COMP 2150 - Winter 2020 - Assignment 2

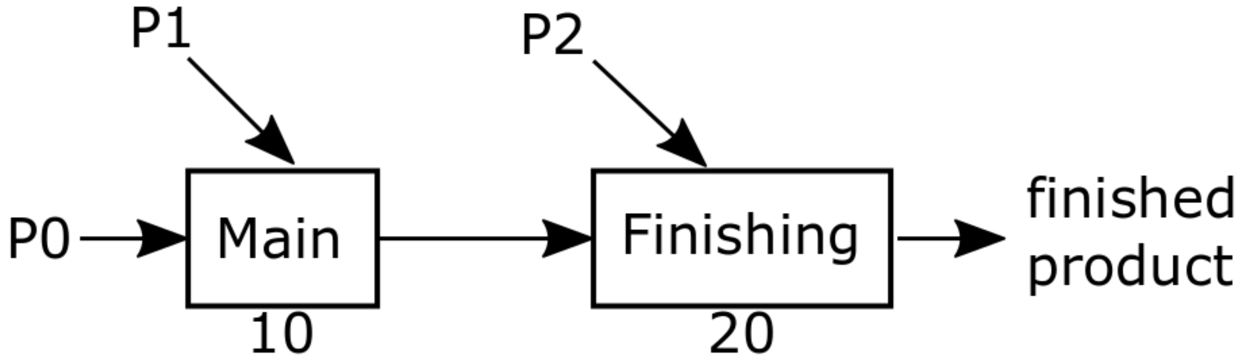
Due February 26, 2020 at 11:59 PM

In a factory, an assembly line is a tool that enables the assembly of products (cars, for instance). In the assembly line, a partially assembled product moves along the line to different ***stations*** and ***parts*** are added at each step. In this assignment, you will create a discrete event simulation of an assembly line. The simulation will involve assembling three ***parts*** to construct a finished product. The parts that are used to assemble the finished product arrive at different times to the two ***stations*** in the assembly line. The simulation will be supplied with a list of events corresponding to the arrival of the parts to the stations. The simulation will rely on a global clock that keeps track of time for the simulation.

This assembly line consists of two stations (the **main station** and the **finishing station**), each of which requires a certain amount of time to assemble two parts, as shown in the diagram below. These times for assembly will vary and will be specified in the input file. The transportation time between stations is assumed to be zero in our simulation.

A figure of the assembly line is given below. Note that:

1. There are two assembly stations in this assembly line (main and finishing). The assembly line assembles three parts (P0, P1 and P2) into the final product.
2. In the diagram, the assembly times for the two stations are given below the station (in this example, ten units of time for the main station and 20 units of time for the finishing station).
3. In the assembly line, the parts move between the main station and the finishing station on a conveyor belt that can accomodate all parts as they are produced by the main station. As a partially assembled product is done being assembled at the main station, it is immediately placed on the conveyor belt and moves to the finishing station.
4. On the other hand, stations can only assemble one item at a time. If a station is currently assembling a product, all other parts/partially assembled products wait (in the order they arrived) for assembly.



The simulation is an ***event-driven simulation***. In your solution, a priority queue of different types of ***events,*** ordered by time, is used to advance the simulation. The types of events are described below. Details on how to construct an event-driven simulation and a basic example are provided to you on the website.

Your solution to this assignment will be in C++. You should use, as appropriate, the OO tools we have developed in the course (see the section on OO Programming below). There are also specific C++ requirements, including using a makefile, using command line arguments and separate compilation (see the section on C++ specifics).

Input File

The simulation is driven by the information in the input file. The first line of the input file has two integers. This specifies the assembly times for the main station (first) and the finishing station (second).

The input file then gives information on the arrivals of parts at different stations. (The times of the partially assembled products arriving at the finishing station is defined implicitly by the arrival times of the parts and the assembly times). Each line of the input file defines one arrival, specified by two integers: the arrival time and the part number. For example, the line

```
10 2
```

specifies that part P2 arrives (at the finishing station, in this case) at time 10. The input files are guaranteed to be ordered by time. The input files are also **guaranteed to contain valid integers** -- your solution does not need to validate the integers in the input file.

Events

There are four general types of events in the system:

Part Arrival: this event specifies that a part arrives at a station. The part numbers always correspond to the station: P0 and P1 go to the main station and P2 goes to the finishing station. If the proper parts/partially assembled product are waiting at a station after a Part Arrival and the station is not assembling a product, then a Start Assembly event should be scheduled. Otherwise, the part that arrives should be added to a waiting queue for that part at the appropriate station (main or finishing).

Start Assembly: these events should cause an End Assembly event to be scheduled with a time equal to the current time plus the assembly time of the station.

End Assembly: for the main station, this event should cause a Product Arrival event to be scheduled immediately for the Finishing station (transportation time is assumed to be zero). If both parts P0 and P1 are waiting at the Main station, and the Main station is available, another Start Assembly event should be scheduled for the current time (at the Main station).

For the Finishing station, the End Assembly event should calculate statistics for the assembly of this product (see the Output section below for more details on the statistics).

Product Arrival: this event corresponds to a partially assembled product arriving at the Finishing station. If the part P2 is waiting at the station at this time and there is no product being assembled at the Finishing station, then a Start Assembly event should be scheduled immediately. Otherwise, the partially assembled product is added to a waiting queue for the Finishing station.

These events are provided for your reference. You may choose to organize your events into more types, but you will need to use one (or more) polymorphic methods and a hierarchy for your events. You are provided with code for the Event class which will be the root of the hierarchy (see below for more information on the provided code) and a pure virtual method called processEvent.

You will need to maintain a priority queue of future events ordered by time. During the assembly line simulation, at any point if there are two events with the same time, then ties are broken arbitrarily.

