CS 215 - Fundamentals of Programming II  
Spring 2008 - Project 4  
40 points

Out: February 20, 2008  
Due: March 17, 2008 (2nd Monday after spring break)

Reminder: Programming Projects (as opposed to Homework problems) are to be your own work. See syllabus for definitions of acceptable assistance from others.

The Game of Dots
Dots is a simple game usually played with pencil and paper. The playing field is a two-dimensional array of dots and the only valid move is to connect two adjacent dots horizontally or vertically. (No diagonal moves are allowed.) Players take turns connecting dots, except if a player connects two dots that complete a box (i.e., the top, bottom, left, and right sides of the box have been drawn), then the player fills in the box with her initial and gets a free turn. When all boxes have been filled in, the player who has claimed the most boxes wins. (Ties can be avoided by making sure that the number of possible boxes is odd).

In the following example, the first row shows the first four moves of a 3x3 dots game. (The size of the game is number of boxes.) The players are A and B, and they alternate turns. The second row picks up the same game after several more moves. The first board in the row is before B's turn. B makes a move, yielding the second board, and A completes the box, yielding the third board. A now moves again, completing another box (fourth board), and must move again (fifth board). It is now B's turn, and B will be able to complete 1 box, but must move again, and any additional move will give player A three more boxes. The final score will be A-5, B-4.

To write a computer program for playing Dots, instead of keeping track of a grid of dots, we need to keep track of a grid of boxes. Each grid location has is a box that has 4 sides (called top, bottom, left, and right), initially undrawn, and a label, initially blank. Adjacent boxes share a common side. For example, if two boxes are side by side, the right side of the left box is the same as the left side of the right box. Play is conducted as follows:

Consider the following specification for a BoxGrid class that represents the Dots playing field. Each grid location contains a Box object, that contains information regarding the state of the box. (The Box class will be provided and is explained below.) Grid locations are given as coordinates (row, col) where (0,0) is the upper left-hand corner of the grid.

**Specification for BoxGrid Class**

**Data Attributes**
For this project, a BoxGrid is modeled using a dynamically-allocated two-dimensional array of Boxes to allow grids of differing sizes to be created. In addition, to facilitate reporting the score of the game, the BoxGrid class keeps track of the player's names, represented by a single character. Thus the data attributes include at least those shown below. You may add additional appropriate attributes.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of rows</td>
<td>int</td>
<td>numRows</td>
</tr>
<tr>
<td>number of columns</td>
<td>int</td>
<td>numColumns</td>
</tr>
<tr>
<td>first player's name</td>
<td>char</td>
<td>player1</td>
</tr>
<tr>
<td>second player's name</td>
<td>char</td>
<td>player2</td>
</tr>
<tr>
<td>pointer to grid</td>
<td>Box**</td>
<td>grid</td>
</tr>
</tbody>
</table>

**Operations**

- Explicit-value constructor - receives the number of rows and columns, and the “names” of two players. Default values are given below. Throws a RangeError exception if initialRows or initialCols is less than 3 (to make sure the grid is large enough) or if the total number of Boxes is not an odd number (so that there always is a winner).

It should dynamically allocate a two-dimensional Box grid of the specified size. Note that since the Box class has a default constructor, the elements of the grid will be initialized automatically.

**Analysis**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Default</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of rows</td>
<td>3</td>
<td>int</td>
<td>variable</td>
<td>received</td>
<td>initialRows</td>
</tr>
<tr>
<td>number of columns</td>
<td>3</td>
<td>int</td>
<td>variable</td>
<td>received</td>
<td>initialCols</td>
</tr>
<tr>
<td>first player's name</td>
<td>'A'</td>
<td>char</td>
<td>variable</td>
<td>received</td>
<td>initialPlayer1</td>
</tr>
<tr>
<td>second player's name</td>
<td>'B'</td>
<td>char</td>
<td>variable</td>
<td>received</td>
<td>initialPlayer2</td>
</tr>
</tbody>
</table>

- Copy constructor - creates a new BoxGrid that is identical to an existing one.

**Analysis**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>original BoxGrid object</td>
<td>BoxGrid</td>
<td>variable</td>
<td>received</td>
<td>original</td>
</tr>
</tbody>
</table>

- Destructor - deallocates the grid

**Analysis** - no objects

- operator= - overloaded assignment operator function. Makes an existing BoxGrid object identical to the original BoxGrid object.

**Analysis**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>original BoxGrid object</td>
<td>BoxGrid</td>
<td>variable</td>
<td>received</td>
<td>original</td>
</tr>
<tr>
<td>this BoxGrid object</td>
<td>BoxGrid</td>
<td>variable</td>
<td>returned</td>
<td>*this</td>
</tr>
</tbody>
</table>
• GetRows - returns the number of rows in the grid

Analysis

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of rows</td>
<td>int</td>
<td>variable</td>
<td>returned</td>
<td>numRows</td>
</tr>
</tbody>
</table>

• GetColumns - returns the number of columns in the grid

Analysis

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of columns</td>
<td>int</td>
<td>variable</td>
<td>returned</td>
<td>numColumns</td>
</tr>
</tbody>
</table>

• GameOver - passes back the scores of the two players and returns true if the game is over, false otherwise

Analysis

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>first player score</td>
<td>int</td>
<td>variable</td>
<td>passed back</td>
<td>p1score</td>
</tr>
<tr>
<td>second player score</td>
<td>int</td>
<td>variable</td>
<td>passed back</td>
<td>p2score</td>
</tr>
<tr>
<td>is game over?</td>
<td>bool</td>
<td>variable</td>
<td>returned</td>
<td>---</td>
</tr>
</tbody>
</table>

• Write - outputs the BoxGrid in a two-dimensional grid to an output stream as shown in the example run below. Note that since the Boxes in the grid share sides, only two connected sides need to be output for each Box with special cases for the remaining edges of the grid. For example, if the bottom and right sides are output, then the top of the first row and the left side of the first box of each row are special cases.

Analysis

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>output stream</td>
<td>ostream</td>
<td>variable</td>
<td>received &amp; passed back</td>
<td>out</td>
</tr>
</tbody>
</table>

• DrawTop - draws in the top side of Box at grid location(row, col). If drawing in this side has caused a box to be completely drawn in, fill in the label of the box with player and return true, otherwise return false. Throws RangeError if row or column index is out of range. Throws DuplicateError if the side has already been drawn in. Note that since Boxes share sides, drawing in a side may cause an adjacent Box's side to be drawn that may cause the adjacent box to be completely drawn in.

Analysis

---

Revised: 02/23/08
<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>row index</td>
<td>int</td>
<td>variable</td>
<td>received</td>
<td>row</td>
</tr>
<tr>
<td>column index</td>
<td>int</td>
<td>variable</td>
<td>received</td>
<td>col</td>
</tr>
<tr>
<td>player's name</td>
<td>char</td>
<td>variable</td>
<td>received</td>
<td>player</td>
</tr>
<tr>
<td>completed box?</td>
<td>bool</td>
<td>variable</td>
<td>returned</td>
<td>-----</td>
</tr>
</tbody>
</table>

- DrawBottom, DrawLeft, DrawRight – same as DrawTop except they draw in the bottom, left, and right sides, respectively.

**Assignment**

The Box class and the Exception classes are provided for this assignment. They may be copied on the Linux server using:

```
    cp /home/hwang/cs215/project4/*.*.
```

There are three files, `except2.h` (same as `except.h` with addition of the DuplicateError exception), `box.h` and `box.cpp`. **You may not modify these files.** However, if you feel you need more operations from the Box class, talk to the instructor. The Box class defines operations that allow the Box object to be manipulated and accessed in accordance with the Dots game rules. The files contain comments explaining what each function does. Hopefully, this is sufficient.

Write the implementation of the BoxGrid class specified above. The BoxGrid class definition should be put in header file `boxgrid.h` with suitable compilation guards. The implementations of the BoxGrid member functions should be put in source file `boxgrid.cpp`. The BoxGrid class must be implemented **using a dynamically-allocated two-dimensional array**. Projects that do not use a dynamically-allocated two-dimensional array will be returned for resubmission with late penalty. The member function names and the order of the parameters must be as specified above. Your code will be linked with a grading driver program that expects this. Note that the main program may not be using all of the specified functions. However, all of the functions must be correct to receive full credit. Note: you may added additional private member function to the BoxGrid class as needed.

Write a main program in file `dots.cpp` that implements the Dots game described above using the BoxGrid class. This game will be text-based and interactive. It should have the following features:

- It should be able to construct a user-specified size game grid by accepting the number of rows, the number of columns, and the single-character names of two players as command-line arguments. I.e., there will be 5 words total on the command line. If the construction of a BoxGrid using arguments throws an exception, the program should print an error message, and continue with the default constructed 3x3 game grid.

  Note: since `argv` is an array of C-strings, the command-line arguments will need to be converted to integers using the function `atoi` that is defined in `<cstdlib>`. It is used as follows:

  ```
  int rows = atoi(argv[1]);
  ```

- The game grid should be output to the screen before every player move, and at the end of the game. At the end of the game, the final score and a declaration of which player won should be output to the screen.

- The main program must be structured in the following way. A player's turn must be encapsulated in a player function with the following analysis:
<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>game grid</td>
<td>BoxGrid</td>
<td>variable</td>
<td>received &amp; passed back</td>
<td>game</td>
</tr>
<tr>
<td>player's name</td>
<td>char</td>
<td>variable</td>
<td>received</td>
<td>player</td>
</tr>
<tr>
<td>another turn?</td>
<td>bool</td>
<td>variable</td>
<td>returned</td>
<td>-----</td>
</tr>
</tbody>
</table>

When a player function is called, it must make a turn. That is, it must draw a side in the game grid on behalf of player. If the turn results in a box being completely drawn in, the function returns true, otherwise it returns false. In addition, player functions must do the following:

- Output which player function is being executed and the player's name
- Output the player's choice for the turn
- Handle all errors via exception handling. That is, each function should choose a row index, a column index, and a side, and attempt to draw the side. Any exceptions should be caught and the choosing process repeated until a legal turn is made.

The main program alternates calling each player's function unless a player has completely drawn in a box as indicated by the player's function returning true, in which case the player gets another turn. Note that this may happen multiple times before the other player gets a turn. This continues until the game is over.

- Two player functions, `RandomPlayer` and `HumanPlayer`, must be written. For `RandomPlayer`, the choice of row index, column index, and side are to be made using the pseudorandom number generator. The pseudorandom number generator under `g++` works as follows. The generator is seeded using `srand()` (defined in `<cstdlib>`), which receives a long integer. Usually we want the game to be different each time, so most programmers use the result of the `time` function (defined in `<ctime>`) to seed the generator by using:

  ```c++
srand(time(0));
  ```

  This should be done once at the beginning of the main program.

  The function `rand()` returns the next random integer between 0 and `RAND_MAX`. Scaling this result to numbers between 0 and `n` can be done by finding the remainder of the result divided by `n`. I.e., `rand() % n`. Note that since the choice of row and column indexes are always in range, this function only has to handle the DuplicateError exception.

  `HumanPlayer` asks the user for their choice of row index, column index, and side. Since a user may enter an index outside the legal range, this function must handle both RangeError and DuplicateError exceptions. This is done by attaching multiple catch blocks to a try block, one for each type of exception.

- The code submitted must use `RandomPlayer` as the first player and `HumanPlayer` as the second player. However, you may want to explore what happens if a human player goes first, or if both players are random players, or if both players are human players.

A sample run of a program meeting these specifications is shown below.

You must submit a makefile named `Makefile.project4` that creates executable name `dots` for your project. Submissions without working makefiles will be assessed up to a 3-point penalty as indicated in the syllabus. It should conform to the examples given in the handout `Very Basic make` and demonstrated in class.
REMINDER: Your project must compile for it to be graded. Submissions that do not compile will be returned for resubmission and assessed a late penalty. Submissions that do not substantially work also will be returned for resubmission and assessed a late penalty.

Follow the guidelines in the *C++ Programming Style Guideline* handout. As stated in the syllabus, part of the grade on a programming project depends on how well you adhere to the guidelines. The grader will look at your code listing and grade it according to the guidelines.

What to submit
Electronically submit a tarfile containing your `Makefile.project4`, `boxgrid.h`, `boxgrid.cpp`, and `dots.cpp` as explained in the handout *Submission Instructions for CS 215*. Turn in a hardcopy of your `Makefile.project4`, `boxgrid.h`, `boxgrid.cpp`, and `dots.cpp`. Please do not submit `box.h`, `box.cpp`, or `except2.h` (either in the tarfile or in hardcopy). The submission system will judge only the BoxGrid class implementation. It will not run the Dots game itself.

Sample run

```
$ ./dots 3 3 A B
  0 1 2
  . . .
  0
  . . .
  1
  . . .
  2

RandomPlayer A's turn:
RandomPlayer A chooses [0,2] Left
  0 1 2
  . . .
  0     |
  . . .
  1
  . . .
  2

HumanPlayer B's turn:
Enter row #, column #, and side (t, b, l, r) separated by spaces: 0 0 t
HumanPlayer B chooses [0,0] Top
  0 1 2
  0     |
  . . .
  1
  . . .
  2

RandomPlayer A's turn:
RandomPlayer A chooses [0,2] Bottom
```
HumanPlayer B's turn:
Enter row #, column #, and side (t, b, l, r) separated by spaces: 0 0 l
HumanPlayer B chooses [0,0] Left

```
  0 1 2
  0  |   |
   0 |   |
  0  |   |
  1  |   |
  1  |   |
  2  |   |
```

8<----snip: later in the game -->8

```
  0 1 2
  0  |   |
   0 |   |
  0  |   |
  1  |   |
  1  |   |
  2  |   |
```

HumanPlayer B's turn:
Enter row #, column #, and side (t, b, l, r) separated by spaces: 0 0 r
HumanPlayer B chooses [0,0] Right

```
  0 1 2
  0  |B| |
   0 |   |
  0  |   |
  1  |   |
  1  |   |
  2  |   |
```

HumanPlayer B's turn:
Enter row #, column #, and side (t, b, l, r) separated by spaces: 2 0 r
HumanPlayer B chooses [2,0] Right

```
  0 1 2
  0  |   |
   0 |   |
  0  |   |
  1  |   |
  1  |   |
  2  |   |
```

8<----snip: later in the game -->8

Revised: 02/23/08

7 of 9
Random Player A's turn:
Random Player A chooses [1,1] Bottom

```
0 1 2
0 |B| |  
1 |     |
2 |  |  |B|
  |   |
```

Random Player A's turn:
Random Player A chooses [1,1] Right

```
0 1 2
0 |B| |  
1 | |   |
2 |  |A|B|
  |   |
```

8<---snip: later in the game -->8

```
0 1 2
0 |B|B|B|
1 |B|B|  
2 |  |A|B|
  |   |
```

Human Player B's turn:
Enter row #, column #, and side (t, b, l, r) separated by spaces: 0 1 b
That side is already drawn in - try again
Enter row #, column #, and side (t, b, l, r) separated by spaces: 1 0 b
Human Player B chooses [1,0] Bottom

```
0 1 2
0 |B|B|B|
1 |B|B|  
2 |  |A|B|
  |   |
```
HumanPlayer B's turn:
Enter row #, column #, and side (t, b, l, r) separated by spaces: 2 0 b
HumanPlayer B chooses [2,0] Bottom

0 1 2
0 |B|B|B|
1 |B|B|B|
2 |B|A|B|
 _._._._.

The final score is: A - 1; B - 8
Player B wins

Extra credit (5 points)
There is one addition that may be completed for extra credit. If you do the extra credit, you are expected to submit two versions of the project, one regular project, and one with the extra credit.

Write a third player function StrategicPlayer that chooses the row index, column index, and side autonomously based on a strategy devised by you. E.g., it might look for boxes with 3 sides drawn in so that it might draw in the fourth side and get another turn. It otherwise must meet the specifications for a player function. The main program submitted for the extra credit must replace the call to RandomPlayer with a call to StrategicPlayer (i.e., HumanPlayer is still used as the second player function). However, you might want to try playing a random player against a strategic player. Hopefully, a strategic player should always beat a random player. The amount of extra credit awarded will depend partially on the sophistication of the strategy implemented.

Submit a tarfile as explained above except that the makefile must be named Makefile.project4EC and the main program file should be named dotsEC.cpp. There will be a separate entry in the submission system for the extra credit. The submission system only will make sure that an extra credit project compiles. It will not run the program.

Revised: 02/23/08