9. Sudoku Helper: Undo and Redo

CSCI 4526 / 6626 Fall 2022

1 Goals

- To finish building the exception classes.
- To build a class that stores the state of the board.
- To use a Stack class as an adapter for the vector template.
- To be able to undo and redo moves.

2 The Frame* Class

The board is an array of 81 Squares, where each Square contains a State. Each Square is a large object: for a traditional board, it contains a vector of many cluster*s. Each cluster* contains nine pointers. All of these pointers are the same throughout the game, and do not need to be backed up after every move.

In contrast, the state is tiny, fits completely in 4 bytes, and changes in complex ways after every move. To support the actions of Undo and Redo, we need to keep track of the set of States initially and after every move. The Frame* class serves as a container for N^2 states.

How to make a Frame*. Dynamically allocate a Frame. In a loop, copy (not move) the State portion of each Square into the appropriate slot in the array. When the Frame* is constructed, it can be pushed onto a Stack. Do this in your Board::move() function at the very end of each move. (Further discussion is below.)

3 The Stack Class

Implementing a Stack. The standard template library has a template named `stack<T>` that does approximately (but not exactly) what we want in this application. 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<Frame>` 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 functionality in `vector<T>`. 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()`
- `Frame* top();` Delegate to `vector::top()`
- `void push( Frame* );` Delegate to `vector::push_back( Frame*)`
- `int size();` Return the number of Frame*s on the stack (in the vector).
- `void zap();` Empty the entire stack by popping everything off it. There is no need to copy blanks or zeros into the space that you free.

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!) One function, `zap()`, must be added. In addition, you may find more functions that will be useful.

4 Changes to Other Classes

Additional exceptions.. The `Game::run()` function displays a menu and accepts selections. Activate the menu option for move (if you have not already done so) and add a second exception base class with derived classes to handle illegal moves requested by the user.

Activate the menu option for turnOff(). This function allows the user to indicate that (by reasoning) he knows that one of the possibilities displayed on the list for a square is, actually, not possible. So he uses this menu item to turn it off.

Changes to the Game class. Activate the menu options for undo and redo. Instantiate two Stacks: the undo stack and the redo stack. Initialize the undo stack with the original state of the board. The top of the undo stack is always 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 objects and is responsible for deallocating them.

Every time you make a move or turn off a possibility, another *Frame* will be pushed onto the `undo` stack. When you ask to undo or redo an action, a *Frame* 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 Frame* object, initialized to the state at the end of 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 Frame*s.

- Pop one Frame* 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.
- Otherwise, pop one Frame* 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( Frame* )`. This will loop through the 81 squares in the parameter Frame* and copy the information into the State portion of each Square. Call this from the Game class after either an undo or a redo.