Program 2: Space Trader text game...

Due date: Monday, February 22, beginning of the lab period.

Purpose

To write a program requiring use of loops, conditionals and arrays.

Programming environment. This is a solo programming project. You are responsible for all the work related to the program development, testing and submission.

Collaboration. **Any** collaboration between peers, as well as any collaboration with outside sources is **strictly prohibited**. If you have any questions, concerning the assignment, please consult the instructor.

Program Description

You will implement a text-based game called Space Trader. The user controls the activities of a space trader who can travel between five planets and buy and sell goods. The game keeps track of the amount of money the trader makes as well as of the resources (spaceship fuel) necessary for the trader to travel from planet to planet.

Instructor's program

Instructor's executable is found on the course web page on the page devoted to Program 2 materials.

Note: Before starting to implement the game, please read these instructions, figure out the game play, *download* instructor's executable, and play a few games. This will help you understand what is expected from you.

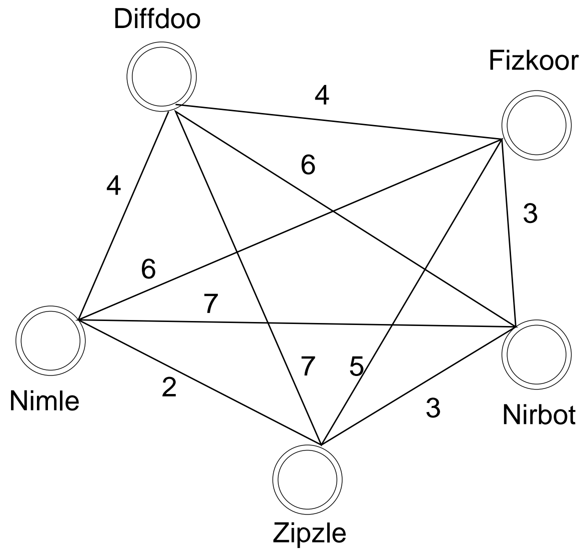

Figure 1: The Space Trader game map.

Program Requirements.

SR0. Implementation. Your program *shall be implemented as a single `int main()` function.* **No other functions** shall be present in your code.

SR1. Game Setting. You are an enterpreneurial space trader living in a world that allows for fast space travel between different planets. You own a spaceship that can transport you, and a certain amount of cargo from planet to planet. Your little corner of the Universe, consists of five planets named Diffdoo, Fizkoor, Nirbot, Zipzle and Nimle after the species that resides on them[1]

The schematic map of your corner of the Universe is show in Figure 1. Each planet has a trade route to all other planets. The distances between the planets, expressed in *light years*, are shown next to each route. Your spaceship, of course, has the ability to cover these distances faster than the speed of light, however, the travel time is still dependent on the actual distance (see below).

As you travel from planet to planet, you carry with you two types of resources: fuel, that allows your ship to travel, and gets burned in the process, and encyclopedia chips, which you can buy and sell. The five planets you are travelling between have a single currency: imus (Interplanetary Monetary Units).

The prices for the two resources will fluctuate over time. The fluctuations will be controlled by some mathematical apparatus we discuss below.

On each turn of the game you start at a specific planet and can:

- buy fuel;
- buy/sell encyclopedia chips;
- travel to different planet;

[1] The names are borrowed with kind permission from a series of assignments developed for IHS by Alyssa Daw. You've not seen the assignments yet, but you might. I figured, we'd reuse cool sounding alien names.

- stay at your current planet.

You start with some money and some fuel for your ship. Your goal is to make as much money as possible. You have 10 turns. Good luck!

S2. The program shall start by prompting the user to choose one of three actions: play the game, see the instructions or quit. If the user chooses to play the game, the space trader game proper starts. If the user chooses to see the instructions, the instructions are displayed. Finally, if the user chooses to quit the program, the program terminates. After instructions have been displayed to the user or after a game has been completed, the same menu of three choices (play, see instructions, quit) shall be offered again.

SR3. Main menu. Upon start, your program shall display the following text:

```
-------------------------------

     Space Trader, Version 1.0

1. Play
2. Instructions
3. Quit

Your choice:
```

In the rest of the document we refer to this text as the *main menu.* The program will then wait for user input.

SR4. Main menu input guard. If user inputs any number other than `1`, `2` or `3`, the program shall repeat the entire greetings/main menu text shown above, and wait again for user input.

