1. Computers, programs and C

Herman Kamper

Every app you open, every search you type and every video you stream runs on software: lists of instructions written by programmers and executed by a machine. This lecture starts from scratch: what a computer actually is, how those instructions are represented, and what your first program looks like.

What is a computer?

Try to define a computer in your head before reading further. Most people's first answer is something about inputs and outputs, and that is right, but it is not quite enough. Consider a bicycle: it has inputs (your pedalling) and outputs (it moves forward), but nobody would call a bicycle a computer. So what else is missing?

The missing pieces are *processing* and *memory*. A computer takes inputs, manipulates them through some processing step, and produces outputs. To do that it needs to store the inputs and intermediate results somewhere: it needs memory. These four components – input, output, processing and memory – give us a working definition that holds up across a wide range of devices.

Memory is worth splitting into two kinds. *Short-term memory*, often called random access memory (RAM) or working memory, holds data while the processor works on it; switch the power off and it is gone. *Long-term storage* persists even after you switch off. A hard drive or solid-state drive (SSD) is long-term storage: it retains data whether or not the machine is running.

Babbage and Lovelace: the first computer and the first program

One of the earliest devices to fit this four-component definition was not electronic at all. In the 1800s, the English engineer Charles Babbage designed a series of mechanical computing machines including the Difference Engine and later the Analytical Engine. His Difference Engine used interlocking decimal gears: you set the gears to an initial configuration, cranked a lever, and the machine worked through a calculation and produced the answer. Babbage never completed his most ambitious design, the Analytical Engine, but even his earlier machines are recognisably computers. The initial gear settings are the input, the result is the output, the gears turning is the processing, and the internal gear positions at any moment constitute the memory. It was around this work that one of the most remarkable figures in computing history became involved: Ada Lovelace.

Figure 1: Ada Lovelace, who wrote what is widely regarded as the first computer program, an algorithm for Babbage's Analytical Engine. Image from Wikimedia Commons.

The Analytical Engine was designed to be programmable via punched cards: make holes in a physical card, feed it into the machine, and the machine follows the encoded instructions. Ada Lovelace worked with Babbage on this engine and wrote detailed notes that included an algorithm for computing Bernoulli numbers. That algorithm is widely described as the first computer program.

