Pytest C

Release

John McNamara

Mar 08, 2018

Contents

CHAPTER 1

The C Unit Tests

This isn't about how to write C unit tests.

The assumption is that you already have working tests that you just want to run and that these tests output some sort of Pass/Fail information.

For the sake of example say we have a test case like this:

```
#include "test_macros.h"

void test_some_strings()
{
    char *foo = "This is foo";
    char *bar = "This is bar";

    ASSERT_EQUAL_STR(foo, foo);
    ASSERT_EQUAL_STR(foo, bar);
}

int main()
{
    test_some_strings();

    return 0;
}
```

And when we compile and run it we get output like this:

```
$ test/test_basic_strings

[PASS] test_basic_strings.c:test_some_strings():8

[FAIL] test_basic_strings.c:test_some_strings():9
  [TST] ASSERT_EQUAL_STR(foo, bar)
  [EXP] This is foo
  [GOT] This is bar
```

This is typical of a lot of test output and although the format is simple it contains a lot of useful information:

- The Pass/Fail condition.
- The C file name.
- The function/test name.
- The line number of the test assertion.
- The assertion that failed.
- The expected and actual output.

So, let's see how we can run this or any number of similar tests automatically and capture and summarise the output.

The first step is to write the *The Pytest hooks*.

CHAPTER 2

The Pytest hooks

One of the many useful features of `pytest` is its easy extensibility. The `pytest` documentation has a full description of Working with non-python tests on which this C unit test extension is heavily based.

At its simplest, `pytest` supports the creation of hooks in a file called `conftest.py` that can be placed in the root folder where we wish to run the tests. This allows us to define, in Python, code to find test files, run them, and collect the results. The nice thing about this is that we can create a hook layer between `pytest` and the C unit tests without changing code in either one.

The `conftest.py` file is actually a per-directory file, so we can have different files for different test directories if required.

2.1 The conftest.py file

First let's look at the C unit test `conftest.py` in its entirety:

```
import subprocess
import pytest
import os


def pytest_collect_file(parent, path):
    """
    A hook into py.test to collect test_*.c test files.

    """
    if path.ext == ".c" and path.basename.startswith("test_"):
        return CTestFile(path, parent)


class CTestFile(pytest.File):
    """
    A custom file handler class for C unit test files.
```

```
    """

    def collect(self):
        """
        Overridden collect method to collect the results from each
        C unit test executable.

        """
        # Run the exe that corresponds to the .c file and capture the output.
        test_exe = os.path.splitext(str(self.fspath))[0]
        test_output = subprocess.check_output(test_exe)

        # Clean up the unit test output and remove non test data lines.
        lines = test_output.decode().split("\n")
        lines = [line.strip() for line in lines]
        lines = [line for line in lines if line.startswith("[")]

        # Extract the test metadata from the unit test output.
        test_results = []
        for line in lines:
            token, data = line.split(" ", 1)
            token = token[1:-1]

            if token in ("PASS", "FAIL"):
                file_name, function_name, line_number = data.split(":")
                test_results.append({"condition": token,
                                     "file_name": file_name,
                                     "function_name": function_name,
                                     "line_number": int(line_number),
                                     "EXP": 'EXP(no data found)',
                                     "GOT": 'GOT(no data found)',
                                     "TST": 'TST(no data found)',
                                     })
            elif token in ("EXP", "GOT", "TST"):
                test_results[-1][token] = data

        for test_result in test_results:
            yield CTestItem(test_result["function_name"], self, test_result)


class CTestItem(pytest.Item):
    """
    Pytest.Item subclass to handle each test result item. There may be
    more than one test result from a test function.

    """

    def __init__(self, name, parent, test_result):
        """Overridden constructor to pass test results dict."""
        super(CTestItem, self).__init__(name, parent)
        self.test_result = test_result

    def runtest(self):
        """The test has already been run. We just evaluate the result."""
        if self.test_result["condition"] == "FAIL":
            raise CTestException(self, self.name)

    def repr_failure(self, exception):
```

```
        """
        Called when runtest() raises an exception. The method is used
        to format the output of the failed test result.

        """
        if isinstance(exception.value, CTestException):
            return ("Test failed : {TST} at {file_name}:{line_number}\n"
                    "          got: {GOT}\n"
                    "     expected: {EXP}\n".format(**self.test_result))

    def reportinfo(self):
        """Called to display header information about the test case."""
        return self.fspath, self.test_result["line_number"] - 1, self.name


class CTestException(Exception):
    """Custom exception to distinguish C unit test failures from others."""
    pass
```

This is less than 100 lines of Python, including comments, but it provides a lot of functionality.

