.
- **Server-side validation is mandatory** for correctness and security.

11.11.3 Implementing the `/register` route and handling mistakes

If a form uses `method="post"`, the corresponding Flask route must accept POST:

```
@app.route("/register", methods=["POST"])
def register():
    if not request.form.get("name"):
        return render_template("failure.html")

    if not request.form.get("sport"):
        return render_template("failure.html")

    return render_template("success.html")
```

During development, if you forget to return something on some path through your function, Flask will produce a server-side error (often shown in the browser as **500 Internal Server Error**) and the terminal running Flask will show a traceback explaining what went wrong (for example, that your function did not return a valid response).

This is a practical debugging reality of web development: the browser shows you *a status code*, while the terminal often shows you *the actual exception and message*.

11.11.4 Better error pages: passing messages into templates

Instead of returning a generic failure page, you can render an error template and pass a specific message:

```
return render_template("error.html", message="Missing name")
```

Then `error.html` can display:

```
<p>{{ message }}</p>
```

This pattern mirrors how real sites provide specific feedback when a form submission fails.

11.12 Never Trust the Client: Validating Allowed Sports Server-Side

Even if your HTML only offers three sports, a user can modify the form in developer tools and submit a different value (for example, “football”). If your server only checks that *some* value exists, bogus values will pass validation.

A robust approach is to maintain an authoritative list of allowed sports in Python and validate against it.

```
SPORTS = ["basketball", "soccer", "ultimate frisbee"]
```

Then validate:

```
sport = request.form.get("sport")
if sport not in SPORTS:
    return render_template("error.html", message="Invalid sport")
```

11.12.1 Eliminating duplication: generating the select options from the same list

If `SPORTS` is the authoritative list, it is better to generate the dropdown from that list rather than hard-code options in HTML.

In `app.py`:

```
@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)
```

In `index.html`:

```
<select name="sport">
  <option disabled selected value="">Sport</option>

  {% for sport in sports %}
    <option value="{{ sport }}">{{ sport }}</option>
  {% endfor %}
</select>
```

Now the same data structure controls:

- What the user can choose (UI)
- What the server will accept (validation)

This dramatically reduces the chance of inconsistencies.

11.13 Radio Buttons vs. Checkboxes (and Handling Multiple Values)

HTML offers multiple ways to represent choices:

- **Radio buttons** (`type="radio"`) are mutually exclusive: you can choose exactly one.
- **Checkboxes** (`type="checkbox"`) are inclusive: you can choose zero or more.

Using Jinja, you can generate a set of inputs from a list:

```
{% for sport in sports %}
  <input name="sport" type="radio" value="{{ sport }}"> {{ sport }}
{% endfor %}
```

Changing `radio` to `checkbox` allows multiple sports to be selected:

```
{% for sport in sports %}
  <input name="sport" type="checkbox" value="{{ sport }}"> {{ sport }}
{% endfor %}
```

However, when multiple checkboxes share the same name, the server receives **multiple values for the same key**. In Flask, `request.form.get("sport")` is no longer appropriate, because it returns only one value.

Instead, you must obtain a list of all submitted values. In Flask, this is done with `getlist`:

```
sports_selected = request.form.getlist("sport")
```

You can then validate each selected sport:

```
for sport in request.form.getlist("sport"):
    if sport not in SPORTS:
        return render_template("error.html", message="Invalid sport")
```

A practical lesson lives here as well: frameworks have their own method names and conventions, and it is normal to consult documentation when you need a detail such as "how do I retrieve multiple values for the same field?"

11.14 Redirects and HTTP 302: Sending the User to Another Route

Often, after processing form data, you do not want to render a page directly; you want to send the user elsewhere (for example, to a list of registrants). In HTTP terms, this is a **redirect**, commonly represented by status code **302 Found**, along with a `Location` header that tells the browser where to go next.

Flask provides a `redirect` function that handles this for you:

```
from flask import redirect


return redirect("/registrants")
```

In browser developer tools (Network tab), you can often observe:

1. The POST request to `/register` returns `302 Found` with `Location: /registrants`
2. The browser automatically performs a GET request to `/registrants` and receives `200 OK`

This is a standard, intentional two-step sequence that appears throughout the web.

11.15 Storing Data in Memory vs. Persisting Data with SQL

If you store registrants in a global Python dictionary, you can generate a "registrants" page by iterating over it in a template. But storing important data in memory has serious limitations:

- If the server restarts, the dictionary is cleared.
- If you run multiple server instances, each has its own memory, and users may see inconsistent data.

To keep data across restarts and to support real scale, you persist it to disk, often with a database such as **SQLite**, using **SQL**.

11.15.1 A global dictionary approach (in-memory "model")

A dictionary model might look like:

- keys: user names
- values: sports

```
registrants = {}
registrants[name] = sport
```

Then a template can iterate and display a table of names and sports.

