CSC161 2010F *Imperative Problem Solving*

Laboratory: Program Correctness and `assert`

Summary: In this lab, you will explore the use of C's `assert` macro.

Prerequisites: Familiarity with functions, separate compilation, and arrays.

Preparation

a. Create a directory for this lab.

b. In that directory, create our standard Makefile.

c. Review the man page for `assert`.

Exercises

Exercise 1: Quotient

Consider the following program, named `quotient.c`, which is intended to take the quotient of two values entered from the command line.

```
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char *argv[])
{
  int x = atoi (argv[1]);
  int y = atoi (argv[2]);

  int result = x / y;
```

```
  printf ("%d/%d = %d.\n", x, y, result);

  return EXIT_SUCCESS;
} // main
```

a. Make a copy of the program and compile it.

b. Verify that the program works as expected when given appropriate inputs, such as 4 and 2.

c. What do you expect the program to do when the second parameter is 0?

d. Check your answer experimentally.

Exercise 2: Adding an Assertion

a. Add a call to `assert` that ensures that the second parameter is nonzero. Then recompile.

b. What do you expect your program to do when the program is given appropriate inputs, such as 128 and 5?

c. Check your answer experimentally.

d. What do you expect your program to do when the second parameter is 0?

e. Check your answer experimentally.

Exercise 3: Cancelling Assertions

The man page for `assert` indicates that we can turn assertion checking off by adding the `-DNDEBUG` flag.

a. Arrange to compile your program with that flag set.

- Add `-DNDEBUG` to the `CFLAGS` entry in the Makefile.
- Resave your `quotient.c` so that make will understand that it has to recompile.
- Run make.

b. What do you expect your program to do when the second parameter is 0?

c. Check your answer experimentally.

d. Remove the `-DNDEBUG` flag.

Exercise 4: Additional Assertions

You program should fail to work correctly if the number of parameters is not 2.

a. Add an `assert` statement that validates the number of parameters.

b. Verify that your added code has the intended effect.

Exercise 5: Even More Assertions

One of your colleagues has suggested using the the following as an alternative to `x = atoi (argv[1])`, noting that the call to `sscanf` returns 0 if it fails to find an integer and 1 otherwise.

```
assert (sscanf (argv[1], "%d", &x) == 1);
```

What do you think about this strategy? Be prepared to discuss your answer with the class.

Exercise 6: Primality

Consider a function, `int is_prime (int n)`, which returns 1 if `n` is prime and `0` otherwise.

What preconditions and postconditions should this function have?

Be prepared to discuss the preconditions and postconditions with the class.

Exercise 7: Infrastructure for Prime Testing

Create

- `numutils.h`, a header file that declares `is_prime`,
- `numutils.c`, a code file that defines a stub of `is_prime` (your stub should always return 0), and
- `prime.c`, a wrapper file that lets the user type a value on the command line and answers whether or not the number is prime.

Compile them together to build a simple tester.

Exercise 8: A Unit Tester

Write unit tests for `is_prime`.

Exercise 9: Code!

Okay, you're finally ready. Write `is_prime` and verify that it works correctly.

Exercise 10: Correctness

Add `assert` statements at appropriate places in `is_prime`.

For Those With Extra Time