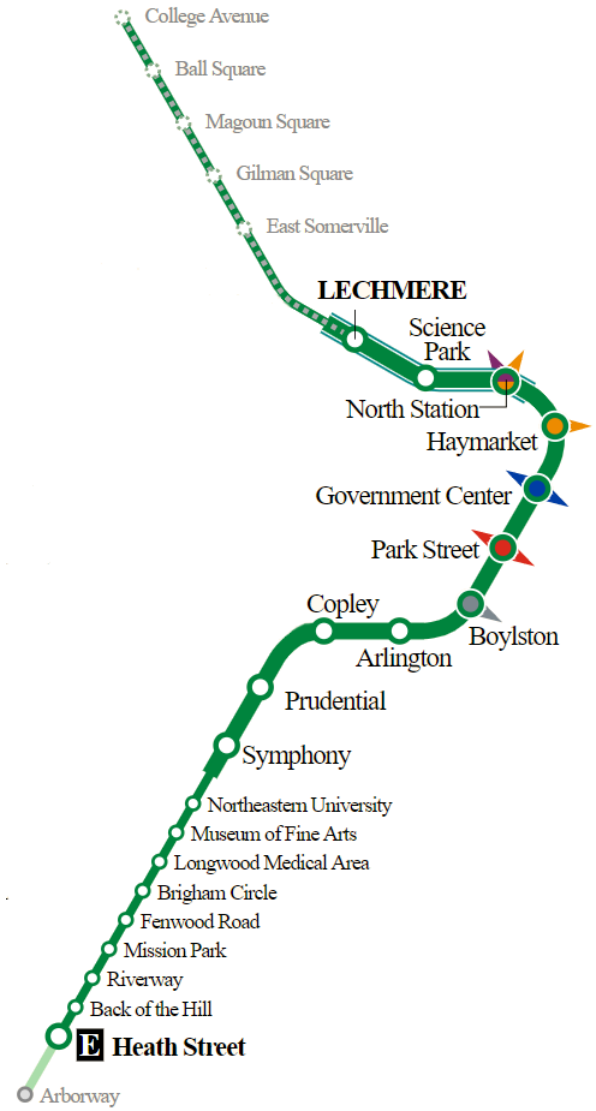
CS 15 Project 1: A Metro Simulator



Introduction

The Green Line is coming 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 `Passengers` as they board and depart at specified stops along the way.

The primary, high-level tasks for this assignment are as follows:

- Create a front-end simulation that works with both files and the command line.
- Design and develop an object-oriented solution to the simulation's back-end logic.
- Connect the front-end and back-end systems.

A few words of caution: This project is far more open-ended than the first few assignments. 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 far more ease, and far less code, *if* you have designed your program well.

Another few words of caution: This project is quite a bit more complex than the first few assignments. You will want to read through this rather lengthy document in full before starting. Things will go *very badly indeed* if you start writing code based upon the first few pages of the document without reading the entire thing, and without doing some planning in advance. You would be better off to wait on coding until you're sure you have a plan. Get answers to any questions you need answered before you hit the IDE. If you don't have a clear idea of what you plan to do in advance, there is approximately *zero* chance that your project will work out well.

Deliverables There are multiple phases of deliverables for this assignment:

- **Part one (due by 26 June):**
  - Design plan and meeting with Prof. Allen. The section on **Design Plan** (page 11) has more details about what you need to prepare for this meeting. (See Piazza for details about setting up a meeting.)
  - The code for your `Passenger` and `PassengerQueue` classes.
- **Part two (due by 03 July):** 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, each of which must contain:
  - an `id`
  - a `starting station`
  - an `ending station`
- `PassengerQueue`s that 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.

Important note: While it may not be obvious at first how to model the components listed above, the rest of this document will give you some guidelines to help you establish the architecture of your `MetroSim` program. We *do not* require that every item listed above is a separate class; instead, it is up to you to decide how each component is to be modeled.

Important exception to the previous note: We *do require* that you represent the `Train` as a list of `PassengerQueue` objects, where each `PassengerQueue` represents a train station. Keep reading this document for more on this requirement.