If you would like to see this functionality in use then move on to *Running the Tests*.

If you are interested in seeing how this functionality is implemented, and how you could extend `pytest` for similar tests then read on below and we will see how the `conftest.py` code works.

Running the Tests or the Gory Details. Choose your own adventure!

2.2 Finding test files

The first function in the code is a `pytest` hook function called `pytest_collect_file()`:

```
def pytest_collect_file(parent, path):
    if path.ext == ".c" and path.basename.startswith("test_"):
        return CTestFile(path, parent)
```

This collects files based on any rule that we wish to write. In this case it collects files that look like `test_*.c` but we could make the rule as specific as we wanted.

Once a file of the correct type is found a `pytest.Node` object of the corresponding type is created and returned. In our case this is a `CTestFile` object which is derived from the `pytest.File` class.

2.3 The CTestFile Class

The `CTestFile` object that we instantiated in the `pytest_collect_file()` hook inherits from the `pytest.File` class which in turn inherits from `pytest.Collection` and `pytest.Node`. The `Node` class and its subclasses have a `collect()` method which returns `pytest.Items` objects:

```
class CTestFile(pytest.File):

    def collect(self):
        # Returns pytest.Items.
```

The `pytest` hierarchy and methods are explained in more detail in the Working with plugins and conftest files section of the Pytest documentation.

Depending on the types of tests that are being run the collected items might be individual test results or even test cases that are being staged to run.

However, in our case we are going to take a simplified approach that lends itself to statically compiled test cases such as C unit tests:

- For each `test_something.c` file assume there is a `test_something` executable.
- Run the `test_something` executable and capture the output.
- Parse the Pass/Fail results and any available metadata.
- Return each test results as a python dict with information that can be used for reporting.

2.4 Collecting the C unit test data

So with this methodology in mind the first step is to run the collected C unit tests and capture the output.

As stated above we are going to assume that the C unit tests are structured so that for each `test_something.c` source file there is a `test_something` executable. This is a reasonable assumption based on the way most test suites are laid out but it may not hold for specific implementations where, for example, multiple `.c` files might be compiled to object files and linked into a single test runner executable. For cases like that more a sophisticated `pytest_collect_file()` implementation can be used.

We are also going to assume, again reasonably, that the unit test executables are tied into a build system and have already been built via `make`, `make test` or something similar.

In `conftest.py` the following code runs the executable that corresponds to the `.c` file and captures the output:

```
def collect(self):

    test_exe = os.path.splitext(str(self.fspath))[0]
    test_output = subprocess.check_output(test_exe)
```

A more robust implementation would probably confirm the existence of the executable and return a fail condition if it wasn't present.

2.5 Extracting the test data

This section of the code shows the main functionality that converts the output of the test cases into a format that can be used by `pytest`. If you are using this document as a guide to running your own tests then this section is the part that you will have to modify to conform to your test output.

In our sample unit test the captured output will look something like the following:

```
[PASS] test_basic_strings.c:test_some_strings():8

[FAIL] test_basic_strings.c:test_some_strings():9
  [TST] ASSERT_EQUAL_STR(foo, bar)
  [EXP] This is foo
  [GOT] This is bar
```

We clean up the unit test output and remove non test data lines as follows:

```
lines = test_output.split("\n")
lines = [line.strip() for line in lines]
lines = [line for line in lines if line.startswith("[")]
```

