Unit 2 – Darts Assignment

Please complete all the pages on this form and upload it along with your .exe, code and report. This form **must not be zipped.**

Please ✓ which tasks you have completed:	
Task 1	✓
Task 2	✓
Task 3	✓ sort of? Partially.

Files for uploading – these must be checked off before uploading

	✓ **to indicate these have been included**
Code – **must be compressed in a .zip** **The zip file name must contain your student number.**	✓
.exe – **this must be included in the .zip**	✓
This form – **uploaded outside of zip**	✓

Please answer the following questions

Questions	
Does the code compile without syntax errors? If no explain what the problems are and how you've attempted to resolve it.	It compiles without any errors.
How many matches does your program simulate?	User decides. Defaults to 10,000 when provided a number less than 1.
Who plays first?	User decides. Defaults to Joe.
What percentage accuracy have you assigned to the players?	**Joe**: 71% to hit bull, 80% to hit on target, 75% to hit double, 69% to hit triple **Sid**: 73% to hit bull, 80% to hit on target, 76% to hit double, 69% to hit triple User can also set custom players with custom accuracy values.

Finn Else-McCormick - 2200864

Report – please write your report below:

The modularity provided by object orientation, now that I am used to working properly with it, has allowed me to work much more effectively by being able to focus in on and solve specific, smaller problems before having to figure them in to the broader picture. I often found that working within the single large scope inherent to procedural programming caused me to try and attack the whole problem from the top down, leading to overcomplicated and messy solutions in comparison with the more decomposition-based style of object orientation.
Additionally, it allows for much easier reuse of logic, such as the 301 and 501 games only differing in their constructors and their overriding of a single virtual function, whilst being able to easily share all the underlying logic which drives them.

One problem I came up against in this project was the storage of variables such as the players' scores, which belong to the player but only for the duration of a game. Rather than storing this data on the player itself as I originally had been doing, I ended up defining a nested struct within Game, storing this instance data and a pointer to the actual player, and creating instances of it out of the Player pointers passed in. This allowed me to decouple Player from the game logic, keeping those variables from continuing outside the scope they were intended for.
I repeated this instance pattern in the Championship class.

The 501 game's logic employs a very basic intelligence by using factors like the current score, which throw it currently is out of the three, whether it will be able to reach zero this round, etc to change tactic. When it is completely out of range of winning, it aims to score as high as possible to get down into that range. Once it's in there, it tries to reach even numbers under 40 where it will be able to make the final throw. When it is the final throw in a turn and it has the potential to reach 0, it prefers that over the other tactics.

I would've liked to expand that logic further but I unfortunately ran out of time to do so.

The Monte Carlo mode only calculates basic stats like the average win chance, but the Championship mode calculates the frequency of each set of scores (in sets) which came up, arranging them by frequency, as well as the mean and modal average score for each player.
See right for the output for the default players after 10,000 rounds in Championship mode.

The program is set up such that all of its modes work for any number of players, if provided them by the user. In retrospect I don't know why I set it up like that as it definitely added extra complexity for relatively little reward, but here we are.

```
 Joe |    Sid | Frequency
   7 |      0 | 0.56%
   0 |      7 | 1.25%
   7 |      1 | 1.6%
   7 |      2 | 3.89%
   1 |      7 | 3.98%
   7 |      3 | 6.53%
   2 |      7 | 7.64%
   7 |      4 | 8.17%
   7 |      5 | 9.89%
   7 |      6 | 10.2%
   3 |      7 | 10.7%
   4 |      7 | 11.8%
   6 |      7 | 11.9%
   5 |      7 | 11.9%
———————————————————
 5.2 |    5.9 | Mean
   7 |      7 | Mode
```

Pseudocode – please write your pseudocode below

Main

Ask user for game type (301 or 501), not moving on until a valid answer is given.

Ask user for char representing game mode ('S'ingle, 'M'onte Carlo or 'C'hampionship), not moving on until a valid answer is given.

If mode is not Single, ask user how many games to run, defaulting to 10,000 if they enter a number less than one.

Create a vector of Players.

Ask user if they would like to use the default players.

If yes, add Joe and Sid to the players vector, with their defaults.

If no, ask user for a name, then for percentage probabilities to hit: bullseye, on target, and, if the game type is 501, double and triple. Create a Player with these values and add it to the players vector, then ask user if they want to add another, repeating until they say no.

If mode is not Championship (where player order is procedural), ask user if they would like to change the order of players.

If yes, output the name of each player in the player vector and their index in the vector. Then ask user for the index of the player they want to move, not moving on until a valid answer is given. Then ask for the index they should be moved to, not moving on until a valid answer is given. Remove the player at the former index and re-insert them at the latter one. Output the names and indicies again to show the modification, then ask user if they want to keep reordering, repeating until they say no.

If mode is Single, run one game of the chosen type, outputting each player's score and the result of their throws every round, and waiting for the user to press enter at the end of every round to allow them to go through step by step.

