PIC 10B: Homework 1

Due January 23

The `zip` file you extracted to find this `pdf` also includes files called `square.hpp` and `main.cpp`. In this assignment, you will submit a file called `square.cpp` that builds with `square.hpp` and `main.cpp` and (upon successful completion) runs to print:

```
* * *
* * *
8 * *

4 9 2
3 5 7
8 1 6

6 7 2
1 5 9
8 3 4

* * *
* * *
8 * *
```

To submit your homework:

- Upload `square.cpp` to **Gradescope** before the deadline.
- Name the file exactly `square.cpp`, with no extra spaces and no capital letters.
- Do **not** upload `square.hpp` or `main.cpp` to Gradescope.
- Do **not** enclose `square.cpp` in a folder or zip it. Do not do other weird things to the file. You should be submitting **exactly one file** and it should have the extension `.cpp`.
- When working on your homework assignment, do **not** edit `square.hpp`.
- Be sure that your code builds and runs with my files on **Visual Studio 2022**.

Review

This homework assignment reviews many important concepts from PIC 10A:

- The data types `bool`, `size_t`, `unsigned int`, and the `std::vector` class template.
- Classes, constructors, member initializer lists, member functions, and `const`.
- `for` loops, nested `for` loops, and `if` statements.
- How to organize code between header files and and `cpp` files.
- Passing explicit arguments by value and by reference.
- A little on pointers.

Learning Objectives

- Some familiarity with the `std::unordered_set` class template.
- The utility of returning a reference.
- A deep understanding of recursion.
- Using the `return` keyword to end a function call before the last line of the function body.

Magic Squares

A magic square is an $n \times n$ arrangement of the numbers 1 through n^2 such that:

- each number only appears once;
- the sum of
  - every row
  - every column
  - both main diagonals

  is the same.

Here's an example of one when $n = 3$:

$$\begin{bmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{bmatrix}.$$

Note that the sums for each row: $4+9+2, 3+5+7, 8+1+6$ are all 15, as are the sums for each column: $4+3+8, 9+5+1, 2+7+6$, as are the sums for both main diagonals: $4+5+6, 2+5+8$.

It can be shown mathematically that the sum of all the entries $1, 2, \ldots, n^2$ in the square is

$$\binom{n^2+1}{2} = \frac{n^2(n^2+1)}{2}.$$

Since there are n rows (and columns) and the sum of each row (and column) is the same, every row, column, and diagonal in a magic square must sum to the

$$\text{magic total} = \frac{n(n^2+1)}{2}.$$

In this homework, you will write the implementations to solve an $n \times n$ magic square where the user may, if they wish, enforce the placement of some numbers.

Overview

`square.hpp` contains the class interface for a class called `Square`. An instance of a `Square` has the following member variables:

- `size`: the side length of the square, i.e. we are modeling a `size` $\times$ `size` square.
- `data`: a vector containing `size * size  unsigned ints`, the data of the square arranged in a "flattened" format. We are interested in magic squares, so genuine data will always have values ≥ 1 and 0 will be used as a placeholder to indicate "no value".

  Recall the data type `unsigned int` cannot store strictly negative integers.
- `magic_total`: the magic square solution; every row, column, and two main diagonals must sum to this value.

Flattened format

In my implementation, the magic square $\begin{bmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{bmatrix}$ is stored by an instance of `Square` with

- `size == 3; magic_total == 15`;
- `data` equal to the vector containing `{ 4, 9, 2, 3, 5, 7, 8, 1, 6 }`.

It makes sense to use two indices to describe the positions in a square, with the first index specifying the row and the second specifying the column:

$$\text{a } 3 \times 3 \text{ square of positions described by two indices } = \begin{bmatrix} \texttt{(0,0)} & \texttt{(0,1)} & \texttt{(0,2)} \\ \texttt{(1,0)} & \texttt{(1,1)} & \texttt{(1,2)} \\ \texttt{(2,0)} & \texttt{(2,1)} & \texttt{(2,2)} \end{bmatrix}.$$

Looking back to the magic square $\begin{bmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{bmatrix}$,

- the value 8 is found in the square at the location indexed by `(2, 0)`;
- because `data` stores a flattened version of the square, 8 is found in data at the location indexed by `6`.

