Program 2: Planting in the Grower's Grove

Due date: Tuesday, November 19, 11:59pm.

Purpose

To write a program requiring use of control structures (loops and conditional statements), arithmetic, relational and boolean expressions, arrays, and functions.

Assignment

This is a **pair programming** assignment. Outside of each pair, all non-collaboration, non-plagiarism policies of the course are in full effect and will be enforced.

Project Description

In a number of labs throughout the course we have developed a variety of code around the concept of a single-person, turn-based strategy game called the Grower's Grove Game. The game involves growing a special kind of berries, *groveberries* on a variety of fields of the place called the Grower's Grove. So far, in the labs we have not really stressed the gaming aspects of Grower's Grove, concentrating instead on getting the mechanics of the game's computations right.

In this assignment, you will implement a simple, one-step portion of the Grower's Grove Game. The general Grower's Grove Game proceeds as follows. On each step, the player has a certain amount of money, and can use all the money, or part of it to plant one or more fields of *groveberries*. After a certain number of days (dictated by the time to harvest each field), the groveberry harvest is collected and sold. The player receives a profit (or a loss) and starts the planting cycle anew.

For this assignment you will implement one step of the game. Specifically, your program will allow the user to

- Browse the Grower's Grove maps of a variety of field parameters that may help the user to decide which fields to plant.
- Add a field to the current planting schedule.
- Remove a field from the current planting schedule.
- Complete the planting schedule and receive an overview of the success of the current planting.

A detailed description of the program is found below.

Assignment Organization and Testing

For this assignment you will work with the following files:

- `grove.c` and `grove.h`: these are the core Grower's Grove Game files containing your functions from Lab 3 and Lab 4. The functions defined in these files must be correct – **please ensure it by testing your functions against the instructor's outputs before proceeding with this assignment**.

  You may use the `grove.c` and `grove.h` files from either of the partners - just pick one implementation that you find more agreeable. If the implementations of both partners work correctly, then the rest of your program will work properly when compiled with **either** of versions. However, you will be submitting only on copy of `grove.c` and `grove.h` files, so pick the ones you want to use.

- game.h: the header file consisting of function declarations for the functions you must implement for this assignment. This file is released to you by the instructor. You can modify the contents of this file, provided you keep every single function declaration in it. You can add other functions, and you can use this file to store your `#define`d constants. You will submit your version of `game.h` file together with other files for this project.

- game.c: the files with the definitions of all the functions declared in `game.h`, and possibly other functions[1]. This is the main file you will be working on.

- play.c: This file will contain the `int main()` function for your program.

In general, you will be compiling your program using the following command:

```
$ gcc -ansi -Wall -Werror -lm -o game play.c game.c grove.c
```

Notice that `badges.c` file is not used in compilation, and is not listed above: your assignment does not have a badge-checking component, therefore, code from the `badges.c` file is not needed.

You submission will include all the files from the list above.

Testing. The executable of the instructor's version of the program is made available to you in different formats. The formats include the 32-bit and the 64-bit versions compiled on Cal Poly's servers and a 64-bit version compiled on my laptop running an old version of Ubuntu Linux. I may offer other formats if they are needed, but note that in order for me to put up an executable for a specific architecture of a computer, I must have access to the appropriate hardare.

At the same time, because the instructor's executable produces only text output, it can be easily run on the CSL servers, even if it requires a remote login. So you all should be in position to run the instructor's executable.

Test Cases. You will need to develop a number of test cases to test the work of your program. While a "full" test case for an interactive game is an exact script of all inputs the user makes, we will limit the formal test cases to simply being a specific planting schedule.

A *planting schedule* is a collection (set) of Grower's Grove fields.

A *formal test case* for this assignment consists of the following components:

1. A planting schedule.

[1] "Other functions" may be any helper functions used in your code that **do not** need to be called from the `main()` functions.

2. The number of fields in the schedule.
3. Total cost of the planting schedule.
4. The longest time to harvest among all fields in the schedule.
5. The total amount of money the planting schedule generates.
6. The profit brought in by the planting schedule.
7. Total amount of money the player will have after executing this planting.

