Lab 3: Testing, Composition, Assignment, Printing

Due date: Tuesday, October 8, 11:59pm.

Note: October 8 is also the day of the lab exam. You will not be able to work on this assignment during the October 8 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. You are continuing your acquaintance with C functions and their use.

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

http://users.csc.calpoly.edu/∼dekhtyar/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.

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 most of these functions using the testing framework introduced in Labs 1 and 2.

In this lab, you will have to write tests for your functions. Your deliverables for this lab include both the implementations of requested functions **and** C programs that test each individual function.

The Task

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

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

The Grower's Grove Game

In this lab you are starting to develop some code for a simple, step-by-step strategy game of growing crops which we call The Grower's Grove Game. We will be developing the detailed rules of the game in followup assignments. For the current lab, you will be modeling portions of the game's growing engine.

The basics of the game are explained below.

The premise. The Grower's Grove Game is a game of growing crops on a single plot of land (Grower's Grove). In the game, the player controls growing crops (plants) on a plot of land over a period of time. The player will decide which crops to plant at which location in the Grower's Grove. The results of the planting actions will yield harvests, which, in turn, will provide the player with the resources to obtain seeds for the next season's planting.

In this lab we concentrate on modeling the growing conditions in the Grower's Grove.

Grower's Grove. The playing field, Grower's Grove is a square plot of land of the size 20×20 acres. Each square of this grid is labeled with a pair of coordinates (x, y) where both x and y range from 1 to 20. We refer to each 1-acre square as the *field*. Each field in Grower's Grove can be planted with a single type of crop at a time.

Growing Conditions. In this lab we emulate the growing conditions for a single type of crop, which we refer to as groveberry. Our ability to grow groveberries on a given field (x, y) is guided by three core properties of the field:

- Soil quality. This parameter represents the suitability of the soil on each field for growing groveberries.
- Sun exposure. Growth of groveberries is affected by the amount of sun the field receives.
- Irrigation exposure. Finally, groveberries like water, but their growth can be stymied by both its lack and the excessive amount. Each field has a certain level of access to irrigation facilities, measured by its irrigation exposure. It determines how much water the field gets.

Irrigation. Irrigation in Grower's Grove is provided by a canal that runs through the main diagonal of the grove starting at the field $(1, 1)$ and ending at the field $(20, 20)$, and going straight through every field (x, y) where $x = y$. Note that we assume that despite the presence of the canal going through these fields, each such field still maintains 1 acre of plantable surface.

Soil Quality. The best soil in Grower's Grove is located around the edges of the grove, while the central portion of the grove is not as fertile and accommodating to groveberries. Additionally, the soils on neighboring fields are arranged in a checkerboard pattern with alternating soil types.

- Type A soil. Fields (x, y) where $x + y$ is an even number have Type A soil. The soil quality in these fields is measured using the following formula:

$$soilQuality(x, y) = 1 + \sqrt{(x - 10)^2 + (y - 10)^2}.$$

- Type B soil. Fields (x, y) where $x+y$ is an odd number that Type B soil. The soil quality in these fields is measured using the following formula:

$$soilQuality(x, y) = 1 + \frac{|x-10| + |y-10|}{2}$$

Sun exposure. Field $(8, 12)$ is on a hilltop and gets the maximum sun exposure among the fields in the Grower's Grove. The landscape gently slopes in all directions from it, and the sun exposure levels follow a shape of a multivariate normal distribution. That is, the sun exposure of a field (x, y) can be computed as follows:

$$sunExposure(x, y) = 10 \cdot e^{-\frac{1}{2}\left(\frac{(x-8)^2}{10} + \frac{(y-12)^2}{5}\right)}.$$

Irrigation exposure. The irrigation exposure of the field is essentially inveresly proportional to its distance from the canal:

$$irrExposure(x, y) = \frac{10}{1 + |x-y|}.$$

Growing groveberries

Soil quality, sun exposure and irrigation exposure of each field affect the growing conditions for groveberries. There are three core growing parameters that we are interested in tracking. In this lab, you will write code to compute the estimates for these three parameters. Each will depend on the soil quality, sun exposure and irrigation exposure. The three parameters are:

- Expected yield: how much of the crop will grow.
- Expected quality: groveberries are fickle - you may get a good yield, but the berries themselves can be quite tasteless. This parameter predicts how tasty the groveberries will be.
- Expected time to harvest: different growing conditions may mean shorter or longer time to harvest.

The estmates of the three paramters above can be computed as follows.

Expected yield. The better the soil and the more sun a field gets, the higher the yield. The best irrigation exposure for high yields is around 7, and it drops off from there. The formula for the estimate is:

$$estimatedYield(x, y) = soilQuality(x, y) \cdot sunExposure(x, y) \cdot \frac{(7 - |irrigationExposure(x, y) - 7|) + 1}{2}.$$

Expected quality. The better the soil, the better berries we expect to grow. The optimal sun exposure is around 10, the optimal irrigation exposure is around 6 to get the best tasting fruit. The drop off in quality is controlled as a bivariate normal distribution:

$$estimatedQuality(x, y) = soilQuality(x, y) \cdot e^{-\frac{1}{2}\left(\frac{(sunExposure(x,y)-10)^2}{5} + \frac{(irrigationExposure(x,y)-6)^2}{5}\right)}.$$

Expected time to harvest. Groveberries require around 75 days to grow give or take about 15 days. The time to grow is controlled by the sun exposure and the irrigation exposure. Too much or too little of both, and the growth slows.

$$harvestTime(x, y) = \left\lceil 60 + \sqrt{sunExposure^2(x, y) + irrExposure^2(x, y)} \right\rceil.$$

Overall Score of the Field

When growing berries at the Grower's Grove, we want to grow a lot of them, we want our berries to be tasty, and we want to have an opportunity to harvest as many of them as possible. The overall score of a field is a measurement of how well the field is suited for growing groveberries. It is computed as follows:

$$fieldScore(x, y) = estimatedQuantity(x, y) \cdot estimatedQuality(x, y) \cdot \frac{365}{harvestTime(x, y)}.$$

Software To Write

For this lab, you will write C functions that compute the seven parameters specified in the game description above, will use the `checkit` macros to test the work of your functions, and will also write one more function, that, given a field, prints out the information about it.

