

Test Coverage on Unit Testing in PlatformIO / VSCode

So you are building a project which is beyond the trivial example projects, and after some time you realize you need serious bottom-up testing in order to get it working as well as to be able to maintain it and still sleep well.

In that case I assume you are already convinced that unit-testing is a really solid approach: you are forced to think about testing upfront and you will try to minimize dependencies so each module can be tested, isolated on its own.

Unit testing is well supported in PlatformIO and I only refer to the documentation [Unit Testing — PlatformIO latest documentation](#) and to some getting started blog-posts [Unit Testing with PlatformIO: Part 1. The Basics | PlatformIO Labs \(piolabs.com\)](#).

Note : If you have never done unit-testing, get this working first, then come back for code coverage :-)

The next step however, is to make sure **all** of your code is being tested, and this is where so-called **code coverage** tools come into play. In short, we are going to run all our unit tests, and afterwards the tools are going to tell us which pieces of the code were tested and which pieces were **missed** by the tests.

Good news is that the GCC already has most of what we need, the only thing missing is a nice visualization of the results. Luckily, for this there is a good VSCode extension available.

Before you get too excited: the how-to below is only taking care of the ‘native’ unit tests, ie. the ones that do not need to run on the target hardware. It is possible to extend towards target unit tests, but this requires extra steps and we will cover them in a future article.

Demo time

For the demo I am going to take a small and simple piece of code, so you won’t waste your time trying to understand the code.

This is <project>/lib/key/key.h

```
1 // #####
2 // ### Author(s) : Pascal Roobrouck - @strooom ###
3 // ### License : https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode ###
4 // #####
5
6 #pragma once
7 #include <stdint.h>
8
9 class key {
10 public:
11     static uint8_t toUpperCase(uint8_t aCharacter);           // convert a character to upper case
12     static bool isHexCharacter(uint8_t aCharacter);           // check if a character is a valid hex character, ie. 0-9 or A-F - assum
13     static uint8_t valueFromHexCharacter(uint8_t aCharacter); // convert a hex character to a value, ie. 'A' -> 10
14     static uint8_t hexCharacterFromValue(uint8_t aValue);     // convert a value to a hex character, ie. 10 -> 'A'
15 private:
16 };
17
18
```

This is <project>/lib/key/kep.cpp

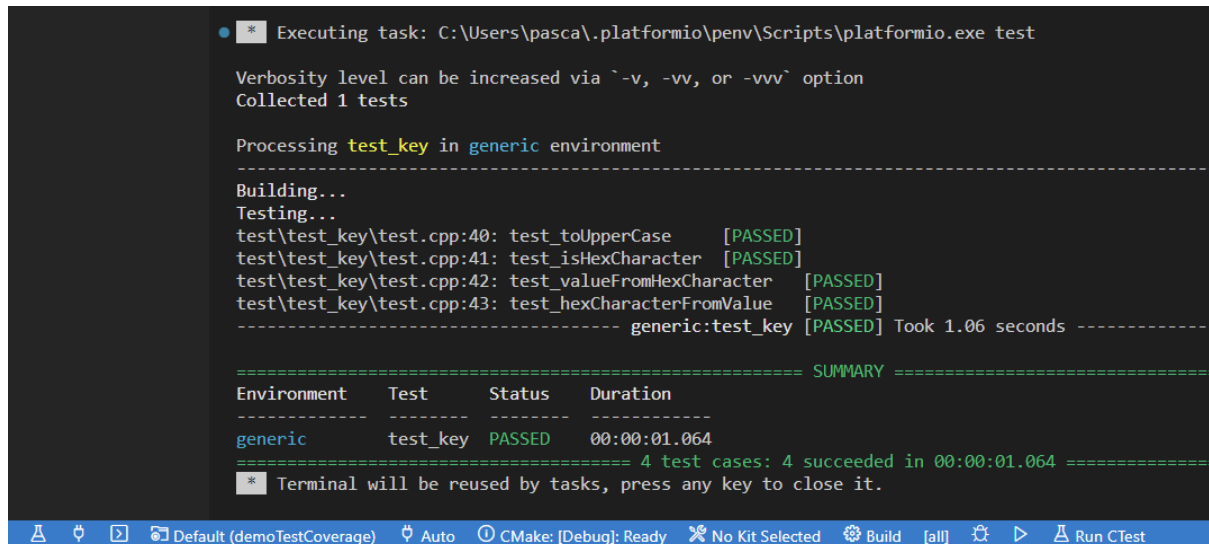
```
1
2 #include "key.h"
3
4 uint8_t key::toUpperCase(uint8_t aCharacter) {
5     if (aCharacter >= 'a' && aCharacter <= 'z') {
6         aCharacter = aCharacter - 'a' + 'A';
7     }
8     return aCharacter;
9 }
10
11 bool key::isHexCharacter(uint8_t aCharacter) {
12     return ((aCharacter >= 'A' && aCharacter <= 'F') || (aCharacter >= '0' && aCharacter <= '9'));
13 }
14
15 uint8_t key::valueFromHexCharacter(uint8_t aCharacter) {
16     if (aCharacter >= '0' && aCharacter <= '9') {
17         return aCharacter - '0';
18     }
19     if (aCharacter >= 'A' && aCharacter <= 'F') {
20         return aCharacter - 'A' + 10;
21     }
22     return 0; // in case the character is not a hex character, return 0 as value
23 }
24
25 uint8_t key::hexCharacterFromValue(uint8_t aValue) {
26     if (aValue <= 9) {
27         return aValue + '0';
28     } else if (aValue <= 15) {
29         return aValue - 10 + 'A';
30     } else
31         return '?';
32 }
33
```

