Sudoku-4: Logical Structure

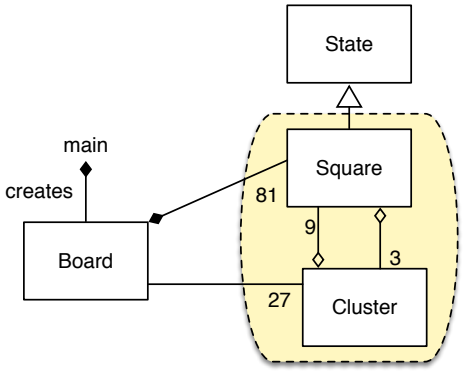
CSCI 4526 / 6626 Fall 2016

1 Goals

- To use an STL vector.
- To use an enumerated type.
- To learn about tightly coupled classes.
- To learn about circular dependencies.
- To use the this pointer.
- To model the logical structure of a Sudoku puzzle

2 Modeling the Sudoku Board

Together, the classes Board and Square model the physical structure of a Sudoku puzzle. We need one more class to model the logical structure of the puzzles: a class that represents the "row" or "column" or "box" relationship among a set of squares. We will call this class `Cluster`. The diagram below shows how these three classes are related.



A UML class diagram shows the relationships among all the classes in an application. In the diagram above, you see that `main()` creates the Board class. The black diamond between `main()` and Board indicates that main aggregates Board, and that they are allocated together and deallocated together. Similarly, Board aggregates an array of 81 Squares, so when you deallocate Board, the Square destructor is automatically called 81 times. Thus, Board is connected in the diagram to Square with a black diamond.

A Board also contains 27 Clusters. The white diamond indicates that the Clusters are created and attached to the Board after the Board is allocated. The Clusters have a shorter lifetime than the Board. They are born after the Board and die before the Board dies.

A cluster is an array of 9 pointers to Squares – all of the squares in a row or a column or a box. It stores pointers to squares, not the squares themselves, because each Square is part of multiple clusters. Whenever the Solver gives a value to a square, the clusters are used to efficiently find all related squares so that the value can be removed from their possibility lists.

Each Square is part of 3 clusters (4 or 5 in some Sudoku variations) and each Cluster points to 9 squares. When two classes are mutually dependent in this way, we say they are *tightly-coupled*. This is shown in the diagram above by a yellow area surrounding Square and Cluster. To compile such a pair, each class must `#include` the header file for the other class. However, if we put the

two `#includes` in the normal place at the top of the two .hpp files, this creates a circular `#include` pattern and the program will not compile.

We satisfy this paradox by using a partial class declaration called a *forward declaration*. Put this statement at the top of Square.hpp:

```
class Cluster;
```

This notifies the compiler that Cluster is a class name and allows it to compile references to Cluster pointers. The line:

```
#include Cluster.hpp
```

must then be placed at the top of Square.cpp so that the Square functions can call Cluster functions. This is the only time you will put two include statements in the .cpp file for a class.

3 Using vectors

In a traditional Sudoku puzzle, three clusters are related to each Square. In some Sudoku variations, a square can be part of more clusters. Therefore, we need to model the relationship between a Square and a variable number of Clusters. Happily, this is easy to do using the STL vector class. A vector is an array of objects or pointers that will grow as long as needed to contain the data you put into it. Each vector knows how many items are stored in it.

To access the data in a vector, you can use subscript, as if it were an ordinary array. We can also use vectors through variables called *iterators* that are very much like pointers. Here is the syntax:

- To use the vector class: `#include <vector>`
- To declare a vector variable named "clues": `vector<Cluster*> clues;`
- To put a new Cluster* into the `vector<Cluster*>` named `clues`:
  `clues.push_back( aClusterPointer );`
- To access an element of the array inside the vector: `clues[k]`. Note that you do not need iterators in this program if you use subscripts.
- To process all the data stored in a vector: `for (Cluster* cl : clues) cout << cl;`

4 Using enumerations

There are three types of clusters in a traditional Sudoku puzzle (more in some of the variations). In order to make debugging easier, each cluster will store its own type (row, column, or box) as an enumeration constant. Enumeration constants are a great way to make code easier to read and less error prone. They have one great fault: you cannot input or output them directly.

The easiest way to input an enumeration constant is to input a char and use the char in a switch that stores the right enum constant in your enum variable.

If you output an enumeration constant directly using `<<`, you see an integer (the internal code for the constant). These integers are not very useful to someone trying to read or debug the output. The civilized way to do the output is to define an array of strings and use the enum constant to subscript the array. For example, the following two declarations work properly together:

```
enum Color{ RED, BLUE, YELO};
static const char* colorStrings[3];
```

The enum declaration belongs near the top of the .hpp file of the class that will use Colors. (Suppose that class is named Palette.) The data declaration goes inside the Palette class, and the static

initializer for it must go into a .cpp file. This can be placed at the top of Palette.cpp:
`const char* Palette::colorStrings[] = "red", "blue", "yellow";`

