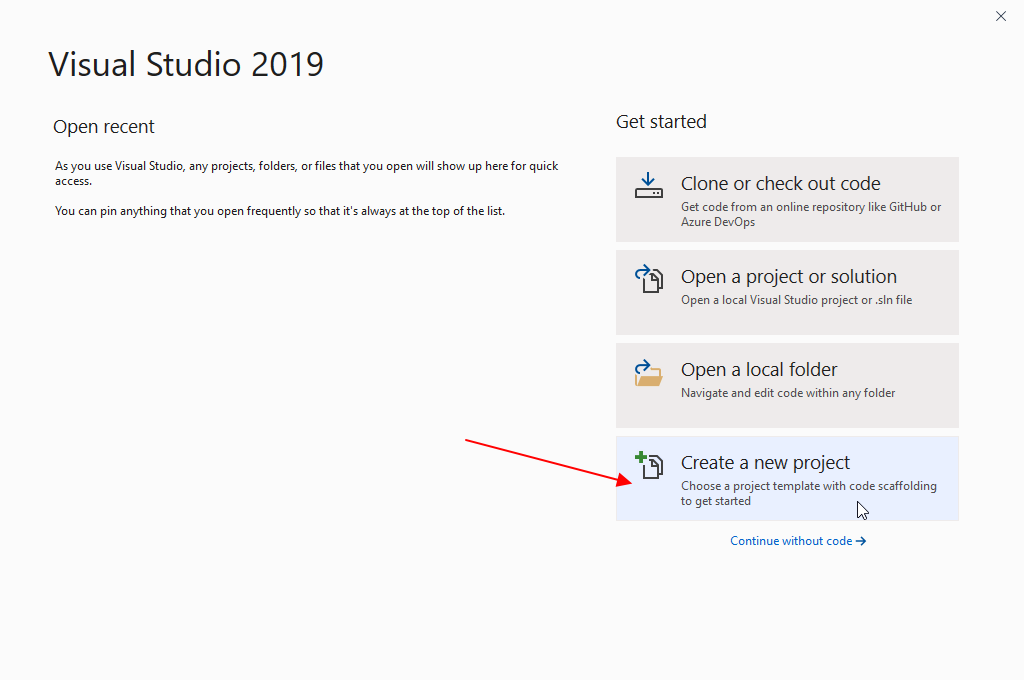
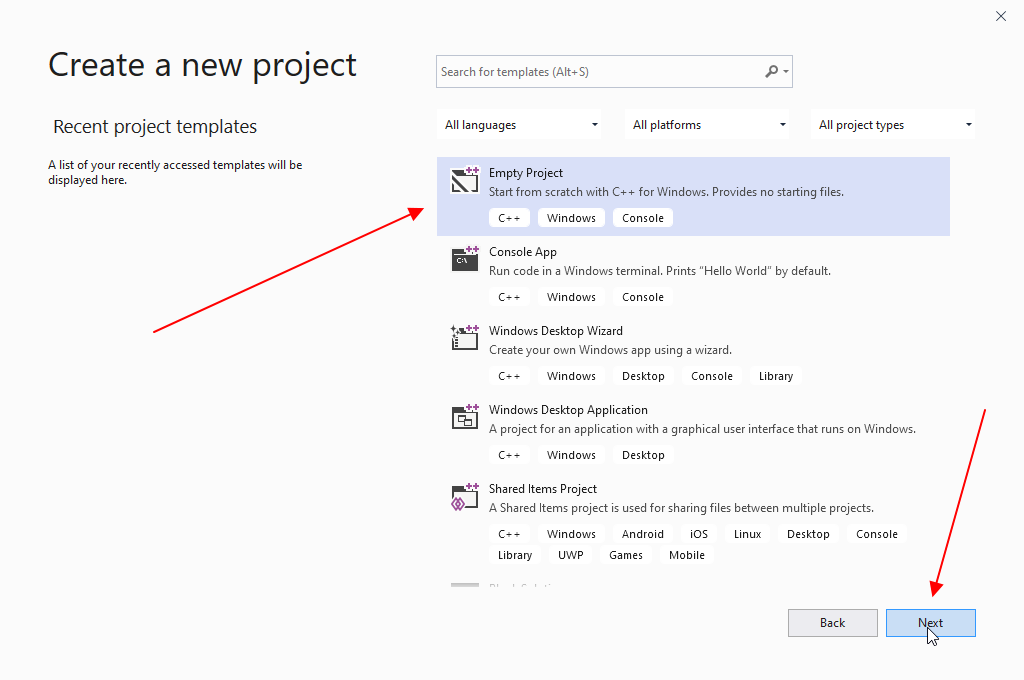