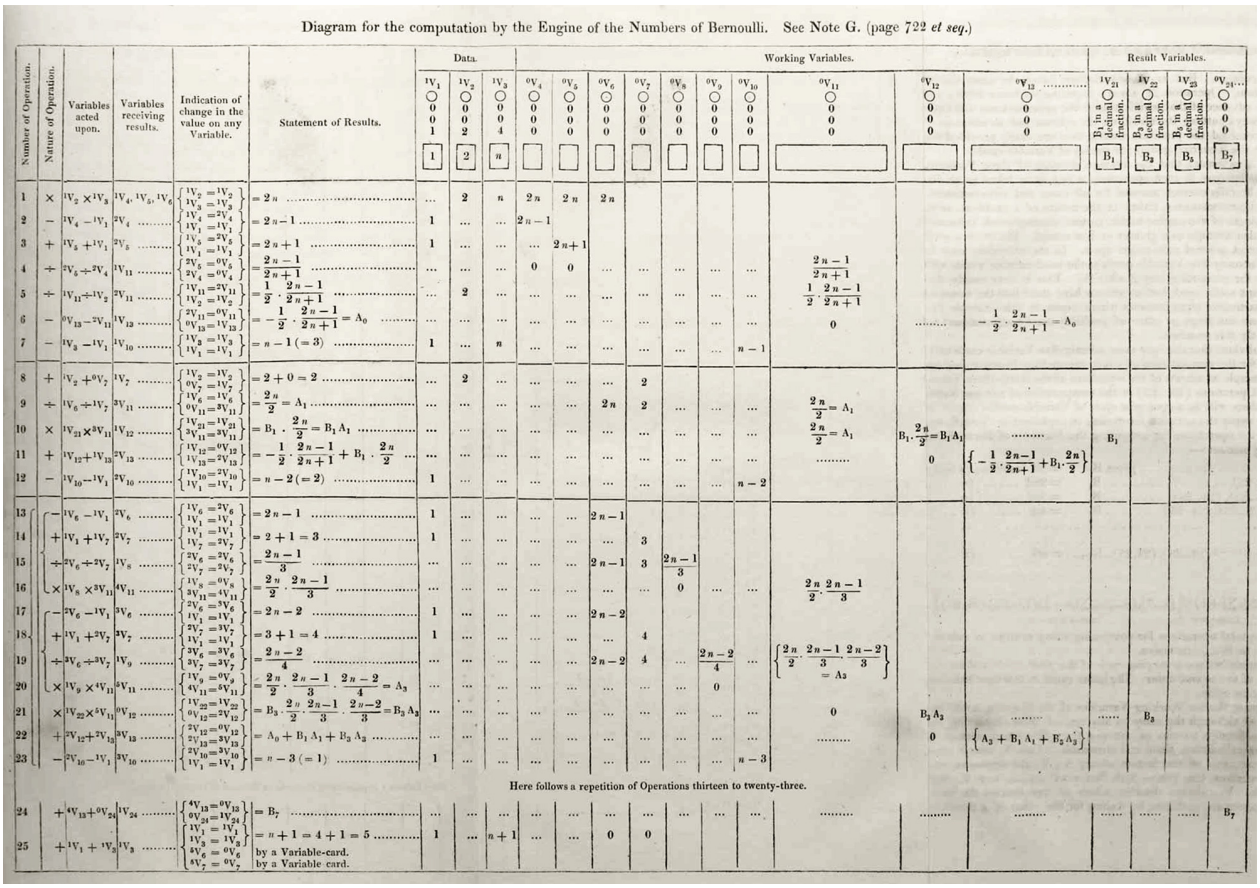

Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data. $^1V_1$	Data. $^1V_2$	Data. $^1V_3$	Working Variables. $^0V_4$	$^0V_5$	$^0V_6$	$^0V_7$	$^0V_8$	$^0V_9$	$^0V_{10}$	$^0V_{11}$	$^0V_{12}$	$^0V_{13}$	Result Variables. $^1V_{21}$ B_1	$^1V_{22}$ B_2	$^1V_{23}$ B_3	$^0V_{24}$... B_7
						1	2	n														
1	×	$^1V_2 \times ^1V_3$	$^1V_4, ^1V_5, ^1V_6$	$\begin{cases} ^1V_2 = ^1V_2 \\ ^1V_3 = ^1V_3 \end{cases}$	$= 2n$	...	2	n	$2n$	$2n$	$2n$											
2	−	$^1V_4 - ^1V_1$	$^2V_4$	$\begin{cases} ^1V_4 = ^2V_4 \\ ^1V_1 = ^1V_1 \end{cases}$	$= 2n-1$	1	...	...	$2n-1$													
3	+	$^1V_5 + ^1V_1$	$^2V_5$	$\begin{cases} ^1V_5 = ^2V_5 \\ ^1V_1 = ^1V_1 \end{cases}$	$= 2n+1$	1	...	...	...	$2n+1$												
4	÷	$^2V_5 \div ^2V_4$	$^1V_{11}$	$\begin{cases} ^2V_5 = ^0V_5 \\ ^2V_4 = ^0V_4 \end{cases}$	$= \dfrac{2n-1}{2n+1}$	...	...	...	0	0	...	...	...	...	...	$\dfrac{2n-1}{2n+1}$						
5	÷	$^1V_{11} \div ^1V_2$	$^2V_{11}$	$\begin{cases} ^1V_{11} = ^2V_{11} \\ ^1V_2 = ^1V_2 \end{cases}$	$= \dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1}$	...	2	...	...	...	...	...	...	...	...	$\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1}$						
6	−	$^0V_{12} - ^2V_{11}$	$^1V_{12}$	$\begin{cases} ^2V_{11} = ^0V_{11} \\ ^0V_{12} = ^1V_{12} \end{cases}$	$= -\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	...	...	...	...	0	.....	$-\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} = A_0$				
7	−	$^1V_3 - ^1V_1$	$^1V_{10}$	$\begin{cases} ^1V_3 = ^1V_3 \\ ^1V_1 = ^1V_1 \end{cases}$	$= n-1\ (=3)$	1	...	n	...	...	...	...	...	...	$n-1$							
8	+	$^1V_2 + ^0V_7$	$^1V_7$	$\begin{cases} ^1V_2 = ^1V_2 \\ ^0V_7 = ^1V_7 \end{cases}$	$= 2+0=2$	...	2	...	...	...	...	2										
