Update Feb 6: added clarification to 6.6.6 - add a Photo object (by reference) as second parameter, not a Photo pointer.
Feb 7 - Album class in UML diagram, changed artist:string to description:string

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. Write an application where we begin to separate into control, view, entity, and collection object classes.
2. Take a first step towards data abstraction instead of raw arrays.
3. Use a UML diagram to implement classes and the interaction between between classes.
4. Implement proper memory management when using dynamic memory.
5. Implement proper encapsulation (using the `const` keyword where appropriate)
   VERY IMPORTANT!!! YOU WILL LOSE MARKS IF YOU DO NOT CONST YOUR FUNCTIONS AND PARAMETERS!!!.

3 Overview

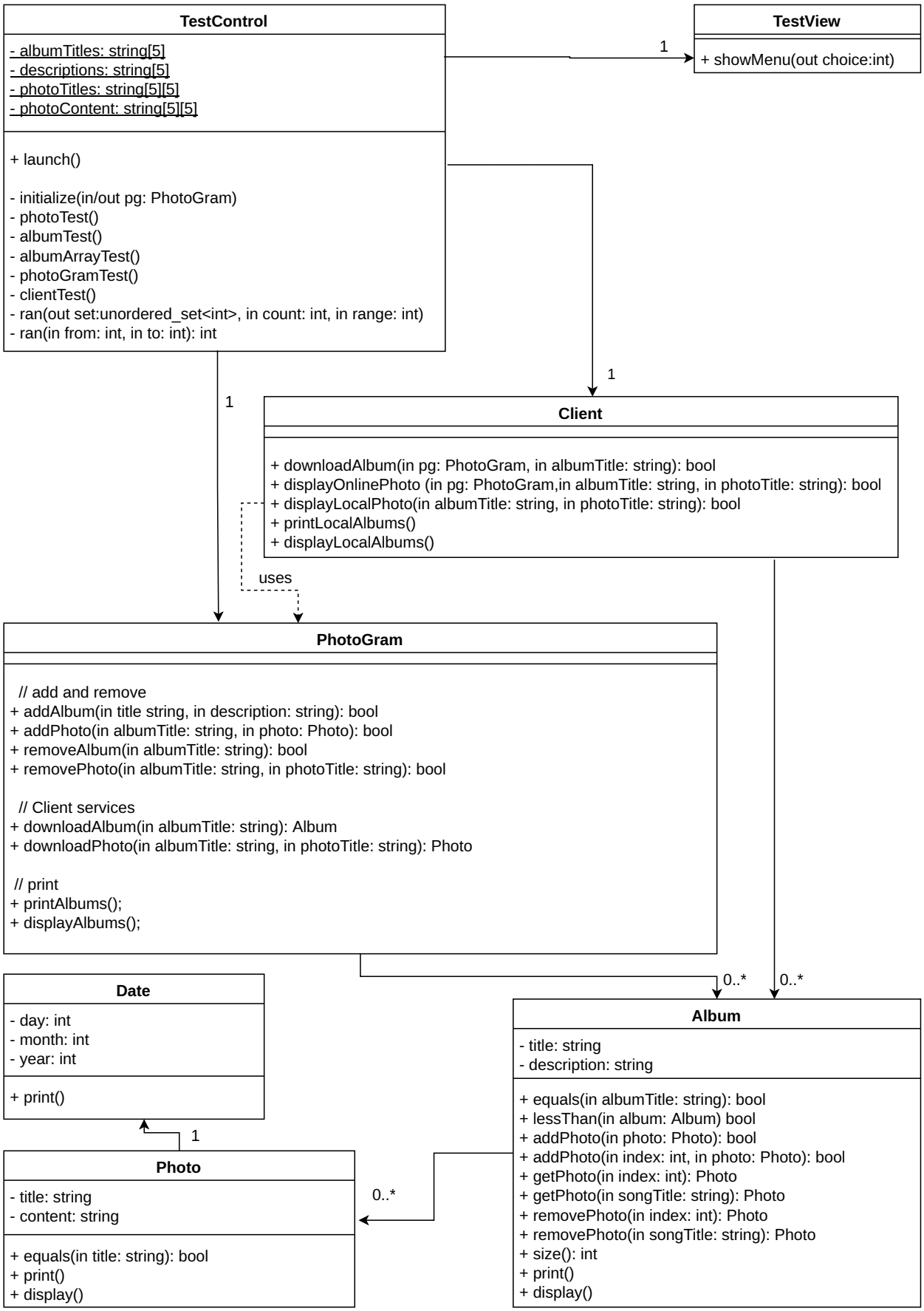
You will be writing C++ code that mimics a photo viewing application (called `PhotoGram`). Any `Photo` on `PhotoGram` must belong to an `Album`. Thus `Albums` are responsible for the memory management of `Photos` and `PhotoGram` is responsible for the memory management of `Albums`. `Albums` have a title and a description and a data structure for storing `Photos`. Each `Photo` will have a title, `Date` taken, and content (which are the pictures themselves). `Photos` may be `print`ed (which is to have their metadata displayed without the content) or `display`ed, which is to print their metadata and `content` together to the console.

