Welcome to midcunit, a lightweight C unit-test framework

midcunit is a very lightweight unit testing framework for C. It is based on the even lighter-weight testing framework called minunit by Jera Design: http://www.jera.com/techinfo/jtns/jtn002.html.

The name "midcunit" is meant to imply that it is a mid-weight C unit testing framework - a little heavier than minunit, but not by much.

Overview

The entire midcunit framework consists of one header file: midcunit.h

It has four function macros, one MD_MAX_MSG_LEN macro and defines and initializes three variables (two int's and a char array of size MD_MAX_MSG_LEN). You will only need to concern yourself with the four macro functions and to keep your error messages smaller than MD_MAX_MSG_LEN - 100 or so (which means you have about 150 bytes by default).

You #include "midcunit.h" into a test file and then define three things:

1. one or more test functions that will call an "md_assert" function
2. a function that passes a function pointer to each of your tests to the midcunit function md_run_test()
3. a main function that calls RUN_TESTS() and passes a function pointer to the method described in step 2 above

Then just compile and run.

Two of the four macro functions are assert functions:

- The first, "md_assert", like assert from <assert.h>, takes only one argument: the expression to evaluate (which evaluates to a C boolean value).
- The second, "md_assertm", like the assert from minunit, takes two arguments: a message to be printed if the test fails and the expression to evaluate.

In either case, if the expression is false, midcunit will print the filename, line number and the assert that failed. md_assertm will also print out the message you provided.

Rationale

In the process of improving my C skills I wanted a unit testing framework that seemed easiest to use: I looked at using regular <assert.h> and then Jera's minunit.

What I like about <assert.h> is that it doesn't require a string message, which can get tedious to type in - just put in your assert test and if it fails it will print out the assert test itself and file and line number, so you can go right to it. But of course, once it fails it stops everything dead in its tracks - you can't run a battery of tests, some of which pass, some of which fail and get a report at the end.

In using minunit, I really liked its lightweight nature and the fact that everything is a macro. However, there were a couple of things I found could be improved for my taste:

1. minunit asserts always require a string message - I'd like that to be optional
2. minunit asserts have no easy way to print out line number, which I find to the most useful thing in quickly jumping to where test failed.
3. when doing multiple tests, minunit stops on the first error and doesn't run the remaining tests - I

want it to stop that particular test, but still run the other remaining test functions

4. the test file you need set up has a lot of boilerplate much of which I thought could be hidden away in the framework itself

So I built midcunit to combine the best of <assert.h> and minunit.c

Feedback on how it could be improved with better functionality or how to better follow C best practices are welcome. I'd love to learn from your knowledge and experience, as I'm still learning C.

Installation

Command: make install

The Makefile provided can be used to install the midcunit.h header file. By default, the Makefile will install the header in /usr/local/include. If you wish to change that, edit the Makefile's HDR_INSTALL_DIR variable.

Quick How-To

midcunit comes with a few example unit tests, the first is barebones, the second depends on another simple C file.

To build them, a Makefile is provided.

Type "make" to build the tests and "make test" to run the tests. "make help" will show you the targets in the Makefile.

Example: Create and run a bare bones unit test file

Step 1: #include midcunit and define a test method that takes no arguments and returns a char *. Have it return 0 at the bottom. Use the md_assert or md_assertm functions to test expressions. If an expression passed to md_assert or md_assertm evaluates to false, then it will prepare a "FAILED assert" message and return it, leaving the current test function where the first assert fails.

```
#include "midcunit.h"
int foo = 7;
static char * test_foo(void) {
 md_assert(foo == foo);
 md_assert(foo == 8);                          /* this will fail */
 md_assertm("foo should equal 7", foo == 7);   /* this line would not execute */
 return 0;                            /* 0 would return only if all tests pass */
}
```

Step 2: Create a function that will run all the tests (I usually call it "all_tests") and have it call the midcunit macro function "md_run_test", which takes a function pointer to the test methods defined in step 1.

```
static void all_tests(void) {
```

```
  md_run_test(test_foo);
  md_run_test(test_bar);
  md_run_test(test_baz);
  md_run_test(test_quux);
}
```

