Files to submit: **all .c and .h files that make up your program hoarding.out. A Makefile to compile those files into hoarding.out**
Time it took Matthew: **3.5 hours**

Requirements

- Program must compile with both -Wall and -Werror options enabled
- Submit only the files requested
- Use doubles to store real numbers
- Print all doubles to 2 decimal points unless stated otherwise
- **NO** global variables allowed
- You **MUST** submit at least two .c files and one .h file
- You **MUST** use structs in your solution

Description

You will be implementing a simplified version of Monopoly called hoarding. The basic premise of the game is as follows: players take turns rolling dice to move themselves around the board, buying spaces as they go. If a player lands on a space another player owns they pay the owner a fee for landing there. The game ends when a sufficient number of players have been eliminated from the game or the turn limit is reached.

Details

Setting up the game

Your program will be provided three command line parameters in this order

1. The name of the file containing the rules
2. The name of the file containing the board
3. The name of the file containing the random numbers to use

Board File

The board file is a .csv file. A .csv file is simply a text file where the values are separated by commas (csv = comma separated values). By default your computer will likely open the file with Excel and this might be a bit confusing. If you want to see what the file really looks like open it with a text editor such as WordPad.

You will need to use the fact that the values are separated by commas to help you parse out the fields of the file. You may or may not find the function strtok helpful in solving this problem. I did not use strtok in my solution.

Format

The board file is structured as follows. Literal constant values are shown in *italics*

- *Number of spaces,* the number_of_spaces in the file
- Two blank lines
- A line for the headers which have the following format
  - *Type, Set ID, Intraset Id, Name, Property Cost, House Cost, Hotel Cost, Rent with House, Rent with Hotel*
- The actual spaces of the board
  - Each space is connected to the space next to it with the last space connecting back to the first space

There are two types of spaces

1. GO space
2. Property Space

GO spaces have the following format

- GO, amount of money to collect when passing go, Name of Space
- The GO space will always be the first space in the board

Property spaces have the format as described by the header field. Here's what each field means

- Type: either Go for a GO space or Property for a property space
- Set ID: Which set this property belongs to. Set's are the same thing as color groups in Monopoly but we are using numbers instead of colors.
  - Set Ids start at 0 and increase sequentially
- Intraset Id: The position of the property within the set. When printing out properties within a set the intraset id is used to figure out what order the properties should be printed. Properties within a set are shown in ascending order based on their intraset id
  - Intraset ids begin at 0 and increase sequentially within the set
- Name: The name of the property
- Property Cost: How much it costs to purchase the property
- House Cost: How much it costs to purchase a house on this property once you own all the properties within the set
- Hotel Cost: How much it costs to upgrade a group of houses to a hotel
- Rent with House: How much money this property makes when there is a single house on it
- Rent with Hotel: How much money this property makes when there is a hotel on it

Example of a Board File

- `Number of Spaces,10,,,,,,,,`
- `,,,,,,,,,`
- `,,,,,,,,,`
- `Type,Set Id,Intraset Id,Name,Property Cost,House Cost,Hotel Cost,Rent,Rent with House,Rent With Hotel`
- `Go,400,MU,,,,,,,`
- `Property,0,0,Kemper A,500,50,50,5,50,2000`
- `Property,0,1,Kemper B,1000,50,50,10,75,2500`
- `Property,1,0,Bainer A,2000,200,200,20,100,3000`
- `Property,1,1,Bainer B,2500,200,200,25,150,3000`
- `Property,1,2,Bainer C,3000,200,200,30,200,3500`
- `Property,2,0,SocialSci A,4000,400,400,40,300,4000`
- `Property,2,1,SocialSci B,4500,400,400,45,400,4000`
- `Property,2,2,SocialSci C,5000,400,400,50,500,4000`
- `Property,2,3,SocialSci D,5500,400,400,55,600,4500`

Rule File

When people play Monopoly they sometimes play with special house rules than are different from the normal standard rules. The rules file allows us to customize certain aspects of how the game plays.

Format

Literal constant values in *italics*.

