Eric Roberts
CS 106B

Handout #10
January 14, 2013

Debugging C++

Parts of this handout were written by Julie Zelenski.

> *As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*
>
> — Maurice Wilkes, 1949

The purpose of this handout is twofold: to give you a sense of the philosophy of debugging and to teach you how to use some of the practical tools that make debugging easier. To become an expert debugger, you will need to learn how to use the debugger environment that comes with your programming environment. The details of using a debugger differ enormously from system to system. This handout, therefore, is supplemented with two specialized handouts on the web site: one for debugging with Xcode on the Mac (Handout #10M) and one for debugging with Visual Studio on Windows (Handout #10W).* Even so, it is even more important that you understand how to employ general debugging strategies that transcend any particular platform.

The philosophy of debugging

> *With method and logic one can accomplish anything.*
>
> — Agatha Christie, *Poirot Investigates,* 1924

Debugging is one of the most creative and intellectually challenging aspects of programming. It can also be one of the most frustrating. To a large extent, the problems that people face debugging programs are not so much technical as they are psychological. To turn you into successful debuggers, I have to get you to think in a different way. There is no cookbook approach to debugging, although Nick Parlante's rules in Figure 1 will probably help. What you need is insight, creativity, logic, and determination.

The programming process leads you through a series of tasks and roles:

Design	—	Architect
Coding	—	Engineer
Testing	—	Vandal
Debugging	—	Detective

These roles require you to adopt distinct strategies and goals, and it is often hard to shift your perspective from one to another. Although debugging is extremely difficult, it can be done. It will at times take all of the skill and creativity at your disposal, but you can succeed if you are methodical and don't give up on the task.

---

* If you are using Linux, the best debugger is **gdb**, which has a different model than the ones for the other platforms. See the documentation at **http://www.gnu.org/software/gdb/documentation/** for details. There is also a tutorial written by Andrew Gilpin at **http://www.cs.cmu.edu/~gilpin/tutorial/**.

Figure 1. The Eleven Truths of Debugging

1. Intuition and hunches are great—you just have to test them out. When a hunch and a fact collide, the fact wins.

2. Don't look for complex explanations. Even the simplest omission or typo can lead to very weird behavior. Everyone is capable of producing extremely simple and obvious errors from time to time. Look at code critically—don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.

3. The clue to what is wrong in your code is in the values of your variables and the flow of control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts.

4. Be systematic. Be persistent. Don't panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.

5. If you code was working a minute ago, but now it doesn't—what was the last thing you changed? This incredibly reliable rule of thumb is the reason your section leader told you to test your code as you go rather than all at once.

6. Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable at a time. It makes the observed behavior much more difficult to interpret, and you tend to introduce new bugs.

7. If you find some wrong code which does not seem to be related to the bug you were tracking, fix the wrong code anyway. Many times the wrong code was related to or obscured the bug in a way you had not imagined.

8. You should be able to explain in Sherlock Holmes style the series of facts, tests, and deductions which led you to find a bug. Alternately, if you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your procedures cannot contain the bug. One of these arguments will contain a flaw since one of your procedures does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.

9. Be critical of your beliefs about your code. It's almost impossible to see a bug in a procedure when your instinct is that the procedure is innocent. In that case, only when the facts have proven without question that the procedure is the source of the problem will you be able to see the bug.

10. You need to be systematic, but there is still an enormous amount of room for beliefs, hunches, guesses, etc. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the procedures you suspect the most first. Good instincts will come with experience.

11. Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. Debugging code is more mentally demanding than writing code. The longer you try to track down a bug without success, the less perspective you tend to have. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep. You cannot debug when you are not seeing things clearly. Many times a programmer can spend hours late at night hunting for a bug only to finally give up at 4:00 A.M. The next day, they find the bug in 10 minutes. What allowed them to find the bug the next day so quickly? Maybe they just needed some sleep and time for perspective. Or maybe their subconscious figured it out while they were asleep. In any case, the "go do something else for a while, come back, and find the bug immediately" scenario happens too often to be an accident.

— Nick Parlante, Stanford University

Debugging is an important skill that you will use every day if you continue in computer science or any related field. Even though it is the last of the tasks in the list, it is certainly not the least important. Debugging will almost always require more time than the first three tasks combined. Therefore, you should always plan ahead and allow sufficient time for testing and debugging, as it is required if you expect to produce high-quality software. In addition, you should make a concentrated effort to develop these skills now, as they will be even more important as programs become more complicated later in the quarter and if you do any programming in your career.

General strategies

The following general strategies will also reduce both the time that debugging takes and the level of frustration you feel in the process:

- *Test your code incrementally rather than all at once.* When you work with any program that contains more than a couple of functions, you should test it on an incremental basis, even if doing so requires you to write additional test code that won't be part of your final submission. By testing each of your functions as you build it, you will be able to zero in on bugs before you put everything together. Realizing there's a bug in a ten-line function by seeing the results of a very simple test program is much easier than testing it in the context of the rest of your program. When you put that function together with the rest of your code, it is much harder to tell whether a bug is a in that piece of the code or somewhere else.
- *Guard against unpredictable behavior.* For programs that use random numbers, it is important to debug the program in a deterministic environment. A deterministic environment is one that allows a piece of code to behave exactly the same way every time you run it. For this reason, you should add a `setRandomSeed` call to the main program. You can take it out eventually, but not while you're debugging. You want your program to work the same way each time, so that you can always get back to the same situation.
- *Avoid value rigidity.* One of the most important techniques to master about debugging is keeping an open mind. So often, the problems that keep your code from working are easy to see if you can simply overcome the psychological blinders that keep you from seeing them. The more you are sure that some piece of code is correct, the harder it is to find the bugs in it.
- *Use particular caution when working with pointers.* Programs that use low-level pointers can be very tricky to debug, because errors in pointer handling can lead to very strange consequences. With pointers, you can, for example, overflow the end of an array or use a variable after it has been freed. In many cases, the problems that such coding errors create don't show up when you're executing the code that's wrong, but at some later point when everything just seems to break down for no reason. Because such bugs are so hard to find and fix, it is important to have your memory model straight before you start coding. In fact, it often pays to draw a memory diagram to make sure you understand what you want, and compare it to what you have actually coded.

- *Make sure you know what your program is doing.* The most common psychological block that people exhibit during debugging is trying to divine why a program *isn't* working without first trying to understand what the program is in fact doing. A buggy program can tell you a lot. In most cases, as soon as you discover why the program is behaving as it is, that will lead you to understand what it *ought* to be doing instead.

Debugging with `cout`

One of the easiest ways to tell what your program is doing is to add **`cout`** statements to any parts of the code where you're not quite sure what is going on. If you include variables in the **`cout`** statement, you can also follow the computation and make sure that those values have the values you expect. As a general rule, you should make sure that each of your **`cout`** statements prints a complete line. In most implementations of the **`iostream`** interface, the data passed to **`cout`** is stored in a temporary buffer whose contents are not actually displayed until the **`endl`** appears or the program calls a function requiring input such as **`getInteger`**.

Although **`cout`** statements are quick and easy, they aren't sufficient to debug complicated programs. For those, you need a real debugger, as described in the companion handouts.