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 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 for a `CellPhone` carrier. This company tracks `CellPhones` and the cellphone `Towers` that they connect to in order to send `Messages`. To keep things simple, we will only track text `Messages` between customers.

4 Classes Overview

This application will consist of 11 classes.

1. The `Location` class (Entity object):
   (a) Contains map coordinates as well as the street names.
2. The `Message` class (Entity object):
   (a) This class tracks who the sender, receiver, and content of the message.
3. The `Entity` class (Entity object):
   (a) This class encapsulates the entities in this company that connect wirelessly - `CellPhones` and `Towers`.
4. The `CellPhone` class (Entity object):
   (a) An `Entity` representing a cellphone. It can connect and transmit through a `Tower` as well as change `Locations`.

5. The `Tower` class (Entity object):
   (a) An `Entity` representing a cellphone tower. It can receive and send information to `CellPhones`, but has a fixed `Location`.
6. The `List` class (Container object):
   (a) A data structure made from a linked list to store `Messages`, with some specialized functions specific to our application.
7. The `Network` class.
   (a) A class that manages the `Entities` (`CellPhones` and `Towers`).
   (b) Records what `Messages` were sent to and from which `CellPhones` and what `Towers` they were routed through.
8. The `Control` class (Control object):
   (a) Controls the interaction of `Network` with the `View`.
9. The `TestControl` class (Control object):
   (a) Controls the interaction of `Network` with the `View` for test purposes.
10. The `View` class (Boundary object):
    (a) Provides a user menu, takes input, and gives output.
11. The `Tester` class.
    (a) Provides some commonly used test functions.

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 two executables, one called `a3` and one called `test` using the command `make all`. The Makefile is provided for you. 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 major changes you made to the specification.

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 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.

5.2 The Message Class

This represents a message sent or received between two `CellPhones`.

1. Member variables:
   (a) `string sender`: The id (see the `Entity` class) of the sender.
   (b) `string receiver`: The id of the receiver.
   (c) `string message`: The content of the message.
2. Make getters for `sender` and `receiver`.
3. Make a `print` function.

5.3 The List Class

This has a similar structure to the `List` class we saw in class (which you may use as a starting point).

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 `Message*`.
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 `List`. This will make it easy to add elements to the back of the `List`.
   (c) An `int` to keep track of the number of `Messages` in this `List`.
3. Constructor - initialize both `head` and `tail` to `NULL`.
4. Destructor - Delete all `Node`s in the `List`.
5. Member functions:
   (a) `isEmpty()` - return true if the `List` is empty.
   (b) `getSize()` - return the number of `Messages` in the `List`.
   (c) `Message* removeFirst()` - return the `Message*` 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.
   (d) `add(Message* message)` - Add `message` to the **back** of the `List`. For full marks this should make use of the `tail` pointer in order to add the `message` without traversing the entire `List`.
   (e) `print()`: Print out all the `Messages` in the `List`.
   (f) `getMessagesWith(const string& id, List& outputList):` This should add all `Messages` where `id` was either sender or receiver to the `outputList`.
   (g) `getMessagesWith(const string& id1, const string& id2, List& outputList):` Find all messages that were sent between `id1` and `id2` (either with `id1` as sender and `id2` as receiver or vice versa). Add these messages to `outputList`.
   (h) `removeMessagesWith(const string& id, List& outputList)`: Find all `Messages` where `id` is either the sender or receiver. Remove them from the current list and add them to `outputList`. This can be tricky - you may want to use a helper function.
   Note: this function is only marked in Tutorial 6, not in the assignment, and it is a bonus. So you may get 6/4 on Tutorial 6 for completing this function, and there is no penalty for not completing it (though you must include the function definition and an empty body for compilation purposes).

5.4 The Entity Class

1. `Protected` member variables:
   (a) `string id`: The unique id of this `Entity`.
   (b) `Location`: The location of this `Entity`.
2. `Private` member variables:
   (a) `List messageHistory`: The `Messages` sent or received by this `CellPhone`, or the `Messages` that have been routed through this Tower.
3. Make a constructor that takes the following 3 arguments in order: `char, int, 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 'C' and 23 as arguments, then `id = "C23"`. The `char` will be used to differentiate between `CellPhones` and `Towers` (and any future types we wish to add).
4. Make a getter for `messageHistory` that returns a pointer to a `List`. **Note:** We return a pointer here rather than a reference as it will make one of the upcoming operations simpler.
5. Make a getter for `Location`.
6. Make an `addMessage(const Message& m)` function. This function should make a copy of `m` and add it to the `messageHistory`.
7. Make a `getNumMessages()` function that returns the number of `Messages` this `Entity` has in its `List`.
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.
9. Make a `print` function that prints out the `id`, the `Location`, and the number of `Messages` (you do not need to print out the `Messages` themselves).

5.5 The Tower Class

This class should inherit from `Entity`.

