

---

# DD2458, Problem Solving and Programming Under Pressure

## Lecture 2: Debugging

Date: 2009-09-16

Scribe(s): Milan Ivanovic, Johannes Svensson och Pontus Walter

Lecturer: Alexander Baltatzis

---

## Contents

<b>1</b>	<b>Test your code</b>	<b>2</b>
1.1	What should you test for? . . . . .	2
<b>2</b>	<b>Errors</b>	<b>2</b>
2.1	Compile errors . . . . .	2
2.2	Execution errors . . . . .	3
2.3	Logical errors . . . . .	3
<b>3</b>	<b>Tracing output</b>	<b>3</b>
<b>4</b>	<b>IDEs and debuggers</b>	<b>4</b>
4.1	Requirements for a good editor . . . . .	4
4.2	Profiling tools . . . . .	5
<b>5</b>	<b>Contract Programming</b>	<b>5</b>
<b>6</b>	<b>General tips if you get stuck</b>	<b>6</b>

## 1 Test your code

There are several ways to test your code but in the beginning you probably start with a table mostly called a test-matrix.

	Conditions	Expected Result	Actual Result
--	------------	-----------------	---------------

It is very important to document your test cases otherwise they are useless. If not one day your early-made test cases fails and you will spend a lot of time finding the problem again instead of fixing it.

### 1.1 What should you test for?

There are several things to test for but it depends a lot of what you want your code to do. Here are some general test cases that could make a difference.

- Boundary conditions and limits
  - *min value, max value, min value - 1, min value + 1*
  - Beginning and end in loops
- Small input:  $-1, 0, 1$ , empty string ""
- Big input: for 32-bit integers check  $2^{31} - 1$
- combinations of extreme values
- unexpected input ex. all input the same
- random input, mainly if you want to see if your program is robust enough to handle even wrong input.

Example of test case:

- test quicksort with a sorted array and a stupid choosen pivot-element

Test cases could be a part of a contract and in that case it is of most importance to have something like a "known limitations".

When you test your code do not do it interactively, you will probably end up making different input and then making your code to do different tests even if your intention was to test the same problem. Have your test cases in a file and have a correct solution in another file. To test your code you only write:

```
./program < program.in > program.out
diff program.out program.ans
```

## 2 Errors

### 2.1 Compile errors

Those are the errors we want to have. If we get them we can fix them before any user gets in contact with the program. Because we like these errors we want as many as we can get. With *gcc* use the flag `-Wall`, this flags means Warning All to show all warnings. There are other flags for specific warnings which could also be

used and it could be good because you do not want to show all warnings if you are not eventually going to fix them, in that case a lot of warnings could mean that important warnings are ignored as well.

If you sometimes use deprecated functions and get a lot of warnings about it you can use the `-Wall` flag combined with the `-Wno-deprecated`.

## 2.2 Execution errors

When you get an execution error this is often accompanied by something describing the problem, such as the text "Segmentation fault", a complete stack trace or a stack backtrace. If you have compiled the binary with debugging enabled you might get some additional information such as where in the application something went wrong.

There are differences between binaries compiled with debug flags (`-g` for `gcc`) and binaries compiled without. One difference is that debug binaries have padding in the memory. `[allocated][p][allocated]`, where `p` is padding. This can result in some hard to find errors where you get a segfault with the non debug binary and when you are trying to find where the segfault occurs, you get no segfault at all with the binary compiled with the debug flag because of the padding. Check the boundaries of your arrays and similar data structures. Add asserts for the input in functions and print tracing output to try to find where the segfault occurs.

## 2.3 Logical errors

Use a debugger - see the part about debuggers.

Use tracing output - see the part about tracing output.

## 3 Tracing output

If you have an advanced algorithm with several steps, it might be useful to print out some information in every step of the algorithm. This enables you to see which parts of the algorithm that do what you want them to do and where it starts to fail.

Another good advice is to make sure that the input to the application is being parsed correctly and has been inserted into your arrays or other data structures in the way you want it to. Do this by printing out the parsed input just after it has been read. When you know that your reading and parsing of the program input is correct, it will be much more easier to debug the rest of the application.

Print the information that is interesting for you, this can often be described with:

where: variable = value, variable = value, ...

Example:

```
foo(): x = 3, y = 4, z = 4294967295
```

## 4 IDEs and debuggers

There are many IDEs (Integrated development environment) that are available to users depending on what operating system is used. Some are better than others, depending on the language and desired functionality.

