SENG1120 – Data Structures
Semester 2, 2024

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Assignment	*2*	**Due Date**	*Sunday 29 September @ 23:59*

Purpose	*To measure the student's ability to solve an underlying problem in C++ by implementing and/or using one or more specialised data structures.*

Introduction

The central branch of AlgoBank has been struggling with long lines and frustrated customers. Each day, hundreds of customers walk through the bank's doors, hoping to make quick deposits, withdrawals, or seek financial advice, but the queues have been growing uncontrollably. The branch manager, Ms. Queueworth, has a problem on her hands: how can she allocate customers efficiently to the tellers so that no one gets too impatient?

Figure 1The AlgoBank Logo

To solve this, Ms. Queueworth proposes a new system – one that can balance the customer load among tellers. This is where you come in. As an aspiring software engineer, you've been tasked with implementing a simulation that models this queuing system. Your mission is to implement a streamlined approach that minimises wait times and ensures customers are allocated fairly to the available tellers.

In this simulation, you'll be modelling the way AlgoBank's tellers handle customers. The tellers work tirelessly behind their counters, each with their own line of customers, ready to serve the next person in line as soon as they're available. Some tellers may work faster than others, but there's no way to predict exactly how long any given customer will take—this is a game of probabilities and averages.

Your job is to build a system that automatically assigns each incoming customer to the teller with the shortest queue, ensuring the branch runs as smoothly as possible under the constraints of unpredictable service times. Every second counts, and it's up to your algorithm to keep the lines moving.

But beware – the customers are impatient. If they find themselves waiting too long, they might reconsider their loyalty to AlgoBank! It's up to you to ensure that no teller is overworked, and no queue becomes so long that customers lose faith in the bank's efficiency.

Your simulation will model the flow of customers and the performance of tellers. With enough precision and insight, you might just help Ms. Queueworth turn things around for AlgoBank!

Assignment Overview

This assignment will test your ability to implement a (linked) queue as well as use the queue functionality to simulate a queuing system for customers at a bank branch. In summary, you will build a simulation of a queuing system where there are a set number of available bank tellers, each with their own customer queue. When customers arrive, they are allocated to the shortest queue (i.e., the teller with the least number of waiting customers). When the teller is free, they will serve the customer for a pre-determined amount of time, before updating their status to free and serving another customer (if one is waiting).

We will model a system that primarily consists of tellers, each containing their own queue of customers, and a bank branch that is effectively a collection (vector) of tellers. Initially, all tellers are free and the first customer to arrive will immediately be placed into the queue for, and subsequently served by, the first teller. More broadly, when customers arrive, they will be placed into the shortest queue among all the tellers. **Note**: this may be the queue for a teller that is busy – we do not perform any "intelligent" checks that could, for example, have a customer enter a queue for a teller that is free. Rather, the customer simply joins the first available queue that has the shortest length. To appropriately model the queueing system, we will need to know the number of tellers and the average time between the arrivals of patients. The length of a customer's service will be randomly generated by the program – this code is provided to you already.

This assignment is worth 100 marks and accounts for 10% of your final grade. Late submissions are subject to the rules specified in the Course Outline.

The Supplied Files

This section gives an overview of the files that you are provided as part of this assignment. **None of the provided files should be modified.** In contrast to Assignment 1, you are not provided with (skeleton) implementation files – you will be expected to create these files from scratch.

You are recommended to take some time to understand the overall structure of the program before starting the implementation phase. **Note:** In some cases, there are methods implemented in the supplied header files – this is done so that you don't have to worry about implementing them. You will be expected to use these methods so be sure to check them out!

- **`main.cpp`** – contains the main function, including the logic to parse the arguments and start the simulation.
- **`queue.h`** – contains the header for the `queue` class, which includes the instance variables and behaviours that your queue should contain.
- **`customer.h`** – contains the header for the `customer` class, which includes the instance variables and behaviours that a customer should contain.

- **`teller.h`** – contains the header for the `teller` class, which includes the instance variables and behaviours that a teller should contain.

- **`bank_branch.h`** – contains the header for the `bank_branch` class, which includes the instance variables and behaviours that the bank branch should contain.

- **`simulation.h`** – contains the header for the `simulation` class, which includes the instance variables and behaviours that your simulation should contain.

- **`makefile`** – the **`makefile`** for the project, which outlines the build recipe. The **`-g`** flag is set for you to make debugging and/or memory leak detection more convenient, should you wish to use these tools.

