5.11 — Introduction to testing your code

BY ALEX ON SEPTEMBER 8TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```cpp
#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);
}
```

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:

```cpp
#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:

```cpp
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:

```cpp
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:

```cpp
#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.

```cpp
#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?
Show Solution

2) What is branch coverage?
Show Solution

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;
    default:
        return false;
    }
}
```

Show Solution

5.x -- Chapter 5 comprehensive quiz

Index

5.10 -- std::cin, extraction, and dealing with invalid text input

C++ TUTORIAL | PRINT THIS POST

58 comments to 5.11 — Introduction to testing your code

alfonso
August 27, 2019 at 1:05 am · Reply

```cpp
// 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;
}
```

This function does not return the number of tests that may fail
If isLowerVowel ('a') returns false, this test is failed and the function test () will return 1 (1 test failed) without testing isLowerVowel ('q') that may also fail and then the number of failed test should be 2 not 1.

```cpp
// returns the number of the test that failed, or 0 if all tests passed
int test()
{
    int failed {0};
    if (isLowerVowel('a') != true) failed++;
    if (isLowerVowel('q') != false) failed++;

    return failed;
}
```

But this above will return the number of failed tests.

nascardriver
August 27, 2019 at 6:49 am · Reply

It's not supposed to return the number of failed tests, it's supposed to return a number that identifies the test the failed.

alfonso
August 27, 2019 at 11:22 pm · Reply

Then it ignores other tests that may fail. It returns only the first failed test. Or other tests may pass but after modifying the program to pass the test in attention, you can damage other tests and you do not know that. Oh well, those are just examples of program, I suppose the programmer will make the needed test implementation.

Benur21
August 5, 2019 at 1:23 pm · Reply

For strings we should also test different types of characters by copy-pasting them from another place, such as non-ascii, €$&£Çñáè☐☻ƒ±Ä, etc.

Nirbhay
July 24, 2019 at 5:05 am · Reply

Hi!

I do not understand the "writing a function that contains both the tests AND the expected answers." logic in the code. i.e. in the int test() function. Can you please elaborate?
Also the comment -- // returns the number of the test that failed, or 0 if all tests passed   is what I don't understand.

Who will help me? The Author Alex himself or the legendary driver nascardriver?

Thank you

nascardriver
July 24, 2019 at 5:25 am · Reply

Hi!

If all tests pass, the function reaches line 24 and returns 0. If a test fails, the function returns the number of the test. This allows the caller to easily check for success and in the case of a failure, quickly figure out which test failed.

```cpp
// Variables can be created in if-statements.
// I don't think Alex covered this.
if (int iTest{ test() }; iTest != 0)
{
  std::cout << "test " << iTest << " failed\n";
}
else
{
  std::cout << "all tests passed\n";
}
```

Nirbhay
July 24, 2019 at 5:33 am · Reply

Hello!

But under what circumstances can a test possibly fail in the example above?

e.g. if (isLowerVowel('a') != true) return 1;
  if (isLowerVowel('q') != false) return 2;

above statements will always be false.. so always 0 will be returned.

I think I need examples and having a hard time imagining them here. Could you provide some?

Thanks :)

nascardriver
July 24, 2019 at 5:36 am · Reply

When you write a test, you know the outcome, but you don't know the implementation. The purpose of a test is to figure out if the implementation matches the expectation.
'a' is a vowel and lower case, so `isLowerVowel` should return true.
'q' is a not vowel, so `isLowerVowel` should return false.
In the case that the implementation is wrong, because maybe you were tired and forgot that 'u' is a vowel, the test will detect it.

Nirbhay
July 24, 2019 at 5:49 am · Reply

Alright!

So let me get this straight.

'a','e','i','o','u' are vowels. So the ideal implementation in the switch case would be as above(how alex has written it).

But let's say there was no 'u' case (which is wrong) then

if (isLowerVowel ('u') != true) return 1;

would evaluate to true and return 1 thereby giving us the number of this test which failed. (test failing meaning the error in implementation here).

Can we not write it as

if (!(isLowerVowel('a'))) return 1;
if (isLowerVowel('q')) return 2;

Thanks

nascardriver
July 24, 2019 at 5:51 am · Reply

yes you can, it's the same

Nirbhay
July 24, 2019 at 5:58 am · Reply

Thanks. It really made things clear :)

Nyther

May 10, 2019 at 9:08 am · Reply

How to test your program:
Give it to a try hard gamer, I am sure he will manage to break, exploit or crash it in some way

Red Lightning
April 23, 2019 at 6:05 am · Reply

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

For the comment readers, here is a bonus: spam comes from Python, eggs are from there too. You use them when you test trivial functions and variables (to save the time of thinking of names).
In C++, foo and bar are used.

nascardriver
April 12, 2019 at 6:26 am · Reply

Hello Alex,

Quiz 3a line 12

```
return yIsVowel;
```

Senna
April 12, 2019 at 6:05 am · Reply

In the first code of this lesson, why do you #include <string> in it as it's NOT used ?
Is it your mistake ?