This works for demonstrations, but it is not durable.

11.15.2 A SQLite approach (persistent "model")

Using the CS50 `SQL` library in Python, you can connect to a database:

```
from cs50 import SQL

db = SQL("sqlite:///froshims.db")
```

A database might contain a table like:

- `id` (primary key)
- `name`
- `sport`

And registration becomes an insert:

```
db.execute(
    "INSERT INTO registrants (name, sport) VALUES (?, ?)",
    name,
    sport
)
```

Notice the use of `?` placeholders. This is the same security principle emphasized earlier with SQL injection: you should not build SQL statements by concatenating user input into strings. Placeholders ensure proper escaping.

11.15.3 Primary keys and hidden inputs (deregistering)

When you need to delete a row (for example, deregister a registrant), names are not reliable identifiers. Multiple users can share a name, so you rely on the database's unique identifier: the primary key `id`.

A template can include a small form per row with a hidden input that carries the row's ID:

```
<form action="/deregister" method="post">
  <input name="id" type="hidden" value="{{ registrant.id }}">
  <button type="submit">Deregister</button>
</form>
```

Then the Flask route can delete by ID:

```
@app.route("/deregister", methods=["POST"])
def deregister():
    id = request.form.get("id")
    if id:
        db.execute("DELETE FROM registrants WHERE id = ?", id)
    return redirect("/registrants")
```

This "hidden input carrying an opaque identifier" is one of the most common patterns in web applications, because it connects a user interface (names, titles, labels) to a backend that needs a reliable key.

11.16 MVC: Model, View, Controller

A useful vocabulary for many web applications is **MVC**:

- **Model**: where data is stored (a Python dictionary in memory, or a SQL database, or both)
- **View**: what the user sees (templates, HTML, CSS, JavaScript)
- **Controller**: the application logic that connects input to output (routes and Python code in `app.py`)

You do not need to force every application into MVC, but the pattern describes a large fraction of real systems and provides a shared language for discussing design.

11.17 Sessions and Cookies: How Sites "Remember" You

HTTP is fundamentally *stateless*: each request can be processed independently. Yet websites "remember" that you are logged in, remember what is in your shopping cart, and keep track of preferences.

The mechanism that enables this is the **cookie**.

11.17.1 The `Set-Cookie` and `Cookie` headers

After successful login, a server can include a header like:

- `Set-Cookie: session=...`

This instructs the browser to store a key–value pair (a "hand stamp"). On subsequent requests, the browser sends:

- `Cookie: session=...`

so the server can recognize that this browser has an existing session.

A cookie value is typically a large random identifier; it generally does **not** contain the user's password, and it often does not contain the username either. Instead, the cookie acts as a token that allows the server to look up the user's state (login status, shopping cart contents) in server-side storage.

Cookies have privacy implications because they can also be used for tracking, but they are also the standard building block for stateful web applications.

11.18 Using Sessions in Flask (`flask_session`)

Flask provides session support, and a commonly used extension is `flask_session`, which helps store session data server-side.

A typical configuration includes:

```
from flask import Flask, render_template, request, redirect, session
from flask_session import Session

app = Flask(__name__)

app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
```

This configuration aims to:

- treat the session as not permanent (often cleared when the browser closes), and

- store session contents on the server's filesystem rather than inside the cookie itself.

11.18.1 A minimal login system (name only)

A login route commonly supports:

- GET: display the login form
- POST: process the login form

```
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        session["name"] = request.form.get("name")
        return redirect("/")
    return render_template("login.html")
```

Then the homepage can check whether a name exists in the session:

```
@app.route("/")
def index():
    name = session.get("name")
    return render_template("index.html", name=name)
```

And the template can conditionally display different content:

```
{% if name %}
  You are logged in as {{ name }}.
  <a href="/logout">Log out</a>
{% else %}
  You are not logged in.
  <a href="/login">Log in</a>
{% endif %}
```

11.18.2 Logging out by clearing the session

To log out, you clear the session and redirect:

```
@app.route("/logout")
def logout():
    session.clear()
    return redirect("/")
```

This is the same basic idea used by many real websites: logging out does not "undo" HTTP, but it invalidates or clears the state associated with the cookie's session identifier.

11.19 A Shopping Cart: Sessions + Lists + Databases

A shopping cart is just a specific kind of session state: a per-user collection of items.

A typical pattern is:

- Store a list of item IDs in `session["cart"]`
- When displaying the cart, query the database for those IDs
- Render them in a template

For example, when adding an item:

```
if "cart" not in session:
    session["cart"] = []

id = request.form.get("id")
if id:
    session["cart"].append(id)
```

When displaying the cart, you might select all books whose IDs are in that list. Some libraries support passing a list as a placeholder so it can expand to a comma-separated list safely, rather than constructing SQL manually.

