Update: Monday March 6: Section 5.5.4 - add a `getNumOrders()` function to the `Entity` class.

1 Submission Instructions

Submit to Brightspace on or before the due date a compressed file (.tar or .zip) that includes

1. Header and source files for all classes instructed below.
2. A working Makefile that compiles and links all code into a single executable. The Makefile should be specific to this assignment - do not use a generic Makefile.
3. A README file with your name, student number, a list of all files and a brief description of their purpose, compilation and execution instructions, and any additional details you feel are relevant.

2 Learning Outcomes

In this assignment you will learn to

1. Implement class hierarchies.
2. Implement static members.
3. Use a Queue (a type of first-in first-out Linked List data structure).
4. Decide on and implement your choice of data structure(s).
5. Decide how to manage your memory.
6. Write a UML diagram that documents classes and the interaction between between classes.

3 Overview

In this assignment you will implement an application that assigns and tracks deliveries for Pierre's Poutine. Pierre's Poutine is a very popular franchise with multiple locations across the city and multiple delivery drivers on duty. This application records orders made by customers, assigns them to a Pierre's franchise location, and then, when a location has enough orders, calls a delivery driver to pickup and deliver the orders.

4 Classes Overview

This application will consist of 9 classes.

1. The `Location` class (Entity object):
   (a) Contains map coordinates as well as the street names.
2. The `Order` class (Entity object):
   (a) Contains information about a customer's `Order` from `Pierres`.
3. The `Entity` class (Entity object):
   (a) Contains `Pierres` user information - base class for `Franchises` and `Drivers`.

4. The `Franchise` class (Entity object):
   (a) An `Entity` representing a `Pierres` franchise location.
5. The `Driver` class (Entity object):
   (a) An `Entity` who pickups and delivers `Orders`.
6. The `Queue` class (Container object):
   (a) A 'first-in first-out' data structure made from a linked list to store `Orders`.
7. The `Pierres` class.
   (a) Manages the `Drivers` and `Franchises` of `Pierres`.
   (b) Coordinates taking, picking up, and delivering `Orders`.
8. The `Controller` class (Control object):
   (a) Controls the interaction of `Pierres` with the `View`.
9. The `View` class (Boundary object):
   (a) Provides a user menu, takes input, and gives output.

In addition, we will be using `std::cout` as the main View output object for error reporting.

5 Instructions

Download the starting code from Brightspace. It includes some global functions that you are to use for testing. All member variables are `private` unless otherwise noted. All member functions are `public` unless otherwise noted. Some return values are not explicitly given. You should use your best judgment (they will often be `void`, but not always). ALL CLASSES MUST HAVE A PRINT FUNCTION unless otherwise noted. This print function should display the metadata of the class using appropriate formatting.

Your finished code should compile into a single executable called `a3` using the command `make all`. Your submission should consist of a single zip file with a suitable name (e.g., `assignment3.zip`) that contains a folder containing all your files. This folder should also contain a README with your name, student number, a list of all files that are included, a directory structure (if applicable), compiling and running instructions, and any other information that will make the TAs life easier when they mark your assignment.

You should manage the memory properly **in any way you see fit.** As such you should add destructors to any class you think appropriate.

5.1 defs.h

This file contains a single preprocessor constant, `MENU_ITEMS`, which gives the number of menu items in a `Pierres` restaurant.

5.2 The Location Class

The `Location` class has been provided for you. The distance you get from the `getDistance` function is calculated using *Manhattan distance*. Since this fictional city is essentially a grid, to go from 0,0 to 2,3 means we travel 2 blocks east and 3 blocks south (in any order), for a total distance of 5 blocks.

5.3 The Order Class

This represents an order made by a customer. To keep it simple we will assume everyone orders exactly 1 menu item.

1. Member variables:
   (a) `string`: The customer's name.
   (b) `int`: The menuItem ordered.
   (c) `Location`: The location to be delivered to.
2. Member functions:
   (a) Make a constructor that takes `string, int, Location` as arguments and initializes the member variables.
   (b) Make a getter for the `Location` member.
   (c) `print` - this should print the `name` and the menu item (use menuItem as an index into `Franchise::menu` - see the `Franchise` class).