- *Starting Cash:* starting value
- *Turn Limit (-1 for no turn limit):* turn limit
- *Number of Players Left To End Game:* number of players
- *Property Set Multiplier:* multiplier
- *Number of Houses Before Hotels:* number of houses
- *Must Build Houses Evenly:* yes or no
    - Yes and no can be lowercase or uppercase or some mixture of the two
- *Put Money In Free Parking:* yes or no
    - Yes and no can be lowercase or uppercase or some mixture of the two
- *Auction Properties:* yes or no
    - Yes and no can be lowercase or uppercase or some mixture of the two
- *Salary Multiplier For Landing On Go:* multiplier

Explanation of Fields

- Starting Cash: How much money each player starts the game with
- Turn Limit (-1 for no turn limit): How many dice rolls in **total** before the game ends

- Number of Players Left To End Game: When the number of players left reaches this value the game ends
- Property Set Multiplier: How much rent doubles on unimproved properties once the entire set is owned
- Number of Houses Before Hotels: How many houses the player must have on a property before the houses can be upgraded to a hotel
- Must Build Houses Evenly: Whether or not the user must build their houses evenly across properties in the set
- Put Money In Free Parking: Whether money paid to the bank should be put in free parking
- Auction Properties: If a user declines to buy a property whether it should be auctioned off or not
- Salary Multiplier For Landing On Go: How much a player’s salary is multiplied if they land exactly on Go

Which fields we will be using this time

Since this is our initial build we will not be implementing all of the options above. The list below shows which rules to pay attention to now. We will implement the remaining rules in future projects and probably add some new ones along the way. For the rules we aren’t using you can just store their values.

- Starting Cash: Using
- Turn Limit (-1 for no turn limit): Using
- Number of Players Left To End Game: Using
- Property Set Multiplier: Using
- Number of Houses Before Hotels: Not using
- Must Build Houses Evenly: Not using
- Put Money In Free Parking: Not using
- Auction Properties: Not using
- Salary Multiplier For Landing On Go: using

Parsing the Fields

Don’t forget that inside scanf you can put literals in the format string and it will only read those values. For example `scanf(“Starting Cash:”);` is valid and will read those characters from the input without storing them anywhere. Same holds for fscanf.

Example of a Rule file

Starting Cash: 1500
Turn Limit (-1 for no turn limit): 10
Number of Players Left To End Game: 1

Property Set Multiplier: 2
Number of Houses Before Hotels: 4
Must Build Houses Evenly: Yes
Put Money In Free Parking: No
Auction Properties: No
Salary Multiplier For Landing On Go: 1

Random Number File

The random number file is used to take random numbers from so we don't have issues with people having different random number generators and getting different answers because of that. The format of the file is simple just a single random number per line. Each random number is between 0 and 1000.

Continuing set up

After reading in all of the files you should ask the user how many players will be playing the game. Each player starts with the starting amount of cash specified in the rules file and 0 properties. Each player's "name" is simply a number from 0 to the number of players - 1. Play begins with Player 0.

Player Turn

At the beginning of the players turn they may choose between the following actions:

- Roll the dice
- Inspect player
- Leave the game

After rolling the dice and resolving the effect of landing on the space the user has the following options to choose from if they are still in the game

- End turn
- Inspect player
- Leave the game

A player can inspect a player as many times as they wish during their turn.

Rolling the dice

During their turn a player roles two separate 6 sided dice. The player then advances their token to the new space. Everytime the player passes the GO space they collect their salary

(Be careful here as it is possible on small boards to go around the . Depending on the space they land on one of the following things happens

- The player lands on the GO space
  - Their salary is multiplied by the salary multiplier found in the rules
- The player land on an unowned property
  - The player is offered the opportunity to purchase the property for its cost if they have enough money to buy it
- The player lands on a property they own
  - Nothing happens
- The player lands on a property another player owns
  - The player pays the owner the rent price

Inspecting a Player

A player may inspect themselves or another player to find out how much money they have and which properties they own.

Format

```
Player Name
  Cash: $How much money they have
  Properties owned
    0: Properties owned in property set 0
    1: Properties owned in property set 1
    2: Properties owned in property set 2
    …
```

Properties within a set are displayed in order based on their intra set id

Example

Let’s say that Player 0 has $56 and owns Kemper B, Bainer A, and Bainer C. If we displayed Player 0 then we would see

```
Player 0
Cash: $56
  Properties owned
    0: Kemper B
    1: Bainer A Bainer C
    2:
```

Leaving the game

Monopoly can be a long game so a player can at any time choose to leave the game. If they do choose to leave the game all of their properties and cash go back to the Bank.

