Initial Assumptions

The problem of processing customer orders for the Oxford-AstraZeneca COVID-19 vaccine involves two parties – the customer and the shipping company. As this program is being developed for a specific shipping company, it is implied that all orders received have been intended for this company. Hence, the details about the shipping company can be ignored, and it is better to represent the relationship between the customer and the shipping company, which is established through orders.

Design Decisions

Naming Conventions

For this project, I decided to adopt the naming conventions adopted by the C++ standard library, as this will help standardise my code for future development in terms of maintenance and potential collaboration with other developers by maintaining consistency. The C++ standard library uses camel case (e.g., example_here) for classes, functions, and variables.

Validation

Throughout the program, I have performed validation on each input line as it is processed. Whereas the alternative method of performing validation on the entire input file before processing it has the benefit of catching errors earlier, it also requires an additional file operation to open the same input file. File operations are typically one of the most computationally costly operations, and I have assumed that the program would be run at relatively frequent intervals (weekly or monthly) to ensure that the shipping company's records remain updated, meaning the file sizes would not be extremely large.

To help the user determine where validation failed, I have outputted all the reasons why it failed the validation, and the line from the input file which caused the error. I specifically sought to perform full validation checks even if an error is found at the start of the validation process, as this means that the user can obtain feedback about lines which fail validation in multiple ways, rather than fixing errors on a line one-by-one.

Design Patterns

I considered using the Observer design pattern in my project, as this problem initially appeared to match the style of problem which this pattern is suited at solving. However, I decided not to implement this because it would have added unnecessary complexity to the final solution, which I believe is currently very simple and easy to understand. The added complexity would have outweighed the benefit of looser coupling between the publisher (the shipping company) and the subscribers (the customers).

For a more complicated problem, the advantages provided by the Observer pattern may have been worth the added complexity, but the input file can currently only result in four types of actions – adding a new customer, adding a new normal order, adding a new express order, and end-of-day shipping. If this were to be the case, I would have implemented the Observer design pattern.

Project Structure

Based on my initial assumptions, I created two classes, which were *customer* and *order*, with a main program to use these classes.

customer

The *customer* class contains the private instance attributes *customer_number*, *customer_name*, *customer_order_quantity*, and *date*. Encapsulation is provided through the getter and setter methods *get_customer_number*, *get_customer_order_quantity*, and *set_date*. This makes the program easier to understand and maintain.

Meanwhile, the next invoice number is tracked using a static member variable, *invoice_number*. This is initialised to start at 1000, so that it matches the desired output from the example input file. Being static means that the space is allocated for the lifetime of the program, which is necessary, and the value of the variable is carried across multiple function calls. After an invoice is generated in the

customer::ship_order method, the invoice number is incremented to prevent issuing duplicate invoice numbers.

A *customer* object is created using a constructor which takes the input line as a parameter in the form of a string. This validates the input line with the helper function *validate_customer_input*, checking that it has a maximum length of 45 characters, and that the customer number only contains digits. An error message is created if any validation fails, and the program will end, as an invalid order likely indicates a problem with the input file itself, which needs to be fixed outside of the program. If they are valid, it abstracts the information from the input line – the customer's number is in columns 2-5, and the customer's name is in columns 6-45. Then, it stores the details in the *customer_number* and *customer_name* instance variables.

The method *ship_order* creates two messages to the standard output stream. One of these is generated from the order processing system, which specifies that a specific customer has shipped a given quantity. The other is generated from the customer, which specifies the invoice, containing the customer number, invoice number, date, and order quantity. At the end of this method, the customer order quantity is reset to 0, as the current quantity has been shipped and is no longer the responsibility of the shipping company.

add_quantity takes an order as a parameter and appends the order quantity from that order to the customer's order quantity, so that it represents the total quantity of pending orders for that customer.

order

The order class contains the private instance attributes *order_date*, *order_type*, *order_customer_number*, and *order_quantity*. For encapsulation, they each have a getter method, *get_order_date*, *get_order_type*, *get_order_customer_number*, and *get_order_quantity*.

