CS 133 A

Project 1: "Making a Date, Marking Time"

Name: _____

It is normal to want a class that contains both the date and the time. It makes sense that a class like this would inherit from both a Date class and a Time class. These in turn should each inherit from a base class that makes sure that you can compare them with each other to order them.

In order to do this, you will need to create four related classes. **All** of these should be in your own namespace (**<*your initials*>DATETIME**). These classes should contain ***at least*** the following:

- **Comparable:** an **abstract base class** containing *at least* the following **pure virtual** methods:
  - **`virtual bool` operator==(const Comparable &other)const** -- returns true if the two objects are identical
  - **`virtual bool` operator!=(const Comparable &other)const** -- returns false if the two objects are identical
  - **`virtual bool` operator<(const Comparable &other)const** -- returns true if this object is less than other
  - **`virtual bool` operator<=(const Comparable &other)const** -- returns true if this object is less than or equal to other
  - **`virtual bool` operator>(const Comparable &other)const** -- returns true if this object is greater than other
  - **`virtual bool` operator>=(const Comparable &other)const** -- returns true if this object is greater than or equal to other
  - **`virtual void` input(istream& sin)** -- inputs object info in proper form -- Could have a return type of istream&
  - **`virtual void` print(ostream& sout)const** -- outputs object info in proper form -- Could have a return type of ostream&
- **Date:** contains information about a single date. This class inherits virtually from **Comparable.**

This will be your own class, separate from any that may be available in C++. It should contain *at least* the following

public properties:

- **`public static constants`** for
  - lowest year allowed (*1760 works well -- a leap year just after Gregorian calendar starts in 1752*)
  - starting day-of-the-week for counting (*January 1 of 1760 was a Tuesday*)
  - months in a year (*12*)
  - days in a week (*7*)

  (note: these could also be declared globally to the namespace)

protected properties:

- **`short year`** -- the full year (1760+)
- **`short month`** -- (0 - 11) -- You may use an enum or enum wrapper type instead of short
- **`short dayOfMonth`** -- (1 - <*numberOfDaysInTheMonth*>)
- **`short dayOfYear`** -- (1 - 365 *or 366*)

- **`short dayOfWeek`** -- (0 - 6 -- *0 is Sunday*) -- You may use an enum or enum wrapper type instead of short

public methods:

- A constructor -- taking in 3 parameters (*dayOfMonth, month, year*) -- all with defaults set to **-1**. The constructor should begin by calling **`setCurrentDate()`**. Any out-of-range parameter will cause that property to remain the same as the current date. ***(e.g., the arguments (9, 3, 99) would make the date be Tuesday, April 9, 2014)***.
- **definitions** for the inherited virtual methods including:
  - **`==, !=, <, <=, >, >=`**
  - **`void print`** -- takes in an ostream& and returns the day-of-week, day-of-month, month, year *(as text), **e.g., Tuesday, April 9, 2014"***
  - **`void input`**-- takes in an istream& and inputs into *this in the form: "**dd/mm/yyyy**". Should be able to properly take in just day, day and month or day, month and year. Any out-of-range parameter will cause that property to remain the same as the current date.
- **`static boolean isLeapYear`** -- takes in a year. The rule is: A year is a leap year if it is divisible by 4 unless it divisible by 100 except if it is also divisible by 400.
- **`static short daysInMonth`** -- takes in a month number and a year and sends back the proper number of days.
- **`static string monthNames`** -- takes in a month number and returns the string equivalent (***e.g.**, monthNames(0) returns **January**)'*
- **`static string dayNames`** -- takes in a day-of-week number and returns the string equivalent (***e.g.**, dayNames(0) returns **Sunday**)'*
- Accessors and Mutators (*sets and gets*) for the appropriate properties. Make sure that mutators change values **ONLY if** the values are valid and that all properties remain consistent with each other. *See the descriptions for **setDayOfYear** and **setDayOfWeek** below.*
- **`Date yesterday`** -- returns a **Date** representing the day ***before** this* day.
- **`Date tomorrow`** -- returns a **Date** representing the day ***after** this* day.
- **`void setCurrentDate()`**

  (*requires an **#include** <ctime>*)

  ***N.B.**: The set of code given here and below in setCurrentTime are the **ONLY** places you are allowed to use built-in time and date utilities.*

```
{
 const short BASE_YEAR = 1900;
 time_t rawtime;
 tm *currentTimePtr;
 time(&rawtime);
 currentTimePtr = localtime(&rawtime);
 m_year = currentTimePtr->tm_year + BASE_YEAR;
 m_month = currentTimePtr->tm_mon;
 m_dayOfMonth = currentTimePtr->tm_mday;
 // add your own additional code to properly set dayOfYear and
 // dayOfWeek (see below)
 // DO NOT use the built-in time and date utilities to do this
 // --do it yourself
}
```

private methods:

- **`static short countLeaps`** -- takes in a year and returns the number of leap years from the base year to it. In general, if you take the difference a number of years, the number of leap years in that range is the number of years ÷ 4 -the number of years ÷ 100 + the number of years ÷ 400. *Be careful if you go in negative years!*
- *An easy algorithm is to start at 1760 (which **was** a leap year) and loop through all of the years to the year sent in, adding one to your count if `isleapyear(yearTested)`.*
- **`void setDayOfYear`** -- properly sets dayOfYear. (*January 1 is day 1*).

