RESHARPER C++ BLOG

Unit Testing C++ with Google Test

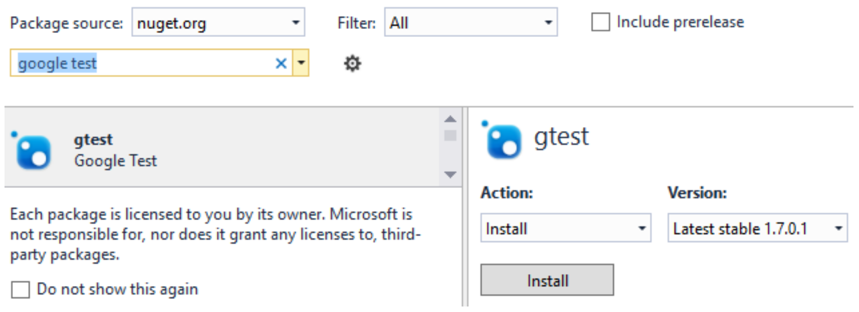
Posted on September 1, 2015 by Dmitri Nesteruk

Unit testing C++ applications is not exactly easy. Without any embeddable metadata, the actual process o
running a unit test has to be defined explicitly in code. Luckily, frameworks such as Google Test provide comprehensive support for this. Let's take a look.

Download and Build

Google Test is not header-only: there are libraries to build. So, as a Visual Studio user, you have essentially options.

Option 1 (probably the easiest): Just install Google Test from Nuget:


Package source: nuget.org
Filter: All
Include prerelease
google test
gtest
Google Test
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.
Do not show this again
gtest
Action:
Version:
Install
Latest stable 1.7.0.1
Install

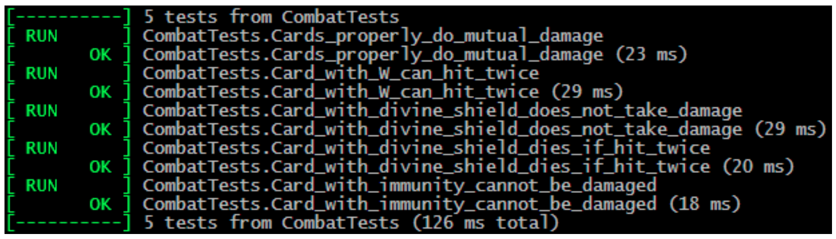
This sets everything up, but **your mileage may vary**: the package comes with pre-build binaries that may r
target the compiler and bitness you want. So the other option is…

Option 2 (more effort required): Download and build Google Test in some local directory (e.g., `c:\gtest`). N
lucky for us, Google Test comes with a set of Visual Studio projects/solutions ready for building the binarie
You'll find those in `\msvc`. Now is a chance for you to open up the solution (e.g., `gtest.sln`), configure thing
how you want, then build it and copy the library files and EXEs to some convenient location (e.g., `\gtest\l`

The icing on the cake is you can now open up the `%LOCALAPPDATA%\Microsoft\MSBuild\v4.0` folder and edit the `.user.props` files to always include the Google Test paths. Or use Visual Studio's own Property Manager. Th
way you'll always have the directories available, though the library to link against depends on whether you
building a Debug or Release version.

How It Works

Let's state the obvious first: **Google Test requires you to build an executable**. It's a command-line app, an
when you run it, it runs your tests. Even makes them nice and colorful:



```
[----------] 5 tests from CombatTests
[ RUN      ] CombatTests.Cards_properly_do_mutual_damage
[       OK ] CombatTests.Cards_properly_do_mutual_damage (23 ms)
[ RUN      ] CombatTests.Card_with_W_can_hit_twice
[       OK ] CombatTests.Card_with_W_can_hit_twice (29 ms)
[ RUN      ] CombatTests.Card_with_divine_shield_does_not_take_damage
[       OK ] CombatTests.Card_with_divine_shield_does_not_take_damage (29 ms)
[ RUN      ] CombatTests.Card_with_divine_shield_dies_if_hit_twice
[       OK ] CombatTests.Card_with_divine_shield_dies_if_hit_twice (20 ms)
[ RUN      ] CombatTests.Card_with_immunity_cannot_be_damaged
[       OK ] CombatTests.Card_with_immunity_cannot_be_damaged (18 ms)
[----------] 5 tests from CombatTests (126 ms total)
```

So, it is entirely possible to keep rebuilding and running the EXEs, visually inspect the results and fix when
necessary. Of course, this works when you have a few tests: if you have hundreds, well… this is where you r

tool support.

At any rate, the first thing you need to do is

- Add the include header

```
#include <gtest/gtest.h>
```

- Reference the appropriate libraries. If you are using the NuGet package, you don't have to do this, but otherwise you need to reference `gtest.lib` for the Release build and `gtestd.lib` for Debug.