```
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);
}
```

Alex
April 14, 2019 at 7:51 am · Reply

Yeah, my mistake. I probably was using std::string originally and optimized it out of the example, but forgot to remove the include. Fixed!

Alireza
April 14, 2019 at 12:11 pm · Reply

You're great sir.
I'm just curious how you optimized it out with including <string> ?
If it's a secret you don't need to say it .
Thanks for this teriffic tutorial ;)

Alex
April 16, 2019 at 9:52 pm · Reply

Unsure. The examples often go through several iterations as I look for ways to make things simpler and more concise. It's possible one of the earlier iterations used std::string and the later iterations didn't. But I don't remember since it was a long time ago.

Glad you're enjoying the site!

Alireza
April 24, 2019 at 3:58 am · Reply

Good luck everywhere, every when

Alireza
April 12, 2019 at 5:59 am · Reply

Greeting,

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

Is this validating more than it'd need ?

```
#include <iostream>

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;
    }
}

int main()
{
    std::cout << std::boolalpha;

    std::cout << isLowerVowel('a',false) << "\n";
    std::cout << isLowerVowel('a',true) << "\n";
    std::cout << isLowerVowel('a',0) << "\n";
    std::cout << isLowerVowel('a',1) << "\n";

    std::cout << isLowerVowel('q',false) << "\n";
    std::cout << isLowerVowel('q',true) << "\n";
    std::cout << isLowerVowel('q',0) << "\n";
    std::cout << isLowerVowel('q',1) << "\n";

    std::cout << isLowerVowel('y',false) << "\n";
    std::cout << isLowerVowel('y',true) << "\n";
    std::cout << isLowerVowel('y',0) << "\n";
    std::cout << isLowerVowel('y',1) << "\n";


    return 0;
}
```

nascardriver
April 12, 2019 at 6:25 am · Reply

Booleans are false/true, not 0/1, so remove half of your tests.
Then you just have one test for 'a' more than Alex, that's fine. Tests only run once, so it doesn't matter if you have too many. It's only bad if you have too few.

hassan magaji
January 13, 2019 at 9:00 am · Reply

Hi Alex,
In the section "The testing challenge" under the code snippet,
I think 4-bytes integer is ~4 billion while 8-bytes int is ~18 quintillion right?
sorry if i've been such a dummy i'll go and revise integer sizes.

YTXbaiaLrs
March 25, 2019 at 7:57 pm · Reply

One 4-byte integer can hold 2^32 distinct values. The compare function takes in two 4-byte integers. The first and second integer can take any of the 2^32 distinct values, independently (they do not need to both be the same value). So there are 2^32 * 2^32 = 2^64 ~= 18 quintillion possibilities.

hassan magaji
March 27, 2019 at 9:44 am · Reply

nice and clear, future readers seems to be sharper than we the past readers.

Thanks anyway, hope you'll enjoy your journey through the rest of the tutorial series.

I'm in love with alex's tutor
August 29, 2018 at 2:33 am · Reply

Dear Mr Alex!

in the last test example found under "Automating your test functions", for int test function that was defined us below
could we use if statement with out else? Please see the comment from where i point my questions out.

```
1  int test()
2  {
3      if (isLowerVowel('a') != true) return 1;
4      if (isLowerVowel('q') != false) return 2;
5  
6      return 0;//why we left else at the above line?
7  }
```

And secondly, can i correct the above function as below ? which one is preferable, since we can use true or false directly to if statement?

```
int test()
{
    if (isLowerVowel('a') = false)
return 1;
    if (isLowerVowel('q') =true)
 return 2;
    else
    return 0;
}
```

Stay blessed sir!!!!!!!!!!

nascardriver
August 29, 2018 at 7:13 am · Reply

Hi!

1] When a return statement is reached, the function stops. Line 6 can only be reached if none of the if-statement's conditions were fulfilled.

2] Yours doesn't work, because you're using the assignment operator rather than a comparison operator. That aside, both are bad, you shouldn't compare booleans to false/true, because they are booleans already.