In order to test these 4 member functions, I have a test-application, using Unity as unit-testing framework, so here is <project>/test/test_key/test.cpp

```
1 #include <unity.h>
2 #include "key.h"
3
4 void setUp(void) {} // before test
5 void tearDown(void) {} // after test
6
7 void test_toUpperCase() {
8     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('A'), key::toUpperCase(static_cast<uint8_t>('a')));
9     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('A'), key::toUpperCase(static_cast<uint8_t>('A')));
10    TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('0'), key::toUpperCase(static_cast<uint8_t>('0')));
11 }
12
13 void test_isHexCharacter() {
14     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('0')));
15     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('9')));
16     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('A')));
17     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('F')));
18     TEST_ASSERT_FALSE(key::isHexCharacter(static_cast<uint8_t>('-')));
19     TEST_ASSERT_FALSE(key::isHexCharacter(static_cast<uint8_t>('0')));
20     TEST_ASSERT_FALSE(key::isHexCharacter(static_cast<uint8_t>('a'))); // we assume uppercase, so lowercase should fail...
21 }
22
23 void test_valueFromHexCharacter() {
24     TEST_ASSERT_EQUAL_UINT8(0U, key::valueFromHexCharacter(static_cast<uint8_t>('0')));
25     TEST_ASSERT_EQUAL_UINT8(9U, key::valueFromHexCharacter(static_cast<uint8_t>('9')));
26     TEST_ASSERT_EQUAL_UINT8(10U, key::valueFromHexCharacter(static_cast<uint8_t>('A')));
27     TEST_ASSERT_EQUAL_UINT8(15U, key::valueFromHexCharacter(static_cast<uint8_t>('F')));
28 }
29
30 void test_hexCharacterFromValue() {
31     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('0'), key::hexCharacterFromValue(0U));
32     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('9'), key::hexCharacterFromValue(9U));
33     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('A'), key::hexCharacterFromValue(10U));
34     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('F'), key::hexCharacterFromValue(15U));
35     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('?'), key::hexCharacterFromValue(16U));
36 }
37
38 int main(int argc, char **argv) {
39     UNITY_BEGIN();
40     RUN_TEST(test_toUpperCase); test\test_key\tests.cpp:40:test_toUpperCase:PASS
41     RUN_TEST(test_isHexCharacter); test\test_key\tests.cpp:41:test_isHexCharacter:PASS
42     RUN_TEST(test_valueFromHexCharacter); test\test_key\tests.cpp:42:test_valueFromHexCharacter:PASS
43     RUN_TEST(test_hexCharacterFromValue); test\test_key\tests.cpp:43:test_hexCharacterFromValue:PASS
44     UNITY_END();
45 }
```

Note : take a look at the paths where are the files are, because PlatformIO makes some assumptions about them, eg. each test should be in a subdirectory with a name starting with 'test_'

So now we can run the tests. There's two ways to do this : from the PlatformIO test icon in the bottom status bar...