So, in order to run the tests, Google Test needs you to implement the `main` entrypoint with code similar to:

```
int main(int ac, char* av[])
{
  testing::InitGoogleTest(&ac, av);
  return RUN_ALL_TESTS();
}
```

The call to `InitGoogleTest()` parses command-line arguments: you can specify your own arguments. To finc which arguments Google Test supports, just run your EXE with the `--help` flag.

What is a Test?

In its simplest form, a test is just a chunk of code that checks if some condition holds. If it holds, the test passes; if it doesn't, the test fails.

There are many ways to test conditions. The simplest varieties are `EXPECT_TRUE()` and `EXPECT_FALSE`, which ca be used as follows:

```
TEST(Addition, CanAddTwoNumbers)
{
  EXPECT_TRUE(add(2, 2) == 4);
}
```

You'll notice that the above expectation is wrapped in a `TEST()` macro. This macro is necessary to tell Goog Test that you are, in fact, writing a test. A single test can contain more than once statement, of course.

A slightly more useful variety is `EXPECT_EQ()`, which tests that the second argument is the same as the first, example

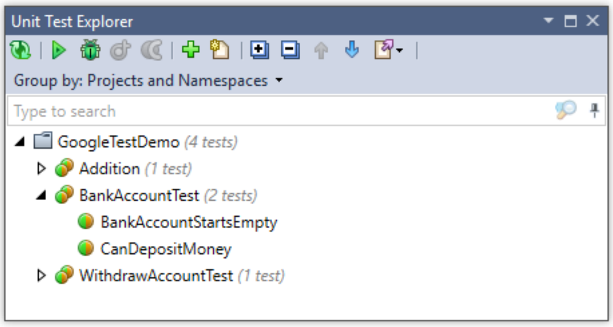
```
EXPECT_EQ(4, add(2,2)) << "Two plus two must equal four";
```

So the above states that 4 should be the result of adding 2 and 2. I also use the `<<` operator to give a hum readable message to whoever is running the test in case it fails.

Now, `EXPECT_EQ` **does not stop execution**. If you have ***several*** `EXPECT_EQ` calls in one test, they will ***all*** run. If y want the first test comparison to stop execution if it fails, use `ASSERT_EQ` instead.

Some refinements of `EXPECT_EQ` (and `ASSERT_`, by analogy) include comparisons (e.g., `EXPECT_LT` is a less-thai check), comparisons of floating-point numbers (`EXPECT_FLOAT_EQ` and similar), and many more.

At any rate, your `TEST()` is now runnable. Its location isn't important so long as it ends up being part of the build. If you want to figure out what tests you've got written, just use R++'s *Unit Test Explorer*, which will locate and list all the unit tests in the project.



```
GoogleTestDemo (4 tests)
  Addition (1 test)
  BankAccountTest (2 tests)
    BankAccountStartsEmpty
    CanDepositMoney
  WithdrawAccountTest (1 test)
```

Test Fixtures

Whenever you're testing stuff, you are probably testing a class or set of classes. Setting up and destroying them on every test might be painful and this is what test fixtures are for.

A test fixture is a class that inherits from `::testing::Test` and whose **internal state is accessible to tests th use it**. This is a critical distinction that might be a bit difficult to understand for users of frameworks in oth languages. Essentially, instead of being part of the test fixture class, the tests related to a fixture are *exter* Yeah, it's weird but that's what we have.

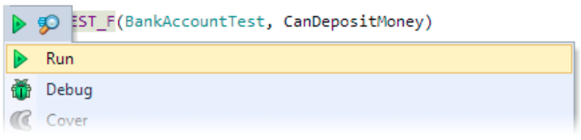
A test fixture can define set-up and tear-down actions in either `SetUp()/TearDown()` or in the constructor an destructor. For example, if I want to test a `BankAccount`, my test fixture might appear as:

```
struct BankAccountTest : testing::Test
{
  BankAccount* account;
  BankAccountTest()
  {
    account = new BankAccount;
  }
  virtual ~BankAccountTest()
  {
    delete account;
  }
};
```

But the test fixture is not an actual test: it's just rules for setting up and destroying objects that you need f testing. (Of course, we could have used a `unique_ptr` but I wanted to show a destructor, so there.) Anyway the actual test now uses a `TEST_F` instead of the usual `TEST`, and its first argument is the fixture:

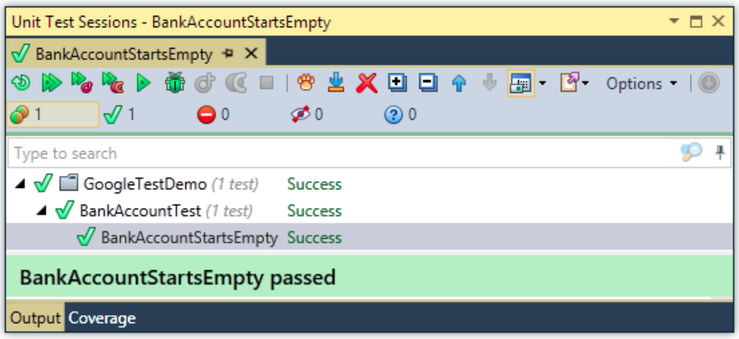
```
TEST_F(BankAccountTest, CanDepositMoney)
{
  account->deposit(100);
  EXPECT_EQ(100,account->balance);
}
```

With ReSharper C++, you can run this test by pressing `Alt`+`Enter` and choosing **Run**:



```
EST_F(BankAccountTest, CanDepositMoney)
```

Choosing this option causes the program to be compiled and the test executed. You can get a visual read-on the state of your tests in the **Unit Test Sessions** window:


Unit Test Sessions - BankAccountStartsEmpty

BankAccountStartsEmpty

```
GoogleTestDemo (1 test)       Success
  BankAccountTest (1 test)    Success
    BankAccountStartsEmpty    Success
```

BankAccountStartsEmpty passed

How Do I Test Lots of Data?

Data-driven tests? I'm glad you asked, because Google Test has something called *parameterized tests*. Essentially, you can just set up a set of values and feed them all consecutively into a test. First of all, set up your structure for running the test:

```
struct account_state
{
  int initial_balance;
  int withdraw_amount;
  int final_balance;
  bool success;
  friend std::ostream& operator<<(std::ostream& os, const account_state& obj)
  {
    return os
      << "initial_balance: " << obj.initial_balance
      << " withdraw_amount: " << obj.withdraw_amount
      << " final_balance: " << obj.final_balance
      << " success: " << obj.success;
  }
};
```

The above is a statement of before-and-after states for a bank account. We are testing the process of withdrawing money. Oh, notice how I used R++ to generate the `operator<<` pretty-print: this is necessary because Google Test has no idea how to print `account_state` objects to the console… and we kind of need t for good-looking tests.

Now comes the tricky part: we want to reuse our previous `BankAccountTest` fixture but, at the same time we want to use `account_state` instances to initialize the balance. Here's how it's done:

```
struct WithdrawAccountTest : BankAccountTest, testing::WithParamInterface<account_sta
{
  WithdrawAccountTest()
  {
    account->balance = GetParam().initial_balance;
  }
};
```

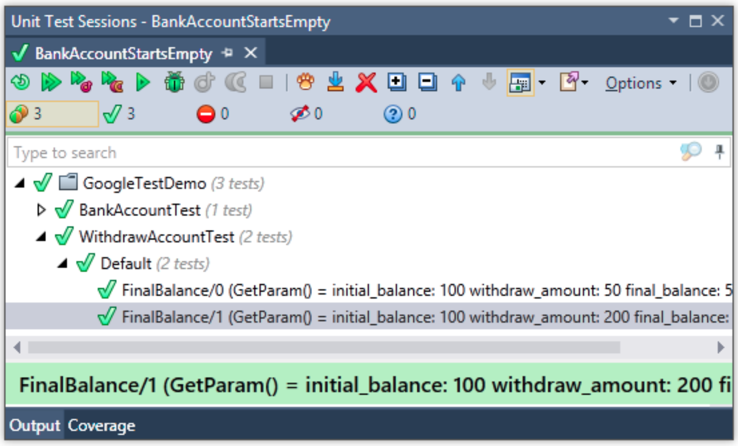
The magic sauce here is in the `WithParamInterface<>` class as well as the `GetParam()` function, which yields th parameter being used in the current test case. Finally, we can write the test itself…

```
TEST_P(WithdrawAccountTest, FinalBalance)
{
  auto as = GetParam();
  auto success = account->withdraw(as.withdraw_amount);
  EXPECT_EQ(as.final_balance,account->balance);
  EXPECT_EQ(as.success,success);
}
```

Notice the use of `TEST_P` here. The rest is pretty much the same: we get the test values with `GetParam()`, ext what we need, perform the test and check not one but two values. So finally, the icing on the cake is in us defining the test cases for this test. Using the magic of uniform initialization we can write it as follows:

```
INSTANTIATE_TEST_CASE_P(Default, WithdrawAccountTest,
  testing::Values(
  account_state{100,50,50,true},
  account_state{100,200,100,false}
  ));
```

When working with parameterized tests, you run them just as you would run a single test or a test based c
test fixture. The difference is in the output: each case takes up a separate line in the tree of test cases:



```
GoogleTestDemo (3 tests)
  BankAccountTest (1 test)
  WithdrawAccountTest (2 tests)
    Default (2 tests)
      FinalBalance/0 (GetParam() = initial_balance: 100 withdraw_amount: 50 final_balance: 5
      FinalBalance/1 (GetParam() = initial_balance: 100 withdraw_amount: 200 final_balance:

FinalBalance/1 (GetParam() = initial_balance: 100 withdraw_amount: 200 fi
```

There's More!

