Debugging Your Code

Written by Keith Schwarz

In the course of working on any programming project you will invariably run into a spot where your code just plain doesn't work the way you want it to. It might not compile and give you some super cryptic error message. It might run but give the wrong answer, or run but never complete. Or it might run and crash for mysterious reasons. When this happens, you'll need to debug your error before you can move on.

As you may know, we have LaIR hours every Sunday through Thursday. If you ever get really, really stuck, we recommend stopping on by to get some help.

Knowing how to debug effectively is one of the single most important skills you can have as a programmer. When you're working outside the classroom – whether it's in a research lab, an independent project, a company, a nonprofit, etc. – you won't have the luxury of the LaIR to help you get unstuck, and yet you'll still be expected to deliver results. Accordingly, if you ask for help in the LaIR, the section leaders are likely to inquire into more depth about how you've tried to fix the error on your own. If you haven't made much of an effort to do so, the SLs are more likely to help show you how to debug your code than they are to find and fix the bug for you.

Before you go to the LaIR to get help, we recommend that you work through this handout. In doing so, there's a good chance that you'll figure out what the underlying error is and how to fix it. In the event that you aren't able to sort out the issue, then you'll have a much more targeted and focused question to bring to the LaIR.

Without further ado, here's our advice about how to debug your code.

Step Zero: Don't Panic

There's an old joke that goes like this:

- Q: What's the first thing you should do when you find yourself stuck in hole?
- A: Stop digging!

When you're writing a program and you find that it doesn't work, your instinct might be to try to take the code you have and change it to fix it. In some cases, this is totally fine. Maybe you look at the code and think "oh, whoops, I forgot a semicolon" or "oh whoops, I forgot a return statement." That's all fine and well.

But if you're in a spot where the code doesn't work and you really legitimately don't know what's going on, one of the worst things you can do is to start making changes to your code without a plan. Now you have two bugs – the bug you originally had, plus the new one you just inserted by changing things.

An important thing to keep in mind when debugging is that ***the bug can't move***. It's sitting there in your code, hiding in plain sight, waiting to do the wrong thing. If you don't make changes to your code, then it can't move anywhere. It can't sneak from one part of the code into the other. It's just waiting to be found.

"When debugging, novices insert corrective code; experts remove defective code."

So don't go changing anything. Instead, switch into debugging mode. Put on your sleuthing hat, and get ready to go on a bug hunt.

With that having been said, your first step is to figure out what the real underlying problem is.

Step One: Identify the Specific Issue You Have

The first question you should ask yourself is

What is the ***specific*** problem that you're running into?

It might seem silly to ask this, but it's important to make sure that you understand what specifically the issue is that you're running into. If you show up in the LaIR and all you can say is "my program doesn't work," it's unlikely that anyone is going to be able to help you fix the issue, since that information by itself isn't precise enough for the section leaders to know what's going on.

As a first step, try to classify the error you're getting into one of four different categories:

- ***Compiler error***: The program doesn't even compile.
- ***Program crash***: The program compiles and runs, but it crashes.
- ***Infinite loop***: The program just seems to hang without anything happening.
- ***Logic error***: The program runs, but it produces the wrong answer.

Once you've identified what class of error you're getting, you can take some next steps to pin that error down more precisely.

Once you've identified what class of error you're getting, you can take some next steps to pin that error down more precisely.

- ***Compiler error***: What is the specific error message you're getting? Is it localized to a particular line of code? If so, what line of code is that? If you have multiple different compiler errors in the same program, always just look at the first of them, since follow-up errors often result from the compiler trying to recover from an earlier error and failing.
- ***Program crash***: On what specific line is the program crashing? What sort of error are you getting back? Can you get a stack trace? Make sure you can answer all of these questions, since they'll be important for smoking out the bug later on. In most cases, you can figure out where the crash occurs by running the program in debug mode and waiting for the debugger to pop up when something goes wrong. Once you've found the crash, you'll need to localize it; see the "Localizing a Bug" section later on.
- ***Infinite loop***: Where in the program is the infinite loop? Don't guess – use the debugger to find out. Run the program in debug mode, and when it seems to be hanging, click the "interrupt" button (it looks like a giant pause button) to pause the program at its current point. That should give you a stack trace of where you are in the program, which will pinpoint the specific area where the infinite loop occurs. You might find, sometimes, that your program isn't in a loop but is just paused and waiting for you to type something in, in which case, problem solved!
- ***Logic error***: If you're getting the wrong answer, what specific function is producing the wrong answer? Once you've found that function, there are a couple reasons why it could be giving a wrong answer. It might, for example, give back the wrong answer because the input that was provided to it was incorrect and the error is actually earlier on in the program. Therefore, your task at this point is to localize where the bug is (see the "Localizing the Bug" section).

A unifying theme in the above discussion is that, when possible, you'll want to use your tools to help you figure out what's going wrong. When you first have a program that isn't working, it can seem really overwhelming, but by asking the right questions you can convert the Hairy Scary problem of "it's broken" into a more concrete question, like "what exactly is it about this particular loop that makes it run forever?" or "why does this particular line of code cause a crash?"