You program is said to **pass the test case** if when given the planting schedule as the constructed input, it produces the correct values for the items (2)–(6) from the list above.

Here is a sample test case (assuming the player starts with 160 units of money):

```
Schedule: (1,1), (2,2)
Number of fields: 2
Total cost of planting: 48.00
Time to full harvest: 93.00
Revenue: 671.19
Profit: 623.19
Money on on hand: 773.19
```

or, represented in a compact form:

```
(1,1), (2,2)
2
48.00, 93.99, 671.19, 623.19, 773.19
```

Output. Your output and the instructor's output shall match exactly, up to spacing, capitalization and other idiosyncracies.

Style. Style manual is strictly enforced.

Grower's Grove Game

The high-level description of your program is presented in this section. The followup section contains all technical implementation details.

Startup. Upon start-up, the program shall output an initial greeting and proceed directly to the main menu of the game. The player will be assigned a specfic starting amount of money.

Main Menu. The main menu of the game shall have the following eight options for the player to choose from:

- View current planting schedule
- Add a field to the schedule

- Remove a field from the schedule
- View planting costs for each field
- View projected harvest yields for each field
- View projected harvest quality for each field
- View projected profit from each field
- Finalize plantings and finish the game

Gameplay. On each step of the game, the player selects one of the menu options. If the selected option is to add or remove a field, the program engages the player in the field selection dialog. For any viewing option, the program prints the formatted map of the appropriate values for each field. If the last option is selected, the program prints out the analytical information about the selected planting schedule and quits.

Adding a field to the schedule. The program shall ask the player to enter the coordinates of the field. Once valid coordinates are entered, the program outputs the information about the field (the cost of planting) and asks the player to confirm the selection. If the player confirms the selection, the field is added to the planting schedule and the amount of money on the hands of the player is updated. If the player cancels the selection, the program proceeds to the main menu without changing the planting schedule.

Removing a field from the schedule. The program shall ask the player to enter the coordinates of the field. Once valid coordinates are entered, the program outputs the information about the field (the cost of planting) and asks the player to confirm the selection. If the player confirms the selection, the field is removed from the planting schedule and the amount of money on the hands of the player is updated. If the player cancels the selection, the program proceeds to the main menu without changing the planting schedule.

Viewing the planting schedule. The program prints a map of the Grower's Grove where each field is labelled with either 0 - if it is not a part of the planting schedule, or with 1 - if *it is* part of the planting schedule.

Viewing planting costs, harvest yields, harvest quality, projected profit. For each of these selections, the program outputs nicely formatted map of the Grower's Grove specifying the value of the given parameter for each field. The values are computer by the corresponding functions from the grove.c function library.

Finalizing plantings and finishing the game. The program outputs the final message detailing the planting schedule, its cost, the time it takes to complete the planting, the collected revenue and the profit. After that, the program completes its work.

Details

Maintaining a Planting Schedule

The simplest way to maintain a planting schedule is by having a 2-dimensional array of 20×20 `int` values. If a field is in the planting schedule, the array will have a value of 1 in the array element corresponding to the field. Otherwise, the array element value will be 0. Note that such arrays will be passed into many of the functions declared in the `game.h` file.

Note. Recall that variables in C need to be initialized. The same applies to arrays. You must make sure that at the beginning of the game, all fields are not in the planting schedule by initializing your planting schedule array to be all zeroes.

Array-to-Field coordinate correspondence. Also recall that field coordinates in Grower's Grove are between 1 and 20, and a field (x, y) mean a field at the intersection of *column* x and *row* y. At the same time, array indexes start with 0, and if `a` is a 2D array, then `a[i][j]` represents an array element at the intersection of *row* i and *column* j. Use this information to determine where in your array the values representing the field (x, y) will be located.

Greeting Message and Startup Money

The *starting money* for the player shall be a `#define`d constant in your code. For the version of the program you submit, set this constant to the amount of 150.

