mutest - A simple micro unit testing framework for C

Author: Leandro Lucarella
Contact: llucax@gmail.com

Version: 1.0
Date: 2008-12-25

Abstract

mutest is a micro unit testing framework for C (with some C++ support). It's mostly an idea (it even comes with 2 implementations of the idea!) with the goal of being easy to use (just write your test cases grouped in test suites and you're set) and so small and simple that you don't mind to copy the files to your project and just use it (i.e., no dependencies).

The idea is simple: a source file is a test suite, a function is a test case (special functions can be used for test suite initialization and termination), which can can have several checks. Checks comes in 2 flavors, one that only prints an error, and one that terminates the current test case too. A (normally) automated test program run all the test suites and print some stats. It fails (returns non-zero) if any test suite fails.

Contents

1. Installation

Download the latest distribution tarball and uncompress it.

You can also download any release from the releases directory or get it using Git directly from the Git repository.

You can get this manual too, or a PDF version of it.

To actually install *mutest* run:

```
$ make install
```

Default installation path is `/usr/local` (because of that, you'll probably need superuser privileges to install to the default location). You can override that by passing the `prefix` make variable, for example:

```
$ make prefix=/opt/mutest install
```

If you want to install just the docs, you can do:

```
$ make install-doc
```

Or even `install-readme`, `install-html` or `install-pdf` if you are too picky.

If you want to install just one particular implementation, to can use the `install-c` and `install-py` targets.

2. Quick Sample

You can find some samples in the sample directory.

This is an example taken from there. A simple *module* called factorial.c with its corresponding test suite (factorial_test.c).

You can see some C++ support in the exception_test.cpp test suite.

2.1. factorial.c

```
/*
 * This file is part of mutest, a simple micro unit testing frame-
work for C.
 *
 * mutest was written by Leandro Lucarella <llucax@gmail.com> and is re-
leased
 * under the BOLA license, please see the LICENSE file or visit:
 * http://blitiri.com.ar/p/bola/
 *
 * This is an example module that calculates a factorial.
 *
 * Please, read the README file for more details.
 */

unsigned factorial(unsigned x) {
if (x <= 1)
return 1;
return x * factorial(x-1);
}
```

2.2. factorial_test.c

```
/*
 * This file is part of mutest, a simple micro unit testing frame-
work for C.
 *
 * mutest was written by Leandro Lucarella <llucax@gmail.com> and is re-
leased
 * under the BOLA license, please see the LICENSE file or visit:
 * http://blitiri.com.ar/p/bola/
 *
 * This is the factorial module test suite. Each (public) function starting
 * with mu_test will be picked up by mkmutest as a test case.
 *
 * Please, read the README file for more details.
 */

#include "factorial.h"

#include "../mutest.h"

void mu_test_factorial_zero() {
unsigned x = factorial(0);
mu_check(x == 1);
}

void mu_test_factorial_one() {
unsigned x = factorial(1);
/* this test is wrong on purpose, to see how it fails */
mu_check(x == 2);
```

```
}

void mu_test_factorial_positive() {
unsigned x = factorial(2);
/* this test is wrong on purpose, to see how it fails */
mu_check(x == 3);

x = factorial(3);
/* we don't want to continue if this fails, because the next result
 * depends on this one. This one will succeed. */
mu_ensure(x == 6);

x = factorial(x);
mu_check(x == 720);

x = factorial(4);
mu_ensure(x == 6); /* same as before, but this one will fail. */

x = factorial(x-15); /* and this will never be executed */
mu_check(x == 362881); /* but if excecuted, will fail */
}
```

2.3. exception_test.cpp

```
/*
 * This file is part of mutest, a simple micro unit testing frame-
work for C.
 *
 * mutest was written by Leandro Lucarella <llucax@gmail.com> and is re-
leased
 * under the BOLA license, please see the LICENSE file or visit:
 * http://blitiri.com.ar/p/bola/
 *
 * This is a C++ module test suite. It shows how to use checks involving
 * exceptions.
 *
 * Please, read the README file for more details.
 */

#include <stdexcept> // std::out_of_range
#include <vector> // std::vector

#include "../mutest.h"

extern "C" {

void mu_test_exceptions() {
std::vector<int> v(1);
// ok
mu_check(v.at(0) == 0);
// throws! This fails
```

```
mu_check(v.at(1) == 0);
// ok, we expect the exception to be thrown, and it does
mu_echeck(std::out_of_range, v.at(1));
// fails! We expect this to throw, but it doesn't
mu_echeck(std::out_of_range, v.at(0));
// fails again, but this time the show is over (note the "ensure")
mu_eensure(std::out_of_range, v.at(0));
// this will never be executed (it should fail if it is)
mu_check(v.empty());
}

} // extern "C"
```

3. Concepts

mutest is about 4 simple concepts: test program, test suite, test case and checks. Well, to be honest you probably will need test suite initialization and termination too =)

3.1. Test Program

A **test program** is the higher level unit of *mutest*. The test program is the one in charge of running all your tests. Probably one of the more important features of *mutest* is that you are not supposed to bother about the test program. So, different implementations have different ways to tackle this. Some need more or less interactions from your part, and each have their pros and cons.

