Goal

1. Design

You will improve your design skills while learning many new design techniques and styles.

2. C++ Programming

You will learn what it takes to develop a larger program in the C++ language.

3. C++ Survival

Some of the techniques you learned in more standard object-oriented languages may not apply here. In addition, C++ has some unique features that you may be able to exploit. This project should help expose you to these issues and show you how to make choices you can live with.

Overview

Abstraction as a Means of Extensible Design

Below you will read about some specific problems you are to solve. However, we will also show you how these problems fit into a more general analysis pattern. If you know this, you can design your solution to this more abstract model, thereby allowing you to plug in new concrete problems with less effort.

Here are the three problems you are to solve. We describe in the background section the common characteristics of these problems.

Fixing the Time on Your Clock

Your clock has gone dead because you forgot to wind it or replace the battery, or you had a power outage. This clock has hands, so you must turn them to adjust the time. Which way, and how far, should you turn the hands to fix the time the most quickly?

You've probably guessed that this will be the easy one of the bunch. In fact, we'll trivialize it even further. The clock only has an hour hand, so the question becomes how many whole hours backwards or forwards the hour hand must be moved. Then we will "complicate" it a bit by turning it into a general modulo-`n` counting problem, by saying that the clock displays `n` hours on its face.

Measuring Water

A man goes to a river with several containers of different sizes to get a specific quantity of water in one of the containers. He can perform a sequence of the following operations:

- Fill any container from the river
- Empty any container
- Pour the contents of any container into any other container until either the destination container is full or the source container is empty.

What sequence of operations does he perform to obtain the desired quantity of water in one of his containers?

For example, if he has a 5 liter container and a 3 liter container how does he get 4 liters? He can perform the following sequence (which is not a shortest approach to the problem):

- Fill the 3 liter container
- Pour the 3 liter container into the 5 liter container
- Fill the 3 liter container
- Pour the 3 liter container into the 5 liter container leaving 1 liter in the 3 liter container
- Empty the 5 liter container
- Pour the 1 liter in the 3 liter container into the 5 liter container
- Fill the 3 liter container
- Pour the 3 liter container into the 5 liter container

We now have 4 liters in the 5 liter container

Sam Lloyd's Sliding Block Puzzle

A shallow rectangular box has several identically sized labelled square wooden blocks in it. If not all of the space in the box is taken up with the wooden blocks then it is possible to slide a wooden block if there is an empty space next to the block. The problem is to arrange a given arrangement of blocks to another specified arrangement of blocks by sliding the blocks around.

The specifics of each problem will be given in the detailed submission instructions for parts 1, 2, and 3. You will be working on these puzzles with other students in the class as a team.

After you and your team have solved these puzzles you will have to solve one additional puzzle on your own. This puzzle will be similar to one of the puzzles described above. The specifics of this problem are given in the detailed submission instructions for part 4.

Background

A Single Abstraction for these Problems

The problems described in the overview section belong to a class of problems that can be characterized as follows:

- There is some kind of *world* that can be in one of many *configurations*. *Actions* cause the configuration of the world to change in some small and incremental way.
- The set of all possible configurations is not known ahead of time; they must be computed by applying actions and seeing where they take us.
- We are presented with an initial configuration, and asked to bring the system to an acceptable *goal* configuration.
- The acceptability of a configuration as a goal configuration is testable (often there is more than just one such configuration).
- The solution is then a sequence of moves that propel the world from the initial configuration to one of the goal configurations. It is enough to list a sequence of configurations that lead to a solution configuration.

Mapping the Abstraction

Let's see how Sam Lloyd's sliding block puzzle maps to this abstraction.

- The world is a box of labelled square blocks. The current configuration of the world is the label and position of each block in the box. An action consists of moving one block one unit horizontally or vertically to a new spot in a legal way (no collisions).
- The initial configuration is just the initial setup of all the blocks. The test for an acceptable final configuration would see whether the blocks have the proper labels at the proper positions.

We will leave it as an exercise to the student to determine the mappings to the other two problems.

The Algorithm

The interesting thing about these problems is that we do not have to think about the concrete problem *instance* in order to describe an algorithm to solve it! Read and make sure you understand the algorithm below:

```
Create an initially empty queue of configurations.
Insert the initial configuration into the queue.
While
  the queue is not empty and
  the first configuration in the queue does not meet the goal,
loop:
    Remove the first configuration from the queue and call it C.
    For each move applicable to C, loop:
      Make the move and enqueue the resulting
      configuration if it has not already been seen.
    end-loop.
end-loop.
The acceptable configuration is now at the head of the queue;
but if the queue is empty, there is no solution to the problem.
```

Did you recognize a pattern in the way the algorithm organizes and traverses its search space? It is a breadth-first search of a tree, where the nodes of the tree are discovered and attached as you go. This algorithm could be made more efficient. As written, it finds a goal configuration, but keeps looping until that configuration gets to the head of the queue. Feel free to improve or even redo the algorithm.

