Lab 2: Simple C Functions/Using Testing Framework

Due date: Wednesday, January 11, beginning of the lab period.

Lab Assignment

Assignment Preparation

Lab type. This is an **individual lab**. Each student will submit his/her set of deliverables.

Collaboration. Students are allowed to consult their peers[1] in completing the lab. Any other collaboration activities will violate the *non-collaboration* agreement. No direct sharing of code is allowed.

Purpose. Wwe are starting to work with the main concepts of C. This lab will allow you to write a few simple C programs. Another important goal of this lab is to introduce the concepts of **debugging** and **testing**. Finally, this is the first lab, where your programming must follow the required style.

Programming Style. All submitted C programs must adhere to the programming style described in detail at

http://users.csc.calpoly.edu/∼cstaley/General/CStyle.htm

When graded, the programs will be checked for style. Any stylistic violations are subject to a 10% penalty. Significant stylistic violations, epsecially those that make grading harder, may yield stricter penalties.

Please note, that not all instructions from the link above are applicable to your Lab 2 assignments. If you do not understand the meaning of an instruction, please consult your instructor.

The following instructions are **in addition** to the rules outlined at the link above.

[1] A peer for the purpose of CPE 101 is defined as *"student taking the same section of CPE 101"*.

- **Short lines.** No line in your program shall be longer than 80 characters.
- **Header Comments.** The header comment **must be supplied** for each file submitted. The header comment must contain the following information:
  - Course number, section.
  - Instructor name.
  - Your name.
  - Name/Purpose of the program.
  - Date.

  Extra information in the header comment may be provided as well (version, extra dates - e.g., one for assignment commencement, one — for submission, etc...).

Testing and Submissions. Any submission that does not compile using the

```
gcc -ansi -Wall -Werror -lm
```

compiler settings will receive an automatic score of 0.

In this lab, you will be creating simple functions that return numeric (for the most part) values. You will be testing these functions using the testing framework introduced in Lab 1. This time, you will use more testing macros, and you will use the testing framework for its intended purpose: to test individual C functions.

For most programs in this lab, the instructor provides you with a `main()` function which contains a set of **public tests**. Your program (i.e., your implementation of the designated functions) must pass all instructor tests.

Please note that public tests are NOT the only tests that will be used when grading your submissions. There is a battery of **private** tests that the instructor maintains for each program. To make sure your program works correctly, make certain you write extra tests. Appendix A contains instructions on how the testing framework works. Make sure you read it before submitting your assignment.

The Task

Note: Please consult the instructor if any of the tasks are unclear.

For this lab, you will write and submit a number of simple programs. The programs will involve simple user-defined functions and the use of numeric (integer, floating point) variables, and some arithmetic operations.

Program 1: House Price Computation (price.c)

The first program is a simple calculator that given the price of a house and its square footage computes price per square foot. (*Note:* there is usually a wide diversity of houses on the market and a wide range of house sizes and prices. Price per square foot is one of the few measures that allows apples-to-apples comparisons between two very different houses. As such, it is commonly used by both real estate agents and perspective buyers to compare a variety of houses to each other).

Non-functional requirements

Non-functional requirements are general requirements that describe the nature of the problem and the expectations from the program. All requirements in the specifications you receive are numbered for your (and my) convenience.

SN1. The file name of your program shall be `price.c`.

Functional requirements

Functional requirements describe the behavior of the program.

SR1. Your program shall consist of two functions: function `main()` and function `computePrice()`.

SR2. The `int main()` function shall consist a sequence of invokations of the `checkit_double()` testing macro. Each such invokation shall represent one test case testing the behavior of the `computePrice()` function.

The instructor will provide you with a copy of the `main()` function containing the public test suite for this program. You can (and should) add more tests to it during the debugging stage, but the version of the program you submit shall be the instructor's `main()` function.

The `main()` function shall end with the `return 0;` statement.

SR3. The `computePrice()` function shall have the following declaration:

```
double computePrice(double housePrice, double footage);
```

SR4. The `computePrice()` function shall use no local variables.

