Lab Title: Simple Sort Puzzle Game
Course: CS 1410 – Object Oriented Programming

Objectives: object-oriented programming, single class program, objects, object instantiation, data members, constructors, member functions, arrays, logic.

Overview

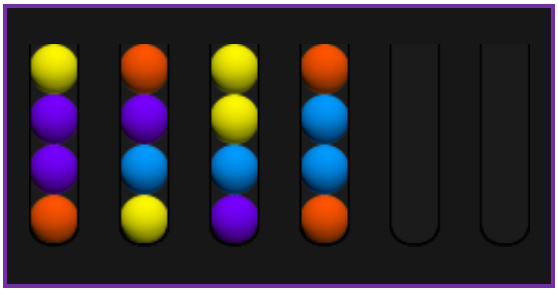
In this lab, you will create a simple sort puzzle game. The purpose of this lab is to practice object-oriented programming with a single class file. The project will consist of programming a single class from a UML diagram, including data members, constructors, and member functions, creating instances of the class from a driver for use within a game, and programming game logic using class instances. Other properties of classes, including use of the `this` keyword, will be employed.

Sort Puzzle Game Rules

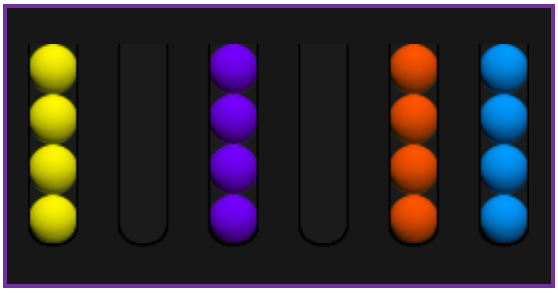
For the unfamiliar reader, a brief explanation of sort puzzles, and generic rules for sort puzzles, ensues. It is recognized that not all sort puzzles follow the same rulesets.

In a sort puzzle, the player is presented with a set of containers, usually depicted as vials. Each container is filled, completely or partially, with objects—often balls, liquids, or sand—that vary by color. The puzzle starts with the objects scrambled. The objective of the game is to group all objects of the same color into a single container. A sample puzzle and solution are provided.

Sample sort puzzle:



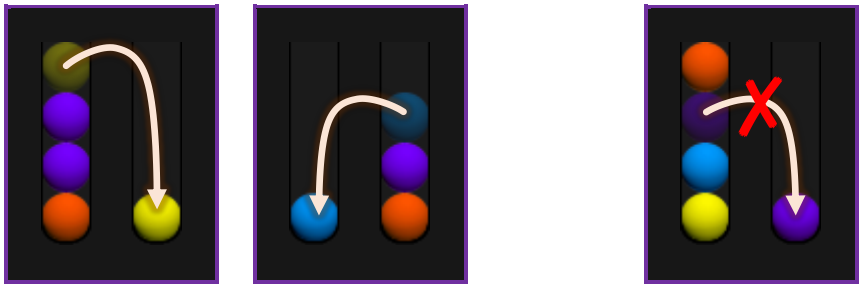
Possible solution:



The rules for movement are relatively simple. An object can be moved from one container to any other container, given the following constraints.

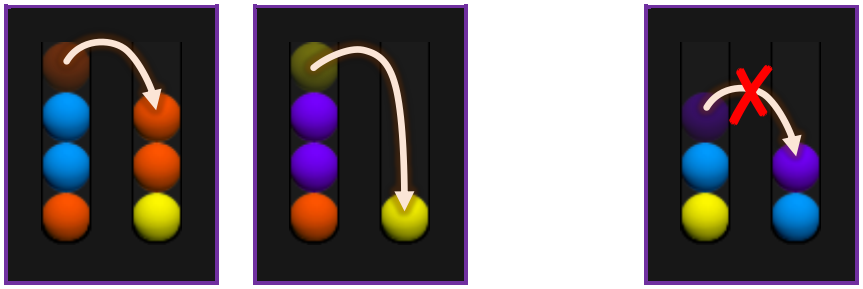
1. An object can only be moved if no other objects are above it.