SR5. Quit. If the user enters `"3"`, the program shall skip a line and output

```
Good bye!
```

After that, the program shall terminate.

SR6. Instructions. If the user enters `"2"` the program shall output the game playing instructions. The text of instructions is shown in Figure 2. After the instructions are printed, the program shall output the main menu and wait for user input.

SR7. Game. If the user selects `"1"`, your program shall start the gameplay.

SR8. Starting Conditions. The starting conditions of the game shall always be the same:

```
 -------- Instructions -------------

You are a space trader hoping to make a fortune selling encyclopedia chips.
In your travels you can visit five planets:
   --> the beautiful Diffdoo
   --> the sunny Fizkoor
   --> the technologically advanced Nirbot
   --> the unforgettable Zipzle
   and the mysterious Nimle
You have a spaceship, which can hold up to 100 units of encyclopedia chips
and up to 100 fuel cells.
As you travel, you burn fuel
At each stop, you can buy more fuel, and buy or sell encyclopedia chips

But beware! Fuel and encyclopedia prices can go up, just as they can go down!
You have 10 game turns to create a fortune for yourself! GOOD LUCK!
```

Figure 2: Space Trader: the instructions

Starting planet:	Diffdoo
Price of fuel:	10 imu/cell
Price of encyclopedia chips:	100 imu/unit
Starting fuel:	10 cells
Starting encyclopedia chips:	0 units
Starting funds:	1000 imu
Starting time:	day 0

Distances between planets. The distances between the planets, expressed in light years are shown in Figure 1. The distances are symmetric. (For completeness, the distance between a planet and itself is 0).

Note: Use a 2D array to store all the distances. You are allowed the array initialization statement which would set up all interplanetary distances *using magic numbers*.

Fuel and encyclopedia chip prices. At each moment of time, each planet will have its own price of fuel and its own price of encyclopedia chips. Your program shall keep track of all five current prices for each type of resource. At the beginning of the game, all prices are set to the values above. As the game progresses, they will be changing independently.

Your program shall perform all necessary assignments to set up a new game as described above After that, the main loop of the game shall commence.

SR9. Current state output. At the beginning of each step of the game (including the first step), your program shall output information about the current state:

- Step number (step 0, 1, 2, . . . 10).
- Number of days since the beginning of the game.
- Current planet.
- Current assets: money, fuel and encyclopedia chips.

```
--------------------------------------------
        Step: 0  of 10
Time Elapsed: 0  days
Current Location: Diffdoo

ASSETS:    Money: 1000.00 imu     Encyclopedia chips: 0 units    Fuel: 10 cells

MARKET PRICES: Encyclopedia chips: 100.00 per unit        Fuel: 10.00 per cell

TRAVEL DISTANCES: Fizkoor:4  Nirbot:6  Zipzle:7  Nimle:4
-----------------------------------------

Choose an action:
1. Buy encyclopedia chips
2. Sell encyclopedia chips
3. Buy fuel
4. Travel to another planet
5. Stay on this planet

Enter your choice:
```

Figure 3: Game status and action menu.

- Current fuel and encyclopedia chip prices on the planet you are in.
- Distances to other planets.

The format of the status report is shown in Figure 3 (it shows the status at the beginning of the game).

SR10. Game choice menu. After the game status is printed, your program shall output the list of possible game actions and ask the user to input an action. The menu is shown in Figure 3. There are five in-game actions:

1. Buy encyclopedia chips.
2. Sell encyclopedia chips.
3. Buy fuel.
4. Travel to another planet.
5. Stay on current planet (in a hotel).

SR11. User selects action. Upon printing `"Enter your choice:"` the program shall wait for the user to enter the desired action.

The program shall check the value of the entered action. If the value is outside the $1 - - - 5$ range, program shall simply reprint the status and the action menu (**SR9**, **SR10**).

SR12. Buying encyclopedia chips. Action "1" is the purchase of encyclopedia chips. The user can purchase as many encyclopedia chips at the current price on the current planet as the money the user possesses allows. The user can also purchase 0 chips — this action is analogous to cancelling the intent to

purchase chips. If the user has less money than the price of a single encyclopedia chip, selecting to purchase 0 chips will be the only action the user will be able to take.

Your program shall proceed as follows. First, it outputs the message

```
== Buying encyclopedia chips at XXX.YY per unit
```

where $XXX.YY$ represent the price of a single encyclopedia chip on the current planet (with precision of two decimal places). On the next line, the program shall output the prompt:

```
How many units?
```

After that, the program shall read the number of encyclopedia chip units the user wants to buy.

