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. One has been provided for you, but you may modify it or use your own.
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. write an application that is (mostly) separated into into control, view, entity, and collection object classes.
2. use a UML diagram to implement classes and the interaction between between classes.
3. implement a "deep copy" of nested objects.

3 Overview

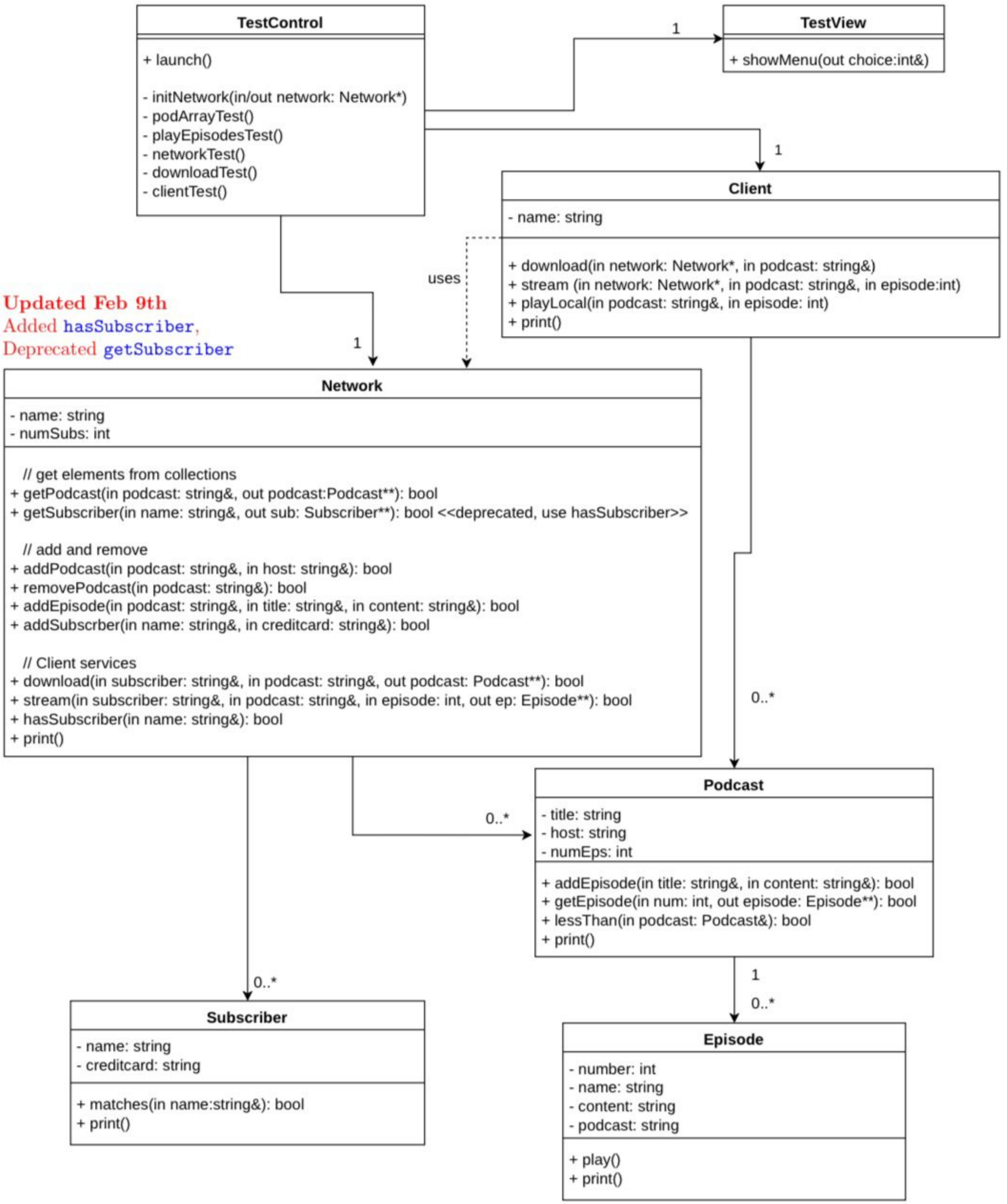
You will be writing C++ code that mimics a podcast network. Podcasts have a title and a host and a data structure for storing 0 or more `Episodes`. Each episode will have some metadata and some content (in our examples the content consists of lorem ipsum gibberish). Episodes may then be "played" (i.e., have their content printed to the console).

The `Network` itself will consist of 0 or more `Podcasts` as well as 0 or more `Subscribers`. A `Client` class will be able connect to the `Network` as long as the `Client name` matches a `Subscriber name`. Once they connect they can "stream" episodes from any podcast. In addition, a `Client` will be able to "download" podcasts. This copies the podcast to "local storage", which, in this exercise, is a data structure in the `Client` class. Subscribers can then play the podcast locally, and this should work even if the network deletes the original podcast (that is, you will be doing a *deep copy* of the `Podcast`).