The greeting message shall look as follows:

```
------------------------------------------------------
-- Welcome to the Grower's Grove                    --
------------------------------------------------------

Grower's Grove is a groveberry-growing farm.
It has 400 1-acre fields, situated in a 20 by 20 grid.
Each field has different growing conditions,
which affect the yield and the quality of
the berries. This affects the price of berries.
------------------------------------------------------
The goal of this mini game is to come up with the best
planting of groveberries with a fixed amount of money.
------------------------------------------------------
You have <XXX.XX> units of money.

Can you excel at planting?
------------------------------------------------------
```

The `<XXX.XX>` shall be replaced with the starting amount of money for the player.

There shall be **three** (3) empty lines betwen the last line from the message above and the first line of the game menu.

Create a special function for printing the greeting message. The function declaration,

```
void printWelcome(double money);
```

is in the `game.h` file. The input parameter is the amount of starting money the player has.

Main Menu

The main menu of the game shall look as follows:

```
Money on hand: <YYY.YY>

       == Game Menu ==
[1] View current plantings
[2] Add a field
[3] Remove a field
[4] View planting costs
[5] View projected yields
[6] View projected harvest quality
[7] View projected profit
[0] Finalize plantings
```

Here, on each step of the game, `<YYY.YY>` is replaced by the amount of money the player currently has on hand. This amount is equal to the starting amount of money minus the cumulative cost of planting for all selected fields.

Immediately following the menu (next line), your program shall print the prompt:

```
Make a selection (0-7):
```

The valid menu selections are numbers $0-7$. All other numeric inputs are ignored and the program shall simply reprint the prompt. Here is an example of a session where and incorrect value was entered twice:

```
Make a selection (0-7): 10
Make a selection (0-7): -1
Make a selection (0-7): 0
```

(after the last selection the program produces the final output according to the information provided below).

The main menu shall be supported by a function

```
int  menu(double money);
```

declared in `game.h` file. The function takes as input the current amount of money on hand. It prints the menu and returns the menu choice the user made.

Adding a Field to the Planting Schedule

If the user enters option "2" as their choice, the program shall conduct the dialog for adding a field to the planting schedule. The dialog proceeds as follows.

The program prints a new line, followed by the following text:

```
=== Adding a Field to the Planting Schedule ===

You have <YYY.YY> units of money

Enter the field coordinates (enter "0 0" to exit):
```

where `<YYY.YY>` is replaced with the player's currently available amount of money (money on hand).

The program then reads a pair of integer numbers representing the coordinates, *until* a valid pair of numbers is enetered. A *valid pair of numbers* is either a pair `0 0`, or an existing field that can be planted. In particular, the program shall perform the following checks and provide the appropriate feedback to the user:

1. **Field coordinates make sense.** Recall that a valid Grower's Grove field will have both coordinates between 1 and 20. If invalid field coordinates are entered, the program shall engage in the following dialog:

   ```
   Enter the field coordinates (enter "0 0" to exit):40 10
   (40, 10) is not a valid field. Try again.

   Enter the field coordinates (enter "0 0" to exit):
   ```

2. **Field already selected for the planting schedule.** A player may, by mistake, enter the coordinates of a field that has already been selected for the planting schedule. The program shall perform this check, and if successful, engage in the following dialog[2]:

   ```
   Enter the field coordinates (enter "0 0" to exit):1 1
   Field (1, 1) is already in the planting schedule

   Enter the field coordinates (enter "0 0" to exit):
   ```

3. **Planting is not affordable.** A player may select a field with a planting cost that exceeds the player's currently available amount of money. If that is the case, the program shall engage in the following dialog:

   ```
      === Adding a Field to the Planting Schedule ===

    You have 65.00 units of money

   Enter the field coordinates (enter "0 0" to exit):11 10
   You cannot afford to plant in field (11, 10)
   Planting cost is 75.00. You have 65.00 units of money

   Enter the field coordinates (enter "0 0" to exit):
   ```

If the player enters coordinates `0 0`, the program shall complete the attempt to add a new field and return to the main menu without making any changes to the schedule and with no further interaction with the player: the player decides to abandon the idea of adding a new field.