SR5. The `computePrice()` function shall implement the computation of the price per square foot of a house. The first argument of the function shall represent the total price of the house in US dollars. The second argument of the function shall represent the size of the house in square feet.

The formular for computing the price per square foot is:

$$\mathsf{PricePerSquareFoot} = \frac{\mathsf{housePrice}}{\mathsf{footage}}.$$

Additional instructions

Your implementation can assume that all test cases for your program will satisfy the following conditions.

- The house price and the footage will be **non-negative** values.

Program 2: Temperature Converter (converter.c)

In the U.S., all temperatures are measured in degrees Fahrenheit, a measurement scale that, in the eyes of the instructor, lacks any meaningful intuition. Most European countries, including instructor's home country of Russia, measure temperature in degrees Celsius, which provide a much more intuitive way of measuring temperature. In particular, 0 degrees Celsius is the freezing point of water, while 100 degrees Celsius is the boiling point of water[2]. The (normal) human body temperature is 36.6 degrees Celsius.

You will write a program that converts temperature in degrees Celsius into degrees Fahrenheit and vice versa.

Non-Functional Requirements

CN1. Name your program `converter.c`.

Functional Requirements

CR1. Your program shall consist of three functions with the following declarations:

(a) `int main()`

(b) `double toFahrenheit(double temperature)`

(c) `double toCelsius(double temperature)`

CR2. The `main()` function for the public test is provided to you by the instructor. It contains the public test suit testing each of the two conversion functions defined in the program. The function ends with the `return 0;` statement.

Except for invocations of the `checkit_double()` macro and the `return` statement, `main()` function should not contain any other code.

CR3. The `toFahrenheit()` function takes one argument, `double temperature`, representing a temperature in degrees Celsius. It shall contain no local variable declarations. The `toFahrenheit()` function shall convert the input termperature into the temperature in degrees Fahrenheit, and return the computed value.

CR4. The `toCelsius()` function takes one argument, `double temperature`, representing a temperature in degrees Fahrenheit. It shall contain no local variable declarations. The `toCelsius()` function shall convert the input termperature into the temperature in degrees Celsius, and return the computed value.

CR5. The relationship between the temperatures in degrees Fahrenheit and degrees Celsius is captured in the formula below:

$$\mathsf{degreesF} = \frac{9}{5} \cdot \mathsf{degreesC} + 32$$

where $\mathsf{degreesF}$ is the temperature in degrees Fahrenheit and $\mathsf{degreesC}$ is the temperature in degrees Celsius.

[2] According to Wikipedia, this is no longer the case. However, for practiacal purposes, water freezes at 0 degrees Celsius and boils at 100 degrees Celsius under normal atmospheric pressure.

Additional instructions

- There are no restrictions on the value of temperature entered in your program (although, realistically, the temperature cannot be less than -273.15 degrees Celsius).
- Use `#define` preprocessor instructions for the constants in your program.

Program 3: Code table

This program shows you how `char` values in `C` programs can be interpeted as `int` values.

One of the simplest (and, unfortunately, easy to crack) ways to encrypt a text, is to replace each character in the text with a different character, by shifting along the alphabet. For example, a "Shift 2" cypher defined with way, would replace `"a"` with `"c"`, `"b"` with `"d"`, etc, until we reach `"z"` which is replaced with `"b"`.

For regular Latin alphabet (as used in English), there are 26 possible shifts from 0 to 25 (shift 0 represents lack of any encryption). You will write a function that, given a character in Latin alphabet and a desired shift produces the replacement character.

Non-Functional Requirements

EN1. Name your program `encode.h`.

Functional Requirements

ER1. Your program shall contain only one function with the following declaration: `char encode(char letter, int shift)`

ER2. Function `encode()` takes as input two arguments. The first argument, `char letter` represents a letter that needs to be encoded. The second argument, `int shift`, represents the alphabet shift for the code table.

The function shall return the replacement character for the input character in the encoding with the given shift.

For example, `encode('a', 3)` shall return `'d'` — a letter that comes three positions after `'a'` in the alphabet. The encoding circles around: so, `encode('y',5)` shall return `'d'` as well.