Running the Program

Of course, you will need to ensure the program is compiled prior to running it. You can use the supplied **`makefile`** for this – it will produce an executable called `Simulation`. The program should be executed using the command:

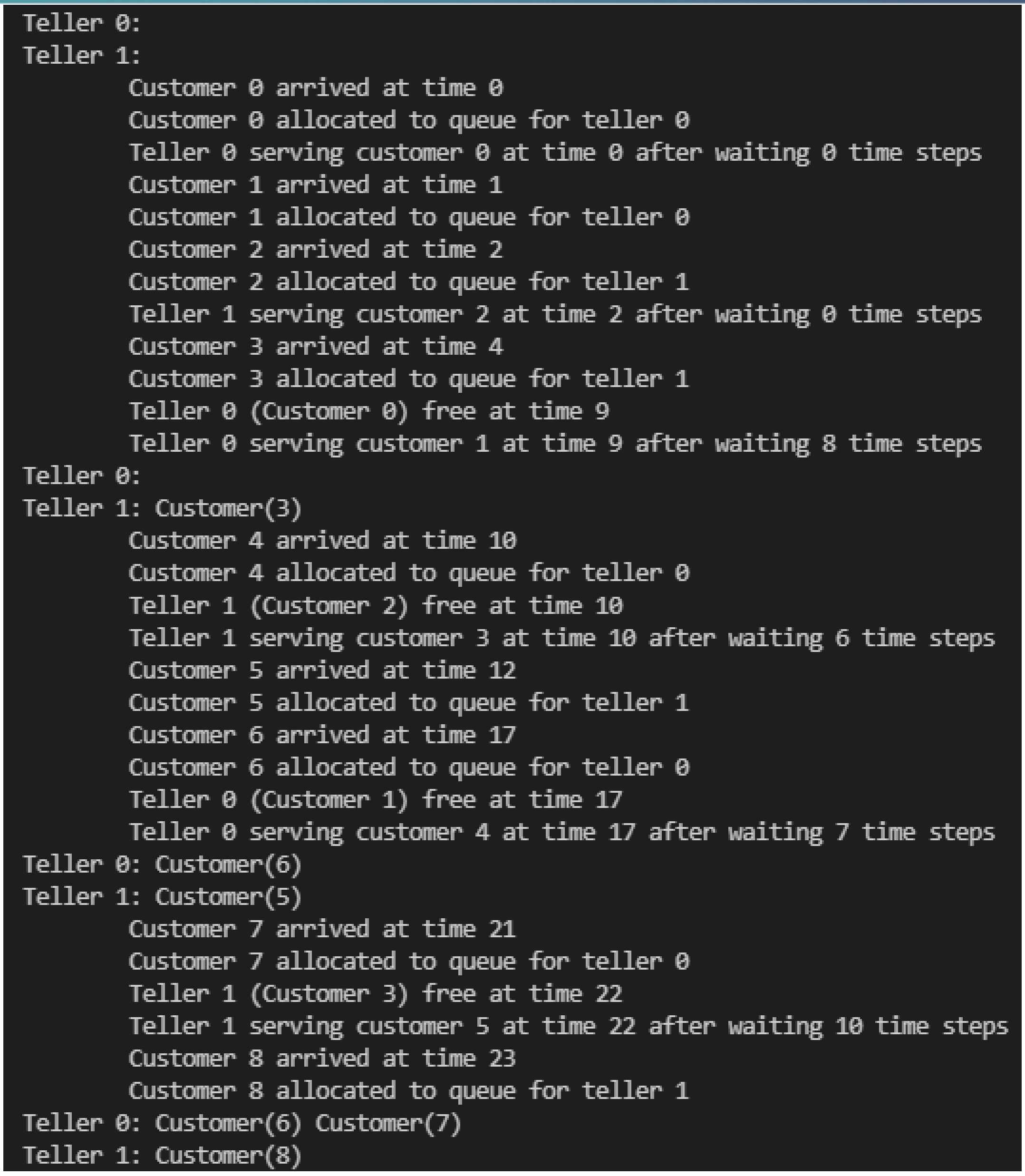
```
./Simulation <seed> <sim_time> <num_tellers> <time_between_arrivals>
```

Each of the arguments are integer values and this means that you need to supply various integer arguments to the program for it to run. For example, you can run it using:

```
./Simulation 0 25 2 3
```

To start a simulation with a random seed of 0, 25 timesteps, 2 tellers, and an average time between customers of 3 timesteps. You are not responsible for parsing of the arguments, nor are you expected to make the program work with incorrect arguments – your program will only be tested with valid integer inputs.

