Learn the essentials of debugging

Systematically take on mysterious errors with confidence

Terence Parr (parrt@cs.usfca.edu), Professor of computer science, University of San Francisco

Summary: Debugging software is challenging. Without a process to follow, resolving problems can seem impossible. Most inexperienced programmers find themselves in precisely that situation when confronted with a bug. In this article, walk through a sample problem-solving session to learn the art of debugging and highlight six essential elements of the debugging process.

Date: 07 Dec 2004
Level: Introductory

Comments: 0 (View | Add comment - Sign in)

★★★★☆ Average rating (106 votes)
Rate this article

I commonly hear vague statements, such as: "My program doesn't work!" from my students. Because I want them to learn self-reliance plus diagnostic and debugging skills, my response is usually, "Well, isn't that interesting," or "How do you think you'll handle it?" While not particularly instructive, those responses usually have the desired effect -- the student goes off and puts his or her brain to work thinking about the problem. That said, I've always been troubled by my brute-force, "kick 'em off into the deep end" approach. I wondered about a way to teach debugging itself.

The most troubling thing to me is that many students -- and even commercial coders -- seem to freeze when unexpected problems arise. Some literally try making random changes or, worse, try to run the program a few more times in hopes that it will start to work again. These actions illustrate two primary issues: They have no idea how to approach debugging, and they have no confidence that they can nail the problem. I hope to ameliorate the situation by providing a clear approach that programmers can follow, or at least a place to begin.

To expose the debugging process, I need to narrow the focus from the many possible programming problems. Algorithm implementation errors are reasonably easy to track down with a debugger. You can just step through looking for an invalid state or bad data (the last statement to execute either is itself wrong or at least points you at the problem). Then some truly bizarre problems defy logic. In languages such as C/C++, you might have a buffer overflow situation that wipes out your stack activation record, leading to really unexpected results; the environment itself becomes corrupted in this instance. In the Java language and others like it, this is not possible; however, you can still have bizarre problems. In this article, I will focus on the type of problem where a trivial bit of code that should work, does not.

For years, I've sought a simple example to illustrate some of the key processes rolling around in my head as I attack one of these weird problems. I needed a really vexing problem, but one so simple that it could be described quickly to any programmer, and whose solution was obvious once explained. This article describes such a problem and the path to its successful resolution. First, I'll define the (mundane) problem itself and then walk through the problem-solving process. In the final section, I'll attempt to summarize the essentials of what I was thinking in the debugging session.

The problem

One Friday afternoon in the fall of 2004, my business partner Tom called me on the phone to ask if he could describe a problem to me. I knew the problem was nasty if he needed to talk to somebody else; his programming and debugging skills are formidable. He really just needed another perspective, I thought (though neither of us is well-versed in the issues he was dealing with). It turns out that a single URL issue had the two of us stumped for a few hours.

Tom was generating HTML e-mail messages with embedded links, but exactly one of those links was not going to the right page on the Web when the user clicked on it. He knew precisely which link was causing the problem, and customers were apparently seeing the same issue. Here it is:

```
<a href="http://foo.com/x?event=goto_add&ha_id=10954648">...</a>
```

Seems fine, right? Definitely follows valid URL syntax. It is identical syntactically to all the other links in the application that did work properly. But the link got to his servlet as this:

```
http://foo.com/x?event=goto_add&ha_id%
```

The "`=10954648`" was being converted to a simple percent sign (`%`)! However, links like the following worked perfectly:

```
http://foo.com/y?event.accept&address=ZWxrQHJlZG
```

That is all the information that he gave me.

The twisted path to a solution

My first thought was a bug in the Mac OS X mailer he was using had extracted the link improperly. He indicated that other users seemed to have the same problem, but we decided to rule out the OS X mailer by using a Web-based mail host like Yahoo's. Same problem -- bad link. Ah! What if the browser and mailer use the same HTML rendering code? We loaded Microsoft Internet Explorer for the Mac and it too had the problem. Hmm ... what was involved in the translation of this URL through to his servlet? If the problem wasn't in the extraction of the URL from the e-mail, it might be in the Web server that received the URL. Did the `resin` Web server have a bug that caused it to mangle the URL arguments it provided to the servlet? We decided that this was pretty unlikely in general and particularly in this case, because the server worked fine for other URLs that were even more complicated.

Lacking further ideas derived from deduction, we tried the following experiments:

- Replaced double quotes around the URL in the tag with single quotes.
- Used no quotes around the URL.
- Changed the name of the arguments; I thought perhaps the underscore in `ha_id` annoyed the mailer.
- Changed the order of the arguments.
- Changed the number of arguments.

