Project 3:
The Reality of Virtuality

Project due by your Final date[1]

Overview

In this project, you will implement Tic Tac Toe in the same pattern as how your Project 2 implemented Othello. With two games now completed, you will change your main to let the user choose which game they wish to play. Both your games will derive from a generic set of base classes that use virtual methods to allow specific games to override generic game-related functions like getting possible moves, applying moves, etc.

You will be given a ZIP file containing starting points for most of the .h files in this project. Those .h files have hints and requirements inside of them; you must add any required functions to the .h files as noted in those files, and then implement every function in a corresponding .cpp file that you will write in its entirety.

Base Classes and Virtual Functions

The "big picture" point of this project is that the *controller* portion of project 2 (the main, which interacts with the user) really doesn't do anything specific to othello. The main prints a "board", shows some "possible moves", reads in a string representing a "move", verifies that move is "valid", and "applies" the move, then repeats. None of that has anyting to do with othello, and in fact the same logic could be used to run any two-player alternating-turns board game... like Tic Tac Toe!

If our main is going to interact solely with generic board games, then we need to design base classes with virtual functions that Othello and TicTacToe-specific classes can derive from. If we separate the "game-specific" logic from the "game-agnostic" logic in the othello classes, we come up with the following base classes representing a generic two-player game. Each class notes any member variables that all game-specific types should have, as well as any functions/behaviors that all game-specific types should implement. The member variables will be protected; the functions will be pure virtual.

- `GameMove`: represents a single move that can be applied to a `GameBoard`.
  - moves can be assigned to with a string (`operator=(const string &rhs)`)
  - moves can be converted to a string (`operator string()`)
  - moves can compare themselves to other moves for equality (a new function, `bool Equals(const GameMove &other)`)
- `GameBoard`: represents all the state needed to track and play a game.
  - boards have member variables for the history of applied moves (a vector of `GameMove` pointers), the value (indicating who is winning/won), and who the next player is
    - boards have simple inherited functions for returning the value, history, and next player
  - boards can create a `GameMove` pointer appropriate to the `GameBoard`, e.g., `OthelloBoard` can create `OthelloMove`s and `TicTacToeBoard` can create `TicTacToeMove`s
  - boards can determine which spaces are valid possible moves for the current board state
  - boards can apply a move that represents a valid possible move
  - boards can undo the last applied move
  - boards can report if they are finished or not according to their own game rules
  - boards can return a custom string to represent the "name" of player 1 or 2. `OthelloBoard` would return "Black" for a player value of 1, whereas `TicTacToeBoard` would return "X".

[1] You *really* don't want to be working on this during finals week...

- `GameView`: prints a `GameBoard` to an `ostream`
  - gets constructed with a pointer to a `GameBoard` object appropriate for the derived `View` type
  - has a single function `PrintBoard` which does game-specific output of the view's board pointer

I have given you .h files for these three classes. You do not need to implement any functions for these classes specifically; all the code you will modify or create will be for Othello or TicTacToe classes.

Changes to Othello Classes

You will need to use **new versions of OthelloMove.h, OthelloBoard.h, and OthelloView.h** that I have provided to you. These files have slight tweaks to the parameters certain methods take. **Any changes to function declarations in a .h file must be matched in your .cpp files**.

In `OthelloBoard`, any Othello-specific return type or parameter to a function has been replaced by the generic Game-type classes. That means `ApplyMove` no longer takes `OthelloMove*`; it takes `GameMove*`. Likewise, `GetPossibleMoves` will take a `vector<GameMove*>` instead of `vector<OthelloMove*>`. You will need to update the functions' definitions in your .cpp files to match the new parameters and return types.

In `OthelloMove`, `operator=` is now virtual and returns a `GameMove&` instead of `OthelloMove&`. There is a new method for testing equality: `bool Equals(const GameMove &other)`, which returns true if the context Move object is equal to the parameter. OthelloMove's operator== is no more.

Finally, `OthelloBoard`/`Move`/`View` must all inherit from the appropriate generic `Game` classes. This gives them access to the protected members of those classes, notably `mValue`, `mHistory`, and `mNextPlayer` from `GameBoard`.

The changes from Othello-specific types to Game-generic types are necessary because the virtual functions defined in the Game base classes cannot take parameters specific to one individual game type. Since derived classes must match the declaration of virtual functions *exactly as declared in the base class*, Othello-specific classes must use functions which take generic Game types.

Pointers and Down-Casting

Recall the use of pointers with class inheritance: "a base-class pointer can point to a derived-class object, but you can't access the derived-class functions through the base class pointer." Since functions like `ApplyMove` now take base-class pointers to a `GameMove*`, those functions won't have access to the private Game-specific members like `mRow` and `mCol` of `OthelloMove`.

IMPORTANT: since you are writing these classes, you can assume (and code) that every `GameMove` pointer passed to `OthelloBoard`'s `ApplyMove` is actually a pointer to an `OthelloMove` object. Since you need information specific to `OthelloMove`, the first thing you should do in most Othello-specific functions will be to **cast** any generic Game-type pointers to Othello-type pointers.

Example:

```
void OthelloBoard::ApplyMove(GameMove *move) {
   // note:  the parameter is now GameMove*, but we can assume it
   // actually points to OthelloMove*, so we can safely down-cast.
   OthelloMove *m = (OthelloMove*)move;
   // work with m->mRow, m->mCol, etc.
}
```

This rule applies anywhere you can be certain that a base-class pointer actually points to a specific derived type. **If you're sure that a Game_____* points specifically to an Othello_______* or TicTacToe_______*, you can cast it freely.**

This also works with references. If you have a variable of type `const GameMove &other` and you need to treat it like an `OthelloMove`, you can cast it: `OthelloMove &casted = (OthelloMove&)other`.

Implementing Tic Tac Toe

Lab 5 should have helped you plan an attack for implementing TicTacToe. Here are some general thoughts:

- You will need to make your own .h files for TicTacToe classes. Base them on Othello's files. These need to derive from appropriate Game-type base classes.
- Much of the work will be copying and tweaking your Lab 2 assignment.
- TicTacToe moves should be similar to Othello moves, except they don't need to remember FlipSets.
- TicTacToe boards require very little work to apply or undo moves.
- `TicTacToeBoard` should implement its own version of `GameBoard's GetPlayerString` so it can return "X" or "O" for player 1/-1 instead of "Black" or "White".
- You will need to update `mValue` each time `ApplyMove`/`UndoLastMove` are called. You can make the final call on how to represent a `TicTacToeBoard's` "value", but whatever you choose should be consistent with `OthelloBoard`: when the game is over, a positive value means X won, and a negative value means O won.
- You may write your own functions that aren't present in `OthelloBoard`. Perhaps a helper function to see if anyone has connected 3 in a row...
- As a reminder, a derived class must **re-declare** all virtual functions that it inherits from its parent.

Changes to Main

"But wait!" you say. "Isn't the point of this inheritance stuff that we get to use the same exact main for either game?" Indeed, except for a few small tweaks. Your main from project 2 creates a few Othello-specific variables: the Board, a View, and (eventually) a Move pointer. These need to be converted to generic Game-type **pointers**, and initialized with appropriate heap-created variables based on the user's choice of game to play. (By the way... you need to ask the user which game they want to play!) Example:

```
GameBoard *board; // now a pointer!!
GameView *v; // likewise!!
cout << "What game do you want to play?  1) Othello; 2) Tic Tac Toe";

if (__ user wants to play othello __) {
   board = new OthelloBoard();
   v = new OthelloView(board);
}
else // guess!
```

Extra Credit Opportunity: Computer (AI) Opponent

For a few extra credit points, you may implement a computer AI opponent for your games. After attending the optional lecture on this topic on 4/28 or 4/29, download the optional "Project3_AI.zip" file from BeachBoard. This zip file contains **replacement** .h files for you to use; they do not change any functions or variables outlined in this document, they merely add more information needed to support an AI. Thus you should be able to place the new .h files in your existing Project 3 solution without overriding any of your existing work.

These are the changes and additions in the .h files:

- In GameBoard.h: a protected member `mWeight`, and an inline function `GetWeight()` which returns that variable. The weight of a board is different than its value; a weight gives the AI an indication of how "valuable" certain moves are, and is used so the AI can make intelligent decisions about which moves to choose. In Othello, for example, a corner piece is very valuable, so an AI should prefer a

move that takes a corner to one that takes several pieces in the middle of the board. See below for more information on board weights.

- In GameMove.h: a pure virtual function `Clone()` which returns a new **identical copy** of a `GameMove`, as a pointer to a heap object. `Clone` will be used by the AI to make duplicates of moves it thinks are "good", moves that would otherwise be deleted by the board. An implementation of this function will be given to you in OthelloMove.h; you will have to write an appropriate implementation in your `TicTacToeMove` class.
- A new file, GameAI.h: a class called `GameAI`, representing an abstract "intelligence" algorithm that can play 2-player board games. Has a single pure virtual method `FindBestMove`, which takes a `GameBoard` pointer and returns a (heap-created) `GameMove` object representing the best move on the board for the current player. The move returned by `FindBestMove` is ready to be applied to the board as the AI-approved "best" possible move for the given board. This class will be inherited by...
- ... MinimaxAI.h: a class `MinimaxAI` which derives from `GameAI`, representing the Minimax with Alpha-Beta Pruning algorithm for 2-player board games. Inherits the function `FindBestMove` from `GameAI`, and uses a private recursive helper function called `MinimaxFindMove` to perform the minimax algorithm on the current board.
- MinimaxAI.cpp: I have started an outline of the `MinimaxAI` implementation in MinimaxAI.cpp. One of your jobs will be to fill in the details of this algorithm so that it works on arbitrary `GameBoard` objects following the outline presented in lecture.
- In OthelloBoard.h: a new static private integer matrix called mWeights. You will initialize this matrix in the .cpp file according to the rules below. You will be responsible for adding a similar matrix to TicTacToeBoard.h and initializing it appropriately.

Changes to Main:

In main, ask the user whether they want to play a 1- or 2-player game. Either way, initialize a `GameAI` pointer to a new `MinimaxAI` object on the heap. Use this pointer in two ways:

1. For a one-player game, the AI player will take player 2's turns. (Challenge: let the user choose whether they want to go first.) Every time the user enters a valid **move** command, immediately use your `GameAI` to find the computer's best move in response and apply it. **Careful** to only do this if the game is not over after the human player's move.
   (a) Tweak your undo method so that `undo 1` undoes one of the *human* player's moves... in other words, you need to actually undo 2 moves so the computer's response is also undone. There should be no way for the human player to "switch sides."
2. Add a new menu option `auto n`, which causes the game to auto-select n "best" moves using the `GameAI` object. For a one player game, n represents how many moves to make on the human player's behalf – thus, you must actually apply $2n$ moves to account for both the human and the computer. For a two player game, n represents exactly how many moves to apply.
3. Add a new menu option `showWeight`, which shows the result of the board's `GetWeight()` function for testing.

Board Weights:

A board's "value" is a poor indicator of which game moves are more valuable than others. In Othello, a move that flips 2 pieces will give a higher value, but if that move then allows the opponent to take a corner, then the move was actually a poor decision. The AI opponent needs information about valuable moves, so we introduce the concept of board weight.

Like board value, your board games will need to continually update their weight as moves are applied. Unlike value, not every move on the board contributes equally to the weight, and you will need to use a matrix (8x8 for othello, 3x3 for tic tac toe) of integer constants representing the weight of each space on the board. When a move is applied, you will update the weight according to the values in the weight matrix.

For Othello, you will need to initialize mWeights anywhere in your .cpp file as such:

```
char OthelloBoard::mWeights[BOARD_SIZE][BOARD_SIZE] = {
{16, 0, 8, 8, 8, 8, 0, 16},
{0,  0, 0, 0, 0, 0, 0, 0},
{8,  0, 1, 1, 1, 1, 0, 8},
{8,  0, 1, 1, 1, 1, 0, 8},
{8,  0, 1, 1, 1, 1, 0, 8},
{8,  0, 1, 1, 1, 1, 0, 8},
{0,  0, 0, 0, 0, 0, 0, 0},
{16, 0, 8, 8, 8, 8, 0, 16}
};
```

This gives a high weight to corners and edge pieces, a low weight to any square next to those valuable spots, and a neutral weight to other squares. Using these constants, we can see that this board would have a weight of -16, showing a strong advantage to White even though the value is 0:

```
- 0 1 2 3 4 5 6 7
0 W . . . . . . .
1 B W B . . . . .
2 . . W . . . . .
3 . . B W B . . .
4 . . . B W . . .
5 . . . . . . . .
6 . . . . . . . .
7 . . . . . . . .
```

For Tic Tac Toe, you will implement a similar matrix with your choice of weight values. Think about Tic Tac Toe strategy and give higher weights to spaces that are "better". For my example solution, I give each space a weight equal to how many different directions a player can win using that square. The center piece, for example, can be used vertically, horizontally, or with two diagonals, for a total weight of 4. You can try to improve on this if you wish.

Memory management in Minimax:

The Minimax algorithm has to create, apply, undo, and delete a *lot* of moves. If you aren't careful, you will quickly blow your heap and crash your program by leaking move pointers. Make sure you delete moves appropriately:

- Any move passed to `ApplyMove` will be deleted by `UndoLastMove` – don't delete it yourself.
- You will need to delete any moves given to you by `GetPossibleMoves`.
- You need to delete any moves returned by recursive calls to MinimaxFindMove (think about this one...).
- Since you need to return your best move for the current board, you need to clone one of the possible moves if you think it's your current best. If you have a move that was previously your best, you need to delete that first.
- The "XX moves outstanding" code from Project 2 can help you debug your Minimax, but you should remove it when finalizing your project – Minimax makes thousands or millions of moves and you don't want that clutter. As you might expect, a properly working Minimax will result in +1 move outstanding when the algorithm is done: the one move returned as the "best move" for the current board.