The tinytest unit testing framework

Tinytest is a small set of C files that you can use to write tests for your C program. It's designed to make writing tests easy and convenient. It supports the features that I find useful in a unit test framework--like running tests in subprocesses so that they can't interfere with global state, or like running or suppressing a subset of the tests.

Tinytest is designed to be so small that I don't mind including its source along with all the C I write.

Tinytest is *not* designed to be a replacement for heavier-weight unit-test frameworks like CUnit. It doesn't generate output in XML, HTML, or JSON. It doesn't include a mocking framework.

This document describes how to use the basic features of tinytest. It will not tell you much about how to design unit tests, smoke tests, integration tests, system tests, or whatever else you might want to use tinytest for. I'm assuming that you already have some idea of what kind of test code you want to write, and you just need a minimal framework that will get out of your way and help you write it.

Your first test

Here's a simple example to start with. It's a standalone program that uses tinytest to check whether strdup() behaves correctly.

```
/* You can save this as "demo.c", and then compile it with (eg)
 *      cc -Wall -g -O2 demo.c tinytest.c -o demo
 */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "tinytest.h"
#include "tinytest_macros.h"

static void test_strdup(void *dummy_arg)
{
    const char *str = "Hello world";
    char *str2 = NULL;

    (void) dummy_arg; /* we're not using this argument. */
    str2 = strdup(str);

    tt_assert(str2);           /* Fail if str2 is NULL */
    tt_str_op(str, ==, str2);  /* Fail if the strings are not equal */

  end:
    if (str2)
        free(str2);
}

struct testcase_t string_tests[] = {
    /* This test is named 'strdup'. It's implemented by the test_strdup
     * function, it has no flags, and no setup/teardown code. */
    { "strdup", test_strdup, 0, NULL, NULL },
    END_OF_TESTCASES
};

struct testgroup_t test_groups[] = {
    /* The tests in 'string_tests' all get prefixed with 'string/' */
    { "string/", string_tests },
    END_OF_GROUPS
};

int main(int argc, const char **argv)
{
    return tinytest_main(argc, argv, test_groups);
}
```

Once you have compiled the program, you can run it with no arguments to invoke all its tests:

```
$ ./demo
string/strdup: OK
1 tests ok.  (0 skipped)
```

You can also control the verbosity of the output:

```
$ ./demo --verbose
string/strdup:
    OK demo.c:18: assert(str2)
     OK demo.c:19: assert(str == str2): <Hello world> vs <Hello world>
1 tests ok.  (0 skipped)
[510]$ ./demo --terse
.
[511]$ ./demo --quiet && echo "OK" || echo "FAIL"
OK
```

Most of the code above is boilerplate. The "main" function just delegates to tinytest_main(), and passes it an array of testgroup_t structures. Each test_group_t points to an array of testcase_t structures -- and each of *those* has (at minimum) a test name, and a function that implements the test.

Note that the test function is declared as 'void test_strdup(void *dummy)'. All test functions must be declared as returning void and taking a single void pointer as an argument. (More about the use of this argument later.)

The body of the test function calls two tinytest check macros: tt_assert() and tt_str_op(). These functions check whether a given condition holds: tt_assert(str2) checks that 'str2' is true; and tt_str_op(str,==,str2) checks that str1 and str2 are equal according to strcmp(). If one of these conditions doesn't hold, the check macro will print a useful error message, mark the test as having failed, and go to the "end:" label to exit the function.

Note that the "end:" label is mandatory, and all the cleanup code in the function goes after the "end:" label. If one of the check macros fails, then it will invoke "goto end;" after it fails, so that the cleanup code gets executed before the function returns.

More tools for writing test functions

The 'tinytest_macros.h' header declares a bunch of macros to help you write test functions.

These macros cause unconditional failure of the current test, and go to the 'end:' label to exit the function:

```
tt_abort();
tt_abort_printf((format, ...));
tt_abort_msg(msg);
tt_abort_perror(op);
```

They differ in how they report an error. The 'tt_abort()' macro will just report that the test failed on a given line number; the 'tt_printf()' macro will use a printf-style function to format its inputs; the 'tt_abort_msg()' macro prints a string, and the tt_abort_perror() macro will report the current value of errno along with a string "op" describing what it was that failed.

These macros check a boolean condition:

```
tt_assert(condition);
tt_assert_msg(condition, msg);
```

Each of these macros checks whether a condition is true, and causes the test to fail if it is not true. On failure, they report the message (if provided), or the condition that failed.