Valid moves: *Not valid:*



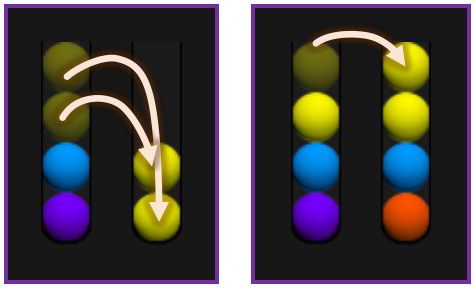
2. An object (or group of objects) of one color can only be placed atop an object of the same color, or within an empty container.

Valid moves: *Not valid:*



3. More than one object can be moved at a time if all objects in the group are the same color, and the destination container has sufficient space. Groups may be separated if the destination container has insufficient space.

Valid moves:



4. An object cannot be placed in a container that is already full.

The documentation for the puzzle sort program is divided into three parts. In the first part, the primary class is depicted in a simple UML diagram. In the second part, the data members, constructors, and member functions for the class are described in context. The third part of the project consists of programming the game logic. Read each part and complete the steps as described.

Part 1: Vial Class UML

In this part of the project, the skeletal structure for the primary class used in the sort puzzle program is defined. Create the class, data members, constructor(s), and member functions as outlined. These will be described and elaborated upon in the next section.

Code the `Vial` class using the UML diagram provided below:

Vial
- contents : char [] - filled : int - label : int
+ Vial() + Vial(contents : char[], size : int, filled : int) + Vial(c1 : char, c2 : char, c3 : char, c4 : char, filled : int) + add(content : char) : bool + display() : void + isComplete() : bool + setLabel(label : int) + transfer(destination : Vial&) : bool

Part 2: Completing the Vial Class

In this section, the data members, constructor(s), and member functions for the `Vial` class (introduced in Part 1) will be explained. Read each description and complete the code as described.

Data members

```
- contents : char []
```

The contents array of type char which will be used to house the “contents” in the vial. (In this console-based rendition of a sort-puzzle, the vial contents will be characters, such as letters, numbers, ASCII, etc.). You may want to initialize the contents array with spaces, which can be done upon declaration or within the default constructor for the class.

`- filled : int`

The filled variable will be used to track how much of the vial is "filled" with contents. For example, if a vial instance has nothing but spaces, the filled index would be 0, if a vial instance has two characters and two spaces, the filled index would be 2, and so on.

`- label : int`

Each vial instance will require a label for the player's reference. The label is listed in this document as an integer type; however, any type would work (string, char, etc.).

Below, examples of the use of these data members are depicted alongside a visual representation of their respective `Vial` class instances, for clarity.

Vial Instance	contents[]	filled	label
1 abb	'a', 'b', 'b', ' '	3	2
2 c	'c', ' ', ' ', ' '	1	2
3 bccd	'b', 'c', 'c', 'd'	4	3
4 bc	'b', 'c', ' ', ' '	2	4

Constructors

The UML diagram in Part 1 listed three possible constructors for the `Vial` class. In fact, none of these are required—as you can instantiate class objects using the default constructor—and you will need to make some decisions regarding how you will implement your project. Essentially, each instance of the `Vial` class will need to be filled, either with character contents, or spaces, or both, and you must decide how to do that in Part 3. An example of the usage for each constructor from the UML diagram is provided below. You may choose to implement any or all of these, or you may choose to design your own constructor(s).

Default constructor, filled with add member function: `Vial()`

```
Vial vial = Vial();
vial.add('a');
vial.add('b');
vial.add('c');
vial.add('d');
```

Parameterized constructor: `Vial(contents : char[], size : int, filled : int)`

```
char contents[] = { 'a', 'b', 'c', 'd' };
Vial vial = Vial(contents, 4, 4);
```

Parameterized constructor: `Vial (c1 : char, c2 : char, c3 : char, c4 : char, filled : int)`

```
Vial vial = Vial('a', 'b', 'c', 'd', 4);
```

Member Functions

The UML diagram from Part 1 depicted five member functions for the `Vial` class. Depending on your project implementation, not all these functions may be needed. For example, if you choose to fill your `contents` array or provide a label via the constructor, you may not need the respective `add` or `setLabel` functions. Conversely, you may wish to include additional member functions that are not listed in the diagram. Provided next is a description of each of the member functions. Program each as described, if needed.