9	÷	$^1V_6 \div ^1V_7$	$^3V_{11}$	$\begin{cases} ^1V_6 = ^1V_6 \\ ^0V_{11} = ^3V_{11} \end{cases}$	$= \dfrac{2n}{2} = A_1$	...	...	...	...	...	$2n$	2	...	...	...	$\dfrac{2n}{2} = A_1$						
10	×	$^1V_{21} \times ^3V_{11}$	$^1V_{12}$	$\begin{cases} ^1V_{21} = ^1V_{21} \\ ^3V_{11} = ^3V_{11} \end{cases}$	$= B_1 \cdot \dfrac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	...	...	...	...	$\dfrac{2n}{2} = A_1$	$B_1 \cdot \dfrac{2n}{2} = B_1 A_1$	.....	B_1			
11	+	$^1V_{12} + ^1V_{13}$	$^2V_{13}$	$\begin{cases} ^1V_{12} = ^0V_{12} \\ ^1V_{13} = ^2V_{13} \end{cases}$	$= -\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} + B_1 \cdot \dfrac{2n}{2}$ ...	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{-\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} + B_1 \cdot \dfrac{2n}{2}\right\}$				
12	−	$^1V_{10} - ^1V_1$	$^2V_{10}$	$\begin{cases} ^1V_{10} = ^2V_{10} \\ ^1V_1 = ^1V_1 \end{cases}$	$= n-2\ (=2)$	1	...	...	...	...	...	...	...	...	$n-2$							
13	−	$^1V_6 - ^1V_1$	$^2V_6$	$\begin{cases} ^1V_6 = ^2V_6 \\ ^1V_1 = ^1V_1 \end{cases}$	$= 2n-1$	1	...	...	...	...	$2n-1$											
14	+	$^1V_1 + ^1V_7$	$^2V_7$	$\begin{cases} ^1V_1 = ^1V_1 \\ ^1V_7 = ^2V_7 \end{cases}$	$= 2+1=3$	1	...	...	...	...	...	3										
15	÷	$^2V_6 \div ^2V_7$	$^1V_8$	$= \begin{cases} ^2V_6 = ^2V_6 \\ ^2V_7 = ^2V_2 \end{cases}$	$= \dfrac{2n-1}{3}$	...	...	...	...	...	$2n-1$	3	$\dfrac{2n-1}{3}$									
16	×	$^1V_8 \times ^3V_{11}$	$^4V_{11}$	$\begin{cases} ^1V_8 = ^0V_8 \\ ^3V_{11} = ^4V_{11} \end{cases}$	$= \dfrac{2n}{2} \cdot \dfrac{2n-1}{3}$	...	...	...	...	...	...	...	0	...	...	$\dfrac{2n}{2} \cdot \dfrac{2n-1}{3}$						
17	−	$^2V_6 - ^1V_1$	$^3V_6$	$\begin{cases} ^2V_6 = ^3V_6 \\ ^1V_1 = ^1V_1 \end{cases}$	$= 2n-2$	1	...	...	...	...	$2n-2$											