If a player enters coordinates for a valid field that is affordable and is not part of the schedule yet, the program shall engage in the following dialog:

```
Enter the field coordinates (enter "0 0" to exit):1 1
Field (1, 1) has planting cost of 21.00
Would you like to plant on it? (yes=1/no=0):
```

The program then waits until the player enters either `"1"` or `"0"` (no other inputs are acceptable).

If the player enters `"0"`, the program shall print

```
Field not added!
```

and proceed to the main menu. This completes the attempt to add a field. No changes are made to the current schedule.

If the player enters `"1"` the program shall do the following:

[2] This dialog happened in an instructor's game *after* the instructor successfully selected field $(1, 1)$ for the planting schedule on the previous step of the game.

- Add the field to the planting schedule
- Compute the new value of the money on hand for the player
- Print

  ```
  Field not added!
  ```

  and return to the main menu

Here are examples of an interaction that adds a field and an interaction that does not add a field:

```
Make a selection (0-7): 2

 === Adding a Field to the Planting Schedule ===

 You have 65.00 units of money

Enter the field coordinates (enter "0 0" to exit):1 1
Field (1, 1) has planting cost of 21.00
Would you like to plant on it? (yes=1/no=0): 1
Field added!

Make a selection (0-7): 2

 === Adding a Field to the Planting Schedule ===

 You have 65.00 units of money

Enter the field coordinates (enter "0 0" to exit):1 1
Field (1, 1) has planting cost of 21.00
Would you like to plant on it? (yes=1/no=0): 4
Would you like to plant on it? (yes=1/no=0): 3
Would you like to plant on it? (yes=1/no=0): 0
Field not added!
```

The functionality for adding a field shall be implemented in a function

```
double addField( int fields[][BOUNDARY], double moneyLeft);
```

declared in `grove.h`. This function takes as input the array representing the current planting schedule and the current amount of money on hand. It produces, as a side effect the appropriate change in the planting schedule (altering the appropriate element of the `fields` array if needed), and returns the new amount of money on hand.

Removing a Field from the Planting Schedule

When the player selects option ”3” the program shall conduct the dialog for removing a field from the planting schedule.

In many respects, this operation is similar to the operation of adding a field, with a number if important caveats.

The field removal dialog starts with the following output printed by the program (after skipping a line):

```
=== Removing a Field from the Planting Schedule ===

You have <YYY.YY> units of money

Enter the field coordinates (enter "0 0" to exit):
```

There are two coordinate entry errors that the program needs to check for:

- **Invalid field.** Same as with adding a field, the coordinates must be between 1 and 20. If the field is invalid, the program shall print the same dialog as in the case of adding a field.
- **Field not in the planting schedule.** A field that is not in the planting schedule cannot be removed from the schedule. If a given field is not in the schedule, the program shall produce the following output:

  ```
  Enter the field coordinates (enter "0 0" to exit):2 2
  Field (2, 2) is NOT in the planting schedule

  Enter the field coordinates (enter "0 0" to exit):
  ```

If the user enters "0 0", the program shall return to main menu without making any changes to the schedule.

If the user enters valid field coordinates, **and** the field is in the current schedule, the program shall report the cost of planting the field (i.e., the amount of money to be recovered), and the overall amount of money the player will have after the change takes effect, and ask the player to confirm the removal of the field. The dialog shall look as follows:

```
Enter the field coordinates (enter "0 0" to exit):1 1
Field (1, 1) has planting cost of 21.00
After removing it you will have 65.00 units of money
Would you like to remove field (1 1)? (yes=1/no=0):
```

A "0" (**no**) answer shall result in the program returning to the main menu without making any changes in the schedule. The program shall output

```
Field not removed!
```

and print an empty line after it before displaying the main menu.