- **`void setDayOfWeek`** -- properly sets dayOfWeek.
  *One algorithm is to start at January1, 1760 (which **was** a Tuesday) and loop through all of the years to the year sent in, adding one to your count for each year (one more if `isleapyear(yearTested)`. Then add in dayOfTheYear and mod by DAYS_IN_A_WEEK.*

You should also, outside of the class, include an operator << that calls the print method, and an operator >> which calls the input method.

CTime: contains information about a single time. This class inherits virtually from **Comparable.**

This will be your own class, separate from any that may be available in C++

properties:

- **`public static constants`** for
  - number of hours in a day (24)
  - number of minutes in an hour/seconds in a minute (60)

protected properties:

- **`short hour`** -- (0 - 23)
- **`short minute`** -- (0 - 59)
- **`short second`** -- (0 - 59)

public methods:

- Two constructors -- a default constructor, which sets the time to the current time, and a constructor taking in 3 parameters (*hour, minute,* second) -- all but the hour with defaults set to **0**. For the constructor which takes parameters, any out-of-range parameter will cause that property to be set to ***0*** ***(e.g., the arguments (10, 3, 99) would make the time be 10:03:00)***.
- **definitions** for the inherited virtual methods including:
  - **`==, !=, <, <=, >, >=`**
  - **`void print`** -- takes in an ostream& and returns the hour, minute, second *(as text),* ***e.g., "6:30:00"***
  - **`void input`**-- takes in an istream& and inputs into *this in the form: "**hh:mm:ss**". Should be able to properly take in just the hour, hour and minute or hour , minute and second. Any out-of-range parameter will cause that property to be set to ***0***.
- **`void setCurrentTime()`**
  (*requires an **#include** <ctime>*)
  ***N.B.**: The set of code given here and above in setCurrentDate are the **ONLY** places you are allowed to use built-in time and date utilities.*

```
{
    time_t rawtime;
    tm *currentTimePtr;
    time(&rawtime);
    currentTimePtr = localtime(&rawtime);

    m_hour = currentTimePtr->tm_hour;
    m_minute = currentTimePtr->tm_min;
    m_second = currentTimePtr->tm_sec;
}
```

- Accessors and Mutators (*sets and gets*) for the appropriate properties. Make sure that mutators change values **ONLY if** the values are valid and that all properties remain consistent with each other.

You should also, outside of the class, include an operator << that calls the print method, and an operator >> which calls the input method.

DateTime: contains information about a single date and time. This class inherits from **Date and Time.**

This will be your own class, separate from any that may be available in C++
public methods:

- Three constructors -- a default constructor, which sets the date to the current date and the time to the current time; a constructor taking in 6 parameters (*dayOfMonth, month, year, hour, minute,* second) -- all but the dayofmonth with defaults set to **0**; and a constructor taking in a ***const date&*** and a ***const time&.*** Any out-of-range parameter will cause that property to remain the same as the current date/time.
- **definitions** for the inherited virtual methods including:
  - **`==, !=, <, <=, >, >=`**
  - **`void print`** -- takes in an ostream& and returns the date and time *as text),* ***e.g., Tuesday, April 9, 2013 16:30:00"***
  - **`void input`**-- takes in an istream& and inputs into *this in the form: "**dd/mm/yyyy hh:mm:ss**" or any valid date input followed by any valid time input.

You should also, outside of the class, include an operator << that calls the print method, and an operator >> which calls the input method.

In separate .h and .cpp files, you should define quicksort, printArray, safeRead and other necessary or helpful functions to be used in the program.

The program that uses these classes should ask the user the number of elements they wish to enter. Using a variable of type **Comparable****, you should dynamically allocate the array, then allow the user to enter in that many DateTime objects to the array. Using an *enhanced* Optimized **quicksort,** written by you as described in class-- including a method to choose a pivot and utilization of a secondary sort for sub-arrays smaller than four or eight (your choice)-- (useable by any array of **Comparable** objects), you will sort the array, and print it out in sorted order. Then, using the same array, allow the user to enter the same number of **Date** objects. Once again, sort and print the array.

Do not forget to prevent any memory leaks.

The program should be fully planned *in advance*. It should be properly documented, and work correctly. Do not forget to test multiple variations on possible input, and test a large enough array to properly test the **quicksort**.

Deliverables:

Physical:

The Project should be turned in inside a clear plastic file folder. This folder should have a simple flap to hold paper in place--NO buttons, strings, Velcro, etc. Pages should be in order, not stapled.

- Assignment Sheet (printed pdf from the web), with your name written on it, as a cover sheet.
- Printed Source Code with Comments *(including heading blocks -- a file header for each file plus a function header for each function. Describe parameters, any input or output, etc., no line wrapping). Print in portrait mode, 10 - 12 point font.*

Electronic:

- All .h, .cpp, .exe(Release Version) zipped together. Do not use rar or any archive format other than zip. Rename the file: "<YourName>_p1.zip".
- Sample Output (as .rtf -- run the program, copy the window using <*Alt*|*PrtScn*>, paste into Paint, invert colors *(*<*Ctrl*|*Shift*|*I*>), copy, open Wordpad, save.)
- A simple test plan including explanations of any discrepancies and reasons for each test. Show actual input and ALL values output as well as ALL expected output. Test each possible action. Save as .xls, xlsx, .doc or .docx file
- Submit this single zip file by going to canvas, select this class, select the Assignment tab on the left, select Assignment 1, select the submission tab at the top right, find the file, and Submit.

Due: Friday, April 29, 2016 9:30 a.m. ***(beginning of lab)***