18	+	$^1V_1 + ^2V_7$	$^3V_7$	$\begin{cases} ^2V_7 = ^3V_7 \\ ^1V_1 = ^1V_1 \end{cases}$	$= 3+1=4$	1	...	...	...	...	...	4										
19	÷	$^3V_6 \div ^3V_7$	$^1V_9$	$\begin{cases} ^3V_6 = ^3V_6 \\ ^3V_7 = ^3V_7 \end{cases}$	$= \dfrac{2n-2}{4}$	...	...	...	...	...	$2n-2$	4	...	$\dfrac{2n-2}{4}$	...	$\left\{\dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{3} = A_2\right\}$						
20	×	$^1V_9 \times ^4V_{11}$	$^5V_{11}$	$\begin{cases} ^1V_9 = ^0V_9 \\ ^4V_{11} = ^5V_{11} \end{cases}$	$= \dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{4} = A_3$ ...	...	...	...	...	...	...	...	...	0	...							
21	×	$^1V_{22} \times ^5V_{11}$	$^0V_{12}$	$\begin{cases} ^1V_{22} = ^1V_{22} \\ ^0V_{12} = ^2V_{12} \end{cases}$	$= B_3 \cdot \dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{3} = B_3 A_3$ ...	...	...	...	...	...	...	...	...	...	...	0	$B_3 A_3$	.....	.....	B_3		
22	+	$^2V_{12} + ^2V_{13}$	$^3V_{12}$	$\begin{cases} ^2V_{12} = ^0V_{12} \\ ^2V_{13} = ^3V_{13} \end{cases}$	$= A_0 + B_1 A_1 + B_3 A_3$	...	...	...	...	...	...	...	...	...	...	.....	0	$\{A_2 + B_1 A_1 + B_3 A_3\}$				
23	−	$^2V_{10} - ^1V_1$	$^3V_{10}$	$\begin{cases} ^2V_{10} = ^3V_{10} \\ ^1V_1 = ^1V_1 \end{cases}$	$= n-3\ (=1)$	1	...	...	...	...	...	...	...	...	$n-3$							
					Here follows a repetition of Operations thirteen to twenty-three.																	
24	+	$^4V_{13} + ^0V_{24}$	$^1V_{24}$	$\begin{cases} ^4V_{13} = ^0V_{13} \\ ^0V_{24} = ^1V_{24} \end{cases}$	$= B_7$	...	...	...	...	...	...	...	...	...	...	.....	.....	.....	.....	.....	.....	B_7
25	+	$^1V_1 + ^1V_3$	$^1V_3$	$\begin{cases} ^1V_1 = ^1V_1 \\ ^1V_3 = ^1V_3 \\ ^4V_6 = ^0V_6 \\ ^4V_7 = ^0V_7 \end{cases}$	$= n+1 = 4+1 = 5$ ..... by a Variable-card. by a Variable-card.	1	...	$n+1$	...	...	0	0										

