5. Sudoku Helper: Logical Structure

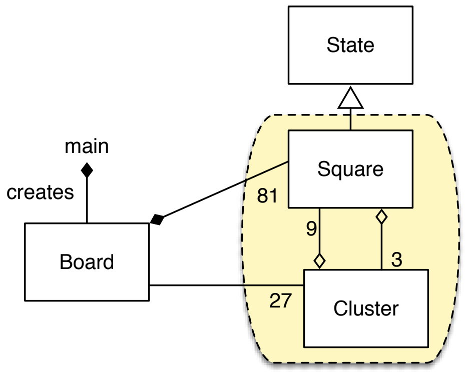
CSCI 4526 / 6626 Fall 2022

1 Goals

- To use an STL vector.
- To use an enum class.
- 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 "same row" or "same column" or "same 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 composes Board, and that they are allocated together and deallocated together. Similarly, Board composes an array of 81 Squares, so when you deallocate Board, the Square destructor is automatically called 81 times. Also, Square composes State, so deallocation Square causes the State destructor to be run.

A Board also contains $3 * N$ Clusters. The white diamond indicates that the Clusters are created and attached to the Board dynamically after the Board is allocated. The Clusters must be explicitly deleted by Board. So Clusters are born after the Board and die before the Board dies. This class relationship is called *aggregation.*

A Cluster is an array of N 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. (Of course, the Squares must exist before you can build the Clusters.) Whenever a player gives a value to a square, the clusters are used to efficiently find all related squares so that the new value can be removed from their possibility lists.

You have built and tested large parts of four classes: State, Square, Board, and Game. This week we will add a new class, Cluster, as well as data and function members for Square. Together, they model the relationships among squares in a Sudoku game.

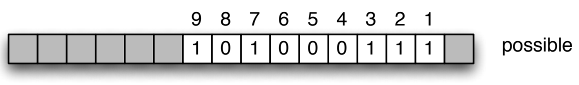
3 The Cluster Class

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. For each Square* you add to the Cluster, use the Square* to call `Square::addCluster()` to add the current cluster to that square's vector. This produces a 2-way access structure that allows you to get from a cluster to its squares and from a square to its clusters. 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 Square::print() .
- Declare an inline method for the output operator for Cluster.
- Define a function `void Cluster::shoop( char val );` that will be called to eliminate possibilities each time the human Solver marks a value in a Square. This is the heart of the application. Do the following
  - Convert the char parameter to an int value. (Character subtraction)
  - 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.

3.1 The State of a Square

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.)



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 by turning off the bit that corresponds to the new value we just marked (use-case 3). I call this operation `shoop`. ("Shoop" is a useful made-up word, a combination of swoop and shoot.)

4 Technical Concerns

4.1 Circular includes.

Each Square is part of several clusters. The number is 3 in a traditional Sudoku 3 or 4 or 5 in a diagonal puzzle and 4 in a sixy-Sudoku. Each Cluster points to N 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 `#include`s in the normal place at the top of the two .hpp files, this creates a circular `#include` pattern and the program will not compile. In the diagram, you can go round and round the Square – Cluster path. This same thing happens to the C++ preprocessor if you give it a circular include pattern.

We solve this paradox by using 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 Clusters and pointers to Clusters. 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 a .cpp file.

4.2 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 than three clusters. Therefore, we need to model the relationship between a Square and a variable number of Clusters. This is easy to do using an STL vector 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 access vectors using *iterators* that are very much like pointers. However, only simple subscripts are needed here. The syntax is:

- 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]`.
- To process all the data stored in a vector: `for (Cluster* cl : clues) cout << cl;`

4.3 Using enum classes.

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 class Color{ RED, BLUE, YELLOW };
    static const string 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 in the .hpp of the Palette class, with a constexpr initializer.

```
static constexpr string colorStrings[3] = {"red", "blue", "yellow"};
```

Your job is to set up an analogous set of declarations for Box.

5 Modifications to Existing Classes

5.1 Modify the Square Class.

- Add a vector of Cluster*.
- To Square.hpp, add a forward declaration for **class Cluster** and in Square.cpp, `#include Cluster.hpp`. This circumvents the circular dependency.
- In Square, make a public inline function, **addCluster(Cluster*)** that pushes the parameter into the Square's vector.
- Add a `shoop()` function: This is a loop that will cycle through all the Clusters in the vector. For each, call the Cluster::shoop() function to remove the new digit from the possibility lists of all the neighboring squares. Delegate actions to the State class, where possible.
- Add a function: **void turnOff(int n);** See discussion below. Turn off position n in the square's possibility list. (Use case 3.) This will be called from `Cluster::shoop()`. The player may also be permitted to turn off a possibility when he deduces that it cannot occur.

5.2 Modify the Board Class.

- At the top of Board.hpp, declare an enumeration called `ClusterT` for the three types of clusters. (More types will be added later.)
- At the top of Board.cpp, declare a static array of three const strings, to be used in the output when you print the cluster. Follow the example code given above.
- Add a data member to the Board class: a vector of Cluster*.
- Modify the Board constructor to build 27 clusters. As each one is constructed, push it into the vector. See discussion below for how to construct a cluster.
- Modify Board::print() to print the 27 clusters by delegating to Cluster::print();

5.3 Creating the Clusters

Define `makeClusters()`, a private helper function to be called from the Board constructor. Creating all the clusters requires a lot of tedious code, and `makeClusters()` could become a very long function. To maximize code readability, define three more private functions to help `makeClusters()`: createRow(), createColumn(), and createBox().

To create one cluster, Board needs to fill an array of nine Square*'s for the nine squares in the cluster. This array can be a local variable and be reused 27 times. Then call the Cluster constructor with this array as a parameter.

createRow(short j). Selecting the right Square*s is easy: write a loop to fill the parameter array with pointers to 9 consecutive squares in the Board array, then create a dynamic Cluster using this array. `makeClusters()` will use a loop to call createRow() 9 times, once for each of the 9 rows and push each finished Cluster* into the cluster-vector.

createColumn(short k). The loop to create the parameter array for a column is only a little harder, since the subscript of the next Square will increase by 9 each time around the inner loop. Again, `makeClusters()` needs to call this function 9 times to get all 9 Ccluster*s stored in the cluster-vector.

createColumn(short j, short k). For the boxes, the code will be more of a pain. Basically, you will need a nest of for-loops. The two outer loops will jump by 3 each time, and locate the upper-left corner of each box. The inner loop will collect pointers to the squares in each of the 3 rows of a box. In the body of this loop, either write a 4th loop or write three assignment statements. Store the Cluster* in the cluster-vector.

6 Due October 4 and 11.

Don't put this off until the last two days. You will not succeed unless you start now and work steadily on it. It is a more difficult assignment than modules 1 through 4. Expect to need to ask questions several times during the week, and come to class on October 4 with a written list of questions. I will collect that list and grade it.

Construct a test plan and unit test for Cluster and update your test plan and unit tests for Square and Board. 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.