`PhotoGram` will consist of 0 or more `Album`s. A `Client` class will be able connect to `PhotoGram` where they can display `Photos` from any `Album` stored on `PhotoGram`. In addition, the `Client` will be able to "download" `Albums`. This copies the `Album` to "local storage", which, in this exercise, is a data structure in the `Client` class. Users can then display the `Album` locally, which should work even if the `PhotoGram` network deletes the original `Album`. In other words, `download`ing consists of making a *deep copy* of the `Album`.

There is a `TestControl` class that connects and test the functionality of the `PhotoGram` and `Client` classes. 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.

To assist you in writing these classes a UML diagram is provided.

4 UML Diagram



5 Classes Overview

This application will consist of 9 classes. In addition to the classes shown in the UML diagram above, there are `AlbumArray` and `PhotoArray` classes. All classes are listed below along with their respective categories. You should refer the instructions and the UML diagram to construct your app.

1. The `Date` class (Entity object):
   (a) Contains date information.
2. The `Photo` class (Entity object):
   (a) Contains information about the photo
   (b) Displays content through a View object, which in this case is `std::cout`
   (c) **Note:** In the interests of space, and since `Date` is given to you, the `Date` UML diagram is incomplete.
3. The `Album` class (Entity object):
   (a) Contains information about the album
   (b) Manages a collection of `Photos`
4. The `PhotoArray` class (Collection object):
   (a) Data structure for `Photos`.
5. The `AlbumArray` class (Collection object):
   (a) Data structure for `Albums`.
6. The `PhotoGram` class (Control object):
   (a) Manages a collection of `Albums`.
   (b) Provides services to the `Client` (such as `display` and `download`).
   (c) Prints error messages to `std::cout`
7. The `Client` class (Control object):
   (a) "Connects" to the `PhotoGram` network to display `Photos` or download `Albums`.
   (b) Manages a collection of downloaded `Albums`.
   (c) Prints error messages when appropriate.
8. The `TestControl` class (Control object):
   (a) Controls the running of tests on your application
   (b) Interacts with `TestView`
9. 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

Download the starting code from Brightspace. It includes some global functions that you are to use for testing as well as the `Date` class. 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 (except for `PhotoArray` and `AlbumArray`). This print function should display the metadata of the class using appropriate formatting.

Your finished code should compile into a single executable called `a2` using the command `make all` or simply `make`. Your submission should consist of a single zip file with a suitable name (e.g., `assignment2.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.

6.1 The Date Class

✓ This class is provided for you. You should apply `const` wherever appropriate.

6.2 The Photo Class

Implement the `Photo` class.

