CompSys202/MechEng270
Data Structures and Algorithms
Assignment #2 (15% of final grade)
Due: 5:00pm, 26 September 2016

Learning outcomes

The purpose of this assignment is to target the following learning outcomes:

- Gain confidence programming in C++.
- Gain confidence in creating and using data structures.
- Apply known algorithms to solve problems.
- Continue to develop your skills in OOP design.
- Apply good coding practices such as naming conventions and code reuse.

1 Simulation of Traffic flow at Intersections

The simulation of traffic flow is often used to try and model the effect of infrastructure projects and predict overall transportation network effectiveness. As part of such a simulation system, you will write part of a program in C++ to simulate traffic flow at intersections, specifically those with **four** paths for traffic flow. For the purposes of the simulation, traffic may only flow in one direction along any given road. An example of the intersections that will be studied in this assignment is shown in figure 1.

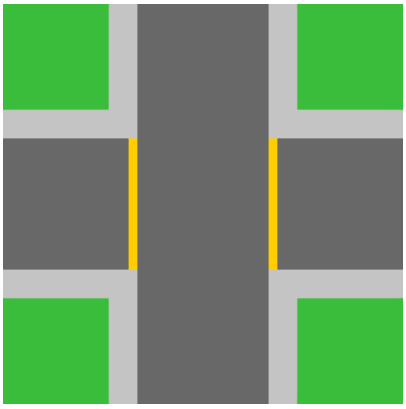


Figure 1: A four–way intersection made up of two incoming lanes and two outgoing lanes.

Intersections will be simulated using queues for each incoming and outgoing traffic lane. The `Intersection` class will have a `simulate` method which allows one or more vehicles to proceed through the intersection each time it is called (assuming there are vehicles waiting at the intersection). Each vehicle will have a type (car, motorcycle, bus) and a direction they want to turn at the intersection (left, right, straight). Using this information, your code will marshal vehicles through the intersection periodically.

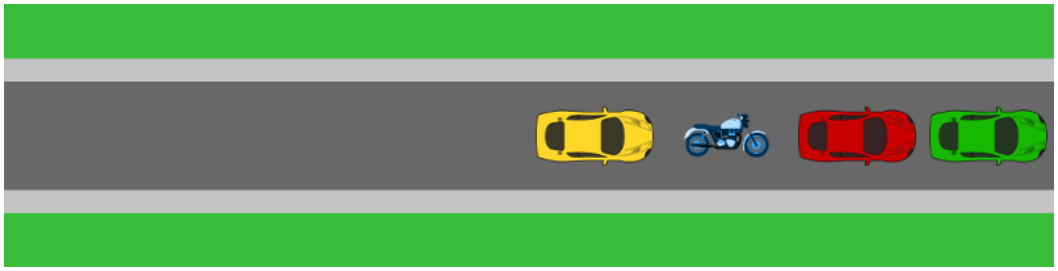


Figure 2: A `SimpleLane` with several `Vehicle`s queued. The yellow car is the most recently `enqueued` `Vehicle`, while the green car will be the next to be returned by the `dequeue` method.

2 Tasks

2.1 Simulating Lanes of Traffic [Task 1]

For the purposes of this assignment, lanes of traffic will be simulated using queues. Your implementation of these lanes must adhere to the `Lane` interface and return dequeued `Vehicle`s in the same order they were enqueued (FIFO). Your lane classes should use a **linked list** data structure internally to manage pointers to the `Vehicle` objects queued in the lane. Note that you should **not** modify the `Lane` class at all; it provides the abstract interface for lanes, and defines how lanes can be used. Instead you will implement lane behavior within subclasses which are derived from the `Lane` class. These classes are explained in sections 2.1.1 and 2.1.2.