Your implementation of the `encode()` function shall have no local variable declarations.

ER3. The following information is helpful. The set of characters understood by `C` forms a so-called `ASCII table`. ASCII maps each character to a number between 0 and 255. All upper-case letters of Latin alphabet appear one after another in the ASCII table, with codes for individual characters increasing in alphabetical order. `'A'` has the ASCII code of 65. All lower-case letters of Latin alphabet appear one after another in the ASCII table, with codes for individual characters increasing in alphabetical order. `'a'` has the ASCII code of 97.

We will only be using lower-case Lating letters as input for this program.

ER4. As mentioned above, there are 26 possible legal shifts. A shift of 0 represents no encoding on the alphabet, i.e., `encode('a',0)` shall return `'a'`, and so on. The remaining 25 shifts are numbers from 1 to 25. However, function `encode()` shall take as input ANY integer number as a value of the shift, and correctly perform the encoding.

Shift of 26 is the same as shift of 0, shift of 27 is the same as shift of 1, etc. We **will not consider** negative shifts.

It is your responsibility to determine how to do this arithmetically.

ER5. Unlike the previous two programs, you are responsible for creating appropriate tests for the function `encode()`. The instructor is providing you with an partially complete program `checkEncode.c`, which has the following code (comments are omitted for space):

```
#include "encode.h"
#include "checkit.h"

int main() {

 checkit_char(encode('a',1),'b'); /* this is a sample test */

 return 0;
}
```

You are responsible for creating a slate of tests using the `checkit_char` macro (following the example above) that tests your program.

ER6. Your submission for this program will consist of both the `encode.h` file and the `checkEncode.c` file. `checkEncode.c` shall contain no less than 10 tests[3]. Your `encode()` function must succeed on **ALL** tests included in `checkEncode.c`.

Additional instructions

- Use `#define` preprocessor instructions for the constants in your program.

Submission.

Files to submit. You shall submit four files: `price.c`, `converter.c` and `encode.h` and `checkEncode.c`.

Your file names shall be as specified above (and remember that Linux is case-sensitive). We use automated grading scripts. Any submission that has to be compiled and run manually will receive a deduction.

Submission procedure. Use `handin` to submit your work. The procedure is as follows:

- `ssh` to `unix1`, `unix2`, `unix3` or `unix4`.
- Upon login, change to your Lab 2 work directory[4].

[3] You probably need more that that to ensure that your program works correctly.

[4] Lab 1 experience should teach you to create a new working directory for each assignment you do for this class inside your `cpe101` directory.

- Execute the `handin` command.

```
> handin dekhtyar lab02 price.c converter.c encode.h checkEncode.c
```

Other submission comments. **Please, DO NOT submit binary files. Please, DO NOT submit binary files.** (this has been an issue in the past.)

Grading

Any submitted program that does not compile earns 0 points.

Any submitted program that compiles but fails at least one **public** (i.e., made available to you) test earns no more than 30% of its full score (and can possibly earn less).

Any submitted program that compiles and succeeds on **all** publically available tests earns at least 50% of its full score.

All programs will be checked for style conformance. Any style violation will be noted. The program will receieve a 10% penalty.

Appendix A. Testing

This appendix contains a short description of the testing framework used in this lab.

Just as in Lab 1, every program you write for this lab shall contain the following preprocessor directive:

```
#include "checkit.h"
```

The `checkit.h` file, created by Dr. Aaron Keen for use in CSC 101 contains a number of *preprocessor macros*, which, essentially, are preprocessor-defined "constants" that replace specific text in the C program with some, specially prepared C code.

In our testing framework, four `testing macros` are defined. Each testing macro looks and feels like a function, and is used in our programs essentially in the same way. The macros are:

- `checkit_int(X,Y)`: for testing functions and expressions that produce integer values.
- `checkit_double(X,Y)`: for testing functions and expressions that produce floating point or double precision values.
- `checkit_char(X,Y)`: for testing functions and expressions that produce character values.
- `checkit_string(X,Y)`: for testing functions and expressions that produce string values.

