Unity Helper Scripts

With a Little Help From Our Friends

Sometimes what it takes to be a really efficient C programmer is a little non-C. The Unity project includes a couple Ruby scripts for making your life just a tad easier. They are completely optional. If you choose to use them, you'll need a copy of Ruby, of course. Just install whatever the latest version is, and it is likely to work. You can find Ruby at ruby-lang.org.

`generate_test_runner.rb`

Are you tired of creating your own `main` function in your test file? Do you keep forgetting to add a `RUN_TEST` call when you add a new test case to your suite? Do you want to use CMock or other fancy add-ons but don't want to figure out how to create your own `RUN_TEST` macro?

Well then we have the perfect script for you!

The `generate_test_runner` script processes a given test file and automatically creates a separate test runner file that includes `main` to execute the test cases within the scanned test file. All you do then is add the generated runner to your list of files to be compiled and linked, and presto you're done!

This script searches your test file for void function signatures having a function name beginning with "test" or "spec". It treats each of these functions as a test case and builds up a test suite of them. For example, the following includes three test cases:

```
void testVerifyThatUnityIsAwesomeAndWillMakeYourLifeEasier(void)
{
  ASSERT_TRUE(1);
}
void test_FunctionName_should_WorkProperlyAndReturn8(void) {
  ASSERT_EQUAL_INT(8, FunctionName());
}
void spec_Function_should_DoWhatItIsSupposedToDo(void) {
  ASSERT_NOT_NULL(Function(5));
}
```

You can run this script a couple of ways. The first is from the command line:

```
ruby generate_test_runner.rb TestFile.c NameOfRunner.c
```

Alternatively, if you include only the test file parameter, the script will copy the name of the test file and automatically append "_Runner" to the name of the generated file. The example immediately below will create TestFile_Runner.c.

```
ruby generate_test_runner.rb TestFile.c
```

You can also add a YAML file to configure extra options. Conveniently, this YAML file is of the same format as that used by Unity and CMock. So if you are using YAML files already, you can simply pass the very same file into the generator script.

```
ruby generate_test_runner.rb TestFile.c my_config.yml
```

The contents of the YAML file `my_config.yml` could look something like the example below. If you're wondering what some of these options do, you're going to love the next section of this document.

```
:unity:
  :includes:
    - stdio.h
    - microdefs.h
  :cexception: 1
  :suit_setup: "blah = malloc(1024);"
  :suite_teardown: "free(blah);"
```

If you would like to force your generated test runner to include one or more header files, you can just include those at the command line too. Just make sure these are *after* the YAML file, if you are using one:

```
ruby generate_test_runner.rb TestFile.c my_config.yml extras.h
```

Another option, particularly if you are already using Ruby to orchestrate your builds—or more likely the Ruby-based build tool Rake—is requiring this script directly. Anything that you would have specified in a YAML file can be passed to the script as part of a

hash. Let's push the exact same requirement set as we did above but this time through Ruby code directly:

```
require "generate_test_runner.rb"
options = {
  :includes => ["stdio.h", "microdefs.h"],
  :cexception => 1,
  :suite_setup => "blah = malloc(1024);",
  :suite_teardown => "free(blah);"
}
UnityTestRunnerGenerator.new.run(testfile, runner_name, options)
```

If you have multiple files to generate in a build script (such as a Rakefile), you might want to instantiate a generator object with your options and call it to generate each runner thereafter. Like thus:

```
gen = UnityTestRunnerGenerator.new(options)
test_files.each do |f|
  gen.run(f, File.basename(f,'.c')+"Runner.c"
end
```

Options accepted by generate_test_runner.rb:

The following options are available when executing `generate_test_runner`. You may pass these as a Ruby hash directly or specify them in a YAML file, both of which are described above. In the `examples` directory, Example 3's Rakefile demonstrates using a Ruby hash.

`:includes`

> This option specifies an array of file names to be `#include`'d at the top of your runner C file. You might use it to reference custom types or anything else universally needed in your generated runners.

`:suite_setup`

> Define this option with C code to be executed *before any* test cases are run.

`:suite_teardown`

> Define this option with C code to be executed *after all* test cases have finished.

`:enforce_strict_ordering`

This option should be defined if you have the strict order feature enabled in CMock (see CMock documentation). This generates extra variables required for everything to run smoothly. If you provide the same YAML to the generator as used in CMock's configuration, you've already configured the generator properly.

`:plugins`

This option specifies an array of plugins to be used (of course, the array can contain only a single plugin). This is your opportunity to enable support for CException support, which will add a check for unhandled exceptions in each test, reporting a failure if one is detected. To enable this feature using Ruby:

```
:plugins => [ :cexception ]
```

Or as a yaml file:

```
:plugins:
  -:cexception
```

If you are using CMock, it is very likely that you are already passing an array of plugins to CMock. You can just use the same array here. This script will just ignore the plugins that don't require additional support.

`unity_test_summary.rb`

A Unity test file contains one or more test case functions. Each test case can pass, fail, or be ignored. Each test file is run individually producing results for its collection of test cases. A given project will almost certainly be composed of multiple test files. Therefore, the suite of tests is comprised of one or more test cases spread across one or more test files. This script aggregates individual test file results to generate a summary of all executed test cases. The output includes how many tests were run, how many were ignored, and how many failed. In addition, the output includes a listing of which specific tests were ignored and failed. A good example of the breadth and details of these results can be found in the `examples` directory. Intentionally ignored and failing tests in this project generate corresponding entries in the summary report.

If you're interested in other (prettier?) output formats, check into the Ceedling build tool project (ceedling.sourceforge.net) that works with Unity and CMock and supports xunit-style xml as well as other goodies.

This script assumes the existence of files ending with the extensions `.testpass` and `.testfail`. The contents of these files includes the test results summary corresponding to each test file executed with the extension set according to the presence or absence of failures for that test file. The script searches a specified path for these files, opens each one it finds, parses the results, and aggregates and prints a summary. Calling it from the command line looks like this:

```
ruby unity_test_summary.rb build/test/
```

You can optionally specify a root path as well. This is really helpful when you are using relative paths in your tools' setup, but you want to pull the summary into an IDE like Eclipse for clickable shortcuts.

```
ruby unity_test_summary.rb build/test/ ~/projects/myproject/
```

Or, if you're more of a Windows sort of person:

```
ruby unity_test_summary.rb build\teat\ C:\projects\myproject\
```

When configured correctly, you'll see a final summary, like so:

```
--------------------------
UNITY IGNORED TEST SUMMARY
--------------------------
blah.c:22:test_sandwiches_should_HaveBreadOnTwoSides:IGNORE

--------------------------
UNITY FAILED TEST SUMMARY
--------------------------
blah.c:87:test_sandwiches_should_HaveCondiments:FAIL:Expected 1 was 0
meh.c:38:test_soda_should_BeCalledPop:FAIL:Expected "pop" was "coke"

--------------------------
OVERALL UNITY TEST SUMMARY
--------------------------
45 TOTAL TESTS 2 TOTAL FAILURES 1 IGNORED
```

How convenient is that?