5.4 The Queue Class

This has a similar structure to the `List` class we saw in class. You may use any code from the `List` class to put into `Queue`, but in addition you will implement `Queue` specific behaviour.

1. Nested class - make a private nested class `Node`. You may use the `Node` class from the `List` class, however, change the `data` to type `Order*`.
2. Member variables:
   (a) `Node* head` - same function as the `head` variable in the `List` class.
   (b) `Node* tail` - similar to `head` except `tail` should always point to the last element in the `Queue`. This will make it easy to add elements to the back of the `Queue`.
   (c) Keep track of the number of `Orders` in this `Queue`.
3. Constructor - initialize both `head` and `tail` to `NULL`.
4. Destructor - Delete all `Node`s in the `Queue`.
5. Member functions:
   (a) `empty()` - return true if the `Queue` is empty.
   (b) `size()` - return the number of `Orders` in the `Queue`.
   (c) `Order* peekFirst()` - return the `Order*` data from the first location if it exists, or return `nullptr` otherwise. DO NOT delete the `Node` - this function returns the data without removing it from the `Queue`.
   (d) `Order* popFirst()` - return the `Order*` data from the first location if it exists, return `nullptr` otherwise. Delete the `Node` if it exists making sure to update the `head` and `tail` pointers if necessary.
   (e) `addLast(Order* order)` - Add `order` to the back of the `Queue`. For full marks this should make use of the `tail` pointer in order to add the `order` without traversing the entire `Queue`.

5.5 The Entity Class

1. Member variables (these should be `protected`):
   (a) `string id`: The unique id of this `Entity`.
   (b) `string name`: The name of the `Driver` or `Franchise` owner.
   (c) `Location`: The location of this `Entity`.
   (d) `Queue orders`: The orders at this `Franchise` awaiting pickup by a `Driver`, or the orders currently being delivered by a `Driver`.
2. Make a setter for `Location`.
3. Make a constructor that takes the following 4 arguments in order: `char, int, string, Location`. This constructor should concatenate the character code with the first int and store it in the `id` string. For instance, if I pass in 'F' and 23 as arguments, then `id = "F23"`. The `char` will be used to differentiate between `Drivers` and `Franchises` (and any future types we wish to add). The `string` is the `name` of the `Franchise` owner or Driver.
4. **Update Monday March 6:** Make a `getNumOrders()` function that returns the number of `Orders` this `Entity` has in its `Queue`.
5. Make an `Order* getNextOrder` function that removes and returns the `Order` at the front of the `Queue` if one exists, or returns `nullptr` otherwise.
6. Make an `addOrder(Order* order)` function that adds the given `Order*` to the back of the `Queue`.
7. Make a `print` function that prints out all the information including `Location` and number of `Orders` (you do not need to print out the `Orders` themselves).
8. Make a `equals` function that accepts a `string id` as an argument and returns `true` if this `Entity::id` is equal to it and `false` otherwise.

5.6 The Driver Class

This class should inherit from `Entity`.

1. Add the following ***class*** variables:
   (a) `const char code`: This should be set to 'D' (for driver).
   (b) `int nextId`: This will produce the next id number to be passed into the `Entity` constructor.
2. Member functions:
   (a) Make a constructor that takes `string name` and `Location` as arguments, in that order. Initialize all member variables. Provide a default constructor by using default arguments.
   (b) Make an `isFree` function. Return `true` if the `Driver` currently has no `Orders` in their `Queue`, and `false` if they have one or more `Orders` in their `Queue` (i.e., they are still delivering).
   (c) Override the `print` function. This should behave similar to `Entity::print` but should specify that this is a `Driver`.

5.7 The Franchise Class

This class should inherit from `Entity`.