`+ add(content : char) : bool`

The add function is used to add a new character to the `contents` array of the `Vial` instance. The return type of `bool` signals to the caller whether the add operation was a success. The process is relatively simple. If the `contents` array is not filled, add the passed-in character to the end of the array using `filled` as an index. Increment `filled` and return `true`. If the contents array is already filled, return `false`.

`+ display() : void`

The display function is used to display the `contents` array, in decorated fashion, onto the console. For simplicity, you may also wish to include the label on the display (although this can be done separately if desired). You may design your vial as you see fit, using keyboard or other characters. The only real requirement is that it displays the entire `contents` array. A few possible output examples are provided.

```
+ isComplete() : bool
```

The `isComplete` function is used to test if a vial has been correctly sorted by the player. The function should return `true` if all the characters in the `contents` array are the same. If any characters are different, the function should return `false`.

```
+ setLabel(label : int)
```

This simple setter function is used to set the label of `Vial` instance. The label is how the player will reference the vial. In this documentation, an integer type is used for simplicity. However, if desired, a `char` or other type may be used.

```
+ transfer(destination : Vial&) : bool
```

The `transfer` function is used to move some of the contents from one vial to another vial. The source vial is the current instance of the `Vial` object (`this`), and the destination vial is passed as an argument to the `transfer` function via reference. The function will need to perform some checks to ensure valid operations. The return type of the function is `bool` to inform the caller if a valid transfer occurred. A summary of the function logic is provided:

1. If the source vial (`this`) is empty, state that the source is empty and return `false`.
2. If the destination vial is filled, state that the destination vial is filled and return `false`.
3. If the last available item in the source vial does not match the last available item in the destination vial, state that the contents do not match and return `false`.
4. If all the above checks pass, copy the last item of the source vial into the first available position of the destination vial. Replace the last item in the source vial with a space. Reduce the `filled` variable of the source vial by 1; increase the `filled` variable of the destination vial by 1.

Note: According to the game rules, more than one content item of the same value may be moved at a time. As an additional challenge, you may continue to check if the source vial contains more items of the same value and move them all to the destination vial in a single move (given sufficient space in the destination vial). This operation is not required for this project.

Part 3: Game Logic

With the `Vial` class completed, you are ready to program the game logic. A brief overview of the logic is provided, however there is some freedom in how the programming is implemented.

Essentially, you will need to decide how many distinct characters and how many vials you will use for the sort puzzle game. There should be an equal number of each type of character, and the total number of characters should evenly fill a vial. Further, you will need to provide some empty vials to allow for movement.

As an example, suppose you choose to use the characters: a, b, c, and d as the "content" characters for this project. If the `contents` array of your `Vial` class holds 4 characters, then you would need 4 a's, 4 b's, 4 c's, and 4 d's. If the `contents` array of your `Vial` class holds 5 characters, you would need 5 a's, 5 b's, 5 c's, and 5 d's, and so on.

You will need at least one vial for each set of characters, and at least one extra vial to allow for movement (although more than one is recommended, as too few vials can lead to the player getting stuck). Given the previous example (a, b, c, d) you would need at least 4 vials to store the characters, and at least 1 empty vial to move the characters around. If you decide to use more characters, say a, b, c, d, and e, then you would need at least 5 vials to store the characters, and at least 1 empty vial to move the characters around. It is recommended, but not required, that you instantiate the `Vial` objects as an array for easier programming operations.

Once you have the characters and the minimum number of `Vial` instances, you will need to distribute the characters randomly into the vials. There are several ways to do this. One simple approach would be to create an array with all your characters, select an index at random, and then add the character to the next available vial. Once a character has been used in the array it cannot be used again, so consider replacing it with a bogus character, such as a space or a hyphen, and checking for that bogus character before selecting it.

