CS 220 - Final Project

Team Code Submissions Due: ~~Thursday, December 6th by 11pm~~
Friday, December 7th by 11pm
NOTE: Individual Contributions Submissions are due ~~Saturday, December 8th by 11pm~~
Sunday, December 9th by 11pm
This assignment is worth 120 points.

Learning Objectives
This homework will give you practice with writing C++ classes using inheritance, dynamic binding, pure virtual functions, and abstract base classes. You will use git for version control.

Collaboration
This project is to be completed in pre-registered teams of three students. No collaboration is permitted outside a team. It is your responsibility to work effectively as a team. Each student will be required to submit a feedback file once the project has ended detailing what contributions each student made to the project (see Contributions Submission section below).

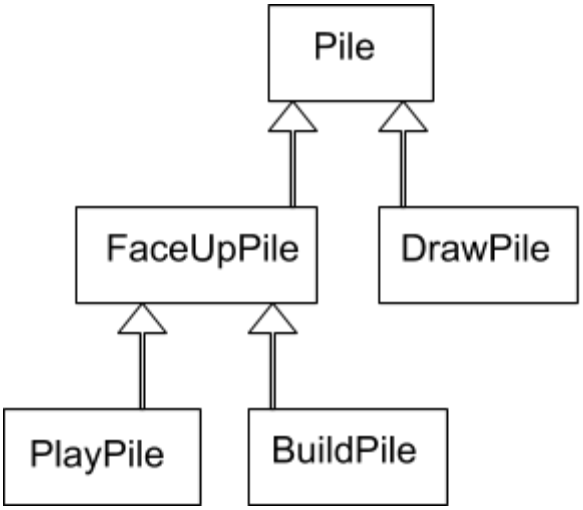
Starter Code
This homework comes with starter source files found in the public repository under the folder named `final-project/`. Teams should ensure they have a proper copy of these files before they start writing any code.

Problem Description
For this final project you will work in teams to write a program for the card game "Skip-Bo". See Mattel for official rules and this site for a helpful explanation. There are also several videos explaining the game, and mobile versions you can try out for yourself.

There are several variations of the game. You will be implementing a multi-player version, for between 2-6 players (no partner play). You also don't have to implement scoring - just one game played through from start to a winner. There are specific requirements (below) regarding game initialization, saving and loading that you must follow so that we can do some automated testing on your project. Make sure you read all the details below and use the provided code!

Object-Oriented Design
The diagram below gives a suggested starting point for the inheritance-related classes you should write to represent the various types of Card collections for the game. PlayPile can be used to represent stock piles and also discard piles. You'll also need to figure out where a Hand class fits in.

Think about how each of these classes is used in the game, adding necessary functions to support their uses (and any data), but only in the classes that absolutely need them. For example, only the DrawPile gets shuffled. Also remember to differentiate how they each get displayed. Take advantage of inheritance as much as possible without abusing it.



Skeleton Code

We are providing you with the following starter files:

1) `Card.h/.cpp` - contains Card class, which has:
    a) `private int value`,
    b) constructor `Card(int v)`
    c) `int getValue() const`
    d) `std::string toString() const` // for output to file
    e) `void display() const` // how to see in game play

2) `Pile.h/Pile.cpp` - contains
    a) `private vector<Card> pile;`
    b) `public virtual addCard(const Card& c);` // puts c at "top", regardless of Pile type
    c) `public toString() const;` // outputs entire contents of Pile, needed for state saving. You must use this in order to create consistent saved states that we can use for testing and autograding.
    d) You'll need to write `void readIn(istream &)`, to perform the inverse of toString in a sense -- make the Pile's contents match the state given by a particular stream.
    e) `public virtual void display() = 0;` // outputs the view of this Pile to cout as appropriate for Pile subtype during game play, but this will need to be implemented in derived classes. We've provided the necessary pieces of those classes also, to insure that your interface is exactly the same as ours.

i) `DrawPile::display()` will only show size of Pile, no actual Card values
ii) `FaceUpPile::display()` will display the top card value using `Card::display` and the pile size
iii) `Hand::display()` will show all cards and "--" for missing cards

3) `Player.cpp` – We have provided `toString` and `display` operations for consistency with gameplay and state saving. Do not modify these functions as we will use them for autograding. You'll need to figure out what to put in `Player.h` in order to support this code, as well as everything else the class needs to handle. You'll need to write `void readIn(istream &)`, to perform the inverse of toString in a sense -- make the Pile's contents match the state given by a particular stream.