If mode is Monte Carlo, run the chosen game type the number of times previously entered, with no output. At the end, output each player's number of wins, average chance of winning for any given game, and average number of turns taken to win.

If mode is Championship, run a championship the number of times previously entered. At the end, output a table with all the distributions of final scores (in sets) between players and their frequencies, as well as each players' mean and mode average scores.

Ask user if they want to play again. If yes, repeat game (but not prior questions).

Game

Create a vector of PlayerInstances from the provided vector of players, allowing us to track each player's score and turn within the game.

Loop over each instance, calling the virtual turn logic with it. This logic is overriden by each game type to implement its rules.

Repeat until a player wins. Register the game with the players, so they can keep track of their statistics.

301 Turn

If given player’s score is between 50 and 100, set goal to score minus 50, to a maximum of 20.

Otherwise, set goal to 50.

If goal is 50, attempt with player’s bullChance.

On success, try to reduce score by 50. On failure, try to reduce score by a random single from the board.

Otherwise, attempt with player’s onTargetChance.

On success, try to reduce score by goal. On failure, 50/50 chance try to reduce score by a score left or right of goal on the board.

If attempted score reduction would bring score below 0 or to between 1 and 49 inclusive, leave score as it was.

Increment player’s turn value.

501 Turn

For each of three throws:

Calculate working score by subtracting current potential score reduction from previous throws this turn from current actual player score.

If out of range to finish this turn (working score is greater than 50 times the number of throws left, to a minimum of 60), set goal to 50.

If able to make final throw (it is final throw for this turn and working score is either 50 or is even and 40 or below), set goal to working score.

If is not final throw and working score is 60 or below, loop from 40 down to 2 in steps of 2, checking if each value is possible as a goal. If it is, set goal to that and break.

Otherwise, if working score is greater than 20, set goal to 20, or else if greater than 16 set goal to 16. Otherwise loop from working score down to 2, checking if each value is possible as a goal. If it is, set goal to that and break.

If goal is 50 or 25 (inner or outer bull) use player’s bullChance. On failure, 10% chance to get the other bull, or else get a random single from the board.

If goal is less than 20 and it’s not the final throw of the turn, shoot for a single.

If goal is less than 40 and even, shoot for a double.

If goal is less than 60 and divisible by three, shoot for a triple.

Singles, doubles and triples all use onTargetChance to determine if the right band is hit (50/50 left or right of it if fail).

Doubles and triples then use the doubleChance or tripleChance respectively to determine whether the respective multiplier was hit, multiplying whatever band was actually hit in the previous step.

Singles have a 3% chance to accidentally hit a double, and a 1% chance to accidentally hit a triple.

Whatever was hit, that value is added on to the potential score reduction.

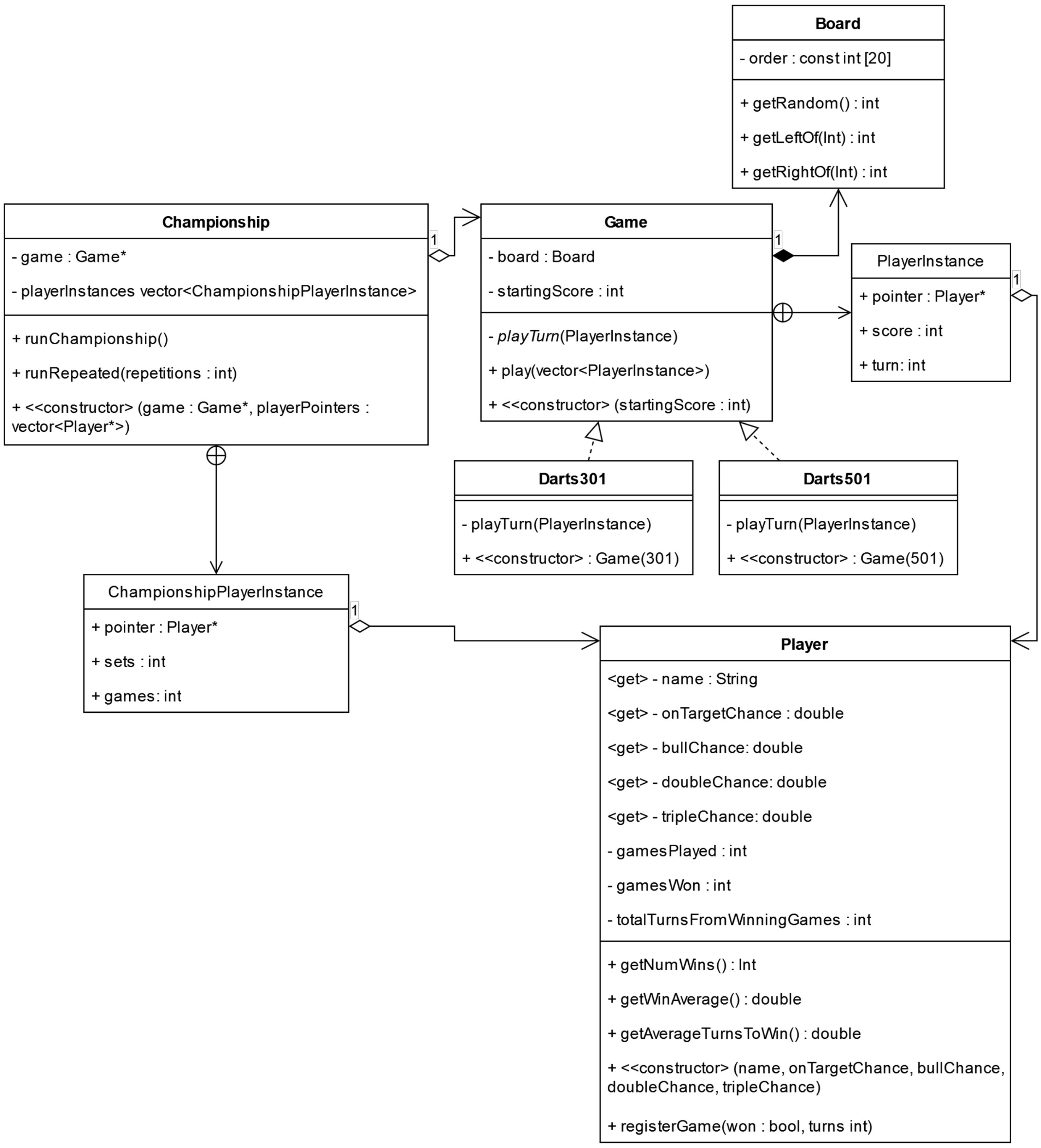
If the reduction would bring the score to a value above 1, or it would bring it to zero exactly and the working score from the final throw this turn was even, the reduction is applied. Otherwise, it is discarded.