User Interface

Also critical to this project is the implementation of the user-facing (front-end) 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 start `MetroSim`, and provide (among other things), a list of stations. This list will be in a file which `MetroSim` will process.
- Once the simulation has been initialized based upon the user's provided input, `MetroSim` will execute a series of *commands*. These commands will be fed to the program in one of two ways:
  1. in a file the user provides at the start, along with the station list; *or*
  2. through standard input, aka `std::cin` (this is the default interface if the user does not provide a command file when they start the program).

- 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 stations to `std::cout`, and (2) print a list of the passengers who have left the train, adding that list to another file specified by the user at the start of the program.
- When no more commands can be read from the appropriate file, or the user inputs command `m f`, the simulation will terminate 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 meet with the instructor to review your design and implementation plan before submitting your first iteration of the project code (see Piazza for more 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 will need to implement the `print()` function for the `Passenger` interface within the `Passenger.cpp` file. This function should format output as follows:

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

where:

- `PASSENGER_ID` is the ID number of the passenger (each `Passenger` should receive a unique, consecutive ID number, starting at 1).
- `ARRIVAL` and `DEPARTURE` are the numbers of the arrival and departure stations.

Note: 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.

The PassengerQueue

You will write a `PassengerQueue` class that implements the following interface exactly:

- `Passenger PassengerQueue::front()`

  Returns, but does not remove, the passenger at the front of the queue.

- `void PassengerQueue::dequeue()`

  Removes the passenger at the front of the queue.

- `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(ostream &output)`

  Prints each `Passenger` in the queue to the given output stream. Passengers are printed from front to back, with no spaces in between and no trailing newline. For example, output might look like:

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

Make sure your `PassengerQueue` interface exactly matches the interface listed above. While you may as usual add as many private helper functions as suits your design, you should not add any other public functions.

Important note 01: You *are allowed* to use existing C++ data structures in your implementation. The `std::vector` or `std::list` implementations may be helpful.

Important note 02: By default, `C++` will define a nullary constructor and the Big Three for you. If the default implementations are not appropriate, then you should implement them, but think carefully first about whether or not you really need to do so.

Important reminder about deliverables

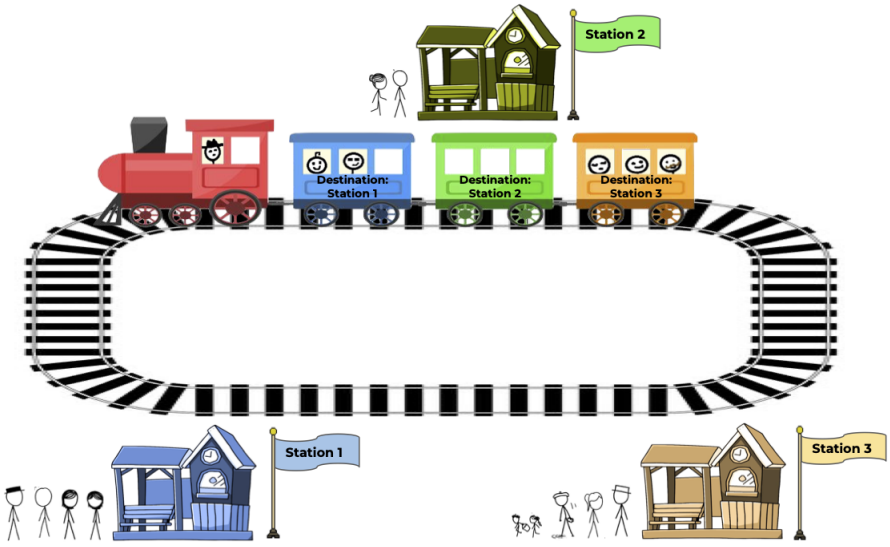
The code for your `Passenger` and `PassengerQueue` classes are due at the end of phase one of the project. If bugs remain in your phase one deliverables, you will want to rectify those ASAP in order to be able to successfully complete the remainder of the project.

During the first phase of the project, you will also need to meet with the instructor in order to review your **Design Plan** (page 11). Piazza has information about setting up this required meeting.