Figure 2: Ada Lovelace's notes describing an algorithm for the Analytical Engine. Image from Wikimedia Commons.

Modern computers

Today's computers work on the same four principles, but instead of decimal gears they use *binary*. At the hardware level everything is zeros and ones: physical locations on a silicon chip that sit at either a high voltage (1) or a low voltage (0). The central processing unit (CPU) is the chunk of silicon that does the processing; RAM is the short-term memory the CPU works with; the hard drive is long-term storage. The figure below shows how these components connect.

This four-component definition applies more broadly than you might expect. A fitness watch has input buttons (or a touchscreen), a display for output, a processor that tracks your heart rate and logs your runs, and long-term storage for your activity history. By our definition a smartwatch is a computer, even though most people would not call it that. Babbage's mechanical engine

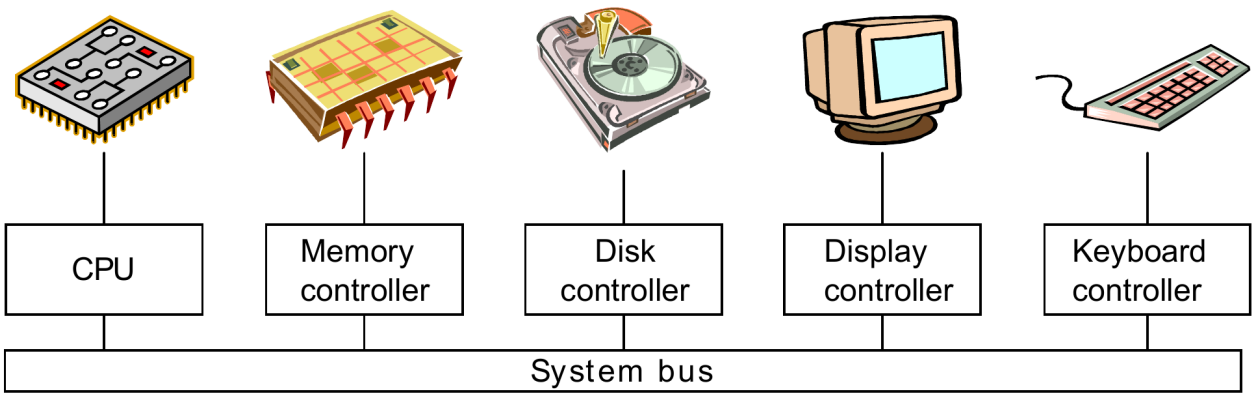

Figure 3: The components of a modern computer: input devices, a CPU for processing, RAM for short-term memory, long-term storage and output devices. Data and instructions ultimately live as zeros and ones inside the silicon.

was also a computer, despite looking nothing like a laptop. What they share is the underlying structure.

What is a computer program?

A computer program is a precise, ordered set of instructions that you give to a computer, to be carried out step by step. The key word is *precise*. A computer has no common sense and no ability to fill in gaps: it does exactly what the instructions say, nothing more. If the instructions are wrong, the output will be wrong, and the computer will not notice.

Ada Lovelace's Bernoulli-number algorithm was one of the first written programs: a sequence of steps explicit enough that a machine could follow them without human judgement. The same requirement holds today.

The question then is how we write those instructions down. In Babbage's engine you set the gears by hand. In early electronic computers you would physically flip switches to encode zeros and ones. Both approaches work, but they are slow and error-prone. Engineers needed a better way.

From machine code to high-level languages

At the lowest level, the instructions that a CPU actually executes are zeros and ones; this is called *machine language*. Every program that ultimately runs on your computer is machine language. But writing programs directly in zeros and ones is painful, so engineers invented layers of abstraction.

The first layer up was *assembly language*. Assembly language uses short English abbreviations such as `ADD`, `SUB`, `LOAD`, `STORE` that map almost one-to-one onto machine language instructions. A program called an assembler translates assembly into machine code. Assembly was a big step forward, but it is still very close to the hardware: you are essentially telling the CPU exactly which low-level operation to perform, one instruction at a time (but at least you don't have to write ones and zeros).

A few examples make the contrast between machine language and assembly clear:

Operation	Binary (machine language)	Assembly
add	`0001`	`ADD`
subtract	`0010`	`SUB`
load data	`0011`	`LOAD`
store data	`0100`	`STORE`

The bigger leap was *high-level languages*: C, C++, Java, Python, C# and others. In these languages you write instructions that look much more like English. In C, for instance, you can simply write `printf("Text that should appear on the screen")`, and a program called a *compiler* translates that into the appropriate machine instructions. (We will look at the compiler in more detail in just a bit.)

There is actually a fourth level worth mentioning: AI tools like ChatGPT and Claude. Some people talk about this as yet another level up in terms of the language hierarchy. You can describe what you want in plain English and the AI will generate code for you. We will come back to what the rise of AI means for learning to program.

The C development cycle

When you write a C program and want to run it, it goes through several steps.

You start with a *source file*: a plain text file with a `.c` extension. You can open it in any text editor, though we will use a dedicated integrated development environment that makes life easier. In that file you write your C code as English-ish sentences. (We will see an example in just a bit.)