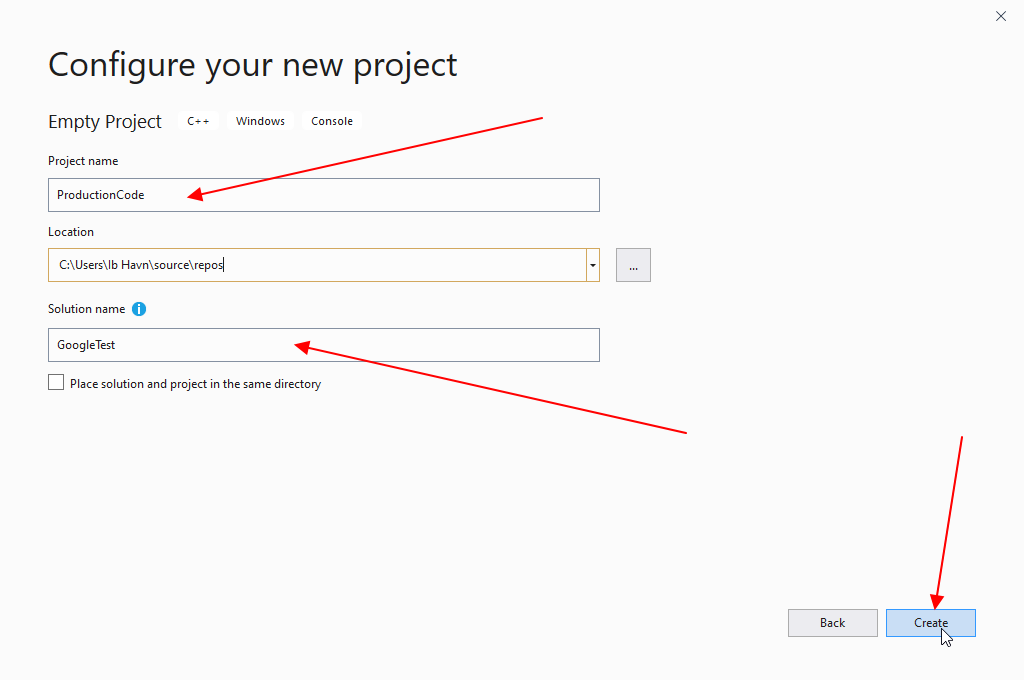


