

Golden Rules for Programming and Debugging

The following are some key principles to keep in mind when programming, and more specifically, debugging.

1) **Never blindly make changes in an attempt to remove an error.**

This is perhaps the most important thing to remember when writing a program. Upon encountering an error, many programmers (particularly new ones) will quickly make changes in a haphazard manner in an attempt to clear the error. This is what is called "hacking." Usually, this is done in place of trying to **understand** what caused the error in the first place. After repeated attempts, the error may disappear - or at least, appear to do so. In many cases, this is only superficial - the programmer has found a way to circumvent the problem in one instance, but the underlying error still exists. Eventually, it will crop up again in another part of the program, and another hack will be applied. As this process continues, more and more hacks are added, in turn making the code longer, more complicated, and harder to understand. It can eventually progress to the point that the programmer has backed his/herself into a corner - there is no way to proceed without removing all of the hacks and rewriting the code. Remember - *Just because it works, doesn't mean it's right!*

An example of this would be a programmer forgetting a closing French brace for a conditional or a loop, resulting in a syntax (compiler) error. Simply trying to insert a French brace will eventually remove the compiler error, though the brace may not be in the correct place. As we have seen, placing braces incorrectly (or not placing them at all) can alter the *meaning* of our code, even though syntactically it is correct. The programmer has taken a syntax error (the easier to deal with) and turned it into a logical error (the hardest). Then, when the output isn't what is expected, the programmer will begin looking at the logic of the code, probably make some more hacks, and wind up chasing his/her tail for hours (when the initial error was a very simple one to correct).

Before making a change, you should be able to answer the following questions:

- What is it about this particular code that is causing the error?
- What will the new code do to fix this error?

The answers to these questions may only be hypothesis, and they may be wrong, but they mean that you are attempting to logically justify your decision. When making a change that you may be less than certain about, comments can be used. Simply comment out the original code and add the new code. If you discover that the change is incorrect, you still have your original code to revert to - simply uncomment the original code and delete the change (or comment it out if you believe that you may eventually use it).

2) Progress isn't necessarily measured in terms of compiler errors.

This can be quite counterintuitive and very troubling for new programmers. Often times one error will "hide" other errors further along in the program. When that error is removed, the compiler can then correctly process the rest of the program, revealing the "hidden" errors. When the programmer corrects the first error and recompiles, the *number* of error actually increases! The immediate (and incorrect) assumption is that they have made things worse, when in fact they removed a very real error. Of course, this doesn't mean that a change that produces more errors is necessarily correct, but it is wrong to think immediately assume that it's wrong. Going back to the first rule, understanding **why** you're making a change goes a long way towards knowing which direction you're moving.

3) A compiled program is not a working program.

This is a corollary to the previous rule. I've often had students tell me that they're almost done with their program - "I've only got two more errors to get rid of and then I'll be done." If you program with this mindset, you will most definitely miss deadlines. There is a saying in computer science (with many variations) that goes something like this: "Ninety percent of your time is spent debugging the last ten percent of your code." Remember, syntax (i.e. compiler) errors are by far the easiest to find and remove - the compiler tell you they exist, about where they are, and sometimes even offers suggestions on how to fix them! Runtime, and more importantly, logical errors are what really consumes your time as a programmer. Remember, until you remove all the syntax errors and get the program to compile, you can even **begin** to search for runtime and logical errors!

4) Tackle errors in order.

When you try to compile and get a list of errors, always start at the top of the list. Remember, errors cascade - fixing a simple error or two at the beginning of your program can reduce the total number of errors from 112 to 8. Fix an error or two and then recompile - it will save you a lot of time trying to remove errors that don't really exist.

5) Use incremental development.

Incremental development, top-down design, divide and conquer - whatever you call it is probably the most important problem solving concept to take away from this course. Trying to tackle the problem in one bite is futile - solve the small problems first and build upwards.

6) Understand the different compiler errors.

As you program, you will continue to see the same compiler errors appearing again and again - learn what they mean. A few of the more common errors are:

- **Undeclared identifier** - a variable or constant was used without being declared.
- **Unexpected end of file** - a French brace is missing somewhere in the program, resulting in no closing brace for `main()`.
- **Binary '>>' ...** (error is much longer) - an error of this type means that you reversed the extraction and insertion operators.

7) Use pictures and examples to help flesh out a problem.

Drawing diagrams - particularly when using filestreams - can be invaluable. Manually stepping through simplified examples can help with program logic. Statements in C++ are executed in order (with function calls being somewhat of an exception) - if you follow the order, you'll be able to answer questions like, "Why did this line get printed out ten times instead of one?"

8) Use `cout` statements and the debugger.

Use `cout` to print out variable values or statements like "Entering while loop" or "Starting function `foo()`" to indicate the current location in the program. For more complicated problem, use the debugger - it can save hours of time!

9) "The Devil's In The Details"

Programming is a very detail-oriented activity. Small typos and other minor mistakes can have drastic consequences. Some common mistakes to be on the lookout for include:

- Capitalization and spelling errors
- Incorrect input file - make sure that the input file has the correct name, both in your program and on your computer. Make sure that you have the **correct** input file. On more than one occasion I have helped a student with input trouble where they had the input file with the correct name and in the correct folder but the file itself was empty or contained something other than the actual input data.
- Statements of the type *if* ($x = 3$) or *while* (*InStream*);
- Using `'\n'` or `'\t'` instead of `'\n'` or `'\t'`
- Forgetting to use French braces within conditional statements or loops
- Priming read - see slides 4.22 - 4.25