Lab 4: Conditional Statements, Code Reuse

Due date: Thursday, October 17, 11:59pm.

Note: Lab 5 assignment will be handed out on October 17.

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.

The Task

The goal of this assignment is to start using conditional statements in a number of different settings. As the vehicle for learning how to use conditional statements, we will continue developing the Grower's Grove Game. In this lab, you will revisit the code you wrote for **Lab 3**, modify some of the math to make the game more balanced and put variable guards.

We will also introduce some new decision-making features into the game, for which you will contribute functions.

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

Part 1: Balancing the game better

Looking at the outputs generated by **Lab 3** code suggests that in computing a number of parameters, we have created significant imbalance in terms of the values the parameters can take. Moving forward, we want decision-making to be more challenging, and the game to be more balanced and not to offer obvious alternatives that are heavily influenced by only one or two different parameters of the game.

We plan to rebalance the game as follows.

Sun exposure computation. The original sun exposure computation makes sun exposure drop drastically as one moves away from the field with the optimal sun exposure conditions. By the time we reach the edges of the Grower's Grove, the sun exposure is negligible. We modify the sun exposure computation formula to make the sun exposure decay more graceful. The new formula is

$$sunExposure(x, y) = 10 \cdot e^{-\frac{1}{2}\left(\frac{(x-8)^2}{70}+\frac{(y-12)^2}{70}\right)}.$$

Irrigation exposure computation. As it turns out, each field in the Grower's Grove has access to groundwater supplies in the amount of 2 units. The irrigation exposure itself still cannot be higher than 10 units. The formula for computing irrigation exposure thus is changed to:

$$irrExposure(x, y) = \min\left(\frac{10}{1+|x-y|}+2, 10\right).$$

Estimated crop quality computation. Estimated crop quality computation also suffered from a very sharp decay in quality when parameter values diverge from the optimum. This has been rectified in a new formula:

$$estimatedQuality(x, y) = soilQuality(x, y) \cdot e^{-\frac{1}{2}\left(\frac{(sunExposure(x,y)-10)^2}{40}+\frac{(irrigationExposure(x,y)-6)^2}{40}\right)}.$$

Expected time to harvest computation. New considerations result in a formula that provides a more meaningful estimate:

$$harvestTime(x, y) = \left\lceil 60 + 3 \cdot \sqrt{sunExposure^2(x, y) + irrExposure^2(x, y)} \right\rceil.$$

Overall score. Finally, the overall score winds up being on the order of 10^3. We can just as easily normalize the score a bit:

$$fieldScore(x, y) = \frac{estimatedQuantity(x, y) \cdot estimatedQuality(x, y)}{100} \cdot \frac{365}{harvestTime(x, y)}.$$

All other parameters are computed as before.

Task 1: rewrite your **Lab 3** functions for computing the parameters of the Grower's Grove Game so that they compute the values according to the new formulas.

Note on file management. In order to preserve your work, you should do the following. Crate a `lab4` working directory. Copy all **Lab 3** files there. Keep a copy of your **Lab 3** submission intact. Modify only the files in the `lab4` working directory. This way you have both the new code and the old code, which you may still need in the future.

Part 2: Variable Guards

As stated in the original game description, Grower's Grove is a 20×20 grid of fields. All functions you created in **Lab 3** take as input a part of `int` values `x` and `y`. Until now, there was no way to verify that the values are in the designated 1..20 range. Your next task is to correct this problem.

In what follows, we consider the values of function input parameters `x` and `y` to be valid if they fall in the range 1..20.

Task 2: modify the code of all your `Lab 3` functions as follows.

- For the `outputScores()` function, check that both `x` and `y` are valid. If both values are valid, proceed with the output according to the **Lab 3** spec for `void outputScores()`. If either `x` or `y` is invalid, output the following text:

  ```
  Field <x>, <y> is invalid!
  ```

  and return. Here put the values of variables x and y in places of <x> and <y> respectively. For example, a call to `outputScores(10, 42)` shall print

  ```
  Field 10, 42 is invalid!
  ```

- For all other **Lab 3** functions, check that both `x` and `y` are valid. If both are valid, compute appropriate parameter according to the specs in this lab (and/or **Lab 3** as necessary) and return the value. Otherwise, return -1.

Part 3: Money

You will be implementing new functionality for the Grower's Grove Game. The purpose of the game is to plant groveberries, collect harvest, sell it, profit, and plan more groveberries. This section describes the economics of the Grower's Grove Game.

Game modes. Grower's Grove Game can be played both in as a single player strategy game (solitaire), and a computer simulation game. Some of the functionality developed for this lab is necessary for the computer simulation part, but not for a solitaire mode of play.

Funds. On each step of the game, the player[2] has some amount of money. Money serves both as the means of scoring the game – the more money the player has, the better off the player is, and as the means of controlling what the player can and cannot do on each step.

