Contents

Introduction

The Winnipeg Airports Authority (WAA) is contemplating doing some extensive pavement repairs on one of Winnipeg Richardson International Airport's (YWG) two runways. The corporation would like to evaluate what effect a runway closure could have on normal aircraft traffic, with the goal of minimizing the time wasted by planes waiting for a clear runway for departure or landing under different scenarios. The WAA would also like to consider how much additional traffic it could handle if it decided to build additional runways in the future.

The WAA asks you, as a bright computer scientist, to build an event-driven simulation program that will help estimate the effect of having *x* number of available runways on the total time wasted by planes waiting for an available runway, depending on a provided scenario of planes requesting to land/take off.

> 🚀 **Learning outcomes**
> Assignment 2 is assessing you on the following outcomes from COMP 2150:
>
> 1. Build a simple hierarchy of classes.
> 2. Use abstract classes and methods.
> 3. Use polymorphism appropriately for a problem that requires many related classes.
> 4. Write standard C++ code for file input/output operations.
> 5. Define class variables in C++.
> 6. Define a destructor in a C++ class.
> 7. Use separate compilation in C++.
>
> And one learning outcome from COMP 2140:
>
> 8. Implement common abstract data types.

Description

Background information

Your airport event-driven simulation needs to handle aircrafts that request to land and take off at different time points. Our unit of time in this assignment will be minutes, and you can consider minutes as integers (there will be no fractional minutes).

Your simulation will need to keep track of which runway is available and not available for use at different times. It needs to consider that landing or taking off is not instantaneous:

- Planes coming in to land need time to complete the final approach (descent towards the runway): we will consider that this always takes 2 minutes.
- Planes are spending some time on the runway when setting up for takeoff or during landing to leave the runway: we will consider that this always takes 1 minute for both takeoff and landing.

Planes also leave behind a disturbance in the air, called wake turbulence. It is dangerous for planes to fly through wake turbulence, so we need to wait for it to clear before we can have another plane take off or land on a runway. Moreover, planes are classified into different weight categories, and each category leaves behind a different amount of wake turbulence. In this simulation, we will consider the following weight categories and their corresponding duration of wake turbulence (for both takeoff and landing):

- small: 0 minute
- large: 1 minute
- heavy: 2 minutes
- super: 3 minutes

Finally, when multiple planes are requesting permission to land or take off, we will always prioritize planes in the air before planes on the ground. In other words, we must land all the planes in the air before we let planes on the ground take off. This makes sense because planes in the air are using a lot more fuel than planes on the ground while they are on a holding pattern.

Input

Your program must use two command-line arguments: the first argument will be a filename (the file will contain a list of planes requesting to take off or land, as described below) and the second argument will be an integer representing the number of runways to consider in the simulation (your code must be able to handle ***any*** integer >= 1).

For example, assuming your executable file is called `a2.exe`, you could run it on the `test.txt` input file with one runway like this:

```
a2.exe test.txt 1
```

The input text file contains plane descriptions on each line, in ascending order of time. Each line has the following format:

```
[time] [callSign] [flightNumber] [size] [requestType]
```

You can safely assume that the input files are always well-formed, and that:

- `[time]`: is an integer value representing the time (minute) of a plane requesting a runway.

- `[callSign]`: is a string (guaranteed to be one token with no spaces) representing the airline.
- `[flightNumber]`: is an integer value representing the flight number.
- `[size]`: is a string representing the size of the plane. It can only be either `small`, `large`, `heavy` or `super`.
- `[requestType]`: is a string representing what the plane is requesting. It can only be either `takeoff` or `landing`.

Since file reading in C++ is out of the scope of this course, both argument parsing and file reading is already provided to you in the `FileReadingExample.cpp` file that you can find on UMLearn. Feel free to use and adapt this provided code. Of course, in your assignment, you should not have the file-reading code directly in the main function, so you can place this provided code in the right class(es). Your main function should only process the command-line arguments, create an instance of a simulation class and start the simulation.

