Update Mar. 21st Section 5.8.2 b - `PhotoGram::addToAlbum` function added.

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. Use operator overloading (with polymorphism).
2. Use templates to (finally) achieve data abstraction.
3. Use multiple inheritance in a diamond hierarchy.
4. Use Factories to separate object creation from application logic.

3 Overview

In this assignment we will use a similar framework to Assignment 2, the `PhotoGram` application. Our previous version was not entirely realistic (to put it lightly). In this version we will have a master list of `Photos`, and use these `Photos` to create `Albums`. A `Photo` can appear in more than one `Album`. Deleting an `Album` should no longer delete the `Photos` contained in them (in fact, in this version, to keep it simple, we will only delete `Photos` **when the application exits** - you still must have no memory leaks). Because we may add `Photo` and `Album` subclasses in the future, and because their creation can involve disk access (downloading from SD card or phone for example), we will use a `MediaFactory` to create these objects.

We are also implementing a simple search function for `Photos`. This will allow us to find `Photos` with certain `Criteria` and add them to an `Album`. We will use a `Criteria` class which encapsulates the search details and allow us to add future search `Criteria` without changing the core algorithm. We will also use a proper templated collection class that we will write ourselves.

4 Classes Overview

This application will consist of 9 base classes and 3 derived classes (of the `Criteria` class).

1. The `Date` class (Entity object):
   (a) Store information about Dates.
2. The `Array` class (Container object):
   (a) An abstract container for data with overloaded operators.

3. The `Photo` class (Entity object):
   (a) Contains information about a particular photo.
4. The `Album` class (Entity object):
   (a) Contains information about the album.
   (b) Maintains a collection of `Photos`.
5. The `Criteria` class (Entity object):
   (a) This class is used to match user searches to matching `Photo` objects. There are an additional three derived classes arranged in a diamond hierarchy, `Cat_Criteria` (category), `Date_Criteria`, and `CandD_Criteria` (both date and category).
6. The `MediaFactory` class (Boundary object):
   (a) This class is used to create `Photos`, `Albums`, and `Criteria` objects. It separates our application logic from the creation of `Photos`, `Albums`, or `Criteria` types.
7. The `PhotoGram` class (Container/Control object):
   (a) This class contains the master list of `Photos` and a collection of `Albums`. Users can add `Photos` to the master list, and add or remove `Albums`. In addition we can search for `Photos` and these `Photos` can be added to an `Album` (we will not worry about removing `Photos` from an `Album` in this assignment). In addition, in conjunction with the `AlbumCreator` and `View` objects, users can browse `Albums` and `Photos`.
8. The `AlbumCreator` class (Control object):
   (a) Controls the interaction of `PhotoGram` with the `View`.
   (b) Maintains its own list of `Photos` found during the latest search.
9. The `View` class (Boundary object):
   (a) Prints menus, `Albums`, or `Photos`, and takes input.

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). 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 a single executable called `a4` using the command `make a4` or simply `make`. Your submission should consist of a single zip file with a suitable name (e.g., `assignment4.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.

✓ 5.1 defs.h

You should initialize the arrays to the sizes given in this file.

5.2 The Date Class

Make the following changes to the `Date` class:

1. Overload the `<`, and `==` and `<=` operators, and any other comparison operators that you feel you will need.
2. Overload the stream insertion (`<<`) operator.

5.3 The Photo Class

Make the following changes to the `Photo` class:

1. Member variables:
   (a) Add a `string category` member variable.
2. In the constructor, add a `string` parameter for `category` just after the `title`. Thus your constructor should take the arguments in this order: `title, category, date, content.`
3. Make getters for `category` and `date`.
4. Have the `print` and `display` functions take an `ostream` as an argument and have them write to the `ostream` instead of `std::cout`.
5. Overload the stream insertion (`<<`) operator.

Your header member functions should look like the following:

```
Photo(const string&  title, const string& category, const Date&, const string& content);

bool  equals(const string& title) const;
const Date& getDate() const;
const string& getCategory() const;

void  print(ostream&) const;
void  display(ostream&) const;
```

5.4 The Criteria Classes

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

5.4.1 The Criteria Abstract Class

1. Make a pure virtual function `bool matches(const Photo&)`.
2. Make a pure virtual `print` function that takes an `ostream` as an argument.
3. Implement the stream insertion operator `<<` **using polymorphism**.