The output of this simulation should look like the following:



```
Teller 0:
Teller 1:
        Customer 0 arrived at time 0
        Customer 0 allocated to queue for teller 0
        Teller 0 serving customer 0 at time 0 after waiting 0 time steps
        Customer 1 arrived at time 1
        Customer 1 allocated to queue for teller 0
        Customer 2 arrived at time 2
        Customer 2 allocated to queue for teller 1
        Teller 1 serving customer 2 at time 2 after waiting 0 time steps
        Customer 3 arrived at time 4
        Customer 3 allocated to queue for teller 1
        Teller 0 (Customer 0) free at time 9
        Teller 0 serving customer 1 at time 9 after waiting 8 time steps
Teller 0:
Teller 1: Customer(3)
        Customer 4 arrived at time 10
        Customer 4 allocated to queue for teller 0
        Teller 1 (Customer 2) free at time 10
        Teller 1 serving customer 3 at time 10 after waiting 6 time steps
        Customer 5 arrived at time 12
        Customer 5 allocated to queue for teller 1
        Customer 6 arrived at time 17
        Customer 6 allocated to queue for teller 0
        Teller 0 (Customer 1) free at time 17
        Teller 0 serving customer 4 at time 17 after waiting 7 time steps
Teller 0: Customer(6)
Teller 1: Customer(5)
        Customer 7 arrived at time 21
        Customer 7 allocated to queue for teller 0
        Teller 1 (Customer 3) free at time 22
        Teller 1 serving customer 5 at time 22 after waiting 10 time steps
        Customer 8 arrived at time 23
        Customer 8 allocated to queue for teller 1
Teller 0: Customer(6) Customer(7)
Teller 1: Customer(8)
```

Note: the random seed will change the arrival time and service time of customers. Running the simulation with the same seed (and all other parameters the same) will produce the same output, while running the simulation with a different seed (and all other parameters the same) will likely produce different output, though not always. This is because the seed controls the sequence of numbers that are generated as “random” – we use this to reproduce the output and make testing easier.

Note: the program will not compile as supplied, as you will not have the necessary implementation files. You are encouraged to write skeleton files, similar to those in Assignment 1, to allow compilation of an incomplete program – this will allow you to work incrementally.

Part 1: Implementation

The first implementation task is to build a templated queue using a linked list as the underlying collection. As the purpose of this assignment is to focus on the behaviour of the queue data structure, you are expected to use **std::list** (std::list - cppreference.com) to provide the implementation of the linked list. However**, you are not permitted to use std::queue**. This provides valuable experience working with an existing collection from the standard template library.

Next, you will implement the various data classes, namely customer, teller, and bank_branch that support the simulation. Then, you will implement the simulation class, which executes the simulation. Further details about each class are given below.

queue

The queue class is a templated queue and should be implemented as discussed in lecture, noting that you are required to use std::list as the underlying list. For a full list of methods required and some important details, you should examine the **queue.h** file and its associated documentation.

customer

Each customer has an ID number, arrival time, waiting time, and service time. The ID numbers will just be set in increasing order (i.e., the first customer to arrive is given ID 0, the second is given ID 1, etc.). The arrival time is the timestep in which the customer arrives at the bank. The service time will be randomly generated when the customer arrives – you will use the generate_random_service_time function in **simulation.h** to generate this value.

The basic operations that must be performed are as follows: get the customer ID number, arrival time, current waiting time, and service time; and increment the waiting time by one time unit. For a full list of methods required and some important details, you should examine the **customer.h** file and its associated documentation.

teller

Like customers, the ID numbers for tellers will be allocated in increasing order of their creation (i.e., the first teller is given ID 0, the second is given ID 1, etc.). At any given time, a teller is either busy serving a customer or is free – we will use a Boolean variable to indicate their status. Each teller has their own, independent queue of customers and, when no longer busy, they select the next customer from the queue to serve, if their queue is not empty. The teller records the information (i.e., ID number) of the customer currently being seen, along with a record of the time remaining to serve this customer.

Some of the basic operations that must be performed by a teller are as follows: check whether the teller is free; return the remaining time to serve the current customer; return a reference to their queue of customers; add a customer to their queue; and update their information at each time step (i.e., reduce the remaining service time for the current customer, if applicable, get the next customer from their queue, if free, and update the waiting time for all customers in their queue).