Calculating Rent

The cost of rent for a property is given in the board file. If a player owns all of the properties within a set then rent is multiplied by the Property Set Multiplier found in the rules file. If a player does not have enough money to pay the rent they go Bankrupt and give all of their money and properties to the player that owned the property.

Ending the Game

The game ends when either the turn count is reached or the number of players is at or below the number of players needed to end the game. A “turn” is considered to be a player ending the turn after rolling the dice or leaving the game. This means that not every player may get the same number of rolls.

Deciding on a Winner

If there is more than 1 player remaining at the time the game ends then the winner is the person with the highest net worth. Net worth is the sum of all of the players cash plus the purchase price of all of the properties they own. If multiple people have the same net worth then they are all winners. Winners are declared in ascending order (Player 0, Player 1, Player 2, …).

Input

Input will always be valid and there is no need to validate it.

Potential Problem Areas

One issue you may run into when defining your structs is that of circular reference. A circular reference happens when you have two structs that both reference each other. This is a problem in C because C is a one pass compiler and it will see the reference to the second struct before it has been defined or declared. For example

```
     typedef struct AStruct{
          B* bp; // what is a B? Compiler hasn’t seen the definition
yet
     }A;

     typedef struct BStruct{
          A* ap;
     }B;
```

Here A refers to B before B has been defined or declared but rearranging the order of the definitions leaves us with the same problem. This problem can be solved by giving a forward declaration of the struct before using it.

```
     typedef struct BStruct B; // problem solved

     typedef struct AStruct{
          B* bp;
     }A;

     struct BStruct{
          A* ap;
     };
```

Now the compiler knows about the type B before you use it. Even with a forward declaration you cannot do something like the following

```
     typedef struct BStruct B; // forward declaration

     typedef struct AStruct{
          B bp; // now bp is a B and not a B*
     }A;

     struct BStruct{
          A ap; // now ap is an A and not an A*
     };
```

Now you have a circular definition. An A contains a B but a B contains an A which also contains a B, which contains an A, and so on and so forth until the end of time. Since each one contains the other you get caught in a circle. Now one could contain the other and the other could contain a pointer back to the first. That is ok. For example

```
     typedef struct BStruct B; // forward declaration
```

```
typedef struct AStruct{
     B bp; // now bp is a B and not a B*
}A;

struct BStruct{
     A* ap; // now ap is back to an A*
};
```

One thing you do to alleviate some of these problems it put all of your struct definitions and declarations in a single file and basically include it everywhere. This post on stack overflow describes that and other methods to resolve this problem.

You might be curious how you could ever run into this problem in your homework so here is a quick example. A Player owns Properties but a Property might have a reference to its owner, which is a Player. So now you have a circular reference.

Hints

- You will want to use structs. Structs make this problem doable. Without structs you will die.
- If you've never played Monopoly I highly recommend doing so. You can play a free version of the game at http://en.gameslol.net/monopoly-1122.html
- Really focus on breaking this problem down into small easy to do bite sized steps. Make use of functions for each of these steps. It will make your life so much easier. I've included the main gameplay loop to show this off and get you started on the right path

Example

The output for this program is quite large so instead of showing you everything what I've done. Is given you a copy of the executable on Kodethon as well as files for you test and play with. If you are on Kodethon the way of running the file

- First use the terminal to cd into the Oracle directory
- Once there the command to run the program is ./hoarding.out rules_file board_file random_file
    - Each of the corresponding type of file is located within the appropriately named directory
        - Rules are in the Rules directory
        - Boards are the Boards directory
        - The random number files are in the RandomFiles directory

Some examples of calling the program given that you are in the Oracle Directory on Kodethon

- ./hoarding.out Rules/ContinentRules.txt Boards/ContinentBoardNoSalary.csv RandomFiles/RandomNums10000.txt
- ./hoarding.out Rules/CampusRules.txt Boards/CampusBoard.csv RandomFiles/RandomNums10000.txt