IMPORTANT UPDATE NOV 24:

Search classes: For the assignment, all `Search` class headers must be in a file `Search.h`, and all `Search` class implementations must be in a single file `Search.cc`.

That is, you will declare `Search`, `H_Search`, `C_Search`, and `HorC_Search` in the same file `Search.h`.

Likewise implementations for `H_Search`, `C_Search`, and `HorC_Search`, and the `<<` operator, must be in a file `Search.cc`.

For the tutorials you may do whatever you wish.

I am using my own `Makefile` on Gradescope for this assignment, which requires everyone to use the same file conventions. Thank you.

I plan on opening the A4 Gradescope server sometime next week. There is still more testing to be done.

Update Nov 23:
5.3(g): add an `add(Episode*)` function to `Podcast` class.
5.7(e): add a `getEpisodes` function to `Podify` class.
Thank you to the students who found these omissions.

1 Submission Instructions

1. You should submit your assignment to **Gradescope** (*NOT Brightspace*). This assignment will be completely autograded, which will allow us to have a later deadline. This is advantageous in some ways, but if your program crashes it can be disastrous. You should comment out code that crashes. There is 1 mark given automatically for every test that doesn't crash.

2 Learning Outcomes

In this assignment you will learn to

1. Use operator overloading (with polymorphism).
2. Use templates to achieve data abstraction.
3. Use multiple inheritance in a diamond hierarchy.
4. Use Factories to separate object creation from application logic.
5. Use a strategy pattern to change behaviour at runtime.

3 Overview

In this assignment we will implement a podcast network called `Podify`. The `Podify` network holds `Podcasts` which have `Episodes`. You will implement a `Search` object which will match `Episodes` that meet certain criteria and put them in a playlist. You can then list all `Episodes` in the playlist, or play all `Episodes` in the playlist. When you play `Episodes`, you have the option of playing with or without a video element (which is some random ASCII art). `Podcasts`, `Episodes`, and `Search` objects will be made using a `PodcastFactory`. Though the `PodcastFactory` will create these objects, you must delete them when you are finished with them.

4 Classes Overview

This application will consist of 10 base classes and 5 derived classes (of the `Search` and `PodcastPlayer` classes).

1. `Podcast` (Entity / Container object):
   (a) Contains `Podcast` meta-data as well as a collection of `Episode`s.
2. `Episode` (Entity):
   (a) Contains `Episode` meta-data and content.
3. `Search` (Behaviour object): Virtual base class of all `Search` objects.
   (a) `H_Search` (Behaviour object): Matches `Episodes` by the `Podcast` `host`.
   (b) `C_Search` (Behaviour object): Matches `Episodes` by the `Podcast` `category`.
   (c) `HorC_Search` (Behaviour object): Matches `Episodes` by the `Podcast` `host` or `category`.
4. `Array` (Container object):
   (a) A simple (templated) data structure.
5. `Podify` (Control object):
   (a) Stores `Podcasts` along with their `Episodes`, allows them to be queried.
6. `PodcastPlayer` (Behaviour object): Determines how to play an `Episode` (either "audio" only, or "audio and video", both of which are simply text).
   (a) `AudioPlayer`: Displays the `audio` of an `Episode`.
   (b) `VideoPlayer`: Displays the `audio` of an `Episode` and some ASCII art which is loaded from a file at runtime.
7. `View` (View object):
   (a) Collects user input and displays `Podcasts`, `Episodes` and other information.
8. `Control` (Control object):
   (a) Manages the interactions of `Podify`, `PodcastFactory`, and `View`. Stores the `Episode` playlist.
9. `TestControl` (Control object):
   (a) Test framework for `Podify`.
10. `Tester`:
    (a) Provides some common test functionality.

5 Instructions

Download the starting code from Brightspace. It includes some starter code as well as testing code. 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). Unless otherwise specified ALL CLASSES SHOULD OVERRIDE THE << OPERATOR. This function should display the metadata of the class using appropriate formatting.

Your finished code should compile into two executables called `a4` and `test` using the command `make all` or simply `make`. If you zip your submission, then it **MUST BE** in a single folder `assignment4` containing all your files. You may also drag and drop an unzipped folder into Gradescope, in which case it can be named whatever. In both cases you **MUST** maintain the original file structure.

5.1 The Search Classes

The `Search` class is an abstract class with 3 concrete derived classes.