Make sure when using lane classes that you only pass pointers to *heap-allocated* `Vehicle` objects to the `enqueue` method. The classes derived from `Lane` (`SimpleLane` and `ExpressLane`) should `delete` any `Vehicle` objects still enqueued when the lanes are destroyed (using their destructors). If `delete` is used on a *stack-allocated* `Vehicle` object, your code will crash.
The `Vehicle` class has been made non-copyable to enforce handling of `Vehicle` object via pointers and references, rather than by copying `Vehicle`s from one location to another.

2.1.1 Implement the `SimpleLane` Class

The `SimpleLane` class represents a typical road lane; vehicles enter at one end (via the `enqueue` method) and exit from the other end (from the `dequeue` method). Implement the methods of `SimpleLane` within the file SimpleLane.cpp. Refer to the comments in SimpleLane.hpp for information on what each method should do.

When a `Vehicle` is added to a `SimpleLane` using the `enqueue` method, the `SimpleLane` takes ownership of the enqueued `Vehicle` and so is responsible for deleting the `Vehicle` if it is still enqueued when the `SimpleLane` is destroyed. Use the `SimpleLane` destructor to implement this behavior. This is an example of a *composition* relationship, where "the whole" (`SimpleLane`) dictates the lifetime of "the part" (`Vehicle`s).

Your code will not compile until you have created implementations for all of the methods defined in SimpleLane.hpp and ExpressLane.hpp, refer to section 2.1.3 for more information.

2.1.2 Implement the `ExpressLane` Class

Figure 3: Motorcycles filtering through traffic at an intersection in Bangkok, Thailand. *CC BY 2.0 Roland Dobbins.*

The `ExpressLane` class will simulate the effects of motorcycle Lane Filtering on traffic flow.Filtering refers to a process where motorcyclists travel at low speed between cars in stopped traffic to make their way to the front of the traffic queue. Since all of the traffic in our simulation is queued, we will assume it is possible for motorcyclists to filter through lanes (even single lanes in this case).

Implement the `ExpressLane` class so that when a `Vehicle` object whose `type` is `VT_MOTORCYCLE` is added to the `ExpressLane` using the `enqueue` method, that `Vehicle` is inserted into the lane in front of all `VT_CAR` or `VT_BUS` `Vehicle` objects (if any are queued), but behind any other motorcycles already queued in the lane. Refer to figure 4 for an example of this behavior.

Except for the changes to the `enqueue` method described above, the `ExpressLane` class should have the same behavior as the `SimpleLane` class. For this reason, `ExpressLane` should inherit from the `SimpleLane` class to re-use the existing implementations of methods which are the same in both classes.

2.1.3 Testing the Lane Classes

Note that your code will not compile or be testable until you have created bodies for all methods defined in the SimpleLane.hpp and ExpressLane.hpp headers. Without bodies for these methods, you will get compiler errors similar to the following:

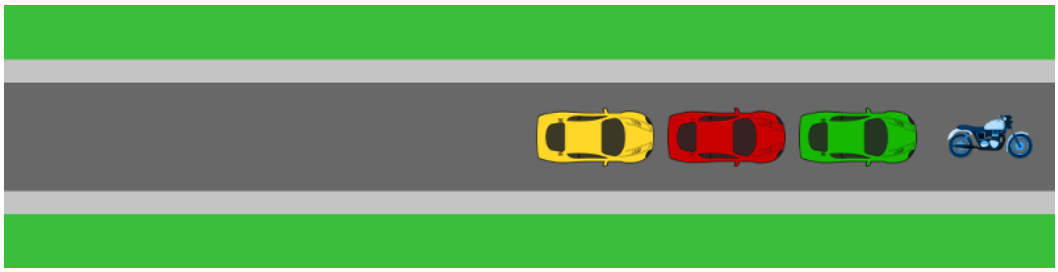
```
/tmp/ccj0fAGl.o: In function `test_SimpleLaneConstruction()':
test.cpp:(.text+0xb4a): undefined reference to `SimpleLane::SimpleLane()'
```

These are errors in the *linking* phase of the compilation process, and indicate that the compiler cannot find a body for one or more function/methods defined in a header. This can be fixed by creating function/method bodies (even empty ones) for functions/methods in the implementation (.cpp) files corresponding to the relevant headers.