The most commonly used IDEs are :

- Microsoft Visual Studio - Visual Studio is a very good IDE and is a must have if one is developing on windows operating systems. The functionality is large and has all the little options and settings one may need. The debugger is probably the best of all other IDEs which makes debugging much easier as accessing and watching variables is simple and fast.
- Eclipse - A good IDE, which is best used when programming in Java. One can extend its function to C/C++ by using plugins. Debugging in C/C++ may be somewhat difficult depending on the size of the program, as Eclipse is written in Java and memory usage will rise upon debugging large chunks of code making it slowly crawl once one starts going through each line of code.
- Xcode - Only used on mac operating systems. Fairly simple and very easy to use which leaves the user to want a lot more options available especially when one is using the Xcode debugger.
- Netbeans - A very good IDE if one is programming in Java, if not the best Java based IDE. C/C++ support on the other hand is not as good as its Java counterpart. Debugger is also lacking in options and settings compared to Visual Studio.
- Code Blocks - C/C++ IDE only. Compiles and runs large chunks of code very fast and has a reliable debugger. Downside is that it is not updated regularly and one is forced to search for "nightly" builds (beta builds) in order to get access to the latest changes.

There are also some powerful text editors that one may use instead of an IDE, like XEmacs or Vim and combine it with command-line debuggers like GDB (GNU Project debugger) in order to be able to debug the written code. One can also expand it with graphical front-end for GDB by using DDD(Data Display Debugger) in order to get a more graphical presentation.

### 4.1 Requirements for a good editor

An editor needs to fulfill some basic requirements for it to be useful when coding. Requirements vary from person to person but the most commonly wanted ones are:

- Syntax highlighting - A must have when programing for longer periods of time. Makes it easier to recognize different categories and terms while writing code by using different colors for different elements.
- Auto-completion - Very useful when one wants to call functions of an object but can't remember the full name, or wants a list of all possible function calls that are available for the given object instead of digging through documentation.

- Indenting - A must have feature. Makes the code readable and helps with the code overview when having a lot of text on screen.
- Refactoring - Default find and replace options are no replacement for good refactoring support, when one wants to rename functions, variables, classes etc.
- Code folding - Helps the people which don't split the code into several other files, but instead have one file with a lot of code in it. Hiding the extra code is often helpful but still can't be compared to splitting the code into several other files.

## 4.2 Profiling tools

Once one has a code that is producing the desired output, the next problem is most often the speed at which the program is running and that there are no memory leaks. Depending on the code and algorithms used one can try to speed up the program by optimizing it. In order to do that one needs to know where the bottlenecks are and that can be done by using profiling tools. Profilers simply observe the program while it is running and are able to give you the information that is necessary for further improvements. Statistics like how many times a certain function is called and how long does it take compared to the rest of the program is very important, especially in decisions where one needs to combine several algorithms and choose the fastest one for the problem at hand.

Common profiling tools :

- Visual Studio profiler(C/C++) - Team Edition of the Visual Studio 2008 has a built in profiler that is very easy to use in order to collect the needed information about the speed bottlenecks in the program. It gives detailed information about functions used, how long they ran and which code blocks were called the most.
- AMD CodeAnalyst Performance Analyzer (C/C++) - Can be integrated with Visual Studio if one doesn't have the team edition. Very simple profiler that is able to represent almost the same information as the Visual Studio's counterpart.
- Gprof - GNU profiler that is used to determine which parts of a program are taking most of the execution time.
- Gcov - Is a tool that one can use in conjunction with GCC to test code coverage in the program.
- Valgrind - A profiler that has exceptional memory debugging and memory leak detection.
- Netbeans profiler (Java) - Netbeans profiler works only with Java and does not have support for profiling C/C++ code.

## 5 Contract Programming

Another way to make working code is using asserts. This uses conditions and if the program cannot satisfy a condition it exits. In this way you know that in all the

states where you use assert you know that your code is running correctly. Of course it is of most importance that you think what conditions your program is depended of and to make good and relevant conditions.

Assertions in C/C++ code<sup>1</sup>:

```
#include <cassert>
..  
assert(condition);
```

Example:

```
assert(i <= 10);
```

In java it is possible to give an error message that will displayed with the stacktrace. <sup>2</sup> :

```
assert condition : "string that will be printed if fail, other  
output or a function that returns some value.";
```

Example:

```
assert interval > 0 && intercal <= 1000 : interval;
```

## 6 General tips if you get stuck

If you get stuck there are a couple of things you can do to maximize your efficiency and solve the problem. One of the key thing is to "disconnect" and "reconnect" to clear your mind and empty your cache. This enables you to easier find another angle for attacking the problem or see why your current code doesn't work.

For example:

- You can take a short break.
- You can work on another problem.
- Tell someone else about the problem. Describe it in as much detail as possible and how you are trying to solve the problem. While doing this you might find yourself suddenly realizing what's wrong with your code. This approach forces your mind to go through the details of the problem and all the steps in your solution.
- Sleep on it.
- Rewrite your code. This approach also forces you to go through the steps of solving the problem and earlier mistakes are sometimes discovered.
- Refactor the code so that it is readable and nicely indented. Commenting on your code might also help since it also forces you to go through what you want the code to do.

---

<sup>1</sup><http://www.cppreference.com/wiki/c/other/assert>

<sup>2</sup><http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>