Step 3: Create a main method and call the midcunit macro function RUN_TESTS, which takes a function pointer to the method you defined in step 2.

```
int main() {
  RUN_TESTS(all_tests);
}
```

[Optional additions:]

You could define a setup() and teardown(), as well as an initial_setup() and final_teardown() if you would like. midcunit does not provide these, but it would be simple to add them like this:

```
static void all_tests(void) {
  initial_setup();

  setup();
  md_run_test(test_foo);
  teardown();

  setup();
  md_run_test(test_bar);
  teardown();

  final_teardown();  /* put a final tear down here, *not* in main after RUN_TESTS.
                        RUN_TESTS has a return in it, so nothing after it can
                        be executed  */
}
```

If I put a simple puts() statement in setup/teardown method and made the tests

fail, here is the output I would get:

```
$ ./midcunit_example
ISU: initial setup
SU : setup
test_stack_foo:14: FAILED assert 'foo == 2'
TD : teardown
```

```
SU : setup
test_stack_bar:19: FAILED assert 'bar == foo'
TD : teardown
FTD: final teardown
2 TESTS FAILED
Tests run: 2
```

API Documentation

Since the macro functions are not really functions, they don't have a return type per se, but they all take arguments. I use javadoc notation, since I find that the most appealing form of function documentation.

```
/**
 * If the expression passed in is true, the macro function does nothing. If it
 * is false, it will stop the test function it is in and create a string defining
 * the function, line number and expression that failed.  It will look like:
 *   test_regexsub_no_match:32: FAILED assert 'strcmp(str,subject) != 0'
 *
 * @param exp - the expression to be evaluated which evaluates to a bool (or int).
 *              Note: there is no need to #include <stdbool.h> with midcunit.
 */
md_assert( bool exp );


/**
 * If the expression passed in is true, the macro function does nothing. If it
 * is false, it will stop the test function it is in and create a string defining
 * the function, line number, the error message passed in and expression that
 * failed.
 * It will look like:
 *   test_regexsub_no_match:32: New and orig strings should be different: FAILED \
 *   assert 'strcmp(str,subject) != 0'
 *
 * @param exp - the expression to be evaluated which evaluates to a bool (or int).
 *              Note: there is no need to #include <stdbool.h> with midcunit.
 * @param msg - message to display if the assert fails (is false)
 */
md_assertm( const char* msg, bool exp );
```

```
/**
 * Runs one test function, passed as a function pointer to the macro.
 * Whenever called md_run_test increments a number of tests counter.
 * If the test fails, it also increments an internal "fail counter"
 * and prints out the failure message that is returned by the md_assert
 *  (or md_assertm) function.
 *
 * @param f pointer to a function that takes no arguments and returns char *
 *          (which is what your test methods should do)
 */
md_run_test( char *(*f)(void) );


/**
 * Calls the function passed in as a function pointer to run all unit tests.
 * Once all tests have run, it either prints (stdout) "ALL TESTS PASSED" or
 * prints the number of tests that failed.  In either case, it also shows the
 * number of tests that were run.
 * It returns the number of tests that failed to end the main() method. Thus,
 * if no tests failed, the return value of the test program itself is
 * 0(EXIT_SUCCESS) otherwise it is > 0 (EXIT_FAILURE).
 *
 * @param f pointer to a function that takes no arguments and returns nothing,
 *          which is what your "all_tests" type method should do.
 */
RUN_TESTS( void (*f)(void) );
```

Status and Possible Limitations

Version 0.1 of midcunit should be considered "alpha" software. I have been using it for a few weeks and it is working great for me, but it has only been used on my dev environment which is:

```
os      : Linux 2.6.38-8-generic
dist    : Ubuntu 11.04 (Natty Narwhal)
compiler: gcc (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2
library : glibc 2.13 (ldd (Ubuntu EGLIBC 2.13-0ubuntu13) 2.13)
```

If others want to start testing it, great! Let me know if you find problems and/or how it can be made more portable. One obvious area of concern is the code I copied from the glibc 2.13 assert.h header. I assume the macro __STDC_VERSION__ is defined in the vast majority of environments nowadays, but I don't know where important exceptions might be.

Author and Date

Michael Peterson

25-July-2011

https://github.com/midpeter444/midcunit