4) `SkipBoGame.cpp` - We have provided `toString` and `display` operations for consistency with gameplay and state saving. Do not modify these functions as we will use them for autograding. You'll need to figure out what to put in `SkipBoGame.h` in order to support this code, as well as everything else the class needs to handle. You'll need to write `void readIn(istream &)`, to perform the inverse of toString in a sense -- make the Pile's contents match the state given by a particular stream.

5) `main.cpp` - is your main driver that collects one of 2 different sets of command line args depending on whether a new game is being started, or a saved game is being reloaded. In both cases, the first argument will be "true" or "false" indicating whether the draw pile should be shuffled (true, for regular (random) game play) or not (false, for testing purposes).
    a. New game args will be *shuffle, numPlayers, stock pile size, deck start file*. These values should be used to construct a new SkipBoGame object. Players should always be named Player0, Player1, etc. The deck start file is a plain text file with an ordering of all the cards in the deck. SkipBo wildcards will be represented with a 0 in these files. Our provided standard deck0.txt file should be used as a default. Once the game is initialized, play should begin and continue until it is interrupted and saved, or there is a winner.
    b. Reload game args will only be *shuffle* and *savedGame.txt* file which contains a complete saved game state. Once loaded, play should continue until it is interrupted and saved, or there is a winner.

   See sampleMainErrors.txt below (in Sample Runs) for a full list of main command-line error conditions that must be handled, and how.
   See sampleLoad.txt below (in Sample Runs) for a demonstration of loading a saved game state.

6) Game play - user interface: (see Sample Runs section below)

a. First Player in a new game: In shuffle==true mode, the first player should be chosen randomly. In test mode, Player0 should always have the first turn. In a reload situation, the savedGame file will determine whose turn it is.
b. Game display: display the game to start, then refresh the display after every player move and when continuing to the next player's turn.
c. Player turns: turn will always pass to the next player (+1) in a round-robin fashion. At the start of every turn, cards from the top of the draw pile will automatically be added to a Player's hand as needed so that they have 5. When the game begins and at the end of every turn (when the current player moves a card from their hand to a discard pile), check to see if the players will continue to **(p)lay, (s)ave** the game, or simply **(q)uit** playing this game without saving. When saving game state: player will need to input the file name for the saved game.
d. While a Player is taking their turn which may include multiple moves, two options must be available: **(m)ove** a card from one pile to another or **(d)raw** enough cards to fill hand. The draw option is only legal if the hand is empty. After each move check to see if that player has won (stock pile is empty). If so, the game should end and the winner be announced by name. Please see Piazza posts @763, @762, @750 regarding the filled Build Piles and how to manage setting them aside. Also see sampleBuild.txt bullet in Sample Runs below.
e. Move input: Player cards/piles will be numbered according to the scheme below and the 4 shared build piles will be indicated by letters a-d. Players will indicate which card to move and where to put it by these numbers and letters:
    i. 0 = stock pile (can only go to a build pile)
    ii. 1-4 = discard piles (from discard to build or from hand to discard)
    iii. 5-9 = cards in hand (to build or to discard)
    iv. a-d = shared build piles, in left to right order

    A sample move would be "m 0 c" to move from the player's stock pile to build pile c. Another example is "m 6 3" to move a card from the hand to a discard pile, ending the player's turn. You must check that each requested move is legal and reject it otherwise with the error message "~~invalid move!~~" - **changed to "illegal command, try again"**

Specific Requirements

- Use header guards in all header (`.h`) files.
- Do not use `using` in header (`.h`) files.
- In C++ source files (`.cpp` files), you may import individual symbols using statements like "`using std::string`". *Do not* use "`using namespace <id>`", either in headers or in source files.
- All variables must be declared inside functions. No variables should be global or `extern`.
- You may not use `auto`.