Simulation and Provided Code

The basic simulation will follow the logic presented in the video on umlearn. However, your simulation will be more detailed. In particular, you will likely observe that:

1. your simulation needs to maintain three queues, one for each of the parts P0,P1 and P2. For this task and **only this task** in the assignment, it is ok to use an array to manage the queues (but the queues are not arrays).
2. your simulation will also need to maintain a queue of partially assembled products waiting at the finishing station.
3. your simulation needs to maintain a status for each station: is something currently happening at the Main/Finishing station?
4. your simulation also needs to know the lengths of time of assembly for the Main and Finishing stations, which are read from the input file.

You are provided with some basic code:

- a main file (A2main.cpp) that starts the simulation.
- the header file for the simulation class that defines the methods that the simulation (Simulation.h). Note that the forward references in the file define other classes you will need.
- the header file for the general event class (Event.h) and partial implementation (Event.cpp). You will use this to define the subclasses of Event.

- an implementation of a Node, ListItem, OrderedItem and Queue classes.

You can modify the files as you see fit (with one exception, below), but most will require minimal modifications: for instance, you will be able to complete the assignment without modifying the Queue, OrderedItem, ListItem and Node files. You will likely need to modify the Simulation.h by adding new fields and methods to it. You should aim to reuse code from the Node and ListItem classes for the Priority Queue that you need to implement.

The one thing you cannot modify about the provided code is you **should NOT change how A2main.cpp reads input from the file**. This must be kept the same so that the markers can be sure all programs operate in the same way.

Object Oriented Programming

Your assignment should use OO programming. In particular:

1. You should use good OO practice in general, including proper information hiding. Fields should not be public.
2. You should have a hierarchy of data items to go into any data structures, and a polymorphic hierarchy of Events. (Note that this means that you will have a hierarchy of Events, roughly corresponding to the types described above. Your processing of events through the processEvent method should be polymorphic.)
3. You should have code reuse as much as possible. If you have duplicated code for the same task , you will likely lose marks.
4. You should have as little non-OO code as possible. You should not add any non-OO code to the A2main.cpp file, or elsewhere in your project.

Hint: One challenge of this assignment (and separate compilation) will be allowing the Events to know information from the larger simulation and possibly modify the state of the simulation. For instance, some Events cause other Events to be read from the file. One way to achieve this is by allowing all Events to have a pointer to other classes, including (possibly) the class that created it.

C++ Requirements

1. You must read the names of the file from a command line argument, as is done in the provided main.
2. You must use separate compilation for all classes.
3. You must use destructors to free all allocated memory. (You do not need to complete the other OCCF methods for this assignment.)
4. You must write a makefile to build your project. The command "make" with no arguments should build the project. (i.e., the default rule should produce the executable.)

Data Structures

The data structures you write must be your own, and you cannot use the C++ standard template library (STL): you must make generic data structures (an ordered list/priority queue) using your own linked structures and making use of C++ OO features. **If you don't write your own data structures, you will lose marks. You should not use arrays for any tasks except for the one specified above.**

For this assignment, you do not need to concern yourself with data structure efficiency. Your Priority Queue should maintain items in sorted order, but you do not need to implement any more efficient operations (e.g., a heap).

When dealing with hierarchies and data structures, you must use safe casting practices.

Unit Testing

Construct a set of unit tests for your code. To do unit testing in C++, you should:

1. Use Catch2 for a simple test framework. Download Catch2 as a single header file from the course website. You must use a C++11 compiler (`clang++ --std=c++11` on aviary) for catch2.
2. Place the catch.hpp in the same directory as your code.
3. Create a new file for the tests.
4. Preprocessor commands: add these commands to the test file:

    ```
    #define CATCH_CONFIG_MAIN

    #include "catch.hpp"
    ```

5. Use the TEST_CASE and REQUIRE to write test cases. (You can see a tutorial for catch2 here: https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md). Each test must have an assertion (REQUIRE) in it.

Your tests should focus on the data structures **you create** for your project. You should include at least five meaningful for your data structures. You should **not** test the Queue class.

You will be graded on your tests, so write useful tests. The markers will be running your tests, as well, so ensure that they pass. If tests do not pass or your code does not compile, you will lose a substantial number of marks. (You can add a rule to your makefile to build your tests if you would like.)

Output

Each event should produce output as it is processed. That is, when processing an event, you should write **one** line of output indicating t**he time and the event that occured** at that time.

After reading the entire input file and completing the simulation, you should report basic statistics:

- how many items have been completely assembled?
- what is the average time to completely build a product? The build time is the time from the time of the first arrival of any part used to build the product until the end of the assembly at the finishing station. The average should be taken over all products that are completely assembled.
- how many parts are left in each of the queues? for the three part queues and the partially assembled product queue, report how many parts are left over in each at the end of the simulation.

There is no predefined format for the output. You should present the information in a readable format. All output should be to the console (i.e., using cout).

Hand-in

Submit all your source code for all classes, including all provided code and the catch.hpp file. Use separate compilation and have each class in its own .h and .cpp file. Your main should also remain in a separate .cpp file.

You **MUST** submit all of your files in a zip file. Additionally, you **MUST** follow these rules:

- Include a README.TXT file (in the zip file) that describes **exactly how to compile and run your code** from the command line. (It is understood that the makefile and command line arguments will be the same for everyone, but in case of problems, the markers will be reading this file). If your code does not compile directly, you will lose marks. Code will be run on a standard linux environment (aviary) using the makefile you provide.
- You should also submit a text document (in the zip file) giving the output on the official test data. (Official test data will not be released until just before the due date.)

The easier it is for your assignment to mark, the more marks you are likely to get. Do yourself a favour. Submit your zip file on umlearn.