Unix and Computer Science Skills Tutorial
Workbook 3:
Debugging

October 2, 2015

This workbook is a tutorial on how to use debuggers and other tools to fix problems with your programs. It covers some general principles on how to manage bugs, then some general principles of debuggers. Then it covers some

As with the other workbooks I recommend that you take the time to work through the examples to get a feeling for how things work. I have also included some questions and prompts, again, to prompt you to take a look at the tools and ideas. Sometimes the answers to the questions will be fairly direct from the workbook and some will require some extra research on your part. The prompts will encourage you to try things for yourself.

This workbooks assumes that you are comfortable doing basic tasks including downloading software and editing, compiling and running code.

1 What is a Bug

A *bug* is an anomaly in software. When we create software, we expect it to do a set of things (which we call features), bugs are the things that that software does that we *didn't* expect it to do.

Bugs come form a lot of different places. Sometimes bugs are mistakes, a programmer writes a <= when they really meant <. Sometimes bugs come from bad assumptions, it seems like a measurement is being recorded in metric when really its being recorded in imperial (and you crash your space probe into mars). Sometimes bugs come from program interactions, if two programs are waiting to access a resource sometimes they can both end up waiting on each other. Sometimes bugs come from hardware, you might have a bad sector on your drive that gives you faulty data or you might have a moth sitting in one of your relays (see Figure 1).

Generally, while learning to program, bugs will be our own fault. There is a lot of material to learn and there are a lot of assumptions made by programmers that have come before us that we will have to figure out. Sometimes we will make typos. Sometimes a particular "edge case" of input means that we will have to think carefully about how our algorithm needs to behave. Sometimes

we will discover that some elegant simplification simplifies too much and our code is chopping off corners that it shouldn't be.

As you get better at programming you will produce fewer bugs, although the total time you spend *fixing* bugs may not drop. When you're able to write more complicated software, you can also create more complicated bugs. You will also start to see more conditional bugs, where a problem doesn't happen as long as you have 10,000 entries in a file, but it does happen if you have 10,001 entries in a file.

Debugging is the process of removing bugs from your software. Debugging is a science, you will have to find facts about why your program isn't working, you will have to develop hypothesis to explain these facts and you will have to develop tests to determine if your hypothesis was correct. If it was then you have to fix the bug, otherwise you have to repeat the process until you until you do find your bug.

Debugging is also an art. It will take time and practice to develop a feeling of where to look, each time you encounter a new bug. The more mistakes you make and the more you practice fixing them

Once you've found your bug, fixing is another story. Sometimes it will be an obvious fix and you'll get your program running again quickly. Sometimes you'll have to think about or learn about a new technique. Sometimes you'll realize that fixing one bug will create another bug in your software that's a worse problem and you'll have to find a way to work around your bug.

This workbook includes a discussion of how to reduce your bugs and how to find them when they do appear. The later part of this workbook also includes how to use some of the most useful tools to find and fix your bugs.

1.0.1 Questions

1. Who is Grace Hopper?
2. What is the link between Grace Hopper and bugs?

2 How to Avoid Bugs

The easiest way to get bugs out of your programs is not to let them in in the first place.

That's a lot easier said than done, however. As I already discussed bugs come from a large number of sources, some of which are totally out of your control. Bugs are inevitable, but you are able to control what goes into your program and there are ways to be better about not adding bugs.

The first way to avoid bugs is to plan. Before you start coding sketch out how your program should work. What does it take for input? What does it give for output? What features does it need? Then you can break it down into smaller parts and determine what structures, functions and variables you will need for each feature of your program. This helps you break assumptions early,