Instead of connecting remotely, there is a `TestControl` object to test the functionality of the `Network` and `Client` classes by simulating a remote connection. This class and the test functions are written for you. You will then be able to run various tests using the `TestControl` and `TestView` objects.

You will write these classes using a UML diagram for guidance.

4 UML Diagram



5 Classes Overview

This application will consist of 7 classes. In addition to the classes shown in the diagram above, there is a `PodArray` class. The classes are listed below along with their respective categories. You should use the instructions and the UML diagram to construct your app.

1. The `Subscriber` class (Entity object):
   (a) Contains information about the Subscriber
2. The `Episode` class (Entity object):
   (a) Contains information about the Episode
   (b) Plays content through a View object, i.e., `std::cout`
3. The `Podcast` class (Entity object):
   (a) Contains information about the Podcast
   (b) Maintains a collection of `Episodes`
4. The `PodArray` class (Collection object):
   (a) Data structure for `Podcasts`.
5. The `Network` class (Control/Boundary object):
   (a) Manages collections of `Podcasts` and `Subscribers`
   (b) Provides services to the `Client` (such as `download` and `stream`).
   (c) Prints error information to `std::cout`
6. The `Client` class (Control object):
   (a) Interacts with the `Network` to stream episodes or download podcasts
   (b) Manages a collection of downloaded `Podcasts`
7. The `TestControl` class (Control object):
   (a) Controls the running of tests on your application
   (b) Interacts with `TestView`
8. The `TestView` class (Boundary object):
   (a) Takes input from the user performing the tests

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

6 Instructions

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 (except the test classes) MUST HAVE A PRINT FUNCTION. This print function should display the metadata of the class using appropriate formatting.

6.1 The Subscriber Class

Implement the `Subscriber` class.

1. Member variables:
   (a) `name` and `creditcard` members, both strings.
2. Make a two argument constructor - `string name, string creditcard` that initializes the member variables.
3. Make a `bool matches(const string& name)` function that returns true if the `name` parameter matches the `Subscriber` name, and false otherwise.

6.2 The Episode Class

Implement the `Episode` class.

1. Member variables:
   (a) `name`, `content`, and `podcast` members, all strings.
   (b) an integer, `number`, which is the number of the episode.
2. Make a constructor that takes the arguments: `const string& podcast, int number, const string& name, const string& content`. Initialize the member variables appropriately.
3. Make a `play` method. This is similar to a `print` function except that you will first print the podcast information, then the episode name and number, and then the episode content to `std::cout`.

6.3 The Podcast Class

Implement a `Podcast` class.

1. Member variables:
   (a) `title` and `host` members, both strings.
   (b) a *dynamically allocated* primitive array of `Episode` pointers. Use the `NUM_EPS` definition from "defs.h" to initialize the size.
   (c) an integer, `numEps`, to track the number of episodes in the `Episode` array.
2. Make a constructor that takes two strings as arguments: `title` and `host`. Initialize all member variables appropriately.
3. Make a copy constructor. This should do a *deep copy* of all data. Pay specific attention to the `Episodes`.
4. Make getters for `title` and `host`. These will only be used to identity the `Podcasts`, not to make changes, so should return a **constant reference** to a `string`. Make a `getNumEpisodes()` function that returns the number of episodes stored.
5. Make a function `addEpisode` that takes two strings, `title` and `content` as arguments. If the `Episode` array is full, return false. Otherwise make a new `Episode` and add it to the back of the `Episode` array. For simplicity, the episode number should be the location of the `Episode` in the array. That means the first `Episode` will be number 0.
6. Make a function `bool getEpisode(int index, Episode** ep)`. If `index` is a valid index, retrieve the episode at that location in the `Episode` array and assign it to `ep`. Return true. If `index` is not valid, return false.
7. Make a `bool lessThan(Podcast& pod)` function. This function returns true if `this->title` comes before `pod.title` in alphabetical order, and false otherwise.

6.4 The PodArray Class

Most of this class is provided for you. Finish the destructor and the `removePodcast` and `getPodcast` implementations. Although the podcasts are stored in alphabetical order, you may use a linear search to find them (that is, simply check every location in order). The `removePodcast` function should close any gaps in the `podcasts` array caused by removing the `Podcast`. It should also return the removed `Podcast`. Although the `Podcasts` are created in the `Network` class, the `PodArray` destructor should delete every `Podcast`. In other words, the `Network` is passing the responsibility for the deletion of `Podcasts` to the `PodArray`.

6.5 The Network Class - Updated Feb 9

Make a `Network` class. Refer to the UML diagram for details and complete function signatures.