When a customer is removed from the queue to be served, an appropriate message should be displayed to the console using the provided `display_customer_served` method. Similarly, when a teller has completed their service (i.e., the remaining service time reaches 0), an appropriate message should be displayed to the console using the provided `display_customer_done` method. For a full list of methods required and some important details, you should examine the **`teller.h`** file and its associated documentation.

bank_branch

The `bank_branch` is a primarily a wrapper that represents a vector of `teller` objects, providing a few helper methods to support various operations for the simulation.

Some of the operations that must be performed are as follows: allocate a customer to the teller's queue that is the shortest; update all tellers at a given time step; and display the customer queue for each teller. When a customer is allocated to a queue to be served, an appropriate message should be displayed to the console using the provided `display_customer_allocated` method. For a full list of methods required and some important details, you should examine the **`bank_branch.h`** file and its associated documentation.

simulation

The `simulation` is the driver of the simulation and handles the high-level queueing process. To run the simulation, we need to know the length of the simulation and the average time between patient arrivals. For each customer that arrives, we will randomly generate the time it takes to serve them. We will use the random number generator and a Poisson distribution to determine whether a customer arrives at each time step – this adds some randomness to our simulation! Don't worry, the code to do this is already provided for you and you should use the function `has_customer_arrived` to determine if a patient arrives at the current time step. **Note:** the argument passed to `has_customer_arrived` function should be `time_between_arrivals`, not the current time.

The bulk of the effort here will be in implementing the `run_simulation` function. The general algorithm for this function is as follows:

```
for (int time = 0; time < sim_time; time++)
{
```

- If the time step is a multiple of 10 (i.e., `time % 10 == 0`), display the content of each of the teller's queues.

- If a customer arrives (i.e., `has_customer_arrived` returns `true`), create a `customer` object with the next available ID number (ID numbers are generated sequentially), the current time step, and a random service time. This should display a message using the provided `display_customer_arrived` method, increment the number of customers by 1, then allocate the customer to a queue.
    - You should use the provided `generate_random_service_time` method to generate the service time for each customer – this method will generate a random number between 3 and 12, both inclusive.
- Update the `bank_branch` (using `update_tellers`) to ensure that each `teller` object is updated at each time step.
    - The `update_tellers` method should simply call `update` on each `teller` in the branch. The `update` method for a `teller` should follow the following process:
        - If the teller is busy, decrement the remaining service time. If the remaining service time is now <= 0, set the status as free, set the current customer ID to -1 (to signal no valid customer), then display an appropriate message to the console using the provided `display_customer_done` method.
        - If the teller is free and the customer queue is non-empty, remove a customer from the front of the queue, display an appropriate message using the provided `display_customer_served` method, update the teller's information accordingly (i.e., the remaining service time and the current customer ID), and set the teller as busy.
        - For each customer in the queue, increment the waiting time by one unit **Note**: you'll have to get clever here since you don't have access to the individual entries in the queue!

}

After the main simulation loop has ended (but before the end of `run_simulation`), you should again display the content of each queue.

As with Assignment 1, you will find more information about the required methods in the supplied hader files.

Part 2: Report

In addition to a working implementation, you will provide a written report (worth 15% of the assessment grade) that discusses and evidences your implementation and testing processes. The report should outline your use of the Problem Analysis-Coding-Execution cycle. The report is expected to be 1-2 pages in length (maximum 12-point font), but there isn't really a strict page limit. However, the report is not meant to be a major endeavour, but rather is to prompt your thinking and ensure that you have considered your solution in a bit more depth than just getting the code to compile and submitting your first draft.

Your report should include the following sections:

1. **Implementation:** While the design has been provided for you, you will discuss how you arrived at your implementation of the various classes. In particular, include a high-level description of how the code works. You should include references to the lecture material (i.e., slide numbers), where appropriate, as evidence of how you arrived at a particular implementation.

   We don't need a detailed discussion of every method, but rather a 1-2 paragraph discussion of the implementation of the entire system, largely to convince us that you understand *what* you did and *why* you did it.

2. **Testing:** Describe how you tested your implementation. Describe any test cases you conducted and discuss the results. The main point is to prompt you to consider some simple tests that cover the functionality, ensuring that the code works as expected.

   You don't necessarily need to test and report on every method but should ensure that you have convinced yourself (and the marker) that you have a working solution beyond the functionality shown in the provided **`main.cpp`**.

