CSC161 2010F *Imperative Problem Solving*

Assignment 6: A Testing Library

Assigned: Friday, 15 October 2010
Due: 11:00 p.m., Wednesday, 27 October 2010

Summary: In this assignment, you will build a library of macros that you can use to test the procedures that you write.

Purposes: To give you further experience with testing. To give you experience writing macros. To give you practice with various aspects of C programming.

Expected Time: Two to three hours.

Collaboration: I encourage you to work in groups of two or three students. However, you may work on your own or in groups of up to size four. You may discuss the assignment with anyone you wish, provided you clearly document such discussions.

Submitting: Email me a tarball of your important files (your `.c` files, your `.h` files, your `Makefile`, and anything else you deem appropriate).

Warning: So that this assignment is a learning experience for everyone, I may spend class time publicly critiquing your work.

Testing

As you no doubt have noted by this time, our strategy for testing is fairly standardized.

- We set up an integer variable, `errors`, that we initialize to 0.
- We test a particular call to a function by comparing the result to an expected value.
  - If the two values are equal, we do nothing.
  - If the two values are not equal, we print an appropriate error message and increment `errors`.
- At the end of the program, we check whether `errors` is still 0. If so, we exit successfully. If not, we exit unsucessfully.

Since that code is so similar, it makes sense to set up a library to help us handle it. However, there's one difficulty: In order to print an appropriate error message, it's useful to know the text of the call. (It may also be useful to know the result, but that's an issue that we can also deal with in the debugger.)

We came up with the following macro for that approach.

```
#define TEST_EQUAL(EXP,RESULT) if ((EXP) != RESULT) { ++errors; printf ("Did not get expected result for %s.\n", #EXP); }
```

For example, we might write

```
TEST_EQUAL (square (5), 25)
```

Now it's time to incorporate the `TEST_EQUAL` macro into something we can use throughout our programming.

Assignment

Part 1: A Header File

a. Create a header file, `test.h`, that declares `errors` and the macro above. (Or a corrected version of that macro.) You will also need a macro for the end of `main`. Call that macro `TESTS_FINISHED`.

b. Verify to your satisfaction that the header file works appropriately.

Note that you should make sure to test some of the more complex ways we now build C programs. For example, what happens if we have three `.c` files, one of which defines a function `testA()`, that does some tests, the second of which defines `testB()`, which does some more tests, and the third of which defines `main()`, which calls `testA()` and `testB()`, and all three of the the files include `test.h`?

Hint: You may have to read about modifiers for variables to learn how to handle situations like this.

Part 2: A Better Header File

Update the definition of `TEST_EQUAL` so that it prints the line number and file of the error. (You'll need to look at the manual for the GNU C Preprocessor to figure out how to access that information.)

Part 3: Checking for Primes

Consider a function, `is_prime (int n)`, that determines whether or note `n` is prime.

a. Write a header file, `primes.h`, that declares `is_prime`.

b. Using `test.h` to help, write a program that experimentally verifies whether or not `is_prime` works correctly.

c. Write a code file, `primes.c`, that defines `is_prime`.

If you've written good enough tests, you should be confident that your implementation of `is_prime` is correct.

Part 4: Fibonacci Numbers

As you may recall, the Fibonacci sequence is defined as follows:

- $F_0 = 0$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for all $i > 1$.

a. Write a header file, `fibonacci.h`, that declares a function, `int fib (int n)`, that computes the nth Fibonacci number.

b. Using `test.h` to help, write a program that experimentally verifies whether or not `fib` works correctly.

c. Write a code file, `fibonacci.c`, that defines `fib`.

If you've written good enough tests, you should be confident that your implementation of `fib` is correct.

Part 5: Is it a Fibonacci number?

a. To `fibonacci.h` add a declaration for a function, `is_fib (int n)`, that determines whether or not `n` is an element of the standard Fibonacci sequence.

b. Using `test.h` to help, write a program that experimentally verifies whether or not `is_fib` works correctly.

c. Add a definition of `is_fib` to `fibonacci.c`.

If you've written good enough tests, you should be confident that your implementation of `is_fib` is correct.