Output

A test file named `test.txt` is already provided to you in the Assignment 2 folder. Here is what the output of your event-driven simulation should look like when running this test file with 1 runway (your output must be formatted identically):

```
Simulation begins...
TIME: 1 -> Delta 1243 (1) heavy ready for takeoff.
TIME: 1 -> Delta 1243 (1) heavy cleared for takeoff on runway 1. (time req. for takeoff: 3)
TIME: 1 -> AirCanada 973 (2) heavy ready for takeoff.
TIME: 4 -> Delta 1243 (1) heavy has cleared the runway 1.
TIME: 4 -> AirCanada 973 (2) heavy cleared for takeoff on runway 1. (time req. for takeoff: 3)
TIME: 5 -> Emirates 1598 (3) super inbound for landing.
TIME: 7 -> AirCanada 973 (2) heavy has cleared the runway 1.
TIME: 7 -> Emirates 1598 (3) super cleared for landing on runway 1. (time req. for landing: 6)
TIME: 8 -> Sunwing 556 (4) large inbound for landing.
TIME: 10 -> Volaris 401 (5) small ready for takeoff.
TIME: 13 -> Emirates 1598 (3) super has cleared the runway 1.
TIME: 13 -> Sunwing 556 (4) large cleared for landing on runway 1. (time req. for landing: 4)
TIME: 15 -> AirTransat 225 (6) large inbound for landing.
TIME: 17 -> Sunwing 556 (4) large has cleared the runway 1.
TIME: 17 -> AirTransat 225 (6) large cleared for landing on runway 1. (time req. for landing: 4)
TIME: 18 -> Delta 5225 (7) heavy ready for takeoff.
TIME: 18 -> UPS 785 (8) super ready for takeoff.
TIME: 18 -> Southwest 3355 (9) heavy inbound for landing.
TIME: 21 -> AirTransat 225 (6) large has cleared the runway 1.
TIME: 21 -> Southwest 3355 (9) heavy cleared for landing on runway 1. (time req. for landing: 5)
TIME: 26 -> Southwest 3355 (9) heavy has cleared the runway 1.
```

```
TIME: 26 -> Volaris 401 (5) small cleared for takeoff on runway 1. (time req. for takeoff: 1)
TIME: 27 -> Volaris 401 (5) small has cleared the runway 1.
TIME: 27 -> Delta 5225 (7) heavy cleared for takeoff on runway 1. (time req. for takeoff: 3)
TIME: 30 -> Delta 5225 (7) heavy has cleared the runway 1.
TIME: 30 -> UPS 785 (8) super cleared for takeoff on runway 1. (time req. for takeoff: 4)
TIME: 34 -> UPS 785 (8) super has cleared the runway 1.

#######################
The simulation has ended.
The number of runways was 1.
The total amount of time wasted because runways were not available was 52 minutes.
#######################
```

Note that the number in between parentheses showing up after the flight number (e.g. *(1)* on the first output line) is representing the ATC ID (see the Event-driven simulation section for more details). Also, you should notice how events that are happening at the same time point are ordered: the event that is associated with the plane with the lowest ATC ID is always processed first. This guarantees that a runway will always be cleared (*i.e.* empty) before clearance (*i.e.* approval) is given to another plane for landing/takeoff.

Now, here is the output that you should get with 2 runways:

```
Simulation begins...
TIME: 1 -> Delta 1243 (1) heavy ready for takeoff.
TIME: 1 -> Delta 1243 (1) heavy cleared for takeoff on runway 1. (time req. for takeoff: 3)
TIME: 1 -> AirCanada 973 (2) heavy ready for takeoff.
TIME: 1 -> AirCanada 973 (2) heavy cleared for takeoff on runway 2. (time req. for takeoff: 3)
TIME: 4 -> Delta 1243 (1) heavy has cleared the runway 1.
TIME: 4 -> AirCanada 973 (2) heavy has cleared the runway 2.
TIME: 5 -> Emirates 1598 (3) super inbound for landing.
TIME: 5 -> Emirates 1598 (3) super cleared for landing on runway 1. (time req. for landing: 6)
TIME: 8 -> Sunwing 556 (4) large inbound for landing.
TIME: 8 -> Sunwing 556 (4) large cleared for landing on runway 2. (time req. for landing: 4)
TIME: 10 -> Volaris 401 (5) small ready for takeoff.
TIME: 11 -> Emirates 1598 (3) super has cleared the runway 1.
TIME: 11 -> Volaris 401 (5) small cleared for takeoff on runway 1. (time req. for takeoff: 1)
TIME: 12 -> Sunwing 556 (4) large has cleared the runway 2.
TIME: 12 -> Volaris 401 (5) small has cleared the runway 1.
TIME: 15 -> AirTransat 225 (6) large inbound for landing.
TIME: 15 -> AirTransat 225 (6) large cleared for landing on runway 1. (time req. for landing: 4)
TIME: 18 -> Delta 5225 (7) heavy ready for takeoff.
TIME: 18 -> Delta 5225 (7) heavy cleared for takeoff on runway 2. (time req. for takeoff: 3)
TIME: 18 -> UPS 785 (8) super ready for takeoff.
TIME: 18 -> Southwest 3355 (9) heavy inbound for landing.
TIME: 19 -> AirTransat 225 (6) large has cleared the runway 1.
```

```
TIME: 19 -> Southwest 3355 (9) heavy cleared for landing on runway 1. (time req. for landing: 5)
TIME: 21 -> Delta 5225 (7) heavy has cleared the runway 2.
TIME: 21 -> UPS 785 (8) super cleared for takeoff on runway 2. (time req. for takeoff: 4)
TIME: 24 -> Southwest 3355 (9) heavy has cleared the runway 1.
TIME: 25 -> UPS 785 (8) super has cleared the runway 2.

#######################
The simulation has ended.
The number of runways was 2.
The total amount of time wasted because runways were not available was 5 minutes.
#######################
```