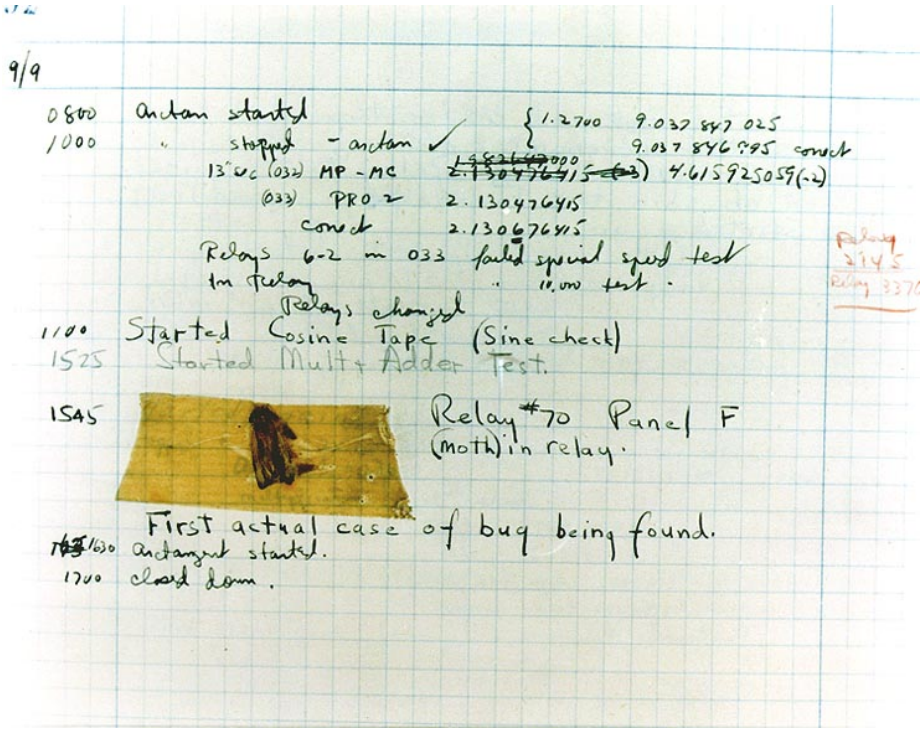

Figure 1: First computer "bug". From the Mark II Aiken Relay Calculator at Harvard University on, 9 September 1947[2]

and will make it easier to see what things go together and can share resources, and what things need to be separate.

The next way to avoid bugs is to pay attention while you are coding your program. Spelling may or may not count in the rest of the world, but misspelled variables or function names are a really quick way to introduce bugs into your program. Similarly, "bad habits" such as, using magic numbers rather than named constants increase your risk of introducing bugs. Being mindful when you code is important, but fortunately there are tools that provide static analysis that can help avoid these types of bugs (I talk more about static analysis is section 4.2).

Compilers also do their best to help keep you from putting bugs into your programs. When you write code that can't be compiled, you will get errors and you won't be able to run your program. This happens when you do something that goes against the language specification (you can't return two values from a Java function, for example). Compilers also give warnings, these happen when you write code that meets the language specification but is often a "bad idea". When ever you write code you should go through all the warnings in exactly the same way you go through all the errors until you are absolutely certain that

the warning doesn't apply to you. Interpreters also give errors and warnings. Always pay attention to the output that you're getting from your compiler or your interpreter.

Another way to avoid bugs is to keep your changes to your program as small as possible. Run your code as often as possible to make sure that everything works the way to expect it to. It's not a bad idea to have a cycle of writing a line of code (or a small block for a statement or loop) and then compiling and running your program. This way you have at least one problem to iron out and you're able to catch it before it goes on to be a bug that gets in your way later. This is another reason why it's important to break your plan for your program down into as small pieces as you can since it makes it a lot easier to see when things aren't working the way they are supposed to.

Finally, you should test your program. Automated testing for Unit Tests and Regression Testing are very helpful for making sure that a) your program does what it's supposed to do and b) it keeps doing what it's suppose to do. Testing is a huge field, and is outside the scope of our debugging discussion, but I recommend you spend time getting comfortable with efficiently testing your code when you can. In a professional environment a large amount of effort is given to testing so it's an important skill to have.

2.0.2 Questions

3. Suppose I asked you to develop a program to take a list of numbers from standard input and produce, a total, mean, mode, median and standard deviation for that list. Write a short plan that outlines what you would need to make that program work, include the functions and variables you need and a rough sketch of how you would need to put the program together (what tasks do you need to do?). State any assumptions you are making about how the program behaves.

3 How to Find Bugs

So, you've followed the first section and written as few bugs as you possibly could into your program. You don't have any warnings and static analysis doesn't have any more hints for things. However something still isn't right with your program, and now you have to find it (and fix it).

The most important part of debugging is creating the division between what your program is supposed to do and what your program is doing. This can often be difficult because you wrote your program knowing what it's supposed to do and in your mind each part of the program is part of what the program is supposed to be doing. You will have to see past your own assumptions about the program in order to fix it. Now is a good time to walk away from your computer for a bit, don't forget to lock your screen.

Once your head is clear, you can start the debugging process. The first thing you should try is telling all of your problems to a rubber duck. (Ok, it doesn't

have to be a rubber duck.) *Rubber duck debugging* is the process of talking through your program and explaining what it should do and what it does.

The name rubber duck debugging comes from a programmer's parable about a new programmer who would go several times a day to another programmer and ask for help finding a bug in their code. As the new programmer would explain what their program was doing, they would see where their bug was and go back to work without the more experienced programmer needing to say anything. After a while the experienced programmer brought in a rubber duck and put it on top of the water cooler and told the junior programmer, "I'm happy to help with any problem you have, but you usually don't actually need me to say anything. So now, stop and explain your problem to the duck and then come to me if you're still stuck."