1. Member variables:
   - ✓ (a) `title` and `content` members, both strings.
   - ✓ (b) A `Date` object.
2. ✓ Make a constructor that takes the arguments: `const string& title, const Date& date, const string& content`. Initialize the member variables appropriately.
3. ✓ Make a function `equals(title)` that returns `true` if the `title` parameter is equal to the `title` member variable of this `Photo`, and `false` otherwise.
4. ✓ Make a `print` function. This should print out the `Photo` metadata (not including the `content`).
5. ✓ Make a `display` function. This is similar to a `print` function except that after printing the metadata you should output the `content`.

6.3 The PhotoArray Class

You may use the `Array` class seen in class (Chapter 8: Object Design Categories) as a starting point. Also note that we want to be able to specify the order that `Photos` are stored in. Thus we will overload the `add`, `get`, and `remove` functions to use indexes in addition to their canonical form.

1. Member variables:
   - ✓ (a) The `PhotoArray` class should use a *dynamically allocated array* of `Photo` pointers as a backing array. Be sure to keep track of the number of `Photos` currently stored.
2. Member functions. Note: there are two each of `get`, `add`, and `remove`:
   - ✓ (a) Make an `isFull` function that returns `true` if the array is full and `false` otherwise.
   - ✓ (b) Make an `add` function that takes a `Photo` pointer as an argument and adds it to the back of the array. Return `true` if successful and `false` if the array is full.

- ✓ (c) Make an `add` function that takes an `index` and a `Photo` pointer as arguments and attempts to add the `Photo` to the array at the `index` indicated. Return `false` if the array is full or the index is invalid. Return `true` if successful.
- ✓ (d) Make a `get` function that takes a `title` as an argument and returns (using a return value) the pointer to the `Photo` with that title. If there is no `Photo` with the given title, return `NULL`.
- ✓ (e) Make a `get` function that takes an `index` as an argument and returns the `Photo` pointer at the index given, or NULL if the index is invalid.
- ✓ (f) Make a `remove` function that takes a `title` as an argument. It should removes the `Photo` pointer from the array and return it (using a return value). If there is no `Photo` with the given title return `NULL`.
- ✓ (g) Make a `remove` function that takes an `index` as an argument. If the index is valid return the `Photo` pointer at that index, otherwise return `NULL`.
- ✓ (h) Make a `size` function returns the number of `Photos` in the array as a return value.

6.4 The Album Class

Implement an `Album` class. This is a light wrapper for a `PhotoArray`. As such it duplicates a lot of `PhotoArray` functions, but adds a few members.

✓ 1. Member variables:

   (a) `title` and `description` members, both strings.

   (b) a dynamically allocated `PhotoArray`.

✓ 2. Make a constructor that takes two strings as arguments: `title` and `description`, in that order. Initialize all member variables appropriately.

✓ 3. Make a copy constructor. This should do a *deep copy* of all data.

✓ 4. Make a destructor. Make sure all dynamically allocated memory reachable by this class is deleted.

5. Member functions. Note that many of the `PhotoArray` functions are duplicated. Be lazy. Whenever possible, make `PhotoArray` do the work:

   ✓ (a) Make a getter for `title`.

   ✓ (b) Make a function `equals` that takes a title as an argument and returns `true` if the `Album` title matches the title parameter and `false` otherwise.

   ✓ (c) Make a function `lessThan` that takes an `Album& alb` as an argument and returns `true` if `this Album` is less than `alb` in alphabetical order by `title`, and `false` otherwise.

   ✓ (d) Make an `add` function that takes a `Photo` pointer as an argument. Attempt to add it to the `PhotoArray`. Return `true` if successful and `false` if the array is full.

   ✓ (e) Make an `add` function that takes an `index` and a `Photo` pointer as arguments. Attempt to add the `Photo` to the `PhotoArray` at the `index` indicated. Return `false` if the array is full or the index is invalid. Return `true` if successful.

   ✓ (f) Make a `get` function that takes a `title` as an argument and returns the `Photo` pointer as a return value if the Photo exists and `NULL` otherwise.

