Purpose: Gain experience in defining a C++ class. Gain experience in using and manipulating static C++ arrays, particularly partially-filled arrays.

Assignment: This assignment deals with an implementation of a calendar which stores a collection of reminders. In this case, the reminders will be stored in a static array. The reminders will be stored in sorted (chronological) order. The array will be able to contain up to MAX_REM (currently defined to be 50) reminders. Since not all elements of the array may contain reminders that are a part of the calendar (i.e., we have a *partially filled array*), we need to keep track of the number of elements that we are using in the array. The Calendar.h file describes all the functions provided by the class.

You will be provided with the source code that supports the reminders. Your job is to implement the Calendar class.

Functional Specifications: You will be supplied seven files:

- **Date.h**: declaration of the Date class (DNC)
- **Date.cpp**: definition of the Date class (DNC)
- **Reminder.h**: declaration of the Reminder class (DNC)
- **Reminder.cpp**: definition of the Reminder class (DNC)
- **Calendar.h**: declaration of the Calendar class.
- **Calendar.cpp**: definition of the Calendar class, which currently holds only method stubs. Your job is to replace the method stubs with correct code to implement each of the methods.
- **CalTest.cpp**: initial test program. You should replace this with the test program you developed in Project 0.

The Calendar you have to implement is a static array of Reminder objects. A predetermined maximum size of the static array is defined by the constant identifier MAX_REM. The Calendar class consists of the static array named `remArr`, and a variable `numRem` that keeps track of the number of array elements that are currently being used. You should store Reminders in the array in sorted order (based on the chronological order of the dates). The existing method stubs in Calendar.cpp have detailed comments describing the desired behavior of the method (they are the same as the comments in Calendar.h). You should not change files marked DNC (DNC == Do Not Change). Changes to Calendar.h are limited to only adding private helper methods (which are optional; helper methods are not required).

Here are the methods of the Calendar class that you are to write:

Calendar class	
Methods	**Function**
`Calendar()`	Default constructor – creates an empty Calendar
`getNumRem()`	Return the total number of Reminders in the Calendar
`addRem(const Reminder &r)`	Add a reminder to the Calendar
`getRem(size_t index)`	Returns the reminder at the specified index
`toString()`	Return a string representation of the calendar, i.e., a string of all the reminders in chronological order
`displayRem(size_t index)`	Return a string of the reminder at a particular index
`displayRem(const string& str)`	Return a string of all reminders whose message matches the provided string
`displayRem(const Date& d)`	Return a string of all reminders for a given date
`displayRem(const Date& d1, const Date& d2)`	Return a string of all reminders in a range of two given dates
`findRem(const Date& d)`	Find first reminder for the given date and return its index
`findRem(const string& str)`	Find first reminder with the given message and return its index
`deleteRem()`	Deletes all reminders earlier than today's date
`deleteRem(size_t index)`	Deletes the reminder at a provided index position
`deleteRem(const string& str)`	Delete all reminders whose message matches a given string

11. The array of Reminders used to represent the Calendar will be a ***partially-filled array***. See the information on partially-filled arrays below. Note that you will lose points if your class methods do unnecessary work; i.e., if you process elements of the array that are not currently being used to represent the Calendar.
12. If you do not understand how a method should work from its description, then can go back to project #0 and try things out – you were given a working Calendar class after all.

Partially filled arrays: Here is some additional information on what is meant by a "*partially filled array*".

In C++ (and Java too), once an array is created its size is set. You cannot make it bigger, nor can you make it smaller. If you ever wanted the array to be bigger/smaller, you would have to reallocate a completely new array (something we will see in lecture very soon). In this assignment, we are never reallocating a new array (nor can we do so due to the way the array was declared).

In C++, once an array is created every element of the array contains a data item of the declared type of the array. I.e., every element of an integer array contains an integer; and every element of a Reminder array contains a Reminder object.

Now, there are times when we don't know ahead of time how many items we will want to store in our array. In such cases, we have to make a reasonable guess at the maximum size needed and allocate an array of that size. In this situation, we will not be using all the array elements all the time, so we have to keep track of the number of array elements that we are actually using in the array and we will use the contiguous array elements starting at index zero. This is known as a *partially filled array*. To make this work we need to keep track of three separate things:
1) the array where we are storing data (e.g., remArr)
2) the size of the array (e.g., MAX_REM)
3) the number of elements of the array that are currently being used (e.g., numRem); again, the numRem data items are stored in the array from index 0 to index numRem-1.

In this partially filled array, you cannot set array elements to some special value to indicate that the element is currently "not used" or "deleted". What value would you use? It would have to be some value that could not be a valid value added by the user -- but any value you can create to indicate "not used" the user could also potentially create to add to the array.

If you ever need to know if an array element is currently being used, you only need to consider the value of numRem. All elements from index 0 to index numRem-1 are used, and all elements from index numRem to index MAX_REM-1 are not used.