Notice some important things about the above algorithm:

- No specific concrete problem is ever mentioned.
- The algorithm is incomplete because it does not finish by telling you the sequence of actions that get you to an acceptable configuration. That, again, is left as an exercise for the student!
- We do not say how to determine if a configuration "`has not already been seen`".

What To Do

The activities in this project will have you design a framework that is easily adapted to all the problems of the classification described above. You will then implement and test all four of the problems using that design.

The general process you should follow goes something like this:

```
Develop the initial framework design in the abstract.
```

```
Write the code for the abstract framework.
For each problem for which you must implement a solution,
  Code the specific problem classes.
  If the previous step forced a modification of your design,
    Modify the code for the design as needed to make it work
    Modify the code for the previous problems as needed
    Submit the code for your latest design and all the problems solved so far
```

Shared Programming Responsibility

Because this is the only project you are doing in this course, and it is mostly a team project, there is a possibility that we will not be able to accurately assess your programming abilities if your teammates do most of the programming. Therefore, each team member must be responsible for an equal portion of the code written *in the activities 1, 2 and 3*. In the header comments, the name of the principal author should show up first, as always, in each code file. The principal author of a piece of code must be able to explain it orally if asked by his/her instructor. Activity 4 is an individual submission based on the work the team has done during submissions 1, 2 and 3.

Part 1

Part 1 is due on 27 October 2014.

In this activity you will design a framework capable of solving any puzzle of a specific type and, as a test of this framework, use the framework to solve a very simple puzzle. In this first activity, you are mainly concerned with the design of the framework. The term *framework* means a set of classes that enable implementation of solutions to certain problems. However, the framework by itself is not a complete program. You will work with abstract notions such as configuration, goal, and find-next-configuration. The problem solver should be able to solve any problem that conforms to an *interface* that you develop in your design. Think carefully about this interface, as you will also have to write classes that conform to it to solve the four problems.

Your design document is a text file that contains a description of the framework that will solve puzzles. It should include a description of the classes and the public methods that the client uses to solve puzzles. This description should explain how the solver will solve puzzles. It is important to realize that the solver must be capable of solving any puzzle and must contain all of the puzzle-solving machinery. The individual puzzles should not contain any puzzle-solving machinery but only contain methods implementing the rules for a particular puzzle. You do not need to design a general puzzle rule mechanism as each puzzle can explicitly code the possible successor states to any (legal) puzzle configuration.

The design document should also explain the flow of control and the sequence of steps that the solver would take when solving a simple puzzle. You should explain how the client uses the interface you have designed and the steps that are taken to solve a specific puzzle. For example, the puzzle problem can be to set the clock to 3 when it now reads 2. The design document explains how this will happen within the general solver framework.

When you design the generic configuration class, make sure you include a `display` function that will print some textual representation of the configuration to standard output. This will be of great help while you are debugging your code. The puzzle solver algorithm can be enhanced by a call to the display function inside the loop. Of course, the implementations of `display()` will only show up in the code for specific puzzles.

This activity will also perform the first validation of your design. You will write the code for your design. Then you will add code for the set-the-clock problem, put the two together, and see how they work. *It is important to note that you are expected to be using a framework that is equally applicable to the other problems.* Clearly, there are far easier solutions to this problem than the one we are having you build! This first puzzle is designed to test your design.

You will have to think about exactly how you will realize your design within the constraints of the C++ language. Although you are free to make your own decisions, some suggested approaches are shown atchoices.html that satisfy the requirement of a framework that adapts well to different "configuration/puzzle" problems. All of the choices given can be made to work. As a hint, students who choose to represent configurations as a vector of ints generally have an easier time.

Getting back to the clock problem, it requires three integers as input:

- number of hours on dial
- current clock time
- true time

These integers are to be provided on the command line in the order shown above. Remember that in C++ command line arguments are strings (arrays of chars) and must be converted to ints before they can be used in your program. You may use `atoi(argv[i])` to convert a command line argument to an integer. If you get the wrong number of arguments, or if the times are out of bounds with respect to the legal hours on the dial, you should report an error on standard error and quit.

