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

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.

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 donâĂŹt 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 wonâĂŹt 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

Debugging is the art of localizing a bug and then finding the fix. Debugging is suitable for both run time errors and logical errors.

2.2.1 Compiler debug flag

With certain compilers, for example gcc and g++[1] for the programing languages C and C++ respectively, the debugging flag changes the memory structure of the program. When the debug flag is active the compiler adds padding around variables in order to find out if something goes wrong during debugging. By changing the memory structure you might alter the behaviour of the program, as it is gentler to overflows and similar errors. Even if the program runs fine with the debug flag it might crash without it. Another problem with the debug flag is that it makes it easier for others to reverse-engineer the program. The reason is that the debug flag incorporates more information about the program itself into the binary. Because of these difficulties it is important to test both with and without the debug flag active.

2.2.2 Trace output

Printing information about the state of the program to the screen is a powerful, yet simple way of debugging. The concept is usually referred to as trace output. A trace output should always contain two parts to be useful, location and state. Location should clearly and unambiguously specify where in the program the output is generated. State describes the current state of the program at that location. A state is almost always described in terms of variables, a name and a value. An example of trace output:

$foo() : bar = 5, a = 10$

The example could be be achieved with for instance printf, which is available in most languages. In C it might look like this:

$printf(foo() : bar = \%d, a = \%d, bar, a);$

2.2.3 Logging

Logging is an extension of trace output. Logging is a good way of handling a lot of output. Usually logging happens at several levels, for example *INFO*, *WARNING* and *ERROR*. The logging levels are supposed to describe the output generated. It might be so that it is only interesting to know about errors in the code, in which case the ERROR level might suffice. If you would like to trace the program execution you might want more verbose output, in which case INFO probably is the right choice.

It is also common practice to direct logging output to files, in order to review them later on. This also helps if there is a lot of output, as it would be to much to go through live as the execution happens.

[1]http://gcc.gnu.org

2.3 Tools

3 Models

3.1 Test-driven

3.2 Contract programming

4 If you get stuck