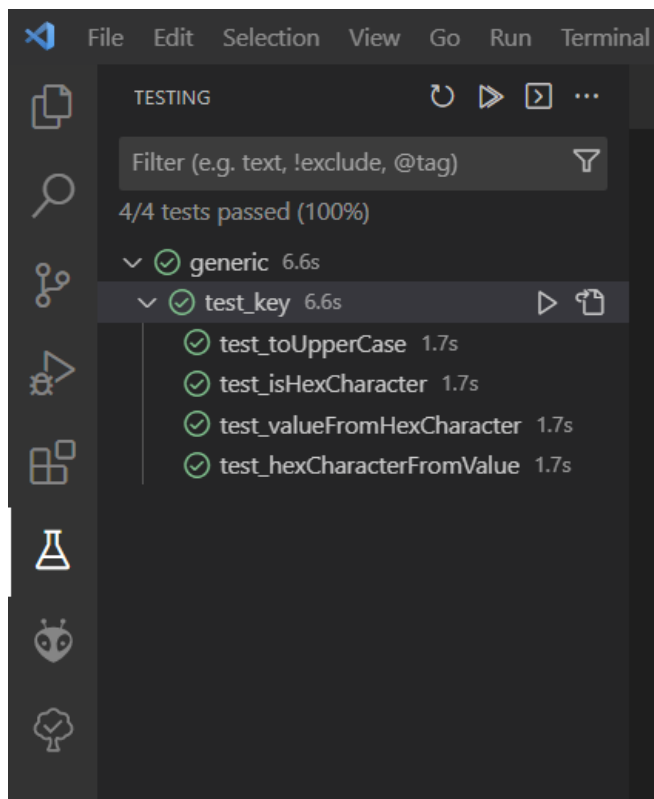
```
• [*] Executing task: C:\Users\pasca\.platformio\penv\Scripts\platformio.exe test

Verbosity level can be increased via `-v, -vv, or -vvv` option
Collected 1 tests

Processing test_key in generic environment
-----
Building...
Testing...
test\test_key\test.cpp:40: test_toUpperCase [PASSED]
test\test_key\test.cpp:41: test_isHexCharacter [PASSED]
test\test_key\test.cpp:42: test_valueFromHexCharacter [PASSED]
test\test_key\test.cpp:43: test_hexCharacterFromValue [PASSED]
----- generic:test_key [PASSED] Took 1.06 seconds -----

===== SUMMARY =====
Environment  Test      Status  Duration
-----
generic      test_key  PASSED  00:00:01.064
===== 4 test cases: 4 succeeded in 00:00:01.064 =====
[*] Terminal will be reused by tasks, press any key to close it.
```

or from VSCode itself,...



So now our tests are running, and we can see which ones pass and which ones fail. Let's now look deeper to see what piece of the code they are really testing. For that we need a few extra settings that tell the compiler (GCC) to generate some extras, so that during the running of our code, it is tracked which statements are executed.

Add this to your platformio.ini :

```
15
16  [env:generic]
17  platform = native
18  lib_ldf_mode = deep
19  build_flags =
20      -D unitTesting
21      -lgcov
22      --coverage
23      -fprofile-abs-path
24
```

