Beej’s Guide to Learning Computer Science

Brian “Beej Jorgensen” Hall

Contents

Chapter 1

Foreword

Are you getting into Computer Science, or thinking about it? Or maybe you're in it already. This super-high-level guide is for you!

I'm not going to talk about how to write code (much). I'll I'm going to talk about in these roughly 40 pages is more about *how to learn* when you're a nascent software developer.

Now, as much as I'd like to know exactly the way that everyone learns (and manage to wedge that into 40 pages), I, to be perfectly honest, don't.

What I do have is 40+ years of programming experience (self-taught before college), 20 years of industry experience, and 8+ years of teaching experience. And a BS and MS in Computer Science.

And I have opinions about how to best way to learn how to program!

Now, let's get this right out of the way: you might completely disagree with what I have to say here. And I'm okay with that.

But I have had the opportunity to see students make a wide variety of mistakes. And hopefully I can head some of these off at the pass for a number of readers.

Students and teachers, alike: if you find something you disagree with or something vital that is missing, please don't hesitate to let me know[1] so I can improve the guide.

Disclaimer: like with all the guides I write, I'm not the master of the subject. And with a squishy topic like how humans learn, I'm even less so.

But give it a read and take what's useful and leave the rest for the boids[2].

1.1 Audience

Undergrad students just getting into programming are the people I had in mind while writing this. So give-or-take a bit around that target audience. People in high school or just looking to learn how to program are also probably out there in the audience, as well.

1.2 Official Homepage

This official location of this document is:

`https://beej.us/guide/bglcs/`[3].

[1] mailto:beej@beej.us
[2] https://en.wikipedia.org/wiki/Boids
[3] https://beej.us/guide/bglcs/

1.3 Corrections

I make these guides available for free in the sincere hope that people will find them maximally useful. If there's something that's not maximally useful (or, you know, "wrong"), I'd love to hear about it so I can fix it in furtherance of my mission.

- Email me[4]
- Report an issue[5]
- Submit a pull request[6]

Thank you!

1.4 Email Policy

I'm generally available to help out with email questions so feel free to write in, but I can't guarantee a response. I lead a pretty busy life and there are times when I just can't answer a question you have. When that's the case, I usually just delete the message. It's nothing personal; I just won't ever have the time to give the detailed answer you require.

As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you're getting, and anything else you think might help me troubleshoot), you're much more likely to get a response.

If you don't get a response, hack on it some more, try to find the answer, and if it's still elusive, then write me again with the information you've found and hopefully it will be enough for me to help out.

Now that I've badgered you about how to write and not write me, I'd just like to let you know that I *fully* appreciate all the praise the guide has received over the years. It's a real morale boost, and it gladdens me to hear that it is being used for good! `:-)` Thank you!

1.5 Mirroring

You are more than welcome to mirror this site, whether publicly or privately. If you publicly mirror the site and want me to link to it from the main page, drop me a line at `beej@beej.us`.

1.6 Note for Translators

If you want to translate the guide into another language, write me at `beej@beej.us` and I'll link to your translation from the main page. Feel free to add your name and contact info to the translation.

Please note the license restrictions in the Copyright and Distribution section, below.

1.7 Copyright and Distribution

[4] mailto:beej@beej.us
[5] https://github.com/beejjorgensen/bglcs/issues
[6] https://github.com/beejjorgensen/bglcs

One specific exception to the “No Derivative Works” portion of the license is as follows: this guide may be freely translated into any language except English, provided the translation is accurate, and the guide is reprinted in its entirety. The same license restrictions apply to the translation as to the original guide. The translation may also include the name and contact information for the translator.

The programming source code presented in this document is hereby granted to the public domain, and is completely free of any license restriction.

Educators are freely encouraged to recommend or supply copies of this guide to their students.

Contact `beej@beej.us` for more information.

1.8 Dedication

The hardest things about writing these guides are:

- Learning the material in enough detail to be able to explain it
- Figuring out the best way to explain it clearly, a seemingly-endless iterative process
- Putting myself out there as a so-called *authority*, when really I’m just a regular human trying to make sense of it all, just like everyone else
- Keeping at it when so many other things draw my attention

A lot of people have helped me through this process, and I want to acknowledge those who have made this book possible.

- Everyone on the Internet who decided to help share their knowledge in one form or another. The free sharing of instructive information is what makes the Internet the great place that it is.
- Everyone who submitted corrections and pull-requests on everything from misleading instructions to typos.

Thank you! ♥

Chapter 2

The Main Goal

> *"Education is not preparation for life; education is life itself."*
>
> —John "Not The Decimal System One" Dewey

> *"The illiterate of the 21st century will not be those who cannot read and write, but those who cannot learn, unlearn, and relearn."*
>
> —Alvin Toffler

What are we learning in school? How to be a Flutter developer? How to be a React developer? How to be a Rust developer? How to be a JavaScript developer? How to be a C++ developer? How to be a C developer? How to be a Pascal developer? How to be a LISP developer? How to be a FORTRAN developer? How to be a COBOL developer?

See what I did there? Besides ask a lot of questions?

Yes, you might be wanting to go to school so you can work on web development or embedded systems in the latest and greatest languages. And maybe if you're lucky, you'll do some of that in school.

But here's the problem:

1. There are too many technologies to cover in four years.
2. All that stuff is going to be obsolete soon, anyway. See how I added COBOL[1] to the list[2]?

So what can you as a student do? There's no way you're covering it all.

This is where the main goal comes into play. Your job as a student is to do one thing:

Learn how to solve any programming problem.

Even if you've never seen the problem or technology in your entire life.

That's the whole goal.

Importantly, it's not to learn to be an iPhone developer or an Android developer or a Go developer. All that stuff is covered by the main goal. You might not learn Go programming in school, but you'll *learn how to learn Go programming on your own*.

Learn how to solve any programming problem.

That's it. Everything else is window dressing.

Being able to learn things on your own is a **required** skill when it comes to software development. It's *really* unlikely that your first job will solely use technologies you've used in school. And, in fact, recently-graduated students might be surprised to find out that **none** of the technologies they've used at school are present in their first gig.

[1] https://en.wikipedia.org/wiki/COBOL

[2] Joke's on me. There are still tons of COBOL jobs out there.

So why on Earth did we just spend four years learning all this crap about operating systems and assembly language and algorithmic analysis and—?

EERRRNT! [buzzer sounds] You didn't just spend four years doing that. You just spent four years *learning to solve any programming problem.*

And think about it. How would you teach people to solve *any* problem? Well, you can't exhaustively teach them all zillion programming languages[3], frameworks, and techniques. So that's off the table. And whichever ones you do pick might or might not be used for a particular person's job.

So we have to get more fundamental. We have to practice solving problems so many times that we develop and enhance our problem-solving skills. Because you're going to be faced with problems in an interview or at work that are completely unfamiliar. You won't be able to draw on any specific languages or algorithms you've learned. The one thing you will be able to use is your problem-solving skill.

Finally, there's a bit of a corollary here: when you're learning to do something (even if you'll never use it at work), *don't cheat*. The goal isn't to learn how to delete the head of a linked list. The goal is to practice solving programming problems! And just looking up the answer deprives you of that practice. Cheat your way through all the assignments at school and you'll never develop the one fundamental skill of software developers: being able to solve any programming problem.

2.1 Chapter Reflection

- What is the main goal of a computer science student?
- Why is learning languages and frameworks *not* the main goal?
- What do you have to do to accomplish the main goal?
- Why is cheating in computer science classes a bad idea in terms of your learning?

[3] https://en.wikipedia.org/wiki/List_of_programming_languages

Chapter 3

Growth Mindset

This is a tough one for me, personally. I don't like failing at things, even if no one can see me doing it, and especially when people do. It knots my stomach and then I say all kinds of bad things to myself that I wouldn't ever say to anyone else.

And I do this even though I know it's a losing game and it contradicts the exact advice I give in this section.

What should I be doing, instead?

Psychologist Carol Dweck[1] popularized the term *growth mindset*.

The gist of it is:

- You can expand your skills with effort
- Embrace lifelong learning
- Challenges are growth opportunities
- Learn from criticism and failure
- Never give up, never surrender![2]

That kind of thing.

It's the opposite of what I tend to do when I lose my bazillionth game of Go[3]. How could I have made so many stupid mistakes? I'll never be good at this!

But that's what Dweck would refer to as a *fixed mindset*. It's my mistaken belief that no matter how much I play the game, I'm up against my own intrinsic limitations that I'll never get past no matter if I study for 500,000 hours!

And when I put it that way, it's kinda silly. No one can spend 500,000 hours doing *anything* without getting better at it.

> *Lose your first 50 games as quickly as possible.*
>
> —Go proverb

So what about 50,000 hours? 500 hours? 50 hours? 5 hours?

Come to think of it, it seems like any amount of practice is going to be an improvement.

Even when you're completely stuck, you're still exploring avenues. Even if they turn out to be dead-ends, you've at least learned that they are!

> *"Isn't it the same thing, like 'flammable' and 'inflammable'? Boy, I learned that one the hard way."*
>
> —Woody, *Cheers*

[1] https://en.wikipedia.org/wiki/Mindset#Fixed_and_growth_mindset
[2] Yes, I'm a *Galaxy Quest* fan.
[3] https://en.wikipedia.org/wiki/Go_(game)

> *"Great. Another f---ing learning experience."*
>
> —My sarcastic mother

Every success is a learning experience. Every failure is a learning experience. Every learning experience improves your skills. Don't fear failure—use it to level up.