SR13. Buying encyclopedia chips: input guards Your program shall provide an input guard for the number of encyclopedia chip units to purchase. An `int` value is expected. If the entered number is negative, your program shall output

```
Cannot buy negative number...
```

and return to the `"How many units?"` prompt. If the entered number is zero, the program shall complete the encyclopedia chip purchase activity, and reprint the current status and the action menu (see **SR9** and **SR10**. If the entered number is positive, the program shall check if the user has enough money to purchase that many units.

If the user does not have enough money, the program shall print

```
Not enough money...
```

and return to the `"How many units?"` prompt.

SR14. Processing a purchase. If the user has sufficient money to purchase the desired number of units, your program shall process the purchase. For this the program shall:

- Update the total number of encyclopedia chips in user's posession.
- Compute the total cost of the newly purchased encyclopedia chips.
- Deduct the computed cost from the current amount of money the user has.

Once the purchase is processed, the `"Buy encyclopedia chips"` action is over. The program shall print the current status and the action menu (see **SR9** and **SR10**).

The action of purchasing encyclopedia chips **DOES NOT advance the step count**.

SR15. Selling encyclopedia chips. Selecting action ”2” allows the user to sell encyclopedia chips at the current price for the current planent. The user can sell any nubmer of encyclopedia chip units, as long as it does not exceed the total number of units the user has. Selling 0 units is analogous to cancelling the action.

Your program shall proceed as follows. First, it shall print the message

```
== Selling encyclopedia chips at XXX.YY per unit
```

where $XXX.YY$ is the current encyclopedia chips price on the planet. After that, the program shall print the prompt

```
How many units?
```

After that, the program shall read the number of units to be sold.

SR16. Selling encyclopedia chips: input guards Your program shall provide an input guard for the number of encyclopedia chip units to sell. An `int` value is expected. If the entered number is negative, your program shall output

```
Cannot sell negative number...
```

and return to the `"How many units?"` prompt. If the entered number is zero, the program shall complete the encyclopedia chip sales activity, and reprint the current status and the action menu (see **SR9** and **SR10**. If the entered number is positive, the program shall check if the user has enough encyclopedia chip units to sell.

If the user does not have this many units, the program shall print

```
You do not have that many...
```

and return to the `"How many units?"` prompt.

SR17. Processing a sale. If the user has enough units, your program shall process the sale. For this the program shall:

- Update the total number of encyclopedia chips in user’s posession.
- Compute the total cost of the sold encyclopedia chips.
- Add the computed cost from the current amount of money the user has.

Once the sale is processed, the `"Sell encyclopedia chips"` action is over. The program shall print the current status and the action menu (see **SR9** and **SR10**).

The action of selling encyclopedia chips **DOES NOT advance the step count**.

SR18. Buying fuel. Action "3" is the purchase of fuel. Fuel is necessary for travel between planets. Each flight spends a certain amount of fuel. When the fuel runs out, the user should purchase more before (s)he can travel.

The user can purchase as much encyclopedia chips at the current price on the current planet as the money the user possesses allows. The user can also purchase 0 fuel cells — this action is analogous to cancelling the intent to purchase fuel. If the user has less money than the price of a single fuel cell, selecting to purchase 0 fuel cells will be the only action the user will be able to take.

Your program shall proceed as follows. First, it shall output the message

```
== Buying fuel at XX.YY per unit
```

where $XX.YY$ represents the price of one fuel cell on the current planet (with precision of two decimal places). On the next line, the program shall output the prompt:

```
How much fuel?
```

After that, the program shall read the number of fuel cells the user wants to buy.

SR19. Buying fuel: input guards Your program shall provide an input guard for the number of fuel cells to purchase. An `int` value is expected. If the entered number is negative, your program shall output

```
Cannot buy negative number...
```

and return to the `"How many units?"` prompt. If the entered number is zero, the program shall complete the fuel cell purchase activity, and reprint the current status and the action menu (see **SR9** and **SR10**. If the entered number is positive, the program shall check if the user has enough money to purchase that many fuel cells.

If the user does not have enough money, the program shall print

```
Not enough money...
```

and return to the `"How many units?"` prompt.