To add an item to a partially filled array (assuming there is space available), you need to increment the count of used elements and place the new item in the appropriate position of the array (shifting existing data up if necessary). To delete an item from a partially filled array, you need to decrement the count of used elements and shift data down to fill in the hole left by the deleted element. Deleting will cause some elements of the array to change from being in the "used" portion of the array to being in the "unused" portion of the array. There is no need to set these elements to some special value -- since they are now in the unused portion of the array, we should not be accessing those values any longer.

When processing the data of a partially filled array, you only process the array elements that lie in the portion that is currently being used. Any operations you perform on the unused portion of the array are unnecessary (and thus wasteful).

Background: A calendar is simply a collection of reminders. A reminder is simply a dated string (i.e., the text of the reminder and an associated date). Reminders are created/supported through two classes: the Date class and the Reminder class.

In this program, a date is recognized in the MM/DD/YYYY format (Note: We have a four-digit year, which indicates that "18" would indicate the year 20, and not 2020, and so on. Also, it is assumed that the year cannot be more than four digits long). A reminder is a string message associated with a date. Since the reminder is essentially a date with an added feature (the associated message), the Reminder class is derived from the Date class by using inheritance.

The properties of the Date and Reminder classes (which are already provided to you) are given below (note: a Date object contains a month, day, and year, and is a very simplified class and so it does not handle leap year issues):

Date class	
Methods	**Function**
`Date()`	Default constructor – creates a date object having today's date
`Date(int m, int d, int y)`	Overloaded constructor that creates a Date object with the values of month, day and year corresponding to the passed values of m, d, and y
`Date(const string& dateStr)`	Overloaded constructor that parses a string in mm/dd/yyyy form and creates a Date object
`void setDate(const Date &d)`	Sets the date of current Date object to the date of the passed Date object
`void setDate(int m, int d, int y)`	Sets the date of the current Date object with the values of month, day and year corresponding to the passed values of m, d, and y
`void setDate(const string& dateStr)`	Sets the date of the current Date object by parsing a string in mm/dd/yyyy form
`void incrementDay()`	Changes the date to the next day
`void decrementDay()`	Changes the date to the previous day
`void incrementMonth()`	Changes the date to the next month
`void decrementMonth()`	Changes the date to the previous month
`void incrementYear()`	Changes the date to the next year
`void decrementYear()`	Changes the date to the previous year
`int getMonth()`	Returns the integer value of the month of the date
`int getDay()`	Returns the integer value of the day of the date
`int getYear()`	Returns the integer value of the year of the date
`string toString()`	Returns a formatted string (**mm/dd/yyyy**) of the date object
`Date operator+= (int n)`	Adds 'n' days to current date
`Date operator-= (int n)`	Subtracts 'n' days to current date
`Date operator+(int n)`	Adds 'n' days to current date and returns the value
`Date operator-(int n)`	Subtracts 'n' days from current date and returns the value
`int operator-(const Date& d)`	Finds the difference between two dates (overloaded operator)
`Date operator++()`	Pre- and Post-increment
`Date operator--()`	Pre- and Post-decrement
`bool operator==(const Date& d)`	Compares date to d. If the two dates are equal, returns true, else returns false
`bool operator!=(const Date& d)`	Compares date to d. If the two dates are not equal, returns true, else returns false
`bool operator< (const Date& d)`	Compares date to d. If date < d, returns true, else returns false
`bool operator<=(const Date& d)`	Compares date to d. If date <= d, returns true, else returns false
`bool operator> (const Date& d)`	Compares date to d. If date > d, returns true, else returns false
`bool operator>=(const Date& d)`	Compares date to d. If date >= d, returns true, else returns false

Note on using overloaded operators: In the above class description you see that several operators have been overloaded for the class; for example "`operator==`" has been defined. You can use these like any other method of the class; for example if `d1` & `d2` were two `Date` objects, you could type: `d1.operator==(d2)` to test if the two dates were equal. However, it is much easier to simply use them as the operators you know and love; for example: `d1==d2`.

Reminder class As the Reminder class is derived from the Date class, all the functions in the Date class can also be used with the Reminder class. There are a few added functionalities, as described below:	
Methods	**Function**
`Reminder()`	Default constructor – creates a Reminder object having today's date and an empty string
`Reminder(const Date& d, const string& str)`	Overloaded constructor that creates a Date object with d as the date and str as the message
`void setMsg (const string& msgStr)`	Sets the message of the reminder to the string msgStr that was passed in the argument
`string getMsg()`	Returns the message of the reminder in form of a string
`Date getDate()`	Returns the date of the reminder in form of a date object

`string toString()`	Returns the reminder in form of a formatted string: **On mm/dd/yyyy: Message** (Where mm/dd/yyyy is the formatted date, and Message is the string message of the reminder)
`Reminder operator+(int n)`	Adds 'n' days to current reminder and returns the value
`Reminder operator-(int n)`	Subtracts 'n' days from current reminder
`bool operator==(const Reminder& r)`	Compares two reminders by comparing their dates and messages.
`bool operator!=(const Reminder& r)`	Compares two reminders by comparing their dates and messages.