This demonstrates a broader theme: once you understand forms, routes, templates, sessions, and SQL, you can recognize many "big" applications as combinations of these small ideas.

11.20 Search and SQL Wildcards: Building an IMDb-like Query

A search feature often begins with:

- a form that sends `q=...`
- a route that queries a database
- a template that displays matching rows

A basic equality search might be too strict:

```
SELECT * FROM shows WHERE title = ?
```

It will only match exact titles.

To support partial matching, SQL provides `LIKE` with wildcards, typically `%`:

- `%office%` matches any title containing `office` anywhere.

To do this safely, you still use placeholders, but you build the wildcard string in Python:

```
q = request.args.get("q", "")
shows = db.execute(
    "SELECT * FROM shows WHERE title LIKE ?",
```

```
    "%" + q + "%"
)
```

This preserves the security benefit of placeholders while still enabling wildcard search.

11.21 AJAX, `fetch`, and APIs: Updating a Page Without Reloading

Traditional web apps often work by:

1. user submits a form
2. browser navigates to a new page
3. server generates and returns a full HTML document

Modern apps increasingly update parts of the page without full reloads. A classic term for this approach is **AJAX** (originally "Asynchronous JavaScript And XML"), though today it usually just means "JavaScript makes background HTTP requests."

11.21.1 A dynamic search box that updates as you type

Instead of a form with a submit button, you can use:

- an `<input>` box
- an empty `<ul>` list
- JavaScript that listens to typing and fetches results

A conceptual flow is:

1. Listen for an input event
2. Request `/search?q=...`
3. Receive response data
4. Insert response into the page (DOM manipulation)

A simplified pattern looks like:

```
let input = document.querySelector("input");

input.addEventListener("input", async function () {
  let response = await fetch("/search?q=" + input.value);
  let text = await response.text();
  document.querySelector("ul").innerHTML = text;
});
```

If the server returns HTML fragments (like a set of `<li>` elements), `innerHTML` can drop them into the existing `<ul>`.

Browser developer tools (Network tab) reveal what is happening: each keystroke can trigger an HTTP request, even though the URL in the address bar does not change and the whole page does

not reload.

11.21.2 JSON as a more standard API format

Returning HTML fragments works, but it is not the most standardized way to exchange data between programs. A more common modern approach is to return **JSON** (JavaScript Object Notation), which resembles Python lists and dictionaries but follows strict rules such as using double quotes for strings.

Flask can convert Python data structures to JSON with `jsonify`:

```
from flask import jsonify

@app.route("/search")
def search():
    q = request.args.get("q", "")
    shows = db.execute(
        "SELECT * FROM shows WHERE title LIKE ?",
        "%" + q + "%"
    )
    return jsonify(shows)
```

Now the server's response is structured data rather than HTML. A front end can then generate its own HTML from that JSON, or use it in other ways. This separation—where the backend provides an API and the frontend consumes it—is increasingly common in large applications, and it is also the conceptual bridge to using third-party APIs (where the server you are querying belongs to someone else).

11.22 Summary

Flask provides a practical way to synthesize many of the web concepts introduced earlier—HTTP methods, routes, forms, templates, databases, and JavaScript—into complete applications.

Key ideas from this chapter include:

- A **route** is a path like `/greet` or `/search` that maps to code, not necessarily a file.
- `render_template` returns HTML from `templates/`, enabling dynamic generation of web pages.
- `request.args` contains GET parameters (query string); `request.form` contains POSTed form data.
- GET is **visible in the URL** and therefore stateful and shareable, but inappropriate for sensitive input; POST hides data from the URL.
- Flask routes must explicitly support POST using `methods=[...]`, otherwise you will see **405 Method Not Allowed**.

- Jinja supports:
  - variable interpolation with `{{ ... }}`
  - control flow with `{% if %}`, `{% for %}`, and their corresponding endings
  - template inheritance with `layout.html`, `{% block %}`, and `{% extends %}`
- Client-side validation (like `required`) improves user experience but is not security; **server-side validation is essential**.
- **Redirects** are commonly implemented with HTTP **302** and a `Location` header; Flask's `redirect` automates this.
- Persisting data with **SQL** and a database (instead of Python globals) avoids losing data when the server restarts.
- Primary keys and hidden form fields (`<input type="hidden">`) are a standard way to connect UI elements to database rows.
- **Sessions** and **cookies** implement "memory" across requests; Flask's session support makes per-user state (logins, carts) tractable.
- Modern "no reload" interfaces use JavaScript `fetch` (AJAX-style) to call server endpoints repeatedly and update the DOM dynamically.
- APIs often return **JSON**, and Flask's `jsonify` helps turn Python data into that standardized format.

With these pieces, you can recognize the structure underneath many familiar systems—from registration sites to shopping carts, from login flows to search engines—and you can begin building your own applications that combine Python logic, HTML templates, SQL persistence, and responsive browser-based interfaces.

