Notes to Accompany Debugging Lecture

Jamie Blustein

28 April 2002

Introduction

This lecture was originally written for Turbo Pascal, then updated for the first ANSI C standard, then object-free C++. It does not address anything specific about object-oriented or real-time programming. I am confident that the general advice will apply well to those types of programming too.

When you wrote your program you thought it would work, but now you've found that it doesn't. 'What's wrong? How can you fix it?' That's debugging. Debugging is not the same as fixing syntax errors that prevent compilation.

'The term originated with the first tube computers': Talk about Grace Hopper and the moth (from 1947).

To Solve a Problem You Must Understand It

Tracing the execution will help you to understand what is happening with your program. If you know which part is wrong then you can trace just that part.

Clarity

We call the written form of our program 'code', as in difficult-to-understand. It must be written in a language that the computer can process but that does not mean that is has to be written so that we cannot understand it too. It is hard enough to read code when we think we know what we are instructing the computer to do. But especially when we know that the program doesn't work right then we need to have clear code and clear output.

Always print to standard error

Your output will appear 'immediately' — it won't get muddled up with some other output. If you use cout for debugging then you can be misled easily.

Always print the function name (see example)

See story about Bill (on page 4)

See also annotated example (starting on page 6)

DEBUG levels

Use larger numbers for more detail or for parts you think work now (but you don't want to remove the DEBUG code just in case you're wrong).

Debuggers

Debuggers can be mesmerizing. I've seen it happen.

Programs will sometimes behave differently in a debugger than in real-life.

There's never time...

It sounds trite but it is true.

Programmers will jump into writing a program or trying to fix a broken one without thinking. You can save yourself much fruitless labour and frustration by planning ahead. The better your plan the less likely your program will be to fail and the more likely you'll be to fix it if it does fail.

Trace Statements

A true-life story

My friend Bill *didn't* do this and spent $1\frac{1}{2}$ days debugging the wrong part of his program. He was printing 'Hi' or something to show when the flow-of-control had reached a statement. The problem was that he had two 'Hi's in different parts of the program. He was seeing the second one – the first was never reached.

I found the problem in one minute by changing 'Hi' to something more descriptive ('inner for loop' or something). Bill had to buy me lunch and I didn't even know the language he was programming in.

If that technique can help a professional programmer save $1\frac{1}{2}$ days, think how the power of debugging can help you.

An ounce of analysis...

Another true-life tale

I was in a computer lab working on a brutal assignment (a recursive BNF parser in Modula 2 — don't ask, trust me it's not super easy) and the grader comes in every 2 or 3 hours to say that he's changing the assignment (again!). Never mind that is beyond his authority. Suffice it to say that I was stressed and could not get my program to work right.

By the way I was sitting beside a fellow who was also in the class. I'll call him Stu Dent (not his real name). Stu was a rower and had practice at 6 the next morning.

Anyway, for some reason I noticed that the program was not due the next day but rather the day after that. It was late and I was tired. I printed a copy of my program took it with me and left. *I slept.*

In the morning, I scribbled an outline of a correct program on a piece of paper, while I waited for my cereal to stop making noise. Then, on my printout I crossed off the lines that did not belong and added the ones that were missing. When I got back to school, Stu was at the same terminal — he'd been there all night, and still didn't have a working program. I typed in my changes and it was perfect. Stu finished his a day later — a 10% penalty — and missed rowing practice.

What's the moral of this true story? Not sleep whenever you have a problem *but*:

- work away from the computer screen

  it is too demanding and you can't see the whole program at once

- think about your program and what it should do
- get enough sleep
  - sleep deprivation is a common torture technique (no joke)!
  - Don't do it to yourself. It is very difficult to think clearly when you are so deprived.

```
int ❶ question(int left, int op, int right) {
/* question()
 * PRE:     ... details omitted ...
 * POST:    The user has been prompted to enter ...
 * RETURNS:  0 => they entered the wrong answer
 *           1 => they entered the right answer
 *          -1 => something went wrong (bad op)
 * NOTES:   Called by add(), divide(), ...
 */❷

 int answer; // the correct answer
 int guess;  // what the user enters as the right answer

 if (DEBUG > 1) {
   cerr << "question(): called with left==" << left ❸
        << ❹ "right==" << right << " op==" << op << endl;
 } ❺

 switch(op) {
   case 1: answer = left + right;
           if (DEBUG) ❻
              cerr << "   question(): op says add " << endl;
           break;
   // ... other cases omitted ... //
   default: ❼
       cerr << "ERROR In question(): impossible op "
            << "(op==" << op << ")" << endl;
       return -1;
 } // switch

 if (DEBUG)
    cerr << "   question(): answer==" << answer << endl;

 // ... code omitted ...

 if (DEBUG > 1)
    cerr << endl << endl;

 return (guess == answer);
} // question()
```

❶ From an old assignment (a child's arithmetic quiz)

❷ Good documentation habits include always explaining the purpose of variables, and using sensible names that are not too similar. I like to see pre- and post-conditions in code that I grade.

The exact form of these documentation habits are a matter of style but you should always provide the same information. (Kernighan & Pike's *The Practice of Programming* (Addison-Wesley, 1999) has another good style.)

❸ It is sometimes useful to show the parameters passed to a function. When you do, also show the names so you can't accidentally pass them in the wrong order. D'oh!

❹ Note the continuation here

❺ I prefer to use {}s for all `if` statements. That way I can easily insert more code without forgetting the {.

❻ There's no { here. That's a problem waiting to happen.

❼ Notice that I always use a `default` case with a `switch`. It takes very little time to code but can save a great deal of time later.