MCIS-4130
Homework Assignment 3

Purpose:

This assignment will explore the issues of identifying objects in a problem statement and implementing small abstract data types, using the C++ struct..

The key is to have the data structures separate from the functions that manipulate them, as described below.

What to hand in:

Hand in descriptions of the objects requested with text or diagrams corresponding to *your* favorite object-oriented documentation techniques. UML, Booch, CRC, and all other methods are acceptable.

For the programming assignment, hand in a listing of the program (which must be formatted in a reasonable style), and sample run(s) of the program. Choose sufficient test cases to ensure your functions work properly.

Problems:

Think about time:

1. Consider the concept of a *Date*. This object will have several uses; the one we will keep in mind is for a ledger or billing system (the date is important, but not time of day).
2. Define the object, a set of logical attributes, and its responsibilities.
3. Document the object in a manner you have learned in the OO course.
4. Please implement at least the following functions related to a date:
    - *print the date instance*
    - *get (return) the day of the month from a date instance*
    - *get (return) the year from a date instance*
    - *get (return) the month from a date instance*
    - *set the date instance to a given value*
    - *validate the date instance (return true (1) or false (0))*
    - *set the date instance to "now"*

- *Optional: Add and subtract a fixed number of days to a date instance*
- *Optional: get the day of the week for a date instance.*

5. Implement and **test** this data type as a *struct*.
6. Be sure to create a header file that describes the type and at least one implementation file to provide the services. A separate file should provide the testing. .

Use *time(), localtime(), strftime()* or *asctime()* to add the idea of "now" in implementing *Date*. These are all expressed in <ctime>

Remember the goal is to implement a date type and operations to manipulate it!

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) */
```

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

```
Date d1, d2, d3; /* Mostly found in the test main program */
```

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:

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

Note that this approach entirely isolates the clients of your code from the internal details: where you might have had a piece of client code (in, say, `testmain.c`) like:

```
Date d1;
d1.month = 3;
d1.year = 1995;
d1.day = 31;
print ( "%d/%d/%d\n", d1.month, d1.day, d1.year);
```

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

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

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

```
int DateSet ( Date *dp, int m, int d, int y ) {
    dp->day = d;
    dp->month = m;
    dp->year = y;
    return DateValid(dp);
}
```

Some become trivial. For instance, DateDay is supposed to return the day of the month for this Date:

```
int DateDay ( Date *d)
{
    return d->day;
}
```

While identifying important objects and attributes, consider what techniques you've learned that make your life easier as a designer and implementor. Consider different methods of selecting objects (e.g., noun parsing, category prompts): which worked better for these very short problems?

The function *time_t time(time_t *t)* returns the number of seconds since 1 Jan 1970 GMT on Unix systems. 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. The function *time_t mktime (struct 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:

```
int DateNow ( Date *dp) {
    time_t t = time(0L); /* Stores the current time in t */
    struct tm *tp;
    if (t == (time_t)(-1))
```

```
            return 0;
        tp = localtime(&t);  /* Turns t into a struct tm */

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

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 testdate.cpp (date.cpp could be subdivided into many other files if you choose).

10	Reasonable model for the OO portion of the exercise
30	Interface file: proper definitions and content, date type, compliant set of operations
30	Implementation file: proper use of C++ for implementation
30	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!***