Chapter 12: Cybersecurity — Passwords, Hashing, Encryption, and Practical Defenses

Over the past chapters, we have built up a programmer's mental model of how computers represent information, how programs manipulate that information, and how networks move that information between machines. The natural "next" question, once you can build systems that work, is how to build systems that continue to work **even when someone is trying to make them fail**.

That is the domain of **cybersecurity**: the practices and mechanisms by which we try to keep secure our systems, our data, our accounts, and the infrastructure that increasingly surrounds daily life—devices on our desks and in our pockets, as well as the cloud services we depend on.

Cybersecurity can feel intimidating because it is not just "more programming." It is adversarial by nature: you are designing not only for correct users and expected inputs, but also for mistakes, manipulation, and deliberately malicious behavior. And, importantly, cybersecurity is full of trade-offs. You can nearly always make something *more* secure, but doing so tends to cost you something else: convenience, time, money, compatibility, or recoverability when things go wrong.

This chapter builds a practical security vocabulary and a set of mental models that help you reason

about threats more methodically, including when it makes sense to trust a mechanism and when it is safer to assume it can fail.

12.1 What Does It Mean for Something to Be "Secure"?

A useful, grounded definition of "secure" is not "unbreakable," but **resistant to attack**.

That definition immediately implies a reality check: resistance is a spectrum, not a binary. A system might be "secure enough" against casual attackers, but not against a determined attacker with time, resources, and motivation.

Another complementary way to describe security is: **you control who has access**. In practice, that control usually comes from two related concepts:

- **Authentication**: proving *who you are* (for example, logging in).
- **Authorization**: deciding *what you are allowed to do* once authenticated (for example, read email vs. change account settings).

Many real systems begin with an authentication mechanism that is still surprisingly primitive: a password. So we begin there—not because passwords are ideal, but because they are everywhere, and understanding their weaknesses leads naturally to better designs.

12.2 Passwords as a Security Mechanism (and Why Humans Struggle)

Security researchers routinely analyze leaked data from real-world breaches. When attackers compromise a service and the service's user database is leaked (often posted publicly or sold), researchers can learn how people actually behave.

A recurring lesson is that, given a choice, many humans choose passwords that are easy to type, easy to remember, and therefore easy to guess.

A representative "top 10" list of common (and therefore terrible) passwords includes examples such as:

- `123456`
- `admin`
- `12345678`
- `123456789`
- `1234`
- `12345`
- `password`
- `123`
- `Aa123456`
- `1234567890`

These examples are not just amusing—they are diagnostic. You can often infer password policies from them. If `123` appears, then some systems are allowing passwords of length 3. If `12345678` appears, many systems likely require at least 8 characters, and users respond by using the simplest 8-character sequence they can think of.

Farther down such lists you will also see patterns that *feel* clever but are predictable once attackers know to look for them:

- `Iloveyou` (memorable, but common)
- `qwertyuiop` (the top row of a US keyboard)
- `p@ssw0rd` (substitutions like `@` for `a` and `0` for `o`)

The key security lesson is that **if you can think of it, an adversary can think of it too**, and modern attackers do not guess manually. They automate.

12.3 Two Common Password Attacks: Dictionary Attacks and Password Stuffing

12.3.1 Dictionary attacks

A **dictionary attack** is exactly what it sounds like: instead of trying every possible password, an attacker tries the most likely passwords first.

A "dictionary" here does not have to be an English dictionary; it can be:

- a list of the top 10 passwords,
- a list of the top 100,000 passwords from past breaches,
- or combinations of common patterns (like `Password1!`, `Summer2026`, and so on).

This is why a password that is "hard to guess by a friend" can still be "easy to guess by software."

12.3.2 Password stuffing

Password stuffing is the attack that turns one breach into many.

If a service is compromised and your username/password is exposed, an attacker can try the same combination on other services (email, banking, social media), betting—often correctly—that many users reuse passwords.

This creates an uncomfortable chain: even if one website is sloppy, the consequences can spill into accounts on well-designed websites, simply because users reused the same secret.

12.4 Brute Force and the Mathematics of "Search Space"

When dictionary attacks fail, attackers can attempt a **brute force attack**.

A brute force attack tries *all possible combinations* in a search space until one works. The key question is not whether brute force is possible in principle, but whether it is feasible in practice under real constraints like time, lockouts, and detection.

12.4.1 Four-digit passcodes: 10,000 possibilities

Consider the traditional phone passcode: 4 digits.

Each position has 10 possibilities (`0` through `9`), so the total number of passcodes is:

- $10 \times 10 \times 10 \times 10 = 10^4 = 10{,}000$

That sounds large until you remember that computers can try thousands or millions of possibilities quickly, especially if they can automate the entry (for example, by connecting a device via a cable and simulating input).

