EECS402, Fall 2025, Project 5

Overview:

For this project, we will be providing you a significant amount of code to start with. This is very different from previous projects where we might give you some function prototypes, but you would essentially be starting from scratch. The provided classes and code make it so you don't have to spend a lot of time thinking about high-level design for the project, but you will need to spend some time understanding the provided code and how the code you will be adding will work with it. In general, this project should be less time-consuming than the same project if it were a "start from scratch" project, and will allow you to focus on the simulation aspects of the project instead of reading files, error checking, determining which attributes your EventClass should have, etc. Since the end of semester is quickly approaching, this seems like a good way to get some experience with an event-driven simulation, and also get experience in working with someone else's code, as is typically necessary when working in a team-based environment.

This project will involve implementation of an event-driven simulation of traffic flow through a 4-way intersection that is managed by a traffic light. Your simulation will include cars arriving at the intersection traveling in all four directions, as well as the light changing state throughout the simulation. After the simulation is complete, you will output some high-level statistics about the traffic flow.

Due Date and Submitting:

This project is due on **Thursday, December 4, 2025 at 4:30pm**. Early submissions are allowed, with corresponding bonus points, according to the policy described in detail in the course syllabus.

For this project, you must submit several files. You must submit each header file (with extension .h), each source file (with extension .cpp), and each template implementation file (with extension .inl) that you create in implementing the project. In addition, you must submit a valid UNIX Makefile, that will allow your project to be built using the command "make" resulting in an executable file named "proj5.exe". Also, your Makefile must have a target named "clean" that removes all of your .o files and your executable (but not your source code!). Finally, don't forget to attach a typescript showing the build of your project and a run of the project in valgrind.

When submitting your project, be sure that **every** source file (.h, .cpp, and .inl files!!!), your valid Makefile, and your typescript file are attached to the submission email. The submission system will respond with the number of files accepted as part of your submission, and a list of all the files accepted – it is ***your responsibility*** to ensure all source files were attached to the email and were accepted by the system. If you forget to submit a file on accident, we will not allow you to add the file after the deadline, so please take the time to carefully check the submission response email to be completely sure every single file you intended to submit was accepted by the system.

Detailed Description:

This project will be split up into two phases. It is very strongly recommended that you perform phase 1 fully, including exhaustive testing, before moving on to work on phase 2.

Phase 1:

In phase 1, you will update some of the data structures you developed in project 4 to be "templated" – that is, to be associated with a generic data type using C++ templates such that the data structures are able to hold more data types than just integers.

The SortedListClass and the FIFOQueueClass must be modified to be templated classes. There is no need to use the LIFOStackClass in this project, so it is not required that you make it a templated class or include it in your submission.

The design, names and functional interface to your templated data structure classes must match those developed in project 4 exactly. In other words, you may not add new member functions, add new member variables, change the number or meaning of function parameters, etc. The only difference between the data structures developed in project 4 and the modified versions developed in project 5 will be that the project 4 versions are specifically tied to integers, while the project 5 versions are templated.

Phase 1 of this project should NOT result in writing much "new code", except for developing test cases to make sure your updated data structures work with different data types.

Phase 2:

Once your data structures are implemented and tested, you will develop an event-driven simulation of cars traveling through an intersection that is managed by a functioning traffic light. Events are a central part of an event-drive simulation, so you will need an EventClass. As discussed in lecture, event objects will be stored in a SortedListClass that is referred to as "the event list". The event list will only contain events that are scheduled to occur in the future – it will NOT store events that have already been handled.

Concept of "Time Tics"

A "time tic" is a unit-less measure of time. When we say something occurs in "3 time tics" is just means 3 advancements of time. It might be 3 seconds, or 3 minutes, or 3 milliseconds, etc. The point is that a "time tic" is just meant to represent a generic unit of time so that we don't have to think in terms of specific units of time. This is often convenient to allow simulations to "scale" as needed – I might set my simulation parameters in a way that I think of a "time tic" as being milliseconds, but another user might set their simulation parameters in a way that they think of a "time tic" as seconds.