3.1 Tenacity

> *"The master has failed more times than the beginner has even tried."*
>
> —Stephen McCranie

I think this is one of the main attributes of anyone who grows to excel at anything. (How's that for a generalization?)

Who is going to get farther, the person who gives up after a failure, or the person who fails and fails and fails and fails and keeps getting up, dusting themselves off, and trying again?

This is what separates the exceptional devs from the unremarkable devs. The exceptional devs have had their asses *kicked* over and over and over, and *they kept attacking* and attacking and attacking until they solved the problem. And they learned something from every single failure.

> *"I have not failed. I've just found 10,000 ways that won't work."*
>
> —Thomas Edison

And something that's related here that students might not realize: your instructors failed and failed and failed and failed, too! Sure, they can live-code the linked list delete in class and have it work on the first try… except that wasn't the first try, was it? It was, like, the 100th time they've done it. You code up a linked list delete from scratch 100 times and you'll be getting it right on the first try, too!

When watching an instructor make something look effortless, you might become disheartened because this material seems so *impossible* for you. And you start to think that your instructors and some of your peers have a natural ability to code that you simply weren't born with. What magical skill were they endowed with in the womb that you will never have? (Hear that fixed mindset talking?)

But here's the secret: there's no difference between you and your instructors other than the number of failures you've had. You'll need to fail a lot more to get as good as they are!

Very, very few people are "natural" coders. 99.999% of the rest of us have to work really hard to learn this stuff.

Relentless tenacity is one of the prime qualities of the top coders of the world.

3.2 You Gotta Want It

When I was thinking about getting a PhD (I didn't), my adviser advised, "You gotta want it." The implication was that it was so much work to get a PhD that I had to have enough drive to put in the time.

And I think this is good advice in general about learning Computer Science.

There's so much really hard stuff to figure out, you have to have enough drive to do it. It's not easy. At all.

I like to use the example of me being an accountant. I know I'm smart enough to do it (math minor!) but I also know I *hate* it. My old roommate became an accountant and I had the misfortune of cracking open one of her study guide books. Never again will I make that mistake.

My brain just shuts down the minute people start talking about it. Imagine something that bores you to tears that you hate. Now imagine spending four years studying it in excruciating detail.

Speaking for myself and projecting broadly, I think it's a lot harder to motivate to put in the work to do something you hate.

I caught a lot of flak on Hacker News for this assertion a while ago. Several people commented that they hated programming and still managed to have a career at it.

First, that's a… bummer. But secondly, I still argue that people who love computer science have an easier time getting their degree than people who hate it.

And the people who seem to be "naturals" are the ones who *really* love it. They think about computer science *all the time* (because who wouldn't?) and, as such, get a lot of practice in. Rather a lot more than the people who hate it tend to.

So, while it's clearly possible to have a career in a lucrative field you dislike, it's (a) going to be harder for you than for people who like it and (b) maybe you should consider a field that you do like?

You gotta want it. Do you want it enough to go through the tremendous amount of effort it takes to learn it? Maybe you hate programming, but you want the money enough. Maybe you don't care about the money, but you want to program every second of the day.

Just make sure you have the drive to make it happen.

3.3 It's Not Easy

> *As you read this scroll*
> *You will smoley begot*
> *Confused by the printed worts*
>
> —"Scroll of Learning Disability", *Dragon* Magazine

Computer Science is *hard*. Like really hard. If you're used to things being relatively easy because you were the smart one in class, it might be a shocker to get to some of these topics and get your ass handed to you.

But it's this hard for everyone. You're not blessed with a special lack of ability to learn computer science. You just have to keep hammering away just like everyone else.

You wouldn't doubt that juggling 11 balls at once is difficult for everyone, would you? And yet, people can do it[4].

> *"You know how to get to Carnegie Hall?"*
> *"Practice!"*
>
> —Joke of unknown origin

Reps, reps, reps, reps.

When you're at school, you actually have it even worse than at a job. In school, you're continuously learning new things and, just when you *barely* have a grasp of a topic, you move on and try to learn something else that's incredibly challenging. An undergrad degree is four solid years of just barely being able to hang on to the massive amount of perpetually novel material you're presented with.

And when you do finally get your degree and a job, the first three months of that job are *also* a high-paced barely-hanging-on learning experience.

But then after three months, you start to get a handle on the code. And it gets easier. The feeling of hanging on by your fingernails does start to abate. And—I know this might be hard to believe now—eventually the job might actually get so easy as to be boring. And then you'll look for another one with new, exciting challenges to tackle.

3.4 Chapter Reflection

- Contrast *growth mindset* with *fixed mindset*.
- Have you ever been limited by a fixed mindset? What are some techniques you might use to switch to a growth mindset?

[4] https://www.youtube.com/watch?v=cJsgM-3L38U

- Why is *tenacity* an important quality for devs to possess?
- What is the main difference between an instructor who makes something look easy and a student who is struggling hard with the same material? What does the student have to do to also make things look easy?

Chapter 4

Problem Solving

This is an idea completely stolen from the book *How to Solve It*[1] by George Pólya. It's a book about attacking math problems. And since computer science is just math under the hood, it completely applies!

Actually it doesn't completely apply, and I just said that because it sounded so good. But it can be bent into shape pretty easily. And I recommend reading the book.

So what is it? It's short and pretty easy to memorize:

1. **Understand** the problem
2. **Plan** how you're going to solve it
3. **Code** up the solution
4. **Reflect** on what you could do better

That's it.

And if you think about it, that's really just a four-step process for solving just about any problem at all. Is your living-room light not working? You can solve it with these steps.

> *"Math ain't about numbers. If you think math is about numbers, you probably think that Shakespeare is all about words. You probably think that dancing is all about shoes. You probably think that music is all about notes. Math ain't about numbers. Math is about logic, it's about beauty, it's about connections, it's about how you get from one place to another."*
>
> —Cliff Stoll

Programming ain't about writing code. **Steps 1, 2, and 4 do not involve coding**[2]. These steps are all in your head and on paper. I would argue that ***solving programming problems does not involve a computer***. That seems clearly nonsensical, but that's where the action actually is! Especially in step 2, coming up with a *Plan*. That's the hard part. That's why you get paid the big bucks. Anyone can type a program in once the problem has been solved.

Step 3, *Coding* it up, is simply writing the solution down. The hard work isn't writing the solution down; the hard work is coming up with it in the first place!

Also, you're unlikely to progress linearly through these steps. You'll probably have to pop back and revisit earlier steps from time to time, but hopefully your overall progress is in the forward direction.

Finally, as a student, you must resist the urge to look up the answers. The goal here is for you to exercise your problem-solving muscles (because no one will hire you as a dev without those skills). Virtually every problem any instructor will come up with has been extensively covered on the Internet or can be solved by AI. Remember that getting the correct answer isn't the point; the point is to practice problem-solving so that you can get an answer to any problem that's thrown at you.[3]

Let's explore the steps.

[1] https://en.wikipedia.org/wiki/How_to_Solve_It

[2] Sometimes they can, actually, but only to write small, throwaway, exploratory proof-of-concept programs.

[3] I don't think AI can solve *all* problems that get thrown at it, meaning that you'll still have a job, but they can solve the relatively basic problems that are commonly used in computer science curricula. It's 2025 now and we'll see how well this footnote ages.

4.1 Understanding the Problem

Everyone, students included, loves to skip the *Understanding* step. Wise instructors will add a quiz to be turned in before coding starts that verifies you understand the problem[4]. But you need to pressure yourself to do this step first, regardless.

Logically, if you don't understand the problem fully, none of the rest of the steps matter. You could come up with the best plan and code it up perfectly, but if you didn't understand the problem to begin with, you've just coded up the perfect solution to the wrong problem![5]

And that's a waste of your employer's time and money, at best. Or, if you're still a student, a waste of time you could have spent studying for that discrete math exam that's coming up. Or whatever.

So don't skimp on this one—it's fundamentally important.

What should you do? Here are some ideas.

- Read the problem slowly and methodically. These descriptions tend to be concise and error-prone. They're not easy to read, so don't even try to speed through them. Reread sentences as many times as it takes to understand what they mean.
- Take notes. Especially note inconsistencies, omissions, errors, and outstanding questions.
- Rewrite the problem in your own words. This is a very effective technique that helps you find gaps in your understanding.
- Research as necessary.
- Ask clarifying questions[6].

You're done with this step when you can teach someone else what the problem is. (You don't need to teach the solution—just the problem.)

It's very, *very* hard to write a complete description of a problem, and the ones you'll get at work or even in school will certainly be wanting. Don't believe me? Write down the rules of tic-tac-toe ("noughts and crosses" if that's what you call it) and I can guarantee I'll find something you didn't write down[7].

Because of this, you'll very likely have to ask clarifying questions.