But this is all you need to know for now, for more details see how the test program is implemented by your implementation of choice.

3.2. Test Suite

A **test suite** is the higher level unit of *mutest* that you should care about =). Is not much more than a way to group test cases. Code-wise, a test suite is a C (or C++) module (or compilation unit). Not clear enough? A unit test is an object file (could be a shared object depending on the implementation). This module should have one or more test cases and it could have any number (including zero) of initialization and termination functions.

A test suite, is inspected by the test program for test cases and initialization and termination functions, and run them.

A test suite fail if one or more test cases fail, and it's skipped if one or more initialization functions fail (or, depending on the implementation, if the test suite can't be loaded at all).

3.3. Test Case

A **test case** is just a plain function with a special signature and name. A test case function name must start with `mu_test`, and take no arguments and return nothing. For example:

```
void mu_test_something(void);
```

A test case (probably) only make sense if it has checks. A test case succeed only if all its checks succeed too.

Test are executed in an implementation-dependant order, but usually the default order is alphabetical.

3.4. Checks

Checks are assertions that a test case must pass (a boolean expression that must evaluate to *true*). There are 2 big flavors of checks: **check** and **ensure**. **check** just print an error (and *mark* the test case as failed) and **ensure** halt the test case execution, jumping to the next one.

For better C++ support there are check macros that assert that a specified exception is thrown (instead of check for a boolean expression to evaluate to *true*).

You can take a look at the reference to see the different flavors of check macros in more detail.

3.5. Initialization

Sometimes you need to setup some environment shared between all the test cases in a test suite. You can use **initialization functions** for this.

An initialization function, like a test case, is a plain C function with a special name and signature. The name must start with `mu_init` and it must take no arguments, and return an *error code* (`0` being success). For example:

```
int mu_init_something(void);
```

All initialization functions are executed before any test case, in an implementation-dependant order, and if one of them fail (returns non-zero), the whole test suite is skipped immediately.

3.6. Termination

Termination functions are just like initialization functions, but they're executed **after** the test cases, their names start with `mu_term` and they return nothing. For example:

```
void mu_term_something(void);
```

4. C++ Support

You can use *mutest* with C++, the only care you must take is that, because of C++ name mangling (and *mutest* relaying on function names), you must declare your test cases and initialization and termination functions as `extern "C"` (see exception_test.cpp for an example).

Checks become *exception-safe* when using *mutest* with a C++ compiler, and 2 extra checks designed for C++ get defined (mu_echeck() and mu_eensure()). They assert that an expression throws a particular exception.

5. Implementations

There are 2 big groups of possible implementations that I can think of: *static* and *dynamic*.

mutest comes with one implementation of each group.

5.1. Static Implementations

Static implementations can be only written in C/C++ (or other language that is link-compatible with C, like the D Programming Language, but since one of the main goals of *mutest* is avoid unnecessary dependencies, you probably don't want to depend on an extra language/compiler to run your tests =).

The main advantage is better debugging support, because you can run the test program in a standard debugger and see what happens with test cases very naturally.

The main disadvantage is, the test suites must be figured out in *compile-time*, usually using some kind of code generation (if you want to avoid writing repetitive code yourself). There's also a limitation in the test case, initialization and termination functions names: they should be unique for all the test program.

5.1.1. C implementation

mutest comes with a C static implementation. Only 3 files are needed: `mutest.c` (the *user-independent* part of the test program), `mkmutest` (a bash script for generating the *user-dependent* part of the test program) and `mutest.h` (the header file that test suites should include).

You can copy this 3 files to your project or install them at system-level and use them globally.

The procedure is simple, You should compile you test suites, `mutest.c` and the generated output of `mkmutest` as object files and link them together.

For example:

```
$ cc -c -o mutest.o mutest.c
$ cc -c -o test1.o test1.c
$ cc -c -o test2.o test2.c
$ mkmutest mutest.h test1.o test2.o | cc -xc -c -o runmutest.o -
$ cc -o testprg mutest.o test1.o test2.o runmutest.o
```

Then you can run the test program invoking it with no arguments:

```
$ ./testprg
```

5.1.1.1. `mkmutest` Invocation This small script take 1 mandatory positional argument: the path to the `mutest.h` file. All remaining positional arguments should be object files representing test suites.

