MCIS-4135
Homework Assignment 6

Purpose:

This assignment will give you the opportunity to begin abstract data type programming in C++. The assignment is written a lot like a recipe, with the hope that you will admire the result, and review the steps that you took to get there.

This assignment in MCIS-4135 is very similar to an assignment that I use as the entry assignment for a full C++ course.

This assignment will illustrate how programs are built of multiple files in a programming environment.

This assignment will also explore ways of establishing encapsulation and data abstraction using multiple source files.

This assignment will also explore some C++ output formatting capability.

What to hand in:

Hand in listings of the program which must be captured in 3 or more files: a header file, a test program, and at least one implementation file. Also, hand in the output of a sample run (or runs) of the program. All of these must be formatted in a reasonable style. Choose sufficient test cases to ensure your functions work properly.

Problems:

Dating Skills:

1. Create a “project” in Visual C++ (or the equivalent in whatever environment you are using). We will create 3 files in this project, one header file called date.h, and two C++ source files, called date.cpp and hw6.cpp.
2. Create a header file (`date.h`) containing a proper definition for a `Date` (see the notes for one good date definition).Add the function prototypes to the header, even before you implement them. Put the appropriate standard `#ifndef/#define/#endif` directives in the header file, in the right places.
3. Add functions for setting and printing a `Date` (`DateSet and DatePrint`). These are further described in the Notes section below.
4. Create a test program (`hw6.cpp`) that uses the functions for setting and printing a Date. Add tests one at a time, building and running your program after each one. This may seem tedious, but really lets you build success on success, and progress on progress.
5. Add an implementation for each of the remaining functions in the header file into the implementation file (`date.cpp`). See the notes section for each implementation. Each time you add a function, go back to your test program (`hw6.cpp`) and add an appropriate test. This way, you will continue making progress on the exercise, one small step at a time. When you are done, there will be at least one test for each function.
6. Compile and run your final program, capturing the output for submission.

Notes:

The goal of the exercise is to think in object-oriented terms. Your Date *structure* serves the role of the *object* in Object oriented terms (and is declared in the `date.h` file):

```
   struct Date
   {
      int month;
      int day;
      int year;
   }; /* One implementation (there are others) */
```

One other aspect of header files is to protect the header against multiple inclusions. This is done by starting the file (date.h) with the lines

```
   #ifndef DATE_H
   #define DATE_H
```

and then ending it with the last line

```
   #endif
```

which makes the content of the file ignored should it be included a second time. Without this, there are situations that cause the compilation to fail because of multiple definitions of the same thing.

The *variables* of that type serve the roles of *instances* of the object:

```
   Date d2, d3; /* Found in the test main program (hw6.cpp) */
```

Thus, each of the functions should manipulate the instances of the object–they are the *messages* of the object-oriented paradigm. The message interface is defined in the `date.h` file (after the structure, but before the #endif):

```
   void DatePrint  (Date);
   int  DateDay    (Date);
   int  DateMonth  (Date);
   int  DateYear   (Date);
   Date DateSet    (int yr,  int mon, int dy);
   Date DateNow    ();
```

Note that this approach begins to isolate clients of your code from the internal details: where you might have had a piece of client code (in a piece of test code before we added DateSet) like:

```
         Date d1;
         d1.month = 3;
         d1.day = 31;
         d1.year = 1995;
         cout << "Printing March 31, 1995: ";
         cout << d1.month << '/' << d1.day << '/' << d1.year << endl;
```

which is dependent on your representation. You will now have code in hw6`.cpp` like:

```
         Date d1;
         d1 = DateSet (1995, 3, 31);
         DatePrint (d1);
```

Functions become simpler and smaller. The function `DateSet`, for example, in `date.cpp`, may resemble:

```
   Date DateSet ( int y, int m, int d ) {
      Date date;
      date.day = d;
      date.month = m;
      date.year = y;
      return date;
   }
```

Even though it may seem strange, providing simple functions to get (and sometimes set) the different components of a structure end up being very useful:

```
   int DateDay ( Date d ) {
      return d.day;
   }
   int DateMonth ( Date d ) {
      return d.month;
   }
   int DateYear ( Date d ) {
      return d.year;
   }
```

By continuing in this vein, one can even make sure that a program printing a date does not really care about how it is represented (here we are using "cout" from <iostream>):

```
   void DatePrint ( Date d ) {
      cout << d.month << '/' << d.day << '/' << d.year;
   }
```

Finally, the most complicated of these functions gets the current date to populate the structure. Time-oriented functions in the standard library are in the standard header **<ctime>**. You must include this header to use these data structures or functions. This header uses pointers, so you can look through this code to see how use of structures and pointers to structures works.

The function *time_t time(time_t *t)* returns the number of seconds since 1 Jan 1970 GMT. The function *struct tm *localtime(time_t *t)* takes (the address of) this result and converts it into a structure with tm_month, tm_day, tm_year, etc. as components. Though we won't use it in this exercise, it is interesting to note that the function *time_t mktime (tm*)* takes one of the time structures, completes it reasonably, and returns the equivalent seconds count.

A function to set the date to the current time may look as follows:

```
   Date DateNow ( ) {
      Date d = { 1, 1, 1970};
      time_t t = time(0L);            /* Stores the current time in t */
      tm *tp;                         /* In C this would have been struct tm */
      if (t == (time_t)(-1))
          return d;
      tp = localtime(&t);             /* Turns t into a tm structure */

      d.day   = tp->tm_mday;
      d.month = tp->tm_mon + 1;     /* Perversely, months are 0-11 */
      d.year  = tp->tm_year + 1900; /* The tm structure uses 2-digit years */
      return d;
   }
```

Evaluation Criteria:

Code will be evaluated both for proper style and for correct content. Your code should be in 3 (or more) files: date.h, date.cpp, and hw6.cpp (date.cpp could be subdivided into many other files if you choose).

30	Interface file: proper definitions and content, date type, compliant set of operations
35	Implementation file: proper use of C++ for implementation
35	Test program: full test of interfaces, including boundary conditions. Inclusion of results. ***Note: If your program doesn't compile, hand in the error listing as the results!***