We then extract the test metadata from the reformatted test output:

```
test_results = []
for line in lines:
    token, data = line.split(" ", 1)
    token = token[1:-1]

    if token in ("PASS", "FAIL"):
        file_name, function_name, line_number = data.split(":")
        test_results.append({"condition": token,
                             "file_name": file_name,
                             "function_name": function_name,
                             "line_number": int(line_number),
                             "EXP": 'EXP(no data found)',
                             "GOT": 'GOT(no data found)',
                             "TST": 'TST(no data found)',
                             })
    elif token in ("EXP", "GOT", "TST"):
        test_results[-1][token] = data
```

Once this is complete we should end up with a collection of data structures like the following:

```
{'condition':     'FAIL',
 'file_name':     'test_basic_strings.c',
 'function_name': 'test_some_strings()',
 'TST':           'ASSERT_EQUAL_STR(foo, bar)',
 'EXP':           'This is foo',
 'GOT':           'This is bar',
 'line_number':   9 }
```

These results are then returned as a `pytest.Item`:

```
for test_result in test_results:
    yield CTestItem(test_result["function_name"], self, test_result)
```

Note, it isn't essential that we capture all of the information shown above. None of it is strictly required by `pytest` apart from the test function name. The idea here is that we try to capture any usefule information that we want to display in the testrunner output. In the next section we will see how we can format and display that information.

2.6 Formatting the test report

The `pytest.Item` that we return in the previous step is an instance of a subclass so that we can control the test result formatting. We also override the constructor so that we can pass the test result data structure as an additional parameter:

```
class CTestItem(pytest.Item):

    def __init__(self, name, parent, test_result):
        super(CTestItem, self).__init__(name, parent)
        self.test_result = test_result
```

To control the test result reporting and formatting we have to override three `pytest.Item` methods: `runtest()`, `repr_failure()` and `reportinfo()`.

In our case, the `runtest()` method isn't actually used to run a test since we already did that in the `collect()` method of our `CTestFile` class. Instead we just check for `FAIL` results and throw an exception when we find one:

```
def runtest(self):
    if self.test_result["condition"] == "FAIL":
        raise CTestException(self, self.name)
```

We use a user defined exception class in order to distinguish it from other exceptions. The exception is then caught and handled in the `repr_failure()` method where we format the output for the failed test case:

```
def repr_failure(self, exception):
    if isinstance(exception.value, CTestException):
        return ("Test failed : {TST} at {file_name}:{line_number}\n"
                "          got: {GOT}\n"
                "     expected: {EXP}\n".format(**self.test_result))
```

Finally we provide one additional piece of information that will be used in verbose test display:

```
def reportinfo(self):
    return self.fspath, self.test_result["line_number"] - 1, self.name
```

2.7 Achievement Unlocked

Congratulations. You made it to the end of the code and have unlocked the **Adventurous** badge.

Now let's see how to we go about *Running the Tests*.

CHAPTER 3

Running the Tests

So finally we are at the stage where we can run the test cases.

If you'd like to try this for yourself the code is included in a project on GitHub so that you try out different options. The examples below show some of the more common ones.

3.1 The Default Output

Here is the default output from `py.test` (the command line testrunner of `pytest`):

```
$ make

$ py.test
========================== test session starts ==========================
platform darwin -- Python 2.7.2 -- py-1.4.20 -- pytest-2.5.2
collected 9 items

test/test_basic_integers.c ..FF
test/test_basic_strings.c .FFF.

=============================== FAILURES ================================
____________________________ test_more_integers() ____________________________
Test failed : ASSERT_EQUAL_STR(313, 33) at test_basic_integers.c:19
         got: 33
    expected: 313

____________________________ test_more_integers() ____________________________
Test failed : ASSERT_EQUAL_STR(12, 2) at test_basic_integers.c:20
         got: 2
    expected: 12

____________________________ test_some_strings() ____________________________
Test failed : ASSERT_EQUAL_STR(foo, bar) at test_basic_strings.c:17
         got: This is bar
```

```
    expected: This is foo

_____________________________ test_more_strings() ______________________________
Test failed : ASSERT_EQUAL_STR(bar, bar + 1) at test_basic_strings.c:25
         got: his is bar
    expected: This is bar

_____________________________ test_more_strings() ______________________________
Test failed : ASSERT_EQUAL_STR(foo, NULL) at test_basic_strings.c:26
         got: (null)
    expected: This is foo

================== 5 failed, 4 passed in 0.19 seconds ==================
```