As you can see above, your program should produce output that indicates the sequence of events processed and when they occurred (ordered by time). At the end of the simulation, your program will display a simple summary of the simulation containing:

- the number of runways
- the total amount of time wasted by planes waiting for a runway

Details and expectations

Event-driven simulation

You must build an event-driven simulation class. Its role is simple: constantly maintain a priority queue (*i.e.* an ordered list) of different types of events that need to be processed. When events are being processed, they must produce some output that will get printed to the console (see the output section), keep track of certain values and create other events (that will get added to the priority queue).

The priority queue of events must keep events sorted by time (so that events get processed chronologically). During the simulation process, ***at any point where the time unit is the same for two events, the plane that appeared earlier*** (*i.e.* that made a request first) ***should be handled first***. To do this, your program will assign an air traffic control (ATC) ID (starting at 1) to each plane when they are first created. This means that as the simulation goes on, you will be maintaining a list of pending events that is ordered primarily by time, but within time, ordered by ATC ID.

Your system will keep track of 5 subtypes of events, which are described more in depth below: **request landing events**, **request takeoff events**, **landing events**, **takeoff events** and **complete events**.

Request landing and request takeoff events

Request landing and **request takeoff** events correspond to the different lines of the input file. The input file is guaranteed to be ordered by time, so you must have only one event read from the file in the event queue at any time (you must not read the entire file all at once). This is because it would unnecessarily clog the priority queue of events. The processing of one **request landing** or **request takeoff** event must cause the next one to be read from the input file and placed in the priority queue. As mentioned above, each plane will get a unique ID number (starting at 1), and each new plane will have an ID of one higher than the one before.

When processing these two types of events, if a runway is available, then you can schedule the landing or takeoff immediately (remember that landing has priority over takeoff). In that case, your code must schedule the corresponding event (either a **landing event** or a **takeoff event**) for the current time and place it in the priority queue of events. Note that a runway can only be cleared for one plane at a time, and remains unavailable until the plane has completed the landing/takeoff (see the next section).

The output of these events should be formatted in the following way (e.g. taken from the output section):

- TIME: 5 -> Emirates 1598 (3) super inbound for landing.
- TIME: 1 -> Delta 1243 (1) heavy ready for takeoff.

Landing and takeoff events

Processing a **landing event** or **takeoff event** means that a runway has been assigned to a plane for landing/takeoff. The length of time it takes to complete these events depends on several factors: if it is a landing or takeoff (final approach time must be counted for a landing only), the time spent on the runway and the size of the plane (which affects wake turbulence). You should probably recheck the background information for more details on the amount of time required for all these things (which is simply a sum of all the relevant factors).

This means that it is possible to calculate how long a plane will take to complete the landing/takeoff (and this amount of time should be outputted, as shown below. When processing a landing/takeoff event, your program must schedule a **complete event** in the priority queue for the appropriate time (*i.e.*, the time that the landing/takeoff began + the time it takes to complete it).

The output of these events should be formatted in the following way (e.g. taken from the output section):

- TIME: 7 -> Emirates 1598 (3) super cleared for landing on runway 1. (time req. for landing: 6)
- TIME: 1 -> Delta 1243 (1) heavy cleared for takeoff on runway 1. (time req. for takeoff: 3)

Complete events

A **complete event** occurs when enough time has passed to complete the landing/takeoff and let the wake turbulence dissipate. After processing the **complete event**, the specific runway that was used is now available for another plane to use. Your program must check if another plane is waiting for a landing or takeoff, and schedule a **landing event** or a **takeoff event** accordingly.

The output of this event should be formatted in the following way (e.g. taken from the output section):

- TIME: 4 -> Delta 1243 (1) heavy has cleared the runway 1.

Data structures

As in the first assignment, any linked structures must be your own, built from scratch. This means that the event priority queue and the waiting list(s) for planes that need to land/takeoff must be designed entirely by you, in a generic way (*i.e.* they should be reusable to store different types of things). Arrays can only be used for temporary operations or to keep track of available runways.