# Production Code Project

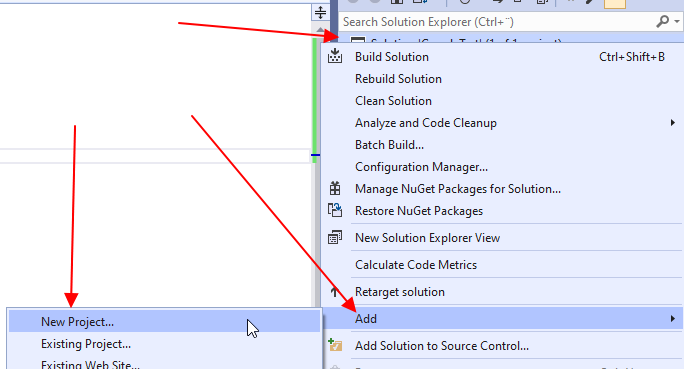
This project can be any C projects (including AVR Studio projects), but for the example it is made as a Visual Studio C project in a where a Test project will be added later on,

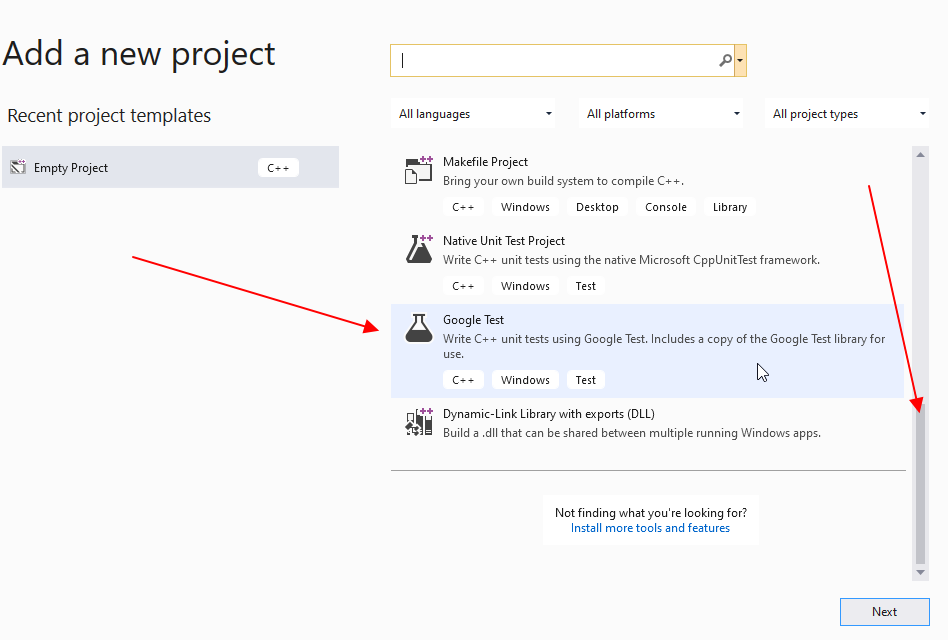


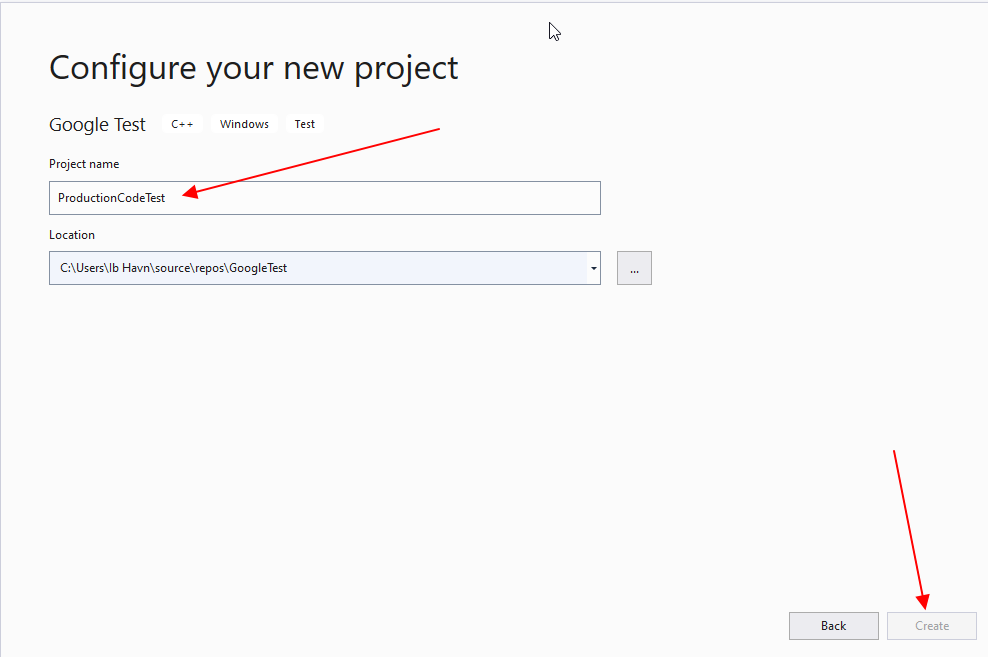


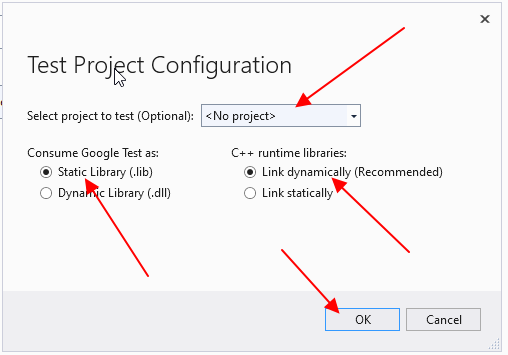


# The Test Project

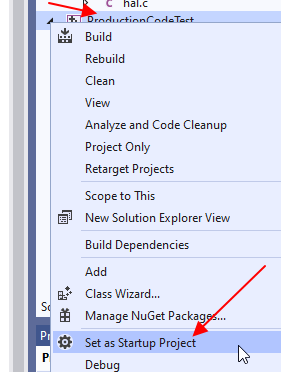




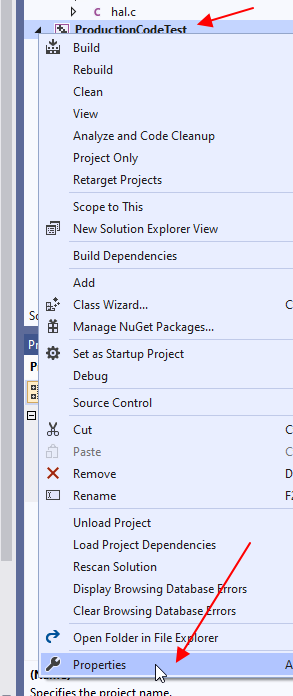