Increment player’s turn value.

Championship

```
Create a vector of ChampionshipPlayerInstances from the provided vector of players, allowing us to track each player’s sets and games within the championship.
Reset all instances’ game and set values.
For the given number of sets (default 13):
    Choose a player at random to start.
    For the given number of games (default 5):
        Create a new vector of Player pointers.
        Add players starting from player chosen to start and wrapping around.
        Play the given game type with these players.
        Increment the winner’s game value.
        If any player has won a majority of this sets’ games (half the number of games rounded up, plus one if the result would otherwise be even), they win the set. Break from this loop, skipping remaining games.
    Reset all instances’ game value.
    Increment the winner’s set value.
    If any player has won a majority of the sets (same logic as with games), they win. Break from this loop, skipping remaining sets.
```

UML – please draw your UML below


Board
- order : const int [20]
+ getRandom() : int
+ getLeftOf(Int) : int
+ getRightOf(Int) : int
Championship
- game : Game*
- playerInstances vector<ChampionshipPlayerInstance>
+ runChampionship()
+ runRepeated(repetitions : int)
+ <<constructor> (game : Game*, playerPointers : vector<Player*>)
1
Game
- board : Board
- startingScore : int
- *playTurn*(PlayerInstance)
+ play(vector<PlayerInstance>)
+ <<constructor> (startingScore : int)
1
PlayerInstance
+ pointer : Player*
+ score : int
+ turn: int
1
Darts301
- playTurn(PlayerInstance)
+ <<constructor> : Game(301)
Darts501
- playTurn(PlayerInstance)
+ <<constructor> : Game(501)
ChampionshipPlayerInstance
+ pointer : Player*
+ sets : int
+ games: int
1
Player
<get> - name : String
<get> - onTargetChance : double
<get> - bullChance: double
<get> - doubleChance: double
<get> - tripleChance: double
- gamesPlayed : int
- gamesWon : int
- totalTurnsFromWinningGames : int
+ getNumWins() : Int
+ getWinAverage() : double
+ getAverageTurnsToWin() : double
+ <<constructor> (name, onTargetChance, bullChance, doubleChance, tripleChance)
+ registerGame(won : bool, turns int)

Code – please paste the file which contains the main function.