> **Anecdote time!** I was on a project where I was writing the front-end of a system and someone else was writing the back-end. There was a very specific document describing the exact interaction between the two.
>
> One of the interactions involved the transmission of the result of a computation. There was an example in the doc that gave the answer, in hex, to the math. It was something like:
>
> > Given the inputs `"abc"` and `"xyz"`, the result will be the number `f319c2c6dcfb`.
>
> I coded it up (it was easy since the algorithm pseudocode was given), and it computed the answer correctly. I added the code to transmit the 6-byte number to the server and that was it.
>
> Except the person writing the server wrote me and said that I was sending garbage.
>
> We had some back and forth, and it turned out they thought the spec meant I'd send a string with those hex digits in it, and I thought the spec meant that I'd send the raw bytes of the result. The spec was no help—it was ambiguous and neither of us caught it.
>
> The easiest thing was for me to convert the number to string and send it, so I added that line of code and the problem went away. But we could have caught it earlier with more careful examination of the spec.

[4] I could stand to be more wise, myself.

[5] Once on a programming challenge website I coded up perfect solutions to *two* problems that weren't the one I was meant to solve. I misunderstood it twice. Took me three-times longer than it should have to get the actual solution in place.

[6] This is actually part of your job description as a dev. People will expect and rely on you to do this in the workplace. So don't be afraid of doing it; be afraid of *not* doing it.

[7] You didn't say how to choose who moves first, if it could be played by three people, or that my "X" couldn't take up more than one square. For example.

4.2 Coming Up with a Plan

> *"Programming is the art of telling another human being what one wants the computer to do."*
>
> —Donald Knuth

Time to start coding? **NO!** Not yet, eager beaver![8]

This step, coming up with a plan, is where programming actually happens. Like I said before, this is what makes the job hard, and is why it pays well. To be a halfway decent dev, you need to excel at coming up with solid plan.

First you must *Understand* the problem well enough to teach it to someone. Or at least you *think* you do. You might learn more later. But understanding the problem isn't the same as knowing how to solve it.

If the problem is really simple and familiar, you might come up with a plan almost instantly. Experienced devs faced with familiar tasks don't need to spend long planning once they've fully understood the problem.

But experienced devs faced with unfamiliar tasks *do* need to spend time at it. It doesn't matter how good you are—if you haven't seen the problem, you're going to need to plan the solution.

Literal pencil and paper can be useful for this step. Here are some things to think about.

- How will data flow through the system and be transformed, from the known input to the desired output?
- During those transformations, what are the subsystems that will perform each step?
- What technologies will you need to perform these steps?
- What are the known unknowns?

One big thing to notice is that we're talking about breaking down a problem into its subcomponents. How to do this isn't always obvious, though the more experience you get, the easier it becomes.

A trick you can use is to think, "This project would be easy if only the input data were in *this* form." Then see if you can come up with some code that converts the data into that form. We'll go into more detail in a later chapter.

Another benefit of breaking up the project into smaller parts is that it can naturally suggest a way to break up your code into smaller functions or classes. This makes your code more maintainable and readable.

You'll have to do some research during this phase to learn what tools you have at your disposal. Expect to do that.

It's *really* common during the planning phase to realize that you've failed to understand the problem fully. When this happens, drop back to the *Understanding* phase to get clarity, then jump back to planning.

You know you're done with planning when you can simulate the run on paper and in your head and you're confident it works. And you can write high-level pseudocode. Bounce your plan off a few rubber ducks[9] to see if it holds water.

4.3 Coding Up a Solution

This is the easy part! You have to translate your pseudocode plans into code.

If you've understood the problem and come up with a solid plan, the code will work. Maybe even on the first try, if you're very lucky[10].

[8] As of 2025, I worked at Oregon State University. Go Beavs!

[9] *Rubber ducking* is sharing ideas with a literal or proxy rubber duck, where the proxy might actually be a person. It helps you clarify your thinking and achieve problem-solving breakthroughs. Also, the Ducks are University of Oregon's football team, Oregon State's longtime rivals. Boo Ducks!

[10] This very rarely happens to me. When it does I quickly peek out the window to make sure skies are clear and I'm not about to be struck by lightning. And I buy a lottery ticket. It's inevitably then that my luck runs out.

If you *didn't* understand the problem and didn't come up with a plan, then this is not the easy part. You will see no end to trouble, and might fail to complete the project. *That's* how important understanding and planning is!

Of course, we're only human and we'll mistype things and make dumb errors, but that's *way* easier to debug than if you have a bad plan, or worse, bad understanding of the problem. Yikes!

The hard parts of coding it up are:

- Knowing the language syntax.
- Knowing what libraries are available.
- Not making trivial mistakes.
- Following the plan exactly.

We'll talk about how to learn languages later, and the second two bullet points can be addressed by being more careful, using something like pair programming[11], or leaning on AI.

While you're coding, you might find a place where your plan doesn't work. This happens quite a bit. When it happens, drop back to the planning phase, fix the plan, and then come back up to the coding phase to implement it.

Again, this phase is supposed to be relatively easy. If you're having trouble trying to get it to work beyond just fixing your syntax errors (i.e. there's something more structural amiss), your understanding or your plan is likely flawed. You should drop back to the planning phase to fix it, and maybe back to the understanding phase, if required.

You're done with this phase when the code works and passes all your tests.

4.4 Reflect on Improvements

> "Coding is a craft. Take pride in your work."
>
> —Yours Truly

Last and definitely not least, look back and cast a critical eye on what you've done. Yes, it works and passes all the tests. But is it as elegant as it could be? Is it as readable and maintainable as it could be? Are there are places where the code is fragile and might fail on unexpected inputs?

No matter what your skill level is, there is always room for improvement. And one of the top ways we learn is to look back on the crappy code we wrote and see how we could have made it better.

Code reviews are fantastic if you can coax someone into taking the time to do it for you. They will make suggestions for things you can improve, and you can fix them now and remember them for next time. And you might disagree and not make those fixes; that's okay, too.

Again, we can leverage AI to help with this. Once you've solved a problem and have it working, ask the AI for suggestions for improvement[12] and see if there are any worth following. This technique is effective on small pieces of code, not large projects, but that makes it a great assistant for undergrad work.

> **But, very importantly for undergrads, you need to solve the problem first**, and only then ask the AI to help you improve it. Your goal in school training (just like in gym training) is to get a workout with feedback, not have someone else do the work for you.

Code can be bad in a number of ways. It can be buggy. It can be inefficient. It can have bad formatting. And it can simply be unreadable. Remember that in order for your code to be maintainable, it needs to be easily understandable by other humans. Make it look sharp. The compiler might be happy with unreadable code, but humans shouldn't tolerate it.

You're done with this phase never. Well, practically you're done with it when you give up and decide you've learned enough about improving the code. But you're probably not going to find the *Ultimate Solution to the Problem Ever* because you're still building your skills throughout your life.

[11] https://en.wikipedia.org/wiki/Pair_programming

[12] Make sure your instructor and/or employer allows this.

That said, this phase is where a **lot** of learning happens. This is where you can effectively build your stats with relatively low effort. And it's your loss if you don't spend just a few minutes after a project to take advantage of it.

At work, this takes the form of a *post-mortem*, where the people involved in the completed project look back and study what went right and wrong.

4.5 Think Like a Villain

When solving problems, I want you to think like a villain; that is, think like someone who is going to abuse the system that you're designing and building.

A real square root function, for example, could be well tested. Give it some perfect squares, some non-perfect squares, some fractions, etc. Works perfectly. You wouldn't pass in negative numbers, right? That would be silly. But you know who would? A villain!

> **I once visited an online shop that allowed you to order negative amounts of product.** The checkout page said they were going to credit my account by tens of thousands of dollars.
>
> I never had the guts to check out, though, since I didn't want to be on the hook for shipping all that product to them.
>
> Plus, I'm not a real villain… I was just thinking like one!

Expect the unexpected in terms of data that your code will receive. Expect malicious actors to feed in data in an attempt to gain unauthorized access or manipulate the system in undesirable ways. Test for that stuff in your code.

This applies to all phases from understanding to reflection.

> **Someone I knew in college worked for the army** looking at plans for things like tanks that had not yet gone into production. His job was to think like a villain and ask questions like, "How are you going to get a wrench between those pipes to tighten that bolt?"

Thinking like a villain can not only catch problems you might not have otherwise considered, but it will lead you to a deeper understanding of the project that will produce more durable and maintainable code.

4.6 Use in Interviews

The four-step process from this chapter is exactly what you should use on interview coding challenges.

See, these interview problems are quite insidious. They don't have obvious answers at all.

And why not? Is it because you haven't studied enough? **No.** That's not it at all. No matter how much you study, interviewers will come up with a question that doesn't have an obvious answer to you, the sadistic devils.

And why would they do that? Do they want you to fail? Not at all. *They want to see you apply your problem-solving skills*.

> **This isn't universal, incidentally.** There are a million interviewing styles, and some of them really do just want to see how many correct answers you get. But I would argue they aren't interviewing optimally. And further, I'd argue that the only way to get correct answers is to apply the problem-solving steps, so you might as well do it in every case.

It's natural when you're under the stress of the interview and are presented with an initially-impossible problem that you seize up like a deer in the headlights. All your knowledge suddenly vanishes in a cloud of mist and you know you'll never get this job now ever and—

DON'T PANIC. Say to yourself these words: "The only thing that matters now is *understanding the problem*." Forget the solution. That's not important. Coding it? Not important. Just focus on step one: understand the problem.

There are two main reasons for this. One is that the interviewer is hoping you'll start there. (And that should be reason enough.) But the other is by starting with understanding the problem, your brain will kick back into gear and automatically start thinking up strategies for solving the problem… and that's step two of the problem-solving framework! You're already on your way.