12.4.2 A simple brute-force program in Python

A brute force program does not need to be complicated. Conceptually, it is just "loop over all candidate passcodes."

One Pythonic way to generate combinations is to use `itertools.product`, which produces the Cartesian product of a set of characters repeated `n` times.

```
from string import digits
from itertools import product

for passcode in product(digits, repeat=4):
    print(passcode)
```

In a real attack, you would not `print` the passcode; you would send it to the target system and test whether it unlocks.

The alarming point is not the exact code, but the fact that **very little code is required** to automate a brute force attempt.

12.4.3 Four letters: ~7.3 million possibilities

If instead you use letters, the search space grows.

If you use lowercase + uppercase letters, that is 52 characters total (`a`–`z` plus `A`–`Z`), giving:

- $52^4 = 7{,}311{,}616$ possibilities (about 7 million)

That is far larger than 10,000, but still small enough that a modern computer can iterate quickly if there is no rate limiting.

In Python, this change is as small as switching from digits to ASCII letters:

```
from string import ascii_letters
from itertools import product

for passcode in product(ascii_letters, repeat=4):
    print(passcode)
```

12.4.4 Adding punctuation and increasing length: why length matters

Websites often require "complexity," such as mixing uppercase, lowercase, digits, and punctuation. A typical set might include:

- 26 lowercase letters
- 26 uppercase letters
- 10 digits
- ~32 punctuation symbols

That totals roughly **94 characters**.

If you choose **8 characters** from a set of 94, the search space becomes:

- $94^8 \approx 6{,}095{,}689{,}385{,}410{,}816$

That is on the order of **6 quadrillion** possibilities.

At that point, brute force changes from "fast" to "prohibitively slow," assuming the attacker must truly test guesses one by one. This is one reason that, from a brute-force perspective, **length is often more important than cleverness**.

12.5 Defending Against Brute Force: Rate Limits, Lockouts, and Their Tradeoffs

If brute force is feasible when guesses are cheap, the practical defense is to make guesses expensive.

A common device behavior after too many failed attempts is a lockout like:

> "Try again in 1 minute."

This does not make brute force impossible, but it changes the economics dramatically:

- It increases **time cost** (seconds become minutes, hours, or days).
- It increases **risk cost** (the attacker must remain present longer, increasing the chance of being noticed).
- It can increase **detection likelihood** (systems may alert the user or administrators).

Often, lockouts become increasingly severe (sometimes exponentially): 1 minute, then 2, then 5, then 1 hour, and so on. Some environments go further: after enough failed attempts, a device can **wipe itself** (a "self-destruct" policy), which is sometimes used on corporate-managed devices to protect sensitive data.

The downside of lockouts

Security features typically impose costs on legitimate users too.

If you forget your passcode, lockouts punish you as well. And if a device wipes itself after too many mistakes, that protects data from attackers—but it can also cause irreversible loss if the legitimate owner (or a curious child, roommate, or sibling) triggers the wipe accidentally.

This is a recurring cybersecurity theme:

- **More security** often means **less convenience** or **less recoverability**.

12.6 The Practical Reality: Security Is Not Absolute

It is tempting to want to say, "my device is secure" or "my website is secure," but it is more accurate to think in terms of probabilities and costs.

Two reasons make absolute security a poor mental model:

1. **An attacker with enough time, motivation, and resources can often break into most systems.**
2. **Defenders must be perfect; attackers only need one mistake.**
   In the physical world, you must lock every door and window; an attacker only needs one open window. In the digital world, a system might be well-designed except for one weak password, one missing validation check, or one unpatched vulnerability.

A more realistic goal is therefore to build a **gauntlet of defenses**:

- prevent easy attacks,
- slow down harder attacks,
- detect suspicious activity,
- and minimize the damage if an attacker succeeds.

This is why services often notify you of unusual activity, such as a login from a new city or a new device.

12.7 Password Managers: A Practical Improvement (with a Real Tradeoff)

Because humans are bad at choosing and remembering many strong passwords, a modern solution is a **password manager**.

A password manager is software that:

- generates long, random passwords for you,
- stores them securely on your device (or in a synced vault),
- and auto-fills them when you revisit a site.

In this model, you might not even know your password for most sites—because you do not need to.

The downside: one "master password" risk

A password manager creates a concentration of risk:

- All your passwords are protected by one **primary** (master) password.
- If an attacker learns that master password, they can potentially access many accounts.
- If you forget that master password, you might lock yourself out of many accounts at once.

That said, this tradeoff is often worth it. Compared with reusing weak passwords across many sites, using a password manager typically reduces overall risk, because it prevents password reuse and makes each site's password high-entropy and unique.

A note of caution

Password managers are software, and software can have bugs. You are trusting the manager's design and implementation. This is not a reason to avoid them entirely, but it is a reason to:

- choose reputable, well-reviewed tools,
- keep your devices updated,
- and use a strong master password.