main.cpp

```cpp
#include "Player.h"
#include "Game.h"
#include "Championship.h"

#include "IOUtil.h" // Custom util functions for easier input and output
#include "RandUtil.h" // Custom util functions for easier randomisation

#include <vector>

int main() {
  int gameType;
  do { print("Enter game (301/501): "); gameType = input::getInt(); }
  while (gameType ≠ 301 && gameType ≠ 501);
  println();

  int mode;
  println("Modes: \n - (S)ingle game\n - (M)onte Carlo\n - (C)hampionship");
  do { print("Enter mode: "); mode = toupper(input::getChar()); }
  while (mode ≠ 'S' && mode ≠ 'M' && mode ≠ 'C');// Keep asking until get valid input
  println();

  int numGames = 1;
  if (mode ≠ 'S') {
    print("Enter number of games to run: ");
    numGames = input::getInt();
    if (numGames ≤ 0) { numGames = 10000; println("Defaulting to 10,000."); }
    println();
  }

  std::vector<Player> players;

  print("Use default players? (Y/N): ");
  bool useDefault = (toupper(input::getChar()) == 'Y');
  println();

  if (useDefault) {
    players = { Player("Joe",71,80,75,69), Player("Sid",73,80,76,69) };
  }
  else {
    bool keepAddingPlayers = true;
    while (keepAddingPlayers) {
      println(" - Add Player -");
      std::cin.ignore();
      std::string name;
      do { print("Enter name: "); name = input::getString(); } while (name == "");
      println("Enter probabilities (as percentages, but without the % symbol):");

      double bull;
      do { print("Bullseye chance: "); bull = input::getDouble(); }
      while (bull < 0 || bull > 100);

      double onTarget;
      do { print("On target chance: "); onTarget = input::getDouble(); }
      while (onTarget < 0 || onTarget > 100);

      double doubleChance = bull; double tripleChance = bull;
      if (gameType == 501) {
        do { print("Double chance: "); doubleChance = input::getDouble(); }
        while ( doubleChance < 0 || doubleChance > 100);
        do { print("Triple chance: "); tripleChance = input::getDouble(); }
        while ( tripleChance < 0 || tripleChance > 100);
```

```
         }
         players.push_back(Player(name, bull, onTarget, doubleChance, tripleChance));

         print("Add another? (Y/N): ");
         keepAddingPlayers = (toupper(input::getChar()) == 'Y');
         println();
      }
   }

   if (mode ≠ 'C') { // order in championship is procedural, nullifying this
      print("Change player order? (Y/N): ");
      bool changeOrder = (toupper(input::getChar()) == 'Y');
      println();
      while (changeOrder) {
         println("Current order:");
         for (int i = 0; i < players.size(); i++) {
            println((i + 1), " | ", players[i].getName());
         }
         int pos;
         do { print("Enter position of player to change: "); pos = input::getInt()-1; }
         while (pos < 0 || pos ≥ players.size());
         int newPos;
         do { print("Enter new position: "); newPos = input::getInt() - 1; }
         while (newPos < 0 || newPos ≥ players.size());
         Player movingPlayer = players[pos];
         players.erase(players.begin()+pos);
         players.insert(players.begin()+newPos,movingPlayer);

         println("\nModified order:");
         for (int i = 0; i < players.size(); i++) {
            println((i + 1), " | ", players[i].getName());
         }
         println();

         print("Keep changing? (Y/N): ");
         changeOrder = (toupper(input::getChar()) == 'Y');
         println();
      }
   }


   random::init();
   Game* game = NULL;
   if (gameType == 301) { game = new Darts301(); }
   else if (gameType == 501) { game = new Darts501(); }

   /* Have to use a vector of pointers instead of the actual vector of players because
the championship needs to call the game's play function from a scope where it can't
access a reference to the player vector */
   std::vector<Player*> playerPointers;
   for (Player& player : players) { playerPointers.push_back(&player); }

   bool playAgain = false;
   do {
      if (mode == 'C') {
         Championship championship(game, playerPointers);
         championship.runRepeated(numGames);
      }
      else {
         for (int i = 0; i < numGames; i++) { game→play(playerPointers, mode == 'S'); }

         if (mode ≠ 'S') {
            set_output_precision(3);
            println();
            println(" - Stats - \n");
            for (Player& player : players) {
                println(player.getName(), ":");
                println("Wins: ", player.getNumWins());
                println("Average Win Chance: ", player.getWinAverage(), "%");
                println("Average Turns to Win: ", player.getAverageTurnsToWin());
                println();
            }
        }
    }
    println();
    print("Play again? (Y/N): ");
    playAgain = (toupper(input::getChar()) == 'Y');
    println();
  } while (playAgain);

  delete game;
}
```

Player.h

```
#pragma once

#include <vector>
#include <string>

class Player {
   std::string name;

   double onTargetChance; double bullChance; double doubleChance; double tripleChance;

   int gamesPlayed; int gamesWon;
   int totalTurnsFromWinningGames;

public:
   Player(std::string name, double bullChance, double onTargetChance,
      double doubleChance, double tripleChance);
   Player(std::string name, double bullChance, double onTargetChance);

   std::string getName();
   double getOnTargetChance();
   double getBullChance();
   double getDoubleChance();
   double getTripleChance();

   int getNumWins();
   double getWinAverage();
   double getAverageTurnsToWin();

   void registerGame(bool won, int turns);
};
```