Simulation Parameters and Input File Format

There are 15 control parameters that will affect how the simulation performs. To make it easy to run many different simulations, the simulation control parameters will be read from a text file, whose name will be provided to the program via a command line argument. The parameters are as follows:

- Random number generator seed: A non-negative integer value for seeding the random number generator to select a pseudo-random sequence for the simulation to use.
- Simulation end time: A positive integer value indicating when the simulation should end. When the next event occurs after this specified number of time tics, the event, and any scheduled after it will not be handled, and the simulation will be deemed completed.
- East-west green time and east-west yellow time: Two positive integers indicating the amount of time the light is green and yellow in the east-west direction. Red duration is NOT specified, as it is assumed to be the sum of the green and yellow times for the north-south direction.
- North-south green time and north-south yellow time: Two positive integers indicating the amount of time the light is green and yellow in the north-south direction. Red duration is NOT specified, as it is assumed to be the sum of the green and yellow times for the east-west direction.
- Mean and standard deviation for the arrival rate of east-bound cars: Two positive floating point values indicating the mean and standard deviation of the pre-determined normal distribution characterizing the arrival rate for cars arriving at the intersection in an east-bound direction.
- Mean and standard deviation for the arrival rate of west-bound cars: Two positive floating point values indicating the mean and standard deviation of the pre-determined normal distribution characterizing the arrival rate for cars arriving at the intersection in an west-bound direction.

- Mean and standard deviation for the arrival rate of north-bound cars: Two positive floating point values indicating the mean and standard deviation of the pre-determined normal distribution characterizing the arrival rate for cars arriving at the intersection in an north-bound direction.
- Mean and standard deviation for the arrival rate of south-bound cars: Two positive floating point values indicating the mean and standard deviation of the pre-determined normal distribution characterizing the arrival rate for cars arriving at the intersection in an south-bound direction.
- Percentage of drivers that will advance through a yellow light: An integer value from 0 to 100 (inclusive) describing how often a driver will choose to advance through the intersection when the light is yellow. A number of 40 indicates that 40% of the time a driver will choose to advance through a yellow light, while 60% of the time a driver will choose to stop at the yellow light.

The input file shall be a simple ASCII file with the 15 values specified in a whitespace-delimited way, in the exact order specified above. An example input file follows:

```
12345
1000
10 2
17 3
3 1.04
15 2.7
7.25 0
12.5 3.3
65
```

This input file indicates the random number will be seeded with the value 12345, the simulation will end when the simulation clock reaches 1000 time tics, the east-west light will remain green for 10 time tics and yellow for 2 time tics, the north-south light will remain green for 17 time tics and yellow for 3 time tics, cars will arrive east-bound via a normal distribution with a mean of 3 time tics and a standard deviation of 1.04, cars will arrive west-bound via a normal distribution with a mean of 15 time tics and a standard deviation of 2.7, cars will arrive north-bound via a normal distribution with a mean of 7.25 time tics and a standard deviation of 0, cars will arrive south-bound via a normal distribution with a mean of 12.5 time tics and a standard deviation of 3.3, and on average, 65% of drivers will choose to advance through the intersection at a yellow light.

Event Handling Overview

The most important part of an event-driven simulation is the handling of events. As discussed in lecture, handling an event will often generate additional events to be scheduled to occur in the future. Additionally, handling an event will often change the state of the simulation (i.e. advance the simulation's "current time" to the time the event being handled at occurs, etc.), and will often change accumulating statistics (like maximum values, counts of objects, etc.).

To keep our simulation less complex, we will implement the cars advancing through an intersection only when the light is changing. Obviously, this is not realistic – in the "real world" cars advance through the intersection while the light is green (or yellow). We will handle this differently though.

Specifically, let's say the light is green in the north-south direction for 10 time tics. When we handle an event that represents a change in the light to being yellow in the north-south direction, we will implement the logic to advance up to 10 cars in the north-bound queue to travel through the intersection and up to 10 cars in the south-bound queue to travel through the intersection. The idea behind this is that when a light is changing to yellow, all the cars that would want to advance through the intersection on the green light will have arrived and be placed in the queues. So it is during the handling of the change to yellow that we advance the cars for the green light. That might seem odd at first, but it makes the processing much simpler, so take a minute to understand what is expected, and look at the sample output to make sure you understand the proper flow of the simulation.

