Introduction to Computer Science I

Summer Sessions 2021 - COM SCI31-1 - STAHL

Project 6

Programming Assignment 6
BlackJack Dice!

Time due: 9:00 PM Friday, August 13th

Introduction

BlackJack, also known as 21, is the most widely played casino card game in the world. With this project, you will be implementing blackjack via pairs of dice, rather than a deck of playing cards. If you are unfamiliar with BlackJack Dice, please review the rules of the game and you can also play for fun to get some practice with the game.

Our game will focus on two players, the Computer and a single human Player, with no betting, no splitting, and no doubling down.

Your task

Your assignment is to complete this C++ program skeleton (XCode VS2019) to produce a program that implements the described behavior. (We've indicated where you have work to do by comments containing the text TODO. Please remove those comments as you finish each thing you have to do.) The program skeleton you are to flesh out defines six classes that represent the six kinds of objects this program works with: BJDice, Player, Board, TurnEvaluator, Die and RandomNumber. You will need to complete code for the Player, Board, TurnEvaluator and BJDice classes. Details of the interface to all of these classes are in the program skeleton, but here are the essential responsibilities of each class:

In order to simulate a real BlackJack game, a RandomNumber class has been provided to you. The RandomNumber class is fully implemented and does not require any student changes.

RandomNumber
- mMinimum : int - mMaximum : int
+ RandomNumber(min : int, max : int, minInclusive : bool = true, maxInclusive : bool = true) + random( ) : int

In order to simulate a real BJDice game, a Die class has been provided to you. By default, a Die object will have six sides. Calls to .roll( ) will randomly toss the die. That value can be retrieved later with calls to .getValue( ). For testing purposes, a die's value can be forced (cheated) with calls to .setValue(...). The Die class is fully implemented and does not require any student changes.

Die	
- mSides : int - mValue : int	
+ Die(sides : int = 6) + roll( ) : void + getValue( ) : int + setValue(value : int) : void	 const

The BJDice game works with two Players, one being the human player and one being the computer player. Each Player has two Dies that can be rolled. With calls to .roll( ), the dies should be randomly tossed. With calls to .roll(Die, Die), play can be forced (cheated). Each Player manages its own score and running total. A Player's running total is the sum of all its die's values up that point. After calls to .roll( ), a call to .addDiceToRunningTotal( ) will update the running total with the value of the Player's two current dies. A Player's prior running total saves the most recent running total value before it is changed. Once a Player's running total is sixteen ~~seventeen~~ or more, a call to .addDiceToRunningTotal(

) should add in only Die1, not both dies.. A call to .resetRunningTotal( ) occurs at the end of each turn of play and sets both the running total and prior running total back to zero. A Player's score represents the number of turns it has won. The score is incremented by one when calls to .wonTurn( ) occur or by two when calls to .won21( ) occur.

```
                         Player
- mDie1, mDie2 : Die
- mRunningTotal, mPriorRunningTotal, mScore : int
---------------------------------------------------------
+ Player( )

+ roll( )
+ roll( d1 : Die, d2 : Die )
+ getDie1( ) : Die                              const
+ getDie2( ) : Die                              const

+ addDiceToRunningTotal( ) : void
+ getRunningTotal( ) : int                      const
+ getPriorRunningTotal( ) : int                 const
+ resetRunningTotal( ) : void

+ getScore( ) : int                             const
+ wonTurn( ) : void
+ won21( ) : void
```

The BJDice games uses a Board to display various pieces of information on the console display as the game proceeds. Trivial getters and setters exist for each of the data members of a Board: humanScore, humanTotal, computerScore, computerTotal and runningTotal.

```
                         Board
- mHumanScore, mHumanTotal : int
- mComputerScore, mComputerTotal : int
- mRunningTotal : int
---------------------------------------------------------
+ Board( )

+ getHumanScore( ) : int                        const
+ setHumanScore( humanScore : int ) : void
+ getComputerScore( ) : int                     const
+ setComputerScore( computerScore : int ) : void
+ getHumanTotal( ) : int                        const
+ setHumanTotal( humanTotal : int ) : void
+ getComputerTotal ( ) : int                    const
+ setComputerTotal( computerTotal : int ) : void
+ getRunningTotal( ) : int                      const
+ setRunningTotal( runningTotal : int ) : void
```

The BJDice class has two Player objects and a Board object. All of the operations described in the sentences that follow need to be completed. The BJDice::humanPlay( ) operation needs to randomly toss the human Player's dice, adjust the human Player's running total and update the Board message that will be displayed in the console game for this round of play. Students should model this operation off of the other version of BJDice::humanPlay(Die, Die) which supports cheating for testing purposes is fully implemented in the skeleton. The BJDice::computerDice( ) operation needs to randomly toss the computer Player's dice, adjust the computer Player's running total and update the Board message that will be displayed in the console game for this round of play. Students should model this operation off of the other version of BJDice::computerPlay(Die, Die) which supports cheating for testing purposes is fully implemented in the skeleton

BJDice	
- mHuman : Player	
- mComputer : Player	
- mBoard : Board	
+ BJDice( )	
+ humanPlay( ) : void	
+ humanPlay(d1 : Die, d2 : Die) : void	
+ humanEndTurn( ) : void	
+ computerPlay( ) : void	
+ computerPlay(d1 : Die, d2 : Die) : void	
+ computerEndTurn( ) : void	
+ shouldComputerKeepPlaying( ) : bool	
+ finishTurn( ) : void	
+ isGameOver( ) : bool	
+ determineGameOutcome( ) : GAMEOUTCOME	**const**
+ getHuman( ) : Player	const
+ getComputer( ) : Player	const
+ getBoard( ) : Board	const

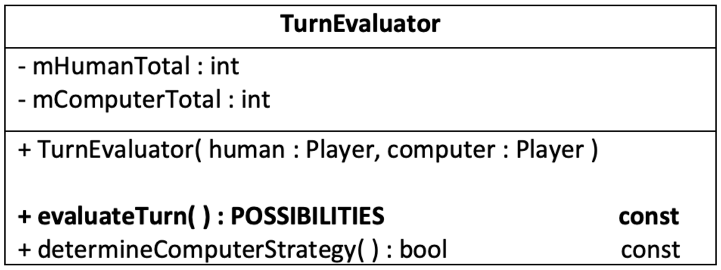
Once a game finishes as well as at any time during game play, the GAMEOUTCOME enumeration on the BJDice class describes the possible state of the game. Either the human player won the game, the computer player won the game or the outcome is not yet known because the game is not over.

<enumeration> GAMEOUTCOME
HUMANWONGAME
COMPUTERWONGAME
GAMENOTOVER

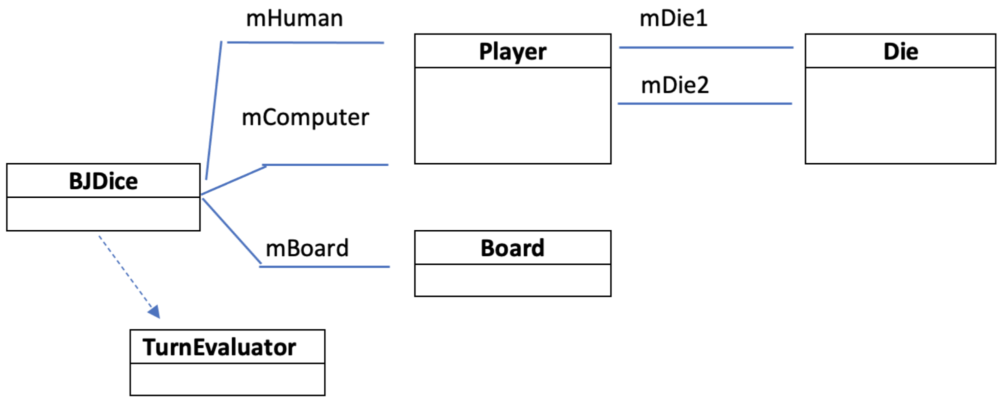
With the end of each turn, the POSSIBILITIES enumeration on the TurnEvaluator class describes the state of that particular turn of play. Either the human player busted (by having a score over 21), the computer player busted (by having score over 21), the human player has blackjack (by having a score of exactly 21), the computer player has blackjack (by having a score of exactly 21), both players had blackjack, the human player won (by having a greater running total than the computer's running total), the computer player won (by having a greater running total than the human's running total) or neither player won (because they each had identical non-blackjack running totals).

<enumeration> POSSIBILITIES
HUMANBUSTED
COMPUTERBUSTED
HUMANBLACKJACK
COMPUTERBLACKJACK
BOTHHAVEBLACKJACK
HUMANWON
COMPUTERWON
NOONEWON

The TurnEvaluator class is provided with the human and computer Player object at constructor time and saves each player's running total into its own data members mHumanTotal and mComputerTotal. Based on these totals, later calls to .evaluateTurn( ) returns one of the POSSIBILITIES choices.


TurnEvaluator
- mHumanTotal : int
- mComputerTotal : int
+ TurnEvaluator(human : Player, computer : Player)
+ evaluateTurn( ) : POSSIBILITIES const
+ determineComputerStrategy( ) : bool const

The following diagram shows how all these classes are interconnected. As you can see, a BJDice class has two Players and a Board. Each Player has two Dies. Through game play, the BJDice class uses a TurnEvaluator at certain points in the game.



Please see the TODO comments in the skeleton for further information.

You are free to create additional public and private methods and data members as you see fit. However, the test cases will only be driving the public methods of the BJDice, TurnEvaluator, Board, Player and Die classes diagrammed here.

The source files you turn in will be these classes and a main routine. You can have the main routine do whatever you want, because we will rename it to something harmless, never call it, and append our own main routine to your file. Our main routine will thoroughly test your functions. You'll probably want your main routine to do the same. If you wish, you may write functions in addition to those required here. We will not directly call any such additional functions. If you wish, your implementation of a function required here may call other functions required here.

The program you turn in must build successfully, and during execution, no method may read anything from `cin` . If you want to print things out for debugging purposes, write to `cerr` instead of `cout` . When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

Please read the posted FAQ for further assistance.

Programming Guidelines

Your program must *not* use any function templates from the algorithms portion of the Standard C++ library. If you don't know what the previous sentence is talking about, you have nothing to worry about. If you do know, and you violate this requirement, you will be required to take an oral examination to test your understanding of the concepts and architecture of the STL.

Your implementations must *not* use any global variables whose values may be changed during execution.

Your program must build successfully under both Visual C++ and either clang++ or g++.

The correctness of your program must not depend on undefined program behavior.

What you will turn in for this assignment is a zip file containing the following 14 files and nothing more:

1. The text files named **BJDice.h** and **BJDice.cpp** that implement the BJDice class diagrammed above, the text files named **Player.h** and **Player.cpp** that implement the Player class diagrammed above, the text files named **Board.h** and **Board.cpp** that implement the Board class diagrammed above, the text files named **TurnEvaluator.h** and **TurnEvaluator.cpp** that implement the TurnEvaluator class diagrammed above, the text files **Die.h** and **Die.cpp** that implement the Die class diagrammed above, the text files named **RandomNumber.h** and **RandomNumber.cpp** that implement the RandomNumber class diagrammed above, and the text file named **main.cpp** which will hold your main program. Your source code should have helpful comments that explain any non-obvious code.

2. A file named **report.doc** or **report.docx** (in Microsoft Word format) or **report.txt** (an ordinary text file) that contains in addition **your name** and **your UCLA Id Number** :
   a. A brief description of notable obstacles you overcame.
   b. A list of the test data that could be used to thoroughly test your functions, along with the reason for each test. You must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.) Notice that most of this portion of your report can be written just after you read the requirements in this specification, before you even start designing your program.

How nice! Your report this time doesn't have to contain any design documentation.

As with Project 3 and 4 and 5, a nice way to test your functions is to use the `assert` facility from the standard library. As an example, here's a very incomplete set of tests for Project 6. Again, please build your solution incrementally. So I wouldn't run all these tests from the start because many of them will fail until you have all your code working. But I hope this gives you some ideas....

```
#include <iostream>
#include <string>
#include <cassert>
#include "BJDice.h"
#include "Board.h"
#include "Player.h"
#include "TurnEvaluator.h"
#include "Die.h"
#include "RandomNumber.h"

using namespace std;
```

```
int main()
{
   using namespace std;
   using namespace cs31;
```

```
// test code for Die

Die d1;  d1.setValue( 1 );
assert( d1.getValue( ) == 1 );
Die d2;  d2.setValue( 2 );
assert( d2.getValue( ) == 2 );
Die d3;  d3.setValue( 3 );
assert( d3.getValue( ) == 3 );
Die d4;  d4.setValue( 4 );
assert( d4.getValue( ) == 4 );
Die d5;  d5.setValue( 5 );
assert( d5.getValue( ) == 5 );
Die d6;  d6.setValue( 6 );
assert( d6.getValue( ) == 6 );

// test code for Player

Player p;
assert( p.getScore( ) == 0 );
p.wonTurn( );
assert( p.getScore( ) == 1 );
p.won21( );
assert( p.getScore( ) == 3 );

Player busted;
assert( busted.getScore( ) == 0 );
assert( busted.getRunningTotal( ) == 0 );
assert( busted.getPriorRunningTotal( ) == 0 );
busted.roll( d5, d6 );
assert( busted.getDie1( ).getValue( ) == 5 );
assert( busted.getDie2( ).getValue( ) == 6 );
assert( busted.getScore( ) == 0 );
assert( busted.getRunningTotal( ) == 0 );
assert( busted.getPriorRunningTotal( ) == 0 );
busted.addDiceToRunningTotal();
assert( busted.getScore( ) == 0 );
assert( busted.getRunningTotal( ) == 11 );
assert( busted.getPriorRunningTotal( ) == 0 );
busted.roll( d6, d5 );
assert( busted.getDie1( ).getValue( ) == 6 );
assert( busted.getDie2( ).getValue( ) == 5 );
busted.addDiceToRunningTotal();
assert( busted.getScore( ) == 0 );
assert( busted.getRunningTotal( ) == 22 );
assert( busted.getPriorRunningTotal( ) == 11 );

Player blackjack;
assert( blackjack.getScore( ) == 0 );
assert( blackjack.getRunningTotal( ) == 0 );
assert( blackjack.getPriorRunningTotal( ) == 0 );
blackjack.roll( d4, d6 );
blackjack.addDiceToRunningTotal();
assert( blackjack.getScore( ) == 0 );
assert( blackjack.getRunningTotal( ) == 10 );
assert( blackjack.getPriorRunningTotal( ) == 0 );
blackjack.roll( d6, d5 );
blackjack.addDiceToRunningTotal();
assert( blackjack.getScore( ) == 0 );
assert( blackjack.getRunningTotal( ) == 21 );
assert( blackjack.getPriorRunningTotal( ) == 10 );

Player seven;
assert( seven.getScore( ) == 0 );
assert( seven.getRunningTotal( ) == 0 );
assert( seven.getPriorRunningTotal( ) == 0 );
seven.roll( d6, d1 );
seven.addDiceToRunningTotal();
assert( seven.getScore( ) == 0 );
assert( seven.getRunningTotal( ) == 7 );
assert( seven.getPriorRunningTotal( ) == 0 );

Player ten;
assert( ten.getScore( ) == 0 );
assert( ten.getRunningTotal( ) == 0 );
assert( ten.getPriorRunningTotal( ) == 0 );
ten.roll( d5, d5 );
ten.addDiceToRunningTotal();
assert( ten.getScore( ) == 0 );
assert( ten.getRunningTotal( ) == 10 );
assert( ten.getPriorRunningTotal( ) == 0 );
```

```
// test code for Board

Board b;
assert( b.getHumanScore() == 0 );
assert( b.getComputerScore() == 0 );
assert( b.getHumanTotal() == 0 );
assert( b.getComputerTotal() == 0 );
assert( b.getRunningTotal( ) == 0 );

b.setHumanScore( 10 );
b.setComputerScore( 20 );
b.setHumanTotal( 30 );
b.setComputerTotal( 40 );
b.setRunningTotal( 50 );

assert( b.getHumanScore() == 10 );
assert( b.getComputerScore() == 20 );
assert( b.getHumanTotal() == 30 );
assert( b.getComputerTotal() == 40 );
assert( b.getRunningTotal( ) == 50 );

// test code for TurnEvaluator
TurnEvaluator t1( busted, seven );
assert( t1.evaluateTurn() == TurnEvaluator::HUMANBUSTED );

TurnEvaluator t2( ten, busted );
assert( t2.evaluateTurn() == TurnEvaluator::COMPUTERBUSTED );

TurnEvaluator t3( blackjack, blackjack );
assert( t3.evaluateTurn() == TurnEvaluator::BOTHHAVEBLACKJACK );

// test code for BJDice

BJDice game;
assert( game.isGameOver() == false );
assert( game.determineGameOutcome() == BJDice::GAMENOTOVER );

game.humanPlay(d1, d2);
game.humanEndTurn();
game.computerPlay(d6, d5);
game.computerEndTurn();
game.finishTurn();

assert( game.isGameOver() == false );
assert( game.determineGameOutcome() == BJDice::GAMENOTOVER );
assert( game.getHuman().getScore() == 0 );
assert( game.getComputer().getScore() == 1 );

game.humanPlay(d6, d5);
game.humanEndTurn();
game.computerPlay(d3, d6);
game.computerEndTurn();
game.finishTurn();
```

```
assert( game.isGameOver() == false );
assert( game.determineGameOutcome() == BJDice::GAMENOTOVER );
assert( game.getHuman().getScore() == 1 );
assert( game.getComputer().getScore() == 1 );

game.humanPlay(d1, d2);
game.humanPlay(d3, d4);
game.humanEndTurn();
game.computerPlay(d2, d1);
game.computerPlay(d4, d3);
game.computerEndTurn();
game.finishTurn();

assert( game.isGameOver() == false );
assert( game.determineGameOutcome() == BJDice::GAMENOTOVER );
assert( game.getHuman().getScore() == 1 );
assert( game.getComputer().getScore() == 1 );

game.humanPlay(d6, d5);
game.humanPlay(d4, d6);
game.humanEndTurn();
game.computerPlay(d3, d6);
game.computerEndTurn();
game.finishTurn();
```

```
        assert( game.isGameOver() == false );
        assert( game.determineGameOutcome() == BJDice::GAMENOTOVER );
        assert( game.getHuman().getScore() == 3 );
        assert( game.getComputer().getScore() == 1 );

        game.humanPlay(d1, d2);
        game.humanEndTurn();
        game.computerPlay(d6, d5);
        game.computerPlay(d5, d5 );
        game.computerEndTurn();
        game.finishTurn();

        assert( game.isGameOver() == false );
        assert( game.determineGameOutcome() == BJDice::GAMENOTOVER );
        assert( game.getHuman().getScore() == 3 );
        assert( game.getComputer().getScore() == 3 );

        game.humanPlay(d6, d5);
        game.humanPlay(d1, d2);
        game.humanPlay(d3, d4);
        game.humanPlay(d5, d6);
        game.humanEndTurn();
        game.computerPlay(d5, d6);
        game.computerPlay(d1, d2);
        game.computerPlay(d6, d6);
        game.computerEndTurn();
        game.finishTurn();

        assert( game.isGameOver() == false );
        assert( game.determineGameOutcome() == BJDice::GAMENOTOVER );
        assert( game.getHuman().getScore() == 3 );
        assert( game.getComputer().getScore() == 3 );

        cout << "all tests passed!" << endl;
        return 0;

}
```

By August 16th, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later.

Linux Server Build Commands

Once you push over all your files, here are the build commands you need to enter to get your code runnable on the Linux servers:

```
g31 -c RandomNumber.cpp
g31 -c Die.cpp
g31 -c Player.cpp
g31 -c Board.cpp
g31 -c TurnEvaluator.cpp
g31 -c BJDice.cpp
g31 -c main.cpp
g31 -o runnable    main.o  RandomNumber.o  Die.o  Player.o  Board.o  TurnEvaluator.o  BJDice.o
./runnable
```

Submission status

Submission status	No attempt

Grading status	Not graded
Due date	Friday, 13 August 2021, 9:00 PM PDT
Time remaining	11 hours 4 mins
Last modified	-
Submission comments	▸ Comments (0)

Add submission

You have not made a submission yet.

◀ Final Exam Timing ...

Jump to...

Project 6 FAQ ▶