3.3 — A strategy for debugging

When debugging a program, in most cases the vast majority of your time will be spent trying to find where the error actually is. Once the issue is found, the remaining steps (fixing the issue and validating that the issue was fixed) are often trivial in comparison.

In this lesson, we'll start exploring how to find errors.

Finding problems via code inspection

Let's say you've noticed a problem, and you want to track the cause of that specific problem down. In many cases (especially in smaller programs), we can quickly home in on the proximity of where the issue is.

Consider the following program snippet:

```
int main()
{
    getNames(); // ask user to enter a bunch of names
    sortNames(); // sort them in alphabetical order
    printNames(); // print the sorted list of names

    return 0;
}
```

If you expected this program to print the names in alphabetical order, but it printed them in opposite order instead, the problem is probably in the *sortNames* function. In cases where you can narrow the problem down to a specific function, you may be able to spot the issue just by looking at the code.

However, as programs get more complex, finding issues by code inspection becomes more complex as well.

First, there's a lot more code to look at. Looking at every line of code in a program that is thousands of lines long can take a really long time (not to mention it's incredibly boring). Second, the code itself tends to be more complex, with more possible places for things to go wrong. Third, the code's behavior may not give you many clues as to where things are going wrong. If you wrote a program to output stock recommendations and it actually output nothing at all, you probably wouldn't have much of a lead on where to start looking for the problem.

Finally, bugs can be caused by making bad assumptions. It's almost impossible to visually spot a bug caused by a bad assumption, because you're likely to make the same bad assumption when inspecting the code, and not notice the error. So if we have an issue that we can't find via code inspection, how do we find it?

Finding problems by running the program

Fortunately, if we can't find an issue via code inspection, there is another avenue we can take: we can watch the behavior of the program as it runs, and try to diagnose the issue from that. This approach can be generalized as:

1. Figure out how to reproduce the problem
2. Run the program and gather information to narrow down where the problem is
3. Repeat the prior step until you find the problem

For the rest of this chapter, we'll discuss techniques to facilitate this approach.

Reproducing the problem

The first and most important step in finding a problem is to be able to *reproduce the problem*. Reproducing the problem means making the problem appear in a consistent manner. The reason is simple: it's extremely hard to find an issue unless you can observe it occurring.

Back to our ice dispenser analogy -- let's say one day your friend tells you that your ice dispenser isn't working. You go to look at it, and it works fine. How would you diagnose the problem? It would be very difficult. However, if you could actually see the issue of the ice dispenser not working, then you could start to diagnose why it wasn't working much more effectively.

If a software issue is blatant (e.g. the program crashes in the same place every time you run it) then reproducing the problem can be trivial. However, sometimes reproducing an issue can be a lot more difficult. The problem may only occur on certain computers, or in particular circumstances (e.g. when the user enters certain input). In such cases, generating a set of reproduction steps can be helpful. **Reproduction steps** are a list of clear and precise steps that can be followed to cause an issue to recur with a high level of predictability. The goal is to be able to cause the issue to reoccur as much as possible, so we can run our program over and over and look for clues to determine what's causing the problem. If the issue can be reproduced 100% of the time, that's ideal, but less than 100% reproducibility can be okay. An issue that occurs only 50% of the time simply means it'll take twice as long to diagnose the issue, as half the time the program won't exhibit the problem and thus not contribute any useful diagnostic information.

Homing in on issues

Once we can reasonably reproduce the problem, the next step is to figure out where in the code the problem is. Based on the nature of the problem, this may be easy or difficult. For the sake of example, let's say we don't have much of an idea where the problem actually is. How do we find it?

An analogy will serve us well here. Let's play a game of hi-lo. I'm going to ask you to guess a number between 1 and 10. For each guess you make, I'll tell you whether each guess is too high, too low, or correct. An instance of this game might look like this:

```
You: 5
Me: Too low
You: 8
Me: Too high
You: 6
Me: Too low
You: 7
Me: Correct
```

In the above game, you don't have to guess every number to find the number I was thinking of. Through the process of making guesses and considering the information you learn from each guess, you can "home in" on the correct number with only a few guesses (if you use an optimal strategy, you can always find the number I'm thinking of in 4 or fewer guesses).

We can use a similar process to debug programs. In the worst case, we may have no idea where the bug is. However, we do know that the problem must be somewhere in the code that executes between the beginning of the program and the point where the program exhibits the first incorrect symptom that we can observe. That at least rules out the parts of the program that execute after the first observable symptom. But that still potentially leaves a lot of code to cover. To diagnose the issue, we'll make some educated guesses about where the problem is, with the goal of homing in on the problem quickly.

Often, whatever it was that caused us to notice the problem will give us an initial guess that's close to where the actual problem is. For example, if the program isn't writing data to a file when it should be, then the issue is probably somewhere in the code that handles writing to a file (duh!). Then we can use a hi-lo like strategy to try and isolate where the problem actually is.

For example:

- If at some point in our program, we can prove that the problem has not occurred yet, this is analogous to receiving a "too low" hi-lo result -- we know the problem must be somewhere later in the program. For example, if our program is crashing in the same place every time, and we can prove that the program has not crashed at a particular point in the execution of the program, then the crash must be later in the code.

- If at some point in our program we can observe incorrect behavior related to the problem, then this is analogous to receiving a “too high” hi-lo result, and we know the problem must be somewhere earlier in the program. For example, let’s say a program prints the value of some variable *x*. You were expecting it to print value *2*, but it printed *8* instead. Variable *x* must have the wrong value. If, at some point during the execution of our program, we can see that variable *x* already has value *8*, then we know the problem must have occurred before that point.

The hi-lo analogy isn’t perfect -- we can also sometimes remove entire sections of our code from consideration without gaining any information on whether the actual problem is before or after that point.

We’ll show examples of all three of these cases in the next lesson.

Eventually, with enough guesses and some good technique, we can home in on the exact line causing the problem! If we’ve made any bad assumptions, this will help us discover where. When you’ve excluded everything else, the only thing left must be causing the problem. Then it’s just a matter of understanding why.

What guessing strategy you want to use is up to you -- the best one depends on what type of bug it is, so you’ll likely want to try many different approaches to narrow down the issue. As you gain experience in debugging issues, your intuition will help guide you.

So how do we “make guesses”? There are many ways to do so. We’re going to start with some simple approaches in the next chapter, and then we’ll build on these and explore others in future chapters.