   ✓ (g) Make a `get` function that takes an `index` as an argument and returns the `Photo` pointer as a return value if the index is valid and `NULL` otherwise.

   ✓ (h) Make a `remove` function that takes a `title` as an argument, removes and returns the `Photo` pointer as a return value if the `Photo` exists and `NULL` otherwise.

- ✓ (i) Make a `remove` function that takes an `index` as an argument and removes and returns the `Photo` pointer as a return value if the `Photo` exists and `NULL` otherwise.
- ✓ (j) Have the `print` function print out all the metadata for this `Album`.
- ✓ (k) Make a `display` function that prints the metadata for this `Album` and `displays` all the `Photos`.

6.5 The AlbumArray Class

Implement an `AlbumArray` class. You can reuse (copy and paste) any appropriate code from the `PhotoArray` class. (Copying and pasting code is generally speaking not good software engineering, though it presents the best solution at this time - we will see better solutions once we learn *templates*.)

1. Member variables:

   - ✓ (a) The `AlbumArray` class should use a dynamically allocated array of `Album` pointers as a backing array. Be sure to keep track of the number of `Albums` currently stored.

2. Member functions. Note: there are two each of `get`, and `remove`, similar to `PhotoArray`. However `AlbumArray` has a single `add` function:

   - ✓ (a) Make an `add` function that takes an `Album` pointer as an argument and adds it adds in order as defined by `Album::lessThan(Album&)`. Return `true` if successful and `false` if the array is full. 2 Make an `isFull` function that returns `true` if the `AlbumArray` is full and `false` otherwise.
   - ✓ (b) Make a `get` function that takes a `title` as an argument and returns (using a return value) the pointer to the `Album` with that title. If there is no `Album` with the given title, return `NULL`.
   - ✓ (c) Make a `get` function that takes an `index` as an argument and returns the `Album` pointer at the index given, or NULL if the index is invalid.
   - ✓ (d) Make a `remove` function that takes a `title` as an argument. It should removes the `Album` pointer from the array and return it (using a return value). If there is no `Album` with the given title return `NULL`.
   - ✓ (e) Make a `remove` function that takes an `index` as an argument. If the index is valid return the `Album` pointer at that index, otherwise return `NULL`.
   - ✓ (f) Make a `size` function returns the number of `Albums` in the array as a return value.

6.6 The PhotoGram Class

Make a `PhotoGram` class. Refer to the UML diagram for details and complete function signatures. For this class, when an operation fails (such as `addAlbum` or `removeAlbum`), be sure to send an appropriate error message to `cout`.

✓ 1. Member variables:

   (a) an `AlbumArray` pointer.

✓ 2. The constructor should initialize all member variables appropriately.

✓ 3. The destructor should delete all dynamically allocated memory reachable by this class.

✓ 4. `addAlbum`: If there is room in the `AlbumArray` create a new `Album` and add it to the `AlbumArray` and return `true`. If the `AlbumArray` is full return `false`.

✓ 5. `removeAlbum`: If there is an `Album` that matches the arguments remove it from the `AlbumArray` and return `true`. If there is no such `Album` return `false`. Make sure to properly manage the memory (i.e., delete the `Album`).

6. `addPhoto`: If we successfully add the `Photo` to the given `Album`, return `true`, otherwise return `false`. Update Feb 6: Please add a Photo object (by reference) for the second parameter, not a Photo pointer.
7. `removePhoto`: If there is a `Photo` with the given `title` in the given `Album`, remove the `Photo` and return `true`. Otherwise return `false`. **Note:** Be sure to properly manage your memory.
8. `downloadAlbum`: Returns an `Album` pointer if the `Album` exists, returns `NULL` otherwise.
9. `downloadPhoto`: Returns a `Photo` pointer if the `Photo` exists, returns `NULL` otherwise.
10. `printAlbums`: This should print all `Albums` stored in `PhotoGram`.
11. `displayAlbums`: This should `display` every `Album` stored in `PhotoGram`.