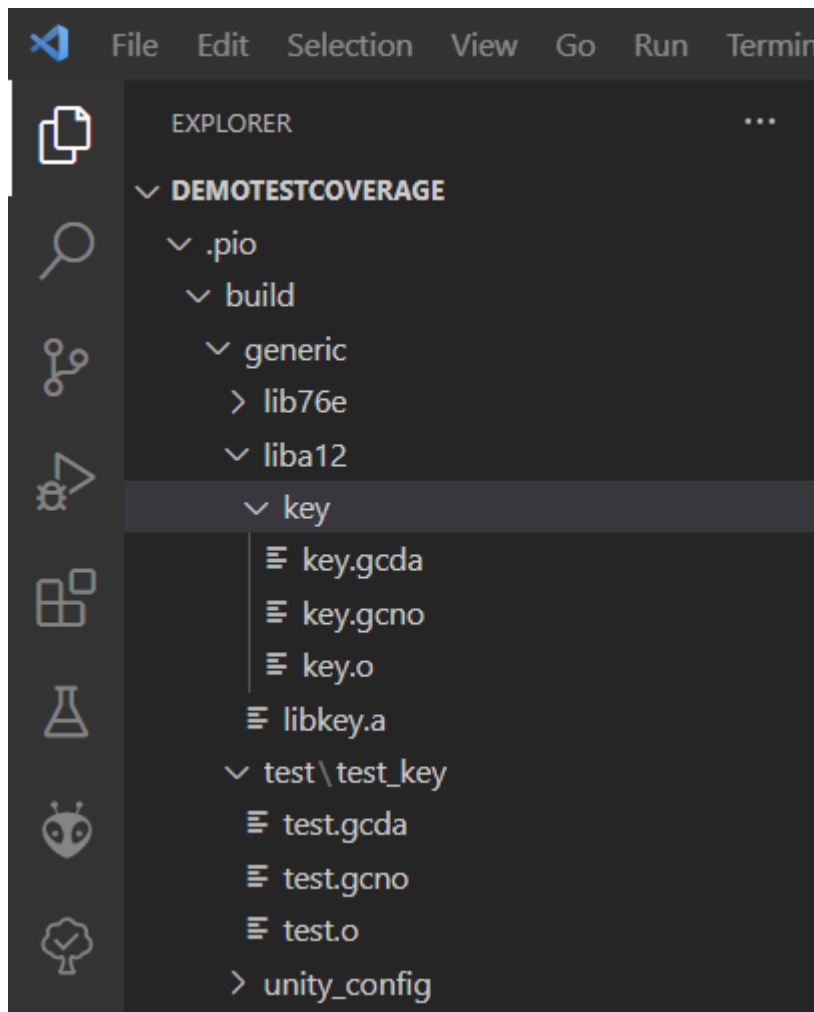
Note the 3 extra build flags:

-lgcov

--coverage

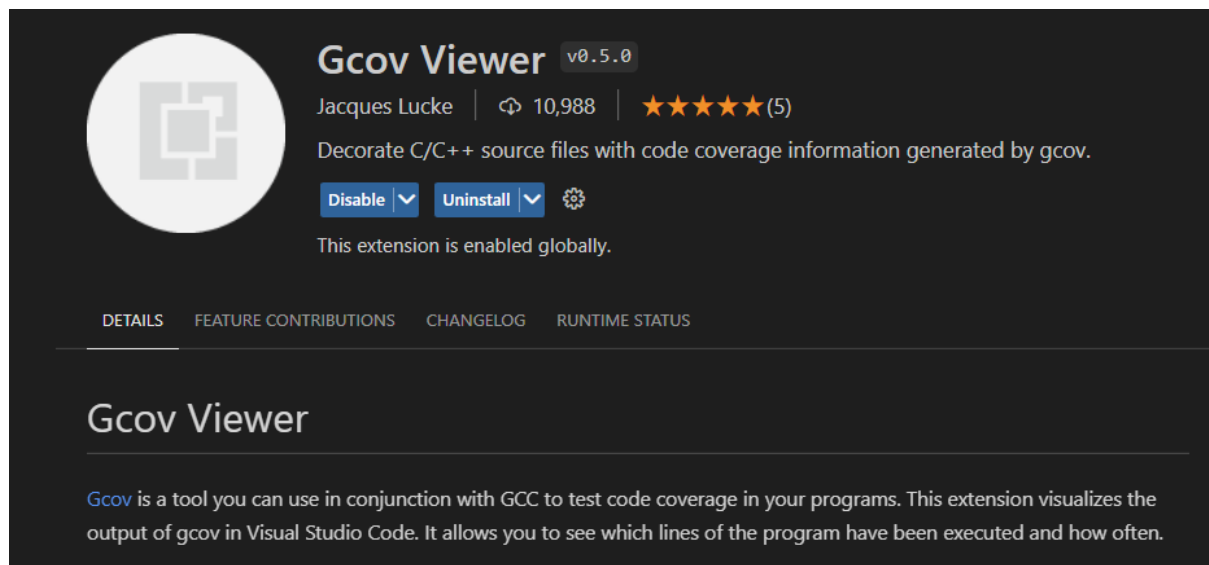
-fprofile-abs-path

With these build-flags, the compiler creates extra files to track code execution, and you find them in <project>/.*pio*/build/<env name>/...