The program is to be called **clock**, which means the `main` function should be defined in a file named **`clock.cpp`**. As submitted, the program must print out the solution by listing the sequence of configurations needed to reach the chosen goal configuration from the starting configuration.

You must also submit a file named `readme` containing your design and any other information about your program you want. The readme file is part of every submission for this project. If you modify the design in the future, which you can do at any time without penalty, you must submit an explanation of changes in the **`readme`** file.

How To Submit

You must submit all the **`.cpp`** and **`.h`** files required to build the **clock** program. It must be possible to compile the **clock** program by executing **gmakemake** and then simply **make**. Your design document must also be submitted as the **`readme`**. The submission must be from your team account.

```
submit -v swm-grd project1 clock.cpp other-needed-code-files readme
```

Part 2

Part 2 is due on 10 November 2014.

The purpose of this activity is to implement the solution for a problem that requires a slightly more involved configuration design. Write the code for the water measuring problem, plug it into your framework, and see how it works.

The program takes command line arguments specifying the initial state of the problem. The first command line argument is an integer representing the desired amount of water. The rest of the command line arguments are integers specifying the capacities of the containers. (The number of containers is the number of remaining command line arguments.) For example, the command

```
    water 4 3 5
```

would specify the problem of obtaining 4 liters or water using two containers of size 3 liters and 5 liters.

The program is to be called **water**, which means the `main` function should be defined in a file named **`water.cpp`**. As submitted, the program must print out the solution that leaves the desired amount of water in one of the containers. If there is no solution your program must detect this and print out an appropriate message.

If you have modified the design, you must submit an explanation of changes in the **`readme`** file.

Configuration Design Suggestions

The world, although more complicated than the clock, is still fairly simple. The configuration basically consists of the amount of water in each of the containers. There must also be a place where the sizes of the containers is remembered but, since these values never change, these capacities do not need to be included in each configuration.

How To Submit

You must submit all the **`.cpp`** and **`.h`** files required to build the **water** program *and* the **clock** program. It must be possible to compile both programs by executing **gmakemake** and then simply **make**. If your underlying design changed, include the changes in the **`readme`** file mentioned above. (If you had to change your design, then you probably need to update the clock program so that it continues to work.) The submission must be from your team account.

```
submit swm-grd project2 clock.cpp water.cpp other-needed-code-files readme
```

Part 3

Part 3 is due on 24 November 2014.

The purpose of this activity is to implement the solution for a problem that at least appears very complex to humans. We hope that you will be surprised how easily your framework discovers a solution to this problem. Write the code for the sliding block problem, plug it into your framework, and see how it works.

Input

Your program will need to be told the initial configuration and the goal configuration. The program will be called `lloyd` and will take two arguments:

- The name of the input file to read for the initial configuration data. If this name is "-" then the initial configuration data is read from the standard input.
- The name of the output file where the solution is to be written. If this name is "-" then the solution is written to the standard output.

If you get the wrong number of arguments or you have difficulty opening, reading, or writing any files you should report an error on standard error and quit.

The puzzle can be considered to be a 2-dimensional array of characters. The character '.' represents an empty space and any other non-whitespace character represents a movable piece. The `lloyd` program will be given two configurations: the starting configuration and the desired ending configuration.

The format for the data (either from a file or standard input) is as follows:

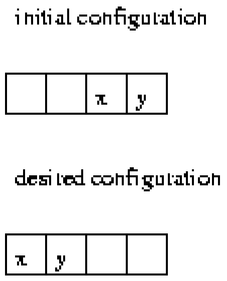
- The first line of the initial configuration data will contain two integers which are the width and height of the box.
- This will be followed by a number of strings, one per line, each having "width" characters. The total number of strings will equal the height of the puzzle. These strings represent the initial configuration of the puzzle and the characters in the strings represent the labels of the pieces with the '.' character representing an empty space.
- This will be followed by an empty line.
- Following this empty line will be a number of strings, one per line, each having "width" characters. The total number of strings will equal the height of the puzzle. These strings represent the desired final configuration of the puzzle and the characters in the strings represent the labels of the pieces with the '.' character representing an empty space.

You should not assume that there is exactly one empty space - there might be none or there might be more than one.

You are responsible for detecting any irregularities in the input and exiting the program with a message to standard error. If there are too many or too few numbers on a line, but it is compensated for in the rest of the input, we do not require that you detect this error. In other words, your input reader does not have to be aware that new lines are a different kind of *white space*.