Try to think (you villain) of anything ambiguous in the problem description. Ask questions to clarify those things. What are the limits on the input? What are the limits on output? What do you do in error conditions? Maybe suggest an example input and output and verify that you have it right with the interviewer.

This tells the interviewer that you give attention to detail, an important trait to have. And it also tells them you start a project by making considered, deliberate choices.

And get this: for many interviewers, seeing you effectively attack a problem in a systematic way is actually more important than you getting the correct answer. And on the flip side, not showing your problem-solving skills when giving an answer might sink you, even if the answer is correct!

> **When I interviewed at Activision**, there were two questions I did not get right. But I hammered my way through them aloud as best I could, showing how I'd attack problems. I got the job.
>
> (The blown questions were: "What is the fastest way to reverse the bits in a byte?" and "Optimize this computation that builds a grid of distances between all soccer players on a field.")

So go through the whole process with the interviewer. And don't forget the final reflect step! What would you do better? How could the solution be improved? What features could be added in the future? Interviewers love that stuff, and you love keeping interviewers happy, right?

4.7 Cost per Phase

One note related to the problem-solving framework is that *the cost of changes to the software increases exponentially the farther you are in development*.

When you're at the *Understanding* phase, changes are really cheap. They're free. You haven't even come up with a plan yet, and you're just spitballing.

Then when you get to the *Plan* phase, changes are still pretty cheap. Not free—if you need to make a change, it might influence other parts of the plan, and those will need to be replanned, or maybe more understanding becomes necessary.

Next, getting to the *Coding* phase, now changes are starting to be painful. Maybe a change involves throwing away and redoing code for thousands or millions of dollars in developer costs. Companies make changes like this on the fly all the time, though. They just do the cost-benefit analysis and decide if it's worth it.

Finally, after the code ships, now changes are *really* expensive. Not only do we have to re-plan, reprogram, rebuild, retest, and reship a bunch of code, but our customers hate the fact that we're requiring updates, and so we have all kinds of hidden secondary costs associated with the change.

From a student perspective, you don't worry so much about how much money your software project is going to cost your company. You're more worried that you'll have enough time to complete it (along with everything else—don't your teachers know you have more than one class?) with a decent grade.

So what you need to do is focus your attention on *Understand* and *Plan* where changes are cheap *in terms of time*. This will get you the best results quickly and efficiently (and hopefully by the due date) with the least programming pain.

4.8 Chapter Reflection

- What are the four stages of problem solving?
- Which of the stages involve sitting at the keyboard writing code?
- How can you make the *Coding* phase more difficult than it needs to be?

- What are the effects of an uncaught mistake made in the *Understanding* phase?
- For each of the phases, how do you know when you've completed it?
- What happens if you skip the *Reflect* phase?
- What does Beej mean by "think like a villain"?

Chapter 5

Breaking Down Problems

> *"There's games beyond the game."*
>
> —Stringer Bell, *The Wire*

If the problem solving steps (*Understand*, *Plan*, *Code*, *Reflect*) had a sequel, this chapter would be it.

Those steps really do get you through every problem, but it turns out those problems exist as fractal problems within problems within problems.

As an example, maybe you want to build a table. That's the entire problem:

- Build a table

Well, that might not be enough if you haven't built a lot of tables. So you break down the problem into dependent subproblems.

- Build a table
    - Attach legs to tabletop
        - Build tabletop
        - Build legs

And maybe that's not enough. How do you make legs? How do you make the tabletop?

- Build a table
    - Attach legs to tabletop
        - Build tabletop
            - Sand table surface
                - Glue trim to top
                    - Cut main top
                        - Learn to use a table saw
                    - Cut trim
        - Build legs
            - Cut legs
            - Turn legs on lathe
                - Learn to use a lathe

And so on. *We keep breaking down the problem until we get the step small enough that we know we can accomplish it.*

When you're first starting out, you might have to boil the problem down into single lines of code. Experienced devs often don't have to break it down that far because they are well-versed in the sub-steps.

For example, a carpenter with modest experience might only need to break down building a table into our second set of steps, above, and not go into such detail.

Like everything, breaking down a problem is a skill, and you get better with practice.

When breaking down problems, think back to our earlier consideration: "This problem would be easy if the input data were in *this* form." That's a hint that you should break out a subproblem that converts the input data into that form, thus making the problem easy.

And once you have a subproblem, pretend that it's the entire problem, just for a bit. Focus closely on it and see if you can solve it in isolation. If not, ask yourself what would make it easy to solve, and break that out into a subproblem.

Repeat.

The more you practice breaking down problems and coding solutions, the better you'll get at it. Soon you won't have to break down problems quite as far as you needed to before, and, like an expert, you'll start recognizing patterns you can reuse.

However, it's not always obvious *how* to break down a problem.

One technique is to imagine a physical manifestation of the thing you're trying to code. (For example, you're writing a sort? Imagine a bunch of alphabet blocks on a table and you have to sort them.)

And then, to push it farther, imagine that you're teaching a non-technical friend how to solve it. How would you describe the steps? The conditionals? When they're done?

If you can physically sort the books on your shelf (whatever "books" are), you can write an algorithm to do that exact same thing. You just need to break down the steps.

5.1 Pseudocode

One of the bigger tools that devs use to explore ideas is to write pseudocode. This is "code for humans". Computers can't read it. (Though some might argue that Python is pretty close to pseudocode.)

But you can use it to outline steps of an algorithm or process to do a sanity check or just explore how you might get something done.

You could write some pseudocode to insert a value into an already-sorted list of values.

```
find correct spot in list
insert the value there
```

But that's not really descriptive enough. We might have to break it down.

```
find correct spot in list
    → find first entry larger than the new one

insert the value there
    → shift all the higher values to the right
    → insert the new value in the newly-opened spot
```

Getting there.

```
find correct spot in list
    find first entry larger than the new one
        → loop through items, stop when you find a larger one

insert the value there
    shift all the higher values to the right
    insert the new value in the newly-opened spot
```

And now it's becoming a little clearer.

```
find correct spot in list
    find first entry larger than the new one
        loop through items, stop when you find a larger one
        → record the index before it

insert the value there
    shift all the higher values to the right
    insert the new value in the newly-opened spot
        → set the list item at the index to the new value
```

And we're getting dangerously close to being able to translate our pseudocode to real code. Maybe it's still unclear how we're going to shift all the values to the right, and we should break that out a bit more.

Sometimes devs add the pseudocode to their real code as comments and implement the real code under them.

This is a powerful tool to use during the *Plan* phase. It can really help solidify your thinking on the overall process.

5.2 Proof of Concept

What if you've broken down the bigger problem into smaller subproblems, but you simply don't know if one of the things is even possible to do.

For example, "Can you render an image to an HTML canvas and then save that image directly to the photo gallery on a mobile phone?" Maybe you've never done that, and you don't know if the phone and web technology even have that capability.

One sure way to find out is to code up a proof of concept.

So you make a web page, add a canvas, draw something distinctive on it, like a rectangle, and then add the code to download it when a button is pressed.

This used to involve a lot of reading books and, later, searching the web. And it still often does. But more commonly now we lean on AI[1] to answer the "is it possible and how" questions, and even come up with some proof-of-concept code.

Once you have the code working, you know two things:

1. It works!
2. How to write the code to do it.

Usually the proof-of-concept code isn't production-ready, but forms the core of what you'll eventually deliver.

Another use of proof-of-concept code is to demonstrate to people what the finished product will look or how it will behave. Sometimes people code up a mock implementation where only a small part of the UI is operational but a viewer can get the gist of how the software will eventually work.

Often the majority of the code you wrote up for the proof of concept will be thrown away, and you might feel some resistance to discarding that work. But don't worry about it. The important part of the proof of concept is the knowledge gained while doing the work, not the work itself.

> *"Plan to throw one away; you will, anyhow."*
>
> —Fred Brooks, *The Mythical Man-Month*

5.3 Chapter Reflection

- Why is breaking down a problem important?
- How far do you have to break down a problem before you can start coding it up?

[1] Again, if allowed in your work or school environment.

- What is pseudocode and why would we write it?
- What is a proof of concept and why is it useful?

Chapter 6

Right Tool for the Job

"What's the best programming language?"

It's a common question from people getting started in programming. And commonly people do have an opinion, especially when they're starting out.

But anyone who has been in the field a while will tell you that there is no "best" programming language without knowing the context of the problem you're solving.

I don't know anyone who would declare that Bash scripting[1] is the best programming language. But there are definitely problems for which is actually is the best solution!

Let's analogize! What's better: a screwdriver or a hammer? Trick question! They're both the best tool, but only in the context of the job they were designed for.

Different programming languages have different strengths and weaknesses. It's your job as a developer to choose the hammer for nails and the screwdriver for screws, and to be proficient with both those tools. "I'm not familiar with the table saw and I don't really like it, so I'm going to use this hammer to cut that plywood" is not the solution your boss is looking for.

> **My scuba instructor had some advice for me.** I was looking at buying some kit and was deciding between standard fare and > DIR[a]-style equipment.
>
> He said, "Be the best diver you can be in any equipment. I don't care if you're diving with a 1960s horse collar and J-valve tanks."
>
> [a] https://en.wikipedia.org/wiki/Doing_It_Right_(scuba_diving)

The best developers have a lot of tools in their belt, and they know how to use them.