5.4.2 The Cat_Criteria Class

Make a class `Cat_Criteria` that is a derived class of `Criteria` using virtual inheritance.

1. Make a private member variable `string category`.
2. Have the constructor take a `string` argument and initialize the `category` member.
3. Override the `matches` function so that it returns true if the `Photo::category` matches the `category` member.
4. Override the `print` function so that it tells the user how this `Criteria` will match a `Photo`. Be sure to print out the `category` somewhere in there (we will be looking for it in the tutorial).

5.4.3 The Date_Criteria Class

Make a class `Date_Criteria` that is a derived class of `Criteria` using virtual inheritance.

1. Make two private member variables, a `start Date` and an `end Date`. This represents a range of `Dates`.
2. Have the constructor take two `Dates`, `start` and `end`, in that order.
3. Override the `matches` function so that it returns true if `Photo::date` is greater than or equal to the `start Date` and less than or equal to the `end Date`.
4. Override the `print` function so that it tells the user how this `Criteria` will match a `Photo`. Be sure to print out both `Dates` (we will be looking for it in the tutorial).

5.4.4 The CandD_Criteria Class

Make a class `CandD_Criteria` that is a derived class of both `Date_Criteria` and `Cat_Criteria`.

1. Have the constructor take a `Date start, Date end` and `string category` arguments, in that order, and initialize the members.
2. Override the `matches` function so that it returns true if the `Photo::category` matches the `category` member and `Photo::date` is greater than or equal to the `start Date` and less than or equal to the `end Date`
3. Override the `print` function so that it tells the user how this `Criteria` will match a `Photo`. Be sure to print out the `category` and both `Dates` somewhere in there (we will be looking for it in the tutorial).

5.5 The Array Class

The array class is provided for you, but it currently only works for `int`s. 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 `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).
3. Overload the `+=` operator so that you can add an element to `Array`. If there is room this should add an element *to the back* of the array. If there is no room do nothing. Be sure to enable *chaining*.
4. Overload the `-=` operator so that you can remove an element from the `Array`. If the element is found remove it. Be sure to enable *chaining*.
5. Add a `clear` function that removes everything from the `Array`.

5.6 The Album Class

The major changes to this class are

- Replacing `PhotoArray` with `Array<Photo*>`.
- We no longer specify the order of the `Photos` by adding at an index. The `Photos` will simply be stored in the order they are added (i.e., we add `Photos` to the back of the `Array`).
- You should not delete any `Photos` in this class. This is the responsibility of `PhotoGram` and the master list of `Photos`.

The changes outlined below will reflect that. Make the following changes to the `Album` class:

✓ 1. Member variables:

   (a) Replace `PhotoArray` with `Array<Photo*>`.

✓ 2. Remove the copy constructor.

✓ 3. Have the `bool addPhoto(Photo*)` function add the `Photo*` to the *back* of `Array`.

✓ 4. Remove the `bool addPhoto(int index, Photo*)` function.

✓ 5. Make an `int size()` function that returns the number of `Photos` in this `Album`.

✓ 6. Modify the `print()` and `display()` functions to accept an `ostream` argument. Have the data printed to the `ostream` argument instead of `std::cout`. As a reminder:

   ✓ (a) The `print` function should print metadata about the `Album` class, including the number of `Photos`.

   ✓ (b) The `display` function should print metadata about the `Album` class as well as `display` each `Photo`.

✓ 7. Overload the stream insertion operator `<<` to call the `print` function.

Your header member functions should look like the following:

```
Album(const string& title, const string& description);

bool   equals(const string& title) const;
void   addPhoto(Photo*);
Photo* getPhoto(int) const;
int    size()const;
void   print(ostream&) const;
void   display(ostream&) const;
```

5.7 The MediaFactory Class

The header for this class is provided. In addition the `uploadPhoto` implementation is provided.

This Factory class reflects that creating a `Photo` object is unlikely to be a simple call to the constructor. We may have to, for example, load an image file from disk to create it. Here we are using simulated image files (the `content` member) but the principle is the same.

A similar principle applies to the `Criteria` class. At some point we might want to introduce more search criteria.