After writing the program, the next step is *compilation*: the compiler reads your source file and translates it into an *object file* containing machine code. On a large project you might have several source files, each compiled into its own object file.

The final step is *linking*: a linker combines all the object files (and any pre-written library files) into a single *executable*. That executable is the file you can actually run. For most of this course,

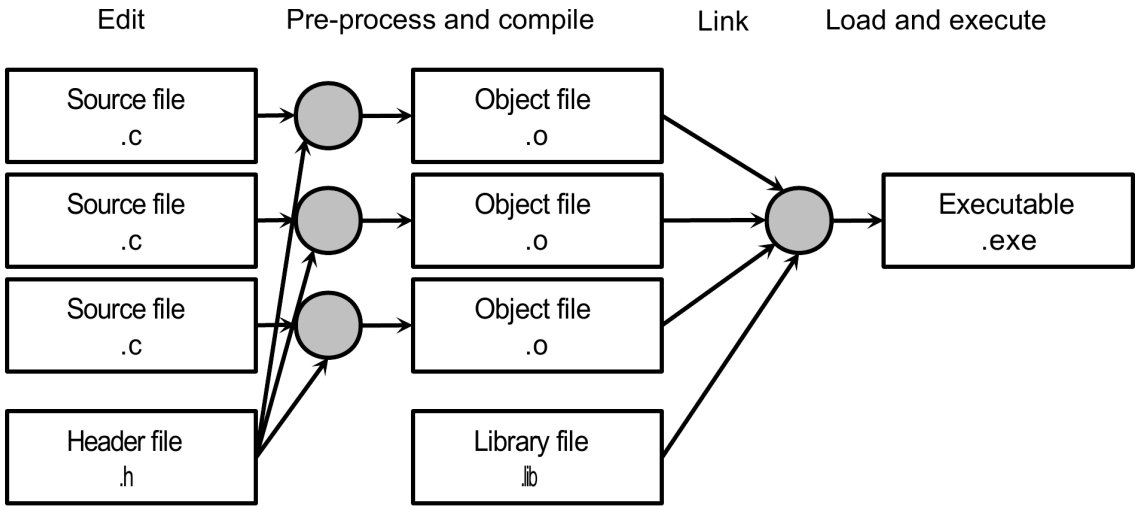

Figure 4: The C development cycle: a source file is compiled to an object file, then linked to produce an executable that the CPU can run.

all of this happens in the background when you click `Build` in your IDE.[1]

Your first C program

Here is the simplest C program that does something useful:

```c
/*
 * My first program.
 * Author: Herman Kamper
 */
#include <stdio.h>

int main()
{
    printf("Hello students!\n");  // outputs text on the screen

    return 0;
}
```

There is quite a lot here that will only make complete sense later in the course. For now, treat most of it as *boilerplate*, code you need not fully understand just yet. The one thing you do need to understand right now is `printf`.

`printf` is an instruction to the computer: put whatever is between the quotes onto the screen. That is it. The semicolon at the end marks the end of the instruction; every statement in C ends with one. Forgetting the semicolon is one of the most common beginner mistakes, and the compiler will refuse to produce an executable if you do: it will show you an error message and

[1] For C, there are many integrated development environments (IDEs). Code::Blocks is one dedicated to C and C++. Visual Studio Code is increasingly being used in practice.

stop.

The `\n` inside the quotes is a special code for a newline: it moves the cursor to the next line. You can also chain multiple `printf` calls one after the other; each will print its text to the screen in sequence. You can try this by modifying the code.

The rest of the boilerplate:

- `#include <stdio.h>` tells the compiler that we want to use some standard input/output functions that someone else has already written, including `printf`.
- `int main()` defines a function called `main`. Every C program must have one; it is the entry point where execution begins.
- `{ ... }` after `main` encloses the program's instructions; this is where the main work happens.
- `return 0;` at the end of `main` signals to the operating system that the program finished successfully. For now, just include it every time.