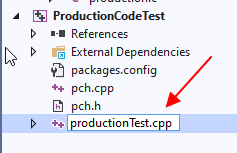
Right click on the test project (*ProductionCodeTest*) and *Set as Startup Project*.



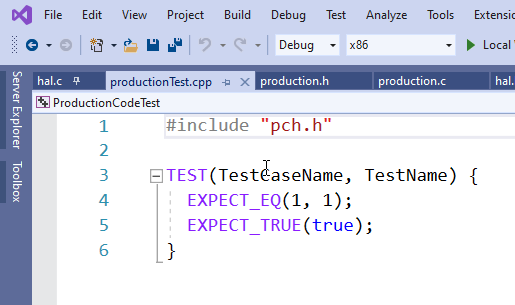
Right click on the test project (*ProductionCodeTest*) and *Properties*.



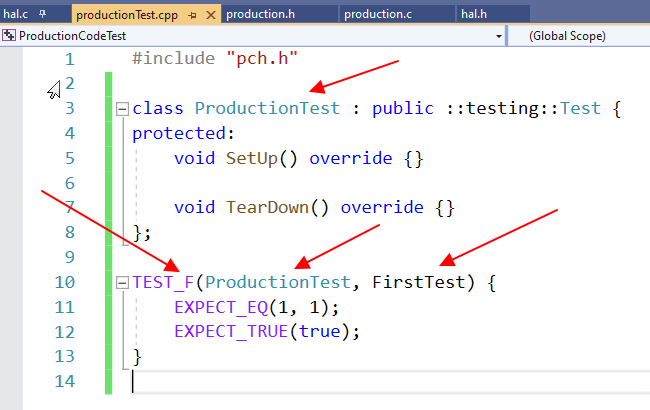
Rename *test.cpp* to *productionTest.cpp* to indicate that this test is testing then production code (*production.c*)



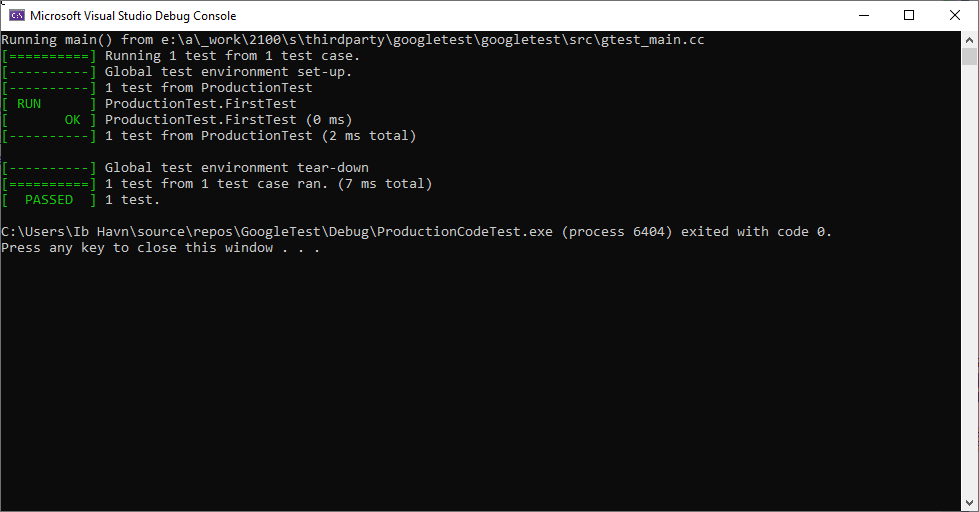
The test file looks like this now:



Change the contents by adding a Test fixture (*ProductionTest*) then we can have *SetUp()* and *TearDown()* functions.



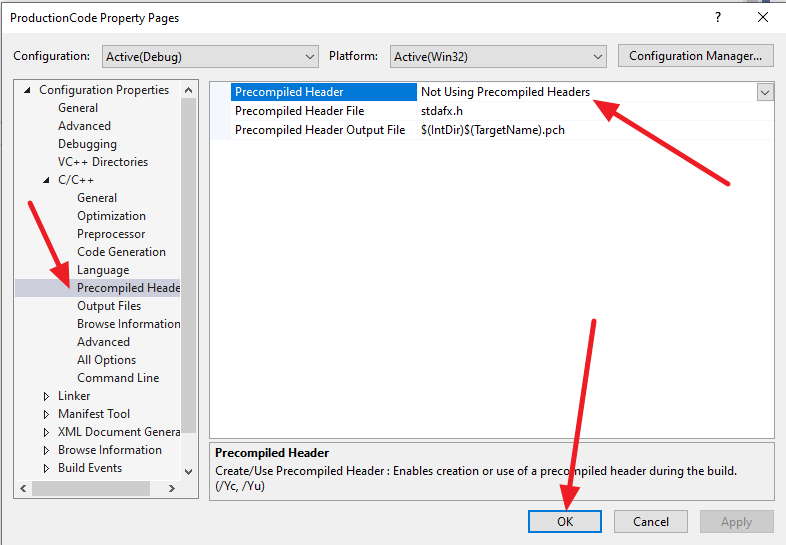
Try to run it (*ctrl+F5*) and you will see the result:



## Disable Precompiled Header

In both the *ProductionCode Project* and the *ProductionCodeTest Project* it is recommended to disable the use of precompiled headers.

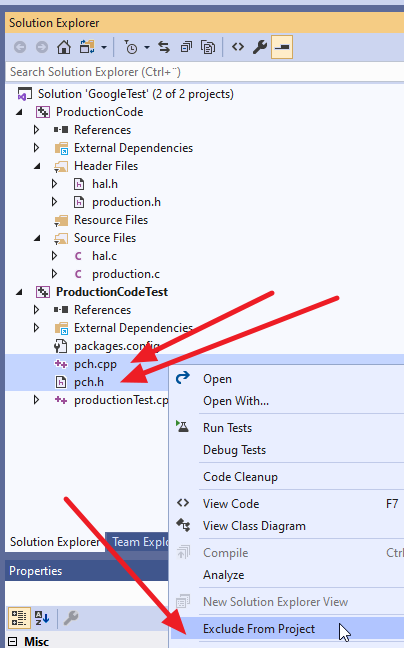
Goto *Properties* of **both projects** and set *Precompiled Header* to *Not Using Precompiled Headers* as shown here



Change the include files from *#include "pch.h"* to *#include "gtest/gtest.h"* in the test source file *productionTest.cpp*.



Remove *pch.h* and *pch.cpp* from the *ProductionCode Test* project. Select the two files in the *Solution Explorer*, Right Click and *Exclude From Build*.