```
if (isLowerVowel('a')) return 1;
if (!isLowerVowel('q')) return 2;

return 0;
```

I'm in love with alex's tutor
August 30, 2018 at 12:15 am · Reply

Dear great Teacher Nas!
Thank you so much! Every thing is clear now. sorry for forgetting to use "==" . That Is

why i love programming. It needs our focus and concentration.I failed. looool
God bless you Sir!

Solomzi
August 23, 2018 at 11:57 pm · Reply

how to declare a seed?

Alex
August 25, 2018 at 8:42 am · Reply

What's the context for this comment? I don't use the word "seed" in the lesson so I'm not sure what you're referring to.

Piyush
November 6, 2018 at 5:28 am · Reply

Are you talking about the seed to generate random numbers.

Andrew
January 18, 2018 at 1:02 pm · Reply

Words order.
" there are few worth talking about in this informal context "
Should be
" there are few worth talking about this in informal context "

Alex
January 18, 2018 at 9:25 pm · Reply

I meant to say, "there are _a_ few worth talking about in this informal context" (was missing the 'a'), but I've rewritten the paragraph a bit. Thanks for pointing out the omission.

□□
September 6, 2017 at 9:27 am · Reply

Hi, Alex, thanks for the tutorial. I am not really understand how the code works in the below, i know it's recursion. May you comment the below code so I can understand better? thanks so much
!

```
void MergeSort(int *a, int low, int high)
{
  int mid;
  if (low < high)
  {
    mid = (low+high) / 2;
    MergeSort(a, low, mid);
    MergeSort(a, mid+1,high);
    Merge(a,low, high, mid);
  }
}
```

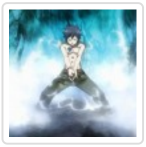
I know when it executes the if statement it will cause a recursion of MergeSort to call itself again, so doesnt it re-initlize the mid again? and when mid become zero, it will executes line8 MergeSort(a, mid+1,high) right? and mid+1 would be '1' right? thanks so much for the help!!

Alex
September 6, 2017 at 10:54 am · Reply

From an algorithm standpoint:
* This a top-down recursive sorting algorithm
* It works by splitting an array of items in half, sorting each half (via the recusive calls to MergeSort), and then recombining the sorted halves (via the call to Merge).
* Each half is continually split in half (recursively) until the arrays each only have one item (low >= high), at which point sorting is trivial.

From a code standpoint:
* Each function call has its own version of mid that gets assigned to the midpoint of high and low.
* Yes, if mid is 0, then MergeSort(a, 1, high) would be called.

If you haven't already, go read lesson 7.11. I talk more about recursion there.



blaze077
May 15, 2017 at 2:33 pm · Reply

Hi, there's a typo - you misspelled "thought": "who may try things you haven’t though of"
Thanks for the tutorials as always.

Jason
February 13, 2017 at 5:18 pm · Reply

Hi Alex,

Let's say that I have a program that has the user enter two 16-bit integers, multiplies them together, then displays the result. What if the user enters two satisfactory integers, but the product doesn't fit in the 16 bits? Would there be a way to correct for that? (Other than trying to increase the memory size of the product variable)

Alex
February 14, 2017 at 1:49 pm · Reply

In C++, you're better off determining whether integer overflow would result from the multiplication before you actually do the multiplication, as opposed to doing the multiplication and then trying to determine if it did happen. See **this thread** for some thoughts on how to do that.

Ola
December 28, 2016 at 8:21 am · Reply

Hey Alex,

The test function in the last example will not return 2 if both test cases fail. I would like to confirm that because of the return statement. Merry Christmas too. Thanks for the great work.

Alex

December 29, 2016 at 1:32 pm · Reply

Correct, only the first failure will be reported. Once that failure is corrected (either by fixing the function being tested, or fixing the test case), then any subsequent failures will be returned.

Deepanshu
November 17, 2016 at 2:14 am · Reply

Hi Alex, you told that we should start testing our code when we write a non-trivial function. So, I want to ask what is a non-trivial function?

Alex
November 17, 2016 at 2:01 pm · Reply

I would consider most functions that contain branching code or loops non-trivial.

Matt
November 5, 2016 at 4:44 am · Reply

In section "Testing tip #5: ", you wrote:
"For floating point numbers, make sure you’ve considered how your function handles precision issues. 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)."

I feel like this paragraph needs more explanation at the end. I understand about the precision issues that floating point numbers have, but I can't figure out how to apply the test values that you suggest.

Alex
November 5, 2016 at 1:50 pm · Reply

It's analogous to the statement on integers, where I say make sure your function handles negative, positive, and zero values. So if you had some function foo(int), you'd test with foo(2), foo(0), and foo(-2) for example. Pretty straightforward, right?

For floating point numbers, those 4 specific values (-0.6, -0.1, 0.1, and 0.6) exercise both positive and negative use cases, and cases where the numbers are represented as numbers slightly better and slightly smaller than expected. So if you had some function goo(double), you'd test with goo(-0.6), goo(-0.1), goo(0.1), and goo(0.6).

Matt
November 5, 2016 at 4:31 am · Reply

In section "Testing tip 3: Aim for 100% branch coverage", you wrote:
" (multiple cases that feed into the same body don’t need to be tested separately, if one works, they all should)".

I think that first comma needs to be a period, semicolon, or dashes maybe.

Matt
November 4, 2016 at 3:51 pm · Reply

In section "Testing tip #1: Write your program in small, well defined units (functions), and compile often along the way".... (well, come to think of it, I think you should shorten that heading)... in the middle paragraph you wrote:
" A symptom might have many causes -- for example, is the car not starting due to a faulty spark plug, battery, fuel pump, or somewhere else?"

I think you might have meant to write "something" in place of "somewhere".

Also in the same paragraph, you veer away from the car-building analogy towards the end, and start referring to programming again. I just had a thought that it might sound better if you stuck with the analogy in that paragraph.

Alex
November 5, 2016 at 1:28 pm · Reply

Updated as you suggest. Thanks!

Matt
November 4, 2016 at 3:22 pm · Reply

In the forth paragraph, the last sentence seems like it should be split into two sentences... or maybe a semicolon should be used in place of the last comma.

Also, under "Testing challenge", second to last paragraph... I think the last sentence could maybe use a comma or two.

Also, in the same section, last paragraph, second to last sentence... I think it's missing a word or two. You wrote:
" But since it's not a C++ specific topic, we'll stick to a brief and informal introduction, covered from the point of view you (as the developer) testing your own code."

Alex
November 5, 2016 at 1:21 pm · Reply

Good comments. I've updated the lesson text accordingly. Many thanks!

reznov
October 6, 2016 at 3:09 pm · Reply

Would the use of a goto statement be legitimate to avoid rerunning your program 30 times when testing a big case structure?
I have something like this in mind:

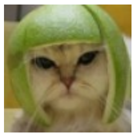
:test

//ask input
//large switch statement
//output case return

goto(test);

Until now I have done this writing a temporary while loop, but just figured this might be a better way. This doesn't change the readability of my code and the scope of my variables and it's way easier to remove the entire thing than the while statement.

Alex

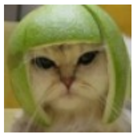


October 6, 2016 at 4:55 pm · Reply

Sure. But even better would be to put the switch statement in a reusable function, and then write a separate test function that tests each switch case. That way you can just remove (or comment out) the call to the test function when you want to disable your tests.

Akhil
September 30, 2016 at 1:03 am · Reply