A "1" (**yes**) answer shall result in the following actions by the program:

- The field is removed from the planting schedule
- A new value of money on hand is computed (the cost of the removed planting is recovered)
- The message

  ```
  Field removed!
  ```

  is printed (with a new line below it) before the program returns to the main menu.

The functionality for removing a field shall be implemented in a function

```
double removeField( int fields[][BOUNDARY], double moneyLeft);
```

declared in `grove.h`. This function takes as input the array representing the current planting schedule and the current amount of money on hand. It produces, as a side effect the appropriate change in the planting schedule (altering the appropriate element of the `fields` array if needed), and returns the new amount of money on hand.

Display of a Current Planting Schedule

In response to the option `"1"` selected by the player from the main menu, the program shall print out the map of Grower's Grove showing for each field either "1", if it is in the current schedule, or "0" if it is not. Upon printing the map, the program shall print a new line and return to the main menu.

The output shall look as shown below (the planting schedule shown is $(1,1),(1,20),(2,2)$ and $(11,11)$:

```
Make a selection (0-7): 1
20 | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
19 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
18 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
17 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
16 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
15 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
14 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 5 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 4 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 2 | 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1 | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 --+-----------------------------------------------------------
   | 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Note. Please note the extra spaces in front of numbers $1-9$. Your output must look exactly the same.

Put the code to print a current planting schedule in the function

```
void printPlantings(int fields[][BOUNDARY]);
```

Displaying Maps of Planting Costs, Yields, Harvest Quality, Profit

Menu options `"4"` through `"7"` shall be handled in a similar way. Each of the four choices shall result in a Grower's Grove map showing the values of the appropriate quantity for each field printed. The output is not unlike what you saw in `print-all` programs in Labs 3,4 and 5, except that maps are better formatted (see below).

There are four different quantities for which the program shall print the maps. Each map shall be printed by a separate function from the list below. All functions are declared in `game.h`:

Function	Prints the map of
`void printPlantingCosts();`	planting costs
`void printYields();`	projected harvest yields
`void printQuality();`	projected harvest quality
`void printProfit();`	projected profit

The output (we use profit here) shall look as follows:

```
Make a selection (0-7): 7
20 | 234.66  498.79  256.75  538.12  157.93  546.16  146.11  533.88   47.31  515.58   46.18  497.43  141.80  481.14  152.97  472.77  163.47  498.92  201.07  156.36
19 | 504.87  268.92  552.56  274.42  561.55  160.78  544.52  141.58  523.12   39.33  510.81   53.82  512.23  161.72  528.82  182.57  587.66  234.18  287.71  188.63
18 | 268.92  557.51  287.27  568.36  170.00  542.93  155.30  508.64   45.45  491.75   45.75  504.93  161.20  548.42  192.77  649.41  261.82  335.48  220.62  418.25
17 | 552.56  287.27  570.43  282.16  536.38  165.55  483.81   50.01  448.29   13.30  457.43   54.75  519.71  193.23  663.19  278.73  366.84  246.45  488.34   58.13
16 | 274.42  568.36  282.16  532.95  170.42  460.78   54.42  397.71   37.69  383.49   40.83  443.75   68.85  612.23  281.86  368.32  262.09  537.54  151.71  326.43
15 | 561.55  170.00  536.38  170.42  452.84   56.25  358.90   40.67  206.48    2.82  223.16   53.51  499.77  107.52  216.67  264.71  548.29  160.36  361.42   45.49
14 | 160.78  542.93  165.55  460.78   56.25  216.47   42.27  187.51    4.10   67.06    8.79  249.43   87.32  191.12   99.66  508.83   63.40  373.76   46.66  273.14
13 | 544.52  155.30  483.81   54.42  358.90   42.27   64.44    3.81   21.94  -12.25   64.72   26.99   40.85   81.20  418.25   55.07  356.36   44.72  288.24   34.57
12 | 141.58  508.64   50.01  397.71   40.67  187.51    3.81   14.78  -12.80    5.24   -1.67  -12.97   23.83  219.99   20.40  198.86   37.96  285.01   17.65  232.79
11 | 523.12   45.45  448.29   37.69  206.48    4.10   21.94  -12.80  -10.48  -49.05  -56.90   -3.58   27.10    4.42  174.37   10.50  171.99   12.61  153.94   11.61
10 |  39.33  491.75   13.30  383.49    2.82   67.06  -12.25    5.24  -49.05  -81.19  -47.66    1.77  -14.08   49.93   -2.05  158.73    5.08  154.03    7.78  134.40
 9 | 510.81   45.75  457.43   40.83  223.16    8.79   64.72   -1.67  -56.90  -47.66  -12.58  -13.87   14.22   -1.59   54.24    6.58  149.46    9.87  137.31   10.33
 8 |  53.82  504.93   54.75  443.75   53.51  249.43   26.99  -12.97   -3.58    1.77  -13.87    8.12   -0.30   45.58    8.08  142.05   11.26  137.83   12.12  120.60
 7 | 512.23  161.20  519.71   68.85  499.77   87.32   40.85   23.83   27.10  -14.08   14.22   -0.30   42.22    8.39   52.77   12.69  135.96   14.26  124.31   13.24
 6 | 161.72  548.42  193.23  612.23  107.52  191.12   81.20  219.99    4.42   49.93   -1.59   45.58    8.39   50.82   13.95  133.13   15.51  124.89   15.04  107.31
 5 | 528.82  192.77  663.19  281.86  216.67   99.66  418.25   20.40  174.37   -2.05   54.24    8.08   52.77   13.95  132.68   16.69  125.01   16.04  109.98   14.11
 4 | 182.57  649.41  278.73  368.32  264.71  508.83   55.07  198.86   10.50  158.73    6.58  142.05   12.69  133.13   16.69  125.30   17.08  112.00   14.82   93.84
 3 | 587.66  261.82  366.84  262.09  548.29   63.40  356.36   37.96  171.99    5.08  149.46   11.26  135.96   15.51  125.01   17.08  111.64   15.72   95.13   13.39
 2 | 234.18  335.48  246.45  537.54  160.36  373.76   44.72  285.01   12.61  154.03    9.87  137.83   14.26  124.89   16.04  112.00   15.72   95.89   13.13   78.21
 1 | 287.71  220.62  488.34  151.71  361.42   46.66  288.24   17.65  153.94    7.78  137.31   12.12  124.31   15.04  109.98   14.82   95.13   13.13   78.94   10.76
 --+-----------------------------------------------------------------------------------------------------------------------------------------------------------------
   |      1       2       3       4       5       6       7       8       9      10      11      12      13      14      15      16      17      18      19      20
```

Pretty Print. All functions shall produce output in, what we call here, "pretty print". This means that each column of the map needs to be perfectly aligned on the right side (see above). To do so, you must "pad" the output of smaller numbers with a necessary number of spaces. For example, in the case above:

- positive numbers that are smaller than 10 need *two extra padding spaces* in front of them.
- positive numbers between 10(inclusive) and 100 (exclusive) need *one extra padding space* in the front.
- negative numbers greater than -10 need *one extra padding space* in the front.
- positive numbers starting at 100 and negative numbers between -100 and -10 do not need any padding.

Also, limit the `double` values printed out by all the functions to two decimal places.

Finalizing the Schedule

Option `"0"` of the main menu finalizes the planting schedule. At this point, the game shall produce final analytical output for the user and terminate.

Each type of analytical output is supported in `game.h` by a declared function. Essentially, the final step of the game is to call all the analytical functions, obtain all the values necessary, print the current schedule, print the values and end the game.

More, specifically, the following functions are declared in support of this part of the game:

Function	Purpose
`double getPlantingProfit(int fields[][BOUNDARY]);`	compute overall profit of a planting schedule
`double getRevenue(int fields[][BOUNDARY]);`	compute overall revenue of a planting schedule
`int countPlantings(int fields[][BOUNDARY]);`	compute number of fields in a planting schedule
`int findMaxDays(int fields[][BOUNDARY]);`	compute the longest number of days in a planting from the schedule
`void printFields(int fields[][BOUNDARY]);`	output to screen the list of fields in a planting schedule

When the user selects "0", the program shall, first compute the four quantities listed above: number of fields in the planting schedule, the duration of the planting schedule (the longest time to harvest among the fields from the schedule), the overall revenue and profit from the sale of *groveberries* using the functions specified above. Additionally, the program compute the total cost of the planting schedule. Then, the program proceeds as follows:

-

- **Outputing the planting schedule.** The program output the total number of fields in the planting schedule, followed by the list of fields in the schedule. This output shall be preceded by a "title bar" stating "Planting is Done!" (see below). This part of the output shall look as follows (the example is for a three-field planting schedule of $(1, 1), (20, 4)$ and $(4, 16)$):