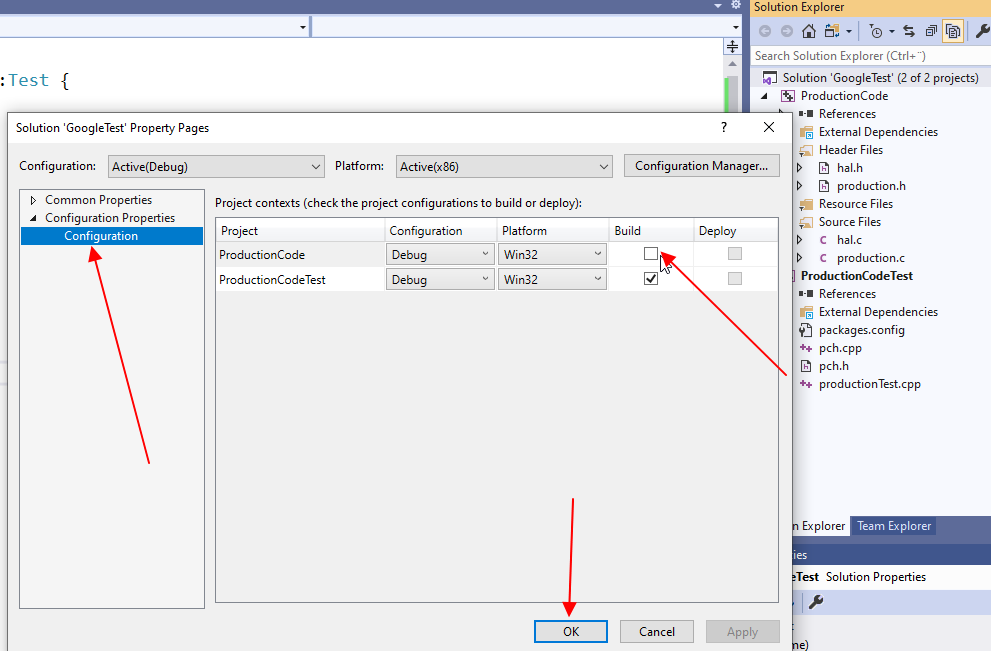


Then clean and rebuild the solution.

## Disable Build of Production Code Project

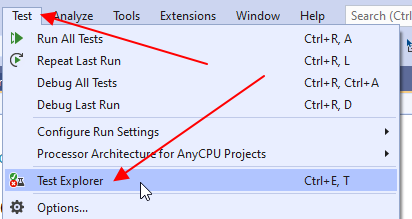
**Note: Disabling build of Production Code Project is only needed if the production code project is in the same solution as the test project.**

Before it can be executed in the build in *Test Explorer* we will disable building of the production code project – that will allow us to have bugs in part of the production code and still be able to test.

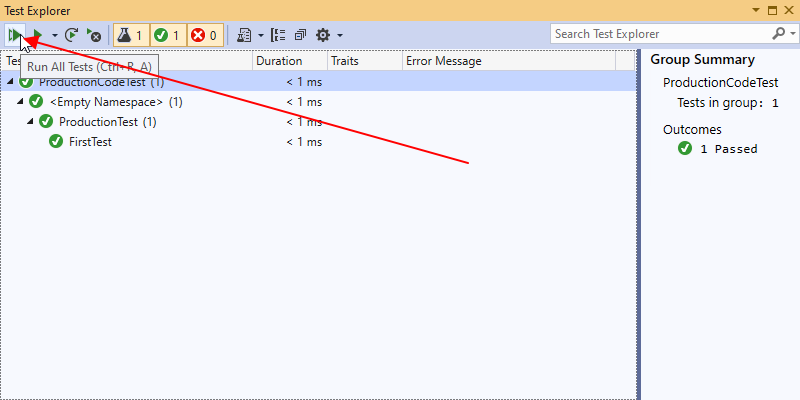


## Running Tests with Test Explorer

Now it is possible to use the *Test Explorer* in Visual Studio.



Run all test and it should be possible to see this:



## Mocking

Here we will use the *Fake Function Framework (fff)*[[1]](#footnote-1) for C. This framework is easy to use and consists of a single header file (*fff.h*) to be include in our Test project.

This framework as a very good documentation and can automatically generate mocks for functions in the production code.

### A small Mock example

A screenshot of a cell phone

Description automatically generated

Figure 1 Class Diagram over a simple example

In the small example seen in Figure 1 we should implement the Production part. The figure includes a Hardware Abstraction Layer (*HAL*) that somebody else will implement but they are not ready yet. We have only access to the interface but not the implementation (*HALImpl*).

