**Customer-Ordering-System** Summary Report

Methodology

I decided to implement a vector data structure to store customer objects. I had previously considered doing this using a list template however, because customers would be inserted and read in order the time complexity of appending a new customer to the vector is identical to that of inserting at any position in a list O(1).

This would not be the case if customers were inserted into the middle of the vector as the speed depends on the number of elements after the point of insertion. If we were to go this route, we might prefer to use a list which would allow for inserting or deleting elements anywhere with constant speed. The reason behind this is that if one wants to add an element in the middle of a vector - all the elements that come after that element will have to be re-allocated to make room for the new entry.

In this example case for a small number of customers this would not be a significant hindrance to performance however, for records with a large number of customers where strict ordering is required the processing speed would be greatly reduced.

Besides the speed considerations vectors also have a number of other key advantages over using lists. Among these are the fact that they handle memory allocation rather than requiring explicit memory management. This allows the data structure to be deallocated automatically when it falls out of the current scope.

Vectors also have the benefit of storing elements in non-contiguous memory, this means they are less likely to run into problems with memory availability.

Originally, I had considered reading in each line of the file and storing these as strings in a vector. I would later then traverse the vector and applying a set of validation functions to check each line before performing a second run to enact the appropriate processes relating to each record as required.

This system would be appropriate for large order files where errors in the file formatting could be caught early on as opposed to midway through processing the file. Instead, I decided that I could improve the time complexity by validating each line as I was processing the file thereby halving the time taken to handle each record. This was deemed appropriate as the order file sizes are likely to be relatively small, perhaps being generated either weekly or monthly. Therefore, it is a better use of resources to avoid reading each line twice and instead attempt to validate every line as it is processed. This would also save the need to create a vector solely for the purpose of storing the data that is already contained in the file.

Furthermore, we know that file operations are relatively slow compared to retrieving values for the data structure so it would not be a good idea to open the file multiple times for validation and processing. My system only reads the file once and is more memory efficient than trying to avoid reading through the file twice by having the aforementioned vector.

My program makes use of a single vector to store pointers to customer objects of this class type. In evaluating customers stored in this data structure I decided against creating a vector iterator in favour of directly indexing the desired object. I felt that this would be better from a readability standpoint and that the iterator would be surplus to requirements. This was best demonstrated when increasing customer’s order quantities in that a simple *for* loop going over each element was sufficient to quickly assign new orders to customers.

By using pointers, I can update values stored within the customer objects. This is vital for accurately tracking order totals and amending the order date.

Previously I had experimented with creating an overloaded operator to handle adding order quantities to customer objects. However, I later found that I could use a single setter method to both increase order totals and reset the counter whenever an end-of-day or express sales order record occurred after generating the invoice for these. With an overloaded operator such as ‘+=’, which in this instance I believe would logically make the most sense, I would only be able to increase the tally, requiring an additional method to reset the totals.

Classes

I created a Customer class that acts as the base class from which Invoice is derived. This required me to set the base destructor to *virtual* in order to correctly free the memory associated with Invoice. By deriving invoice from customer, I can inherit all data contained with a customer object and append *static* invoice numbers. These invoice numbers begin at 1000 and are incremented by one with each invoice generated hence the need to keep track of the current value using *static* datatypes. The static keyword ensures that the variable is not deleted when the function ends.

An alternative approach would be to initialise each customer class with an invoice number that is only assigned at the time the invoice is generated. While this is a perfectly valid approach, I prefer my method as it creates a clear distinction between the Customer and Invoice object types.

In making the customer class I decided to overload the default constructor to instead take the string new customer record so that attributes such as CustomerNum, customerName, quantityOrdered and date can all be assigned when the customer object is created, without needing to call four different setter methods individually.

Finally, I have a sales order class that contains data relating to individual orders such as which customer an order belongs to as well as the quantity, date and whether the order is of the express type. Again, these values are assigned during initialisation using the sales order record as part of the constructor method.

Project Structure

As part of my design decision, I decided to create separate modules to handle all the functionality associated with customers, sales orders, invoices, record validation, record processing and file handling as well as the ordering main function. The class definitions are stored in the respective header files for each of the three types of objects. This allows for a clear separation between the class declarations and their implementation while allowing class definitions to be reused across multiple files.

The ProcessRecord header includes macros for #define directives which related to the three types of record present in the order file as well as the two kinds of orders. This is better for clarity as references in the source code that make use of these definitions can be better understood whereas comparisons between these characters could otherwise result in ambiguity.

The Customer, SalesOrder and Invoice modules contain their respective getter and setter method for managing values stored with the class object of the same types.

There is a separate ReadFile module that is responsible for fileIO using file streams.

The Validation module contains functions for checking the formatting of the records.

Validation

*Below is a list of all the validation steps my program is able to perform:*

* Check the length of each of the three types of record in the order file.
* Ensure each line in the file begins with one of the three record identifiers.
* Verify the last seven digits in any order record contain only integer values.
* Validate order records are identified as either normal or express orders.
* Establish that customer number does not exceed four digits.
* Confirm that the customer name does not exceed forty digits.
* Inspect the date field in both end-of-day records and sales.
  + Certifies that the day, month, and year are all within valid ranges.