1. Member variables. These should all be ***class*** variables:
   (a) `const char code`: This should be set to 'F' (for Franchise).
   (b) `int nextId`: This will produce the next id number to be passed into the `Entity` constructor.
   (c) `const string menu[MENU_ITEMS]`: This array should be initialized to:
   `{"1. Large Poutine", "2. Medium Poutine", "3. Small Poutine"};`
2. Member functions:
   (a) Make a constructor that takes `string name` and `Location` as arguments, in that order. Initialize all member variables. Provide a default constructor by using default arguments.
   (b) Override the `print` function. This should behave similar to `Entity::print` but should specify that this is a `Franchise`.
3. Class functions:
   (a) `printMenu`: This should print out all elements of the `menu` class variable.
   (b) `string getMenu(int index)`: This should return `menu[index]` if the `index` is valid, or "unknown menu item" otherwise.

5.8 The Pierres Class

This class will will store `Drivers` and `Franchises` and manage their interactions. To do this it will maintain a pair of data structures **of your choice**. You may use `vectors`, or use any class from any previous assignment or class code. For example, to store `Drivers`, you can use `vector<Driver*>`, or you can adapt `PhotoArray` to store `Driver*` instead of `Photo*`, or adapt our linked list example to store `Driver*` as data.

1. Member variables:
   (a) A data structure that contains `Drivers`.
   (b) A data structure that contains `Franchises`.
2. Constructor - initialize any necessary member variables.
3. Member functions - IMPORTANT - Any function that fails in some way should output an appropriate error message to `cout` or `cerr`.
   (a) `addDriver(string name, Location)` - add a new `Driver` with the given parameters.
   (b) `addFranchise(string name, Location)` - add a new `Franchise` with the given parameters.
   (c) `takeOrder(string customerName, int menuItem, Location)` - make a new `Order` with the given criteria and add it to the `Franchise` that is **closest** to the given `Location`.
   (d) `driverPickup(string franchiseId, int numOrders)`: If there is no `Franchise` with the given `id`, output an error message. Otherwise, find the **closest available** `Driver` (the closest `Driver` whose `isAvailable` function returns true). This `Driver` should change their `Location` to match the `Franchise` `Location`, and transfer `numOrders` `Orders` from the `Franchise` to the `Driver`, or all `Orders` if `numOrders` is larger than the number of `Orders` at the `Franchise`.

(e) `driverDeliver(string driverId, int numOrders)`: If there is no `Driver` with the given `id`, output an error message. Otherwise have the `Driver` "deliver" `numOrder` `Orders` (or as many `Orders` as the `Driver` has if the `Driver` has fewer than `numOrders`). For each `Order` delivered, have the `Driver` change their `Location` to the `Location` of that `Order`, and output something useful to `cout` such as "Delivering: " and then print the `Order`. After the `Order` is delivered, you may delete the `Order`. After delivering all `Orders`, the `Driver` should be at the `Location` of their last delivery.

(f) `printFranchises` - print all the `Franchises`.

(g) `printDrivers` - print all the `Drivers`.

4. You may wish to add private member functions to use as helper functions (but you don't have to). For example, you may wish to write a `Driver* findClosestDriver(Location)` function, or a similar function for `Franchises`.

5.9 The Controller and View Classes

These classes have been done for you. Also a main.cc file is provided to launch your application.

5.10 UML Class Diagram

Draw a UML class diagram of the finished application using any UML drawing program you like (though draw.io is excellent). You must represent inheritance and composition but do not need to represent "uses". Be sure to represent all member variables and member functions (with the exceptions of simple getters, setters, and print). Do **NOT** show collection classes in your `UML` diagram.

6 Grading

The marks are divided into three main categories. The first two categories, **Requirements** and **Constraints** are where marks are earned. The third category, **Deductions** is where you are penalized marks.

6.1 Specification Requirements

These are marks for having an application that works as requested (even when not implemented according to the specification, within reason).

General Requirements

- All marking components must be called and execute successfully to earn marks.
- All data handled must be printed to the screen to earn marks.