From here the logic is simple:

1. Display all the vials on the console.

2. Allow the player to set a source vial and a destination vial.

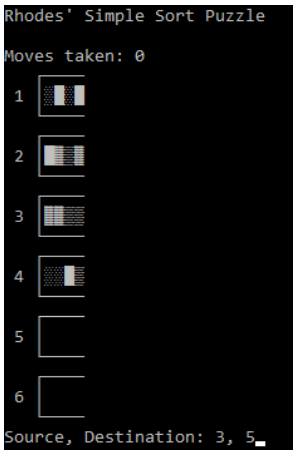
3. Use the `transfer` function to move the contents from the source vial to the destination vial (if programmed correctly, the `transfer` function from Part 2 should include all required logic checks).

4. Repeat the process until all vials are correctly filled (use the `isComplete` member function).

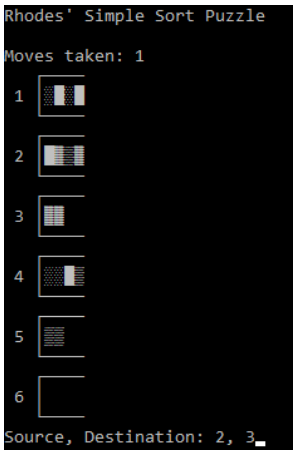
For reference, a complete sample interaction is provided here. In this example, extended ASCII characters were used as `contents` characters instead of letters.

Sample sort puzzle session:

