How to Get Your Code Bug-Free . . . and Keep it That Way!

Mech 587/588

Version Date: January 13, 2023

Broadly defined, a bug is something that makes it so that your code doesn't give correct output for valid input. That definition includes a few things we don't ordinarily think of as bugs, because we find them in very different ways, but it's a useful generalization.

Techniques for ensuring your code gives correct output can be divided into two categories: testing and debugging, each of which comes in more than one flavor.

1 Before You Run Your Code

Before you even attempt to run your code, you can already remove some bugs from it. The most obvious of these are compilation errors, which are definitely bugs in the sense defined above! You don't even have to look very hard for these, because your compiler tells you about them automatically.

There are at least two other things you can do to find potential problems in your code before you ever run it.

1.1 Automated Static Code Analysis

Static code analysis examines your source code for potential errors. For instance, if you type

```
if (i = 3) {
  ...
}
```

in C, that's correct code, so your compiler should accept it. But it's probably not what you meant; you probably meant

```
if (i == 3) {
  ...
}
```

Static analysis will find questionable constructs like this, and a whole lot more. If you're using an Integrated Development Environment (like Visual Studio, CodeBlocks, or eclipse), your IDE will do static analysis for you on request. If not, standalone tools like cppcheck will do the same thing for you, though it'll be harder to work back and forth between the output of those tools and your source code, compared with using a tool in your IDE.

1.2 Compiler Warning Flags

Your compiler will also happily warn you about code idioms that are questionable. How you turn these on depends on your compiler, and whether you're running the compiler directly or via an IDE.

For my research group's codes, we build using gcc/g++, and we turn on almost all of the warnings available:

```
-Wall -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings
-Woverloaded-virtual -W -Wshadow -Winline -Wctor-dtor-privacy
-Wnon-virtual-dtor -Wreorder -Wold-style-cast -Wsign-promo
```

Yes, that's right: -Wall doesn't turn on all warnings.

Both of these approaches will warn you about things that are a Bad Idea to have in your code. Fixing them will eliminate things that could cause problems. Yes, the first time you do this on a sizable piece of code, you'll get a whole lot of warnings. But your IDE can help you keep track of those efficiently, so they're relatively easy to find and fix. And once you figure out what *bad* coding habits caused all of those warnings, it's not hard to develop good habits that produce no warnings, unless you made a mistake. That's because you've learned to avoid questionable constructs in your code.

2 Assertions

One of the best things you can do, where feasible, is to include *assertions* in your code. These are statements about pre-conditions or post-conditions for your function that you know must be true. For instance, for a compressible flow flux function, you would want to check that the density and pressure are positive, because that must be true physically.

Assertions are typically implemented in terms of a pre-processor macro, which in turn is defined conditionally. By default, the assertion will terminate the program, with an informative error message, if its argument is false. When NDEBUG is defined, the assert macro expands to nothing, so there's zero time penalty for properly compiled release-mode code.

Here's an example:

```
#include <assert.h>

double acos(double arg) {
  assert(arg >= -1 && arg <= 1);
  [... do some computing ...]
  assert(result >= 0 && result <= M_PI);
  return result;
}
```

In this example, we check both the pre- and post-conditions. The value we're taking the arccos of must lie between -1 and 1, inclusive, and the result must lie between 0 and π, inclusive.[1]

3 Unit Testing

It's easier to identify and fix a bug if you know that it's within some very small chunk of code. One way to easily narrow down bugs to a single, short routine is to *test* single, short routines.

For example, if you have a macro named TEST defined, you could write something like:

```
double result = acos(0);
TEST(result == M_PI/2);
result = acos(1);
TEST(result == 0);
result = acos(-0.5);
TEST(result == 2*M_PI/3);
```

and so on.

Yes, these are simple, obvious tests; that's the point. But it wouldn't be surprising if your code failed some of these originally. Your goal is to test as much of the function's behavior as possible with a relatively small number of tests that run very quickly. The *reason* you want this to run quickly is that you want to run your unit tests frequently; ideally, every time you compile. This way, if you break something in your code, you know about it almost immediately.

You can write your own TEST macro, or you can use one of the many unit testing frameworks out there. These have facilities for defining tests, for checking results, for helping set up data for a group of tests, and for accumulating the number of tests that pass and fail. Perhaps the two most commonly used for C/C++ are the frameworks from Boost and Google, both of which are open source.

One school of thought among serious programmers, called *test-driven development* (TDD), holds that you shouldn't just write tests, you should write tests *before* you write the code that you're going to test. Having defined the tests that your code must pass, you then write the actual code, with the goal of including only those things required to pass the tests. If there's additional functionality that needs to be in the code, your tests aren't complete. Also, if you aren't sure what tests to write in advance, that's a sign that you don't understand what you're trying to do.

I can definitely understand this point of view. I'm sure that it leads to code that has more, simpler functions, that aren't trying to do multiple things in one routine, because otherwise, it's hard to write tests. I've never quite had the discipline to do this from start to finish on a project, so I don't have first-hand experience with it, but I know that a group at MIT has written a high-order finite-element flow solver from scratch using this approach as fully as they can manage, and they're happy with how that's gone.

[1] This implementation isn't compliant with the ANSI standard for acos; out-of-range input values are supposed to result in a NaN being returned.

