MUnit: Lightweight Unit Testing in C

Introduction

MUnit provides light, functional and easy unit testing for C functions. The module provides all the essential functionality required from a testing framework and all it takes to use is to include a header file in your project and reference it in a C file.

Using MUnit

In order to use MUnit, you first have to import the 'unit.h' header file inside your testing program. The actual test is defined as follows:

```
TEST(first_test_name)
{
/* functionality set up */
ASSERT("Message", condition);
}
```

The `ASSERT` directive will test if the condition (a Boolean condition, e.g. ==, < or >=) is true and output the message if the test fails. If the test runs successfully, no message is printed. Independent of the result of any particular test, test suite keeps executing until all tests are finished.

In order to create a test suite, use the `TEST_SUITE` directive. Example of usage could be as follows.

```
TEST_SUITE(test_suite_name){
ADD_TEST(first_test_name);
ADD_TEST(second_test_name);
}
```

Including tests in a suite is necessary to get those tests to actually run. The `first_test_name` in the `ADD_TEST(first_test_name)` line has to be the same as the name of a test defined earlier. Additionally, the line `TEST_SUITE(test_suite_name)` has to be included in any header file. This header file has to be included in the file where tests are executing.

To run a test suite, use the following template.

```
void c_function{
     INIT_TESTING();
RUN_TEST_SUITE(test_suite_name);
PRINT_DIAG();
}
```

Running the `RUN_TEST_SUITE(test_suite_name);` macro will execute all the tests in the `test_suite_name` test suite. Additionally, the status of test run will be printed in the end. The status will specify the number of asserts passed, the number of asserts failed, the total number of asserts made and the number of tests run.

Example

There is a .zip folder with an example of how to use MUnit called test_suite_demo.zip. In it, test_suites.h is defined as follows:

```
#ifndef TEST_SUITES_H
#define TEST_SUITES_H
#include "unit.h"
TEST_SUITE( led_tests );
#endif /* TEST_SUITES_H */
```

This tells the compiler there is one test suite to run: `led_tests`. Then, the test suite file are defined as follows:

led_tests.c:

```
#include "test_suites.h"
#include "led.h"

TEST( led_on )
{
/* This test should confirm that the function to turn on an led
 * works. Here an led is really just an int for simplicitty.
 */
int testLed = 0; //off
int *ledPointer = &testLed;
TurnLedOn(ledPointer);
ASSERT( "Led did not turn on.", testLed != 0);
}
TEST( led_toggle )
{
/* This should toggle led, but on to off toggle will fail, to
 * show what happens when a test fails
 */

int testLed = 0; //off
int * ledPointer = &testLed;
ToggledLed(ledPointer);
ASSERT( "led failed to toggle (off to on)", testLed != 0 );

ToggleLed(ledPointer);
ASSERT("led failed to toggle (on to off)", testLed != 1);
}

TEST_SUITE( led_tests )
{
ADD_TEST( led_on );
ADD_TEST( led_toggle );
}
```

This creates tests `led_on` and `led_toggle`, and bundles them in test suite `led_tests`.

The functional code so the tests can actually run is as follows:

led.c:

```
#include "led.h"

void TurnLedOn(int *led)
{
*led = 1;
}

void ToggleLed(int *led)
{
if(led == 0)
      *led = 1;
else
      *led = 1; //This is wrong on purpose so the test fails
}
```

led.h:

```
#ifndef LED_H_
#define LED_H_

void TurnLedOn(int *led);
void ToggleLed(int *led);

#endif/*LED_H_*/
```

Finally, the main function, main.c, looks like this:

```
#include "test_suites.h"
int main( void )
{
INIT_TESTING( );
RUN_TEST_SUITE( led_tests );
PRINT_DIAG( );
return 0;
}
```

Now, main.c will run the test suite `led_tests` and will print results to console.

To compile, type in: gcc -Wall led.c led_tests.c main.c -o exec
and then to see the output type in: ./exec