Similarly, when we handle an event representing the light changing from yellow in the north-south direction to green in the east-west direction, that is when we will advance cars through the intersection in the north-south direction for the yellow light that is changing.

Light Cycle

The light will always start as a green light in the east-west direction for our simulation!

The traffic light cycles as follows:

Current Light State	Changed Light State	Cars Advancing During Change Event Handling
Green East-West (Red North-South)	Yellow East-West	East-West traffic through green light
Yellow East-West (Red North-South)	Green North-South	East-west traffic through yellow light
Green North-South (Red East-West)	Yellow North-South	North-south traffic through green light
Yellow North-South (Red East-West)	Green East-West	North-south traffic through yellow light

When advancing cars through a green light, cars advance one-per-time-tic. A green light that lasts for 20 time tics will allow up to 20 waiting cars in both directions to advance.

When advancing cars through a yellow light, you must consider that some drivers will advance through a yellow light, but others will stop at the light. For our simulation, we will determine a driver's decision using a uniform random number such that a specified percentage of drivers will be expected to advance through the light and the remainder will be expected to stop. Obviously, if a driver decides to stop at the yellow light, cars that are behind it will be forced to stop as well. Let's say the simulation parameter that controls this is set to 75. This would mean that 75% of drivers will advance through the yellow light, while 25% will choose to stop at the light. If you request a uniform random number between 1 and 100, you can determine the driver will advance if the random number is less than or equal to 75, while a result greater than 75 will indicate the driver will NOT advance through the light. This is just an example, and you would instead compare your uniform random number to the corresponding simulation control parameter instead of always using the literal value 75. If a driver decides to advance through a yellow light, the following driver will make their own decision, so you'll draw another uniform random number as needed for that driver, and so on.

Event Handling Specifics

There are 8 types of events that can occur in our simulation.

- Car arrival east-bound (EVENT_ARRIVE_EAST)
    - When handling an event of a car arriving at the intersection heading east, it gets added to the east-bound queue of cars waiting to get through the intersection. The next east-bound arrival event is

    also scheduled during the handling of this type of event, based on the east-bound arrival distribution parameters (mean and standard deviation).

- Car arrival west-bound (EVENT_ARRIVE_WEST)
    - Same as "Car arrival east-bound" but for a west-bound car instead.
- Car arrival north-bound (EVENT_ARRIVE_NORTH)
    - Same as "Car arrival east-bound" but for a north-bound car instead.
- Car arrival south-bound (EVENT_ARRIVE_SOUTH)
    - Same as "Car arrival east-bound" but for a south-bound car instead.
- Light change from "Green in East-West" to "Yellow in East-West" (EVENT_CHANGE_YELLOW_EW)
    - When the light changes FROM a green light in the east-west direction TO a yellow light in the east-west direction, you will advance cars through the intersection for the green light in the east-west direction. If the light is green in the east-west direction for 10 time tics, then when handling this event type, you will allow up to 10 cars waiting in the east-bound queue to advance through the intersection (or less than 10 if there are less than 10 cars waiting in the queue), and up to 10 cars waiting in the west-bound queue to advance through the intersection. Finally, you'll also schedule the next light change event to occur at the appropriate time in the future (the next light change event in this case would be a light change from "Yellow in East-West" to "Green in North-South".
- Light change from "Yellow in East-West" to "Green in North-South" (EVENT_CHANGE_GREEN_NS)
    - Advance cars through the intersection for the yellow light in the east-west direction, and schedule the next light change (from "Green in North-South" to "Yellow in North-South"
- Light change from "Green in North-South" to "Yellow in North-South" (EVENT_CHANGE_YELLOW_NS)
    - Advance cars through the intersection for the green light in the north-soth direction, and schedule the next light change (from "Yellow in North-South" to "Green in East-West"
- Light change from "Yellow in North-South" to "Green in East-West" (EVENT_CHANGE_GREEN_EW)
    - Advance cars through the intersection for the yellow light in the north-south direction, and schedule the next light change (from "Green in East-West" to "Yellow in East-West"

Design and Provided Code

EventClass

I will provide most of the EventClass for you to use in your project. Since objects of the EventClass will be stored in a SortedListClass (which was templated in phase 1), **you will need to update the EventClass to overload any operators that are used on the templated type within your SortedListClass**. EventClass objects are sorted based on their scheduled time to occur. So, for example, an event scheduled to occur at time 5 will come before an event scheduled to occur at time 7 in your sorted event list.