Application Requirements: 14 marks

- 2 marks: `Driver` and `Franchise` print correctly.
- 2 marks: `add Driver` and `Franchise` work correctly.
- 2 marks: `printFranchises` and `printDrivers` work correctly.
- 1 mark: `takeOrder` assigns an `Order` to a `Franchise`.
- 1 mark: `takeOrder` assigns an `Order` to the **closest** `Franchise`.
- 1 mark: `driverPickup` has a `Driver` correctly pick up `Orders`.

- 1 mark: `driverPickup` has the **closest** `Driver` pick up `Orders`.
- 2 mark: `driverDeliver` correctly delivers the `Orders`.
- 2 marks: Proper error reporting in `Pierres.`

UML Requirements: 10 marks

- 10 marks: UML diagram is correct.

Requirements Total: 24 marks

6.2 Constraints

The previous section awards marks if your program works correctly. In this section marks are awarded if your program is written according to the specification and using proper object oriented programming techniques. This includes but is not limited to:

- Apply "`const`"-ness to your program.
  - Print statements, getters, and any member function that does not change the value of any member variables should be `const`.
  - Any returned object that will not be changed should be `const`.
  - Any parameter object (passed by reference) that will not be modified should be `const`.
- Proper declaration of member variables (correct type, naming conventions, etc).
- Proper instantiation of member variables (statically or dynamically)
- Proper instantiation of objects (statically or dynamically)
- Proper constructor and function signatures.
- Proper constructor and function implementation.
- Proper use of arrays and data structures.
- Passing objects by *reference* or by *pointer*. Do not pass by value.
- Reusing existing functions wherever possible.
- Proper error checking - check array bounds, data in the correct range, etc.
- Reasonable documentation (remember the best documentation is expressive variable and function names, and clear purposes for each class - I am not a stickler on this, but if you write code that could be confusing, add some comments).

6.2.1 Constraint marks:

1. 2 marks: Proper implementation of the `Order` class.
2. 2 marks: Proper implementation of the `Queue` classes.
3. 2 marks: Proper implementation of the `Entity` class.
4. 2 marks: Proper implementation of the `Driver` class.
5. 2 marks: Proper implementation of the `Franchise` class.
6. 2 marks: Proper implementation of the `Pierres` class.

Constraints Total: 12 marks

Requirements Total: 24 marks

Assignment Total: 36 marks

6.3 Deductions

The requirements listed here represent possible deductions from your assignment total. In addition to the constraints listed in the specification, these are global level constraints that you must observe. For example, you may only use approved libraries, and your programming environment must be properly configured to be compatible with the virtual machine. This is not a comprehensive list. Any requirement specified during class but not listed here must also be observed.

6.3.1 Packaging and file errors:

1. 5%: Missing README
2. 10%: Missing Makefile (assuming this is a simple fix, otherwise see 4 or 5).
3. up to 10%: Failure to use proper file structure (separate header and source files for example), but your program still compiles and runs
4. up to 50%: Failure to use proper file structure (such as case-sensitive files and/or Makefile instructions) that results in program not compiling, but is fixable by a TA using reasonable effort.
5. up to 100%: Failure to use proper file structure or other problems that severely compromise the ability to compile and run your program.

As an example, submitting Windows C++ code and Makefile that is not compatible with the Linux VM would fall under 4 or 5 depending on whether a reasonable effort could get it running.

6.3.2 Incorrect object-oriented programming techniques:

- Up to 10%: Substituting C functions where C++ functions exist (e.g. don't use `printf`, do use `cout`).
- Up to 10%: Memory leaks - be sure to check your code with valgrind.
- Up to 25%: Using smart pointers.
- Up to 25%: Using global functions or global variables other than the `main` function and those functions and variables expressly permitted or provided for initialization and testing purposes.

6.3.3 Unapproved libraries:

- Up to 100%: The code must compile and execute in the default course VM provided. It must NOT require any additional libraries, packages, or software besides what is available in the standard VM.
- Up to 100%: Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*