Note that the Reminder class has a getDate() method that returns the date of the reminder. However you should rarely need to use it. Because the Reminder class is derived from the Date class, every Reminder object is also a Date object and can be treated as a Date object. So when you want to compare the dates of two Reminders, you do not need to extract the dates first before comparing, rather you can directly compare the two Reminders.

Many of the methods of the Date and Reminder classes are declared to be *const* methods; indicating that they do not modify the object on which they are called. Both the Date and Reminder classes have overloaded insertion operators so that the objects can easily be displayed with the << insertion operator.

You should **NOT** make any changes to the Date or Reminder classes.

`deleteRem(const Reminder& rem)`	Delete all occurrences of the given reminder
`deleteRem(const Date& d)`	Deletes all reminders on a particular date
`deleteRem(const Date& d1, const Date& d2)`	Deletes all reminders between a range of two given dates

Implementation details: Here are a few notes that might be helpful:

1. You are to download a zip file that is provided with this specification. The zip file contains a CLion project that includes the starter code for this project. You ***must*** unzip/extract the files before you work on them. Once you unzip the file, you can open the project by starting CLion, then specify that you want to "Open Project", and then navigate to the folder you just extracted (please review the instructions that were given with project #0 on how to open provided CLion projects). When you open the project, let CLion do its initialization work and load symbols. The code as provided compiles cleanly and the program should run as given – though the results are incorrect since required functionality is missing. Again, you ***must*** unzip/extract the files before you work on them. Again, please review the instructions that were given with project #0 on how to open provided CLion projects and make sure your CMakeLists.txt file contains the required compiler flags.
2. Write your code in terms of MAX_REM (the predefined maximum size of the static array) and not the constant 50. We may want to change the size of the array in the future, and such a change should only require changing one line of code.
3. Note that you do not need to implement the methods in the order they appear in the file. You can implement them in any order. So THINK first! It is also highly recommended that you write your code incrementally. You should implement one or two methods at a time and fully test them before moving on to other methods. You can simply comment out code in the CalTest.cpp program that is testing methods that you have not yet implemented, then uncomment the code when you want to reactivate those tests (CLion has the ability to quickly comment/uncomment highlighted code with an option found under the Code drop-down menu). You may also want to enhance your test code as you go along, so that it does a better job of testing "edge conditions" – conditions that are valid but are out of the norm.
4. You will likely be performing some operations with C++ strings. C++ string operations are described here.
5. The class constructors are required to use the base member initialization list as much as possible.
6. The methods that add & delete Reminders from the Calendar are expected to do their work "*in place*". That means that they do all their work in the existing array inside the object. You will not receive full credit if you create local temporary arrays or local temporary Calendar objects and then copy data to/from the array. You should only use the given array in the object and shift data up or down in the array as necessary. Moving data to & from a temporary array or a temporary object is inefficient and unnecessary work.
7. An additional stipulation of the `addRem()` method is that it must run in linear time. This implies that you must insert the newly added reminder in its correct location, shifting other reminders in the array up to make room for it. If you simply add the new reminder at the end of the array and then sort the array you will lose points for efficiency.
8. Several methods of the Calendar class have parameters of type size_t (an unsigned integer type). The compiler may give you conversion warning messages if you attempt to assign/compare such a parameter to a variable of type int. Your homework submissions are expected to compile cleanly (no errors or warnings) in CLion with the clang compiler. To eliminate warning about size_t conversions, you should use variables of type size_t when appropriate, or you can cast the value of the size_t parameter to an int when necessary, or cast int values to size_t as appropriate.
9. The Calendar.h specifies that some methods throw an exception if someone attempts to perform an operation that is not allowed (e.g., adding a reminder to a full calendar). To have your method throw an exception in C++, you can use the follow statement: `throw std::overflow_error("Array is full.");` This statement requires that you include the following at the top of your Calendar.cpp file: `#include <stdexcept>`. Note 1: leave the #include statement in the .cpp file regardless of what the CLion IDE tells you. Note 2: there is no "**new**" operation needed to throw exceptions as in Java. You do not need to specify that the method may throw an exception; you can simply throw the exception if the error condition occurs. More information on C++ exceptions can be found in Chapter 21 of our text (zyBooks). Note: the Calendar class simply throws exceptions in error situations – it does not attempt to handle the error. Thus your Calendar class should not contain any try-catch blocks – that is the job of the program that is *using* the Calendar class (i.e., the CalTest.cpp program).
10. Since we are using a static array in the Calendar class, the C++ compiler takes care of creating the array for you whenever a Calendar object is created. Your constructor should not have any "**new**" operations (I state this for the benefit of the Java programmers in the class).