An easy way to read this input is to use formatted input to read the strings, e.g., `is >> str;`. Note that this will skip whitespace so it will also skip any number of blank lines. You do not have to check for the presence or absence of blank lines in your program.

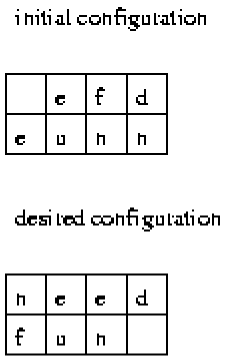
A sample input file that shows a rather easy version of this puzzle can be found at lloyd1.in. It is an example that is easily solved by hand. Be sure and use it as an early test case. Here is what it looks like:



A Graphical Representation of lloyd1.in

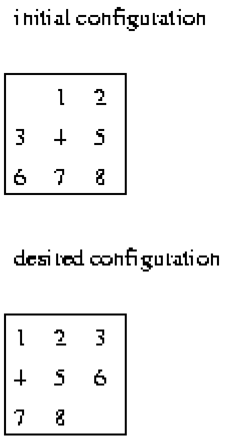
The problem is to move two blocks to the left.

A more complicated example is at lloyd2.in. It represents a 2 x 4 puzzle. Note that two of the pieces have the same letter. When two (or more) pieces have the same letter you should consider configurations identical even if two identically labelled pieces are interchanged.



A Graphical Representation of lloyd2.in

A more complicated example is at lloyd3.in.



A Graphical Representation of lloyd3.in

The object of this puzzle is to keep the numbers in order but have the empty space at the bottom-right instead of the top-left.

Submission Details

The program is to be called **lloyd**, which means the `main` function should be defined in a file named **`lloyd.cpp`**. As submitted, the program must print out the solution by listing the sequence of configurations needed to reach the chosen goal configuration from the starting configuration. An action consists of one block moving one square up, down, left, or right. For example, moving up 3 positions would be considered 3 actions.

Note that there is a possibility that no solution exists. If that is the case for a particular input, the program should print, "`no solution exists`" on the output (file or standard out), and then exit.

If you have modified the design, you must submit an explanation of changes in a file named **`readme`**.

Configuration Design Suggestions

The world is now more complicated. You may recall that one of the framework approaches was to represent the configurations as a vector of integers. Even if you choose another design, you can still put a vector of integers into your configuration class. For this puzzle, a 2D matrix might be easier to work with. Think about indexing a single vector with an accessing function to represent a 2-d matrix with a 1-d vector. You could use the character codes as the integers in your vector. You could then cast the integers to `char` for printing.

Note that there is no need to distinguish between blocks with the same label.

How To Submit

You must submit all the `.cpp` and `.h` files required to build the **lloyd** *and* **water** and **clock** programs. It must be possible to compile all three programs by executing **gmakemake** and then simply **make**. If your underlying design changed, include the `readme` file mentioned above. (If you had to change your design, then you probably need to update the other two programs so that they continue to work.) The submission must be from your team account.

```
submit swm-grd project3 clock.cpp water.cpp lloyd.cpp other-needed-code-files
readme
```

Part 4

Part 4 is due on 08 December 2014.

This activity is to be done individually. You may use all of the files that you generated with your team but you are not allowed to share any code you write for this particular activity with anyone. You may use the same framework you developed for the previous submission to solve a variant of the clock puzzle.

You will need to copy the files from the team account to your personal account before working on this submission. You will then, in your own account, write the code necessary to solve the variant clock puzzle.

This puzzle is similar to the first puzzle except that instead of being able to move the clock forward and back one hour there are additional command line arguments (which may be positive or negative) that indicate how many hours the clock can be advanced (or set back if negative). Only values indicated may be used. Specifically, the first command line argument is the number of hours on the clock, the second is the starting time, the third is the desired ending time, and the rest of the arguments are the values that the clock may be advanced or set back (if negative). The solution of the puzzle is to get to the desired ending time using the fewest number of steps as determined by any combination of increments given as the command line values.

You may use the original clock code as a starting point.

You should add any comments to the readme file regarding changes to the design (you are still allowed to change the design).

You will also submit a team member evaluation form. Only a textual evaluation form is submitted. It is available as team-evaluation in the web directory for this project.

How To Submit

This submission *must* be submitted from your own account. There should be no code specific to the variant clock puzzle in the team account. From *your own account*, make your submission as follows:

```
submit -v swm-grd project4 clock.cpp water.cpp lloyd.cpp vclock.cpp other-
needed-code-files readme team-evaluation
```