DD2458, Problem Solving and Programming Under Pressure

Lecture 3: The Hitchhiker's Guide to Debugging and Testing

Date: 2011-02-01
Scribe(s): Joel Bohman, Linus Wallgren, Oskar Werkelin Ahlin
Lecturer: Alexander Baltatzis

Debugging is the art of finding bugs in code and to be able to get rid of them. Nowadays, this art has become a serious profession for many programmers, who spend their working days hunting bugs. The first step in learning about debugging is to know about one of the most well-know bugs in history. In 1946, at Harvard University, a moth caused a system failure by being trapped inside a relay in one of the electromechanical computers used at the time. Even though it is not common nowadays that program bugs are caused by actual insects, their source is still often evasive and hard to detect. This is a short guide for anyone interested in learning more about debugging and testing.

1 Testing

Testing the code you have written is of great importance. It is the way you confirm that your program runs the way you expect it to. Having a well tested program means that you can be sure the program will not behave unexpectedly. There are a number of testing techniques available and the most important ones are listed below.

1.1 Documenting tests using a test matrix

You should always document your tests, in order to be able to look back and see what you tested and did not test before, and what happened if you ran the program in that specific way. A test matrix is a good way to implement the fundamental ideas of documented testing. It contains a collection of tests, and for each test we list four points:

- Prerequisites - what are the conditions for running this test?
- The test itself - what part of the program does it execute?
- The expected result - what do we expect to achieve?
- The results from the program - what result did we get?

Following this principle, a matrix is formed as we fill it up with rows, for example:

Test #	Prerequisites	The test	Expected result	Test results
1	Compiled with g++ -O2 -g on an Intel x86-64 architecture, debian stable.	Provoke the program to divide by zero.	An error mentioning divide by zero.	The system crashed and burned.

This is a comprehensive way to document your testing. As the matrix fills up, more knowledge is gained about the programâĂŹs functionality, and actions can be taken thereafter. Documenting your tests in a similar manner is strongly advisable.

1.2 State-based testing

1.2.1 Boundary values

1.2.2 Well defined behaviour

1.3 Behavioural testing

1.4 Testing manually

1.5 Automated testing

1.5.1 Unit testing

2 When an error occurs

2.1 Errors

2.1.1 Compile errors

This is an error that you receive when you compile your code. The compiler stops and tells you there is an error that must be fixed before it can be compiled. Compile time errors are the most preferable errors, as you notice them right away and can correct them fast. In some programming languages, some logical errors will transform into compile errors, due to strong typing in that particular language.

2.1.2 Warnings

Compile time warnings generally do not stop the compiling process, but they are usually good to take care of. It might be that you get warnings which are unavoidable, for example warnings about deprecated methods. If a large amount of these are generated at compile time, more serious warnings might be overlooked, because of the sheer amount of warnings. If you get warnings that you know will nott do any harm, it is advisable to use compiler flags to hide these. In gcc, this can be done using the -W flag.

2.1.3 Run time errors

A run time error occurs after compilation and during the execution of the program. Run time errors are programming errors not detectable by the compiler, making

them hard to detect and remove. An example of a typical run time error is when you try to access an array element outside of the array boundaries or when you try to follow a null pointer.

2.1.4 Logical errors

Logical errors are the most difficult kinds of errors. They are especially difficult to detect as there is nothing technically wrong. The error is only in the behaviour of the program. As these are the most difficult errors to handle we need to employ a couple of debugging techniques.

2.2 Debugging

2.2.1 Trace output

2.2.2 Logging

2.3 Tools

3 Models

3.1 Test-driven

3.2 Contract programming

4 If you get stuck

If you managed to read this far you are now most certainly ready start your quest to perfect the art of debugging. Because hunting bugs can be a tiresome activity that can make the most sane person insane, you need a couple of tips to carry with you. If you find yourself in the situation where you are stuck and all hope seems gone, try one of the following:

- Do something else! A fresh mind and body can do miracles. Take a walk. Have something to eat. Sleep on it.
- Try to explain the problem to another person. By explaining your problem in words it will get you to think about the problem and you might see it from a new perspective.
  - If this does not help, show that person or any other person your code. Another pair of eyes can see something you might have missed. This is called peer-reviewing.
- Start from the beginning. Understanding the logic to solve an assignment is what a programmer spend most time doing. When you have the logic down you can write code in a higher pace. Chances are you will not hit the same problem the second time around. Refactoring your code is often faster than trying to figure out what was wrong with your first version.