   This may be easiest to provide in tabular form, such as below:

Test ID	Test Description	Expected Result	Test Result
1	…	…	…

3. **Reflection:** Reflect on what you learned from this assignment and how you could improve your implementation. For example, are there areas where error checking would make sense, but were not included in the specification? Is there a different design you think would be better? Have you learned about a data structure that would be more efficient for some of the operations?

   Again, we are mainly just looking for a 1-2 paragraph summary of how your skills were improved through this assignment and if you have any ideas for ways things could be improved in your solution.

   You can find more information about reflective writing in the library guide found here: What is Reflective Writing? - Reflective writing and blogs - LibGuides at University of Newcastle Library.

Marking

Your implementation will account for 85 marks and will be assessed on both correctness and quality. This means, in addition to providing a correct solution, you are expected to provide readable code with appropriate commenting, formatting, best practices, memory management, etc. **Code that fails to compile will likely result in a zero for the functionality correctness section.** At the discretion of the marker, minor errors may be corrected (and penalised) but there is no obligation, nor should there be any expectation, that this will occur.

Your code will be tested on functionality not explicitly shown in the supplied demo file (i.e., using a different **`main.cpp`** file). Hence, you are encouraged to test broader functionality of your program before submission, particularly as you will need to discuss this in your written report.

Your report will account for 15 marks (5 marks for each of the required sections) and will be assessed on general quality and maturity of the provided information and discussions.

Marking criteria will accompany this assignment specification and will provide an indicative guideline on how you will be evaluated. **Note**: the marking criteria is subject to change, as necessary.

Submission

Your submission should be made using the Assignment 2 link in the Assignments section of the course Canvas site. Assignment submissions will not be accepted via email. Incorrectly submitted assignments may be penalised.

Your submission should include only the completed versions of **`queue.hpp`**, **`customer.cpp`**, **`teller.cpp`**, **`bank_branch.cpp`**, and **`simulation.cpp`**. You do not need to include any other files in your submission. Be sure that your code works with the supplied files. Do not change the files we have supplied as your submission will be expected to work with these files – when marking your code, we will add the required files to your code and compile it using the supplied **`makefile`**.

Compress the required files into a single **`.zip`** file, using your student number as the archive name. Do not use **`.rar`**, **`.7z`**, **`.gz`**, or any other compressed format – these will be rejected by Canvas. For example, if your student number is c9876543, you would name your submission:

`c9876543.zip`

If you submit multiple versions, Canvas will likely append your submission name with a version number (e.g., **`c9876543-1.zip`**) – this is not a concern, and you will not lose marks.

Remember that your code will conform to C++ best practices, as discussed in lecture, and should compile and run correctly using the supplied **`makefile`** in the standard environment (i.e., the Debian virtual machine). **If you have developed your solution using any other environment, be sure to test it with the VM prior to submission.** There should be no segmentation faults or memory leaks during or after the execution of the program. Please see the accompanying marking criteria for an indicative (but not final) guideline to how you will be evaluated.

Helpful Tips

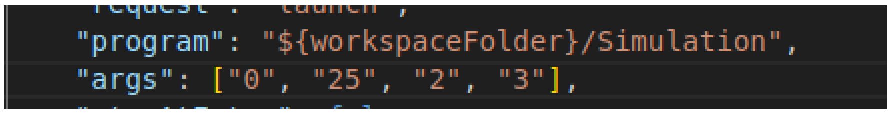
A few tips that may help you along your journey.

1. Read the header files carefully – they provide further information on the specification for various methods.

2. Work incrementally – don't try to implement everything at once. Consider getting a barebones version to compile, which will enable you to test methods as you implement them.

3. Remember that you can debug your program in VS Code. See Lab3a for a brief guide. Of course, the name of the executable in your **`launch.json`** file will be different than the example in the lab, but this gives you a great way to inspect your data structures and other objects during program execution. You will need to think carefully about what you would expect the data to look like.

   Another point is that you need to supply command-line arguments to the simulation for it to work. Otherwise, it will immediately return an error message. To do this, we need to specify the simulation parameters we want to use in our **`launch.json`** file. Each of the arguments will be a string and will be entered in the array labelled "args". For example, to run the simulation in the debugger as `./Simulation 0 25 2 3`, you would enter the following in your **`launch.json`**:

   ```
       "program": "${workspaceFolder}/Simulation",
       "args": ["0", "25", "2", "3"],
   ```

4. You will be expected to supply a program with no memory leaks. You are encouraged to use **`valgrind`** to assess whether you have any leaking memory in your program.

5. Start early! The longer you wait to start, the less time you will have to complete the assignment.

Good Luck!