COMP 2404 B/C – Assignment #1

Due: Thursday, February 1 at 11:59 pm

1. Goal

For this assignment, you will write a program in C++, in the Ubuntu Linux environment of the course VM, that manages a user's calendar of events. The program allows the end user to view all the events in the calendar, or view the events for one day only, or view the events belonging to one category, or add a new event.

2. Learning Outcomes

With this assignment, you will:

- implement simple, correctly designed C++ classes with composition associations between them
- practice passing parameters by reference using C++ references
- write and package a program following standard C++ and Unix programming conventions

3. Instructions

Your program will implement classes for a calendar and its events, as well as classes for the date and time of each event.

Every class in the program must follow the implementation conventions that we saw in the course lectures, including but not restricted to the correct separation of code into header and source files, the use of include guards, basic error checking, and the documentation of each class in the class header file. A more comprehensive list of constraints can be found in the *Constraints* section of this document.

3.1. Modify the `Date` class

You will begin with the `Date` class provided in the base code in the `a1-posted.tar` file posted in *Brightspace*, and you will modify it as follows:

3.1.1. Implement a `void set(Date& d)` member function that reuses an existing function to set the `Date` object's data members to the same values as the object in the `d` parameter.

3.1.2. Implement a `bool equals(Date& d)` member function that compares the values in the `Date` object with the given `d` parameter, and returns whether or not they are the same.
Remember: In C++, an object *always* has access to all the members of an object of the same class, even its `private` members.

3.1.3. Implement a `bool lessThan(Date& d)` member function that compares the values in the `Date` object with the given `d` parameter, and returns whether or not the `Date` object occurs earlier.

3.1.4. Implement a `bool validate(int d, int m, int y)` member function that verifies each parameter as a valid day, month, and year. The day and month parameters must both be within the correct range, and the year must be a positive number. The function returns true if all parameters are valid, and false otherwise.

3.2. Modify the `Time` class

You will modify the `Time` class provided in the base code, as follows:

3.2.1. Implement a `void set(Time& t)` member function that sets the `Time` object's data members to the same values as the object in the `t` parameter.

3.2.2. Implement a `bool lessThan(Time& t)` member function that compares the values in the `Time` object with the given `t` parameter, and returns whether or not the `Time` object occurs earlier.

3.2.3. Implement a `bool validate(int h, int m)` member function that verifies the parameters as valid hours and minutes. We will use a 24-hour clock, so valid times will range from midnight (00:00) to 23:59. The function returns true if both parameters are valid, and false otherwise.

3.3. **Implement the `Event` class**

You will implement an `Event` class that represents a single event in the calendar. The class must contain the following data members:

3.3.1. the unique *identifier* of the event, declared as an integer

3.3.2. the *name* of the event, stored as a C++ standard library `string` object

3.3.3. the *category* of the event, also stored as a string; for example, a user may wish to label some events as "Work" events, and others as "Home" events

3.3.4. a primitive array of strings representing the *participants* in the event, and the current number of elements in the array

3.3.5. the *date* of the event, as a statically allocated `Date` object

3.3.6. the *start time* of the event, as a statically allocated `Time` object

You will implement the following member functions:

3.3.7. a *default constructor* that takes as parameters an id, an event name, a category, a participant name, the year, month, day, start hours and start minutes of the event; the constructor must initialize all the required data members, using existing member functions as much as possible

3.3.8. getter member functions for the id, category, and date

3.3.9. a `void set(Event& e)` member function that sets the `Event` object's data members to the same values as the object in the `e` parameter, by performing a deep copy; remember that, for a deep copy, each array element must be copied individually, and **not** by using the assignment operator on entire arrays

3.3.10. a `void addParticipant(string p)` member function that adds the given participant to the participants array

3.3.11. a `bool lessThan(Event& e)` member function that compares two events and returns whether or not the `Event` object begins chronologically earlier than the given `e` parameter; both dates and times must be compared here

3.3.12. a `void print()` member function that prints out every data member of the event, including all participants, using the exact same format shown in the workshop video

(a) **do not** use C-style string formatting for this; you must use the `iomanip` library instead

(b) you must reuse existing member functions everywhere possible

3.4. **Implement the `Calendar` class**

You will implement a `Calendar` class that represents a user's calendar of events. The class must contain the following data members:

3.4.1. the *name* of the user to whom the calendar belongs, stored as a string

3.4.2. the calendar's *events* collection, stored as a primitive array of `Event` objects, and the number of events in the array

3.4.3. a *next event id* integer that stores the unique identifier that will be assigned to the next event added to the events array

You will implement the following member functions:

3.4.4. a default constructor that takes a string for the user name and initializes all the required data members; a calendar always starts with no events, and the initial value for the next event id can be found in a provided constant

3.4.5. a `bool addEvent(string n, string cat, string part, int yr, int mth, int day, int hrs, int mins)` member function that *inserts* the new event information provided in the parameters in its correct place in the events array, so that the events are always stored in *ascending* (increasing) chronological order, as follows:

(a) verify that there is sufficient capacity in the array to add the new event

(b) validate the date and time parameters by calling existing functions previously implemented

(c) if any of the above steps results in an error, a detailed error message must be printed to the screen, and the event cannot be added to the array

(d) declare a new *statically allocated* event as a local variable, using the parameter values and the next available event id, so that unique ids are generated in a sequential manner