These macros perform binary comparisons, and print the values of their arguments on failure:

```
tt_int_op(a, op, b)
tt_uint_op(a, op, b)
tt_ptr_op(a, op, b)
tt_str_op(a, op, b)
tt_mem_op(a, op, b, len)
```

In each case, the "op" argument should be one of "==", "<=", "<", "!=", ">", or ">=". The test succeeds if "a op b" is true, where the rules of the comparison are determined by the type: tt_int_op performs a comparison after casting to long; tt_uint_op compares after casting to unsigned long; and tt_ptr_op checks after casting to void *. The tt_str_op macro uses strcmp() to compare its arguments, and tt_mem_op uses memcmp() with the given length value.

The tt_*_op() macros behave the same as would a corresponding tt_assert() macro, except that they produce more useful output on failure. A failed call to "tt_assert(x == 3);" will just say "FAIL: assert(x == 3)". But a failed call to tt_int_op(x == 3);" will display the actual run-time value for x, which will often help in debugging failed tests.

The following macros behave the same as the above macros, except that they do not "goto end;" on failure. The tt_fail macros correspond to tt_abort, the tt_want macros correspond to tt_assert, and the tt_want_*op macros correspond to the tt*_op macros:

```
tt_fail();
tt_fail_printf((format, ...));
tt_fail_msg(msg);
tt_fail_perror(op);

tt_want(condition)
tt_want_msg(condition, msg);

tt_want_int_op(a, op, b)
tt_want_uint_op(a, op, b)
tt_want_ptr_op(a, op, b)
tt_want_str_op(a, op, b)
tt_want_mem_op(a, op, b, len)
```

Tips for correct cleanup blocks

Remember that most of the check macros can transfer execution to your "end:" label, so there may be more paths through your function than you first realize. To ensure that your cleanup blocks are correct, I recommend that you check-and-release every resource that could be allocated by your function.

So here's a good pattern:

```
{
    char *x = NULL;
    tt_int_op(some_func(), ==, 0);

    x = malloc(100);
    tt_assert(x);

          /* more tests go here... */
 end:
    if (x) free(x);
}
```

And here's how not to do it:

```
{
    char *x;
    tt_int_op(some_func(), ==, 0);

    x = malloc(100);
    tt_assert(x);

  end:
    /* Whoops! x might be uninitialized! */
    if (x) free(x);
}
```

And here's another way to mess up:

```
{
    tt_int_op(some_func(), ==, 0);

    {
        char *x;
        x = malloc(100);
        tt_assert(x);
        /* ... more tests that use x ... */
        free(x);
    }

  end:
    /* uh-oh! we might wind up here, unable to free x! */
}
```

You can also program your way around these problems through appropriate use of control flow, but it's generally better to just use the recommended pattern above.

Running tests in a separate process

Sometimes you might need to run tests in isolation so that they can have no effect on global state. (It sure gets confusing when tests interfere with each other.) To do this, you can set the TT_FORK flag on any test: Doing this will make the tinytest use fork() [on *nix] or CreateProcess [on Windows] to run that test in a sub-process.

For example, if you wanted the test_strdup() test from the first thing above to run in a subprocess, you would declare string_tests[] as:

```
struct testcase_t string_tests[] = {
    { "strdup", test_strdup, TT_FORK, NULL, NULL },
    END_OF_TESTCASES
};
```

Setup and tear-down functions

Sometimes you want to share setup and cleanup code for a large number of tests. To do this, write a function that creates an environment and another that tears it down, and wrap them in a testcase_setup_t structure:

```
struct env {
    const char *test_string;
    char buf[1024];
};

void *create_test_environment(const struct testcase_t *testcase)
{
    struct env *env = calloc(sizeof(*env), 1);

    if (! env)
        return NULL;  /* Return NULL to indicate failure */

    /* In this case, we're assuming setup_data is a string, and we
       can store it into the environment unprocessed. But really,
       setup_data can be any void *, so long as the setup function knows
       how to use it. */
     env->test_string = testcase->setup_data;

     /* If there were other fields of env, we would initialize them here.
      */
     return env;
}

void *cleanup_test_environment(struct testcase_t *, void *env_)
{
    struct env *env = env_;

    /* on error, we should return 0 from this function */
    return 1;
}

struct testcase_setup_t env_setup = {
     setup_test_environment,
     cleanup_test_environment
};
```

And now we can write tests that use these environments, as in:

```
static void testcase(void *arg)
{
    struct env *env = arg;
    /* test here can use fields in env */
  done:
    ;
}

struct testcase_t some_tests[] = {
    { "test_name", testcase, 0, &env_setup, (void*)"a string" },
    END_OF_TESTCASES
};
```