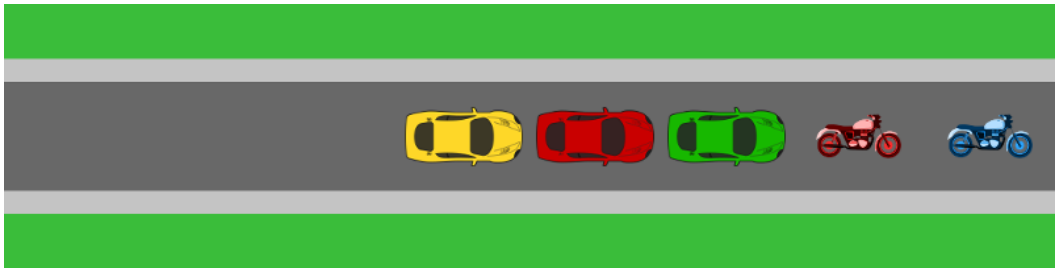
Some tests for the `SimpleLane` and `ExpressLane` classes have been included in the test.cpp file given to you for this assignment. To run the tests, run the command `make test` from a command-line terminal within the directory containing the code that you were provided for this assignment. The `test` target in the `Makefile` builds the executable `traffic_test` from the code in test.cpp and runs it, executing all enabled tests. If you need to run

(a) An `ExpressLane` with two cars queued.

(b) The `ExpressLane`'s `enqueue` method is used to add a blue motorcycle, which is immediately put ahead of the non-motorcycle `Vehicle`s already in the lane.



(c) A third (yellow) car is added to the `ExpressLane` using the `enqueue` method, and is added to the end of the lane.



(d) A second (red) motorcycle is added to the `ExpressLane`, moving ahead of all cars in the lane, but behind the first motorcycle.

Figure 4: A demonstration of the lane filtering behavior of the `ExpressLane` in action.

a specific test you can run `./traffic_test 3` to run only the third test (replace 3 with the number of the test that you want to run).

The test executable will report any test failures and print out information about why the test failed. For example:

```
FAILURE: test.cpp:79
    Expression 'c.occupantCount() == 2' evaluated to false
FAIL: Test 0 failed.
```

In this example the test executable reports that the test is failing because the return value of `c.occupantCount()` is not equal to 2. If we inspect the test code (test.cpp line 79, according to the error message) we can see the context for this message:

```
/*
Test setting the vehicle type and number of occupants in the Vehicle constructor
*/
TestResult test_VehicleContstruction() {
    {
        Vehicle c(Vehicle::VT_CAR, 2);
        ASSERT(c.type() == Vehicle::VT_CAR);
        ASSERT(c.occupantCount() == 2); // <--- failing here
    }

    {
        Vehicle b(Vehicle::VT_BUS, 14);
        ASSERT(b.type() == Vehicle::VT_BUS);
        ASSERT(b.occupantCount() == 14);
    }

    {
        Vehicle b(Vehicle::VT_MOTORCYCLE, 1);
        ASSERT(b.type() == Vehicle::VT_MOTORCYCLE);
        ASSERT(b.occupantCount() == 1);
    }

    return TR_PASS;
}
```

So the test is creating a new `Vehicle` of the `VT_CAR` type with 2 passengers, and then checking that the passenger count has correctly been set to 2. Inspecting the code for the `Vehicle` class might reveal that the passenger count is never initialized in the class constructor:

```
Vehicle::Vehicle(Type newType, unsigned int occupantCount) : vehicleType(
    newType) { }
```

Once the error is fixed by properly initializing the passenger count, the test will pass:

```
Vehicle::Vehicle(Type newType, unsigned int occupantCount) : vehicleType(
    newType), occupants(occupantCount) { }
```

```
INFO: Executing test 1
PASS: Test 1 passed.
```