We will not be using the last macro, `checkit_string(X,Y)` for a few more weeks, but in this lab and in the labs that follow, we will make active use of the other three macros.

A s discussed in Lab 1, a testing macro is used in a relatively straighforward way. Each macro is designed to accept as arguments two expressions of the same type (matching the name of the macro). The macro evaluates the expressions, compares them and reports success if the computed values were the same. It reports a failed comparison and provides some feedback (the values observed) when the computed values are diffrerent.

For `checkit_double(X,Y)` macro, expressions X and Y may evaluate to numbers that are slightly different from each other. To pass the `checkit_double(X,Y)` test the two values must be within 0.0001 of each other. Try, for example, these calls:

```
checkit_double(1.0, 1.1);
checkit_double(1.0, 1.01);
checkit_double(1.0, 1.001);
checkit_double(1.0, 1.0001);
checkit_double(1.0, 1.00001);

checkit_double(1.0, 0.9);
checkit_double(1.0, 0.99);
checkit_double(1.0, 0.999);
checkit_double(1.0, 0.9999);
checkit_double(1.0, 0.99999);
```

and see, which tests succeed and which fail.

I n Lab 1 we used the testing framework to compare two expressions to each other. To test functions, we need to do essentially the same thing, except we will be using a function call expression as one argument to the testing macro. We will then determine which value the function should return given the inputs to it, and supply that value as the second argument.

This procedure is illustrated on the following example. Consider the function

```
int sumSquares(int x, int y) {
  return x*x + y*y;
}
```

Suppose, we want to test this function using the `checkit_int(X,Y)` macro.

Our first step is to come up with the first test case. `int x` and `int y`, the formal parameters to function `sumSquares()` can take any integer values. It is easy for us to start with a situation when both `x` and `y` are set to 0. The appropriate function call is

```
sumSquares(0,0)
```

In this test case, we want to compute the value 0^2+0^2. This is an arithmetical expression we can evaluate **manually**. The result we get is 0. If `sumSquares()` is implemented correctly, the function call `sumSquares(0,0)` must return the value `0`. This allows us to form our first test case using the `checkit_int(X,Y)` macro:

```
checkit_int(sumSquares(0,0),0);
```

Here, we substitute our function call expression for the first argument of the macro, and the manually computed (and thus correct) expected result for the second argument.

We can continue devising test cases: $1^2 + 0^2 = 1$, $0^2 + 1^2 = 1$, $2^2 + 1^2 = 5$, $2^2 + (-2)^2 = 8$, $100^2 + 5^2 = 10,025$, and so on. For each test case we created and manually verified, we can supply the appropriate invocation of the testing macro:

```
checkit_int(sumSquares(1,0),1);
checkit_int(sumSquares(0,1),1);
checkit_int(sumSquares(2,1),5);
checkit_int(sumSquares(2,-2),8);
checkit_int(sumSquares(100,5),10025);
```

If all these tests are put into the `main()` function of a `C` program, which is subsequently compiled and run, you will be able to see the results of each individual test.

All testing macros are designed to refer back to the line of code (int `.c` file which contains the `main()` function. This way, you can trace each success/failure message produced by your program to the specific test macro invocation.

Selecting good tests. Some tips on how to create test cases.

In the current context a **text case** is essentially a collection of values for all attributes for a `C` function that is being tested.

- Start with *simple* test cases. Your first test cases should be easy to evaluate manually.
- Think, what "interesting" values of the arguments exist. For example, functions often have to handle values of 0 and/or 1 in a special way. Make sure you constuct test cases that contain these values.
- Sometimes a function you are testing can take as input only a small finite number of inputs. In this case, it may make sense to conduct **exhaustive testing**, i.e., to create one test case per possible input to the function.
- Often, function arguments have *marginal values.* E.g., house prices must be non-negative, and square footages of the houses must be positive. Create test cases that check the marginal values to make sure these values are handled properly.

More hints and tips will be provided throughout the quarter, as we learn more `C` syntax.