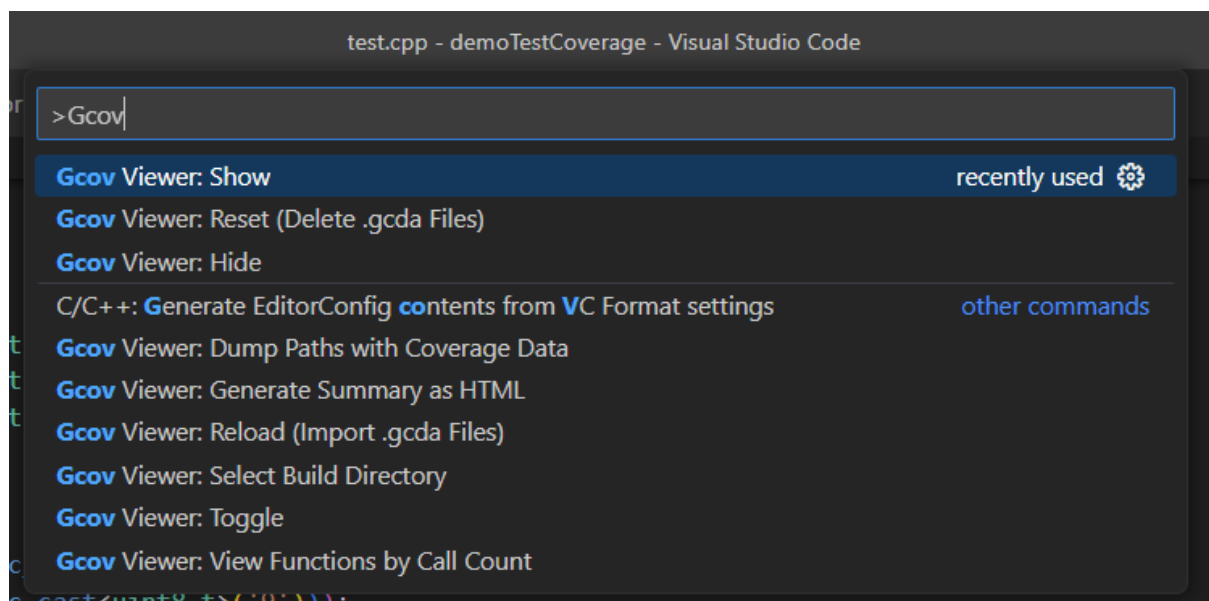
notice the **.gcda** and **.gcno** files for each library, and one for the test application.

Now we need to visualize this. For this we will install an extension **GCOV Viewer**



This extension is controlled via the command line, which is a minor burden but you'll get used to it quickly:

Ctrl-Shift P and type Gcov, this will show you all the Gcov commands. The 3 most important are Show, Hide and Reset and they will settle themselves on top with your 'recently used' commands.



So select 'Gcov Show' and it will create an overlay on your source code files, showing which lines have been executed and which not.

Note : if you want the non-executed lines to show in red, you need to activate a setting of the Gcov extension 'Highlight Missed Lines'. I prefer this, as this way it is more obvious which lines are missed. Several lines of your source code do not translate to executable code, and those lines just remain uncolored.

```

1
2  #include "key.h"
3
4  uint8_t key::toUpperCase(uint8_t aCharacter) { 3x [100.0%]
5      if (aCharacter >= 'a' && aCharacter <= 'z') { 3x
6          aCharacter = aCharacter - 'a' + 'A'; 1x
7      }
8      return aCharacter; 3x
9  }
10
11 bool key::isHexCharacter(uint8_t aCharacter) { 7x [100.0%]
12     return ((aCharacter >= 'A' && aCharacter <= 'F') || (aCharacter >= '0' && aCharacter <= '9')); 7x
13 }
14
15 uint8_t key::valueFromHexCharacter(uint8_t aCharacter) { 4x [83.3%]
16     if (aCharacter >= '0' && aCharacter <= '9') { 4x
17         return aCharacter - '0'; 2x
18     }
19     if (aCharacter >= 'A' && aCharacter <= 'F') { 2x
20         return aCharacter - 'A' + 10; 2x
21     }
22     return 0; // in case the character is not a hex character, return 0 as value
23 }
24
25 uint8_t key::hexCharacterFromValue(uint8_t aValue) { 5x [100.0%]
26     if (aValue <= 9) { 5x
27         return aValue + '0'; 2x
28     } else if (aValue <= 15) { 3x
29         return aValue - 10 + 'A'; 2x
30     } else
31         return '?';
32 }
33

