

## 9.2 — Code coverage

 **ALEX**  **DECEMBER 28, 2023**

In the previous lesson [9.1 -- Introduction to testing your code](https://www.learncpp.com/cpp-tutorial/introduction-to-testing-your-code/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-testing-your-code/>), we discussed how to write and preserve simple tests. In this lesson, we'll talk about what kind of tests are useful to write to ensure your code is correct.

---

### 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.

### 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)
{
    int 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.



While aiming for 100% statement coverage is good, it's often not enough to ensure correctness.

## Branch coverage

**Branch coverage** refers to the percentage of branches that have been executed, each possible branch counted separately. An `if` statement has two branches -- a branch that executes when the condition is `true`, and a branch that executes when the condition is `false` (even if there is no corresponding `else` statement to execute). A switch statement can have many branches.

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

The previous call to `foo(1, 0)` gave us 100% statement coverage and exercised the use case where `x > y`, 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 will be needed to give you 100% branch coverage: one (such as `isLowerVowel('a')`) to test the first cases, and another (such as `isLowerVowel('q')`) to test the default case. Multiple cases that feed into the same body don't need to be tested separately -- if one works, they all should.

Now consider the following function:



```
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 the first and second `if statement` and executes the `else statement`. Thus, we can say this function is reliably tested with 3 calls (which is slightly better than 18 quintillion).

### Best practice

Aim for 100% branch coverage of your code.

## 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>

void 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`.

### Best practice

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

## 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. 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 `double` type values to test with 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, an alphanumeric string, strings that have whitespace (leading, trailing, and inner), and strings that are all whitespace.



If your function takes a pointer, don't forget to test `nullptr` as well (don't worry if this doesn't make sense, we haven't covered it yet).

**Best practice**  
Test different categories of input values to make sure your unit handles them properly.

Quiz time

Question #1

What is branch coverage?


[Show Solution](#) (javascript:void(0))


Question #2


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;
        default:
            return false;
    }
}
```

[Show Solution](#) (javascript:void(0))

 **Next lesson**  
9.3 [Common semantic errors in C++](#)

 **[Back to table of contents](#)**

 **Previous lesson**  
9.1 [Introduction to testing your code](#)



Leave a comment...

 Name\*

 Email\*




Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!

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

57 COMMENTS

Newest ▼

We and our partners share information on your use of this website to help improve your experience.

Do not sell my info: ☐

OKAY