Other supporting classes

Your program will need other supporting classes that are not explicitly described above. Compartmentalization is always recommended in OOP: separate distinct concepts into separate objects (compartments) that are smaller and easier to handle on their own. You might create up to 12-15 (or more) different short classes to complete this assignment, and that is ok. Do not forget to use class hierarchies, abstract classes/methods, and polymorphism when appropriate, and always minimize code duplication.

Submission requirements

You should submit your assignment to UMLearn as a `.zip` file. Your assignment submission should include:

- All your source code (`.h` and `.cpp`) files you created for this event-driven simulation.
- Either a text file named `README.txt` or a Markdown-formatted file named `README.md` describing how to compile and run your program.

All submitted programs must be able to build and run on Aviary.

Assessment

Your submission will be assessed in a few major ways: the stated learning outcomes, the output that your program produces, and the quality of your code (adhering to the course's programming standards).

Programming standards

Your submission will be assessed on code quality out of a total of 5 points. Code quality in this course is met by adhering to the course's programming standards.

Level	Description
0	The submitted code does not meet any programming standards and is generally very poor quality.
1	The submitted code meets very few of the programming standards and is generally poor quality.
2	The submitted code meets some or most of the programming standards and is generally average quality.
3	The submitted code meets all of the programming standards and is generally high quality.

Output correctness

Your submission will be assessed on output out of a total of 5 points. You will be provided with a sample input that matches the expected output shown above in the output section, but your program will be evaluated on additional inputs that are not provided to you.

Learning outcomes

> 🚀 **Learning outcomes**
> Assignment 2 is assessing you on the following outcomes from COMP 2150:
>
> 1. Build a simple hierarchy of classes.
> 2. Use abstract classes and methods.
> 3. Use polymorphism appropriately for a problem that requires many related classes.
> 4. Write standard C++ code for file input/output operations.
> 5. Define class variables in C++.

> 6. Define a destructor in a C++ class.
> 7. Use separate compilation in C++.
>
> And one learning outcome from COMP 2140:
>
> 8. Implement common abstract data types.

Your submission will be assessed on the stated learning outcomes as follows:

Outcome	Levels
(1) 4 points	• **Level 0**: Submission does not build any hierarchy of classes at all. • **Level 1**: One or several hierarchy of classes are present, but they are inappropriate for the stated problem. • **Level 2**: One or several hierarchy of classes are present and they are approaching appropriate for the stated problem, but need work to improve. • **Level 3**: One or several appropriate hierarchies of classes are present for the stated problem.
(2) 2 points	• **Level 0**: Submission does not use any abstract class / method to solve the stated problem. • **Level 1**: Submission uses some abstract classes and methods, but not always when appropriate. • **Level 2**: Submission always uses abstract classes and methods when appropriate.
(3) 4 points	• **Level 0**: Submission does not employ polymorphism at all to solve the stated problem. • **Level 1**: Submission sometimes uses polymorphism to solve the stated problem, but inconsistently (some use of dynamic class binding and some use of dynamic dispatch). • **Level 2**: Submission consistently uses polymorphism to solve the stated problem when appropriate.
(4) 2 points	• **Level 0**: Submission is unable to process the input file and/or does not output anything to the console. • **Level 1**: Submission processes the entire input file all at once and/or makes unclear outputs to the console. • **Level 2**: Submission processes the input file one arrival at a time and makes clear outputs to the console.

Outcome	Levels
(5) 2 points	• **Level 0**: No class variables are used to store counts or constant values. • **Level 1**: Some class variables are used, but some hardcoded values are present. • **Level 2**: Class variables are always used when appropriate.
(6) 2 points	• **Level 0**: No destructors are defined in the submission. • **Level 1**: Some destructors are defined, but not all the memory is freed at the end of the program execution. • **Level 2**: Destructors are defined and correctly free all the memory.
(7) 2 points	• **Level 0**: Program does not compile at all with instructions stated in `README` file, or does not compile at all with typical C++ compile commands on Aviary (*i.e.*, `clang++ *.cpp -std=c++11`). • **Level 1**: Program compiles but separate compilation is not used consistently. • **Level 2**: Separate compilation is used and the program compiles with instructions stated in `README` file on Aviary.
(8) 4 points	• **Level 0**: Submission does not independently implement a linked data structure to represent a generic priority queue (e.g., uses STL data structures). • **Level 1**: Submission uses arrays or some other non-linked data structure to represent a generic priority queue. • **Level 2**: Submission independently implements linked data structures that are not generic (multiple data structures are used for each type of data held). • **Level 3**: Submission independently implements a linked data structure to represent a generic priority queue.