```
bool isLowerVowel(char c)
{
    switch (c) // statement 1
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true; // statement 2
        //shouldn't there be a break statement here?
    default:
        return false; // statement 3
    }
}
```



Alex
September 30, 2016 at 8:19 pm · Reply

No break needed. Return exits the function that the switch is contained in immediately, so even if you had a break, it wouldn't execute.

Sharjeel Safdar
September 10, 2016 at 12:31 pm · Reply

Hi, Alex!
Thanks for the great tutorials.

Here are a few typos in this lesson (just to make these great tutorials greater):
1_ In the code right under the "The testing challenge" heading, there is a line.

```
std::cout << x << " is equal to than " << y << '\n'; // case 3
```

It should be

```
std::cout << x << " is equal to " << y << '\n'; // case 3
```

This mistake is repeated again where the same code is reffered again under the testing tip #3.

2_ Just after the aforementioned example, you have used the word "quintrillion" (in two different paragraphs) instead of "quintillion". And again the same mistake just above "Testing tip #4: Aim for 100% loop coverage" heading.

3_ Just under the heading "How to test your code: Informal testing",
"...writes some code to test the code the unit that was just added..."
This is a bit confusing. Here you might have intended to write something like
"...writes some code to test the unit that was just added..."

4_ Just above "Testing tip #4: Aim for 100% loop coverage",
"3 calls are needed to get to get 100% branch coverage here..."
"to get" is repeated.

Alex
September 12, 2016 at 9:04 am · Reply

All fixed. Thanks for pointing these out!

Ignacio
September 9, 2016 at 11:14 am · Reply

Typo

In the following sentence you repeat the word each

Build (or buy) and test each each car component individually before installing it

Thanks a lot for this amazing tutorial!!!

Alex
September 9, 2016 at 2:12 pm · Reply

Thanks for pointing out the typo. Fixed.

Halim
September 9, 2016 at 1:05 am · Reply

Hi Alex, in lesson 4.4a(casting) you said that you will discuss about dynamic_cast in pointer, but I didn't find any lesson that discuss about dynamic_cast, are you forget about it? If so please discuss about dynamic_cast later. Thank you

Alex
September 9, 2016 at 8:12 am · Reply

We discuss dynamic_cast when we talk about inheritance. It's not useful to discuss before that point. I've updated lesson 4.4a to note that inheritance is a prerequisite to this conversation as well.