CarClass

I will provide a CarClass for you to use in your project. The concept of a "car object" in this project isn't meant to describe the car itself (i.e. the color, make, model, number of doors, etc.), but rather to maintain some data about a car that is participating in our simulation. Therefore, you'll see the CarClass attributes include a unique identifier value (auto-assigned in the primary ctor), the direction the car is going as it approaches the intersection, and the time at which it arrived at the intersection.

IntersectionSimulationClass

This is the "over-arching" class for the project. It will be the class that stores the control parameters of the simulation, advances the state of the simulation as it runs, and maintains statistics about the simulation as it runs. It will contain the all-important event list containing all events currently scheduled, as well as queues of vehicles awaiting their turn to advance through the intersection.

I will provide a portion of the IntersectionSimulationClass for you to use. The provided code will include the class attributes, the functionality to read simulation parameters from an input file, and print the final statistics. **You will need to add functionality to this class, though, to actually handle events appropriately and maintain the statistics values throughout.**

constants.h

I will provide a set of pre-defined constants for your use in this project. Of course, if you want to add additional constants to this file, you may do so. I did not "hide" any of my constants from you, so I don't anticipate you needing to add constants, but if your implementation would benefit from additional constants, that is fine.

random.h and random.cpp

These files contain some high-level functionality to obtain random numbers according to either a uniform distribution or a normal distribution. You may NOT modify these functions. You can use them as provided.

project5.cpp

This is the main function for the project. I will provide my solution's entire main function for you to use. You should not need to modify the provided project5.cpp, but you may make minor modifications if for some reason you find that you need to in your implementation. If you find you need to modify it, though, you may want to consult with the course staff to make sure the high-level design is proper.

Things NOT to Worry About

This project is intended to be an introduction to event-driven simulations, so there are some "obvious" things that probably should be part of a realistic simulation that we won't be worrying about in this project. I'm sure this isn't a full list, but it will capture some of the things that you might want to include a more realistic simulation than what we are developing.

- Driver reaction times: In our simulation, we are going to blatantly assume that, in a given direction, one car can pass through the intersection in exactly one time tic. In the real world, some drivers proceed through an intersection slowly, while others speed through. In our simulation, if a light is green for 10 time tics, then we will expect the lower number of either 10 cars, or the total number of cars waiting in that direction to be able to advance through the intersection. In other words, if the queue length is 12 cars, then 10 will advance through the intersection on a green light lasting 10 time tics. Clearly, though, if the queue length is only 5, then only 5 cars will advance through the intersection for the 10 tics the light is green.
- Drivers that don't follow the rules: In our simulation, all drivers stop on red. No driver will advance through the intersection when the light is red in their direction.
- Turns: Our intersection does not allow cars to turn. This greatly simplifies things by not being concerned with oncoming traffic during a left turn, which, in the real world, would prevent cars from advancing through the intersection.
- Turn lights: Our intersection doesn't have "left turn lights" or "right turn lights". That makes sense, since we don't allow turns at all.
- Malfunctions: Our traffic light never malfunctions or loses power, etc. It will cycle as expected throughout the entire simulation.
- Red in both directions: In the real world, traffic lights often allow the signal to be red in both directions for a short time. This is primarily to allow those drivers who are speeding through the very end of a yellow light to get through the intersection before the other direction's signal turns green. Since our drivers also follow the rules, this is not a concern for us. This means that the instant the light turns red in one direction, it turns green in the opposite direction.

- Other blockages: Our intersection is never blocked due to construction, emergency vehicles, funeral processions, etc.

Approach and Output

This simulation ***__MUST__*** be implemented as an event-driven simulation as described in lecture. Note that the word "must" is all caps, bolded, underlined, and italicized! If your simulation is implemented as a time-driven simulation (or any variant on a time-driven simulation) or you otherwise generate and handle events in a way that is not the way described in lecture for an event-driven simulation, you will not receive credit for your project.