Nothing changed; the link was still bad in all cases. We also tried changing the equals sign (`=`) to the URL escape `%3D`, thinking that some odd escape issue was the cause, but the Web server then didn't see `ha_id` as an argument (the equals sign after it was escaped).

If we assumed that the URL was fine, then something must have been wrong with the HTML code surrounding the URL. But that code looked totally fine as well, and moving the surrounding HTML around had no affect. We reached a dead end there.

We decided that, like it or not, something in that particular URL, and probably that one argument, was invalid. I asked if any JavaScript or other preprocessing occurred that might alter the URL before the mailer sent it to the browser. But we did not find or think of any.

If the URL had something wrong with it, it was time to compare the URL to something that worked. What's the difference between the arguments `ha_id=10954648` and `address=ZWxrQHJlZG`? Clearly, one argument is a number and the other is not. We put a `Z` in front of the `10954648` and inserted code into the processing servlet to strip the first character from the `ha_id` argument. It worked! It was pretty unsatisfying, however, because we didn't know why.

We now knew that something about the equals sign followed by a number caused the problem -- but what? We knew that the equals sign is a Base64 MIME type padding character, but that didn't explain how it and the argument were converted to a percent sign. But something was definitely wrong with the equals sign followed by a number sequence.

We tried lots of Google searches on terms such as "equals email URL problem," but found nothing appropriate. But did this jog anything in my own memory?

Back when I was using a text-based e-mailer, I sometimes got e-mail that mostly looked like text, but had a bunch of `=0A`s and other wacky things in it. We looked up what 10 hex (the first two characters of 10954648) was in terms of ASCII; it's the *data link escape* character. In decimal, it's a linefeed. 109 hex (first three characters from the ID) is the lowercase letter `m`. So `=10` didn't seem to yield the percent character (ASCII code 45 decimal).

We tried Google again for e-mail encoding types, as it had to be something related to that. Finally, we found the keyword *quoted-printable*, which led us to realize that it explained all known behavior -- because its escape character is the equals sign! We looked, and indeed the header for the e-mail said:

```
Content-Transfer-Encoding: quoted-printable
```

We changed `ha_id=10954648` to `ha_id=3D10954648` because `3D` hex is the equals ASCII character, and *voilà!* It worked! Removing the escape and the `Content-Transfer-Encoding` header made the original URL and all others work just fine.

The solution is obvious once you see it, but a combination of logic, research, and experimentation was required to find it (as is often the case). This type of problem can take forever to solve if you do not have a decent debugging instinct.

The essentials

What can be generalized from the process above? I conclude that you must employ six essential elements:

1. **Reproducibility.** First, find a way to reliably reproduce the error, which, in itself, often points you straight at the problem through deductive reasoning. The bug may happen in precisely one circumstance, which can only happen in one place in the code. A bug that appears randomly is essentially unsolvable unless you have a leap of insight. You need a guarantee of cause and effect to make inferences about changes you introduce. A change in the code that *fixes* the problem may or may not really fix it, as the problem randomly appears and disappears.

2. **Reduction.** Reduce the problem to its essence. Determine the smallest input that will cause the bug. The simpler the data or path to the bug, the more easily you can deduce or track down the problem. A large data set introduces a great deal of noise that camouflages the essential cause of the trouble. If you have a large data input file that causes the problem, do a binary search type reduction. Cut the file in half, throwing out the last half. If you still have a problem, then you can ignore the final half. If the problem goes away, then start whittling down the final half, as it must contain the input that causes the problem.

3. **Deduction.** This is your primary weapon once you have a small input that reliably causes a problem. What is the general path through the program used by the input? What components might be the problem or mangle the data so that a future component fails? What is the difference between the input that doesn't work and some other input that does? Try to reduce the scope of possibilities by forming and eliminating hypotheses. For example, we eliminated the possibility that the Mac OS X e-mailer had a bug by reproducing the bug using Internet Explorer and a Web-based mailer. In a sense, this process is similar to that followed by experimental physicists, who try to explain natural phenomena with a theory or an equation. To support their claims, they carefully design experiments that, if successful, have only one likely explanation -- namely, their theory. Other physicists try to reproduce the results to verify or refute the hypothesis.

4. **Experimentation.** Psychologists study the human mind by testing it in varying situations with different stimuli. They use brain scans, response times, and so on to support their hypotheses about how human brains work. Similarly, you must change the conditions of the test to see if your bug disappears. If you correct the bug with a change, that change might tell you what the problem is, or at least give you a big hint about what is going on. You form hypotheses with your logic and deductive reasoning skills and then filter them by experimentation and observation. For example, by experimentation, we concluded that the specific argument caused the problem. Finally, we narrowed it down (by using some experience) to the *equals as escape* problem.