An *order* object is created using a constructor in a similar way to the *customer* constructor. It takes the input line string as a parameter. This input line is then validated using the helper function *validate_order_input*. It validates that the line has a length of 17 characters, the date only contains digits, the type is either 'N' or 'X', the customer number only contains digits, and the order quantity only contains digits. In addition, it uses the helper function *is_valid_date* to check whether the date is valid (by checking the year, month, and day, with leap years accounted for). If all of these are valid, the details are obtained from the input line based on the columns – the order date is in columns 2-9, order type is in column 10, customer number from the order is in columns 11-14, and order quantity is in columns 15-17. These details are stored in their respective instance variables.

Main Program

At the start of the program, the parameters provided are validated. The function *validate_parameters* checks that only one argument is provided. If this is not the case, an error message is given, and the program is ended, as it means an error has been made in the operation of the program.

To store the *customer* objects, I used *set* from the STL containers. I chose this specifically because the customers should be unique, and sets automatically ensure that no duplicate objects are stored, meaning no extra processing is required. Compared to vectors, sets are much more efficient in time taken to insert an item. The time taken to insert a value in a set is proportional to log(n), whereas it is proportion to n for vectors – for insertion, sets have a time complexity of O(log n) compared to O(n) for vectors. For larger amounts of data, such as inserting 1,000,000 customers, this would take ~20 units of time using a set, compared to 1,000,000 units of time using a vector. While sets are immutable, this is not an issue, as there is no reason to change *customer* objects in my program.

The processing of the input file is handled in the *process_input_file* function. It takes the file name provided by the program parameter, and attempts to open that file. If it cannot open this file, it returns an exception as an error message. Then, it iterates through each line in the file using the standard *getline* function. It will either process a customer record, sales order, or end the day depending on whether the first letter of the line contains 'C', 'S', or 'E'. It processes these by calling the *process_customer_record*, *process_sales_order*, and *process_end_of_day* functions. If the line does

not begin with any of the letters, it will give an error message, and end the program, as this would mean that the input file has not been formatted correctly.

process_customer_record creates a new *customer* object by passing the input line to the constructor, and then inserts it into the customer records. It creates a system message to state that the customer has been added, referencing the customer number using the getter method *get_customer_number*.

process_sales_order creates a new *order* object by passing the input line to the constructor. Then, it calls the helper function, *process_order_details*, to record the date and quantity of the order, and decide what to do with the order. This helper function goes through all the customer records until it finds a customer number which matches the one in the order. Then, it sets the date of the order, and adds the quantity to the customer's total unshipped quantity. Next, it outputs the details of that order to confirm that it has processed the order. The function also uses the marking of 'N' or 'X' to determine whether the order is a normal one or an express order. If it is an express order, it must be shipped immediately rather than at the end of the day, so it calls the method *ship_order* for the customer. If it finds a matching customer number, it returns *true*, which also stops the function from continuing to search for customer records. Once the function has iterated over all the customer records, it returns *false*.

If *process_order_details* returned false, an error message is returned to the user. It then stops the program, as an error here means that the customer number contained in the order was invalid, and that there must have been an error in creating the order.

ProcessEndOfDay starts by validating the input through the helper function, *validate_end_of_day_input*. This helper function validates for the length of the input line – it should always have a length of 9 characters. It also checks that the date only contains digits, and that the date is valid. If any of these validation checks fail, then the function quits the program, as this indicates that there must have been a mistake in the input file.

After validation has passed, the date completed is obtained from columns 2-9 of the input line, and a system message is created to state that the end of the day has been processed. At the end of the function, the helper function, *ship_pending_orders,* is called. This checks the customer records for any customers who have unshipped orders, and ships these orders by calling the method *customer::ship_order* for each customer affected.

Once all input lines have been processed in *process_input_file*, and the input file has been closed, all remaining memory allocated for each customer in the record is freed using the function *free_allocated_customer_memory*. This goes through the *customer_record* set, and frees memory allocated for each customer using the *delete* keyword. Finally, the successful end of the program is signalled by returning *EXIT_SUCCESS*.

Valgrind

To ensure that memory was correctly being freed, and that my program would not lead to memory leaks, I ran multiple Valgrind memory checks. As shown in the results below, my program does not have any problems with memory leaks.

```
==31732== HEAP SUMMARY:
==31732==     in use at exit: 0 bytes in 0 blocks
==31732==   total heap usage: 119 allocs, 119 frees, 12,774 bytes allocated
==31732==
==31732== All heap blocks were freed -- no leaks are possible
==31732==
==31732== For counts of detected and suppressed errors, rerun with: -v
==31732== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```