5.1.1 The Search Abstract Class

1. Pure virtual member functions:
   (a) `bool matches(const Episode*)`.
   (b) `print(ostream& ost)`.
2. Implement the stream insertion operator `<<` **using polymorphism**.

5.1.2 The H_Search Class

Make a class `H_Search` that is a derived class of `Search` using virtual inheritance. H stands for "host".

1. Make a member variable `string host`.
2. Have the constructor take a string and initialize the `host` member.
3. Override the `matches` function so that it returns `true` if the host of the given `Episode` is the same as the `host` member variable, and `false` otherwise.
4. Override the `print` function and explain this matches an Episode with the given host. Be sure to print out the `host` variable.

5.1.3 The C_Search Class

Make a class `C_Search` that is a derived class of `Search` using virtual inheritance. C stands for "category".

1. Make a member variable `string category`.
2.
3. Have the constructor take a string and initialize the `category` member.
4. Override the `matches` function so that it returns `true` if the category of the given `Episode` is the same as the `category` member variable, and `false` otherwise.
5. Override the `print` function and explain this matches an Episode with the given category. Be sure to print out the `category` variable.

5.1.4 The HorC_Search Class

Make a class `HorC_Search` that is a derived class of both `H_Search` and `C_Search`.

1. Have the constructor take a host and category in that order. Initialize all members.
2. Override the `matches` function so that it returns `true` if the category OR the host of the given `Episode` is the same as the `category` or `host` member variable, and `false` otherwise. For example, if this is a search for host "Joe" in the "Marijuana" category, then any `Episode` with Joe as host or Marijuana as category would cause this function to return true. If the host is not Joe **and** the category is not Marijuana, it would return false.
3. Override the `print` function and explain this matches an Episode with the given host or category. Be sure to print out the `host` and `category` variables.

5.2 The Array Class

The array class is provided for you, but it currently only works for `strings`. Make the following modifications. Also note that both the class definition and implementation are in the same .h file, and since we are making it into a template, you should leave it that way.

1. Convert the `Array` class to a Template class so that it may store any type.
2. Change the `add` function so that it instead overloads the `+=` operator. It should work identically to `add`.
3. Change the `remove` function so that it instead overloads the `-=` operator. It should work identically to `remove`.
4. Change the `get` method into a method that overloads the `[]` operator. Make two versions of this function, one `const` and one not `const`. If the supplied index is out of bounds, print an error message to `cerr` and call `exit(1)` (which will exit the program).

5.3 The Podcast Class

1. Member variables:
   (a) `Array<Episode*>`.
   (b) `string title, host.`
2. Member functions:
   (a) Make a constructor that takes a `string title` and `host` as arguments (in that order) and initializes the name member variable.
   (b) Make a function `bool equals(const string& title)` that returns `true` if the title argument matches the title member.
   (c) Make an `Episode* get(int index)` function that returns the `Episode` at the given index. If the index is out of bounds, call exit(1).
   (d) Make a `getSize()` function that returns the number of `Episodes` in this `Podcast`.
   (e) Make a `print(ostream&)` function that prints the title and host.
   (f) Make a `printWithEpisodes(ostream&)` function that prints the title and host and then prints all the `Episodes`.
   (g) Update Nov 23: Make an `add(Episode*)` function that adds this `Episode` to the `Array<Episode*>`.
3. Overload the stream insertion operator `<<` to call the `print` function.

5.4 The Episode Class

This class will be very straightforward.

1. Member variables (all strings):
   (a) `podcastTitle, host, episodeTitle, category, audio, videoFile`.
2. Member functions:
   (a) Make a constructor that takes enough strings to initialize all members in the order that they are listed.
   (b) Make `getters` for each member (we will eschew fancy design patterns here in favour of simplicity).
   (c) Make a `print(ostream&)` function that prints the podcast title, host, category, and episode title.
3. Overload the stream insertion operator `<<` to call the `print` function.

5.5 The PodcastPlayer Classes

The `PodcastPlayer` class is an interface with a single pure virtual member function. There are two concrete subclasses.

1. Pure virtual function: `play(const Episode& m, ostream& ost)`
2. The `AudioPlayer` class should be a derived class of `PodcastPlayer`. It should override the play function so that it outputs the `Episode audio` member to the given `ostream`.
3. The `VideoPlayer` class should be a derived class of `AudioPlayer`. It should override the play function so that it outputs the `Episode audio` member to the given `ostream`. In addition, it should open a file using the `videoFile` member and read that file line by line and output it to the given `ostream`. Don't forget to close the file.