The Train and Stations

As stated in the program overview, you are required to implement your train as a list of `PassengerQueue` instances (i.e., each element of the list is a `PassengerQueue`). Upon boarding a train, each passenger is organized into a `PassengerQueue`, based upon their destination. That is, the first element of the train-list will be a `PassengerQueue` containing all passengers going to the first station (among all stations to which some passenger on the train is traveling), the next item will by a `PassengerQueue` containing all passengers going to the second such station, and so on.

See the image below for a visual representation. In this image, the train is at Station 2, and all passengers bound for that station have disembarked, emptying the relevant queue. The train contains two other queues, the first of which contains passengers heading to Station 1 and the second containing those headed to Station 3. Each of those latter stations has its own queue of passengers waiting to board the train when it arrives.



In implementing these list objects, you are free to re-purpose your `ArrayList` or `LinkedList` code from this class, or you may use existing data structures like `std::vector` or `std::list`. The latter will greatly reduce how much code you will need to write/rework, although you may need to do a little reading before you get started. If you go this route, the documentation on `https://www.cplusplus.com` is a great resource. 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 `Station`s, one per line;
- `outputFile` is a file to which simulator output will be sent;
- `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, as above.

If the program is invoked with a different number of parameters (i.e., less than 2 or more than 3), the program should exit after printing the following message to `std::cerr`:

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

Initialization

In the initialization phase, the program will open the `stationsFile` and process any `Station` data it contains. The `stationsFile` will be in plain text, with one `Station` name per line. You may assume the `stationsFile` has at least two `Station` names, and that each 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., and your code does not have to account for such possibilities. Write your program under the assumption that any input files are correctly formatted. You *do*, however, have to handle cases in which a file cannot be opened because the user has made a simple error like entering the wrong file-name (see below for the specific message and behavior).

After processing the `stationsFile`, the program will print the initial map with the `Station` data it has read, following the output specifications detailed below. The `Train` should start the program at the first `Station`. Initially, there will be no `Passengers` on the `Train`, nor at any `Station`.

If the program cannot open any input file, it should terminate and print the following to `std::cerr` (where `FILENAME` is the name of the first file that could not be opened):

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

The starter distribution for this project includes some sample `Station` and command files (`stations.txt` and `test_commands.txt`, respectively). We will test your code with other files, and you should make similar ones for your own testing.

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 an invalid number). Your program can do anything in cases in which invalid command inputs are given (segfault, quit, be angry, etc.).

If a command file is supplied, your program will iterate through it, processing each command in order. If no command file is supplied, then your program will use `std::cout` to prompt for a command. In *either case*, the program will print `"Command? "`), and then will read in a command—either from the file, or from user input via `std::cin`—and process it. The program 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 (i.e., in order of the `p` commands), starting with the number 1 (not 0). That is, the first `Passenger` to 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 back to the first `Station` if it is currently at the last `Station`, as if it were traveling on a circular track.
    - When the `Train` arrives at a `Station`, any `Passenger` whose final destination is that `Station` will exit—that is, they will be removed from the simulator entirely. Exiting passengers *will not* be included in the next state of the simulation printed to `std::cout`. That is, passengers exit *before* we print the state, as described in the section on **Printing the Simulation State** (page 9).

      Note: While exiting passengers do not appear in the simulation output to `std::cout`, their exit *will* be logged in the separate 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
      ```

      where `ID` and `STATION_NAME` are the `Passenger` ID and the `Station` name.
- As the train leaves a `Station`, every `Passenger` at that `Station` will get on the train, regardless of which `Station` they are going to. They board in the same order as they arrived at that `Station`. Each is added to the correct queue, as described in the section on **The Train and Stations** (page 6).

- `m f` (metro finish)
    - The command `m f` should terminate the simulation, as described in the **Ending the Simulation** section (page 10).

Printing the Simulation State

You should print the state of the simulation 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 so that the numbers of the stations all start at the same place in a line.
    (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 (page 4).

