COMP 2404 -- Assignment #4

<u>Due:</u> Tuesday, December 4, 2018 at 12:00 pm (noon)

Goal

You will modify your event management program from Assignment #3 to interact with a Façade class that acts as an interface to a simulated cloud-based storage service. You will also implement a class template and overloaded operators.

Learning Outcomes

With this assignment, you will:

- practice the integration of your code with an existing class
- work with the Façade design pattern
- implement a class template and overloaded operators in C++

Instructions

1. Prepare to integrate the new Façade class

We are going to change the program so that it can synchronize with a simulated cloud-based storage service. Our Façade class will be the `EventServer` class, which is provided for you. This class will simulate the retrieval of a master list of the user's events from cloud storage at the beginning of the program. The program will then run as usual, with the user adding new events. Then at the end of the program, the event server will simulate the sending all of the user's events (both the original ones and the new ones) back to the cloud storage, where they can be retrieved the next time the program runs. Of course, the `EventServer` class that you are given will only *simulate* this behaviour, so there is no actual cloud service. However, you will be modifying your program to integrate with the event server, as if it was working for real. You will need to download from *cuLearn* the `a4Posted.tar` file, which contains the `EventServer` class, as well as the `Array` class that it requires.

In order to work correctly, the `EventServer` and `Array` classes expect the following:

- your data classes must be named in accordance with previous assignment requirements (`Event`, `SchoolEvent`, `WorkEvent`)
- your `Event` class must provide a `print()` function, in accordance with previous assignments
- both `SchoolEvent` and `WorkEvent` constructors must have the following prototypes:
```
SchoolEvent(string name, int priority)
WorkEvent(string name, int priority)
```

2. Integrate the new Façade class

Most of the changes for this functionality will occur in the `Control` class. However, you will need to implement some utility functions in other classes first:

- You will add a `copy` member function to the `List` class. This function will have the prototype: `void copy(Array&)`. It will traverse the linked list and add every `Event` pointer that is currently in the list to the parameter array. Note that this will be a shallow copy. We are not creating new events! We are simply copying all the pointers from one container (the list) into another (the array).

- You will add a `copyEvents` member function to the `Calendar` class. The function will have the prototype: `void copyEvents(Array&)`. It will call its event list's `copy` function, which we implemented in the previous step.

Now we can make the changes to the `Control` class:

- Add an `EventServer` object as a data member of the `Control` class; you must declare this data member **after** the `Calendar` objects are declared in your `Control` class, otherwise your events will be destroyed before the `EventServer` is finished printing them
- Add a default constructor to the `Control` class that retrieves all of the user's events from the event server (and the simulated cloud storage) at the beginning of the program. The constructor will:
  - declare two local `Array` objects: one to hold the school events and one to hold the work events that are currently in cloud storage
  - call the event server's `retrieve` function with those two `Array` objects; this function will populate the arrays with the data from the cloud storage
  - loop over each array separately, and add each event currently in the array to the correct calendar (for example, the events in the school events array should be added to the school calendar)
- Add a destructor to the `Control` class that sends all the events from the school and work calendars to the cloud storage using the event server at the end of the program. The destructor will:
  - declare two local `Array` objects: one to hold the school events and one to hold the work events that will be sent to the cloud storage
  - use the `copyEvents` function of each calendar to copy the calendar's events to the corresponding `Array` object (for example, events from the school calendar will be copied to the school event array)
  - call the event server's `update` function with the two `Array` objects; this function will simulate the upload of the data to the cloud storage

The event server prints out everything from its master list at the end of the program. Check that all the events are printed out in the correct order.

3. Modify the List class as a class template

You will modify the `List` class to make it a class template. Note that the data will no longer be assumed to be a pointer! You should be able to use any data type in the list. You will also have to make the `Node` class a class template.

Change the `Calendar` class to use a templated list of `Event` pointers.

4. Add overloaded operators

You will modify the `Time`, `Date`, `Event`, `SchoolEvent`, and `WorkEvent` classes to replace all the `lessThan()` functions with the overloaded less-than (<) operator. Change all the classes that use this function so that they now use the new operator.

5. Test the program

- You will modify the `in.txt` file so that it provides sufficient datafill for a minimum of 15 events. The type of events, and the ordering of event dates, times, and priorities in the file must be such that the program is thoroughly tested.
- Check that the event information is correct when all the calendars are printed out at the end of the program. School events should be ordered by date and time, and work events by priority.
- Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

Constraints

- your program must follow correct encapsulation principles, including the separation of control, UI, and entity object functionality
- do not use any classes, containers, or algorithms from the C++ standard template library (STL)
- do not use any global variables or any global functions other than `main()`
- do not use structs; use classes instead
- objects must always be passed by reference, not by value
- your classes must be thoroughly documented in every class definition
- all basic error checking must be performed
- existing functions must be reused everywhere possible

Submission

You will submit in *cuLearn*, before the due date and time, the following:

- one **`tar`** or **`zip`** file that includes:
  - all source, header, and data files, including the code provided
  - a Makefile
  - a readme file that includes:
    - a preamble (program and revision authors, purpose, list of source/header/data files)
    - compilation. launching, and operating instructions

Grading (out of 100)

Marking components:

• 50 marks:	correct integration with Façade (`EventServer`) class	
	10 marks:	correct implementation of `copy` functions in `List` and `Calendar` classes
	21 marks:	correct implementation of `Control` class constructor
	19 marks:	correct implementation of `Control` class destructor
• 25 marks:	correct implementation of `List` class template	
	17 marks:	correct definition of class members and `Node` class
	8 marks:	correct changes to `add()` function
	Note: no marks if `List` class template is not used in the code	
• 25 marks:	correct implementation of overloaded operators	
	Note: no marks if operators are not used in the code	

Execution requirements:

- all marking components must be called, and they must execute successfully to receive marks
- all data handled must be printed to the screen for marking components to receive marks

Deductions:

- Packaging errors:
    - 10 marks for missing Makefile
    - 5 marks for a missing readme
    - 10 marks for consistent failure to correctly separate code into source and header files
    - 10 marks for bad style or missing documentation
- Major programming and design errors:
    - 50% of a marking component that uses global variables, global functions, or structs
    - 50% of a marking component that consistently fails to use correct design principles
    - 50% of a marking component that uses prohibited library classes or functions
    - 100% of a marking component that is *replaced* by prohibited library classes or functions
    - 50% of a marking component where unauthorized changes have been made to the base code
    - up to 10 marks for memory leaks, depending on severity
- Execution errors:
    - 100% of a marking component that cannot be tested because it doesn't compile or execute in VM
    - 100% of a marking component that cannot be tested because the feature is not used in the code
    - 100% of a marking component that cannot be tested because data cannot be printed to the screen