So when you find yourself saying, "This language sucks and I hate it compared to that other language that I love!" consider the use cases for that language that you hate. Because it was created for a reason, and identifying it can let you know when you *should* use it.

And since you're dying to know, my favorite language is Rust.

Or Python. Or C. Or JavaScript. Or SQL. Or AWK. Or Bash. Or… well, it just depends on what problem I'm solving!

6.1 Be Opinionated

Didn't I just finish telling you that you should give all languages a fair shake and not play favorites?

Yes. But I swear this doesn't contradict that.

Why be opinionated? It's because you should be deliberate and well-informed in the choices you make. And a good way to make that happen is to be opinionated with a rationale. It forces you to dig into the

[1] https://en.wikipedia.org/wiki/Shell_script

strengths and weaknesses of various technologies, and that helps you choose the best tool from your kit for a particular job.

None of this is meant to imply that you should be a jerk about it. It's not about who's right or wrong—nowhere is any of this written in deific stone. Everyone is both right and wrong at the same time. It's very Schrödinger of us.

You should be opinionated about the tools that you choose. Out of everything at your disposal, you chose *this* language and build system to use. You chose *this* editor or *that* library. You should have a rationale for *why* they are the best, even though plenty of other languages or tools would have sufficed.

You should be opinionated about the data structures you choose. Coding is a creative endeavor—you can make everything a Turing machine[2] if you wanted and solve the problem that way, but there are a lot of other options. Choose the best ones and have a rationale for *why* you chose them out of all the other options available.

You should be opinionated about the algorithms you choose. Mergesort, right? Just use that. But there are actually times that lowly insertion sort is better! Binary search? Why use it if linear search is good enough for the job? Have a rationale as to *why* you chose that algorithm when you could have used another one.

When deciding, you might be considering things like performance, or maybe readability of the code, or development time required, or tool cost, or what tech the existing codebase uses, or all kinds of other factors.

You must always be willing to learn more. No matter what you choose with your rationale, half the Internet is going to disagree with you. And you know what? Your future self might even disagree with you!

As you learn more, you will learn specific cases where some particular technology is best, where before you thought it wouldn't be. Be open to learning, and change when appropriate. This can be something as big as the choice of framework for the product that defines your entire company, or something as small as whether or not array variables are plural.

With growth, your choices will change. And, as long as you have a rationale behind it, that's a good thing.

6.2 Chapter Reflection

- Describe a typical experienced dev's position on which programming language is best.
- Why be opinionated about tech?
- What are some factors to consider when deciding which technology is the right tool for the job?
- What are some factors to consider when deciding which technology is the right tool for the job that aren't listed in this chapter?
- Why is continuing your learning so important when choosing technologies?

[2] https://en.wikipedia.org/wiki/Turing_machine

Chapter 7

Hacks and Techniques for Learning

There are a number of tips and tricks for maximizing your speed of durable learning. Devs know these help, and yet we stubbornly ignore them all the time.

But just in case you want to get faster at learning, here are a few things that work for some people. No guarantees; everyone's different, and you might have your own path that works better.

> **Music** is one of those things. Some people swear by silence or white, pink, or brown noise. Some people listen to classical music, or electronica, or metal. Do what works for you.

7.1 Flow

Flow[1] is a mental state you get in where you're focused and ideas are connecting freely. There are no interruptions.

Programmers like to get in flow for maximum productivity.

Here are the characteristics of that state stolen directly from Wikipedia:

1. Intense and focused concentration on the present moment
2. Merging of action and awareness
3. A loss of reflective self-consciousness
4. A sense of personal control or agency over the situation or activity
5. A distortion of temporal experience, as one's subjective experience of time is altered
6. Experience of the activity as intrinsically rewarding

You've probably already experienced this in some aspect of your life. Keep in mind that it can be very beneficial for programmers.

7.2 Reading Ahead

I know when I was a student, if something was due on Sunday, I'd commonly read about it for the first time on Sunday. Anyone else do that? Yeah.

But here's another idea: read the assignment as soon as you get it. Maybe you don't know enough yet for it to make sense, but that's okay. Just read it, maybe not even that closely.

And another idea: read the assignment when you've done some of the other reading and lecture, but days before you plan to start working on it. Again, just read it, not worrying about solving it. Maybe read it right away **and** partway through the week!

What this does is prime your brain with the information. You won't retain it or understand it all, but your brain will start chewing on it in the background and will make the project easier to tackle when you finally get around to it at 9 PM Sunday night.

[1] https://en.wikipedia.org/wiki/Flow_(psychology)

And when you do start the four-stage problem solving process, the material will seem less foreign and more approachable.

A powerful variant of this is to complete the *Understanding* phase early. Don't need to *Plan* or *Code* it up yet.

7.3 No Copy-Paste Coding

You need to solve a problem, and look right there on the Intertubes! There's a solution! Time to break out the ol' `CTRL-C`/`CTRL-V` skills!

It used to be that people frequently found this solution on the programming site Stack Overflow[2]. And still do. But nowadays they tend to punt to some AI.

Beginning developers should not do this. Remember the main goal: develop excellent problem-solving skills. Copy-paste coding does **nothing** to further this goal.

One exception to this rule is if you already struggled with the problem for some time, as mentioned in the *30 Minute Rule* section, below. But even then, you **must** understand the code you're copying 100% completely before using it.

7.4 The 30 Minute Rule

If you've been stuck for 30 minutes and you've **really** tried to attack the problem from a variety of directions and have still come up dry, it's time to reach out for help.

Yes, you can go longer than 30 minutes without help (we all do), but 30 minutes is a good balance between working hard and bring productive with the limited time you have for schoolwork.

Here's a story. One of my favorite books for learning about programming is called *The Structure and Interpretation of Computer Programs*[3], or SICP for short. (I particularly like the Scheme version—really helps you learn recursion.)

The programming problems in that book can be really challenging. But I gave myself a six hour time limit per problem. (I didn't do all six in a row, usually.) After the time was up, I looked at the answer.

And here's what that does and why it's important: while you work hard on a problem, you're busy building a mental framework around it, trying to support it. And if you get the answer, great! But even if you don't, *you've still built that framework.*

So when you give up after your time limit, very often the solution you read fits neatly into the framework you've already built. It's just a tiny step from your framework to the solution, and it's *way* easier to take that little step to the solution than to try to grok[4] the entire thing.

Contrast that to when you just look up the answer immediately without building the framework. The solution has nowhere to "sit" in your brain, and doesn't connect to anything else. And it's a much bigger step to achieve understanding.

The struggle is vital! Don't skip the struggle! But at the same time, don't keep it up forever. When you timeout, ask for help from a peer, a tutor, an instructor, or (if allowed) an AI.

Also, maybe follow this up with going for a walk.

7.5 Go for a Walk

Yes, I'm serious. You're stuck, and nothing seems to be working, and no additional plans of attack are coming to mind. What do you do?

Go for a walk.

[2] https://stackoverflow.com/

[3] https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs

[4] https://en.wikipedia.org/wiki/https://en.wikipedia.org/wiki/Grok#In_computer_programmer_culture

As you try different approaches to a problem, it's like you're leaving ruts behind in the road, and your brain tends to focus on the existing ruts rather than trying something new. You're locked in on the approaches you've tried but aren't working. "If only I could just tweak this one thing that almost works…" but you're not seeing how.

Stand up and stroll. Maybe you're at work and this just means you're making laps in the hallway. Or maybe you can go outside on a balcony, out front, or on the roof.

This frees your mind to get out of those ruts and explore new approaches.

There have been times where I've decided to go for a walk and have gotten two steps out the door when a new approach to the problem has occurred to me.

This method of getting unstuck is tried-and-true.

7.6 Rubber Duck

Talk to someone about the issue. This is such an effective technique that it even works if you're talking to an inanimate rubber duck, giving rise to the name *rubber ducking*.

The basic idea is that you're going to lead the other person through the problem-solving steps, effectively teaching it to them. Get them to understand the problem, and have them help with a plan.

Here's the really amazing thing about this: *it works even if the other person (or duck) is non-technical*.

One of the reasons is that to understand a problem and come up with a plan, you really don't need to know anything about programming. They can still help.

And here's the *really* amazing thing: they don't even have to say anything. The mere act of teaching someone about the problem is very often enough for you to find the answer on your own. Maybe it was a piece of understanding that you missed, or there's a non-obvious hole in your plan. Talking it through can help you find these things.

One time, in a combination of going for a walk and rubber ducking, a coworker of mine walked up to my cube, raised his hand as if to ask a question, paused a beat, then said, "Never mind, I figured it out." That was all it took.

7.7 Write Down Questions

When poring over a problem description or learning a tool or language, there are basically two kinds of questions that crop up.

1. **Blocking questions** are questions that you need an answer to right now because they're blocking your progress. You can't do anything else until you get the answer.
2. **Non-blocking questions** are things that come up of the course of development that are interesting, but you can keep going without knowing the answer right now.

I like to write down non-blocking questions and get answers to them later. Things like, "Does this language support destructuring assignments?" or "Can the library also provide random numbers in an integer range?" or "What other networking protocols are built into the standard library?"

They were things that I was curious about, but didn't need to know the answer to immediately.

