COMP 2404 A/C – Assignment #1

Due: Wednesday, February 3 at 11:59 pm

1. Goal

For this assignment, you will write a C++ program, in the VM provided for this course, to manage a user's personal agenda of appointments. Your program will implement an agenda that stores multiple calendars to hold the different types of appointments, for example a work calendar and a personal calendar. Each calendar will contain the dates that represent the appointments and events in the user's busy life. You will practice writing correctly designed code with simple classes in C++, as well as working with dynamically allocated memory.

2. Learning Outcomes

With this assignment, you will:

- write correctly designed code with simple C++ classes with composition associations between them
- work with dynamically allocated memory and pointers
- write and package a program following standard C++98 and Unix programming conventions

3. Instructions

Your program will implement several new classes representing a user's agenda, the different calendars contained in the agenda, and the dates contained within each of those calendars.

For example, some users may choose to manage separately the events in their work or school life from those in their personal life. Therefore, they may have one calendar for work or school events, and a different calendar for their personal appointments. The combination of the user's multiple calendars together will make up their agenda. Your program will initialize the user's multiple calendars with several dates. It will print out the content of the individual calendars, as well as the entire agenda with the combined dates of all the calendars.

This program requires the implementation of several new classes. All new classes 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 below.

3.1. Modify the `Date` class

You will begin with the `Date` class that we worked on during the lectures. You can find this class in the in-class coding examples posted in *cuLearn*, in section 1.5, program #2.

Modify the `Date` class to add the following data members:

3.1.1. an event name, which is a C++ standard library `string` object

3.1.2. a start time for the event, which is an integer representing the number of minutes since midnight

3.1.3. the duration of the event, in minutes

3.1.4. a flag to represent whether the event is recurring, stored as a boolean

3.1.5. the number of weeks for which the event recurs

3.1.6. a flag to indicate whether the date information contained in the object is valid, stored as a boolean

The `Date` class will contain the following additional member functions:

3.1.7. a default constructor that takes 9 parameters, in the following order: the event name, day, month, year, starting hour, starting minute, duration, recurrence, and number of recurring weeks; the default constructor will initialize all the data members from these parameters, as follows:

(a) the name and recurrence data members can be set directly from parameters

(b) the valid flag will be set to the result of the new `validate()` member function, described below; if the parameters are not valid, then default values must be used

(c) the existing `setDate()` function must be called to validate and set the day, month, and year

(d) the given starting hour and starting minute parameters must be validated, and the start time of the event must be stored as a number of minutes since midnight, using these parameters

(e) the duration must be validated and set from a parameter

(f) if the event is recurring, the number of recurring weeks must be set to the corresponding parameter, otherwise it must be initialized to zero

(g) all error checking must be performed

(h) the constructor must specify default values for all parameters; by default, events do not recur, and the default number of recurring weeks must be 10

3.1.8. a copy constructor, as seen in the coding example in section 1.5, program #4

3.1.9. a `validate()` member function that takes as parameters the values for the day, month, year, start hour, start minute, and duration, and returns whether or not these values are valid

3.1.10. a `bool lessThan(Date* d)` function that compares two `Date` objects, and returns true if the date in the `this` object occurs earlier than the date in the `d` parameter

3.1.11. getter member functions for the recurrence and valid flags

3.1.12. a `void print()` member function that replaces both `printShort()` and `printLong()`; this function prints to the screen all the data members, except the duration; instead, the start time and end time of the event must both be printed to the screen in the HH:MM format; you can assume that all events start and end on the same day

3.1.13. any other private helper member functions that are necessary for good design

3.2. Implement the `Calendar` class

You will create a new `Calendar` class that manages a primitive array of dates and provides the required member functions to manipulate them.

The `Calendar` class will contain the following data members:

3.2.1. a calendar name, which will be a C++ standard library `string` object

3.2.2. a collection of the dates in the calendar

(a) this will be a *dynamically allocated array* of `Date` object **pointers**

(b) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

3.2.3. the current number of dates in the array

The `Calendar` class will contain the following member functions:

3.2.4. a default constructor that takes a calendar name as parameter, and initializes all the data members

(a) when allocating the dynamic array, you will define a preprocessor constant for the maximum number of dates; this can be set to a reasonable number such as 64;

(b) it is **not** necessary to initialize each array element to null, since your code should never access any element beyond the current number of dates

(c) it is bad form to initialize data members in the class definition; you must do this in the body of the constructor instead

3.2.5. a destructor that deallocates the memory for each date in the calendar, as well as the dates array itself

3.2.6. a getter function for the calendar name

3.2.7. a `bool add(Date*)` member function that inserts the given date in the array, **directly in its correct place**, in ascending (increasing) order by date

(a) you must **shift** the elements in the array towards the back (the end) of the array to make room for the new element in its correct place; **do not** add to the end of the array and sort; do not use any sorting function or sorting algorithm

(b) you must perform all basic error checking; invalid dates cannot be added to the array

(c) this function returns true if no errors occurred, and false otherwise

3.2.8. a `void merge(Calendar* c)` member function that takes every date in the given calendar `c`, creates a copy of this date, and adds the copy to the current (the `this`) calendar; existing functions must be reused everywhere possible

3.2.9. a `void printDates()` member function that prints to the screen the information for each date in the calendar; using correct design principles, this function must call an existing function on each `Date` object

3.2.10. a `void print()` member function that prints to the screen all the calendar information, including the calendar name and all the information for each date in the calendar

3.3. Implement the `Agenda` class

You will implement a new `Agenda` class that manages a collection of calendars for the user.