Making sure that your program passes all of the tests you have been given will not guarantee that you will get 100% of the marks for this assignment, but it will mean that you get at least some marks for the parts that work according to those tests. Writing your own tests or extending the ones you have been given is strongly recommended, as there are many requirements in this assignment that the existing tests do not cover. **If you have created some awesome new test cases, you are welcome to share these with your peers (e.g. on Piazza). Make sure that only test case code is shared, not any portion of your solution for the assignment!**

Take special care when dealing with pointers; if you try and use (dereference) an invalid pointer (e.g. a null pointer) you will most likely trigger a crash during testing, usually a segmentation fault (also called a `SIGSEGV` or Segfault).

```
make:  *** [Makefile:10:  test] Segmentation fault (core dumped)
```

If your code crashes during testing, you will receive no marks for the tests that crash. Pay special attention to the comments in the headers that tell you which methods should accept a null pointer. Some methods may be required to *return* null pointers in certain scenarios; make sure you don't accidentally pass these null pointers into other methods that cannot handle them.
If you create a bug which results in a crash, you can debug the crash using the `gdb` command line tool. `gdb` is the GNU Debugger, and part of the same tool set as the `gcc`/`g++` compiler. The executables produced by the `Makefile` included with this assignment already have debug information added to them during the build process. For example, you can debug the `traffic_test` executable using the command `gdb traffic_test`. For more information refer to one of the many gdb tutorials available online.

2.2 Simulating Intersection Road Rules [Task 2]

2.2.1 Implement the `Intersection` Class

In this task you will create a new class called `Intersection` that *aggregates* four `Lane` objects and manages the exchange of traffic between them. Each connected `Lane` is attached to one side of the 4-way intersection. These sides are labeled north, east, south and west, similar to the directions on a compass (refer to figure 6). When the `simulate` method is called on an `Intersection`, one or more `Vehicle`s will be moved from the incoming `Lane`(s) of the `Intersection` to the outgoing `Lane`(s). Which `Vehicle`(s) are moved in each `simulate` call will be determined by the following simple road rules:

- Right-turning `Vehicle`s must give way to other `Vehicle`s waiting to turn left or travel straight through the intersection.
- Left-turning `Vehicle`s must give way to `Vehicle`s waiting to go straight through the intersection.
- Any vehicles waiting at the intersection which comply with the above two rules may proceed through the intersection. No more than one vehicle from each incoming `Lane` may proceed through the intersection in each call to the `simulate` method.
- You may assume that `Vehicle`s will not try to make invalid turns, such as turning into an incoming `Lane`.

Note that with the above road rules, multiple `Vehicle`s traveling straight through an `Intersection` (for example) may all proceed through the `Intersection` at once (within the same call to `simulate`). In real life this would cause an accident between vehicles in adjacent incoming lanes of an intersection, but for the purpose of simplifying this simulation it will be allowed.

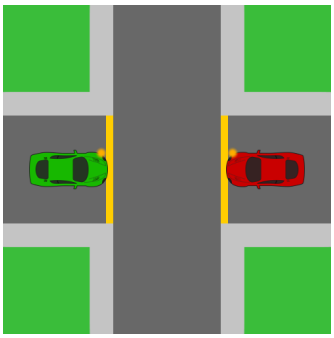
When a `Vehicle` goes through an intersection:

- The `Vehicle` should be removed from the `Lane` it is in using the `Lane`'s `dequeue` method.
- The `Vehicle` should be added to the `Lane` it is trying to get into using that `Lane`'s `enqueue` method.
- The `Vehicle`'s `makeTurn` method should be called so that it can update the `nextTurn` property.

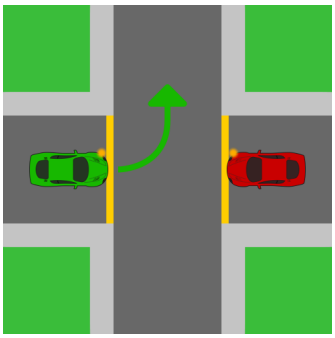
An example of what the `Intersection` does when its `simulate` method is called and how the road rules apply is shown in figure 5.