Coming back and getting the questions answered later can help build a more complete picture of the systems you're working with and make you a more effective developer.

7.8 Build a Tapestry of Knowledge

This is where it all comes together.

When you first start coding, you're in the middle of the vast unexplored world of knowledge. You've learned how to print `Hello, world!` on the screen, but that's it.

So you start mapping it out. You see that there are functions and variables and I/O operations and you see how those are connected. And you learn about networking and see how that's connected to the I/O system in the OS, and you connect them on the map.

As your map grows, you draw connections between many of the things you've learned, and you gradually see that the world of development is more interconnected than not. A lot of problems are very similar to a lot of other problems.

And when you know a lot of problems like that, that's a lot of power you can bring to bear on new challenges you face. "Oh, this problem *x* reminds me of problem *y*. Maybe I can solve it in a similar way."

> **You have a group of 10 people numbered 0 to 9** and they are all lining up at a bank window. Your simulation needs them in random order with no repeats in $O(n)$ time. How do you code this up?
>
> Maybe earlier you'd written a program to shuffle a deck of cards using the famous Fisher-Yates algorithm[a]… wait! All you have to do is make a list of people numbered `0` to `9` in order, then shuffle the list like a deck of cards!
>
> It's the same problem!
>
> [a] https://en.wikipedia.org/wiki/Fisher–Yates_shuffle

Beginning developers solve programming problems through sheer logic and reasoning.

Experienced devs also use logic and reasoning, but they primarily rely heavily on pattern matching. What coding pattern do I know that best solves the type of problem that I'm currently facing?

In short, they rely on their interconnected tapestry of knowledge they've built up over their years of programming.

As you learn to code, look for ways that the thing you're learning about now connects to the rest of the programming world you've already explored. Make those connections so you can exploit them later.

7.9 Get and Give Code Reviews

It can be really hard to put your code out there and ask someone to give you advice on how to make it better. It's easy to take that advice personally.

But fight that urge, and treat every code review you get like a gift. Someone was willing to spend their time helping you become a better coder, usually at no cost to you.

> **Be nice during code reviews** no matter if you're the reviewee or the reviewer. Be supportive with your feedback, be modest, and don't take negative feedback personally.

And if you can't find a human to help, feed it to an AI chatbot and ask it to review your code.

Even if you're not well-versed in the subject matter, that shouldn't stop you from giving a code review if you can. Just reading other people's code exposes you to different coding styles and algorithmic patterns that you might not have been aware of.

But remember to always be opinionated about the feedback you get, and use critical judgment when deciding whether or not to incorporate it.

7.10 Join a Club

Coding has historically and generally been a solo endeavor—at least the part where you're sitting at the keyboard typing things. And even during the *Understand* and *Plan* phases it can be tempting to work alone.

Joining a like-minded group of people where you can bounce around ideas (or even just vent) can really help your brain get unstuck from ruts and help you approach problems in ways you hadn't considered or even been aware of.

It's also good for networking, and lots of clubs have interesting presentations you can attend. Or even better, present at!

Clubs can make the fight feel a lot less lonely. We're all in this together.

7.11 Chapter Reflection

- What do you (personally) do to get into *flow*?
- What are the advantages to reading an assignment way before you intend to work on it?
- What are the disadvantages of working on a problem for too short a time before asking for help?
- What are the disadvantages of working on a problem too long before asking for help?
- When is it acceptable to use copy-paste coding? What happens if you use it unacceptably?
- Why is going for a walk a good idea when stuck?
- How can an inanimate rubber duck be a good programming partner?
- What does the author mean by *tapestry of knowledge* and how does it relate to your skill as a developer?
- What do you gain by getting a code review? What do you gain by giving one?

Chapter 8

Debugging

Before we begin, the best way to debug a program is to not have bugs to begin with. Though we're only human and we'll certainly mak mstakes, the best way to avoid bugs is to adhere to the problem solving framework. Remember that the programming battle is in the *Understand* and *Plan* phases. The more completely and correctly you complete those phases, the fewer bugs you'll have when coding it up.

That said, let's talk about what to do when the inevitable happens.

8.1 Mental Model

This is one of the most important things about being a developer: *have a mental model of computation*.

That is, you should be able to read code and know what's going to happen.

```
def foo(n):
    i = 0

    while i < n:
        i = i + 1 + (i % 2)

    print(i)

foo(5)
```

Read the nonsense Python code, above. Mentally compute the output. Then see if you're right. (I wrote the code, and it still took me a solid minute to mentally model the answer. But I was right!)

If you cannot "run" the code in your head, you cannot debug. Yes, I'm going that strong. I'm sure some people disagree with me, but I want to drive home how important this is.

Debugging is the art of finding the part of the code where your mental model of the computation and the reality of the computation diverge. And then fixing it.

If you don't have a mental model, you don't have anything to compare against and you'll make little progress.

How do your improve your mental model of computation?

- **Study code**. Trace through it. There's a ton of code out there to practice with, e.g. on GitHub, on HackerRank, your peers' codebases, your own stuff that you wrote four months ago and have forgotten how it works, etc.
- **Predict the output**. As you learn the code, try to predict how it will behave when run.
- **Manually trace a run**. Use a whiteboard to manually track what values variables take on, what functions get called, and what line of code is executing.

- **Write a specification**. Study some code and "reverse engineer" it. Figure out what it does, then write a human-readable spec that perfectly describes the algorithm or codebase to the degree that a reader could reimplement it from scratch.
- **Single-step through with a debugger**. Have the computer show you how the program is flowing. We'll talk more about that, below.

You'll definitely improve this skill with practice.

8.2 Reproducing the Bug

First things first: see if you can get the bug to happen consistently. Being able reproduce ("repro") the bug is the first step in being able to squash it.

Sometimes this is actually the hard part. You saw something go wrong once (or someone reported they saw it), and now you can't repro it.

> **At this point you might be tempted to utilize a programming technique known as "prayer"** in that you're praying that either you didn't really see it or the bug will never rear its ugly face again in this world. But here's the unfortunate truth: if someone saw this rare bug *just once* in testing, thousands or millions of people will see the bug when it goes into production. Murphy's Law *and* statistics? You have no chance against that.

If you can't repro it, you'd just be shooting in the dark trying to fix it. Your goal at this point is to just get it to happen at all. Work backward logically. What *must* have happened in the program flow in order to see what the bug reporter saw? Look there first. Even if you're sure some condition *can't possibly* be true, if it must have been true to see the bug, then it *must* have been true! Keep tracking back, looking for where your mental model and the code diverge and use that to repro it yourself.

If you can only sporadically repro it, you have a chance, but it's going to be hard work. Your goal is to get it to repro consistently. This saves you spending forever trying to get a rare repo. If you could make it happen consistently, that's a big time saver, and it helps you narrow down where the bug can be.

Once you find how to repro it consistently, now you can get even more systematic about things. You want to find the minimum number of steps that can cause the bug to manifest. For example, you're playing a game and you find a bug when you complete 20 laps around the race course and then drive into a tree. Maybe try just driving into the tree first. If you're lucky, you saved yourself 20 laps and narrowed the bug down to the tree. If nothing happens, you know that the 20 laps is an integral part of the bug. Maybe try a single lap followed by the tree. Does that repro?

Getting the minimum number of steps not only helps you narrow down even further where the bug is in the code, but it makes it easier to test fixes because you don't have to spend so long reproing the issue.

8.3 Finding the Bug

There is a bug somewhere. You know that because when you give your code some input, and it cranks away at it, eventually it gives you unexpected output.

Somewhere in that big process the reality of the computation diverges from your mental model of the computation.

At first, all you might know is that somewhere in 10,000 lines of code something went wrong. So you've narrowed it down to that. Your mental model said that if the input was `2`, the output would be `3490`. And instead the output was `299792458`.

Therefore you know the bug is somewhere between the input and the output.

You *could* just start changing random things in the code to see if it gets fixed. This is sometimes called *shotgun debugging* or *prayer debugging*, and it very, very, very, very, *very* rarely works. Way more often you'll just mess things up and make the problem even harder to find. It's like trying to fix your car's electrical issues by randomly adding and cutting wires. It's not even worth debugging this way.

And yet despite that, it's a very common technique practiced, in vain, by students worldwide. You, however, should not use it.

Instead, it's time to be systematic. Somewhere in that pipeline of computation the intermediate computed values diverge from your mental model. Your job is to find out where.

So you start probing *inside* the program at various points to see where things go off the rails. Binary search is great—jump to the middle of the process and examine the values during a run (see below). If they are as expected, the bug must therefore be between the middle of the program and the end! You've just halved your search space. Now do it again until you narrow it down far enough to see the bug.

I would contend, though some might disagree, *the bug is not found until you understand it.* That is, you **must** understand exactly how your program was giving the output `299792458` instead of the expected `3490`. Gaining that full understanding has a number of benefits:

- You can be more confident you've fixed the bug for-realsies.
- You will learn to recognize the patterns that led to this bug, allowing you to better avoid it in the future.
- You're working out your problem-solving skills while you do this.

Once you understand how the wrong output was produced, then decisively and correctly fix the issue, and know why the fix will work.

Finally, if you're just filing a bug report (i.e. someone else will fix it), being able to give them the minimum steps needed to repro will make you their hero for the day. Consider it from the reverse perspective; would you rather fix a bug with a vague, long sequence of steps to repro, or one with a few steps that caused it to repro every time? The more specific things are, the happier the bug-fixer is, whether that's you or someone else.