Player.cpp

```
#include "Player.h"

Player::Player(std::string name, double onTargetChance, double bullChance,
      double doubleChance, double tripleChance) {
   this->name = name;
   this->onTargetChance = onTargetChance; this->bullChance = bullChance;
   this->doubleChance = doubleChance; this->tripleChance = tripleChance;
   gamesPlayed = 0; gamesWon = 0;
   totalTurnsFromWinningGames = 0;
}
Player::Player(std::string name, double onTargetChance, double bullChance) :
   Player(name,onTargetChance,bullChance,bullChance,bullChance) {}

std::string Player::getName() { return name; }
double Player::getOnTargetChance() { return onTargetChance; }
double Player::getBullChance() { return bullChance; }
double Player::getDoubleChance() { return doubleChance; }
double Player::getTripleChance() { return tripleChance; }
int Player::getNumWins() { return gamesWon; };
double Player::getWinAverage() { return ((double)gamesWon / (double)gamesPlayed) * 100; }
double Player::getAverageTurnsToWin() { return (double)totalTurnsFromWinningGames /
   (double)gamesWon; }

void Player::registerGame(bool won, int turns) {
   gamesPlayed++;
   if (won) {
      gamesWon++;
      totalTurnsFromWinningGames += turns;
   }
}
```

Game.h

```cpp
#pragma once

#include <vector>

#include "RandUtil.h"
#include "IOUtil.h"
#include "Player.h"

class Board {
   const int order[20] = { 20, 1, 18, 4, 13, 6, 10, 15, 2, 17, 3, 19, 7, 16, 8, 11, 14,
9, 12, 5 };

   int getIndexOfValue(int val);
public:
   int getRandom();

   int getLeftOf(int val);
   int getRightOf(int val);
};


class Game {
protected:
   struct PlayerInstance {
      Player* pointer;
      int score;
      int turn;

      PlayerInstance(Player* pointer, int startingScore);
   };

   virtual bool playTurn(PlayerInstance& playerInst, bool output = false) = 0;

   Board board;
   int startingScore;

public:
   Game(int startingScore);

   Player& play(std::vector<Player*>& players, bool output = false);
};



class Darts301 : public Game {
   bool playTurn(PlayerInstance& playerInst, bool output = false);
public:
   Darts301();
};

class Darts501 : public Game {
   bool playTurn(PlayerInstance& playerInst, bool output = false);
public:
   Darts501();
};
```