6.7 The Client Class

Make a `Client` class. Refer to the UML diagram for detail and complete function signatures. As in the `PhotoGram` class, when an operation fails, be sure to send an appropriate error message to `cout` (unless there is already an error message being displayed from `PhotoGram`).

1. This class should have a `AlbumArray` pointer.
2. Make a constructor which initializes member variables appropriately.
3. Make a destructor which deletes member variables appropriately, i.e., delete all dynamic memory reachable from this class.
4. `downloadAlbum`: Attempt to download an `Album` from `PhotoGram` with the given title. If successful, and there is room in the `AlbumArray`, make a copy of the `Album` and add it to the `AlbumArray` and return `true`. If unsuccessful return `false`.
5. `displayOnlinePhoto`: If `PhotoGram` contains an `Album` with the title given, and this `Album` has a `Photo` with the title given, then display this `Photo`. If successful return `true`. If unsuccessful return `false`.
6. `displayLocalPhoto`: If the `Client` contains an `Album` with the title given, and this `Album` has a `Photo` with the title given, then display this `Photo`. If successful return `true`. If unsuccessful return `false`.
7. `printLocalAlbums`: `print` every `Album` stored in the `Client`.
8. `displayLocalAlbums`: `display` every `Album` stored in the `Client`.

6.8 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.

6.9 The `main` Function

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

7 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.

7.1 Specification Requirements

These are marks for having a working application (even when not implemented according to the specification, within reason). The test suite will automatically allocate some marks. Other marks are designated *manual marks*, or you may be asked to *assign marks manually*. These places are indicated in the output by the string `**MANUAL MARK**`. In these cases you must visually inspect the output to determine if the code has run correctly, then assign the necessary marks.

You are still responsible for, and may be penalized for, any errors the test suite does not catch. This is especially important here, as the testing requirements for this application are quite rigorous, and there may be some additional cases added to the final test suite. Any drastic departure from the specification may still result in a penalty (such as using outside 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` and `display` all print useful information).

Application Requirements: 24 marks

- 2 marks: `Photo` works correctly.
- 5 marks: `Album` and `PhotoArray` work correctly.
- 5 marks: `AlbumArray` works correctly.
- 7 marks: `PhotoGram` works correctly.
- 5 marks: `Client` works correctly.

Requirements Total: 24 marks

7.2 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 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 *within reason*. There are times where duplicating tiny amounts of code makes for better efficiency.
- Proper error checking - check array bounds, data in the correct range, etc.
- Release all dynamically allocated memory - check for memory leaks using `Valgrind`.

7.2.1 Constraints: 18 marks

1. 2 marks: Proper `const`-ing of the `Date` class.
2. 2 marks: Proper implementation and `const`-ing of the `Photo` class.
3. 2 marks: Proper implementation and `const`-ing of the `PhotoArray` class.
4. 2 marks: Proper implementation and `const`-ing of the `Album` class.
5. 2 marks: Proper implementation and `const`-ing of the `AlbumArray` class.
6. 3 marks: Proper implementation and `const`-ing of the `PhotoGram` class.
7. 3 marks: Proper implementation and `const`-ing of the `Client` class.
8. 4 marks: Manage memory properly (no memory leaks)
   (a) You will lose 1 mark per test section that produces a memory leak or a double free error, to a maximum of 4.

Constraints Total: 20 marks

Requirements Total: 24 marks

Assignment Total: 44 marks

7.3 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.

7.3.1 Documentation and Style

1. Up to 10%: Improper indentation or other neglected programming conventions.
2. Up to 10%: Code that is disorganized and/or difficult to follow (use comments when necessary).

7.3.2 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.

7.3.3 Incorrect object-oriented programming techniques:

- Up to 10%: Substituting C functions where C++ functions exist (e.g. don't use `printf` or `scanf`, do use `cout` and `cin`).
- 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.

7.3.4 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.*