1. Member variables:
   (a) a `string` for the name of the `Network`
   (b) a `PodArray` pointer.
   (c) a *statically allocated* primitive array of `Subscriber` pointers. Use `MAX_SUBS` from `defs.h` for the size.
   (d) an integer `numSubs` that keeps track of the current number of `Subscribers` in the `Subscriber` array.
2. The constructor should initialize all member variables appropriately.
3. The destructor should deallocate all the necessary member variables.
4. **Updated Feb 9:** Make a `getPodcast` function. The `getSubscriber` function has been deprecated in favour of `hasSubscriber`, so you may make a `getSubscriber` function or not.
5. `bool addPodcast`: This function should create a new `Podcast` object and add it to the `PodArray` if there is room. If successful, return true. If the podcast cannot be added because the `PodArray` is full return false.
6. `bool removePodcast`: Remove the podcast with title `podcast` from the `PodArray`. Make sure to properly manage the memory (i.e., delete the `Podcast`).
7. `bool addEpisode`: If the `Podcast` exists attempt to add a new `Episode` to it. Return true if successful, false otherwise.
8. `bool addSubscriber`: If the `Subscriber` array is full, return false. Otherwise make a new `Subscriber` object and add it to the back of the array.
9. `bool download`: If the given `Subscriber` and the `Podcast` exist, then assign the `Podcast` to the output parameter and return true. We are not copying the `Podcast` here, only returning it. Otherwise output an error message to `std::cout` with details of what went wrong (for example, "no such subscriber") and return false.
10. `bool stream`: If the `Subscriber`, `Podcast`, and episode number (`epNum`) exist, then assign that `Episode` to the output parameter and return true. Otherwise output an error message to `std::cout` with details of what went wrong (for example, "no such subscriber") and return false.
11. `bool hasSubscriber`: If the `Subscriber` with name matching the input parameter exists, return true, otherwise return false.

6.6 The Client Class

Make a `Client` class. Refer to the UML diagram for detail and complete function signatures.

1. In addition to the member variable listed in the UML diagram, this class should have a `PodArray` pointer.
2. Make a constructor which initializes member variables appropriately.
3. Make a destructor which deletes member variables appropriately.
4. `download`: Attempt to download the podcast with the name `podcast` from the `Network`. If successful, and there is room in the `PodArray`, make a copy of the `Podcast` and add it to the `PodArray`. If unsuccessful, output an error message to `std::cout`.
5. `stream`: Attempt to stream the given podcast episode from the `Network`. If successful, `play` the `Episode`. You do not need an error message here since the `Network` should produce one if something goes wrong.
6. `playLocal`: Attempt to retrieve the podcast episode from the `PodArray` member variable. If successful, `play` the `Episode`. If unsuccessful, output an error message to `std::cout`.

6.7 The TestControl and TestView Classes

These classes have been done for you. They work as follows. The `launch` function in the `TestControl` class instantiates and displays a `TestView` object to gather user input. Based on the input, it calls one of 5 private test functions from the `TestControl` class. This repeats until the user selects 0, at which point the program exits.

6.8 The `main` Function

This has also been provided for you. It instantiates a `TestControl` object and calls `launch`.

7 Constraints

Your program must comply with all of the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

1. 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.
2. Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*
3. Your program must be written in Object-Oriented C++. To wit:
   (a) Do not use any global functions or variables other than `main`.
   (b) Do not use `struct`s, use classes.
   (c) Do not pass *objects* by value. Pass by reference or by pointer.
   (d) Except for simple getters or error signalling, data must be returned from functions using output parameters.
   (e) Reuse existing functions wherever possible. If you have large sections of duplicate code, consider consolidating it.
   (f) Basic error checking must be performed.

(g) All dynamically allocated memory must be deallocated. Every time you use the `new` keyword to allocate memory, you should know exactly when and where this memory gets `delete`d. It is recommended you verify proper memory management using `valgrind`.

4. All classes should be reasonably documented (remember the best documentation is expressive variable and function names, and clear purposes for each class).

8 Grading

8.1 Marking Components

1. 5 marks: The `Subscriber` class
2. 10 marks: The `Episode` class
3. 14 marks: The `Podcast` class
4. 6 marks: The `PodArray` class
5. 20 marks: The `Network` class
6. 10 marks: The `Client` class

Total Marks: 65 marks

8.2 Execution and Testing Requirements

1. All marking components must be called and execute successfully to earn marks.
2. All data handled must be printed to the screen to earn marks (make sure `print` prints useful information, such as the object member variables, where appropriate).

8.3 Deductions

8.3.1 Packaging errors:

1. 10% : Missing Makefile
2. 5% : Missing README
3. up to 10%: Failure to separate code into header and source files.
4. up tp 10%: Readability - bad style, missing documentation.

8.3.2 Major design and programming errors:

1. 50%: marking component that uses global variables or `struct`s.
2. 50%: marking component that consistently fails to use correct design principles.
3. 50%: marking component that uses prohibited library classes or functions.
4. up to 10%: memory leaks reported by `valgrind`.

8.3.3 Execution errors:

1. 100% of any marking component that cannot be tested because it doesn't compile or execute in the course VM, or the feature is not used in the code, or data cannot be printed to the screen. In short: your program must convince, without modifcation, myself or the TA that it works and works properly. TAs are not required to debug or fix non-working code. Working code that does not exactly match the specification is preferable and will get you more marks than non-working code.