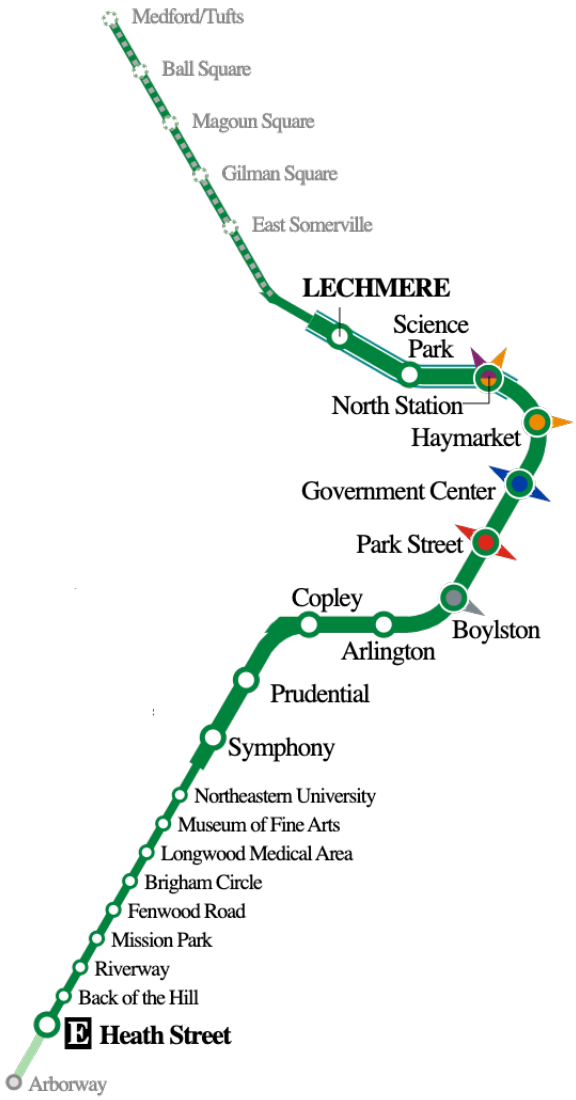
CS 15 Project 1: A Metro Simulator



Contents

Introduction **3**

Simulation Overview **4**
Data Abstraction 4
User Interface 4

Program Details **5**
Program Design 5
The Passenger 5
The PassengerQueue 6
The Train and The Stations 6

Program Flow **7**
Running From the Command Line 7
Initialization 8
Controlling the Simulation 8
Printing the State of the Simulation 9
Ending the Simulation 10
Special Note: File Input vs. std::cin 10
std::cin Example 10
File Input Example 10

Getting Started **11**
The Reference Implementation 11

Design Checkoff (Required) **11**

Phase One (Required) **12**

Testing **12**

Makefile **13**

Submitting **14**

Introduction

The Green Line has come to Tufts! In this assignment, you will create a program that simulates action on the Green Line. Your program will monitor a train as it travels a simplified course of the Green Line Extension, and will manage `Passenger`s as they board and depart at specified stops along the way. The primary, high-level tasks for this assignment are as follows:

1. Create an interactive front-end simulation that works with both files and the command line
2. Design and develop an object-oriented approach to solving the back-end logic for the simulation
3. Seamlessly connect the front-end and back-end systems

Caution! The training wheels are off! There are some requirements (see below) for your implementation, but we have given you considerably more freedom to architect your solution to this problem than in previous assignments. This is a double-edged sword! To succeed, it would be wise to read the following spec thoroughly and to start designing a solution as early as possible. This will give you time to go to office hours to discuss your plan with the course staff before you start coding. This program can be done with less code than the previous assignment **if** you have designed your program well.

Deliverables There are multiple phases of deliverables for this assignment:

- **Phase one:**
  - Design checkoff
  - Code for your `Passenger` and `PassengerQueue` classes
- **Phase two:** Your final `MetroSim` program.

Continue reading to learn more about these deliverables!