A unifying theme in the above discussion is that, when possible, you'll want to use your tools to help you figure out what's going wrong. When you first have a program that isn't working, it can seem really overwhelming, but by asking the right questions you can convert the Hairy Scary problem of "it's broken" into a more concrete question, like "what exactly is it about this particular loop that makes it run forever?" or "why does this particular line of code cause a crash?"

Localizing a Bug

In some cases, you'll end up finding that your program does the wrong thing at a particular step – maybe it reads off the end of a string (oops), or perhaps it computes and returns the wrong value (oops). When that happens, you'll have to identify the root cause of the problem before you can move on so that you know where to focus your efforts.

As an example, let's suppose that you want to write a function that takes in a string and counts up how many distinct characters it has. For example, "hi" has two distinct characters, while "hello" has four.

One way you can do this is to sort the characters in the string alphabetically, then count up how many different "runs" of equal characters there are. For example, given the string "Banana," you'd form the string "aaaBnn," which has three runs of equal characters (aaa, B, nn) and therefore has three different characters. You could imagine writing the function like this:

```
int distinctCharactersIn(string input) {
        string sorted = sort(input);
        return countRunsIn(input);
}
```

Now, imagine you're running this function and you find that it gives back the wrong answer on the input "Banana," and this shows up because the starter code compares your answer to a reference answer and says that it has the wrong answer. You can be fairly confident that the issue is that the **`distinctCharactersIn`** function is returning the wrong value, but until you know *why* it's returning the wrong value it's going to be hard to do anything about it. For example, here are four different things that could go wrong here:

- The input provided to the function is somehow wrong, so you're counting the number of distinct characters in the wrong string. (You sometimes hear this referred to as "garbage in, garbage out:" if you ask the wrong question, you get the wrong answer.)
- The **`sort`** function doesn't correctly sort the characters of the string.
- The **`countRunsIn`** function doesn't correctly count how many runs of characters there are.
- The **`countRunsIn`** function is being called with the wrong arguments.
- Fundamentally, the logic being used is incorrect.

Without digging into your code a bit more, it's hard to know for sure which of these errors is the real one. Therefore, when you find this bug, the first step is to localize it to a particular region of the code. Here are some things you may want to do:

- Run the program with the debugger turned on. Set a breakpoint at the top of the function.
- Confirm that the input to the function is what you think it should be. If it's supposed to be "Banana," make sure that that is indeed what you're working with. If not, you've found your error. The question then is "why is it that my function isn't getting the input I thought it was going to get?," and you can work backwards to see how that happened.
- Step over the **`sort`** function. What output do you get from **`sort(input)`**? What output did you expect to get from **`sort(input)`**? If these disagree, you've found what's going on – the real culprit is that **`sort(input)`** is broken. Now you can start looking at **`sort`** to see what's wrong.
- Step over the **`countRunsIn`** function. What output do you get from it? What output did you expect to get from it? If they disagree, you've found the error – you're not properly counting runs, and that means you should look at **`countRunsIn`**.
- Step into the **`countRunsIn`** function. What is the input to the function? How does **`countRunsIn`** operate, and what calculation will it make to that input?
- Look at the overall return value. If you correctly sorted the input and correctly counted runs and yet you *still* aren't getting the right answer, that means that the error is possibly a logic error. Maybe sorting the characters and counting runs just isn't a good way to solve this problem.

The process described here – check that the inputs are right, go step by step and make sure that each step does the right thing, then check that the output matches what you expected – is an incredibly valuable way to figure out what's going on in your program. In the course of performing these steps, you might end up discovering the exact error and won't need to proceed any further. If you can show up in the LaIR and tell the SLs that you've worked through this process and done your best to localize the error, you're in great shape.

Step Two: Determine How to Trigger the Problem

Okay, you've now localized the error. If it's a compiler error, you know what line it's on. If it's a crash, you know which line crashes and you've figured out whether the crash is because the line got the wrong input or because it has the right input and still crashes anyway. If it's an infinite loop, you know where the loop is. If it's a logic error, you've figured out where specifically things aren't going right. In other words, ideally, at this point, you can confidently answer this question:

Where in the program is the root cause of the issue you're seeing?

The next question is to figure out how you can trigger the issue. Let's go back to our example from before. Suppose the code gives the wrong answer when given the input "Banana." Here's a question: will the code *always* give back the wrong answer, regardless of the input, or is there something special about "Banana?" If there's something special about "Banana," does it have to do with the fact that it has some letters that repeat (a and n), some letters that don't (B), or a mix between upper-case and lower-case letters? Or does it have to do with which letters are repeated? Or does it have to do with how many total characters there are? Your next step is to answer this question:

In what cases does this error occur?

In coming up with an answer to this question, you'll narrow down the scope of the cause of the problem so that you can get a sense of what to look for.

Let's go back to our "Banana" mishap from before. A great next step in that case would be to try running the program on inputs *other* than "Banana" so that you can try to sort out what's going wrong. It wouldn't be a bad idea to try running the program on a bunch (no pun intended) of other strings so that you can peel back (pun intended) the veil of mystery about what's going on.

