---

title: The `mintest` test support library: The little engine that could published: false description: Presentation of the testing support library `mintest` tags:

cover_image: https://direct_url_to_image.jpg

Use a ratio of 100:42 for best results.

published_at: 2024-07-18 15:13 +0000

The testing is an important element to improve the quality of a software. It's an old concept, almost as old as the software development itself. Fortunately, ever since JUnit many testing frameworks and libraries have been proposed that alleviate the chores of writing software tests.

There are many software testing frameworks. Some of them are very complex, and in exchange to their testing power, they can be intrusive to the software being adapted to their use. That's a problem particularly for legacy projects.

But there's minunit by David Siñuela Pastor, aka *siu*, that uses a minimalistic philosophy, with clever and elegant solutions, that makes it easy to insert the testing practice to any software project.

A basic documentation is already present at the repository page, but here follows a more detailed documentation to hopefully explain it a little more to the busy programmers that maybe would like to have more information before trying to use the interesting `minunit` code.

Architecture

All of `minunit` is contained in a single header file, named minunit.h. It's cross-platform and can be used in Microsoft Windows, Linux and many other Unix varieties -- for instance MacOS.

`minunit` lists the CPU and the clock wall timings in high resolution, and since this is OS-specific, preprocessor directives invoke the right OS primitives to time the tests.

The preprocessor is heavily used in `minunit`. That will disgust C++ purists, but there are no cryptic preprocessor tricks and `minunit` code is very easy to read and understand. Since it works so well, usually there's no need to look under the hood to learn how the tests are done.

The `minunit` tests are integrated with the conventional program code, and there's not an external runner program that will run the tests.

Many of the elements of `minunit` are parameterless functions. That imposes the need of global variables, but this limitation can be circumvented by the use of -- let's say -- a "singleton-like" struct, a package of data that will contain the program parameters, avoiding the global namespace pollution and avoiding many of the problems associated with global variables. See the example code below.

Units

Two elements of `minunit` that are present in most testing frameworks and libraries are:

- **Fixture** a function that will create a sane environment for the tests;
- **Teardown** a function that will release the data allocated by the fixture.

Both elements are used as function pointers and any of them can be set to the `NULL` pointer.

The basic units of `minunit` are

1. *test suite* is the largest unit in `minunit`. It allows the definition of the fixture and teardown function pointers, and can generate a report of the passed and failure tests under it. It will **not** release the resources allocated by the fixture, by calling the teardown function, if one was provided -- this is a *test* responsability.

   Any number of tests can be associated with a suite, that's defined using the macro `MU_TEST_SUITE()` that annotates a test suite function.

   Inside the test suite function, the macro `MU_SUITE_CONFIGURE()` will set the fixture and teardown function pointers that will be used by each individual test -- see below.

   In the calling program -- possibly `main()` -- the test suite will be run using -- you got it -- `MU_RUN_SUITE()`, that will also set to `NULL` the fixture and teardown function pointers in the end of its execution.

   After the execution of the tests, the macro function named `MU_REPORT()` will generate a report to inform the number of tests executed and the ones that failed, as well as the processing time of the tests, and the elapsed time for the whole suite;

2. *test* is one active unit of `minunit`: it's a function that will call the fixture to set a sane environment, run all of its test assertions and then call the test teardown, to release the resources.

   A test is defined by the annotation `MU_TEST` and run by... `MU_RUN_TEST()`.

3. *assertion* is the atomic element of `minunit` that will assert one test.

   Find a list of assertions below.

- `mu_check(test)` will run the test expression and will generate a fail message that shows the expression and tell that it failed;
- `mu_assert(test, message)` will run the test expression and show the string in `message`;
- `mu_fail(message)` will simply report a failure and show the string in `message`. It's probably useful in the case when a complex situation is detected in the code and it's just the case of reporting a failure;
- `mu_assert_int_eq(expected, result)` will compare the `int` expressions `expected` and `result`, and fail if they're not equal;
- `mu_assert_double_eq(expected, result)` will compare the `double` expressions `expected` and `result`, and fail if they're not equal;

- `mu_assert_string_eq(expected, result)` will compare the string expressions `expected` and `result`, and fail if they're not equal.

Assertions containing other C data types can be constructed using carefully the appropriate type casts.

A sample program

Here goes the file `minunit_example3.c`, derived from the example code available in the minunit repository.

```
#include <stdlib.h> // calloc(), free()
#include "minunit.h"

struct two_vecs {
    size_t length;
    int* naturals;
    int* evens;
} param;

void test_setup(void) {
    size_t ind;
    param.length = 100;
    param.naturals = (int*)calloc(sizeof(int), param.length);
    param.evens    = (int*)calloc(sizeof(int), param.length);

    for (ind = 0; ind < param.length; ind++) {
        param.naturals[ind] = ind;
        param.evens[ind] = 2*ind;
    }
}

void test_teardown(void) {
    free(param.naturals);
    free(param.evens);
    param.length = 0;
}

MU_TEST(test_assert_pos_eq) {
    mu_assert_int_eq(param.evens[0], param.naturals[0]);
    mu_assert_int_eq(param.evens[1], param.naturals[2]);
    mu_assert_int_eq(param.evens[5], param.naturals[10]);
}

MU_TEST(test_assert_one_eq_fail) {
    mu_assert_int_eq(param.evens[1], param.naturals[1]);
}

MU_TEST_SUITE(test_suite) {
    MU_SUITE_CONFIGURE(&test_setup, &test_teardown);

    MU_RUN_TEST(test_assert_pos_eq);
    MU_RUN_TEST(test_assert_one_eq_fail);
}
```

```
int main() {
    MU_RUN_SUITE(test_suite);
    MU_REPORT();
    return EXIT_SUCCESS;
}
```

In `test_setup()`, two `int` vectors are allocated. The vector `naturals` is filled with integer numbers from 0 to 99. The vector `evens` contain even numbers 0, 2, 4, ... 198.

The test `test_assert_pos_eq()` checks the correct premise that positions 0, 1, and 5 in `evens` are equal to positions 0, 2, and 10 in `naturals`.

The test `test_assert_one_eq_fail()` asserts the wrong statement that the position 1 in `evens` contains the same value as the position 1 in `naturals`.

As expected, all the assertions in the 1st test pass, while the assertion in the 2nd test fail, as seen in the output below.

```
...F
test_assert_one_eq_fail failed:
        minunit_example3.c:35: 2 expected but was 1


2 tests, 4 assertions, 1 failures

Finished in 0.00014870 seconds (real) 0.00014400 seconds (proc)
```

In this listing the 3 dots indicate the 1st set of tests that passed, while the `F` indicates the last test that failed. The error message indicates the line where the failing code was, and the two values are shown.