Complete the definition and implementation for this class in the files Intersection.hpp and Intersection.cpp. You may not modify the existing method declarations in Intersection.hpp, but you may add new declarations of your own. Refer to the comments in Intersection.hpp for more information about how to implement the `Intersection` class.

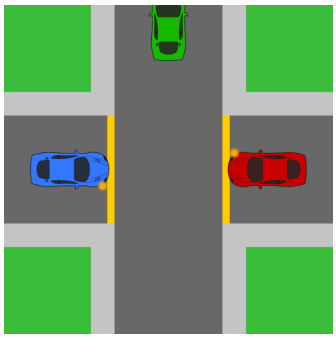
Your Intersection class will be initially tested with the `Lane` configurations shown in figure 7. It is recommended that you add tests to test.cpp for the configurations you have not been given tests for to ensure that your `Intersection` and `Lane` classes can handle all four configurations.



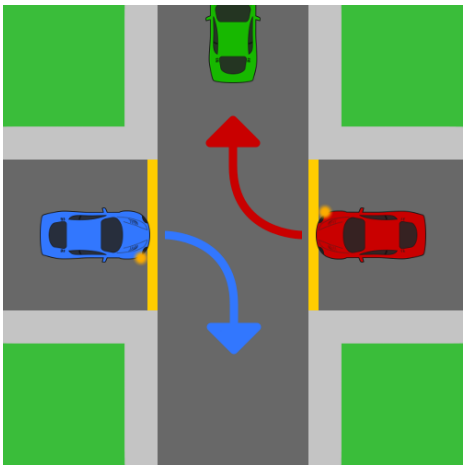
(a) The initial state of the `Intersection`, with a green car at the front of the incoming western `Lane` waiting to turn left, and a red car at the incoming eastern `Lane` waiting to turn right.



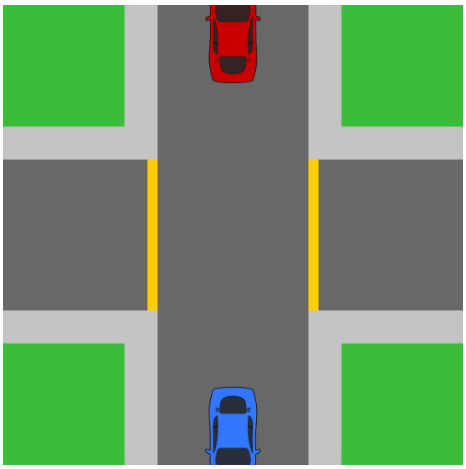
(b) The `Intersection`'s `simulate` method is called. The red car cannot proceed as it must give way to the left-turning green car. The green car is dequeued from the western `Lane` and enqueued into the northern outgoing `Lane`.



(c) After the first `simulate` call, the green car has been enqueued in the northern `Lane`, and there is now a right-turning blue car at the front of the western `Lane`.



(d) The `Intersection`'s `simulate` method is called again. This time, none of the `Vehicles` at the front of the incoming `Lanes` need to give way to each other. Both vehicles are dequeued from their current `Lane` and enqueued in the `Lane` they are trying to turn into.



(e) After the second simulate call, the red car has been enqueued in the northern `Lane`, and the blue car has been enqueued in the southern outgoing `Lane`. There are no more `Vehicles` enqueued in the incoming `Lanes`, so further calls to the `Intersection`'s `simulate` method will have no effect. The northern outgoing `Lane` has two `Vehicles` enqueued (the green and red cars), while the southern outgoing `Lane` has one `Vehicle` enqueued (the blue car).

Figure 5: Demonstration of give-way behavior and vehicle progression through an intersection when the `Intersection`'s `simulate` method is called.