Skipping tests

Not every test needs to be on by default, and not every test needs to be on for every platform. If you want to mark a test as not-to-be-run, you can set the TT_SKIP flag on it. If you want to mark it as off-by-default (perhaps because it is slow or buggy), you can set the TT_OFF_BY_DEFAULT flag.

```
#ifdef _WIN32
#define SKIP_ON_WINDOWS TT_SKIP
#else
#define SKIP_ON_WINDOWS 0
#endif

struct testcase_t some_tests[] = {
    { "not_on_windows", test_skipped1, SKIP_ON_WINDOWS, NULL, NULL },
    { "not_by_default", test_skipped2, TT_OFF_BY_DEFAULT, NULL, NULL },
    END_OF_TESTCASES
};
```

If you want to decide whether to skip a test at runtime, you can do so by calling tt_skip():

```
if (wombat_init() < 0) {
   /* Couldn't initialize the wombat subsystem. I guess we aren't
      running this test. */
   tt_skip();
}
```

As you might expect, tt_skip() macro calls "goto end" to exit the test function.

Inside test functions: reporting information

All output from a test case is generated using the TT_DECLARE macro. If you override TT_DECLARE before including tinytest_macros.h, you can replace its behavior with something else. TT_DECLARE takes two arguments: a prefix that , and a set of printf arguments in parenthesis.

Example use:

```
if (sec_elapsed > 120)
    TT_DECLARE("WARN",
      ("The test took %d seconds to run!", sec_elapsed));
```

If tinytest is running verbosely, the TT_BLATHER() macro takes a set of printf arguments and writes them to stdout using TT_DECLARE:

```
  x = get_wombat_count();
  TT_BLATHER(("There are %d wombats", x));
```

Note the extra set of parenthesis in both cases; they are mandatory.

Managing many tests

Once you start to have a lot of tests, you can put them in separate modules. You don't need to export the functions from the modules; instead, just export the arrays of testcases. So in file1.c, you might say:

```
 struct testcase_t string_tests[] = {
    { "strdup", test_strdup, 0, NULL, NULL },
    { "strstr", test_strstr, TT_FORK, NULL, NULL },
    { "reverse", test_reverse, 0, NULL, NULL },
    END_OF_TESTCASES
};
```

And in file2.c you might say:

```
struct testcase_t portal_tests[] = {
    { "orange", test_portal, 0, &portal_setup, (char*)"orange" },
    { "blue", test_portal, 0, &portal_setup, (char*)"blue" },
    { "longfall", test_longfall, TT_FORK, NULL, NULL },
    END_OF_TESTCASES
};
```

And in test_main.c you could say:

```
extern testcase_t string_tests[];
extern testcase_t portal_tests[];
struct testgroup_t test_groups[] = {
     { "string/", string_tests },
     { "portal/", portal_tests },
     END_OF_GROUPS
};
```

When you run these tests, the names of the string tests will be prefixed with "string/", and the names of the portal tests will be prefixed with "portal/".

Invoking tinytest

You can control the verbosity of tinytest from the command line. By default, tinytest prints the name of every test as it runs, and tells you about failed check macros and tests.

If you pass the "--terse" command line option, tinytest will print a dot for each passing test, instead of its name. If you pass the "--quiet" option, tinytest will only tell you about failed tests. And if you pass "--verbose" flag, tinytest will print a line for every check macro, including ones that are successful.

You can list all the test cases, instead of running them, by passing the "--list-tests" flag.

You can control which tests will be run by listing them on the command line. The string ".." is a wildcard that matches at the end of test names. So for examine, in the section above, you could run only the portal tests by passing "portal/.." on the command line.

To disable tests, prefix their names or a wildcard with the : character. So to turn off the longfall and string tests, you could pass ":portal/longfall :string/..".

To run an off-by-default test, prefix its name (or a wildcard pattern matching its name) with the + character. (Thus, you can run all tests, including off-by-default tests, by passing "+.." on the command line.

If you need to run a test in a debugger, and the debugger doesn't follow fork() or CreateProcess() very well, you can turn off the pre-test forking with the "--no-fork" option. Doing this when you're running more than a single test can give you unexpected results, though: after all, you're turning off the test isolation.

Legal boilerplate

There is no warranty on tinytest or on any code in this document.

Tinytest itself is distributed under the 3-clause BSD license.

Absolutely everybody has permission to copy any of the code *in this document* freely into your own programs, modify it however you want, and do anything else with it that copyright laws would otherwise restrict. You do not need to give me credit or reproduce any kind of notice.

But if it would make you more comfortable to have a a formal license, you may also use all of this code under the terms of the "3-clause BSD license" quoted here: