9: undo and redo

CSCI 4526 / 6626 Fall 2016

1 Goals

- To build a Stack class by using an stl template.
- To be able to undo and redo moves.

2 The BoardState Class

Although our Board class stores a lot of information, most of it is structural, identifying each square by its coordinates and attaching each square to its neighbors. During the game, only three things change: the value stored in a square, its possibility list, and the number of remaining possibilities. These changing values constitute the state of the game, and only these values must be saved in order to undo a move or save and restore the game.

Define a new class, BoardState to be an array of 81 States. The BoardState constructor should receive a const Board* or const Board`&` parameter and copy the State portion of each of the 81 Squares into the new object.

Define the subscript function for this class to do the normal thing, no bounds-check needed.

3 Changes to Other Classes

Changes to the Game class. The `Game::run()` function displays a menu and accepts selections, which are processed by a switch. The switch should call action functions to carry out most of the user's choices. Try to limit the code that is actually inside teach case to three lines. If you have not already done so, move longer blocks of code out of the switch into separate functions.

Activate menu options undo and redo. Also instantiate two stacks of BoardState*s: the undo stack and the redo stack. Use the Stack class defined below, which is adapted from the stl Stack template. The undo stack always holds a copy of the current state of the game. The redo stack holds states that have been undone, until they are ready to be redone. The Game class has custody of these dynamically allocated objects and is responsible for deallocating them whenever one is discarded.

Every time you make a move or turn off a possibility, another $BoardState*$ will be pushed onto the `undo` stack. When you ask to undo or redo an action, a $BoardState*$ will be popped from one of the stacks and used to update the state of the *Board*. When you stop doing undos and redos and again call move or turnoff, the redo history becomes invalid. Thus, any call to move or turnoff must zap (empty) the redo stack. At the end of a game, the undo stack contains a complete history of your successful moves. (This history could be written to a file if anyone was interested in it.)

Add to the actions for the Move menu item. Immediately after the user makes a move, do the following:

- Create a new BoardState object, initialized to the state after the move.
- Push the pointer onto the undo stack.
- Clear the redo stack.

Implement the undo menu item, keeping in mind that you can't undo unless the stack has 2 or more BoardStates.

- Pop one BoardState* off the undo stack, then

- Push the pointer onto the redo stack.
- Then restore the Board state to the state on top of the undo stack, as defined below.

Implement the redo menu item:

- Return without doing anything if the redo stack is empty.
- Pop one BoardState* off the redo stack, then
- Push the pointer onto the undo stack.
- Then restore the Board state to the state on top of the undo stack, as defined below.

Changes to the Board class. Add a function `restoreState( BoardState* )`. This will loop through the 81 squares in the parameter BoardState and copy the information into the State portion of each Square. Call this from the Game class after either an undo or a redo.

4 The Stack Class

The standard template library has a template named `stack<T>` that does not really fill our needs in this application. The stl class does not provide the access necessary to iterate through the data that is needed for the realize and serialize functions.

Implementing a Stack. However, it is very easy to adapt the `vector<T>` template and create a stack with the right functionality. Our strategy will be to instantiate the stl template as `vector<BoardState*>` and at the same time, on the same line, derive the class *Stack* from it. Private derivation is needed to close off access to many unwanted `vector<T>` functions. However, private derivation forces us to define, in *Stack*, all of the functions we want to keep. In the derived class, you will need these functions; define them all as inline functions:

- Constructor and destructor.
- `void pop();` Delegate to `vector::pop_back()`
- `BoardState* top();` Delegate to `vector::top()`
- `void push( BoardState* );` Delegate to `vector::push_back( BoardState* )`
- `int size();` Return the number of BoardStates on the stack (in the vector).
- `void serialize(ostream& out) const;`
  Write all the BoardState data from the stack to the out stream.
- `void realize(istream& in);`
  Construct a new BoardState and read BoardState data into it from the in stream. Push the new BoardState onto the stack.
- `void zap();` Empty the entire stack by popping everything off it.

This *Stack* class is an adapter class: it adapts `vector` to our needs by adding, removing, and renaming functions. The first three functions, above, rename the functions inherited from `vector`. To implement them, simply call the appropriate function from the `vector` class. One function, `size()`, has the same name as the underlying function in `vector`. For that one function, you must use the `::` operator to call the `size()` function from the base class. (Otherwise, an attempt to delegate becomes an infinite recursion and your program will end with a segmentation error. I unthinkingly tried it. Bad news!)