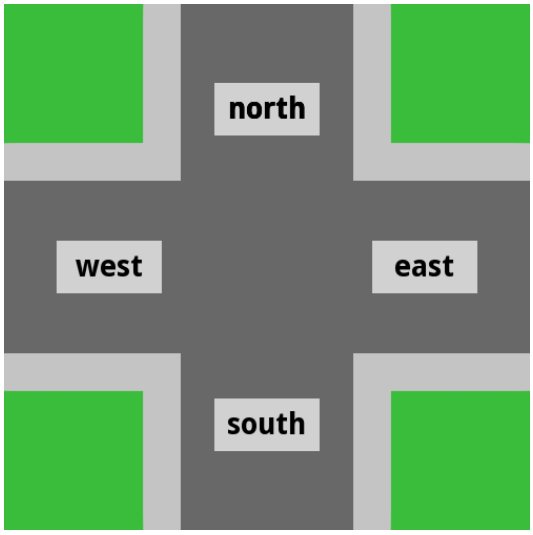
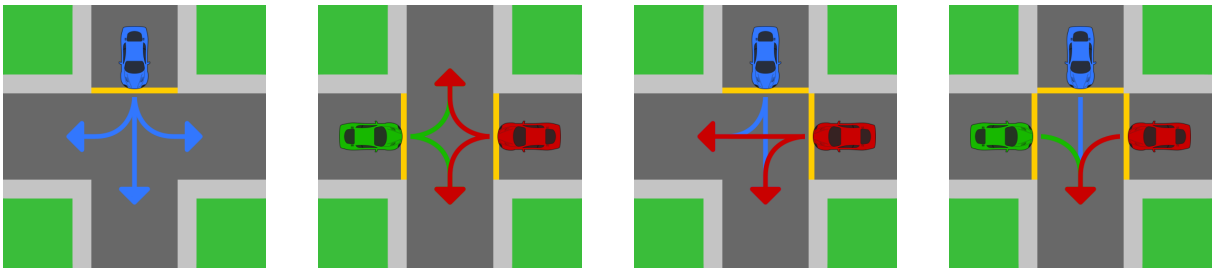


Figure 6: The relative positions of `Lane` objects connected to an `Intersection`.



(a) `Intersection` with one incoming `Lane` and three outgoing `Lanes`.

(b) `Intersection` with two opposing incoming `Lanes` and two opposing outgoing `Lanes`.

(c) `Intersection` with two adjacent incoming `Lanes` and two adjacent outgoing `Lanes`.

(d) `Intersection` with three incoming `Lanes` and one outgoing `Lane`.

Figure 7: `Intersection` configurations

2.2.2 Creating a Traffic Network

Your `Lane` and `Intersection` classes will be used to simulate a small traffic network made up of **four** interconnected `Intersections` (see figure 8). In particular, your `Intersection` class must call the `makeTurn` method on `Vehicles` as they are propagated through the `Intersection` so that `Vehicles` iterate through their internal list of turns correctly. If you have fully and correctly implemented the behavior described in section 2.2.1 then your code should pass this test without any additional changes.

The traffic network will be simulated by calling the `simulate` method on each `Intersection` in the network, starting with the top left `Intersection` and proceeding clockwise around the network. It is strongly advised that you create a test case to check that your code works for this scenario.

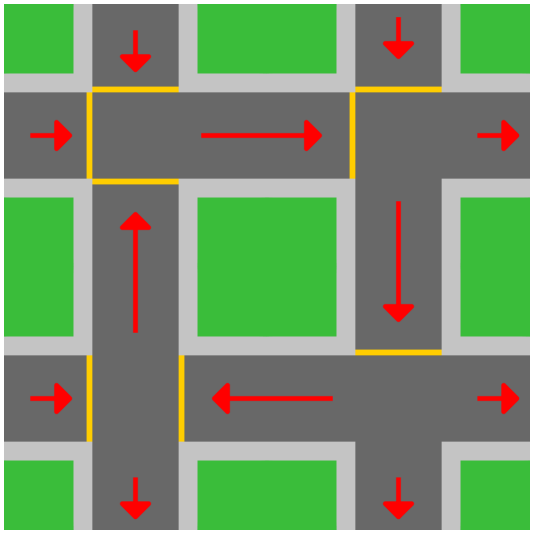


Figure 8: Traffic flow for the multiple-`Intersection` traffic network test.