`size_t` versus unsigned `int`

Recall that `size_t` is the appropriate data type for storing:

- indices (the numbers you use to describe positions in a vector or container);
- the size of a vector or container.

For this reason the size of a Square is a `size_t` and if you use a pair of numbers `(i, j)` to describe a position in a Square, then `i` and `j` should be `size_t`s. On the other hand, the entries of a `Square` are not indices; the values contained in a `Square` have type `unsigned int`. Because `magic_total` is the sum of some of the values in a `Square`, it is also an `unsigned int`.

Tasks

1. Write an appropriate definition for the constructor `Square::Square(size_t n)`.
   - This constructor should construct a `Square` of a specified size with no values.
   - Remember that we are using 0 as a placeholder to indicate "no value".
   - All member variables should be initialized using the initializer constructor list.
2. Write an appropriate definition for
   `const unsigned int& get(size_t i, size_t j) const;`.
   - This function should allow you to get the value in row `i` and column `j` from the flattened data.
   - When I solved the homework, my flattened data consisted of the first row's data followed by the second row's data, etc. Using the first column's data followed by the second column's data, etc. would have resulted in slightly slower code.
   - The indexing of rows and columns should start from 0.
3. Write an appropriate definition for
   `unsigned int& Square::set(size_t i, size_t j)`.
   - The point of this function is to allow you to update the value in row `i` and column `j`.
   - For example,

     ```
     Square s(3);
     s.set(2, 0) = 8;
     ```

should set `s.data[6]` to 8 (although we cannot check this directly because data is a private member variable). This is very cool and is allowed because we are returning a reference (not to `const`).

4. Write a definition for the member function `void Square::print() const` so that executing

```
Square s(3);
s.set(2, 0) = 8;
s.set(1, 1) = 5;
s.set(0, 2) = 2;
s.set(2, 2) = 6;
s.print();
s.print();
std::cout << "I am demonstrating \\n behavior, too." << std::endl;
```

results in the following being printed to the console:

```
* * 2
* 5 *
8 * 6

* * 2
* 5 *
8 * 6

I am demonstrating \n behavior, too.
```

5. Write an appropriate definition for the member function `bool Square::is_magic() const`. This function should return `true` exactly when the `Square` has data making it a magic square, that is:

   - when each of the numbers 1 through n^2 appears exactly once;
   - the sum of
     - every row
     - every column
     - both main diagonals

     is equal to `magic_total`.

   Hint: You may find `std::unordered_set<unsigned int>` useful for checking the first part.

6. Write an appropriate definition for `bool Square::row_has_magic_total(size_t row) const`. This member function should check whether the `row`-th row has values summing to `magic_total`.

7. Write an appropriate definition for `void Square::see_magic_potential()`. Based on the value of data, this member function should print out all possible magic squares. I advise storing the values that have already been used in the `Square` in a `std::unordered_set<unsigned int>` called `used_up` and then calling the function you are about to write in task 8 with `slots_considered` specified as 0.

8. Write an appropriate definition for

   ```
   void help_see_magic_potential(
                       std::unordered_set<unsigned int>& used_up,
                       size_t slots_considered);
   ```

   For full credit, you must use recursion when defining this function.

   The very long comment in `square.hpp` gives guidance on writing the function.

   For full credit,

   ```
   Square s(4);

   s.set (0, 0) = 1;
   s.set (1, 1) = 13;
   s.set (2, 2) = 11;
   s.set (3, 3) = 9;

   s.see_magic_potential();
   ```

   must finish executing in under 60 seconds.

Grading

I will use some carefully chosen test cases to check your constructor and member functions. For the final member function, `Square::see_magic_potential`, there will be some 3×3 test cases and some 4×4 test cases. The 4×4 test cases will start with some positions filled in and should run in a reasonable time. (Solving a 4×4 square from scratch is slow: there are many cases to check and more than 7000 solutions to find!) To test a 4×4 square, give yourself a main diagonal (read left-to-right, top-to-bottom) of 1, 13, 11, 9 – there should be 8 magic squares with this diagonal – or give yourself that same diagonal, but impose a 16 in the top-right – there should be four magic squares using these values in these positions.