When you are rubber duck debugging you are walking yourself (and maybe your duck) through your code and looking at where your assumptions about the code don't fit the reality. Start by asking yourself these questions[1]:

1. What makes you say your code isn't working?
2. What did you expect your code to do and why?
3. What did your code do instead and how do you know?

Walk through your code, looking at what each line does, as you ask yourself these questions. You don't have to have a rubber duck to talk to, although it may help to talk it through out loud, since this engages different parts of your brain then simply thinking too yourself. Make sure that you're discarding your assumptions about what your code does and really looking at it.

If simple rubber duck debugging doesn't help you, then you might want to do a *desk check*. In a desk check you play the part of the computer. Take some paper and manually trace (look at each line of your program) what your program does on each important line of code. You should set up a column for each of your variables and then write For example, if you have a variable called `sum`, every time `sum` is updated, write down the new value in your table. This helps in two ways, one it gives you imperial evidence about the behaviour of your program and two it further lets you check your model of what your program is doing compared to what's actually happening.

A desk check can be thorough, but they take time to complete and can be hard to do when the data your program works with is very complicated. Fortunately there are *debuggers* which allow you to trace your program and to watch variable values. Like a desk check it allows you to follow all or some your variables as well as to step-through the program to see what happens on each line. In Section 4 we go into detail about how to use the debugger.

Sometimes you might also want to debug with print statements. For small programs this can often be a faster way to quickly look at what you suspect might be a problem. For example if you think that an if statement always evaluates to true, you might put a print statement both the true and false

branch to see which branch is being executed. This is basically the same as checking with the debugger, but can be faster for simple questions.

Debugging by print statements is sometimes necessary when the debugger would disrupt the natural flow of the program (usually when the program runs in multiple threads, which is well beyond what we need to cover here). Sometimes however the print statements can be causing the bugs in your program in which case using the debugger is the better option. If you do debug by print statements make sure you clean up after yourself once you have your program bug free. No one needs to know about the internals of your program.

Debugging can be a slow process. Usually fixing a bug is quite quick, often you only need to change a couple of words or lines of code. However you will often fix one bug to reveal another underneath (this is why I recommend trying to add new code to your program in as small a chunk as possible).

4 Tools for Debugging

There are three tools that can make debugging simpler, debuggers, static analysis and profilers.

- Debuggers allow you to step through code and check the values of different variables.
- Static analysis tools look at your code and compare it to a database of good and bad practices to let you know where you might be doing something you didn't mean to.
- Profilers allow you to look at the overall behaviour of your program and see where it's using resources. You can see how much memory different operations take and how long it takes to do various things in your program. Profilers can be very useful for tracing some problems, but as with testing they're a useful tool that's beyond the scope of this workbook.

4.1 How a Debugger Works

Debuggers generally allow you to inspect your code as your program is run, or to look at the results of a program after it has crashed. We are mostly interested in debugging Python and Java for early computer science, so we will focus on "live" debugging.

The debugger is able to "pause" your program while it runs. Usually the debugger will stop on the first (meaningful) line of the program. You can then either *step* through the program or tell the debugger to run to the next *breakpoint*.

Each time the debugger pauses your program, you can then check the values of various variables. Exactly how much information you can see depends on the debugger and the language you are debugging, but for any primitive types (integers, floating points, characters) you will usually be able to see the values.

For more complex data types (structures or objects) you may only see a reference number or you may be able to see all of the data inside (this is nearly always true of Strings).

There are usually two options when stepping through a program. The first is to **step into**. In this case if it can, the debugger will switch to (step into) the first line of the function being called (It will usually need access to the source of the function to show anything). When the function is done then the debugger returns to the previous function. The way to step is to **step over**. here the debugger will run the function that would be stepped into but will not switch into it. You will usually want to step over calls to language and system libraries when you are debugging your code. (That being said if you want to see how elements of the language you are using are written, then this is a good way to see them.) Step into and step over may have different names in different debuggers.

> Some debuggers will allow you to step backwards. This can be useful if you need to go back and "replay" the last few steps of the program to see what happened again.

Stepping all the way through your program is fine if you will only execute a few lines of your program, but there area lot of cases where you will have many more lines to go through. This is the function of breakpoints.

Breakpoints allow you to tell the debugger, "run to here". You set breakpoints by telling the debugger which lines you would like it to stop on. When the debugger has paused your program, you can tell it to continue running. The debugger will then run your program as normal until it reaches the next break point where it will pause again. Then you can either continue running it to the next breakpoint or start stepping through the program.