Following are some important requirements about how the simulation itself must work:

- The SortedListClass will contain events and act as a priority queue that contains your "event list" as described in lecture
- When a car arrives at the intersection, you must utilize a FIFOQueueClass to have the car wait at the light until it is able to advance
- You must generate new objects only when you need them. That is, do NOT generate a full list of light changes or car arrivals at the beginning of the simulation. Instead, generate a single car arrival in each direction and the initial light change event to start off the simulation. Then, only when handling one event would you schedule the next related event.
    - This means, at any given time, your event list will only hold one scheduled east-bound arrival, one scheduled west-bound arrival, one scheduled north-bound arrival, one scheduled south-bound arrival, and one scheduled light change.
- It is important that your simulation produce output that is the same as the posted samples. Due to the autograder, some of the output lines are required to match *exactly.*
    - The entire statistics output section must match the format of the sample outputs exactly. The function `IntersectionSimulationClass::printStatistics` will be provided to you and must not be modified in any way.
    - Any lines that indicate cars advancing through the intersection or not advancing through the intersection must be included exactly as shown in the sample outputs.
    - Any lines that indicate an event is being handled must be included exactly as shown in the sample outputs.
    - Any lines that summarize the number of cars advancing through a light during a light change (and that also show the remaining queue size) must be included exactly as shown in the sample outputs.
    - Even outside these specific indicated output lines, you should try to have your simulation's output be formatted the same as the sample outputs with the same information being printed in the same way.
- Remember, back in project 4, the order in which you insert items that are "duplicate" was specified. It may have been difficult to tell if your project 4 inserted duplicates in the expected order, because you can't tell the different between one integer value and duplicate integer value (i.e. the first "17" you insert can't be differentiated from the second "17"). In this project, however, it becomes obvious. Consider some of the outputs in sampleOutputA.txt – look at time 10. You'll notice that there are two events that occur at time 10. The "Light Change to EW Yellow" event at time 10 is handled first because that event was scheduled first. The "East-Bound Arrival" event at time 10 is handled second, because it was scheduled at time 10 AFTER the "Light Change to EW Yellow" event was scheduled at time 10. Since this was clearly specified in the project 4 specs, it is required that your project 5 event handling handles events occurring at the same time *in the order in which they were scheduled*! It would be worth verifying your program works this way, as you may have easily overlooked this requirement in project 4. Please note, this was NOT tested during grading of P4, because it is impossible to determine whether your list worked this way or not for a list if simple "int"s when only looking at the output.
- To keep the project less complex, you will only be required to maintain some very simple "statistics". These include:
    - The longest queue length formed in the east-bound direction

  - The longest queue length formed in the west-bound direction
  - The longest queue length formed in the north-bound direction
  - The longest queue length formed in the south-bound direction
  - The total number of cars that advanced through the intersection in the east-bound direction
  - The total number of cars that advanced through the intersection in the west-bound direction
  - The total number of cars that advanced through the intersection in the north-bound direction
  - The total number of cars that advanced through the intersection in the south-bound direction
- Note that normally, you would want a much richer set of statistics, but the primary purpose of this project is to implement the simulation itself and make sure the events are handled properly and the simulation flows as expected

"Specific Specifications"

These "specific specifications" are meant to state whether or not something is allowed. A "no" means you definitely may NOT use that item. We have not necessarily covered all the topics listed, so if you don't know what each of these is, it's not likely you would "accidentally" use them in your solution. Those types of restrictions are put in place mainly for students who know some of the more advanced topics and might try to use them when they're not expected or allowed. In general, you can assume that you should not be using anything that has not yet been covered in lecture (as of the first posting of the project).

- Use of Goto: No
- Global Variables / Objects: No
- Global Functions: If needed
- Use of Friend Functions / Classes: Only as included in the provided code
- Use of Structs: No
- Use of Classes: Yes – required!
- Public Data In Classes: No (all data members must be private)
- Use of Inheritance / Polymorphism: No
- Use of Arrays: If needed
- Use of C++ "string" Type: Yes
- Use of C-Strings: No
- Use of Pointers: Yes – required!
- Use of STL Containers: **No** (No!!!)
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: Yes – required!
- Use of exit(): No
- Use of overloaded operators: Yes – required!
- Use of float type: No