SR20. Processing fuel purchase. If the user has sufficient money to purchase the desired number of fuel cells, your program shall process the purchase. For this the program shall:

- Update the total number of fuel cells in user's posession.
- Compute the total cost of the newly purchased fuel cells.
- Deduct the computed cost from the current amount of money the user has.

Once the purchase is processed, the `"Buy fuel"` action is over. The program shall print the current status and the action menu (see **SR9** and **SR10**).

The action of purchasing encyclopedia chips **DOES NOT advance the step count**.

```
 == Travelling to another planet
Select a planet:
0. Stay on Diffdoo(0 light years; 0 fuel cells to travel)
1. Move to Fizkoor(4 light years; 5 fuel cells to travel)
2. Move to Nirbot(6 light years; 7 fuel cells to travel)
3. Move to Zipzle(7 light years; 8 fuel cells to travel)
4. Move to Nimle(4 light years; 5 fuel cells to travel)
Where to?
```

Figure 4: Travelling choices (action "4" of the game).

SR21. Travel to another planet. The key to earning money in this game is travel to different planets. As the user travels, time elapses, fuel gets spent, and both fuel and encyclopedia chip prices on each planet change. Action "4", `"Travel to another planet"` allows the user to pick the planet to travel to. Once the planet is picked, if the user has enough fuel to travel to it, the travel happens, and the state of the game gets updated.

This action proceeds as follows. First, the program shall print the text:

```
 == Travelling to another planet
Select a planet:
```

Next, your program shall print the list of the planets the user can travel to. For simplicity, all five planet, including the current planet will be shown as choices. The use can cancel interplanetary travel by selecting the current planet.

The planets are output in the order `0.Diffdoo`, `1.Fizkoor`, `2.Nirbot`, `3.Zipzle` and `4.Nimle`. For each planet, your program shall specify whether the user would be moving to it or staying at it (for the current planet), and provide, in parentheses the distance to the planet in light years and the amount of fuel necessary to travel.

After the list of planets to travel is printed, the program shall output the

```
Where to?
```

prompt.

The sample output (travelling from `Diffdoo` on step one of the game) is shown in Figure 4.

SR22. Fuel to travel. The amount of fuel it takes to travel from a planet to planet is determined as follows. The amount of fuel depends on the distance between the planets of the weight of the ship, which we represent as the number of encyclopedia chip units on board. The formula is:

$$fuel = distance \cdot \left(1 + \frac{cargo}{10}\right) + 1.$$

Here *distance* is the distance between the two planets and *cargo* is the number of encyclopedia chip units the user has. Fuel price is a floating point number.

SR23. Travel. Input guards. The program shall read the value indicating the planet to travel it. An `int` value is expected. If the number entered at the

`"Where to?"` prompt is not between 0 and 4, then the program shall simply repeat the `"Where to?` prompt.

If the planet selected for the travel is the same as the current planet, then the interplanetary travel action is completed, the user stays on the same planet, and **the step of the game remains the same**. The program shall print the following message:

```
Staying on the same planet. Fine...
```

After that, the program shall output the status screen and the action menu (see **SR9** and **SR10**).

SR24. Not enough fuel. If the user selects to travel to a different planet, the program shall check if the user's ship has enough fuel on board to reach the planet. If there is not enough fuel, the program shall output the following message:

```
Not enough fuel to get there
```

and shall return back to the `"Where to?"` prompt.

SR25. Processing travel. If the user chooses one of the four planets to travel to (not the current planet), and there is enough fuel to reach the planet, the program shall perform the following actions.

- **Advance to the next step** of the game.
- Adjust the amount of fuel on board of the ship by subtracting the amount of fuel needed to reach the planet from the current amount of fuel on board.
- Compute the number of days in travel. The number of days in travel depends on the distance between the two planets and the amount of cargo (encyclopedia chips) carried by the ship. The formula is:

  $$travelTime = 2 \cdot distance \cdot \left(1 + \frac{cargo}{10}\right) + 1$$

  Here, *distance* is the distance between the two planets, and *cargo* is the number of encyclopedia chip units the user possesses. Travel time is measured in days and it is an `int` value.
- Update current time, by adding the travel time to the number of days elapsed on the previous step.
- Update fuel prices **on all planets**. Fuel prices are subject to fluctuation, and can go up or down. The formula for computing the fuel price on a specific planet on a specific day is:

  $$fuelPrice = 10 + 8\cos\left((5 - p) \cdot time\right).$$

  Here, 10 is the starting price of the fuel on all planets on day 0 of the game, *time* is the number of days elapsed since the beginning of the game, and p is a planet code from the table below:

Planet	Code for price computations
`Diffdoo`	0
`Fizkoor`	1
`Nirbot`	2
`Zipzle`	3
`Nimle`	4

Fuel price is a floating point value.

