CSE 303, Winter 2006, Assignment 4
Due: Monday 13 February, 9:00AM

Last update: 2 February

You will complete a small C program. The sample solution is about 125 additional lines of code. Download the rest of the program and sample output from the course website.

A Made-Up Game: Players can start at any time; they begin with a "score" of 80. Players are "in a circle" and for each game, one player is "first" and play proceeds "around the circle". One game's second player is the next game's first player (unless players leave; see below) and so on. A game works as follows:

- If there are fewer than two players, skip to the next game.
- The "game budget" starts at 0.
- Each player receives a random integer between 1 and 100. They choose to "play" or "not play". If they play, their score reduces by 10 and the game-budget increases by 10. Else their score reduces by 1 and the game-budget increases by 1. Players know how many previous players decided to play.
- Then each player who chose to play chooses "high" or "low". Other players do not know their choice. Non-playing players do not choose.
- Of the players who "chose low", the player with the lowest number has their score increased by half the game-budget (rounded down). Similarly, of the players who "chose high", the player with the highest number has their score increased by half the game-budget. In case of ties, all players who are part of the tie have their score increased by half the game-budget.
- After a game, any player with a negative score is removed. (A score can be negative during a game.)

Note: The game is *not* "zero-sum": the sum of scores after a game can be more than before (if there are ties) or less than before (if nobody chooses low or nobody chooses high or there is rounding).

The Set-Up: You will write a program that simulates this game. We are providing "players", some support code, and some type definitions, which you must use. All you do is write the code that "runs" the game; keeping track of players and "asking them" their plays and choices.

Files: `gamePlayers.h` defines players and `new_player` creates a new player. `gamePlayers.c` defines several (pretty bad) strategies that `new_player` uses when creating a new player. You need not change (or really understand) `gamePlayers.c`. `gameSupport.h` declares two functions defined in `gameSupport.c` and described below. You need not modify `gameSupport.c`, but you may want to for testing since you should not assume it will stay the same. `gameRunner.c` is where your code goes. You need to add definitions for the four functions declared. (You may add helper functions, but the sample solution did not.)

Players: `struct Player` defines a player. `name` is just a string. `player_state` is whatever "private state" a player needs to maintain. For most players in `gamePlayers.c` it is `NULL`, but that is not your concern. `play_function` and `choose_function` are function pointers you call to see if a player "plays" (1 for yes, 0 for no) and (if they play) whether they choose "high or low" (1 for high, 0 for low). You should pass both the `player_state` (which the function will cast as necessary) and the number for the player for the current game. For `play_function` you also pass a `play_info_t` indicating how many players have already made their decision, how many have not yet decided, and how many decided to play. So the `numBefore` and `numAfter` fields always sum to 1 less than the number of current players and `numPlaying` is always $\leq$ `numBefore`. For `choose_function` you also pass the total number playing, including the current player.

Finally, `state_destructor` is for memory management. It should be called with `player_state` when the player is being deallocated. This function frees the object pointed to by `player_state` and (presumably) all other objects reachable from this pointer that are no longer needed. (This is why each player must provide its own destructor; the caller does not know what type `player_state` "really has".)

Player list: You must use a `player_list_t` to store a ***circular doubly-linked list*** of current players. In such a list, each element points to its next and previous elements, the last element's next is the first element,

and the first element's previous is the last element. A zero-element list is `NULL` and a one-element list has itself as its next and previous elements. So to iterate through a non-empty list, you cannot test for `NULL` (you will not terminate). One option is to test for the "current pointer" being pointer-equal to the "first pointer"; a do-while loop is helpful. Another option is to keep the size of the list (which you will want anyway) and test for "have I processed the right number of list elements".

The fields in `struct PlayerList` should be self-explanatory. `num`, `playing`, and `choice` are "temporary" places to hold results during a game.

To add a player, make a new `player_list_t`, initialize `player` and `score`, add it to the beginning of the list, and return the new `player_list_t`. You need a special case for when the old list is `NULL`.

To remove a player, if it is the only player return `NULL`. Otherwise, adjust the previous element's next field and the next element's previous field. To avoid a space leak, also deallocate the removed element. This involves calling the player's state-destructor, freeing the name string, freeing the player, and freeing the list element. Do not leak and do not follow dangling pointers.

Running the game: You will need to keep track of the current players (and how many there are), who is first for the next game, and what game number is being played. Have an outer loop that does the following:

1. Call `add_new_player` with the current game number. If it returns -1, return (the game is over). If it retunrns 1, add a player (put them first in the list, so after step (3) they'll be last). Else do nothing.
2. If there are fewer than two players, print "game n: fewer than 2 players" and a newline and proceed to the next game.
3. Rotate who goes first.
4. Initialize the game-budget and play-info.
5. In a loop, give each player a number and then ask for their play. Update the player-list, the game-budget, and the play-info.
6. In a loop, ask each playing player for their choice and track what the best low-number and best high-number are.
7. Print "game n:" and a newline.
8. In a loop, adjust each player's score (nothing if they lost, increasing if they won) and then print out their status with a line like this:

```
printf("\t%s: %d %d %d %d\n",current->player->name,
       current->score, current->playing, current->choice, current->num);
```

9. Remove players with negative scores and adjust the number of players. This can be tricky. It is best to have two loops: one for while the first player has a negative score and one that removes other players.

Extra Credit:

1. Add a player-strategy to `new_player` that involves a human, by printing the information to standard out and getting decisions from standard in.
2. Add a function `voter` to `gamePlayers.c` that takes 3 pointers to players and returns a pointer to a new player that for each decision asks the 3 players and decides with the majority. Assume the 3 players are distinct and not otherwise used (this matters only for the state-destructor). Hint: Store the pointers to players in `player_state`.

Assessment and turn-in: Your solutions should be:

- Correct C programs that compile without warnings using `gcc -Wall` and do not have space leaks.
- In good style, including indentation and line breaks
- Of reasonable size

Use `turnin` for course cse303 and project hw4. If you use late-days, use project hw4late1 (for 1 late day) or hw4late2 (for 2) instead of hw4.