Game.cpp

```cpp
#include "Game.h"

int Board::getRandom() {
   int index = random::randInt(0, 19);
   return order[index];
}

int Board::getIndexOfValue(int val) {
   for (const int& i : order) {
      if (i == val) { return i; }
   }
   return -1;
}

int Board::getLeftOf(int val) {
   int index = getIndexOfValue(val);
   if (index == -1) { return 0; }
   index -= 1;
   if (index < 0) { index += 20; }
   return order[index];
}

int Board::getRightOf(int val) {
   int index = getIndexOfValue(val);
   if (index == -1) { return 0; }
   index += 1;
   if (index >= 20) { index -= 20; }
   return order[index];
}


Game::Game(int startingScore) { this->startingScore = startingScore; }

Game::PlayerInstance::PlayerInstance(Player* player, int startingScore) {
   pointer = player;
   score = startingScore;
   turn = 0;
}

Player& Game::play(std::vector<Player*>& players, bool output) {
   std::vector<PlayerInstance> playerInstances;
   for (Player* player : players) {
      PlayerInstance instance(player, startingScore);
      playerInstances.push_back(instance);
   }

   Player* winner = NULL;
   if (output) { std::cin.ignore(); } // Must clear input buffer before using cin.get()

   bool gameIsOver = false;
   do {
      if (output) { println("-------\n"); }
      for (PlayerInstance& playerInst : playerInstances) {
         gameIsOver = playTurn(playerInst, output);
         if (gameIsOver) { // If over, break from for loop, skipping later players' goes
            winner = playerInst.pointer; break;
         }
      }
      if (output) {
         if (gameIsOver) { println(winner->getName(), " is the winner!"); }
         else { std::cin.get(); }
      }
   } while (!gameIsOver);
```

```cpp
  for (PlayerInstance& playerInst : playerInstances) {
    // Register game with players
    playerInst.pointer->registerGame(playerInst.score <= 0, playerInst.turn-1);
  }
  return *winner;
}



// --- 301 ---

Darts301::Darts301() : Game(301) {}

bool Darts301::playTurn(PlayerInstance& playerInst, bool output) {
  int goal = 50;
  if (playerInst.score > 50 && playerInst.score < 100) {
    goal = std::min(playerInst.score - 50, 20);
  }

  int scoreReduction = 0;
  if (goal == 50) {
    bool hitBull = random::randChance(playerInst.pointer->getBullChance());
    if (hitBull) { scoreReduction = 50; }
    else { scoreReduction = board.getRandom(); }
  }
  else {
    bool hitIntended = random::randChance(playerInst.pointer->getOnTargetChance());
    if (hitIntended) { scoreReduction = goal; }
    else { scoreReduction =
      random::randChance(50) ? board.getLeftOf(goal) : board.getRightOf(goal);
    }
  }

  bool canReduce =
    playerInst.score - scoreReduction == 0 || playerInst.score - scoreReduction >= 50;
  if (canReduce) { playerInst.score -= scoreReduction; }

  if (output) {
    println(playerInst.pointer->getName(), " turn ", playerInst.turn, ":");
    print("Attempted to hit "); (goal == 50) ? print("bull") : print(goal);
    if (goal == scoreReduction) { print(" succesfully"); }
    else { print(", hit ");
      (scoreReduction == 50) ? print("bull") : print(scoreReduction); } println(".");
      if (!canReduce) { println("Hit was invalid.");
    }
    println("Score: ", playerInst.score);
    println();
  }

  playerInst.turn++;
  return playerInst.score == 0;
}



// --- 501 ---

Darts501::Darts501() : Game(501) {}

bool Darts501::playTurn(PlayerInstance& playerInst, bool output) {
  static auto isValid = [](int goal) {
    return (goal <= 20 || goal == 25 || goal == 50 ||
      (goal <= 40 && goal % 2 == 0) || (goal <= 60 && goal % 3 == 0));
  };
  int scoreReduction = 0;
```

```
if (output) {
  println(playerInst.pointer→getName(), " turn ", playerInst.turn, ":");
}

int finalWorkingScore = 0;
for (int i = 0; i < 3; i++) { // For each throw
  int workingScore = playerInst.score - scoreReduction;
  finalWorkingScore = workingScore;  // As it's set every loop, the version from the
                                     // final loop remains afterwards
  int goal = 1;
  if (workingScore > 50 * (3 - i) && workingScore > 60) {