3.2 The Verbose Output

Here is the verbose output:

```
$ py.test -v
========================== test session starts ==========================
platform darwin -- Python 2.7.2 -- py-1.4.20 -- pytest-2.5.2
collected 9 items

test/test_basic_integers.c:13: test_some_integers() PASSED
test/test_basic_integers.c:14: test_some_integers() PASSED
test/test_basic_integers.c:19: test_more_integers() FAILED
test/test_basic_integers.c:20: test_more_integers() FAILED
test/test_basic_strings.c:16: test_some_strings() PASSED
test/test_basic_strings.c:17: test_some_strings() FAILED
test/test_basic_strings.c:25: test_more_strings() FAILED
test/test_basic_strings.c:26: test_more_strings() FAILED
test/test_basic_strings.c:27: test_more_strings() PASSED

============================== FAILURES ==============================
_____________________________ test_more_integers() ______________________________
Test failed : ASSERT_EQUAL_STR(313, 33) at test_basic_integers.c:19
         got: 33
    expected: 313

_____________________________ test_more_integers() ______________________________
Test failed : ASSERT_EQUAL_STR(12, 2) at test_basic_integers.c:20
         got: 2
    expected: 12

_____________________________ test_some_strings() ______________________________
Test failed : ASSERT_EQUAL_STR(foo, bar) at test_basic_strings.c:17
         got: This is bar
    expected: This is foo

_____________________________ test_more_strings() ______________________________
Test failed : ASSERT_EQUAL_STR(bar, bar + 1) at test_basic_strings.c:25
         got: his is bar
    expected: This is bar

_____________________________ test_more_strings() ______________________________
```

```
Test failed : ASSERT_EQUAL_STR(foo, NULL) at test_basic_strings.c:26
         got: (null)
    expected: This is foo

================== 5 failed, 4 passed in 0.23 seconds ==================
```

The first part of this is shown is colour:

```
============================= test session starts ==============================
platform linux2 -- Python 2.7.3 -- py-1.4.20 -- pytest-2.5.2 -- /usr/bin/python
collected 9 items

test/test_basic_integers.c:14: test_some_integers() PASSED
test/test_basic_integers.c:15: test_some_integers() PASSED
test/test_basic_integers.c:21: test_more_integers() FAILED
test/test_basic_integers.c:22: test_more_integers() FAILED
test/test_basic_strings.c:16: test_some_strings() PASSED
test/test_basic_strings.c:17: test_some_strings() PASSED
test/test_basic_strings.c:26: test_more_strings() FAILED
test/test_basic_strings.c:27: test_more_strings() FAILED
test/test_basic_strings.c:28: test_more_strings() PASSED
```

3.3 Minimal Output

Here is some "quiet" output with the trace back hidden:

```
$ py.test -q --tb=no
..FF.FFF.
5 failed, 4 passed in 0.19 seconds
```

3.4 Filtered Tests

Here are results from all tests filtered to show only ones that match "strings" in the name:

```
$ py.test -k strings
========================== test session starts ==========================
platform darwin -- Python 2.7.2 -- py-1.4.20 -- pytest-2.5.2
collected 9 items

test/test_basic_strings.c .FFF.

================================ FAILURES ================================
____________________________ test_some_strings() ____________________________
Test failed : ASSERT_EQUAL_STR(foo, bar) at test_basic_strings.c:17
         got: This is bar
    expected: This is foo

____________________________ test_more_strings() ____________________________
```

```
Test failed : ASSERT_EQUAL_STR(bar, bar + 1) at test_basic_strings.c:25
         got: his is bar
    expected: This is bar

_____________________________ test_more_strings() _______________________________
Test failed : ASSERT_EQUAL_STR(foo, NULL) at test_basic_strings.c:26
         got: (null)
    expected: This is foo

================== 4 tests deselected by '-kstrings' ===================
=========== 3 failed, 2 passed, 4 deselected in 0.19 seconds ===========
```

3.5 Other py.test Options

Other testrunner options are shown in the Pytest Usage and Invocations documentation.