Simulation Overview

Data Abstraction

Your design choices will be of chief importance in this project. It is **your responsibility** to plan out how you will build and utilize various components of the simulation. These components include:

- `Passenger`s, which must contain:
  - an `id`
  - a `starting station`
  - an `ending station`
- `PassengerQueue`s, which contain `Passenger`s who are waiting to board or depart from trains.
- A list of `Station`s.
- A `Train`, which carries `Passenger`s between `Station`s.

While it may not be obvious at first how to model these components, we will give you some guidelines to help you establish the architecture of your `MetroSim` program. We don't require every item in the above list to be its own class, so it will be up to you to decide how each component is to be modeled. **However, we do require that you represent the `Train` as a list of `PassengerQueue` objects, where each `PassengerQueue` represents a train station**. But more on that later.

User Interface

Critical to this project is the implementation of the user-facing (front-end) command-line interface. That is to say, the user will interface with this program through the command-line, and it is your responsibility to implement the logic for that interface correctly. More detail will be provided later, but here is a brief overview of how the program will work:

- The user will run the command `./MetroSim`, and provide (among other things), a list of stations. This list will be in a file which `MetroSim` will process.
- Once you have initialized the simulation based on the provided material, `MetroSim` will process a series of commands, which will be fed to the program in one of two ways:
  1. By a file (provided at the program's start along with the station list)
  2. Through standard input, aka `std::cin`, which is the default if no file is provided.
- Every command will perform one of the following operations:
  - Add a passenger to the simulation

- Move a train to the next stop
- End the simulation

- After every command, `MetroSim` will (1) print an updated view of the train and `Station`s to `std::cout`, and (2) print a list of passengers that have left the train to a file specified by the user at the start of the program.
- When either no more input can be read from a file, or the user inputs the `m f` command, the simulation terminates successfully (and with no memory leaks!!).

Program Details

Program Design

Note: you should carefully consider your design and implementation plan before writing any code. In fact, we require you to come to office hours to review your design and implementation plan before writing any code (continue reading for details).

The Passenger

We have provided you with the interface for a `Passenger` object (within `Passenger.h`). You must use this `Passenger` interface, and *you may not modify the contents of* `Passenger.h` You must implement the `print` function for the `Passenger` interface within the provided implementation file (`Passenger.cpp`). The `Passenger` `print` function should format output as follows

```
[PASSENGER_ID, ARRIVAL->DEPARTURE]
```

where

- `PASSENGER_ID` is the `Passenger`'s `ID` (each `Passenger` receives a unique consecutive id number, starting at 1).
- `ARRIVAL` and `DEPARTURE` are the station numbers of the arrival and departure stations, respectively.

The format in which you print must match the above line exactly. Note that there is a space between ARRIVAL and the preceding comma, and that when each passenger is printed, there should be no additional whitespace outside the square brackets. Also note that the boxes surrounding output text in this document are **not** to be included in your output - they are just here to help highlight the text.

The PassengerQueue

You're required to write a `PassengerQueue` class from scratch that implements *exactly* the following interface:

- `Passenger PassengerQueue::front()`

  Returns, but does not remove, the element at the front of the queue. You may throw an error if this function is called on an empty queue, but you are not required to do so.

- `void PassengerQueue::dequeue()`

  Removes the element at the front of the queue. You may throw an error if this function is called on an empty queue, but you are not required to do so.

- `void PassengerQueue::enqueue(const Passenger &passenger)`

  Inserts a new passenger at the end of the queue.

- `int PassengerQueue::size()`

  Returns the number of elements in the queue.

