9.1 — Introduction to testing your code

So, you've written a program, it compiles, and it even appears to work! What now?

Well, it depends. If you've written your program to be run once and discarded, then you're done. In this case, it may not matter that your program doesn't work for every case -- if it works for the one case you needed it for, and you're only going to run it once, then you're done.

If your program is entirely linear (has no conditionals, such as *if statements* or *switch statements*), takes no inputs, and produces the correct answer, then you're probably done. In this case, you've already tested the entire program by running it and validating the output. You may want to compile and run the program on several different systems to ensure it behaves consistently (if it doesn't, you've probably done something that produces undefined behavior that just happens to work anyway on your initial system).

But most likely you've written a program you intend to run many times, that uses loops and conditional logic, and accepts user input of some kind. You've possibly written functions that may be reusable in other future programs. You may have experienced a bit of **scope creep**, where you added some new capabilities that were originally not planned for. Maybe you're even intending to distribute this program to other people (who are likely to try things you haven't thought of). In this case, you really should be validating that your program works like you think it does under a wide variety of conditions -- and that requires some proactive testing.

Just because your program worked for one set of inputs doesn't mean it's going to work correctly in all cases.

Software testing (also called **software validation**) is the process of determining whether or not the software actually works as expected.

The testing challenge

Before we talk about some practical ways to test your code, let's talk about why testing your program comprehensively is difficult.

Consider this simple program:

```
#include <iostream>

void compare(int x, int y)
{
    if (x > y)
        std::cout << x << " is greater than " << y << '\n'; // case 1
    else if (x < y)
        std::cout << x << " is less than " << y << '\n'; // case 2
    else
        std::cout << x << " is equal to " << y << '\n'; // case 3
}

int main()
{
    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;

    std::cout << "Enter another number: ";
    int y{};
    std::cin >> y;

    compare(x, y);

    return 0;
}
```

Assuming a 4-byte integer, explicitly testing this program with every possible combination of inputs would require that you run the program 18,446,744,073,709,551,616 (~18 quintillion) times. Clearly that's not a feasible task!

Every time we ask for user input, or have a conditional in our code, we increase the number of possible ways our program can execute by some multiplicative factor. For all but the simplest programs, explicitly testing every combination of inputs becomes impossible almost immediately.

Now, your intuition should be telling you that you really shouldn't need to run the above program 18 quintillion times to ensure it works. You may reasonably conclude that if case 1 works for one pair of `x` and `y` values where `x > y`, it should work for any pair of `x` and `y` where `x > y`. Given that, it becomes apparent that we really only need to run this program about three times (once to exercise each of the three cases in function `compare()`) to have a high degree of confidence it works as desired. There are other similar tricks we can use to dramatically reduce the number of times we have to test something, in order to make testing manageable.

There's a lot that can be written about testing methodologies -- in fact, we could write a whole chapter on it. But since it's not a C++ specific topic, we'll stick to a brief and informal introduction, covered from the point of view of you (as the developer) testing your own code.

In the next few subsections, we'll talk about some *practical* things you should be thinking about as you test your code.

Test your programs in small pieces

Consider an auto manufacturer that is building a custom concept car. Which of the following do you think they do?
a) Build (or buy) and test each car component individually before installing it. Once the component has been proven to work, integrate it into the car and retest it to make sure the integration worked. At the end, test the whole car, as a final validation that everything seems good.
b) Build a car out of all of the components all in one go, then test the whole thing for the first time right at the end.

It probably seems obvious that option a) is a better choice. And yet, many new programmers write code like option b)!

In case b), if any of the car parts were to not work as expected, the mechanic would have to diagnose the entire car to determine what was wrong -- the issue could be anywhere. A symptom might have many causes -- for example, is the car not starting due to a faulty spark plug, battery, fuel pump, or something else? This leads to lots of wasted time trying to identify exactly where the problems are, and what to do about them. And if a problem is found, the consequences can be disastrous -- a change in one area might cause "ripple effects" (changes) in multiple other places. For example, a fuel pump that is too small might lead to an engine redesign, which leads to a redesign of the car frame. In the worst case, you might end up redesigning a huge part of the car, just to accommodate what was initially a small issue!

In case a), the company tests as they go. If any component is bad right out of the box, they'll know immediately and can fix/replace it. Nothing is integrated into the car until it's proven working by itself, and then that part is retested again as soon as it's been integrated into the car. This way any unexpected issues are discovered as early as possible, while they are still small problems that can be easily fixed.

By the time they get around to having the whole car assembled, they should have reasonable confidence that the car will work -- after all, all the parts have been tested in isolation and when initially integrated. It's still possible that unexpected issues will be found at this point, but that risk is minimized by all the prior testing.

The above analogy holds true for programs as well, though for some reason, new programmers often don't realize it. You're much better off writing small functions (or classes), and then compiling and testing them immediately. That way, if you make a mistake, you'll know it has to be in the small amount of code that you changed since the last time you compiled/tested. That means fewer places to look, and far less time spent debugging.