Note. According to the formula above, the price of fuel can fluctuate from 2 to 18 imu per single fuel cell.

- Update encyclopedia chip prices **on all planets**. Encyclopedia chip prices also fluctuate depending on the number of days elapsed, and the fluctuation patters are different for each planet. The exact formula is:

$$cargoPrice = 100 + 50 \cdot \sin\left((p + 1) \cdot time\right).$$

Here, 100 is the initial price of one unit of encyclopedia chips on all planets (50 is half that price), $time$ is the number of days elapsed since the beginning of the game, and p is the planet code from the table above (same code as used in fuel price computation).

Encyclopedia chip price is a floating point value.

Note. According to the price formula, the price of one unit of encyclopedia chips can be as low as 50 imu and as high as 150 imu.

- Set the current planet to be the destination planet of the trip.

After all these activities are complete, the **"Travel to another planet"** action is over. The program shall display the current state of the game, and the menu of in-game actions (see **SR9** and **SR10**).

SR26. Staying on the same planet. The last action available to the user is to stay for a few days on the current planet. The stay requires renting a hotel room, which **always costs** 5 imu per night. The user can select how many days to stay on the planet. When the user selects to stay more than one day, **this action advances the step of the game by 1**. r **Note.** This action is different than selecting action 4 (travel to another planet) and then choosing to stay on the current planet. The latter action advances neither the game clock, nor the number of steps in the game. Selecting action 5 (stay on this planet) and then entering 0 nights yields the same result, but entering any other number will advance the step of the game, the number of days elapsed and update the user's cash on hand. Also, since the number of days elapsed changes, fuel and encyclopedia chip prices will be updated as well.

When action "5" is selected, the program shall behave as follows. First, the program shall print the following message:

```
 == Staying on XXXXXXX
Hotel charge: 5 imu/night
```

Here `"XXXXXX"` is the name of the current planet. After that, the program shall print the

```
How many nights?
```

prompt and wait for user input.

SR27. Staying on the same planet. Input guards. The program shall expect an `int` value. Once the value is obtained, the program shall perform the following checks on this value:

- entered number of nights to stay is not negative. If the input is negative, the program shall respond with the following message

  ```
  Cannot do negative number of nights...
  ```

  and shall go back to the `"How many nights?"` prompt.

- the user has enough money to pay for the entered number of nights. If that is not the case, the program shall print the following message

  ```
  Not enough money...
  ```

  and shall go back to the `"How many nights?"` prompt.

SR28. Processing the hotel stay. If the input is valid, and the user has enough money to pay for the requested hotel stay, the program shall take the following actions:

- **Advance to the next step** of the game.
- Update current time, by adding the number of nights to stay at the hotel to the number of days elapsed on the previous step.
- Update fuel prices **on all planets**. See **SR25** for the formula.
- Update encyclopedia chip prices **on all planets**. See **SR25** for the formula.
- Keep the current planet as the planet the user is on.

After all these activities are complete, the **"Stay on this planet"** action is over. The program shall display the current state of the game, and the menu of in-game actions (see **SR9** and **SR10**).

SR29. Duration of the game. The game starts on step 0. The game shall last until **step 10** is complete. Upon completion of step 10, the game ends, and the program proceeds to execute actions described in the next requirement (**SR30.**).

SR30. End of game. After the end of step 10, the user will reside on one of the five planet, and will have in his/her posession some money, some number of unsold encyclopedia chip units, and some number of fuel cells. The planet the user is on will have (computed on the last step via either **SR25** or **SR28**) specific prices for encyclopedia chips and for the fuel.

At the end of the game, your program shall do the following:

- compute the monetary value of the unsold encyclopedia chips.
- compute the monetary value of the on-board fuel cells.

- compute the monetary value of the total user assets (money plus encyclopedia chips plus fuel).
- compute the percent of increase/decrease of the user's assets. Recall that the user started with 1000 imu and 10 fuel cells worth 10 imu each.
- determine if the user has suffered a loss, or achieved a gain. (Note: treat no gain as a gain).