Can we implement our part (*Production*) and be sure that it works? Yes, we can! The answer is mocking.

First, we will design how our *ProductionImpl* will interact with the HAL.

A screenshot of a cell phone

Description automatically generated

Figure 2 Sequence Diagram showing interaction with the HAL.

From Figure 2 we can see what we must test in our *ProductionImpl*. To be able to test that *ProductionsImpl’s* *create()* calls create(10) in *HAL*, here we need to mock out *HALImpl* since we have no access to it.

Let’s see how this can be done.

### Include FFF in Test Project

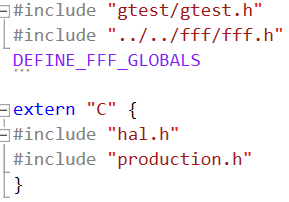
Clone or download FFF from <https://github.com/meekrosoft/fff>

Include “fff.h” in the test project from where it is located on your computer.

Use the macro *DEFINE\_FFF\_GLOBALS* to setup the FFF framework.

Include the header files *hal.h* and *production.h* to make the test file able to know the functions defined in these files. Remember *extern “C” {}* around them because they are implemented in C and not C++.

The start of the test file (*ProductionTest.cpp*) must now looks like this:

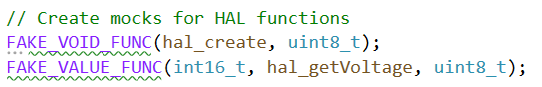


Now we are ready to make a mock for the HAL functions



This is done by using the macros *FAKE\_xxx\_FUNC* from FFF. The *xxx* must be substituted with return kind of the function.

The mocks for the above two functions are generated with these macro calls:

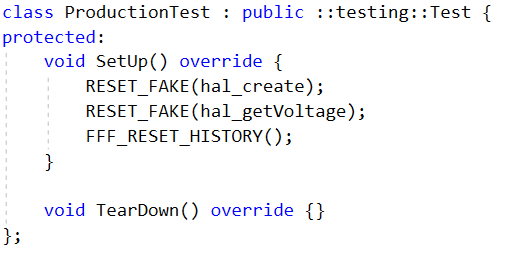


*hal\_create* is a void function so *FAKE\_VOID\_FUNC* is used, the first argument is the name of the function (*hal\_create*) and the second argument is the type of the first parameter (*uint8\_t*) the function will take.

*hal\_voltage* function is returning a value so *FAKE\_VALUE\_FUNC* is used, the first argument is the return type for the function (*int16\_t*) the second argument is the name of the function (*hal\_voltage*) and the third argument is the type of the first parameter (*uint8\_t*) the function will take.

That is all it takes to create mock functions 😊

Before each test is executed it is necessary to reset the mocks, this is done by calling the *RESET\_FAKE* macro on each mocked function. It is also necessary to reset the call history by calling *FFF\_RESET\_HISTORY* macro. These macros is typically called in the *Setup()* function in the test class.



Now the real functions are mocked, and the mock functions are ready to be used.

The mock functions are prefixed width *\_fake* so that *hal\_create* is mocked to *hal\_create\_fake*.

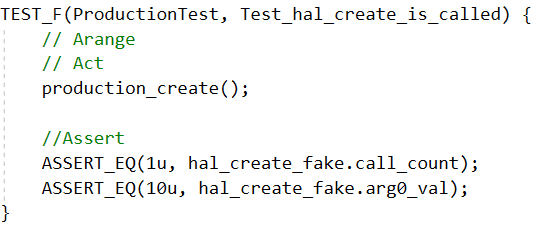
### Using Mocked functions in Test cases

We would like to test that *production\_create()* will call *hal\_create(10)* when *production\_create()* is called as seen in the SD:

A screenshot of a cell phone

Description automatically generated

The following test case will test that



There is nothing to *Arrange* for this test, the *Act* section calls the *production\_create()* function.

In the *Assert* section we check that the *hal\_create* (*hall\_create\_fake*) has been called exactly once *ASSERT\_EQ(1u, hal\_create\_fake.call\_count)*. And that it was called with the argument *10* as the first argument (*arg0\_val*) *ASSERT\_EQ(10u, hal\_create\_fake.arg0\_val)*.