5 Modify the Square Class.

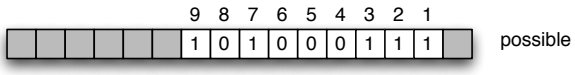
- Add a vector of Cluster*.
- To Square.hpp, add a forward declaration for **class Cluster**.
- Add a data member: a short int for the possibility count.
- In Square.cpp, `#include Cluster.hpp`.
- In Square, make a public inline function, **addCluster(Cluster*)** that pushes the parameter into the Square's vector.
- Add a `move()` function by adding a loop that will cycle through all the Clusters in the vector. For each cluster in the vector, call the Cluster::shoop() function to remove the new digit from the possibility lists of all the neighboring squares. Delegate the actual move action to the State class.
- Add a function: **void turnOff(int n);** See discussion below. Turn off position n in the square's possibility list and decrement the possibility count if position n changed. (Use case 3.) This will be called from `Cluster::shoop()`. The player may also turn off a possibility when he deduces that it cannot occur.
- Add code to State::print() to print the possibility count. My output looks like this:

```
Square [1, 1]  Value: 4  Possibilities = 5:  12-45--8-
```

5.1 The State of a Square

You have built and tested the skeleton two classes: Square and Board This week we will add data and function members to Square to model the relationships among squares.

Possibilities. Each square either contains a digit from 1 to 9, represented as a character, or is empty (represented on the output as a dash). At any moment, an empty square can legally hold only a subset of the nine digits. (It cannot hold the same digit as any other square in the same row, column, or box.) We will model this situation by adding a data member to the State class: the number of digits that are still possible in that square.



Changing the Possibility List. Each square on the board is part of three *clusters* of squares, which we will call its *neighbors*. These clusters represent the row, column, and box of the square. When we `mark()` a square with a value (use-case 2 in the overview), the possibility lists of all its neighbors must be adjusted (use-case 3) by turning off the bit that corresponds to the value we just marked. I call this `shooping` the neighbors. (Shoop is a made-up word, a combination of swoop and shoot.)

Did it change? When we shoop a digit from a cluster of squares, some of the squares in the cluster will remain unchanged because that digit had been previously shooped. Other squares will be changed, and it is essential to know which case happens. If a possibility list changes, the associated count must be decremented. If the bit was already turned off, a decrement should not happen. To know whether a change occurred, save the possibility list before the turn-off operation and compare it to the result after the operation.

6 Modify the Board Class.

- At the top of Board.hpp, before the beginning of the class definition, declare an enumeration called `ClusterT` for the three types of clusters. (More types will be added later.)
- On the next line, declare a static array of three const char*, to be used in the output when you print the cluster. Initialize this static array at the top of Board.cpp, following the example code given above.
- Modify the Board constructor to build 27 clusters. See discussion below.
- Modify Board::print() to print the 27 clusters using Cluster::print();

6.1 Creating the Clusters

This is a long, long function. Define it as a private helper function to be called from the Board constructor.

To create one cluster, Board needs to create an array of nine Square*'s for the nine squares in a row, column, or box. This is easy for a rows: the array will contain pointers to 9 consecutive squares in the Board array, and the Board will use an outer loop to do this 9 times. It is only a little harder for a column, since the subscript of the next Square will increase by 9 each time around the inner loop.

However, for the boxes, the code will be a pain. Basically, you will need a nest of 4 levels of for loop. The two innermost will collect the nine pointers for one box. The two outer loops will jump by 3 each time, and locate the upper-left corner of each box.

7 The Cluster Class

Implement a Cluster class with these parts:

- `#include Square.hpp` at the top of Cluster.hpp because a Cluster will contain an array of Square pointers.
- The Cluster class should have these data members:
  - A const char* to store the print name of the cluster type
  - An array of 9 Square*
- The Cluster constructor will create a tightly cross-linked and efficient data structure for making the necessary calculations during the game. It must have two parameters, a ClusterType enum code and an array of 9 Square pointers. Use the parameters to initialize the data members of the cluster. Then, for each Square* you add to the Cluster, use the Square* to call `Square::addCluster()` to add the current cluster to the square's vector. You will need to use the keyword `this`.

- Define a print function that prints the type of the cluster followed by the 9 squares in that cluster, one per line, with a blank line after the 9th square. Delegate the task of printing a square to the print function in the Square class.
- Declare an inline method for the output operator for Clusters.
- Define a function `void Cluster::shoop( char val );` that will be called to eliminate possibilities each time the human Solver moves a value in a Square. This is the heart of the application. Do the following
  - Use character subtraction to convert the char parameter to an int value.
  - For each of the cluster's nine Square pointers, use the Square* to turn off the bit corresponding to the parameter value. Do this for all Squares in the Cluster, even if they are fixed or equal to the Square that initiated the move. This will help to make the possibility display useful.

8 SOMETHING is Due October 10, the rest by October 17.

If you cannot finish this on time, submit what you have on the 10th and submit the rest on the 17th. There will be late penalties if I do not get some substantial effort by the 10th.

Update your test plan and unit tests for Square and Board and construct a test plan and unit test for Cluster. To test the new parts of Board and Square, it is enough to print the 27 clusters on the Board.

To test Cluster, it is adequate to show that shoop works for rows, columns, and boxes.

Call all the test functions from `main()`. Turn in a zipped folder containing copies of your test plans, your source code (.cpp and .hpp files) and your output.