4.1.1 Using a Debugger

As I mentioned debugging is an art (as well as a science). To get good at debugging you must practice debugging. In this section we will play with the debugger a little to give you a feeling of how the debugger works. You will have to keep using it on your own code in your own projects to really start learning how to use it.

I have provided two simple example programs that will be interesting to debug. One is in Python and the other is in Java. You can get them from the course directory `/home/courses/unix/debug_examples`. Both programs do the same thing; they generate two random numbers and add them together. If the sum of the two numbers is greater than 75 then the program stops. If the sum is less than 75 than it generates a new pair of random numbers and tries again. I chose this as an example for debugging because it's difficult for us to tell what the new random numbers are and we can't tell when the program will end. Both

of these programs work as expected (I think), so we will be debugging them to see how they work as opposed to when they don't work.

Choose the language you would prefer to work in and choose a debugger.

- For Python, there is a debugger built into the IDE IDLE, which is present on all department machines with Python:
  - There's a written tutorial on how to start debugging with IDLE here: `http://www.cs.uky.edu/~paulp/CS115S11/DebugIdle.htm`
  - There is a YouTube video tutorial on how to start debugging with IDLE here: `https://www.youtube.com/watch?v=kpyDMK9ZeV4`
- The JetBrains IDE PyCharm also has a built in debugger which is slightly more powerful than the one built into IDLE.
  - You can download PyCharm here: `https://www.jetbrains.com/pycharm/`
  - There is a tutorial for debugging in PyCharm here: `https://www.jetbrains.com/pycharm/help/debugging.html`
- You can also debug Python from the command line or from inside scripts using `pdb` the built in Python Debugger:
  - A description of how to use `pdb` is here: `https://docs.python.org/3/library/pdb.html`
  - There is a video tutorial on using `pdb` here: `https://www.youtube.com/watch?v=bZZTeKPRSLQ`
- For Java, the command line debugger is `jdb` which is included in the JDK:
  - Documentation for `jdb` is provided here: `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html`
- Debugging is built into the popular Java IDE Eclipse, which is installed on all department machines.
  - A tutorial for debugging in Eclipse is here: `http://www.vogella.com/tutorials/EclipseDebugging/article.html`
- The NetBeans IDE was developed by the same group that develops Java and has both a debugger and a profiler:
  - You can download NetBeans from here: `https://netbeans.org/`
  - There is a video tutorial on debugging with NetBeans here: `https://www.youtube.com/watch?v=06B9tsOKtZE`
- JetBrains also makes the Java IDE IntelliJ IDEA:
  - You can download IDEA here: `https://www.jetbrains.com/idea/`

- There is a tutorial on debugging in IDEA here: `https://www.jetbrains.com/idea/help/debugging-your-first-java-application.html`

Once you are comfortable with your chosen debugger, try debugging the random while example in your chosen language. Practice setting breakpoints, stepping through and stepping over. Set a breakpoint in the while loop to let you check on the process of the program without having to step through all the code.

4.1.2 Questions

4. **Write a brief description of what happens when you tried to debug the program. Start by telling me which version you chose to debug and then describe your process of debugging.**

4.2 Static Analysis

Static Analysis tools review your code and compare it to a database of hints about coding in a particular language. These hints are usually categorized into types of issues as well as severity. Sometimes these points may not apply to the program you are writing but often you will find mistakes you've made or get hints for how to do something more efficiently.

- There are several good static analysis tools for Python includeing:
  - Prospector (`https://prospector.readthedocs.org/en/master/`)
  - Pylint (`http://www.pylint.org/`)

  however we are not able to install these on our own so I recommend looking at:
- The IDE PyCharm (`https://www.jetbrains.com/pycharm/`) provides a good static analysis tool that we can install on the department machines.
- For Java the best known static analysis tool is called FindBugs (`http://findbugs.sourceforge.net/`) which you can down load as a stand alone client.
- The IDE IDEA by JetBrains also has an excellent static analysis tool set (`https://www.jetbrains.com/idea`)

Take a minute to apply static analysis to the random while program I've written and see what mistakes I've made while coding.

5 What Next?

Debugging other programs is a great way to learn. However when learning to debug there is no substitute for the practice of looking at your own code. As I said earlier, debugging is an art and you'll have to learn that art by practice.

Start your practice by using these tools on the project you have most recently finished. Take at least a few minutes to step through your program running and see what happens as the program runs.

References

[1] **Teaching Novice Programmers How to Debug Their Code**, Farmer, J. code:union Blog, September 3, 2014. (`http://blog.codeunion.io/2014/09/03/teaching-novices-how-to-debug-code/`)

[2] U.S. Naval Historical Center Online Library Photograph, Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. (`https://en.wikipedia.org/wiki/File:H96566k.jpg`)