(e) find the insertion point for the new event

  (i) this step requires that you traverse the elements in the array, and find where the new event belongs in ascending order by date and time

  (ii) you **must** reuse an existing function to perform the comparisons

(f) once you have found the insertion point, move every event from the insertion point to the end of the array one position towards the back (i.e. the end) of the array, to make room for the new event in the correct position

  (i) elements are “moved” by setting the event data members from the source object to the destination object; you must reuse an existing function for this

  (ii) **do not** add to the end of the array and sort; do not use any sort function or sorting algorithm or memory copy function anywhere in this program

(g) set the event element at the insertion point to the values in the new event

(h) increment the number of elements in the array

3.4.6. a `bool addParticipant(int id, string n)` member function that searches the calendar’s events array to find the event with the given `id`, and adds the provided name `n` to that event’s participants array; the function returns true if the event was found, and false otherwise

3.4.7. a `void print()` member function that prints out the calendar user’s name and the details of each event in the events array, using an existing function

3.4.8. a `void printByDay(int yr, int mth, int day)` member function that prints out the details of each event in the events array that occurs on the date provided in the parameters, as follows:

(a) the parameters must first be validated; if any parameter is invalid, then a detailed error message must be printed to the screen, and the print operation cannot be performed

(b) the calendar user’s name and the user-entered date must be printed to the screen

(c) existing functions must be reused to print the user-entered date, to perform the comparisons between dates, and to print each event to the screen

3.4.9. a `void printByCat(string cat)` member function that prints out the calendar user’s name and the details of each event in the events array that belongs to the provided category

NOTE: Every member function in this class must reuse existing functions that you implemented in a previous step.

3.5. Implement the main control flow

You will implement the control flow in the `main()` function as follows:

3.5.1. Declare a local `Calendar` object to represent the user’s calendar in the program. Let’s call this user Timmy Tortoise.

3.5.2. Load Timmy’s calendar data into the local calendar object by calling a global function provided in the base code.

3.5.3. Repeatedly print out the main menu by calling the provided `showMenu()` function, and process each user selection as described below, until the user chooses to exit. Verify that the user enters a valid menu option. If they enter an invalid option, they must be prompted for a new selection.

3.5.4. The “view calendar” functionality prints out all the events in Timmy’s calendar, and it must be implemented by calling an existing function.

3.5.5. The “view day calendar” functionality must prompt the user to enter a year, month, and day, and print out the calendar events for that day only.

3.5.6. The “view category calendar” functionality must prompt the user to enter a category and print out the calendar events that belong to that category only.

3.5.7. The “add an event” functionality must prompt the user to enter the event name, category, main participant, year, month, day, and start hours and minutes. An existing function must be called to create a new event from the user-provided information and add it to Timmy’s calendar. The program must inform the user whether or not this operation was successful.

NOTE: It may be necessary to use a C++ library function to read in a multi-word string for the event name, for example: `getline(cin, eventName);`

3.6. **Packaging and documentation**

3.6.1. Your code must be correctly separated into header and source files, as we saw in class.

3.6.2. You must provide a `Makefile` that separately compiles each source file into a corresponding object file, then links all the object files to produce the program executable. Your `Makefile` must also include the `clean` target that removes all the object files and the program executable. **Do not** use an auto-generated Makefile; it must be specific to this program.

3.6.3. You must provide a plain text `README` file that includes:

(a) a preamble (program author, purpose, list of source, header, and data files, if applicable)

(b) compilation and launching instructions, including any command line arguments

3.6.4. Use either the `tar` or the `zip` utility in Linux to package into one `.tar` or `.zip` file **all the files** required to build and execute your program.

3.6.5. Do not include any additional files or directories in your submission.

3.6.6. All class definitions must be documented, as described in the course material, section 1.3. Please **DO NOT** place inline comments in your function implementations.

4. Constraints

Your design and implementation must comply with all the rules of correct software development and programming conventions that we have learned during the course lectures, including but not restricted to:

4.1. The code must be written using the C++11 standard that is supported by the course VM, and it must compile and execute in that environment. It must not require the installation of libraries or packages or any software not already provided in the course VM.

4.2. Your program must not use any library classes or programming techniques that are not used in the in-class coding examples. Do not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.3. If base code is provided, do not make any unauthorized changes to it.

4.4. Your program must follow basic OO programming conventions, including the following:

4.4.1. Do not use any global variables or any global functions other than the ones explicitly permitted.

4.4.2. Do not use `struct`s. You must use classes instead.

4.4.3. Objects must always be passed by reference, never by value.

4.4.4. Functions must return data using parameters, not using return values, except for getter functions and where otherwise permitted in the instructions.

4.4.5. Existing functions and predefined constants must be reused everywhere possible.

4.4.6. All basic error checking must be performed.

4.5. You may implement helper classes and helper member functions, but only if the design is consistent with the provided instructions. Design and implementation choices that undermine the learning outcomes will not earn any marks.

5. Submission Requirements

5.1. You will submit in *Brightspace*, before the due date and time, one `tar` or `zip` file that includes all your program files, as described in the **Packaging and documentation** instructions.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *Brightspace* will be graded. Submissions that contain incorrect, corrupt, or missing files will be graded as is. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading Criteria

- 40 marks: code quality and design
- 30 marks: coding approach and implementation
- 25 marks: program execution
- 5 marks: program packaging