✓ 1. `toDate` and `uploadPhoto` have been done for you. Complete the rest of the function implementations (they are as simple as they look).

5.8 The PhotoGram Class

The new specification for `PhotoGram` is as follows:

1. Member variables:
   - (a) Replace `AlbumArray` with `Array<Album*>`.
   - (b) Add a master list of `Photos` using `Array<Photo*>`.
   - (c) Add a `MediaFactory` - you will use this to create all objects used in `PhotoGram`.
2. Member functions:
   - (a) `addAlbum`: This should take a `title` and `description` as arguments. Use `MediaFactory` to make a new `Album` and add it to the collection of `Albums`.
   - (b) Update Mar 21: `addToAlbum`: This should take an `int index` and an `Array<Photo*>` as arguments. The `Photos` in the `Array` should be added to the `Album` located at the given index.
   - (c) `uploadPhoto`: This should take a `title` as an argument. Use `MediaFactory` to make a new `Photo` and add it to the list.
   - (d) `deleteAlbum(int index)`: Attempt to remove and delete the `Album` at the given index. If it is a bad index give an error message and return.
   - (e) `getPhotos`: This should take a `Criteria` object and an `Array<Photo*>` (passed by reference!). All `Photos` in the master list that match the `Criteria` should be added to the `Array<Photo*>`. Do not make copies - simply add their pointers to the `Array<Photo*>`.
3. The following functions will make use of the `View` class for printing and displaying `Albums` and `Photos`. Your functions should forward their calls to the similar `View` functions. This is known as double-dispatching, and it is a popular design pattern for drawing objects on a `View`:
   - (a) `void displayAlbum(int index, View&)`: display the `Album` at the given index using the `View`. Output an error message if the index is out of bounds.
   - (b) `int printAlbums(View&)`: Forward this call to the `View`. Return the number of `Albums` currently stored in `PhotoGram`.
   - (c) `void displayPhoto(int index, View&)`: display the `Photo` at the given index on the `View`. Output an error message if the index is out of bounds.
   - (d) `int printPhotos(View&)`: Forward this call to the `View`. Return the number of `Photos` currently stored in `PhotoGram`.

5.9 The View Class

The `View` class is done for you.

5.10 The AlbumCreator Class

Most of the `AlbumCreator` class is done for you.

1. Write the implementation of the `addAlbum` function.
2. Write the implementation of the `deleteAlbum` function. It should function similarly to the `displayAlbum` function in that it should print all `Albums`, prompt the user for their choice, then delete the `Album` at the index indicated. It should only be a few lines of code, and you should be calling existing functions wherever possible.

5.11 The `main` Function

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

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 (even when not implemented according to the specification, within reason). 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 20. If you have implemented everything correctly, this will *probably* be your mark. However, you are still responsible for, and may be penalized for, any errors the test suite does not catch, and drastic departure from the specification may still result in a penalty (such as using outside libraries). As such, 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`, `display`, and the stream insertion operator << print useful information).

6.1.1 Application Requirements: 26 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.
If you use the data structures from Assignment 2 then you get 0.

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

- 2 marks: Add `Album` works correctly.
- 2 marks: Delete `Album` works correctly.
- 2 marks: All `Albums` are printed correctly.
- 2 marks: A single `Album` is displayed correctly.
- 2 marks: All `Photos` are printed correctly.
- 2 marks: A single `Photo` is displayed correctly.
- 2 marks: Getting photos by `category` works correctly.
- 2 marks: Getting photos by `date` works correctly.
- 4 marks: Getting photos by `date` and `category` works correctly.
- 2 marks: Print current `Photo` list works correctly.
- 2 marks: Display current `Photo` list works correctly.
- 2 marks: Add current `Photo` list to `Album` works correctly.

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:

- 2 marks: Proper implementation of the `Date` class.
- 2 marks: Proper implementation of the `Array` class.
- 2 marks: Proper implementation of the `Photo` class.
- 2 marks: Proper implementation of the `Album` class.
- 2 marks: Proper implementation of the `PhotoGram` class.
- 2 marks: Proper implementation of the `MediaFactory` class.
- 2 marks: Proper implementation of the `AlbumCreator` class.
- 4 marks: Proper implementation of the `Criteria` classes.

Constraints Total: 18 marks

Requirements Total: 26 marks

Assignment Total: 44 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.*