Player actions. The core action of a player on a given step is to plant a field of groveberries. This action is not free: it comes with a cost. The cost of the action depends on the field conditions (i.e., how hard it is to actually plant) and on the time to harvest - the longer one has to wait for the harvest, the more it costs to plant it. The cost computation formula for a field (x, y) is:

$$plantingCost(x,y) = \left\lceil \left(10 - \frac{soilQuality(x,y)}{2}\right) \cdot \frac{sunExposure(x,y) + irrExposure(x,y)}{2} \right\rceil.$$

(informally: good soil means it is easier to plant, more sun and more irrigation means it is harder to plant.)

Pricing the harvest. The player returns the money by selling the harvest on the market. The total amount of money the player gets for the groveberry harvest is affected by the yield and the quality of the harvest. In particular, the price per unit of the groveberry harvest is affected by the quality of the harvest alone as follows:

- If the quality of the harvest is less than 2, then the price of one unit of groveberry harvest is 0.5.
- If the quality of the harvest is between 2 and 4, the price is 0.75 per unit.
- For harvest quality 4 (inclusive) to 5 (exclusive), the price is 1 per unit.
- For harvest quality 5 (inclusive) to 6 (exclusive), the price is 2 per unit.
- For harvest quality 6 and above the price is computed according to the formula

$$pricePerUnit(x,y) = \frac{estimateQuality(x,y)}{2}.$$

The overall revenue of the harvest is then computed in a straightforward way:

$$revenue(x,y) = pricePerUnit(x,y) \cdot estimateYield(x,y).$$

Other revenue-related computations. We can now compute a few more values. First, the profit/loss on a field is

$$fieldProfit(x,y) = revenue(x,y) - plantingCost(x,y)$$

We can also compute the return on investment factor:

$$roi(x,y) = \frac{fieldProfit(x,y)}{plantingCost(x,y)} \cdot 100$$

[2] Either a human, or a computer simulaton.

Finally, we can compute expected annual revenue:

$$annualRevenue(x, y) = revenue(x, y) \cdot \frac{365}{harvestTime(x, y)}.$$

Task 3: Implement the following `C` functions computing the respective values specified above:

```
double plantingCost(int x, int y);
double pricePerUnit(int x, int y);
double revenue(int x, int y);
double fieldProfit(int x, int y);
double roi(int x, int y);
double annualRevenue(int x, int y);
```

Each function shall be implemented in the same way, you are implementing **Lab 3** functions in this lab: i,e., with variable guards for the input parameters.

Define all functions in the `grove.c` file, place all necessary declarations and constant definitions in the `grove.h` file.

Part 4: Strategy, Decision-Making

In this part, you will write three functions that compare two fields to each other as far as the fields' money-making potential is concerned, and output (print) the information about the better field.

`void compareProfits(int x1, int y1, int x2, int y2)` . The function takes as input coordinates of two fields, `(x1,y1)` and `(x2,y2)`. It validates **all input parameters**. If at least one parameter is invalid, the function prints

```
Invalid input!
```

and returns. If all inputs are valid, the function compares the *estimated profit of a single planting* for the two fields. It determines the field with the higher estimated profit and outputs the coordinates of the field in the following format:

```
(<x>,<y>)
```

where `<x>` and `<y>` are replace with the actual values of the input parameters for the winning field.

`void compareAnnual(int x1, int y1, int x2, int y2)` . The function takes as input coordinates of two fields, `(x1,y1)` and `(x2,y2)`. It validates **all input parameters**. If at least one parameter is invalid, the function prints

```
Invalid input!
```

and returns. If all inputs are valid, the function compares the *estimated annual revenue* that can be collected from each of the two fields. It determines the field with the higher estimated profit and outputs the coordinates of the field in the following format:

(<x>,<y>)

where <x> and <y> are replace with the actual values of the input parameters for the winning field.

void betterInvestment(int x1, int y1, int x2, int y2) . The function takes as input coordinates of two fields, (x1,y1) and (x2,y2). It validates **all input parameters**. If at least one parameter is invalid, the function prints

```
Invalid input!
```

and returns. If all inputs are valid, the function compares the return on investment for each of the two fields. It outputs the coordinates of the field in the following format:

(<x>,<y>)

where <x> and <y> are replace with the actual values of the input parameters for the winning field.

Task 4: implement the three abovementioned functions.

Task 5: testing. You shall create checkit.h-style tests for each new function you create in **Task 3**. These files need not be submitted - I will be testing your code in a different ways, but you must create the testing programs for your own use.

You shall also create a C program decisionTest.c, which will contain 10 calls to each of the three functions compareProfits(), compareAnnual() and betterInvestment() (30 calls total). A simple version of this program will be made available to you over the weekend. Your test cases must be different from instructor's test cases. They also must produce correct output.

Submission.

Files to submit. You shall submit the following files: grove.h, grove.c and decisionTest.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 3 work directory.
- Execute the handin command.

```
> handin dekhtyar lab04  <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.)