Google Test is a really big and comprehensive framework. Together with its sister, Google Mock, they prov
ample possibilities for unit testing. Check out the advanced guide to find out more about sophisticated
Google Test practices.

And here's a video illustrating the story described above.

Hey, did I mention that ReSharper C++ supports Google Test? Well, it does. So check it out if you haven't
already. ■

This entry was posted in ReSharper C++ Tips & Tricks and tagged google test, ReSharper C++, unit testing. Bookmark the permalink.

16 Responses to *Unit Testing C++ with Google Test*

Victor *says:*
September 2, 2015 at 3:13 pm

Another (easier) option to set up Google Test is to use the fused version which only requires adding one file, gtest-all.cc
the build. Also you don't have to implement main yourself, as gtest_main.cc already provides the implementation.

Reply

Dmitri Nesteruk *says:*
September 4, 2015 at 12:51 pm

Thanks for the pointer!

Reply

Harry *says:*
October 16, 2015 at 4:11 am

This seem to fail with visual studio 2015. It seem that nuget is doing something wrong.

Reply

Harry *says:*
October 16, 2015 at 9:29 pm

I got it working by make a few compiler changes

Reply

Ammar *says:*
February 7, 2016 at 11:44 am

Can you please explain what changes are required and how, for visual studio 2015?

Reply

Kel *says:*
March 3, 2016 at 4:56 pm

What changes did you make?

Reply

Kaustubh *says:*
May 8, 2016 at 4:22 am

Can you elaborate how you got it to work?

Reply

Igor Akhmetov *says:*
June 25, 2016 at 5:38 pm

The easiest way is to make a fused version of Google test using the fuse_gtest_files.py script from Google test distrib, and to include the resulting source files into your project.

Reply

ddb *says:*
October 16, 2015 at 3:52 pm

gtest prints some important information to the console for failing cases e.g.:
ASSERT_EQ(expected, value) << "this line will be printed to console, it includes some important info";

1. gtest will print the expected value and the value that did not match
2. gtest will print the text, after ASSERT macro that suppose to explain even better the failing case.

My question is, where can I find the gtest console output, when using Resharper C++ unittest environment?
thank you.

Reply

Igor Akhmetov *says:*
October 19, 2015 at 5:30 pm

All test output is saved and should be accessible in the output pane of the “Unit Tests Sessions” window – just m
sure that the pane is visible (there’s the “Show Output” toolbar button that controls visibility of the pane).

There are a couple of known issues:
1) Output of 64-bit executables is not logged while they are debugged.
2) Output pane sometimes does not get properly updated between the runs (https://youtrack.jetbrains.com/issue/RSRP-447841) – we hope to fix this in ReSharper 10.

If you still can’t see the output and the issue is not covered by one of the former cases, we’d love to know how tc
reproduce it – please create an issue on our tracker (https://youtrack.jetbrains.com/issues?q=%23RSCPP) with th
required steps.

Reply

Guglielmo *says:*
February 6, 2016 at 11:18 am

It seems that x64 is not supported, is it?

Reply

Igor Akhmetov *says:*
June 25, 2016 at 5:33 pm

x64 binaries are supported with one caveat – after you debug some unit tests, their test status will be inconclusi
ReSharper C++ is not able to intercept the output of a x64 binary under debugger.

Reply

hung tran *says:*
February 25, 2016 at 3:02 pm

hi,
I am trying to setup and do one simple unit test using GoogleTest but always struggling in Visual Studio.
Do you have any reference to the fused version as far as setting it up .
thank you

Reply

Igor Akhmetov *says:*
June 25, 2016 at 5:31 pm

Hung, you have to run the googletest/scripts/fuse_gtest_files.py script from the Google test distribution to obta
the fused sources – then just add them to your solution.

Reply

Phil Miller *says:*
April 15, 2016 at 4:50 pm

Do you have plans to extend these capabilities to the Boost.Test framework or will this remain a google-test only capab

Reply

Igor Akhmetov *says:*
June 25, 2016 at 5:28 pm

Phil, ReSharper C++ 2016.1 added support for running tests written using the Boost.Test framework.

Reply

ReSharper C++ Blog
Proudly powered by WordPress.