```

Note how each line has a color, and a number of how many times it has been executed. The edge case where `valueFromHexCharacter` receives an invalid input, is missed by our test, so we should fix that.

We saw that GCC creates a coverage file for each `.cpp` file in our application. This means that also our test application itself is tracked.

```

test > test_key > test.cpp > test_isHexCharacter()
4 void setUp(void) {} // before test 3x [100.0%]
5 void tearDown(void) {} // after test 3x [100.0%]
6
7 void test_toUpperCase() { 1x [100.0%]
8     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('A'), key::toUpperCase(static_cast<uint8_t>('a'))); 1x
9     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('A'), key::toUpperCase(static_cast<uint8_t>('A'))); 1x
10    TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('0'), key::toUpperCase(static_cast<uint8_t>('0'))); 1x
11 } 1x
12
13 void test_isHexCharacter() { 1x [100.0%]
14     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('0'))); 1x
15     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('9'))); 1x
16     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('A'))); 1x
17     TEST_ASSERT_TRUE(key::isHexCharacter(static_cast<uint8_t>('F'))); 1x
18     TEST_ASSERT_FALSE(key::isHexCharacter(static_cast<uint8_t>('-'))); 1x
19     TEST_ASSERT_FALSE(key::isHexCharacter(static_cast<uint8_t>('0'))); 1x
20     TEST_ASSERT_FALSE(key::isHexCharacter(static_cast<uint8_t>('a'))); // we assume uppercase, so lowerc
21 } 1x
22
23 void test_valueFromHexCharacter() { 1x [100.0%]
24     TEST_ASSERT_EQUAL_UINT8(0U, key::valueFromHexCharacter(static_cast<uint8_t>('0'))); 1x
25     TEST_ASSERT_EQUAL_UINT8(9U, key::valueFromHexCharacter(static_cast<uint8_t>('9'))); 1x
26     TEST_ASSERT_EQUAL_UINT8(10U, key::valueFromHexCharacter(static_cast<uint8_t>('A'))); 1x
27     TEST_ASSERT_EQUAL_UINT8(15U, key::valueFromHexCharacter(static_cast<uint8_t>('F'))); 1x
28 } 1x
29
30 void test_hexCharacterFromValue() {
31     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('0'), key::hexCharacterFromValue(0U));
32     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('9'), key::hexCharacterFromValue(9U));
33     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('A'), key::hexCharacterFromValue(10U));
34     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('F'), key::hexCharacterFromValue(15U));
35     TEST_ASSERT_EQUAL_UINT8(static_cast<uint8_t>('?'), key::hexCharacterFromValue(16U));
36 }
37
38 int main(int argc, char **argv) { 1x [100.0%]
39     UNITY_BEGIN(); 1x
40     RUN_TEST(test_toUpperCase); test\test_key\test.cpp:40:test_toUpperCase:PASS 1x
41     RUN_TEST(test_isHexCharacter); test\test_key\test.cpp:41:test_isHexCharacter:PASS 1x
42     RUN_TEST(test_valueFromHexCharacter); test\test_key\test.cpp:42:test_valueFromHexCharacter:PASS 1x
43     //RUN_TEST(test_hexCharacterFromValue);
44     UNITY_END(); 1x
45 }

```

Look how one of the tests is not being run, because I commented it out in the test-runner main().

When you improve your tests and rerun them, the results keep being **added** to the .gcno and .gcda files already there. At some time you will want to reset your results and start over. This can be done with the command **Gcov Reset**