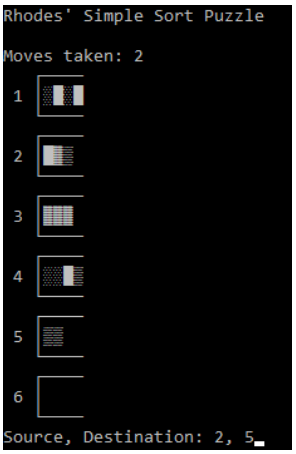
1



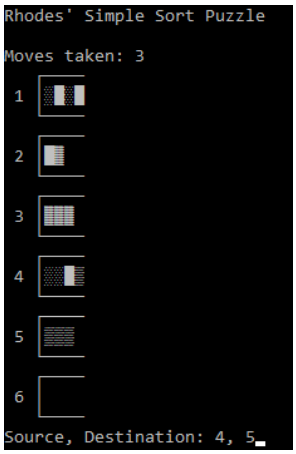
2



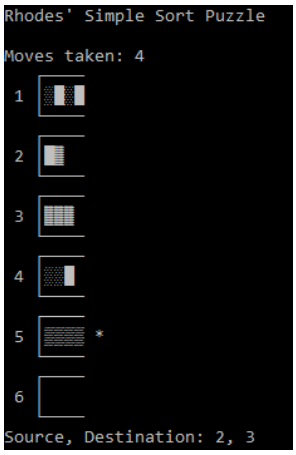
3



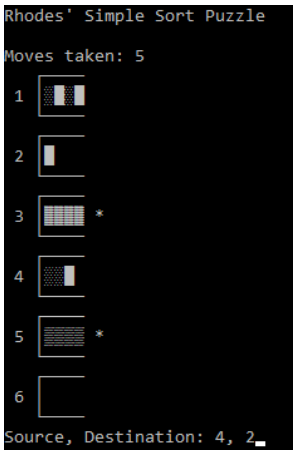
4



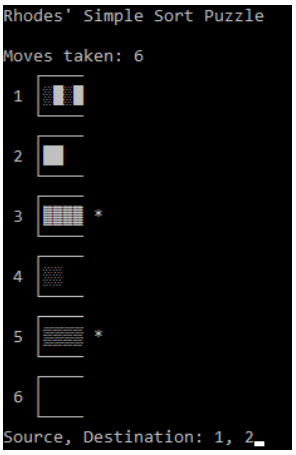
5



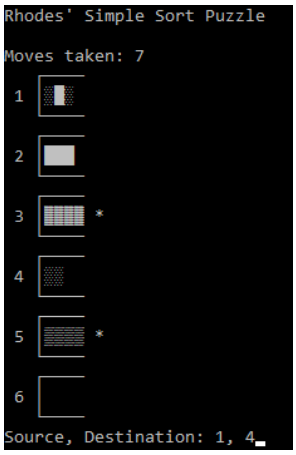
6



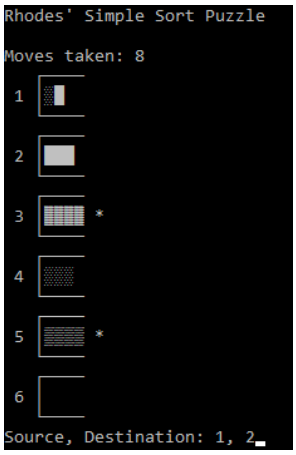
7



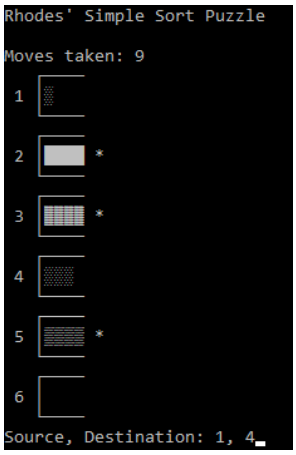
8



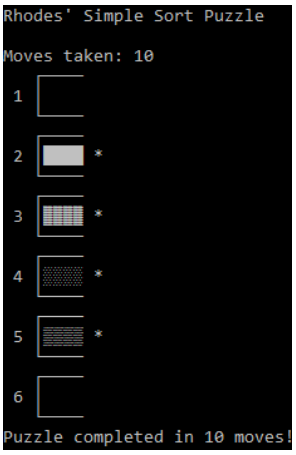
9



10

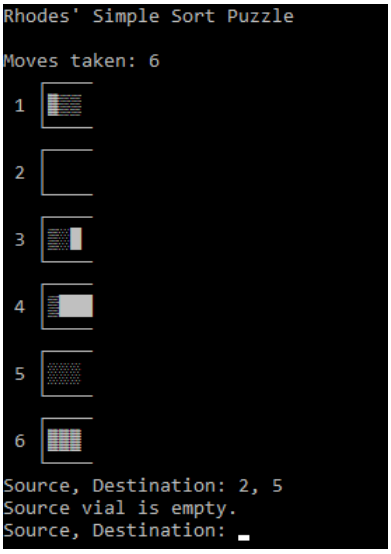


11

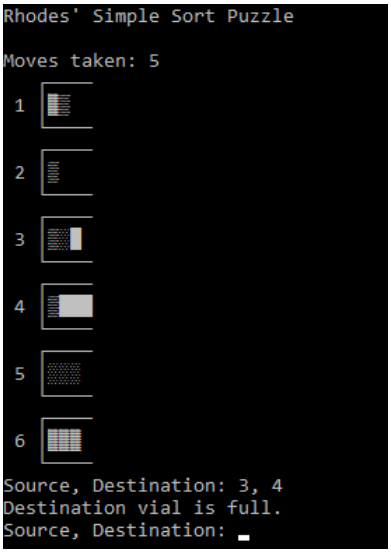


Also provided are examples of the various error conditions:

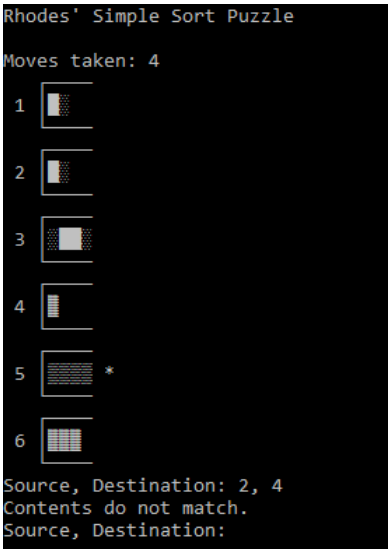
Source empty



Destination full



Mismatch contents



As always, thoroughly test your project for both completeness and mistakes. Submit the completed .cpp file online for grading. A grade rubric is provided on the next page.