- `void PassengerQueue::print(std::ostream &output)`

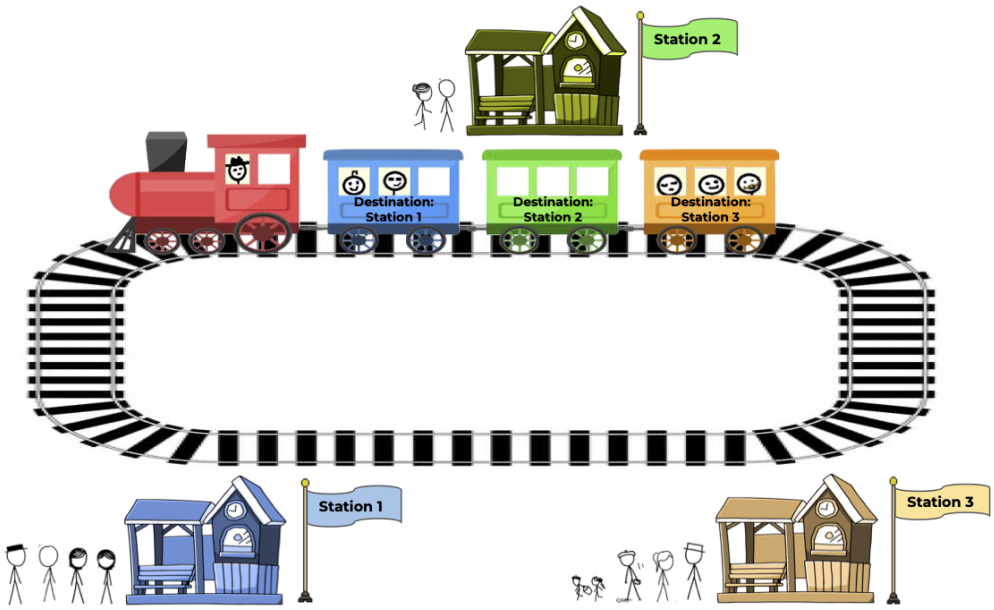
  Prints each `Passenger` in the `PassengerQueue` to the given output stream from front to back, with no spaces in between and no trailing newline. For example:

```
[1, 1->2][2, 1->3][3, 2->3][4, 2->3]
```

Make sure your `PassengerQueue` interface matches the interface listed above *exactly*. You may not add any other public functions. **You must use either `std::vector` or `std::list` to implement this class.** Note that, by default, `C++` will define a default constructor and the Big Three for you; as long as your class doesn't allocate any memory on the heap with `new`, the default constructor/Big Three will work fine for you. Code for the `Passenger` and `PassengerQueue` classes will be due at the end of week one, along with your design checkoff.

The Train and The Stations

As stated in the program overview, you are required to implement your train as a **list of** `PassengerQueue` instances (each element of the list is a `PassengerQueue`). Upon boarding the train, each passenger is organized into a `PassengerQueue` based on their destination. See the image below for a visual representation of the `Train`.



You may find that you will need to represent "lists" of various data types. For this, we require that you use `std::vector` or `std::list`, since it will greatly reduce how much code you'll need to write/rework, although you may need to do a little reading before you get started. The `std::vector` and `std::list` documentation on `https://www.cplusplus.com` is a great resource. If you feel that this reference is too verbose for your taste, feel free to seek out additional resources. As always, any material that you use (including reused code from this class) should be noted in the **Acknowledgements** section of your `README`.

Program Flow

Running From the Command Line

You will write a program called `MetroSim` that accepts either 2 or 3 command-line arguments (in addition to the program name), like this:

```
./MetroSim stationsFile outputFile [commandsFile]
```

where:

- `stationsFile` is an input file containing names of the `Stations`, one per line
- `outputFile` is the file to send simulator output to

- `commandsFile` (optional) is a second input file containing commands to execute. If this parameter is not given, then you should read input from `std::cin`. Note: it's a common convention to denote optional parameters by putting them in square brackets.

If the program is invoked with a different number of parameters you should quit execution by returning `EXIT_FAILURE` after printing the following message to `std::cerr`:

```
Usage: ./MetroSim stationsFile outputFile [commandsFile]
```

Initialization

In the initialization phase you will open the `stationsFile` and process any `Station`s you find. The `stationsFile` is in plain text, with one `Station` name per line. You may assume the `stationsFile` has at least two `Station`s, and each `Station`'s name is a string of at least one character. `Station` names may be more than one word, e. g., "South Station".

Note: do not worry about malformed files. We will not test your program with empty `Station` files or files with duplicate `Station` names, etc. Just program assuming any input files are correctly formatted. You **do**, however, have to handle cases in which a file cannot be opened (see below for the specific message and behavior).