Many operating systems now include password-management features directly, which can reduce the need for third-party tools.

12.8 Two-Factor and Multi-Factor Authentication (2FA / MFA)

If passwords are "something you know," then **two-factor authentication** adds a second kind of proof.

A **factor** is a category of evidence:

- **Something you know** (password, PIN)
- **Something you have** (phone, hardware token, USB security key)
- **Something you are** (fingerprint, face, retina—biometrics)

With 2FA, logging in typically requires:

1. your password, and
2. a second factor such as a one-time code sent to your phone, generated by an app, or produced by a physical device.

This reduces risk because an attacker who learns your password still needs access to your phone or token.

One-time passcodes and replay resistance

2FA codes are often **one-time passcodes** that expire quickly. Even if an attacker intercepts a code, it might be unusable after it has been used once or after the time window closes, which reduces the value of intercepted information.

The downside of 2FA

The most common downside is straightforward:

- **You can lose access to the second factor.**

If your phone is lost, dead, stolen, or unavailable while traveling, 2FA can lock you out unless you have backup codes or a recovery process.

Again, security improves—but at the cost of potential inconvenience.

12.9 How Websites *Should* Store Passwords: Hashing (Not Plaintext)

When you click "Forgot password?" on a well-designed site, the site should *not* email you your old password.

If a site can send you your password, it means the site either:

- stored your password in plaintext (in the clear), or
- stored it in a reversible form that it can decrypt.

Both are dangerous designs, because if the database is compromised, attackers gain immediate access to user passwords.

12.9.1 Hash functions: one-way by design

A **hash function** takes an input (like a password) and produces an output (a hash value). A core security property is that it is intended to be **one-way**:

- easy to compute the hash from the password,
- infeasible to recover the password from the hash.

This is not encryption. With encryption, you must be able to reverse the process (decrypt). With hashing, reversal is not supposed to be possible.

A secure password-storage design therefore stores, conceptually, something like:

- username → hashed_password

So instead of storing:

- `alice` → `apple`

- `bob` → `banana`

the system stores:

- `alice` → `hash(apple)`
- `bob` → `hash(banana)`

When Alice logs in and types `apple`, the server hashes `apple` again and compares the result with what is stored. If the hash matches, the password is considered correct—without the server ever storing the original password.

12.9.2 Rainbow tables: "precomputing" hashes

A **rainbow table** is a large precomputed mapping of:

- candidate_password → hash(candidate_password)

If attackers have a rainbow table for common passwords, they can look up a stolen hash and find the original password *when that password was common enough to be in the table.*

Rainbow tables are most effective when the password search space is small, such as:

- 4-digit PINs (10,000 possibilities),
- short, common words,
- or predictable patterns.

They are much less practical when passwords are long and random because the table becomes astronomically large (for example, on the order of quadrillions of rows for 8 characters from a broad character set).

12.10 Salting: Preventing Identical Passwords from Producing Identical Hashes

Even if a system stores only hashes, there is still a subtle information leak:

- If two users have the same password, they will have the same hash.
- An attacker who sees duplicate hashes learns that some users share passwords.
- If the attacker cracks one of those hashes, they have effectively cracked all identical ones.

To mitigate this, systems use **salting**.

A **salt** is a random value added to the hashing process so that two identical passwords produce different hashes.

Conceptually, instead of hashing:

- `hash(password)`

the system hashes:

- `hash(password + salt)`
  (or, more generally, hashes both inputs together according to a defined scheme)

Each user gets a different salt, and that salt is stored alongside the hash so the server can use it again during login verification.

The result is that even if Carol and Charlie both choose the password `cherry`, their stored values differ because their salts differ. This prevents attackers from learning "which users share passwords" just by scanning the database.

12.11 How Password Resets Work When Passwords Aren't Stored

If a site stores only hashed (and salted) passwords, then when you forget your password, the site cannot "remind you" what it is. It genuinely does not know.

Instead, password resets typically work by issuing a **temporary, unique reset link**, often containing a long random token.

A typical flow is:

1. You request a password reset.
2. The server generates a random token and stores it (often with an expiration time).
3. The server emails you a link containing that token.
4. When you click the link, the server verifies the token and allows you to set a new password.

This is similar in spirit to one-time passcodes: the reset token is difficult to guess and short-lived, which makes it unlikely that an attacker can successfully "guess" a valid reset URL.

A practical takeaway is that if a service ever sends you your existing password, that is a strong warning sign that it is not using modern password-storage practices.

12.12 Cryptography Revisited: Symmetric vs. Asymmetric Encryption

Hashing helps store passwords safely, but cybersecurity problems often involve protecting data **in transit**, not just at rest.

That brings us back to **cryptography**, the art and science of scrambling information in a *reversible* way.