2.2.3 Testing `Intersections` and Road Rules

The tests for the `Intersection` class can be enabled by uncommenting the following line at the top of test.cpp:

```
// #define ENABLE_T2_TESTS
```

Once the Task 2 tests are enabled, run `make test` to execute all of the currently enabled tests. For more details on testing this assignment, refer to section section 2.1.3. Remember that not all aspects of the `Intersection` class may be tested in the tests you have been given, so **don't assume the task is complete once all of the tests pass**.

Important: how your code will be marked

- Your code will be marked using a semi-automated setup. If you fail to follow the setup given, your code **will not be marked**. All submitted files must compile without requiring any editing. Use the provided tests and `Makefile` to ensure your code compiles and runs without errors. Any tests that run for longer than 10 seconds will be terminated and will be recorded as failed.
- As this is a data structures and algorithms assignment, you may not use any data structures or algorithms that you have not written yourself. **You may not use any data structure or algorithm libraries from the C++ standard library/STL.** This includes:
  - `<vector>`
  - `<deque>`
  - `<list>`
  - `<set>`
  - `<map>`
  - `<stack>`
  - `<queue>`
  - `<algorithm>`
  - `<numeric>`
- You **may not** modify `Vehicle.hpp` or `Vehicle.cpp` at all.
- Although you may add more to them (e.g. member variables, `#include` statements, or helper functions), you must not modify the existing interface of classes defined in the following files (e.g. do not delete or modify the existing functions declared):

  - `Traffic/Intersection.hpp`
  - `Traffic/Lane.hpp`

- You may make any modifications to the `SimpleLane` and `ExpressLane` classes provided that:
  - These classes retain their current names (`SimpleLane` and `ExpressLane`).
  - These classes are both derived either directly or indirectly from the `Lane` class.
- Do not move any existing code from the `Traffic` directory, and make sure all of your new code files are created inside the `Traffic` directory.
- You may modify `test.cpp` as you please (for your own testing purposes); this file will not be marked at all. Be aware that your code must still work with the original `test.cpp`.
- Your code will also be inspected for good programming practices, particularly using good object-oriented principles. Think about naming conventions for variables and functions you declare. Make sure you comment your code where necessary to help the marker understand **why** you wrote a piece of code a specific way, or **what** the code is supposed to do. Use consistent indentation and brace placement.

Submission

You will submit via Canvas. Make sure you can get your code compiled and running with both main.cpp and test.cpp (`make run` and `make test`) on the university Ubuntu computers. Submit the following, in a single ZIP archive file:

- A **signed and dated declaration** stating that you worked on the assignment independently, and that it is your own work. Include your name, ID number, the date, the course and assignment number. You can find this Cover Sheet on Canvas. All code will be checked against other submissions. Submissions detected as being similar to others will ensure that the students involved are forwarded to the Misconduct Committee.
- The entire contents of the src_for_students folder you were given at the start of the assignment, including the new code you have written for this assignment. Ensure you **execute `make clean` before zipping the folder** so your submission doesn't include any executable files (your code will be re-built for marking).

Academic honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help. Under no circumstances should you take or pay for an electronic copy of someone else's work.
- All submitted code will be checked using software similarity tools. Submissions with suspicious similarity will result in an Investigative Meeting and will be forwarded to the Disciplinary Committee.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never show or give another person your code.
  - Never put your code in a public place (e.g. Reddit, Github, forums, your website).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.

Late submissions

Late submissions incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late (dropbox automatically closes)

You must double check that you have uploaded the correct code for marking! There will be no exceptions if you accidentally submitted the wrong files, regardless of whether you can prove you did not modify them since the deadline. No exceptions.