For a line with 4 stations, and the train at station 1, the output might therefore be:

```
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 again that `Passenger`s on the train are in ascending order, based upon their *destination* `Station`, as described in the section on **The Train and Stations** (page 6). If there are multiple passengers headed to the same station, they are ordered by when they boarded the train, first to last.

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 we have supplied. Refer to the **Testing** section (page 11) for more information.

Ending the Simulation

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

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

Important note: 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 different, depending on whether the input comes from `std::cin` or from a commands file. In particular, when a user types at the terminal, there will be a newline when they press "enter" or "return" to enter input, but no newline will be there if you take input from a file.

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 reference solution we supplied, `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 all the starter files you need by running the following on the homework servers:

```
/comp/15m1/files/proj01/setup
```

Note that you should *not* manually run `cp` as usual— run the above program instead. The files will be copied for you, and it will also give you access to a reference implementation of the program that you can run.

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 own version of the program. We will be comparing your program's output to the reference solution's, so you should

be sure that you are adhering to the same output format. If any differences between the output of your implementation and the reference implementation exist, you will lose points on the functional correctness portion of your implementation. Refer to the **Testing** section (page 11) for more information.

Design Plan (Required)

By the time you get started on implementing the final phase of this assignment, we require that all students develop an implementation plan and meet with the instructor to check it off. We will provide a scheduling link on Piazza so that you can schedule this meeting. Things that we would expect to see from a good design plan might include:

- Drawings and diagrams to help one visualize data structures and/or program flow.
- Descriptions of each class/module to be built, including the functions/variables of each class.
- A rough schedule of what you plan to accomplish each day of your implementation, with room for unforeseen delays.
- A list of possible test-cases you will have to consider.

During the meeting, the instructor may ask you questions or give you feedback on your plan. If they feel that the design is not thorough enough, they may ask you to make follow up with some changes. You will not receive full points for your design unless the instructor feels 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-in meeting is more to help you establish a game plan, as well as clear up any misconceptions you may have.

Testing

Test thoroughly and incrementally. You should be sure that your `PassengerQueue` behaves as intended before you start implementing your overall program `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 before testing.

You can place all tests in the file `unit_tests.h`, 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_test` rule in the Makefile, as described in the **Makefile** section (page 13).

Note that you are *not required* to use our `unit_test` framework. Alternatively, you can create your own testing file, with its own `main()` function that calls out to testing functions. Then, you'll have to compile your code with the testing `main` function instead of the simulator `main`. The choice of how to test your program is up to you—but either way, you must test your program!

Also 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 (don't forget this!).

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 should match exactly. Here are some tips to help you:

- Redirect standard output (`std::cout`) to a file using the `>` symbol.

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

  In this 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 not appear in the console, but rather be saved in `stdout.txt`.

- Redirect a file to standard input (`std::cin`) using the `<` symbol.

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

  In this example, we run `MetroSim` with `stations.txt` as the stations file and `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, which tests whether the program can handle user input, without us needing to type in all of that input line by line.

- Redirect *both* standard output and standard input.

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

  With the above, 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`.

- Use `diff` to compare the contents of two files.

  You can compare your output files with that of the reference solution, using `diff`. For example, if you run the reference implementation so that it produces an output file called `stdout_demo.txt` and run your own implementation so that it produces a file `stdout_personal.txt`, you can compare them with the following command:

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

  It can be difficult sometimes to understand the output of `diff` (although in this case, the less you see the better). 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_test` rule, with some dependencies already added, which can be used by the `unit_test` program, if you decide to use that way of testing.
- 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_test` 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!

README

As in prior assignments, the *final* phase of this project should be accompanied by a complete README file. See prior assignments specifications for details.

Submitting

You will submit in two phases:

- In the first phase, you will submit your `Passenger` and `PassengerQueue` implementations. They will be tested via our autograding system. You should be able to pass all those tests. If your code does not pass these tests, *get help as soon as possible*—if these classes aren't correctly implemented, then the rest of the simulation will almost certainly not produce correct results.

  For this phase, you merely need to submit the `.h` and `.cpp` files needed to compile those two classes. If there is something you want to document for the course staff to read, you can include a `README` file, but this is optional at this stage.

- In the second phase, you will submit all of your code, including testing files. Code should be fully documented as for prior assignments, with a proper `README` file, in-file comments, and the like. You must also include a `Makefile` that will allow us to compile and test your code. You do not need to include the reference implementation.

> Be sure that the final version of your program builds correctly using the commands `make` and `make MetroSim`, because we will use those commands to build your program for testing. A useful test is to make a submission directory, copy all your files in there, then run the appropriate `make` commands to ensure the program builds. Test this and then submit everything at that point.