12.12.1 Symmetric (secret-key) cryptography

In **symmetric cryptography**, the sender and receiver share the same secret key.

- You encrypt with the key.
- You decrypt with the key (or with an equivalent form of it).

This is efficient and fast, but it has a bootstrapping problem:

- How do two parties agree on the secret key in the first place, especially if they have never communicated before?

If you have to whisper the key over an insecure channel, you have not gained much.

12.12.2 Asymmetric (public-key) cryptography

Asymmetric cryptography (also called **public-key cryptography**) solves this "chicken-and-egg" problem by using a **key pair**:

- a **public key** that can be shared openly,
- a **private key** that is kept secret.

The typical model is:

1. Someone publishes their public key.
2. Anyone can use that public key to encrypt a message to them.
3. Only the corresponding private key can decrypt that message.

The security comes from mathematics: the keys are chosen such that knowing the public key does not practically reveal the private key, because doing so would require infeasible computation (effectively a brute force effort so large it would take far longer than a human timescale).

This is the core primitive behind how secure connections like HTTPS begin: public-key cryptography helps establish initial trust and shared secrets, after which systems often switch to symmetric cryptography for speed.

You may encounter algorithm names such as RSA, Diffie–Hellman, or elliptic curve cryptography; these are different mathematical constructions for achieving the same broad goal: secure communication without having pre-shared secrets.

12.13 Passkeys and "Passwordless Login" (Digital Signatures in Practice)

A newer, increasingly common alternative to passwords is the **passkey**, often described in user interfaces as **passwordless login**.

The high-level idea is:

- Your device generates a public/private key pair for a specific website.
- The website stores your public key.
- Your device keeps the private key secret.
- Later, you log in by proving you have the private key—without typing a password.

12.13.1 Challenge–response and digital signatures

A common mechanism is a **challenge–response** protocol using **digital signatures**:

1. The website sends a random challenge (a random number or string).
2. Your device uses your private key to produce a signature over that challenge.
3. The website verifies the signature using your public key.

This reverses the direction from "public key encrypts, private key decrypts" into the related but distinct notion of "private key signs, public key verifies."

12.13.2 The upside and the new risk

Passkeys reduce reliance on human-memorable secrets. That can dramatically reduce phishing and password reuse problems, because there is no password to steal or reuse.

However, passkeys shift trust toward your devices:

- you must protect the devices where your passkeys live,
- and you need account recovery plans if you lose a device.

12.14 End-to-End Encryption: Protecting Data *Even From the Middle*

Not all encryption provides the same privacy guarantees.

A key distinction is whether encryption is:

- **client-to-server** (protects data in transit to a service), or
- **end-to-end** (protects data so that only the endpoints can read it).

12.14.1 Why HTTPS is not necessarily end-to-end

When you use HTTPS to access a service like webmail, your traffic is encrypted between you and the service's servers. That protects you from eavesdroppers on the network.

But the service provider itself can still access the plaintext once it reaches the server, because the provider terminates the encrypted connection and processes your data.

12.14.2 End-to-end encryption (E2EE)

End-to-end encryption means that if Alice sends a message to Bob, and it passes through an intermediary (a server), the intermediary still cannot read it. Only Alice and Bob have the keys needed to decrypt.

Many modern messaging services advertise end-to-end encryption, and you can increasingly find it in areas like video conferencing too.

12.14.3 A practical example: encryption settings in conferencing

Some systems offer multiple "encryption" options, where the more marketable-sounding option is not necessarily the strongest.

For example, you might see options like:

- "enhanced encryption" (protects data between you and the provider), versus
- "end-to-end encryption" (protects data so the provider cannot see it)

Turning on end-to-end encryption can have concrete feature costs. For instance, if the provider cannot see your meeting, it cannot produce server-side features like cloud recordings. You may still be able to record locally, but the provider cannot record what it cannot decrypt.

This illustrates again a central security tradeoff:

- **more privacy** often means **fewer convenience features**.

12.15 Deleting Files Isn't What It Seems: Secure Deletion and Full-Disk Encryption

When you delete a file on many systems, even if you empty the trash or recycle bin, the data may not be physically erased immediately.

Often, the operating system simply marks the file's storage blocks as "available," meaning:

- the system forgets where the file is,
- but the underlying bits can remain until overwritten by new data.

This is why forensic tools can sometimes recover "deleted" files: the data is still present on the storage medium.

12.15.1 Secure deletion (and why it's difficult)

A classic approach to **secure deletion** is overwriting:

- replace the file's blocks with zeros,
- ones,
- or random data,

so the old content cannot be recovered.

However, modern storage (especially SSDs) complicates this. Devices can remap failing areas transparently, which means an overwrite request may not overwrite the exact physical cells you think it is overwriting. As a result, remnants can persist.