8.4 Print Debugging

The good old-fashioned standard way of probing software in the middle of a run is called *print debugging*, or, if you're a C programmer, *printf debugging* (pronounced "print eff").

This is basically just tactically placing print statements inside your code to see what state your program is in.

There are a couple common uses of this:

- Print anything at all to see if some part of the code actually executes.

  ```
  print("A")

  x = foo()

  print("B")

  if x == 3:
      print("C")
      x *= 2
  else:
      print("D")
      bar()

  print("E")
  ```

  Notice that when I run the code, I can see how far it gets before a crash, and I can determine if `x` were `3` or not.

- Print some specific values to see what they are.

Here's an example where we're getting data from some sensor in a loop for processing. We suspect that some of the data is wonky (maybe the sensor is busted) so we're printing it out to see what we get.

```
while not done:
    data = get_sensor_data()

    print(f"Got sensor data: {data}")

    process_sensor_data(data)

    done = data < 0
```

> **Don't f---ing use profanity in your debugging statements.** Murphy's Law says that if you do use profanity, you'll forget to take it out, and even though it was in some part of the code that you're certain will never run, it will inevitably pop onto the screen while you're doing a client demo with your boss on the day he's assessing you for a raise.
>
> I know as well as anyone how infuriating programming can be. And when I'm feeling that way and forget to take a deep breath and recenter, I print this:
>
> ```
> print("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
> ```
>
> Not only does it stand out nice and clear on the screen, but it's really easy to type in frustration and helps dispel some of my bad energy. And if the client sees it, it's a minor transgression.
>
> Another friend of mine suggested printing various non-offensive emojis, as well, which seems like a fun way to diffuse ones frustration.

Now, print debugging is kinda frowned upon as a lesser means of debugging compared to using a real debugger (as in the following section). But everyone does it at some point or another, and some people even swear by it.

The place that I think it really shines is when you need to gather a lot of data about the run to see a larger pattern emerge, or when you need to catch an infrequent event. If something happens one run in 10,000, single stepping through with a standard debugger is going to take forever. You can add some print statements and script a run 10,000 times and watch the output to see when it manifests.

One thing to watch out for is that if you're printing a lot, it can be tough to visually parse the output, and error messages might be lost in it. I'd recommend redirecting the output to a file and then bringing it up in an editor to search.

And, finally, don't forget to remove all your print statements before you ship your work!

8.5 Debuggers

Debuggers are tools that help you find bugs. There are many different debuggers, but virtually all of them share a common set of features. The main features are:

- Add *breakpoints* where the program will stop running and you'll get control in the debugger
- "Single step" through a program a line at a time
- Examine the values of variables

Additional features that are common are:

- Stepping into a function
- Continuing out of a function
- Stepping over a function
- Setting the values of variables
- Examining the call stack
- Setting breakpoints that trigger conditionally

Rarer are *time-travel debuggers*. In addition to allowing you to step forward through your program, they allow you to step backward, as well! This is great if you step past the bug by accident and want to step back to see it.

All major IDEs[1] have debugger functionality. (It's part of what's "integrated", the "I" in "IDE".) There are also standalone debuggers that you can run. And all mainstream languages have some kind of debugger support.

As you might imagine, with those features, debuggers are really powerful.

If you suspect a bug in function `foo()`, you can set a breakpoint there, run the code, and then get control of the debugger when `foo()` executes. Then you can step through it a line at a time, looking at how the values of variables change. And there's no need to add any print statements.

Note that in VS Code, getting your debugger set up might be trivial, or you might have to edit some esoteric JSON files to get it going.

In any case, learning to use a debugger is a valuable skill that can save you massive amounts of time while you're trying to track down that pesky gremlin in your code.

8.6 Chapter Reflection

- What is the *mental model of computation* and how is it important for debugging?
- Why is finding a bug's minimum reproduction case an important step in fixing that bug?
- How do you narrow down where a bug is in your code?
- Contrast print debugging versus using a debugger. What do you think the relative strengths and weaknesses are?

[1] https://en.wikipedia.org/wiki/Integrated_development_environment

Chapter 9

Learning a New Language

The first language you learn is the hardest. Not only are you learning the language, but you're also learning the *concepts* that are used within the language. By concepts, I mean things like how to organize functions and pass arguments, how to run loops, how to do conditionals, etc.

All languages have these same concepts (more or less—more on that later), and so learning the second language is just learning how to apply those same concepts you already know.

It's like if you already know Spanish, learning Italian isn't that big of a jump.

This chapter is about learning additional languages after you've learned your first one. This is important because you're going to be learning new languages your entire career. Luckily, though, learning new languages it itself a skill, and the more new languages you learn, the easier it becomes.

There are two big pieces to learning a new language in a paradigm you already know (procedural, object-oriented, functional, etc.)

1. **Learn the syntax**. Like `if` and `while`, and how to declare variables and functions, etc.
2. **Learn the standard library**. This is the built-in functionality that you can take advantage of, like reading and writing a file, or printing to the screen, or connecting to a web server.

Learning the syntax is often the easier of the two. Most languages have relatively simple syntax.

By analogy, you can learn what verbs, nouns, and adjectives are, and how to diagram sentences[1]. But that's not enough to write a masterful literary work. You also need to know what words you have at your disposal.

And that's the more complex part. Many standard libraries have a *lot* of built-in functionality. Scroll through the Python standard library for an example[2].

9.1 Learning the Syntax

Jump right in! Follow a tutorial and write "toy" programs. These are programs that just exercise some aspect of the language.

How do we do conditionals in Rust? Let's write a toy program to check it out.

```
fn main() {
    if (1 == 2) {
        println!("Something is horribly wrong.");
    } else {
        println!("That's correct.");
    }
```

[1] https://en.wikipedia.org/wiki/Sentence_diagram
[2] https://docs.python.org/3/library/index.html

```
}
```

Wait—are those parens necessary around the `if`?

```
$ rustc foo.c
  warning: unnecessary parentheses around `if` condition
```

No, they aren't! It's a toy program; we're just using it to learn.

To learn all the necessary syntax, take the concepts you already know and look up how to apply them in the new language.

- The main function
- Variables
- Conditionals
- Loops
- Classes
- etc.

It will be frustrating to you at first because you have to look up every. Single. Thing. Like with a new human language, you know conceptually that you want to go to the supermarket, but you have to look up all those Italian words if you don't know Italian.

The good news is that the syntax of a computer language is *way* simpler than a human language. And you can get it down pretty quickly.

9.2 Learning the Library

The standard library is the pre-baked functionality that ships with a language. It's nice because you know everyone who has the language installed has all these functions already and they don't have to download any additional third-party dependencies.

But you need to be familiar with that language's standard library so that you know what the language can do out of the box, and so that you don't reinvent the wheel when you don't have to.

One recommendation is the skim the standard library for a language you're using. You don't have to know exactly how to use Python's IMAP[3] functionality, but knowing it's there in case you do is very valuable. At the very least, it lets you know that Python is a contender for choice of language if you need to do some IMAP work.

Then when you do need some of that functionality, you can dig into the documentation and examples and see how it works.

I tend to learn libraries piecemeal, learning in detail only what I need to get a job done. I know the rest of what it *can* do (because I skimmed the docs), but I only know bits and pieces well enough to code with them.

And that's okay, since the libraries are massive, and it's unlikely you're going to achieve mastery of everything in them. You just need to be able to learn what you need to complete your work.

9.3 Learning a New Paradigm

First, what is a *programming paradigm*? It's a way of modeling a problem so that you can come up with a solution. I know that's vague but bear with me for a couple paragraphs.

Imagine doing your taxes. (Sorry.) When you do them, it's a sequence of steps one after another. Fill in your name. Fill in your income. If your income is more than some value, do *x*. Else do *y*. It's a procedure that you're following. You can model it as a series of steps.

[3] https://en.wikipedia.org/wiki/Internet_Message_Access_Protocol

Imaging you're simulating a 3D fantasy world. In that world you might have a type of creature called an orc, and there might be many creatures of that type running around. And they all have their own independent coordinates, and their own hit points[4], but they all have the same behavior when you walk up to them. You can model them as a collection of objects that are independent but have similar behavior.

These are two different ways of solving a problem, either by modeling them as a sequence of steps, or as objects.

We call these differing models *programming paradigms*. The first example is "procedural programming" (kind-of; I'm hand-waving a bit), and the second is "object-oriented programming".

There are a lot of paradigms[5], but the Big Three are procedural, object-oriented, and functional.

Here's the bummer: learning a new paradigm is hard. A lot harder than just learning another language in the same paradigm.

If you know Spanish, learning Italian is relatively easy. But learning Chinese, that's something else! Not nearly as easy. Keeping with the analogy, it's a different paradigm. You have to learn new techniques and concepts you might not even be aware of from the romance languages.

> **I learned Erlang a while ago**. Erlang[a] is a functional language, and I was weak with the functional paradigm.
>
> For example, in Erlang, once you set a "variable", you can never change it. And every single way I knew for modeling problems involved changing variables!
>
> I mean, how are you supposed to get *anything* done if you can't change a variable?!
>
> But clearly, massive systems had been implemented successfully in Erlang, so there was a way. But I had to change my thinking about how I modeled problems, and learning that new way was a significant challenge.
>
> [a] https://www.erlang.org/

My main piece of advice here is to use a *lot* of examples to see how that language performs basic tasks. That is, gather and study a lot of toy programs.

