COMP 2404 -- Assignment #3

Due: Tuesday, November 20, 2018 at 12:00 pm (noon)

Goal

You will modify your event management program from either Assignment #2 or from the base code to manage two calendars instead of just one. Your program will also create a small inheritance hierarchy of events and use polymorphism to implement the ordering behaviour in each calendar.

Learning Outcomes

With this assignment, you will:

- apply the OO concepts of inheritance and polymorphism
- work with virtual and pure virtual functions in C++

Instructions

1. Draw a UML diagram

We are going to create a small inheritance hierarchy of events. The `Event` class will become abstract, and two concrete sub-classes will be created: one for school events, and one for work events. The main difference between the two is how they will be ordered in a calendar. School events will be added in ascending chronological order, and work events will be added in ascending order of priority.

To implement this, we will add a pure virtual `lessThan()` function to the `Event` class, and the appropriate behaviour will be implemented in the concrete sub-classes. Update your UML diagram from Assignment #2 (or create a new one) to represent the new class associations in the program. The UML diagram that you submit should reflect the design of the entire program for this assignment.

2. Modify the Event class

You will modify the `Event` class as follows:

- add a data member for the priority, which can be an integer
  - update the constructor and print function accordingly
  - we will need a getter member function for the priority
- add a `lessThan()` member function that is pure virtual; this function takes an `Event` pointer as parameter and returns a boolean
- existing private members will be better re-classified as protected

You will update the `List` class to compare events instead of dates when adding to the list.

3. Add two classes derived from Event

You will create two new classes: `SchoolEvent` and `WorkEvent`, both derived from the `Event` class. Each of the two new classes will contain:

- a constructor that invokes the base class constructor
- an implementation for the `lessThan()` member function
  - the comparison function for `SchoolEvent` will compare event dates
  - the comparison function for `WorkEvent` will compare event priorities

4. Modify the View class

You will modify the `View` class as follows:

- add a new member function to prompt the user for an event type (“School” or “Work”)
- update the read event information member function to prompt the user for the event priority

5. Modify the Control class

You will modify the `Control` class as follows:

- replace the calendar data member with two calendar data members: one calendar to hold school events and one to hold work events
- update the `launch()` function to:
  - using the `View` object, prompt the user to enter the type of event to be created
  - create either a new `SchoolEvent` or a new `WorkEvent` object, depending on the user’s selection
  - add the new event to the correct calendar: school events must be added to the school calendar, and work events to the work calendar
  - use the `View` object to print both calendars to the screen at the end of the program

6. 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 both 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:

- a UML class diagram (as a PDF file), drawn by you with a drawing package of your choice, that corresponds to the entire program design
- 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:

- 20 marks: correct UML diagram
  - 10 marks: correct previously existing classes and associations
  - 10 marks: correct new classes and associations
- 15 marks: correct changes to `Event` and `List` classes
  - 3 marks: correct definition and initialization of event priority
  - 5 marks: correct definition of `lessThan()` member function in `Event`
  - 7 marks: correct usage of event's `lessThan()` function in `List` class
- 30 marks: correct implementation of `SchoolEvent` and `WorkEvent` classes
  - 5 marks: correct implementation of **each** derived class constructor
  - 10 marks: correct implementation of **each** overridden `lessThan()` function
- 10 marks: correct changes to `View` class
  - 7 marks: correct implementation of read event type function
  - 3 marks: correct changes to read event info function
- 25 marks: correct changes to `Control` class
  - 4 marks: correct declaration and initialization of two calendar objects
  - 17 marks: correct creation and initialization of concrete event objects
  - 4 marks: correct printing of both calendars

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