The `Agenda` class will contain the following data members:

3.3.1. an agenda name, stored as a `string`

3.3.2. a collection of the calendars in the agenda

(a) this will be a *statically allocated array* of `Calendar` object **pointers**

(b) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

3.3.3. the current number of calendars in the array

The `Agenda` class will contain the following member functions:

3.3.4. a default constructor that takes an agenda name as parameter, and initializes the necessary data members

3.3.5. a destructor that deallocates the dynamically allocated calendars contained in the calendars array

3.3.6. a `bool add(Calendar*)` function that adds the given calendar to the back (the end) of the calendars array; this function returns true if the calendar was successfully added, and false otherwise

3.3.7. a `bool find(string n, Calendar** c)` function that searches the calendars array for the calendar with the name specified in parameter `n`, and returns that calendar pointer using parameter `c`; you **must** use the parameter to return this data; **do not** use the return value

3.3.8. a `bool add(Date* d, string n)` function that does the following:

(a) find the calendar to which the date will be added; this is the `Calendar` object with the given calendar name `n`; if the corresponding calendar is not found, the date cannot be added

(b) perform all relevant error checking

(c) add the given date `d` to the calendar's dates array

(d) return true if the date was successfully added, and false otherwise

3.3.9. a `void print()` function that prints out all the **combined dates** contained all the calendars, in ascending order; the function does the following:

(a) create a temporary calendar object

(b) for every calendar in the agenda, merge the content of that calendar into the temporary calendar; this copies every date and inserts it into the temporary calendar in the correct position

(c) print the agenda name to the screen

(d) print out the dates in the temporary calendar to the screen

Note #1: You must use correct design principles in the implementation of all the above functions. This means that you must reuse existing functions everywhere possible, and perform all error checking.

Note #2: Do not implement any additional getter or setter functions, as these are not necessary. However, you may implement additional "helper" member functions (not global functions) as needed.

3.4. Write the `main()` function

Your `main()` function must test your program thoroughly. To do so, it will initialize an agenda object and two calendar objects. It will initialize several **different** dates and add them to the different calendars, and it will print the calendars and the agenda data to the screen.

Your program must use a *global function* to initialize the dates, specifically the `initDates(Agenda&)` function that is provided in the `a1-posted.cc` file posted in *cuLearn*.

Your `main()` function will do the following:

3.4.1. declare an agenda object with an appropriate name; this object can be statically allocated

3.4.2. dynamically allocate two calendar objects; you will give one of them the name "Work" and the other "Home"

3.4.3. add both calendars to the agenda

3.4.4. initialize the data by calling the `initDates()` function provided, using the agenda that you declared; this will create multiple dates and add them to the agenda's "Work" and "Home" calendars

3.4.5. print out each calendar separately; this will show the individual dates in each calendar, in ascending order

3.4.6. print out the agenda; this will show the dates in both calendars combined, in ascending order

3.5. **Packaging**

Every assignment in this course is required to follow the conventional packaging rules for Unix-based systems:

3.5.1. Your code must be correctly separated into header and source files, as seen in class.

3.5.2. You must provide a Makefile that compiles and links all your code into a working executable.

(a) **do not** use a generic Makefile; it must be specific to this program

3.5.3. You must provide a README file that contains a preamble (program author, purpose, list of source, header, data files if applicable), as well as compiling and launching instructions.

3.5.4. **Do not submit** any additional files, including object files, executables, or supplementary files or directories (macOS users must remove their additional hidden directories).

3.6. **Test the program**

You must provide code that tests your program thoroughly. For this program, the use of the provided global function will be sufficient. Specifically:

3.6.1. Check that the date information is correct when it is printed at the end of the program.

3.6.2. Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

4. Constraints

Your program must comply with all the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

4.1. The code must be written in C++98. It must compile and execute in the default course VM, without any libraries or packages or any software that's not already provided in the VM.

4.2. Your program must not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

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

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

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

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

4.3.4. Existing functions must be reused everywhere possible.

4.3.5. All basic error checking must be performed.

4.3.6. All dynamically allocated memory must be explicitly deallocated.

4.4. All classes must be thoroughly documented in every class definition, as indicated in the course material, section 1.3.

5. Submission

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

5.1.1. One `tar` or `zip` file that includes:

(a) all source and header files

(b) a Makefile

(c) a README text file that includes:

(i) a preamble (program author, purpose, list of source and header files)

(ii) compilation and launching instructions

NOTE: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

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 *cuLearn* will be graded. Submissions that contain incorrect, corrupt, or missing files will receive a grade of zero. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading

6.1. **Marking components:**

- 25 marks: correct modifications to `Date` class
- 35 marks: correct implementation of `Calendar` class
- 34 marks: correct implementation of `Agenda` class
- 6 marks: correct implementation of `main()` function

6.2. **Execution requirements:**

6.2.1. all marking components must be called and execute successfully in order to earn marks

6.2.2. all data handled must be printed to the screen for marking components to earn marks

6.3. **Deductions:**

6.3.1. Packaging errors:

(a) 10 marks for missing Makefile

(b) 5 marks for missing README

(c) up to 10 marks for failure to correctly separate code into header and source files

(d) up to 10 marks for bad style or missing documentation

6.3.2. Major design and programming errors:

(a) 50% of a marking component that uses global variables or `struct`s

(b) 50% of a marking component that consistently fails to use correct design principles

(c) 50% of a marking component that uses prohibited library classes or functions

(d) up to 100% of a marking component where Constraints listed are not followed

(e) up to 10 marks for bad style

(f) up to 10 marks for memory leaks

6.3.3. Execution errors: 100% of any marking component that cannot be tested because it doesn't compile or execute in the provided course VM, or the feature is not used in the code, or data cannot be printed to the screen, or insufficient datafill is provided.