Comments are text that the compiler ignores entirely. There are two forms in C. A single-line comment starts with `//`. Everything after those two slashes on that line is ignored. A multi-line comment is surrounded by `/*` and `*/`. Everything between them is ignored, even if it spans several lines. Comments are for other humans who will read your code, or for your future self returning to something you wrote six months ago. You cannot make a syntax mistake in a comment: write whatever you like.

That's your first program!

Why learn to program if AI can do it?

Let's return to an earlier question: if AI tools like ChatGPT and Claude can generate code, why study programming at all?

The first answer is that programming teaches you a way of thinking. Learning to break a problem into a precise, unambiguous sequence of steps (an algorithm) is a transferable skill. It shapes how you approach any problem, not just software. You learn to multiply even though a calculator can do it for you, because the underlying concept matters. My four-year-old son asks why he needs to learn to multiply two numbers if a calculator exists, and the answer is: because the concept of multiplication is important for life, and the calculator is just a convenience on top of it. Programming is similar.

The second answer is practical: solve the problem first, then write the code. For simple problems you can jump straight to typing, and it works. For harder problems, many programmers start writing code immediately, and most of those attempts fail. The ones who succeed first sit down, think through the solution on paper, and only then open the IDE. The same principle applies when using AI: if it is a difficult problem, you would probably have to solve a large part of it conceptually before starting to compose your prompt.

The third reason is that AI does make mistakes, particularly in C. Memory management in C is genuinely difficult, and AI tools get it wrong more often than they do in Python. If you do not understand C, you will not catch those errors.

Finally, fluency compounds: programmers who already know the language get disproportionately more leverage from AI assistance than those who cannot evaluate what the AI produces.

Why C?

C is not the gentlest first language. So why are we using it? A look at the most popular languages in use today gives part of the answer.