5. **Experience.** Experience has no substitute. To become a good programmer, apprentice yourself to a good programmer or fumble your way through it yourself for a few years (making lots of mistakes either way). Experience helps in the debugging process in two ways: first, you hone your ability to execute the previous four elements; and second, you might have seen a similar bug or just plain know more about a particular problem. For example, I had seen the equals sign as an escape character in e-mail before, and the next time I see or somebody asks me about a URL problem in HTML mail, the first thing I'll ask about is the encoding. Borrowing the experience of other developers is also important. Searching the Web and talking to other developers can save you a huge amount of effort by leveraging other peoples' experience. Interestingly, just explaining the problem to another developer (or even your spouse or a friend) can line things up properly in your head so that simple deduction tells you where the problem lies.

6. **Tenacity.** All bugs are caused by computers doing exactly what they are told; absolutely no mystery in that, and hence all problems are solvable. You must begin with the attitude that you will never give up no matter how long it takes to find the problem and correct it. You will become more and more confident each time you solve a problem. You must be tenacious. I have never, ever left a bug unsolved that I considered worth fixing. Sometimes the lengths you have to go are extreme, but result in good war stories. Once, when debugging a small device controlling a robot, my only link to the outside world was an LED that I could blink. I had to attach an oscilloscope to look at how the software was wiggling the LED! In the process of solving insidious bugs, you will become a much better programmer as you start to anticipate errors and to code in a manner less likely to produce mysterious bugs.

In practice, these six elements are used in combination and their sequence may vary, though points 1 and 2 make sense to do first.

Once you have found the bug

Just as the process of debugging itself is important, knowing when you can stop is also crucial. Internalize two key principles so you can recognize when you have squashed a bug.

First, if you have stumbled onto a solution that seems to fix the problem, but you don't understand how it does so, you cannot rely on it. Many programmers stop when something they change in the code makes the bug disappear, even though they have no idea why. This is not a solution. The bug still exists; you have merely hidden it for the time being. For example, Tom and I did not stop after we made the link work by prefixing the ID number with `z`; we had merely hidden the problem. "Never leave an enemy at your back," as the adage says.

Second, almost as bad as no explanation is a theory that does not explain all known behavior. A single unexplained issue implies that you have not found the real solution, or that multiple bugs are at play. Recall that a theory in physics or math can be shot down with single counterexample. A story is told about Einstein during the 1930s. He was asked by a journalist if he was worried that so many prominent German physicists were coming forward to discount his theory of relativity. Einstein replied simply: No, if relativity were incorrect, it would only take a single physicist to show it.

Summary

Debugging can be one of the most difficult and frustrating aspects of being a programmer, but your attitude makes a big difference. I approach a nasty problem as if it were an interesting or challenging mystery to solve. Moreover, I attack the problem with confidence, knowing that eventually I will solve it. Your attitude can also mean the difference between being a good programmer or a bad programmer. I asked a pilot friend of mine once if he was disturbed when he had to perform a difficult landing in high winds. He replied that, no, most of the time flying is routine -- pilots earn their money during difficult landings, and some enjoy the opportunity to demonstrate their prowess. Similarly, programmers distinguish themselves most clearly when confronted with difficult bugs to solve. The essentials described here give students and new programmers a strategy to follow, thus increasing their confidence to find and correct their bugs.

I'd like to thank Tom Burns and Sriram Srinivasan for their helpful suggestions on this essay.

Resources

- Read Eric Allen's column for developerWorks, Diagnosing Java code. The focus is on debugging, with an emphasis on what he calls *bug patterns*.
- For more on debugging with WebSphere, check out "Debugging WebSphere Application Server Version 5."
- For more debugging tips, read "An introduction to debugging using IBM WebSphere Studio Site Developer and Application Developer," Pete Nicholls (*IBM WebSphere Developer Technical Journal,* January 2002).
- Browse for books on these and other technical topics.
- Visit these valuable resources on the IBM developerWorks site:
  - The developerWorks Web Architecture zone specializes in articles covering various Web-based solutions.
  - Browse for books on these and other technical topics.

About the author

Terence Parr is a professor of computer science at the University of San Francisco, where he continues to work on his ANTLR parser generator. Terence recently returned from years working in the industry, where he co-founded jGuru.com, herded programmers, and implemented the large jGuru developer's Web site. His new templating engine, StringTemplate, evolved during development of the jGuru.com server. Terence holds a Ph.D. in computer engineering from Purdue University. He can be reached at parrt@cs.usfca.edu.

Close [x]

developerWorks: Sign in

If you don't have an IBM ID and password, register here.

IBM ID:
Forgot your IBM ID?

Password:
Forgot your password?
Change your password

After sign in: Stay on the current page

Keep me signed in.

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

The first time you sign into developerWorks, a profile is created for you. This profile includes the first name, last name, and display