Then come up with related challenges (or find some online, or ask an AI to generate some) that allow you to work out to build your skills and find gaps in your understanding.

9.4 Chapter Reflection

- Why is learning your first programming language more difficult than learning the second?
- What is the difference between learning syntax and learning a library?
- What's the difference between a programming language and a programming paradigm?
- Why is learning a new paradigm commonly more difficult than learning a new programming language in a paradigm you already know?

[4] https://en.wikipedia.org/wiki/Health_(game_terminology)#Hit_points
[5] https://en.wikipedia.org/wiki/Programming_paradigm

Chapter 10

Use of AI

As of this writing in 2025, AI is the new hotness[1]. Everyone is using it, and no one can stop talking about it.

The question no one seems to have an answer for is what's going to happen. Sam Altman[2] and others will tell you that AI is any second now going to take over the world and solve everything and humans will be rendered obsolete.

I like to think that will not happen, and that even if AI starts solving everything, people will still want to use their ingenuity to push the envelope farther than AI is able to.

What does all that mean for you as a computer science student (as of this writing)?

Let's talk about how you should use an AI as a student and at work, because those are two different things.

But before that, let's talk about what you're *not* supposed to do.

10.1 How *Not* to Use AI as a Student

I've mentioned elsewhere that I like to study the book SICP[3] to improve my skills. And how I give myself a six-hour time limit to solve the problems presented.

Now, these problems aren't real-world problems at all. They're contrived training problems. And—get this—the answers are all freely available on the Internet in a wide variety of places.

So why do I spend up to six hours? What a waste, right? Why not just clone someone's repo and declare that it's implemented so I'm done?

Or, if not that, why not just copy what my friend made?

Or, if not that, why not hire someone to write the answers for me?

Of course, you know the answer. I don't do that because if I do, I haven't learned anything. More specifically, *I haven't struggled with the problems* so the learning that would come from that is lost.

I think you see where this is going: just asking AI to solve the problem isn't going to increase your skills at all. It's just asking someone else to do the hard work.

It's like I went to the gym and asked someone else to lift the weights for me. Sure, I can walk out saying the weights had been lifted successfully, but I gained nothing from the experience.

I want you to name an activity that, aside from being at the gym, involves lifting weights all day. Answer: there are none (generally speaking for the majority of the population). Then why do we spend time lifting dumbbells at the gym if there are zero other activities that involve it?

[1] Unlike the expression "new hotness", which is now tired. "Tired" is also tired, but that's self-referential so I think it has bottomed out.

[2] https://en.wikipedia.org/wiki/Sam_Altman

[3] https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs

The weights, of course, are just tools we use toward the greater goal of being generally stronger.

School is exactly like this. The programming problems you get in school are dumbbells. They're not real. They're designed to give you a workout so that when you get to the job, you have the strength to do the work.

And because the problems aren't real, AI can solve them all *really* easily. There's tons of training material out there for them to learn from.

But don't be fooled. Just because AI can solve your school problems doesn't mean it can solve the real-world problems you're going to face in your work. (As of now, it can't.)

(And if it could solve all those problems, how much do you think devs would earn? There's a reason being a dev pays well, and it's because the work is hard. If it was as easy as typing in an AI prompt, it would pay minimum wage. That should clue you in to the fact that if *all* you can do is prompt AI, you're not getting a high-paying gig.)

But that's not to say you shouldn't get good at using AI; it's just that while you're a student, you have to use it the right way to maximize your skills development.

The TLDR of this section is this: never ask AI to solve your entire programming project. It probably can do that, but you'll learn nothing. The goal of the project is not to complete the problem; it's to get a workout while you complete it.

10.2 How to Use AI as a Student

First things first: if your school or instructor has banned AI, that's the rule. And you have to ignore what I've written here. Sorry. I'm going to assume it hasn't done that foolish thing, and proceed.

So how should you use it? Use it like you're working with a decent tutor who knows a lot of things kinda well. The AI tutor definitely makes mistakes and gives poor advice from time to time, so you should cast a critical eye on everything you learn from it.

When you're stuck on a small part of a project, ask about that. When you can't remember syntax, ask about that. Ask about the idiomatic way to write a loop that removes elements from the array it's iterating over. Ask about what a particular operator does or how to use it. When you have questions about the language or library, ask those. Little bite-sized pieces are fine to ask it about. It can be way faster than a standard Internet search.

When you've completed the project (and it works completely), then you can feel free to ask AI for a solution that you can use for comparison. Or, better still, feed your solution into the AI and ask for improvements.

Keep in mind that some of the "improvements" aren't going to be improvements at all, and you'll want to ignore them. Cast that critical eye. Be opinionated and have a rationale for which advice you're accepting and which you're rejecting.

Lastly, there's another time it's fine to use AI to solve entire projects: when your instructor says you can. This happens when they're giving you more real-world practice as opposed to lifting weights. Both are valuable uses of your study time.

10.3 How to Use AI at Work

First, a caveat: the last time I was working in production (and I did so for 20 years), AI as we know it today did not exist. That said, I do use it to get things done more quickly today. So that's my level of "expertise".

I'd mentioned earlier that new devs solve problems with logical reasoning, but experts, in addition, recognize patterns. As a more-experienced dev, you have a better understanding of which building blocks make up a problem. You recognize the pieces that you need, and you logically reason about how to assemble them.

In other words, experienced devs are better at *Understand* and *Plan*. (They're better at all phases, but recall that *Understand* and *Plan* is where the battle is.)

As such, they can leverage AI to help them write the code for those building blocks. They can say things like, "I need to filter those results for anything that matches this regular expression"—and then they ask an AI to code up that building block, they expertly verify that the code is correct and modify it to suit their needs, and then move on.

Even with technologies they aren't familiar with, this can help them get the job done. But they still need to rely on their expertise to know when they need to learn more. That is, experienced devs have a nose for dangerous code and know when they need to proceed carefully and gather more knowledge.

In this regard, AI can be really useful for doing proof-of-concept work and rapid prototyping where the code is often throwaway.

> *"Let us hurry! There is nothing to fear here!"*
> *"That's what scares me."*
>
> —Satipo and Indiana Jones, *Raiders of the Lost Ark*

Again, as a student, you can't bring that experience (that you haven't yet acquired) to bear, and if you just try to use AI like a seasoned dev, you're going to have buggy, fragile code that you don't know how to fix. And worst of all, you won't be developing the skills you need.

But as you gather more experience, you can definitely rely on AI to write a large amount of boilerplate code for you that you already know the logic behind, anyway.

10.4 AI and the Jobs Market

When you read this, I want you to know that I, the author, thought Yahoo![4] was a dumb idea when it first launched. I kinda still do think it was, but it made a bazillion dollars since then, so I was wrong. At least capitalistically.

One thing old devs have been hearing their entire careers is that we're on the verge of the "no code" revolution, and that this tool or that tool will finally put coders out of a job because everyone everywhere will be able to produce software.

Every one of these predictions has something in common in that they were all fantastically wrong.

But things are only wrong until they aren't, and LLMs like ChatGPT are definitely novel beasts that don't play by the previous rules.

> **I do recognize the stock pumping, though.** All those "no code" companies were talking a big game trying to get big returns for their investors. OpenAI and other AI players also talk that big game.
>
> That's not to say they won't realize those gains; but it is to say it smells familiar.

And LLMs are showing incredibly coding expertise. I am perpetually amazed by what they can do. But can they do it all?

I have a thought experiment for you. Let's say there's an AI so good that I can tell it, "AI, design and implement a corporation that will crush all my competitors and make me the richest person on Earth," and it will actually successfully do it.

But the catch is that everyone has access to the same AI and they can all make the same request. Where does that land us? We're back to square one where we're on equal footing.

As a capitalist, though, I don't like equal footing. I want to get an edge on my competition. So I start thinking, "What can we do that's slightly different than what the AI is telling my competitors?"

And just like that, humans are back in the game!

I think that trend's going to continue, maybe forever.

[4] https://www.yahoo.com/

What will happen, I predict, is that the easy boilerplate jobs that exist now will experience a massive tightening. AI can solve lots of those easy problems, and you don't need a big team of engineers behind them. The more novel problems will still need a lot of human work.

But maybe not as much work as before. AI can help in a variety of ways, outlined above, so it helps speed things up.

Going back in time, consider when most programs were written in assembly language[5] and it took a lot of specialized knowledge to get these error-prone programs written. And then compilers became popular and now no one[6] writes in assembly any longer; those jobs are toast, destroyed by the faster, easier coding that compilers afford.

In short, I think we're going to keep pushing it, and AI will become a very useful tool, but only with humans at the helm. I think. We shall see.

> *"And… Always look on the bright side of life…"*
>
> —Lead Singer Crucifee, *Monty Python's Life of Brian*

10.5 Chapter Reflection

- Contrast the useful and non-useful ways students can use AI.
- What goes wrong if you use AI badly as a student?
- How do things differ with AI at work versus at school?
- What are some of the hazards of using AI at work?

[5] https://en.wikipedia.org/wiki/Assembly_language

[6] Well, a few people do. You should try it for fun. It's something else.

Index