SR31. End of game. Output. Once your program performs all necessary end of game computations, it shall output the following information.

- A `... AND THE GAME IS OVER ...` message.
- The number of days elapsed since the beginning of the game.
- The final planet destination of the user.
- User's assets:
  - Encyclopedia chips: number of units on board, current planetary price and total monetary value.
  - Fuel: number of fuel cells on board, current planetary price and monetary value.
  - Cash on hand assets.
- A message specifying the percent of the money the user lost/gained comparing to the initial total assets of the user. Please, remember that if the final total assets form, say, 140% of the initial total assets, then the gain achieved in the game is $140 - 100 = 40\%$. Similarly, if the final assets are, say, only 75% of the initial assets, then you lost $100 - 75 = 25\%$ of the money.

After the above output is printed, the program shall print two empty lines.

The specific format for the output (two cases: one for the winning and one for the losing messages) is shown in figure 5

SR32. Back to main menu. After the game is over and the results of the game are shown to the user (see **SR31**, the program shall return to the main menu (see **SR3**).

This completes the list of requirements for the program.

Design notes.

Legacy. The overall organization of this program is remniscent of the many legacy text-based games, and other text-based programs that operated for years and years (and some – still operate) on text-based terminals. Such programs usually consisted of a main menu that allowed the user to select what (s)he wanted to do with the program by entering a number, and the code that combined interaction with the user with the computations necessary to perform the tasks/play the game.

```
... AND THE GAME IS OVER...
after 25 days...:
Planet: Zipzle
ASSETS:
   Encyclopedia chips: 1 units at 74.68 imu per unit = 74.68 imu
   Fuel: 2 cells at 17.72 imu per cell = 35.44 imu
   Money: 850.00 imu
-------------------------------------------------------------------------
Total Assets: 960.12 imu
You lost 12.72% of your money. Better luck next time!
```

```
... AND THE GAME IS OVER...
after 106 days...:
Planet: Nimle
ASSETS:
   Encyclopedia chips: 0 units at 140.06 imu per unit = 0.00 imu
   Fuel: 6 cells at 15.49 imu per cell = 92.95 imu
   Money: 1354.42 imu
-------------------------------------------------------------------------
Total Assets: 1447.37 imu
You earned 31.58%! GOOD JOB!
```

Figure 5: End of game reports for a losing (top) and winning (bottom) games.

While most of the programs today use different means of user interaction, and while current graphical user interfaces make the legacy text-based interactive programs obsolete, it is well worth understanding the design and internal organization of such programs.

Overall organization. Essentially, your program should consist of two main loops. The outter loop is responsible for the main menu and reactions to the main menu choices. The reaction to the `"Quit"` choice is to leave the loop and terminate the program. The reaction to the `"Instructions"` choice is the print some text and start a new iteration of the outter loop. The inner loop of the program will appear in the code responsible for the game play. Each step of the game play loop covers one user action (which may or may not advance the step of the game).

Data. Your program shall use one-dimensional arrays to store current fuel and encyclopedia chipp prices. It shall also use a 2D array to store interplanetary distances. The latter array shall be initialized at the beginning of the program. You are allowed to use magic numbers when initializing the distances array. **USE OF ALL OTHER MAGIC NUMBERS IN THE PROGRAM IS PROHIBITED!** Use `#define` to define all constants and give your constants meaningful names!

Planet names. I recommend dealing with planet names as follows. Create five constant declarations that look as follows:

```
#define PLANET_ONE "Diffdoo"
```

(feel free to you any constant names). Whenever a name of a planet needs to be printed, run a `switch` statement on an integer experession that has the index of the current planet. Use the constants above in the `printf()` statements inside the `switch`.

Code length. For your information, the instructor's version of the program is around 450 lines long (with some empty lines and lines that only contain comments).

Submission Instructions

Submission.

Program name. Name your program `trader.c`.

Files to submit. You shall submit the `trader.c` file.

Testing. We will test your programs manually. That is, we will create a number of scenarios and will run your implementation on them. We will compare the results of your program to the results of the instructor's program.

Submission procedure. You will be using `handin` program to submit your work.

Section 01:

```
> handin dekhtyar-grader program02-01 trader.c
```

Section 09:

```
> handin dekhtyar-grader program02-09 trader.c
```

Late submission. You may submit late for a 24-hour period following the deadline. Late submissions are subject to the standard 10%—30% penalty at the instructor's discretion.