3.10 — Finding issues before they become problems

👤 **ALEX** · 🕐 **APRIL 23, 2022**

When you make a semantic error, that error may or may not be immediately noticeable when you run your program. An issue may lurk undetected in your code for a long time before newly introduced code or changed circumstances cause it to manifest as a program malfunction. The longer an error sits in the code base before it is found, the more likely it is to be hard to find, and something that may have been easy to fix originally turns into a debugging adventure that eats up time and energy.

So what can we do about that?

Don't make errors

Well, the best thing is to not make errors in the first place. Here's an incomplete list of things that can help avoid making errors:

- Follow best practices
- Don't program when tired
- Understand where the common pitfalls are in a language (all those things we warn you not to do)
- Keep your programs simple
- Don't let your functions get too long
- Prefer using the standard library to writing your own code, when possible
- Comment your code liberally

Refactoring your code

As you add new capabilities to your programs ("behavioral changes"), you will find that some of your functions grow in length. As functions get longer, they get both more complex and harder to understand.

One way to address this is to break a single long function into multiple shorter functions. This process of making structural changes to your code without changing its behavior (typically in order to make your program more organized, modular, or performant) is called **refactoring**.

So how long is too long for a function? A function that takes up one vertical screen worth of code is generally regarded as far too long -- if you have to scroll to read the whole function, the function's comprehensibility drops significantly. But the shorter the better -- functions less than ten lines are good. Functions that are less than five lines are even better.

> **Key insight**
>
> When making changes to your code, make behavioral changes OR structural changes, and then retest for correctness. Making behavioral and structural changes at the same time tends to lead to more errors as well as errors that are harder to find.

An introduction to defensive programming

Errors can be not only of your own making (e.g. incorrect logic), but also occur when your users use the application in a way that you did not anticipate. For example, if you ask the user to enter an integer, and they enter a letter instead, how does your program behave in such a case? Unless you anticipated this, and added some error handling for this case, probably not very well.

Defensive programming is a practice whereby the programmer tries to anticipate all of the ways the software could be misused, either by end-users, or by other developers (including the programmer themselves) using the code. These misuses can often be detected and then mitigated (e.g. by asking a user who entered bad input to try again).

We'll explore topics related to error handling in future lessons.

Finding errors fast

Since not making errors is difficult in large programs, the next best thing is to catch errors you do make quickly.

The best way to do this is to program a little bit at a time, and then test your code and make sure it works.

However, there are a few other techniques we can also use.

An introduction to testing functions

One common way to help uncover issues with your program is to write testing functions to "exercise" the code you've written. Here's a primitive attempt, more for illustrative purposes than anything:

```cpp
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

void testadd()
{
    std::cout << "This function should print: 2 0 0
-2\n";
    std::cout << add(1, 1) << ' ';
    std::cout << add(-1, 1) << ' ';
    std::cout << add(1, -1) << ' ';
    std::cout << add(-1, -1) << ' ';
}

int main()
{
    testadd();

    return 0;
}
```

The testadd() function tests the add() function by calling it with different values. If all the values match our expectations, then we can be reasonably confident the function works. Even better, we can keep this function around, and run it any time we change function add to ensure we haven't accidentally broken it.

This is a primitive form of **unit testing**, which is a software testing method by which small units of source code are tested to determine whether they are correct.

As with logging frameworks, there are many 3rd party unit testing frameworks that can be used. It's also possible to write your own, though we'll need more language features at our disposal to do the topic justice. We'll come back to some of this in a future lesson.

An introduction to constraints

Constraints-based techniques involve the addition of some extra code (that can be compiled out in a non-debug build, if desired) to check that some set of assumptions or expectations are not violated.

For example, if we were writing a function to calculate the factorial of a number, which expects a non-negative argument, the function could check to make sure the caller passed in a non-negative number before proceeding. If the caller passes in a negative number, then the function could immediately error out rather than producing some indeterminate result, helping ensure the problem is caught immediately.

One common method of doing this is via assert and static_assert, which we cover in lesson <u>7.17 -- Assert and static_assert</u> <u>(https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/)</u>.

Shotgunning for general issues

Programmers tend to make certain kinds of common mistakes, and some of those mistakes can be discovered by programs trained to look for them. These programs, generally known as **static analysis tools** (sometimes informally called linters) are programs that analyze your code to identify specific semantic issues (in this context, static means that these tools analyze the source code). The issues found by static analysis tools may or may not be the cause of any particular problem you are having, but may help point out fragile areas of code or issues that can be problematic in certain circumstances.

You already have one static analysis tool at your disposal -- your compiler! In addition to ensuring your program is syntactically correct, most modern C++ compilers will do some light static analysis to identify some common problems. For example, many compilers will warn you if you try to use a variable that has not been initialized. If you haven't already, turning up your compiler warning and error levels (see lesson <u>0.11 -- Configuring your compiler: Warning and error levels</u> <u>(https://www.learncpp.com/cpp-tutorial/configuring-your-compiler-warning-and-error-levels/)</u>) can help surface these.

<u>Many static analysis tools exist</u> <u>(https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C,_C++)</u>, some of which can identify over 300 types of programming errors. And on large programs, this can surface tens or hundreds of potential issues. On our small academic programs, use of a static analysis tool is optional (but may help you find areas where your code is non-compliant with best practices). On larger non-academic programs, use of a static analysis tool is highly recommended.

> **Best practice**
>
> Use a static analysis tool on your programs to help find areas where your code is non-compliant with best practices.

> **For Visual Studio users**
>
> Visual Studio 2019 comes with a built-in static analysis tool. You can access it via Build > Run Code Analysis on Solution (Alt+F11).

B **U** **URL** **INLINE CODE** **C++ CODE BLOCK** **HELP!**

Leave a comment...

👤 Name*

@ Email* ?

🐞 Find a mistake? Leave a comment! ?

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: 🔔 **POST COMMENT**

22 COMMENTS

Newest ▾

Ⓧ