    // If out of range to finish, just try to get the score down
    goal = 50;
  }
  else if ((i == 2) && (workingScore == 50 || (workingScore ≥ 40 &&
        workingScore % 2 == 0))) {
    // If can make final throw, try
    goal = workingScore;
  }
  else if (workingScore ≤ 60 && (i ≠ 2)) {
    // Aim for evens when score getting low
    for (int desiredFinalScore = 40; desiredFinalScore > (i == 0 ? 2 : 0);
        desiredFinalScore -= 2) {
      int possibleGoal = workingScore - desiredFinalScore;
      if (possibleGoal > 0 && isValid(possibleGoal)) { goal = possibleGoal; break; }
    }
  }
  else {
    // Focus on 20 and 16 towards the end of turns
    if (workingScore ≥ 20) { goal = 20; }
    else if (workingScore ≥ 16) { goal = 16; }

    // If nothing else, try and score as high as possible
    for (int possibleGoal = workingScore; possibleGoal > 1; possibleGoal--) {
      if (isValid(possibleGoal)) { goal = possibleGoal; break; }
    }
  }

  int thisDartScoreReduction = 0;

  // Shoot for outer bull
  if (goal == 25) {
    bool hitIntended = random::randChance(playerInst.pointer→getBullChance());
    if (hitIntended) { thisDartScoreReduction = goal; }
    else {
      thisDartScoreReduction = random::randChance(10) ? 50 : board.getRandom();
    }
  }
  // Shoot for inner bull
  else if (goal == 50) {
    bool hitIntended = random::randChance(playerInst.pointer→getBullChance());
    if (hitIntended) { thisDartScoreReduction = goal; }
    else {
      thisDartScoreReduction = random::randChance(10) ? 25 : board.getRandom();
    }
  }
  // Shoot for single
  else if (goal ≤ 20 && (i ≠ 2)) {
    bool hitIntended = random::randChance(playerInst.pointer→getOnTargetChance());

    bool accidentalDouble = random::randChance(3);
    bool accidentalTriple = random::randChance(1);
    int multiplier = 1;
    if (accidentalTriple) { multiplier = 3; }
    else if (accidentalDouble) { multiplier = 2; }
```

```
      if (hitIntended) { thisDartScoreReduction = goal * multiplier; }
      else {
        thisDartScoreReduction = (random::randChance(50) ? board.getLeftOf(goal) :
          board.getRightOf(goal)) * multiplier;
      }
    }
    // Shoot for double
    else if (goal <= 40 && (goal % 2 == 0)) {
      int baseNumber = goal / 2;

      bool hitIntendedNumber = random::randChance(playerInst.pointer->
        getOnTargetChance());
      bool hitDouble = random::randChance(playerInst.pointer->getDoubleChance());

      int multiplier = 1; if (hitDouble) { multiplier = 2; }

      if (hitIntendedNumber) { thisDartScoreReduction = baseNumber * multiplier; }
      else {
        thisDartScoreReduction = (random::randChance(50) ?
          board.getLeftOf(baseNumber) : board.getRightOf(baseNumber)) * multiplier;
      }
    }
    // Shoot for triple
    else if (goal <= 60 && (goal % 3 == 0)) {
      int baseNumber = goal / 3;

      bool hitIntendedNumber = random::randChance(playerInst.pointer->
        getOnTargetChance());
      bool hitTriple = random::randChance(playerInst.pointer->getTripleChance());

      int multiplier = 1; if (hitTriple) { multiplier = 3; }

      if (hitIntendedNumber) { thisDartScoreReduction = baseNumber * multiplier; }
      else {
        thisDartScoreReduction = (random::randChance(50) ?
          board.getLeftOf(baseNumber) : board.getRightOf(baseNumber)) * multiplier;
      }
    }

    scoreReduction += thisDartScoreReduction;

    if (output) {
      std::string numWords[3] = {"First", "Second", "Third"};
      print(numWords[i]," dart attempted to hit ", goal);
      if (goal == thisDartScoreReduction) { println(" successfully."); }
      else { println(", hit ", thisDartScoreReduction,"."); }
    }
  }

  bool canReduce = playerInst.score - scoreReduction > 1 || (playerInst.score -
scoreReduction == 0 && finalWorkingScore % 2 == 0);
  if (canReduce) { playerInst.score -= scoreReduction; }

  if (output) {
    if (!canReduce) { println("Score was invalid, hits discarded."); }
    println("Score: ", playerInst.score);
    println();
  }

  playerInst.turn++;
  return playerInst.score == 0;
}
```