4 Debugging

The techniques described above don't actually fix any bugs for you; they're designed to make it easy to identify bugs and narrow down their location. At some point, you have a pretty good idea where the error is, and it's time to get serious about pinpointing the cause. There are three commonly used techniques here, which I'll list in order of increasing power.

Reading the code. The idea here is that when you read a section of code that you know has a bug in it, you're likely to find the error. Eventually, of course, whatever approach you take, you end up here, but in my opinion, you have to have a very good idea where the problem is (that is, relatively few lines of code where the problem can be) and also what the problem is (that is, why the code is mis-calculating something). In other words, the efficiency of this approach drops rapidly with the size of code chunk you're reading.

Printing data. Here, you're printing out intermediate results so that you can compare those to what the values should be. This can be pretty effective, assuming that you're really looking in the right place and that you print the proper variables. An extension of this is to output data files for visualization. Essentially, what printing data does for you is to help narrow your search for the bug.

Using a debugger. You can think of this approach as a combination of the other two, where you can interactively decide what data to print, as well as choosing (by setting breakpoints) where to start monitoring code behavior. Since your debugger shows you the source code that corresponds to where execution has stopped, you can read the code as needed. You can also step through one line at a time while monitoring data. I spend a lot of time working with my debugger, as I find this the most efficient way to find and fix bugs.

Before you can use a debugger, you have to compile with debugging in mind. This means you need to recompile the code[2] for debugging. You'll typically want to use the compiler flags -O0 -g for this. -g turns on debugging, and -O0 turns off optimization; the latter prevents the compiler from re-ordering operations, which in turn makes it easier to step through code and know what has for sure been done by your program and what hasn't.

5 Regression Testing

Suppose that your code now works for some simple case (let's say, flow in a box with a moving top on a uniform Cartesian mesh). You're now going to modify the code to do something more sophisticated (support curvilinear meshes, for example), but you don't want to lose capability that you already had.

This is where regression testing comes in. A *regression* is a scenario where something that your code once did successfully no longer works. *Regression testing* is the process of testing your code so that you can demonstrate whether there are any regressions.

[2] Or at least, the parts you want to debug, but it's generally best to do everything.

A good regression test has clearly defined inputs and clearly defined, quantitative outputs for which you have a historical baseline. Preferably, inputs are read from a file or from the command line, not interactively, because this maximizes the chance that you actually are running the same test. Your output can be a success/failure result if you're comparing the result to the historical result internally; I don't really recommend this for things more complex than "the code ran and successfully produced a result", because then your code has to have numerical results coded in for a variety of cases. Instead, you probably want to have some output (either to the console or to a file) that can be compared to the True Result afterwards.

For regression tests to maximally useful, they should be automated, which means they should be scripted. Otherwise, you as the human have to compare results for all your test cases. Having a script that runs cases and compares output reduces the effort of running the tests significantly, and therefore makes you likely to run them more often.[3]

To give you a sense of reasonable practice, my group's research flow solver has something like 400 test cases at present. These are designed to test all major features of the code (though we still miss some). Many of these, especially the recently added cases, are cases from past papers or theses. The cases are bundled by flow physics model, and run in parallel; the whole process takes about an hour.

6 Source Code Management

Source code management (SCM) allows you to keep track of multiple versions of your code and to collaborate more easily with others on large projects. The best-known example, currently, is github, though this (including git run locally on your own machine) is far from the only option.

When you're working on a project by yourself, the advantage to SCM is that you don't have to worry about losing that old version of your code. Before you start working on that risky new feature, or on re-implementing some existing feature, you *commit* code to the SCM repository. This gives you a checkpoint to fall back on in the worst case. You can also go back and run that one case that no longer works in the current version; most SCM systems will even help you with the process of figuring out when you broke that old test case, so that you can fix the bug.

If that were all SCM could do for you, that would already be enough. But wait! There's more!

If you're working on a project with other developers, SCM systems will allow you to create separate *branches*, so that you and your colleague are working on your new features independently, including each of you being able to commit your changes at any time without affecting the other. Later on, when your code is stable and tested, you can *merge* the two branches together. SCM systems make these process as automatic as it can be made. If your changes and your colleague's are in different places in the source code, the SCM will combine these changes with no problem; even changes in different places in the same file are easy. The hard cases — where a human must intervene — are the ones where two people have change

[3] We'll come back to another advantage in Section 6.

the same lines of code. Then the second person to try to merge their code will have to figure out how to resolve those conflicts.

Regardless of which mode you're using an SCM in, commit early and commit often. "I wish I hadn't made that commit," said nobody ever. Likewise, in a multiuser setting, merge often. As in, every time you have code that passes tests, even if you're only partway through implementing some big feature. As an example, if you're adding a new turbulence model to a large project, merge when you have a correct flux function; don't wait for complete end-to-end verification testing and screaming fast convergence.

One thing your SCM won't do for you automatically is run tests. However, you can add scripts that are run at commit time that run tests and reject a commit if it fails testing. Somewhat more sophisticated than that, you can run a *Continuous Integration* server (jenkins is a prime example) to do this for you. For instance, my group has jenkins configured to accept all commits to user branches. It then tests whether the new commit will merge cleanly with the main development branch. If it does, then jenkins builds and tests the merged code[4]; if the tests pass, then the merged code is committed to the main development branch. Note that this is the *only* route to commit something to the main development branch; not even I have permissions to commit something directly to that branch in the central repository!

[4]This is where it's massively useful to have a script for your regression tests.