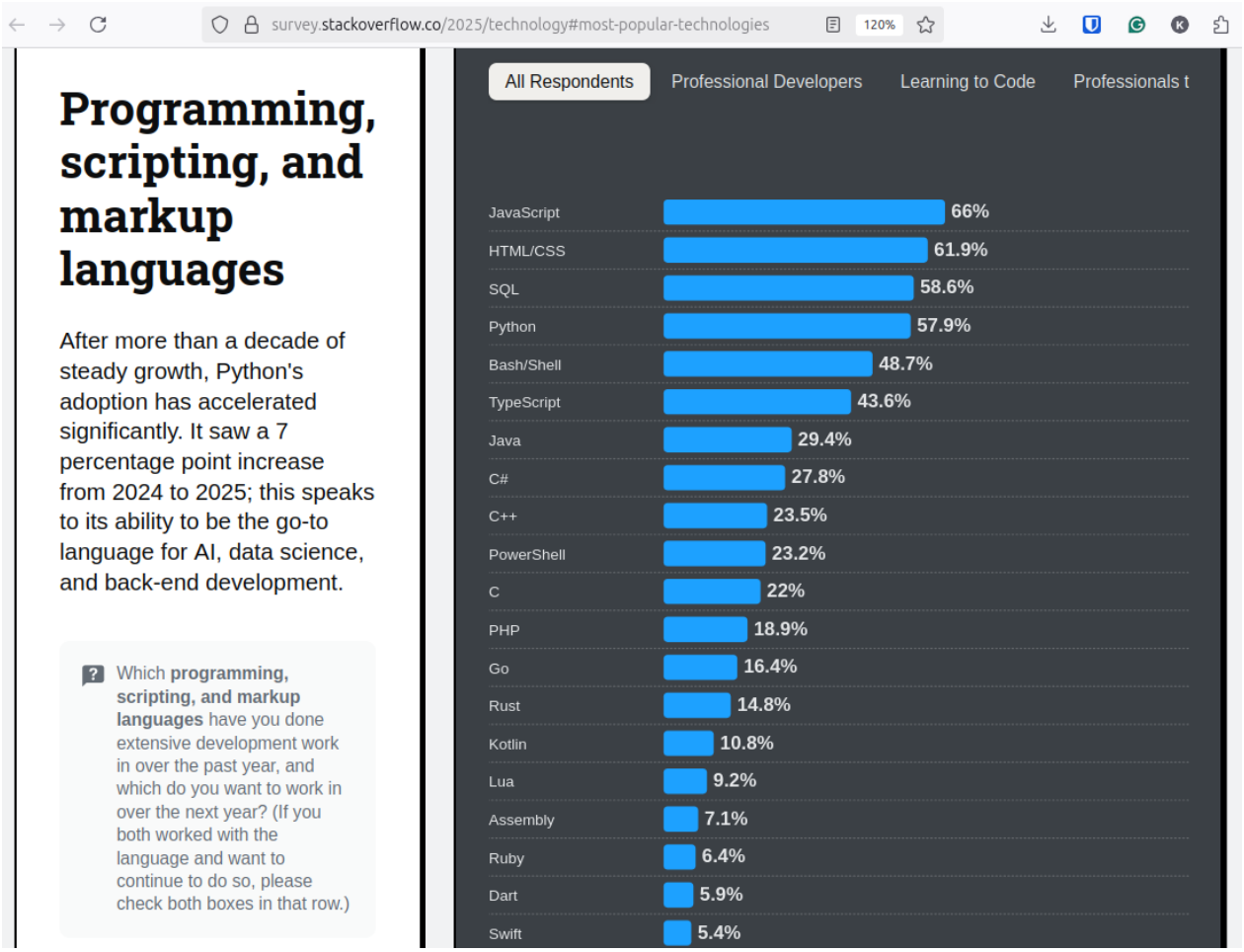

Figure 5: Most popular programming languages based on a 2025 developer survey. Several of the top entries (C++, C#, Java) are directly related to C.

The main reason is that C is a foundation. If you can program in C, picking up C++, C# and Java requires only a modest amount of additional learning, because the core concepts are already familiar. Python, which reads almost like pseudocode, becomes quite easy. C forces you to understand how programs actually work at a lower level: memory, data types, control flow. Those fundamentals transfer directly to other languages and systems.

There is also a practical reason for engineering students: embedded systems are overwhelmingly programmed in C. Microcontrollers, automotive electronics and similar devices often do not

support higher-level languages. If you are an electrical, electronic or data engineering student in particular, you will encounter C in industry.

Looking forward

In this lecture we have answered two questions: what is a computer, and what is a program? We have also written our first C program and have seen how it gets compiled and run. From here the course builds systematically. The next lecture looks at variables: how to store values in short-term memory so that your program can work with them. After that we look at taking input from the user, making decisions, and repeating steps. By the end of the course you will be able to write programs that solve real engineering problems.

Exercises

1. A simple alarm clock has a display (output) and buttons (input), but no processor and no memory beyond a single stored alarm time. Does it meet the four-component definition of a computer from this lecture? Justify your answer.

2. Describe the difference between machine language, assembly language and a high-level language like C. For each one, give a brief example of what an instruction in that language might look like.

3. Consider the following C program:

```c
#include <stdio.h>

int main()
{
    printf("CP143\n");
    printf("Computer Programming\n");
    return 0;
}
```

   (a) What output do you expect on the screen when this program runs?
   (b) What would happen if you removed the `\n` from the first `printf`?
   (c) What would happen if you removed the semicolon after the first `printf`?

4. Describe the two main steps in the C development cycle: compilation and linking. What does each step take as input and produce as output? What is the final file that comes out of the process, and what can you do with it?

5. Add a comment block at the top of the program in Exercise 3 that includes the program name, your student number and a one-sentence description of what the program does. Then add a single-line comment next to each `printf` describing what it outputs.

6. A student says: “There is no point learning to program when AI tools like ChatGPT can write code for me.” Write a short paragraph responding to this claim.