Suggestions

- Make use of **gdb** to debug and also run **valgrind** to confirm your code has no memory leaks. (Though really, your code for this project shouldn't be using any direct dynamic memory allocation.)
- Test thoroughly! The public autograder tests are minimal and will fail to expose many common code errors. Part of the challenge of the project is for you to think of the different situations that need to be tested. And remember, unit testing is your friend!

Makefile & Compiling

You must create and submit a Makefile with a rule that creates a target executable called `skipbo`. The target should compile with no errors or warnings using the typical C++ compilation command: `g++ <source> -Wall -Wextra -std=c++11 -pedantic`. Do not use any special libraries that require different additional/different compiler options.

The autograder does an automatic check to see if it can compile the target. ***It is very important that you make sure your submission successfully compiles, otherwise your final score for the entire assignment will be 0.***

Git log

In the team repository you made for the final project, use `git add`, `git commit` and `git push` regularly, with meaningful commit messages. ***All team members should be writing code, so all team members should have a record of commits in the repository.***

Code Submission (Team)

Submissions of your project should be made to the **Final Project - Code (Team)** link on Gradescope. This will be a group submission, where each team member's name is indicated to Gradescope as part of the submission.

Your submission must include a copy of the output of `git log` showing at least four commits per team member to the repository. Save the `git log` output into a file called `gitlog.txt` (e.g. by doing `git log > gitlog.txt`).

Your submission must also include your `Makefile` as described above, and a `README` file explaining anything the graders should know about your submission, including any missing/incomplete functionality.

Note that while we do absolutely expect you to write code that tests your implementation, you are not expected to submit your tests to us for grading.

Contributions Submission (Individual)

Separately, after your team's code is submission is complete, each team member should individually create a brief ~~text~~ file called ~~`contributions.txt`~~ contributions.pdf (or an image of a text file - see Piazza @864) and submit on Gradescope via the **Final Project - Contributions (Individual)** link. The file should have up to three sections:

- A section titled 'TEAM' which lists each team participant's name and JHED id
- A section titled 'CONTRIBUTIONS' (required): A brief summary of which team members contributed what to the code submission.
- A section titled 'ABOVE AND BEYOND' (optional): If you feel that a team member contributed a particularly large amount of effort to the final product, name them here. You may name at most one team member. Do not name yourself.

Note that the deadline for this individual submission is 48 hours after the code submission deadline. Each team member must submit their own version of this file (in their own words) in order to get credit for their work on the final project.

Sample Runs

We will be adding various sample runs to show you the exact user interface that is expected. You should look at them carefully and make sure that your solutions produce exactly the same things when given the same input.

- sampleRun1.txt - demonstrates game play, legal and illegal moves, winner
- sampleRuns.txt - demonstrates command-line args and error handling, save and quit options
- sampleMainErrors.txt - demonstrates the full complement of command-line argument error handling that is required
- deck1.txt, sampleBuild.txt, save2.txt - demonstrates the interactions with completed build piles, including specific output messages to indicate which one was set aside. deck1.txt is the starting deck, play with 4 players stock size 24 no shuffling, resulting saved game state is save2.txt.
- sampleLoad.txt demonstrates the user interface when you load a saved game state, including changing the shuffle mode. To test your load functionality - start a game from a saved game state, immediately save it with a different name, and then "diff" the files to see that they are the same.
    - save0.txt is a saved game state after creating a new game with deck0.txt, 4 players, 30 cards each in the stock piles, no actual game play (turns) took place
    - save1.txt is a saved game state after creating a new game with deck1.txt, 4 players, 24 cards each in the stock piles, no actual game play (turns) took place

- sampleDraw.txt demonstrates the usage of the player option 'd' to draw cards into its hand. It begins by loading in saved game state from save1.txt (same file as above).
- sampleRand.txt demonstrates the a new game in random mode, saved to saveRand.txt
- Here's what quitting the game should look like:

```
ugrad> ./skipbo false save2.txt

 >> Player2 turn next
(p)lay, (s)ave, or (q)uit ? q
thanks for playing
ugrad>
```