5.1.1.2. Test Program Invocation The test program can be invoked without arguments, but can take some extra options:

`-v` Be verbose. This is accumulative, when you add extra `-v` you will get extra verbosity.

By default, you just get failed checks printed. If you use a single `-v`, a summary of failed/passed test suites, test cases and checks will be printed. If an extra `-v` is used, you'll see the current test suite being executed. Another `-v` and you'll get the current test case, and another one, and you'll get each check.

5.1.1.3. Dependencies Even when dependencies are kept minimal, there always be a few ;)

To use this implementation you just need:

- A C compiler (you already needed that, so...)
- The `nm` program (from GNU Binutils, included in virtually any *NIX)
- The GNU Bash shell interpreter (also included in virtually any *NIX)

5.2. Dynamic Implementations

Dynamic implementations, on the other hand, can be written in any language that can access to shared objects. The idea is to inspect a shared object for test suites and run them, without requiring any information about test suites at compile time.

There are several advantages in this kind of implementations. The dynamic nature let you completely separate the test program from the user-written test suites and you can choose at *run-time* what test suites to execute by just selecting the correct shared objects. Also, test case, initialization and termination functions names only have to be unique in the scope of the test suites, because test suites are completely isolated in separate shared objects.

But everything comes at a price, and the higher price to pay is *debuggability*. It's a little harder to plug a debugger to a shared object.

5.2.1. Python implementation

This implementation is much simpler and elegant than the C implementation. Only 2 files are needed: `mutest` (test program written in Python using ctypes module to access the shared object symbols) and `mutest.h` (the header file that test suites should include).

Since both implementations provided by *mutest* share the same `mutest.h`, you should define the `MUTEST_PY` macro when compiling the test suites if you will run them using this implementation.

As with the C implementation, you can copy this 2 files to your project or install them at system-level and use them globally.

The procedure is even simpler than the C implementation: compile and link you test suites as shared objects and then run the `mutest` program passing the shared objects as arguments. For example:

```
$ cc -c -fPIC -DMUTEST_PY -o test1.o test1.c
$ cc -shared -o test1.so test1.o
$ cc -c -fPIC -DMUTEST_PY -o test2.o test2.c
$ cc -shared -o test2.so test2.o
$ mutest test1.so test2.so
```

That's it.

5.2.1.1. `mutest` Invocation

`mutest` program takes test suites shared objects to run as positional arguments. It accepts the same options as the C implementation's test program and some extra options are accepted too:

`--verbose` Alias for `-v`.

`-q, --quiet` Be quiet (no output is shown at all).

`-s, --search` Search for test suites (*.so) in the current directory and add them to the list of test suites to run.

`-h, --help` Show a help message and exit.

5.2.1.2. Dependencies

As with the C implementation, some minor dependencies are needed:

- Python (2.5 or later)
- The `nm` program (from GNU Binutils, included in virtually any *NIX)

You will need a C compiler for building the test suites too, but technically is not needed by *mutest* itself ;)

6. Reference

6.1. `mu_check()`

Synopsis `mu_check(expression)`

Description Check that the `expression` evaluates to *true*. Continue with the test case if fail.

Availability Always

Example

```
void mu_test(void)
{
    mu_check(5 == 4); /* fail */
    mu_check(5 == 5); /* excecuted, pass */
}
```

6.2. `mu_ensure()`

Synopsis `mu_ensure(expression)`

Description Check that the `expression` evaluates to *true*. Interrupt the test case if fail.

Availability Always

Example

```
void mu_test(void)
{
    mu_ensure(5 == 4); /* fail */
    mu_check(5 == 5); /* not excecuted */
}
```

6.3. `mu_echeck()`

Synopsis `mu_echeck(class, expression)`

Description Check that the `expression` throws a specific exception `class` (or subclass). Continue with the test case if fail.

Availability C++ only

Example

```
#include <stdexcept>

extern "C"
{
    void mu_test(void)
    {
        mu_echeck(std::exception, true); /* fail */
        mu_echeck(std::exception,
                throw std::runtime_error("!")); /* excecuted, pass */
    }
}
```

6.4. `mu_eensure()`

Synopsis `mu_eensure(class, expression)`

Description Check that the `expression` throws a specific exception `class` (or subclass). Interrupt the test case if fail.

Availability C++ only

Example

```
#include <stdexcept>

extern "C"
{
    void mu_test(void)
    {
```

```
        mu_eensure(std::exception, true); /* fail */
        mu_echeck(std::exception,
                throw std::runtime_error("!")); /* not excecuted */
    }
}
```

7. About

This manual was written using reStructuredText.