We will now test that a call to *production\_meassureTemperature(12)* will call *hal\_getVoltage(12)* we will also fake that *hal\_getVoltage(12)* will return *15* and that a call to *production\_getTemperature(12)* will return *30* as seen in the design SD:

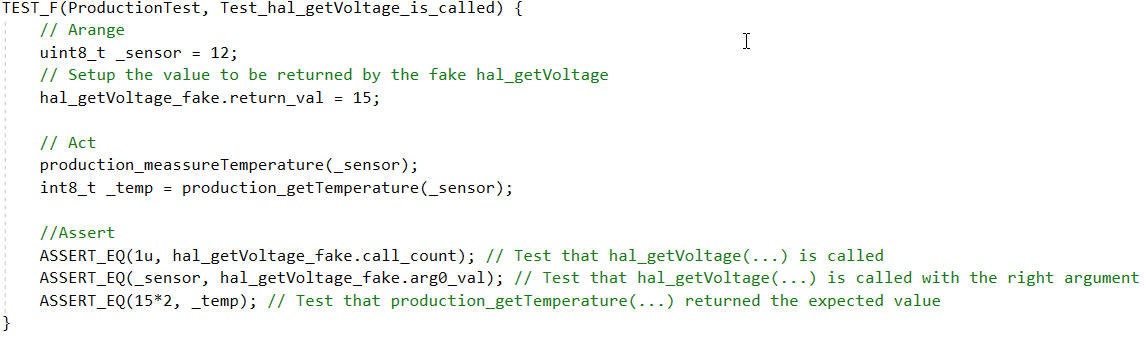
A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

The following test case will test that

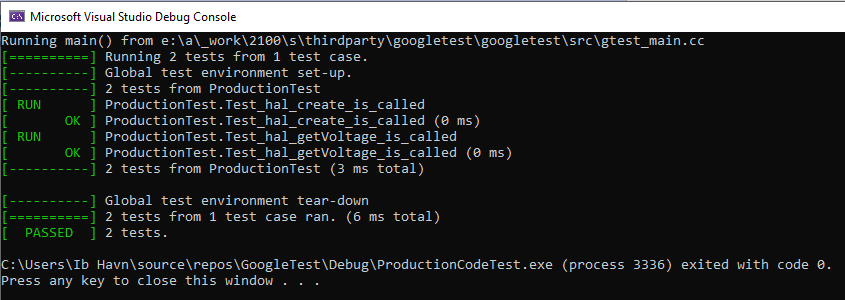


In *Arrange* for this test, the tell the fake *hal\_getVoltage* that it should return *15* when it is called *hal\_getVoltage\_fake.return\_val = 15*.

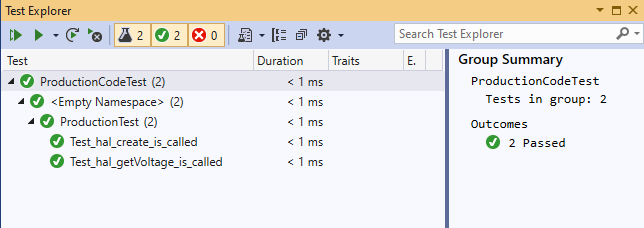
In *Act* we call *production\_meassureTemperature(12)* and *production\_getTemperature(12)*.

In the *Assert* section we check that the *hal\_getVoltage* (*hall\_getVoltage\_fake*) has been called exactly once *ASSERT\_EQ(1u, hal\_getVoltage\_fake.call\_count)*.   
And that it was called with the argument *12* as the first argument (*arg0\_val*) *ASSERT\_EQ(10u, hal\_getVoltage\_fake.arg0\_val)*.  
Finally we check that the value returned from *production\_getTemperature(12)* was *15\*2* *ASSERT\_EQ(15\*2, \_temp).*

The result looks like this if the test project is executed normally:



And like this if it is executed with the *Test Explorer* in Visual Studio:



# GitHub

All can be found in <https://github.com/ihavn/GoogleTestDemo>

1. <https://github.com/meekrosoft/fff> [↑](#footnote-ref-1)