If you have no idea what the issue might be, try running some different inputs through the program. Try small inputs: 0, the empty string, etc. Try large inputs. If you're working with strings, try lower-case strings, UPPER-CASE STRINGS, and Mixed-CaSE strings. See what you find. Does the program work correctly on some cases? Write those cases down. Does the program fail on some other cases? Write those down too.

If you're working on a program and the only way you know how to trigger the error is on a very large input (say, a large sample file), then it's doubly important to find another case where the error occurs. Chances are the issue has less to do with the fact that the input is big and more to do with the fact that the input contains some pattern somewhere that you don't handle properly. One technique for figuring out the issue is to take the test file you're getting the wrong answer on and to dramatically shrink it. Cut out the first half, or last half, and see if you still get an error. If so, cut that half in half, etc. Eventually, you'll zero in on the issue.

If that doesn't work, look at the smaller test cases that you are passing. Is there anything that they all have in common? If so, try creating your own test cases that look different from those ones. Working with text files, and all the inputs are free of spelling errors? Mash the keyboard and generate some garbage. Working with numbers? Try odd numbers, even numbers, negative numbers, 0, fractions, etc. Just throw things at the program until you can reproduce it.

If you *can't* get the error to show up in other cases, or if you can only get the error to show up in super large files, that in itself might be worth a trip to the LaIR so that you can get the section leaders, who all have their share of Debugging Battle Scars, to offer some input.

If you *do* have a better sense of what sorts of inputs cause the error and which sorts don't, look over them and see if there's a pattern. In many cases there will be something obvious – you get the answer wrong for any strings of length two or more, or the function crashes on anything of odd length, or it fails on anything ending in the letter e, etc. – and that's really, really good to know because it will inform what you should look for in the next step. Alternatively, if the program *always* gives the wrong answer, then that's good to know as well – it means that there's something fundamentally amiss and that you're not looking for edge cases.

Step Three: Rubber-Duck Debug

At this point, you should be able to answer this question:

Where in the program is the root cause of the issue you're seeing, and what type of input seems to cause the problem?

Your next step is to walk through your code, one step at a time, to see why things aren't working.

Our recommendation for this step is to use a time-honored software engineering technique amusingly called rubber-duck debugging. The idea is the following. Pull out a tiny yellow rubber duck, or any other inanimate object, and then go one line at a time through the buggy region of the code and explain why every single thing you've written is *clearly* and *obviously* correct, being as specific and precise as possible. Calling a function? Explain to the duck why you call that function and why every input to the function is *clearly* and *obviously* the right input to that function. Running a loop? Explain to the duck why *of course* the bounds of the loop are *totally* correct and that you don't have an off-by-one error in them. Proceeding recursively? Tell the duck about how it's *really obvious* why you chose the right base case, and why that base case is correct, and why you return the right thing in that base case.

From experience, I'd say that nine times out of ten you'll run into a line of code where, in explaining it to the duck, you realize something's not quite right. And congrats! You've probably found your bug.

When you're just getting started programming, you may run into another case – you come across a line of code that for the life of you you can't explain to the duck. You're not sure why it's there or what you were thinking as you were writing it. When that happens, that's a sign that you may have written something that really shouldn't be there. That's a great indicator that you may want to remove that line of code and then restart your process. That might instantly fix the issue, or it might expose another.

Oh, and if you're working with a partner, your partner is a great stand-in for a rubber duck, though (hopefully) the reverse isn't true. 😃

Step Four: Really Dive Deep

At this point, you can answer the following question:

Where in the program is the root cause of the issue you're seeing, what type of input seems to cause the problem, and what does every line of code in that part of the program do?

When this happens, it's time to go and ask for some help. This means that you're at a point where you're pretty sure you're looking in the right place, and you think you know what causes it, and you think that every line of code is correct. Clearly one of those assumptions isn't right, and at this point you should go to the LaIR and get one of the section leaders to look over your code and offer some feedback.

You might find you localized the error to the wrong place in the code. You might find you misinterpreted the directions or just flat out made an error when writing test cases and thought that something working was broken or vice-versa. And you might find there's some weird syntactic nuance in the program where a line of code does something fundamentally different than what you expected. In each case, having outside review would be super useful, and that's precisely why you should ask for help!

Once you've found the error, make a note of what you did wrong. Was it a simple typo you didn't know to look for? Was it a logic error that hadn't occurred to you? Or was it just a boneheaded mistake, the sort of thing that everyone makes every now and then? Regardless, see what you can learn from it. Maybe the SL showed you a new way to debug things, or maybe they showed you a different way of thinking about things. Maybe you now know how to read an error message you previously didn't. In any case, hopefully you walk away having learned something new.

Getting good at programming takes time, and I honestly think that a large part of it is just having had enough time to make a lot of mistakes. Over time, you'll build up an immune system for bugs – things will look fishy and you'll catch the error before you make it.

Until then, best of luck, and happy debugging!