After you have finished processing the `stationsFile`, print the `Station`s you have just read (output specifications are below). Remember to place the `Train` at the first `Station`. Initially, there are no `Passengers` in the `Train` or at any `Station`.

If your program cannot open any of the provided file(s), you should quit by returning `EXIT_FAILURE` and print the following error message to `std::cerr`

```
Error: could not open file FILENAME
```

where `FILENAME` is the name of the file that could not be opened.

We have given you sample `Station` and command files (`stations.txt` and `test_commands.txt`, respectively). We will test your code with others, and you should make other similar files for your own testing, e.g., on a shorter list of `Station`s or with a sequence of commands that causes particular code in your program to run for test purposes.

Controlling the Simulation

`MetroSim` will accept input from either a `commandsFile` that contains a list of commands, or from `std::cin` if no `commandsFile` was given.

Commands are case sensitive. You may assume input data is valid. (e.g., a `Passenger` will never want to go to a `Station` with a negative number, etc...). Your program can do anything in such cases (segfault, quit, be angry, etc.).

Your program will prompt on `std::cout` for a command by printing the text `"Command? "`), and then will read a command in and process it. Your code will handle the following commands:

- `p ARRIVAL DEPARTURE`
    - The command `p ARRIVAL DEPARTURE` adds a new `Passenger` to the simulation who boards at the `ARRIVAL Station` and departs at the `DEPARTURE Station`. `ARRIVAL` and `DEPARTURE` are specified by a number (note that the first `Station` is `Station 0`).
    - `Passenger`s are enqueued at the `ARRIVAL Station`.

- `Passenger`s should get consecutive `ID`s assigned in the order they arrive starting with the number 1 (not 0). I. e., the first `Passenger` to ever arrive will have `ID` 1, the second one `ID` 2, and so on.

- `m m` (metro move)
  - The command `m m` moves the `Train` from the current `Station` to the next one in the line. The train should move to the first `Station` if it is currently at the last `Station`.
  - As the train leaves a `Station`, all `Passenger`s at the departing `Station` will get on the train, regardless of which `Station` the `Passenger`s are going to. They board in the same order as they arrived at that `Station`.
  - When the `Train` arrives at a `Station`, any `Passenger`s whose final destination is that `Station` will exit the `Train` and the `Station`—you can remove them from your simulator. Exiting passengers *will not* be included in the next state of the simulation printed to `std::cout` (i.e., passengers exit *then* we print the state), however, their exit will be logged in the output file...
  - For each `Passenger` that exits at a `Station`, the following line should be written to the output file (**not** `std::cout`), followed by a single newline:

    ```
    Passenger ID left train at station STATION_NAME
    ```

    replacing `ID` and `STATION_NAME` with that `Passenger`'s `ID` and the `Station`'s name, respectively.
  - Note: You may notice that the list of `Station`s is circular; that is, when the train reaches the bottom of the list, it returns to the top. We recognize that you have recently learned about a particular circular data structure, but it's up to you to determine whether it's necessary to use. Be sure to consider the use cases of the data structures you implement and whether they apply to the program you build. (Feel free to check in with a TA if you're confused about this note!)

- `m f` (metro finish)
  - The command `m f` should terminate the simulation, as described in the **Ending the Simulation** section.

Printing the State of the Simulation

You should print the state of the simulation (described below) after each command that runs, according to the following format:

1. Print the text "Passengers on the train:" followed by a space, followed by a curly brackets enclosed list of passengers on the train, followed by a newline.
2. For each station on the train line, print:
   (a) `TRAIN:` followed by a space if the train is currently at that `Station`, otherwise print seven spaces.
   (b) The `Station` number enclosed in square brackets, followed by a space.
   (c) The `Station` name, followed by a space

(d) A curly brackets enclosed list of the passengers currently waiting at the `Station`, followed by a newline.

Whenever you print a `Passenger`, you should follow the format described in **The Passenger** section above.

Example output for train line with 4 `Station`s, where the `Train` is currently at `Station` 1:

```
Passengers on the train: {[1, 0->2][2, 0->3]}
       [0] station_0 {}
TRAIN: [1] station_1 {}
       [2] station_2 {[3, 2->3][4, 2->3]}
       [3] station_3 {}
```

Note that `Passenger`s on the `Train` are in **ascending** order based on the **destination** `Station`. If there's a tie, the `Passenger` that boarded the train first will precede the `Passenger` that boarded afterwards.