Function Implementations

Create a file `grove.c` and put the definitions of the following eight C functions in it. The first seven functions must perform computations according to the formulas specified above.

- `double soilQuality(int x, int y)`: this function takes as input the (x, y) coordinates of a field and returns the soil quality of the field.
- `double sunExposure(int x, int y)`: this function takes as input the (x, y) coordinates of a field and returns its sun exposure value.
- `double irrigationExposure(int x, int y)`: this function takes as input the (x, y) coordinates of a field and returns its irrigation exposure value.
- `double estimateYield(int x, int y)`: takes as input the (x, y) coordinates of a field and returns the estimate of the groveberry yield at harvest time.

- `double estimateQuality(int x, int y)`: takes as input the (x, y) coordinates of a field and returns the estimate of the quality of groveberry harvest.
- `double harvestTime(int x, int y)`: takes as input the (x, y) coordinates of a field and computes the time to harvest on it.
- `double fieldScore(int x, int y)`: takes as input the (x, y) coordinates of a field and computes the overall score of the field.

The eighth function, `void outputScores(int x, int y)` takes as input the input the (x, y) coordinates of a field and outputs all seven values computed by the functions above. The output format must follow the specifications below **exactly**.

Output specifications. The first line of the output is a string of 30 `'-'` characters. The second line of the output is left empty. The third line of output looks as follows:

```
Field: <X>,<Y>
```

where `<X>` and `<Y>` are replaced with the actual coordinates x and y of the field. The fourth line of the output is empty.

The fifth through the thirteenth lines of output shall look as follows:

```
       Soil quality: <xxxxxxx>
       Sun exposure: <xxxxxxx>
Irrigation exposure: <xxxxxxx>

    Estimated yield: <xxxxxxx>
  Estimated quality: <xxxxxxx>
    Time to harvest: <xxxxxxx>

      Overall score: <xxxxxxx>
```

where on each line `<xxxxxx>` is replaced with the appropriate for that line value.

The fourteenth line is empty and the last, fifteenth line of output is again a string of 30 `'-'` characters.

In addition to putting function definitions in `grove.c`, create a header file `grove.h` containing all function declarations as well as any `#define`d constants.

Testing Grower's Grove functions

Unlike **Lab 2**, where all functions you were writing were independent of each other, in this lab, functions you build work towards the construction of a single piece of software, the Grower's Grove Game. You are constructing this game in a *brick-by-brick* fashion, by creating C functions that compute a variety of individual aspects of the game. You can already see how implementing some of these functions helps you implement more functions in the future: e.g., computing estimates for quality and quantity of the groveberry harvest uses computations of soil quality, sun exposure and irrigation exposure.

In order to ensure that the game will eventually simulate growing groveberries the right way, we need to test and *validate* (i.e., assure that it produces the outputs we want) every function we build. The process of validating individual functions/functionality in a large software project is called *unit testing.* Here a *unit* is a single component of a program. In this lab, each `C` function is a "unit".

To unit-test your functions, create seven `C` programs named `test-<function>.c`, where `<function>` is the name of the `C` function you are testing (`test-soilQuality.c, test-sunExposure.c, test-irrigationExposure.c` and so on). In each file, include the `main.c` function that uses **appropriate** *checkit* macro calls to test the work of the designated function.

When creating tests, follow the instructions below:

- **Use only valid inputs.** You do not need to worry about what functions return on input values outside of the $[1, 20]$ range. Later, we will include boundary checks of the input parameters in the code of all your functions, but we do not need to worry about it now.
- **Include some boundary conditions tests.** A boundary condition is a test case that makes your function include a marginal result - e.g., reach a maximal, or a minimal value. Alternatively, a boundary condition is test case that supplies values that are on the boundary of the region on which the function is defined. E.g., a $(1, 1)$ field lies on the boundary of Grower's Grove and therefore can be used as a boundary test. So is $(20, 10)$ or $(20, 20)$.
- **Test evenly.** Make certain your test cases cover the space of all possible test cases nicely. Using $(1, 1), (1, 2), (2, 1), (2, 2)$ as the only test cases may be a bad idea because a large area of the Grower's Grove is missed and left unchecked.
- **Test all possible situations.** For example, soil quality is computed differently depending on whether the sum of the field coordinates, $x + y$ is even or odd. You need to test both situations.

Each file must include at least 10 test cases.

Note: The instructor will release five test cases per function by the end of the day *Friday.* Your eventual test cases must be different.

Testing your output function. You need to also make sure that your `outputScores()` function works properly. To do so, you will create a `display.c` program, which contains a `main()` function with one or more calls to the `outputScores()` for different Grower's Grove fields.

On Friday, the instructor will release the exact output his version of `display.c` produces. You can use that output and compare it to yours to ensure that your function prints the right thing.

However, **you do not need** to submit `display.c`. This program is needed only so that you could validate `outputScores()` for yourself.

Submission.

Files to submit. You shall submit the following files: `grove.h`, `grove.c` and the seven `test-<function>.c` files.

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 3 work directory.
- Execute the `handin` command.

```
> handin dekhtyar lab03  <files>
```

(note, you can submit files one by one, rather than using one command.)

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