```
 ========= Planting is Done! ===============

Planted 3 fields:
(1, 1)
(20, 4)
(4, 16)
```

- **Outputing other analytical information.** The program then prints an empty line, followed by printing the following four quantities:
  1. Total cost of the planting schedule.
  2. Days to wait for the full harvest.
  3. Overall revenue from the planting schedule.
  4. Overall profit from the planting schedule.
  5. Total amount of money on hand after the profit is realized.

  This is followed by two empty lines and a "Can you do better next time?" question, after which the program terminates.

Example. For a planting schedule of $(2, 2), (12, 2), (20, 2), (20, 3), (4, 5)$ and $(19, 19)$, the full output of the planting finalization stage shall look as follows:

```
Make a selection (0-7): 0
 ========= Planting is Done! ===============

Planted 6 fields:
(2, 2)
(12, 2)
(20, 2)
(20, 3)
(4, 5)
(19, 19)

    Total cost of planting: 132.00
   Days to wait for profit: 93
Revenue from the plantings: 1266.49
-------------------------------------
            Overall profit: 1134.49
       Total money on hand: 1152.49


Can you do better next time?
```

General Notes

Use of functions. You are given a set of 14 functions to implement. Implementing each function helps you keep your main gameplay code nice an clean - any time you need to execute a specific subtask, you call the requisite function and it does it for you.

My implementation has other functions in it that make it easier to achieve specific tasks. For example, the most *annoying* feature of the program is the *pretty print* of the four maps that output `double` values (pretty print of the planting schedule is not as bad, because all values are either 0 or 1 - take the same amount of space in a column). My implementation has functions that figure out how much to pad each individual value, and what the maximum amount of padding for a given map should be (it is based on the length of the largest value on the map).

You are allowed to declare and implement any functions above and beyond the ones declared in `game.h`.

Program size. My `play.c` file, which contains the `int main()` supporting the game play is around 100 lines of code. My `game.c` file which contains all functions: both those declared in `game.h` and my internal ones, is around 500 lines of code. Overall, this program is about 600-650 lines of code, which siginificant, but doable by two people in the timespan allotted.

Submission

What to submit. Submit the following files:

- Your program: `grove.h`, `grove.c`, `game.h`, `game.c`, `play.c`
- Your team: `README` file containing the names of the people on the team and their Cal Poly email addresses. The `README` file can also contain any other information regarding your submission: any bugs that were not fixed, any special features of the program, and compilation details.
- Test Cases. Submit a file `testcases.txt` containing a list of at least 10 test cases represented either in full or in compact form.

The `testcases.txt` file shall be created long before your program is completed. I strongly recommend that you first study this document, then create a list of test cases you will be using to debug/test your program, and then only start writing the program. Being able to produce correct test cases makes you understand much better what your program will be doing. To facilitate this, **please submit your `testcases.txt`** and a version of your `README` files **by Thursday, November 14, 23:59pm**.

All the code is due **Tuesday, November 19, 23:59pm**.

Use the following `handin` command when submitting (as usual, submit from one of the `unix` servers):

```
$handin dekhtyar program02 <files>
```

Good Luck!