5.6 The PodcastFactory Class

This class is done for you.

5.7 The Podify Class

1. Member variables:
   (a) `Array<Podcast*>`.
2. Member functions:
   (a) `addPodcast`: This should take a `Podcast*` argument. Add it to the collection of `Podcasts`.
   (b) `addEpisode`: This should take an `Episode*` and `string podcastTitle` as arguments. If the given `Podcast` exists, add the `Episode`. Otherwise output an error message.
   (c) Make a `getPodcasts` function that returns a const reference to the `Array` of `Podcasts`.
   (d) Make two `getPodcast` functions. Both should return a `Podcast` pointer. One should take an `int index` as an argument, and return the `Podcast` at that index. The other should take a `string title` as argument and return the Podcast with the given title. If the `Podcast` does not exist give an error message and call `exit(1)`. (The test files should always supply legitimate values, so if your application crashes, there is a bug somewhere, probably in your code, but at least you will get an error message.)
   (e) Update Nov 23: Make a `getEpisodes` function. It should take a `Search` object as the first argument and an `Array<Episode*>` as the second argument. The `Array<Episode*>` is an output parameter. You should find all the `Episodes` from every `Podcast` in `Podify` that matches the `Search` parameter and add them to this `Array`.

5.8 The View Class

The `View` class is ALMOST done for you. You should add an `AudioPlayer` and `VideoPlayer` as member variables, and set the `player` member to the `AudioPlayer` in the constructor. Then complete the `toggleVideo` function so that `player` points to either the `AudioPlayer` or `VideoPlayer` as appropriate. If you only complete the `AudioPlayer`, then you may leave `toggleVideo` blank.

5.9 Controller, TestControl, Tester

The `Controller` and `TestControl` classes are done for you. Feel free to add print statements for testing purposes, etc.

The `Tester` class is also there. You should do not need to adjust this class.

When you submit your assignment, new `Tester` and `TestControl` classes will be copied in with your submission. They will be functionally the same as what you've been given, but with some extra code that will communicate with the Gradescope marking script to output your mark.

5.10 main.cc and test.cc

These have been provided for you.

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 a working application. These are the only marks in this case, though deductions may still be applied. These are the same marks shown in the test suite, repeated here for convenience. Marks are awarded for the application working as requested.

The test script provides a mark out of 22. We reserve the right to modify the mark given by the test script in the event that your code has errors or uses forbidden libraries.

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 (make sure `print`, `play`, and the stream insertion operator << print useful information).

6.1.1 Application Requirements: 22 marks

Important: To get the marks for application requirements, you had to have implemented these classes with the `Array<T>` class rather than the data structures from Assignment 2 or 3.
If you use the data structures from Assignment 2 or 3 then you get 0.
Each test gives 1 mark automatically if your code does not crash.

The marks below are based on the correct execution of each of the menu items.

- 3 marks: Test add and print Podcasts.
- 3 marks: Test add and print Episodes
- 5 marks: Test get Episodes by host, and print episodes from View class.
- 3 marks: Test get Episodes by category.
- 5 marks: Test get Episodes by host or category
- 3 marks: Test play current Episode list with video.

Requirements Total: 22 marks

6.2 Constraints

The previous section awards marks if your program works correctly. No marks are given in this section, though the test script may implicitly test for things like const parameters. 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.
- Proper error checking - check array bounds, data in the correct range, etc. Call exit(1) and crash gracefully where appropriate.
- 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).

Constraints Total: 0 marks

Requirements Total: 22 marks

Assignment Total: 22 marks

6.3 Deductions

The requirements listed here represent possible deductions from your assignment total. If your code crashes you will not get any marks for that part. 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. This is not a comprehensive list. Any requirement specified during class but not listed here must also be observed.

6.4 Memory Leaks

1. **Up to 5 marks:** Memory leaks - be sure to check your code with valgrind. The marking script on Gradescope *will* check for memory leaks.

6.4.1 Packaging and file errors:

1. 5%: Missing README - if I have to look at your code for any reason, I should find a README file.
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.

6.4.2 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 5 marks: Using smart pointers. You must manage your own memory.
- 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.

6.4.3 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.*