Championship.h

```cpp
#pragma once

#include "Game.h"

#include <vector>
#include <map>

/* This championship stuff could probably be done more elegantly but this project is due
   tomorrow so I frankly don't have the time */

class Championship {
   struct ChampionshipPlayerInstance {
      Player* pointer;
      int sets; int games;

      ChampionshipPlayerInstance(Player* player);
   };

   std::vector<ChampionshipPlayerInstance> playerInstances;

   Game* game;

public:
   Championship(Game*, std::vector<Player*>& playerPointers);

   void runChampionship(int numSets = 13, int numGamesPerSet = 5);

   // Run x many times and output statistics
   void runRepeated(int repetitions);
};
```

Championship.cpp

```cpp
#include "Championship.h"

Championship::ChampionshipPlayerInstance::ChampionshipPlayerInstance(Player* player)
{ pointer = player; sets = 0; games = 0; }


Championship::Championship(Game* game, std::vector<Player*>& playerPointers) {
   this->game = game;
   for (Player* player : playerPointers) {
      playerInstances.push_back(ChampionshipPlayerInstance(player));
   }
}

void Championship::runChampionship(int numSets, int numGamesPerSet) {
   for (ChampionshipPlayerInstance& instance : playerInstances) {
      instance.sets = 0; instance.games = 0;
   }
   for (int set = 0; set < numSets; set++) {
      bool championshipOver = false;
      int startIndex = random::randInt(0, (int)playerInstances.size()-1);
      // Choose starting player at random
      for (int i = 0; i < numGamesPerSet; i++) {
         bool setOver = false;
         std::vector<Player*> playerPointers;
         // Create vector of player pointers to pass into game in order of play
         for (int j = startIndex; j < playerInstances.size(); j++) {
            playerPointers.push_back(playerInstances[j].pointer);
         }
         for (int j = 0; j < startIndex; j++) {
```

```
        playerPointers.push_back(playerInstances[j].pointer);
      }
      Player& winner = game->play(playerPointers);
      startIndex++;
      if (startIndex >= playerInstances.size()) {
        // Increment starting player, wrapping around
        startIndex -= (int)playerInstances.size();
      }

      for (ChampionshipPlayerInstance& instance : playerInstances) {
        if (instance.pointer == &winner) { instance.games++; }
        if (instance.games >= std::ceil((numGamesPerSet % 2 ?
                                  numGamesPerSet : numGamesPerSet + 1) / (double)2)) {
          // First to gain majority wins set
          instance.sets++; setOver = true;
        }
      }
      if (setOver) { break; }
    }
    for (ChampionshipPlayerInstance& instance : playerInstances) {
      instance.games = 0;
      if (instance.sets >= std::ceil((numSets % 2 ? numSets : numSets + 1)
                        / (double)2)) {
        // First to gain majority of sets wins
        championshipOver = true;
      }
    }
    if (championshipOver) { break; }
  }
}

// This is a bad name for this function but I can't think of anything better right now.
// Runs championship repeatedly, calculates stats and outputs them.
void Championship::runRepeated(int repetitions) {
  int numPlayers = (int)playerInstances.size();
  std::map<std::vector<int>, int> frequencies;

  for (int i = 0; i < repetitions; i++) {
    runChampionship();
    std::vector<int> sampleVector;
    for (ChampionshipPlayerInstance& instance : playerInstances) {
      sampleVector.push_back(instance.sets);
    }
    if (frequencies.contains(sampleVector)) { frequencies[sampleVector]++; }
    else { frequencies[sampleVector] = 1; }
  }

  std::vector<int> sumFx(numPlayers);
  std::vector<std::map<int, int>> setFreqs(numPlayers);
  std::vector<std::pair<double, std::vector<int>>> sortedFrequencies;

  for (auto& freqPair : frequencies) {
    std::vector<int> sample = freqPair.first; int freq = freqPair.second;
    for (int i = 0; i < numPlayers; i++) {
      if (setFreqs[i].contains(sample[i])) { setFreqs[i][sample[i]]++; }
      else { setFreqs[i][sample[i]] = 1; }
      sumFx[i] += sample[i] * freq;
    }
    double percentageFreq = ((double)freq / (double)repetitions) * 100;
    sortedFrequencies.push_back(make_pair(percentageFreq, sample));
  }
  std::sort(sortedFrequencies.begin(), sortedFrequencies.end(),
    [](auto& a, auto& b) { return b.first > a.first; });

  std::vector<double> mean(numPlayers); std::vector<int> mode(numPlayers);
  for (int i = 0; i < numPlayers; i++) {
```

```cpp
        mean[i] = (double)sumFx[i] / (double)repetitions;
        std::pair<int, int> tempModalFreq = { 0,0 };
        for (auto& pair : setFreqs[i]) {
            if (pair.second > tempModalFreq.second) { tempModalFreq = pair; }
        }
        mode[i] = tempModalFreq.first;
    }

    std::vector<int> nameLength(numPlayers);
    std::vector<int> buffer(numPlayers);
    for (int i = 0; i < numPlayers; i++) {
        nameLength[i] = (int)playerInstances[i].pointer->getName().size();
        buffer[i] = std::max(nameLength[i] + 1, 5);
    }
    set_output_precision(3);
    println();
    for (int i = 0; i < numPlayers; i++) {
        print_whitespace(buffer[i] - nameLength[i]);
        print(playerInstances[i].pointer->getName(), " | ");
    }
    println("Frequency");
    for (auto& freqPair : sortedFrequencies) {
        double percentageFreq = freqPair.first; std::vector<int> sample = freqPair.second;
        for (int i = 0; i < numPlayers; i++) {
            print_whitespace(buffer[i] - 1); print(sample[i], " | ");}
        println(percentageFreq, "%");
    }
    for (int i = 0; i < numPlayers; i++) { for (int j = 0; j < buffer[i]; j++) {
        print("-");
    } }
    println("---------");
    set_output_precision(2);
    for (int i = 0; i < numPlayers; i++) {
        print_whitespace(buffer[i] - ((mean[i] == std::floor(mean[i])) ? 1 : 3));
        print(mean[i], " | ");
    }
    println("Mean");
    for (int i = 0; i < numPlayers; i++) {
        print_whitespace(buffer[i] - 1);
        print(mode[i], " | ");
    }
    println("Mode");
}
```

(These aren't relevant to the implementation but I'll include them for good measure)

RandUtil.h

```
#pragma once

#include <ctime>
#include <random>

namespace random {
   void init();
   void init(unsigned int seed);

   bool randChance(double percentage);
   int randInt(int min, int max);
}
```

RandUtil.cpp

```
#include "RandUtil.h"

void random::init() { std::srand((unsigned int) time(NULL)); }

void random::init(unsigned int seed) { std::srand(seed); }

bool random::randChance(double percentage) { return (rand() % 100) < percentage; }

int random::randInt(int min, int max) {
   return min + (int)std::round(rand() % (max - min + 1));
}
```

IOUtil.h

```
#pragma once

#include <iostream>
#include <iomanip>
#include <string>

void set_output_precision(int n);

void print_whitespace(int spaces);
void print_blank_lines(int lines);

void println();

// The variadic functions have to be defined in the header file for compiler reasons
// (you get linkage errors otherwise)
template <typename ... Args>
void print(Args&& ... args) {
   (std::cout << ... << args);
}
template <typename ... Args>
void println(Args&& ... args) {
   (std::cout << ... << args) << std::endl;
}

namespace input {
   std::string getString();
   char getChar();
   int getInt();
   double getDouble();
}
```

IOUtil.cpp
```
#include "IOUtil.h"

void set_output_precision(int n) { std::cout << std::setprecision(n); }

void print_whitespace(int spaces) { for (int i = 0; i < spaces; i++) { std::cout << "
"; } }

void print_blank_lines(int lines) { for (int i = 0; i < lines; i++) { std::cout <<
std::endl; } }

void println() {
   std::cout << std::endl;
}

std::string input::getString() {
   std::string s;
   getline(std::cin, s);
   return s;
}

char input::getChar() {
   char c;
   std::cin >> c;
   return c;
}

int input::getInt() {
   int i;
   std::cin >> i;
   return i;
}

double input::getDouble() {
   double d;
   std::cin >> d;
   return d;
}
```