5.11 — Introduction to testing your code

BY ALEX ON SEPTEMBER 8TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 18TH, 2018

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 or switch statements), takes no inputs, and produces the correct answer, then you're done. In this case, you've already tested the entire program by running it and validating the output.

But with C++, more likely you've written a program you intend to run many times, that uses loops and conditional logic, and accepts user input. You've possibly written functions that can be reused in other programs. Maybe you're even intending to distribute this program to other people (who may 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 in all cases.

Software verification (a.k.a. software testing) is the process of determining whether or not the software works as expected in all cases.

The testing challenge

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

Consider this simple program:

```
#include <iostream>
#include <string>

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);
}
```

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 quickly untenable.

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 conclude that if the statement that executes when x > y is true works for one pair of x and y values, 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 it about three times (one for each branch) 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.

How to test your code: Informal testing

Most developers do informal testing as they write their programs. After writing a unit of code (a function, a class, or some other discrete "package" of code), the developer writes some code to test the unit that was just added, and then erases the test once the test passes. For example, for the following isLowerVowel() function, you might write the following code:

```
#include <iostream>

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

int main()
{
    std::cout << isLowerVowel('a'); // temporary test code, should produce 1
    std::cout << isLowerVowel('q'); // 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, so you can erase that temporary test code, and continue programming.

Testing tip #1: Write your program in small, well defined units (functions), and compile often along the way

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 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 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. It's still possible that something happened while connecting all the parts, but that's a lot less fewer things to have to worry about and potentially debug.

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, 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 many less places to look, and far less time spent debugging.

Rule: Compile often, and test any non-trivial functions when you write them

Code coverage

The term **code coverage** is used to describe how much of the source code of a program is executed while testing. There are many different metrics used for code coverage. We'll cover a few of the more useful and popular ones in the following sections.

Testing tip #2: Aim for 100% statement coverage

The term **statement coverage** refers to the percentage of statements in your code that have been exercised by your testing routines.

Consider the following function:

```
int foo(int x, int y)
{
    bool z = y;
    if (x > y)
    {
        z = x;
    }
    return z;
}
```

Calling this function as foo(1, 0) will give you complete statement coverage for this function, as every statement in the function will execute.

For our isLowerVowel() function:

```
bool isLowerVowel(char c)
{
    switch (c) // statement 1
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true; // statement 2
    default:
        return false; // statement 3
    }
}
```

This function will require two calls to test all of the statements, as there is no way to reach statement 2 and 3 in the same function call.

Rule: Ensure your testing hits every statement in the function.

Testing tip 3: Aim for 100% branch coverage

Branch coverage refers to the percentage of branches that have been executed, with the affirmative case and negative case counting separately. An if statement has two branches -- a true case, and false case (even if there is no corresponding statement to execute). A switch can have many branches.

```
int foo(int x, int y)
{
    bool z = y;
    if (x > y)
    {
        z = x;
```

```
    }
    return z;
}
```

The previous call to foo(1, 0) gave us 100% statement coverage and exercised the positive use case, but that only gives us 50% branch coverage. We need one more call, to foo(0, 1), to test the use case where the if statement does not execute.

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

In the isLowerVowel() function, two calls (such as isLowerVowel('a') and isLowerVowel('q')) will be needed to give you 100% branch coverage (multiple cases that feed into the same body don't need to be tested separately -- if one works, they all should).

Revisiting the compare function above:

```
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
}
```

3 calls are needed to get 100% branch coverage here: compare(1,0) tests the positive use case for the first if statement. compare(0, 1) tests the negative use case for the first if statement and the positive use case for the second if statement. compare(0, 0) tests the negative use case for second if statements and executes the else statement. Thus, we can say this function is testable with 3 calls (not 18 quintillion).

Rule: Test each of your branches such that they are true at least once and false at least once.

Testing tip #4: Aim for 100% loop coverage

Loop coverage (informally called "the 0, 1, 2 test") says that if you have a loop in your code, you should ensure it works properly when it iterates 0 times, 1 time, and 2 times. If it works correctly for the 2 iteration case, it should work correctly for all iterations greater than 2. These three tests therefore cover all possibilities (since a loop can't execute a negative number of times).

Consider:

```
#include <iostream>
int spam(int timesToPrint)
{
    for (int count=0; count < timesToPrint; ++count)
        std::cout << "Spam!!!";
}
```

To test the loop within this function properly, you should call it three times: spam(0) to test the zero-iteration case, spam(1) to test the one-iteration case, and spam(2) to test the two-iteration case. If spam(2) works, then spam(n) should work, where n>2.

Rule: Use the 0, 1, 2 test to ensure your loops work correctly with different number of iterations

Testing tip #5: Ensure you're testing different categories of input

When writing functions that accept parameters, or when accepting user input, consider what happens with different categories of input. In this context, we're using the term "category" to mean a set of inputs that have similar characteristics.

For example, if I wrote a function to produce the square root of an integer, what values would it make sense to test it with? You'd probably start with some normal value, like 4. But it would also be a good idea to test with 0, and a negative number.

Here are some basic guidelines for category testing:

For integers, make sure you've considered how your function handles negative values, zero, and positive values. For user input, you should also check for overflow if that's relevant.

For floating point numbers, make sure you've considered how your function handles values that have precision issues (values that are slightly larger or smaller than expected). Good test values are 0.1 and -0.1 (to test numbers that are slightly larger than expected) and 0.6 and -0.6 (to test numbers that are slightly smaller than expected).

For strings, make sure you've considered how your function handles an empty string (just a null terminator), normal valid strings, strings that have whitespace, and strings that are all whitespace. If your function takes a pointer to a char array, don't forget to test nullptr as well (don't worry if this doesn't make sense, we haven't covered it yet).

Rule: Test different categories of input values to make sure your unit handles them properly

How to test your code: Preserving your tests

Although writing tests and erasing them is good enough for quick and temporary testing, for code that you expect to be reusing or modifying in the future, it might 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 it into a test() 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 test()
{
    std::cout << isLowerVowel('a'); // temporary test code, should produce 1
    std::cout << isLowerVowel('q'); // temporary test code, should produce 0
}

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

How to test your code: 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. We can do better by writing a function that contains both the tests AND the expected answers.

```
#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 test()
{
    if (isLowerVowel('a') != true) return 1;
    if (isLowerVowel('q') != false) return 2;

    return 0;
}

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

Now, you can call test() at any time to re-prove that you haven't broken anything, and the test routine will do all the work for you. This is particularly useful when going back and modifying old code, to ensure you haven't accidentally broken anything!

Quiz time

1) When should you start testing your code?
Hide Solution

As soon as you've written a non-trivial function.

2) What is branch coverage?
Hide Solution

Branch coverage is the percentage of branches that have been executed, with the affirmative case and negative case counting separately.

3) How many tests would the following function need to minimally validate that it works?

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

Hide Solution

4 would be optimal. One to test the a/e/i/o/u case. One to test the default case. One to test isLowerVowel('y', true). And one to test isLowerVowel('y', false).