You can also *Learn More* about this document and the sample test code.

CHAPTER 4

Learn More

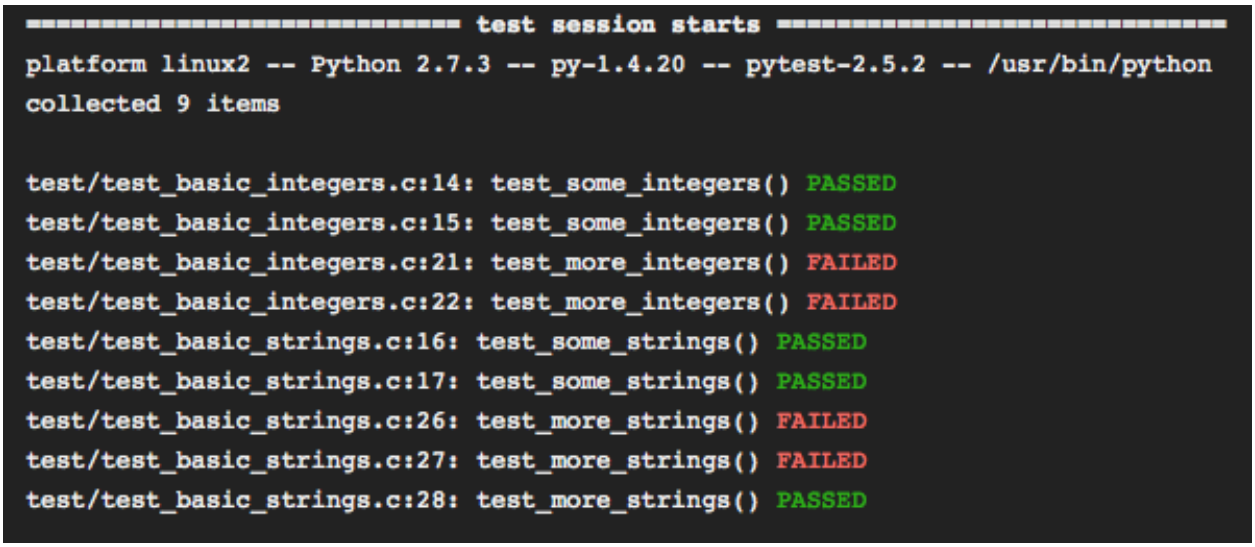
The source code for the Python hooks and C unit tests are on GitHub: http://github.com/jmcnamara/pytest_c_testrunner

The documentation is on ReadTheDocs: http://pytest-c-testrunner.readthedocs.org

CHAPTER 5

Using pytest as a testrunner for C unit tests

This document shows how to use the Python `pytest` test tool to run unit tests written in C.



CHAPTER 6

Wait, what?

I'm going to show you how to run C unit tests using the Python test tool `pytest`.

CHAPTER 7

Are you sure that is a good idea?

No. I'm pretty sure it isn't.

If you want to write and run C unit tests then there are a lot of better alternatives such as Unity or GoogleTest (for C/C++ testing) or many others.

CHAPTER 8

So, what is the use case?

Established C test frameworks are a better alternative when you are starting a project from scratch. However, if you have existing C unit tests that you just want to execute, then rewriting them to conform to a new framework probably isn't worth the effort.

Equally, writing a testrunner that is flexible enough to run all or a subset of the tests, that can output the results in JUnit or other formats, or that can filter results into concise reports probably isn't worth the effort either.

In which case it would be better to use an existing testrunner that supports all these features and that can be easily extended to capture the output from existing C unit tests without having to modify them.

CHAPTER 9

Why Pytest?

Pytest is a really nice Python testing tool.

It has good documentation, clean code, lots of tests, a large but clear set of options for running tests and collecting results and best of all it is easily extensible.

CHAPTER 10

Okay. I'm still with you, but barely

Then read on and I'll see if I can convince you with a working example.

So let's start with *The C Unit Tests*.