1. Add the following ***class*** or ***static*** members:
   (a) `const char code`: `Tower`'s id will have the prefix 'T', thus set this code to 'T'.
   (b) `int nextId`: This will produce the next id number to be passed into the `Entity` constructor.
   (c) Make a `resetId()` function - this functions should set `nextId` to 0.
2. Add the following member functions:
   (a) Make a constructor that takes a `Location` as an argument that has a default value. Initialize the location member.
   (b) Override the `print` function. This should behave similar to `Entity::print` but should specify that this is a `Tower`.

5.6 The CellPhone Class

This class should inherit from `Entity`.

1. Add the following ***class*** or ***static*** members:
   (a) `const char code`: `CellPhone`'s id will have the prefix 'C', thus set this code to 'C'.
   (b) `int nextId`: This will produce the next id number to be passed into the `Entity` constructor.
   (c) Make a `resetId()` function - this functions should set `nextId` to 0.
2. Member variables:
   (a) Add a `string number` variable.
3. Member functions:
   (a) Make a constructor that takes `string number` and `Location` as arguments, in that order. Initialize all member variables.
   (b) Make a `setLocation` function.
   (c) Make a `getMessagesWith(const string& id, List& messages)` function. Add all `Messages` that have `id` as the sender or receiver to the list of `messages`.
   (d) Override the `print` function. This should behave similar to `Entity::print` but should include the phone number.

5.7 The Network Class

This class will will store `CellPhones` and `Towers` and track their associated `Messages`. 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 `CellPhones`, you can use `vector<CellPhone*>`, or you can adapt `AppArray` to store `CellPhone*` instead of `App*`, or adapt our linked list example to store `CellPhone*` as data. A couple of notes:

- When a parameter type is not specified exactly, you should infer it and use best practices (pass by reference, make it const if possible, etc).
- When asked to *report* certain information, you should write an appropriate message to `cout`.

1. Member variables:
   (a) A data structure that contains `CellPhones`.
   (b) A data structure that contains `Towers`.
2. Constructor - initialize any necessary member variables.
3. Member functions:
   (a) `addCellPhone(number, location)` - add a new `CellPhone` with the given parameters.
   (b) `addTower(location)` - add a new `Tower` with the given parameters.
   (c) `routeMessage(message):` The `message` contains the ids of the sender and receiver. Use these ids to either find the matching `CellPhones`, or report that you could not find them. If both `CellPhones` exist you should add this `message` to both of them. In addition you should find the **closest** `Tower` to each `CellPhone`. They might both be closest to the same `Tower`. Add the `message` to this/these `Tower`(s). This is done so that there is a record of how often a `Tower` is used. You may wish to write a helper function here - see part 4 below.

(d) `moveCellPhone(id, location)`: Find the `CellPhone` with the given id or report that you could not find it. If found, update the location.

(e) `getMessageHistory(const string& id, const List** messages)`: The parameter types must be exactly as shown. Find a `CellPhone` or `Tower` with the given id. If you cannot find either report an error. Otherwise retrieve the message history from the `Entity` and assign it to `messages`.

(f) `getMessagesWith(const string& id1, const string& id2, List& outputList)`: Find all messages that were sent between `id1` and `id2` (either with `id1` as sender and `id2` as receiver or vice versa). Add these messages to `outputList`.

(g) `resetIds()`: reset the `nextId` variable of both the `Towers` and `CellPhones` (this was necessary for testing purposes).

(h) `printTowers` - print all the `Towers`.

(i) `printCellPhones` - print all the `CellPhones`.

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 `Tower* findClosestTower(Location)` function, or a getter that returns the `Tower` or `CellPhone` with the given `id`.

5.8 TestControl, Control, and View Classes

These classes have been done for you, as has the Makefile. A main.cc and test.cc file are provided to launch your application in either interactive mode (`./a3`) or test mode (`./test`).

Note on the `Control` class: when you are selecting either a `Tower` or `CellPhone`, you may simply use the number instead of the whole id. That is, if you wanted to select `CellPhone C4`, instead of entering 'C4' you may just enter '4'.

5.9 Tester

You should not change this class (though you may add `cout` or `cerr` statements to track errors).

5.10 UML Class Diagram

Draw a UML class diagram of the finished application using any UML drawing program you like (though draw.io works pretty well). You do not need to include `Control`, `View`, `TestControl` or `Tester`. 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: 16 marks

1. 2 marks: Test add and print `Towers`.
2. 2 marks: Test add and print `CellPhones`.
3. 2 marks: Test send message.
4. 2 marks: Test print messages in `Tower`.
5. 2 marks: Test print messages in `CellPhone`.
6. 2 marks: Test print conversation.
7. 2 marks: Test move `CellPhone`.
8. 2 marks: Proper messaging and error reporting in `Network.`

UML Requirements: 10 marks

- 10 marks: UML diagram is correct.

Requirements Total: 26 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 `Message` class.
2. 2 marks: Proper implementation of the `List` classes.
3. 2 marks: Proper implementation of the `Entity` class.
4. 2 marks: Proper implementation of the `CellPhone` class.
5. 2 marks: Proper implementation of the `Tower` class.
6. 2 marks: Proper implementation of the `Network` class.

Constraints Total: 12 marks

Requirements Total: 26 marks

Assignment Total: 38 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*.