You can assume that each `Station`'s name has at least one character, and that there will be at least two `Station`s provided.

Note: follow the format exactly, including spaces. We strongly encourage you to run the `diff` command to compare the output of your solution with that of the reference implementation. Refer to the **Testing** section for more information.

Ending the Simulation

If your program runs out of commands in the input (end of file is reached) or the command `m f` is given, your program should print the following text to `std::cout`, followed by a single newline and terminate:

```
Thanks for playing MetroSim. Have a nice day!
```

Always remember to deallocate all previously heap-allocated memory and close all previously opened files before your program terminates!

Special Note: File Input vs. std::cin

When you look at your program's output, be aware that it will look differently depending on whether the input comes from `std::cin` or from a commands file. When a user types at the terminal, there will be a newline when they press "enter" to enter input, but no newline will be there if you take input from a file. See the examples below.

std::cin Example

```
Command? m m
Passengers on the train:
```

File Input Example

```
Command? Passengers on the train:
```

We are aware of this difference (indeed, you will notice that `the_MetroSim` behaves that way). This difference is expected, and you will not need any code to try to "correct" this.

Getting Started

You can get the starter files by running the following command. Note that the `--no-dereference` arugment will keep the reference implementation as a symbolic link; this is so if we update `the_MetroSim`, you will not have to re-copy the file. As usual, the `.` indicates the current folder you're in; feel free to change it as needed for your files. Of course, you should run this command on the halligan server.

```
cp --no-dereference /comp/15m1/files/proj_MetroSim
    /* .
```

The Reference Implementation

After running the above command, you will have access to a reference implementation of the program named `the_MetroSim`. We **strongly** encourage you to play around with it both to learn what is expected of your program and to test your your program. We will be comparing your program's output to the reference, so be sure that you are adhering to the output format. If the `diff` command reports any differences between the output of your implementation and the reference implementation, then you will lose points on the functional correctness portion of your implementation assessment. Refer to the **Testing** section in this document for more information.

The setup script makes a link to the reference implementation rather than your own copy. You use it exactly the same as you would use any other program, but this will guarantee you always use the most recent version.

Design Checkoff (Required)

Before you get started on implementing this assignment, we require that all students develop an implementation plan and have it checked off by the course staff. You must sign-up for a design meeting as described in class.

The items below are required for a good design:

- Drawings and diagrams that help you visualize the data structures and/or program flow.
- Descriptions of each class/module you will build, including the names of the functions/variables of each class.
- A rough schedule of what you plan to accomplish each day, with room for unforeseeable delays.

- A list of possible edge cases you will have to consider and will want to test.

During the check-off, we may ask you questions or give you feedback on your plan. If we feel that the design is not thorough enough, or find that there are certain elements that need reworking, wey may ask you to make some changes and come back. You will not receive full points for your design until we feel that your plan is sufficient.

Note: it is completely okay to deviate from this initial plan. In fact, we encourage you to continue to evaluate the structure of your solution and fine-tune it as you go – that's what programmers do in the real world, anyway. The purpose of this check-off is more to help you establish a game plan, as well as clear up any misconceptions you may have.

Phase One (Required)

For phase one of this project, you must complete the implementations of `Passenger` and `PassengerQueue`, and you must thoroughly test these classes. See the course calendar for the due date.

Testing

Test thoroughly and incrementally. You should be sure that your `PassengerQueue` behaves as intended before you start implementing your `MetroSim`. Your program will have many components—it will be *significantly* easier to debug issues if you test components as you implement them, rather than implementing everything then testing at the end.

Use the unit testing skills that you have developed in earlier assignments. You can place all tests in the file `unit_txests.cpp`, which we provide for you. Just make sure to `#include` any corresponding `.h` files needed for the tests, and to update the dependencies for the `unit_tests` rule in the Makefile (see Makefile section, below).