Testing a small part of your code in isolation to ensure that “unit” of code is correct is called **unit testing**. Each **unit test** is designed to ensure that a particular behavior of the unit is correct.

Best practice

Write your program in small, well defined units (functions or classes), compile often, and test your code as you go.

If the program is short and accepts user input, trying a variety of user inputs might be sufficient. But as programs get longer and longer, this becomes less sufficient, and there is more value in testing individual functions or classes before integrating them into the rest of the program.

So how can we test our code in units?

Informal testing

One way you can test code is to do informal testing as you write the program. After writing a unit of code (a function, a class, or some other discrete “package” of code), you can write some code to test the unit that was just added, and then erase the test once the test passes. As an example, for the following isLowerVowel() function, you might write the following code:

```
#include <iostream>

// We want to test the following function
bool isLowerVowel(char c)
{
    switch (c)
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true;
    default:
        return false;
    }
}

int main()
{
    // So here's our temporary tests to validate it works
    std::cout << isLowerVowel('a') << '\n'; // temporary test code, should produce 1
    std::cout << isLowerVowel('q') << '\n'; // temporary test code, should produce 0

    return 0;
}
```

If the results come back as `1` and `0`, then you’re good to go. You know your function works for some basic cases, and you can reasonably infer by looking at the code that it will work for the cases you didn’t test (‘e’, ‘i’, ‘o’, and ‘u’). So you can erase that temporary test code, and continue programming.

Preserving your tests

Although writing temporary tests is a quick and easy way to test some code, it doesn’t account for the fact that at some point, you may want to test that same code again later. Perhaps you modified a function to add a new capability, and want to make sure you didn’t break anything that was already working. For that reason, it can make more sense to preserve your tests so they can be run again in the future. For example, instead of erasing your temporary test code, you could move the tests into a testVowel() function:

```
#include <iostream>

bool isLowerVowel(char c)
{
    switch (c)
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true;
    default:
        return false;
    }
}

// Not called from anywhere right now
// But here if you want to retest things later
void testVowel()
{
    std::cout << isLowerVowel('a') << '\n'; // temporary test code, should produce 1
    std::cout << isLowerVowel('q') << '\n'; // temporary test code, should produce 0
}

int main()
{
    return 0;
}
```

As you create more tests, you can simply add them to the `testVowel()` function.

Automating your test functions

One problem with the above test function is that it relies on you to manually verify the results when you run it. This requires you to remember what the expected answer was at worst (assuming you didn't document it), and manually compare the actual results to the expected results.

We can do better by writing a test function that contains both the tests AND the expected answers and compares them so we don't have to.

```
#include <iostream>

bool isLowerVowel(char c)
{
    switch (c)
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true;
    default:
        return false;
    }
}

// returns the number of the test that failed, or 0 if all tests passed
int testVowel()
{
    if (!isLowerVowel('a')) return 1;
    if (isLowerVowel('q')) return 2;

    return 0;
}

int main()
{
    int result { testVowel() };
    if (result != 0)
        std::cout << "testVowel() test " << result << " failed.\n";
    else
        std::cout << "testVowel() tests passed.\n";

    return 0;
}
```

Now, you can call `testVowel()` at any time to re-prove that you haven't broken anything, and the test routine will do all the work for you, returning either an "all good" signal (return value `0`), or the test number that didn't pass, so you can investigate why it broke. This is particularly useful when going back and modifying old code, to ensure you haven't accidentally broken anything!

For advanced readers

A better method is to use `assert`, which will cause the program to abort with an error message if any test fails. We don’t have to create and handle test case numbers this way.

```
#include <cassert> // for assert
#include <cstdlib> // for std::abort
#include <iostream>

bool isLowerVowel(char c)
{
    switch (c)
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true;
    default:
        return false;
    }
}

// Program will halt on any failed test case
int testVowel()
{
#ifdef NDEBUG
    std::cerr << "Tests run with NDEBUG defined (asserts compiled out)";
    std::abort();
#endif
    assert(isLowerVowel('a'));
    assert(isLowerVowel('e'));
    assert(isLowerVowel('i'));
    assert(isLowerVowel('o'));
    assert(isLowerVowel('u'));
    assert(!isLowerVowel('b'));
    assert(!isLowerVowel('q'));
    assert(!isLowerVowel('y'));
    assert(!isLowerVowel('z'));

    return 0;
}

int main()
{
    testVowel();

    // If we reached here, all tests must have passed
    std::cout << "All tests succeeded\n";

    return 0;
}
```

We cover `assert` in lesson 9.6 -- Assert and static_assert.

Unit testing frameworks

Because writing functions to exercise other functions is so common and useful, there are entire frameworks (called **unit testing frameworks**) that are designed to help simplify the process of writing, maintaining, and executing unit tests. Since these involve third party software, we won't cover them here, but you should be aware they exist.

Integration testing

Once each of your units has been tested in isolation, they can be integrated into your program and retested to make sure they were integrated properly. This is called an **integration test**. Integration testing tends to be more complicated -- for now, running your program a few times and spot checking the behavior of the integrated unit will suffice.

Quiz time

Question #1

When should you start testing your code?

Show Solution