At the extreme end, physical destruction of a device can be effective—but it is expensive and impractical as a routine deletion strategy.

12.15.2 Full-disk encryption as a practical solution

A widely recommended solution is **full-disk encryption**.

With full-disk encryption enabled, the entire drive's contents are stored in encrypted form. Without the decryption key (usually derived from your login credentials and protected by hardware), the disk appears as random bits.

This provides strong protection if:

- a laptop is stolen,
- the disk is removed and connected to another machine,
- or someone gains physical access to the storage.

Operating systems provide this under names such as:

- **FileVault** (macOS)
- **BitLocker** (Windows)

Full-disk encryption is especially valuable because it reduces the need to perfectly "securely delete" each file; if the disk is encrypted, remnants of deleted files are far less useful to an attacker without the key.

The downside

The downside is severe but clear:

- If you forget your disk encryption credentials and have no recovery key, you may permanently lose your data.

This is not a "bug" in encryption; it is a consequence of encryption working correctly.

12.15.3 Ransomware: encryption used offensively

Encryption is a powerful tool, and attackers can use it too.

In **ransomware** attacks, an attacker gains access to an organization's systems (hospitals, municipalities, businesses) and encrypts critical data with a key the attacker controls. The attacker then demands payment—often in cryptocurrency—to provide the decryption key.

This is a reminder that cybersecurity is not about "encryption is good" versus "encryption is bad," but about who controls keys, how systems are defended, and how organizations plan for recovery.

12.16 A Small Check for Understanding (Cybersecurity Concepts)

As a quick recap of the mechanisms discussed above, consider these representative questions and their intended answers:

- **Best way to create a password:** have a password manager generate it for you.
- **Downside of two-factor authentication:** you might lose access to the second factor.
- **What you see on an encrypted disk:** a random-looking sequence of zeros and ones.
- **Most secure encryption type among common options:** end-to-end encryption.
- **When it makes sense to store your password on a sticky note by your computer:** never.

These are not trivia; they are practical rules of thumb that translate directly into safer system design and safer personal security habits.

12.17 Practical Takeaways: A Minimal "Action List"

Cybersecurity improvements are easiest to adopt incrementally. A realistic approach is to prioritize accounts and devices that matter most: financial, medical, and highly personal accounts.

Three concrete actions have outsized benefit:

1. **Use a password manager (or passkeys where available)**, at least for your most sensitive accounts, so passwords are long, random, and unique.
2. **Enable two-factor authentication** anywhere it is offered, especially on email and financial accounts.
3. **Prefer end-to-end encryption** when discussing sensitive information, recognizing that you might lose some convenience features.

A fourth action is often worthwhile as well:

- **Enable full-disk encryption** (or confirm it is enabled), and build habits that reduce the window of opportunity for physical access—such as locking your screen when stepping away.

12.18 Looking Ahead: Tools, Communities, and Continuing Learning

As you transition from a guided environment to your own projects, it helps to know what to install, where to host projects, and where to continue learning.

12.18.1 Building on your own machine

To recreate a modern development workflow on your own laptop, common steps include:

- using a terminal on macOS or Windows,
- learning Git for version control (saving multiple versions of your code systematically),
- installing Visual Studio Code (VS Code) locally if you prefer not to rely on a cloud IDE.

Git, in particular, is widely used in industry for collaboration and tracking changes over time.

12.18.2 Hosting: static vs. dynamic

If you want to publish a static portfolio site, common free or low-cost hosting options include:

- **GitHub Pages**
- **Netlify** (with a free tier)

For dynamic web applications (like Flask apps), options include large cloud providers (such as AWS, Azure, and Google Cloud) as well as platforms oriented toward deploying web apps.

If you are eligible as a student, programs like GitHub's student/education offerings can provide credits or discounts for developer tools and hosting.

12.18.3 Staying current

Technology evolves quickly, and communities can help you track what changes and what remains foundational. Many developers learn continuously by reading technical news and discussions, asking questions in online forums, and increasingly using AI tools (carefully and critically) for reference and productivity.

And, if you want to go deeper into specific areas—Python, SQL, web development, AI, game development, or cybersecurity itself—there are many structured, free learning resources available online.

12.19 Summary: A Security Mindset

Cybersecurity is best approached as a mindset rather than a checklist.

- Think in terms of **probabilities and costs**, not absolutes.
- Expect tradeoffs: security competes with convenience and recoverability.
- Build **layers of defense**: prevent, slow down, detect, and limit damage.
- Prefer mechanisms that reduce dependence on human memory and behavior (password managers, passkeys, 2FA).
- Use encryption thoughtfully, including end-to-end encryption where appropriate and full-disk encryption to protect devices at rest.

As you continue building software—whether web applications, scripts, or larger systems—these concepts become not just "security topics," but everyday design constraints that shape how trustworthy your systems are in the real world.