Note that many functions within your `MetroSim` implementation will likely be private; this means that you cannot test them directly from a testing function in another file. You have two options:

1. Test private functions indirectly via public functions.

2. Temporarily make the private functions public to test them, and then make them private once you have tested them (make sure you don't forget to make them private again!).

After you have written and debugged the `PassengerQueue` class and other classes, and you have an implementation of `MetroSim` that you think works as expected, the best way to test your results will be to compare the output from your implementation with the output from the reference implementation. Your output and the reference implementation's output must match **exactly**. Here are some tips to help you compare your implementation to the reference:

- Redirecting the standard output stream (`std::cout`) to a file using the `>` symbol on the command line.

  In the following example, we run `MetroSim` with `stations.txt` as the stations file, `output.txt` as the output file, and `commands.txt` as the commands file. Any output that `MetroSim` sends to `std::cout` will be saved in `stdout.txt`.

```
./MetroSim stations.txt output.txt commands.txt > stdout.txt
```

- Redirecting the contents of a file into the standard input stream (`std::cin`) using the `<` symbol on the command line.

  In the following example, we run `MetroSim` with `stations.txt` as the stations file, `output.txt` as the output file. We are **not** passing `commands.txt` as a command-line parameter; rather, we are sending the contents of the `commands.txt` file to `std::cin` of the `MetroSim` program.

```
./MetroSim stations.txt output.txt < commands.txt
```

- Redirecting both the standard output (`std::cout`) and standard input (std::cin) streams.

```
./MetroSim stations.txt output.txt < commands.txt > stdout.txt
```

  With the above example, any time that `MetroSim` tries to read from `std::cin`, it is actually reading from `commands.txt`. Any output that `MetroSim` sends to `std::cout` is saved in `stdout.txt`.

- Using `diff` command to compare the contents of two files.

  It is highly recommended that you compare your output files with that of the demo using `diff`. So, for instance, if, given the same inputs, the reference implementation produced `stdout_demo.txt` and your implementation produced `stdout_personal.txt`, you could compare them with the following command:

```
diff stdout_demo.txt stdout_personal.txt
```

  It can be difficult sometimes to understand the output of `diff`. Here is one reference that may help:
  `https://linuxize.com/post/diff-command-in-linux/`.

Makefile

Since you are in charge of the structure of your implementation, we cannot know exactly what files you will have. This means you will need to update the given `Makefile` to build your program correctly.

The `Makefile` we provide you with already includes:

- A `MetroSim` rule, with some listed dependencies, and *no recipe* (yet!)
- A rule for building `PassengerQueue.o`

- A `unit_tests` rule, with some dependencies already added, which will be used to test your C++ classes
- A `clean` rule which removes object code, temporary files, and an executable named `a.out` (if one exists)

You will need to update the `Makefile` with the following:

- Every `.cpp` file will need a corresponding `.o` rule in the Makefile. This includes `MetroSim.cpp`, `main.cpp`, `Passenger.cpp`, and any new `.cpp` files that you write. You can use the given `PassengerQueue.o` rule as guidance.
- The dependencies for the `MetroSim` and `unit_tests` rules must be updated with the new `.o` files as needed.
- You need to write the recipe for the `MetroSim` rule, which links all of the necessary .o files together

We have added `TODO` comments throughout your Makefile, corresponding to the updates listed above. You must make these updates!

Submitting

Be sure your files have header comments, and that those header comments include your name, the assignment, the date, and acknowledgements for any help you received (if not already credited in the `README` file).

For your design checkoff, there is nothing to provide, but you must have your design reviewed by a member of the course staff. Come in to office hours and see someone. Bring your design, which should include pictures of your data structures, lists of functions you will need, etc.

Submitting your code

See Canvas for instructions on how to submit your code.

Be sure your program builds correctly using the commands `make` and `make MetroSim`, because we will use those commands to build your program in testing. The freedom you have this assignment means that we cannot give you with complete `provide` command. When you submit, be sure to include every file required to compile your code, along with any tests, and your `Makefile` and `README`. You don't have to give us the reference implementation. A useful test is to make a submission directory, copy all your files in there, then run make and see if the program builds. Test it, then submit everything.