The **Chapter 11 lab is based loosely on two of the Programming Challenges** at the end of Chapter 11: **2) Day of the Year**; and **3) Day of the Year Modification**, both on page 768 (7ed), page 786 (8ed), page 810 (9ed), or page 816 (10ed). To make this more useful, it will be used in a calendar helper application. Startup code is provided.

See **class Day** in the startup code. You will expand on this class. This class has one and only one data member. This is an int number for day of year. The value 1 represents January 1, 2021. The value 365 represents December 31, 2021. The value 366 represents January 1, 2022. And on it goes.

The class is designed to avoid stale data. Read "Avoiding Stale Data" on page 400 (7ed), page 422 (8ed and 9ed) or page 426 (10ed) in the textbook. Robust, dependable code avoids the possibility of redundant, out-of-sync data.

**For the year, assume a BASE\_YEAR of 2021. That is, the year starts at 2021 for day-of-year 1. When the day-of-year exceeds 365, the day-of-year represents a day in following years. For example, if day-of-year is 366, that represents January 1, 2022. There is no need to store the year as a separate member of class Day, because the year information is stored inside the day-of-year. Have only one data member that contains values from 1 to 365 for January 1, 2021 to December 31, 2021 and beyond into 2022, 2023, etc. Leap years can be ignored. The calendar will be good for non-leap years 2021, 2022, 2023.**

There are several ways to construct a Day. By default, make a day with a value of 1 (Jan 1, 2021). A 2nd way to create a new instance of class Day is to provide a number for day-of-year. An easy way to set this number is to accept any positive number 1 or more. Do not let this go negative. You can assume the calendar starts at day 1 which is January 1, 2021 and does not go earlier.

A 3rd way is to construct a new instance of class Day is with a month name (the first 3 letters will do) and a number for a day in that month from 1-28, 1-30, 1-31, depending on the month.

**Day of the Year Modification.** Provide a 3rd constructor that accepts a string for the month and the day of the month.

Altogether, there are 3 constructors needed for class Day:

1) default constructor – day is Fri Jan 1, 2021  
2) constructor provided one int – day is set to that number, but must be 1 or more.  
3) constructor provided month name and month number – day is set to a single int number 1..365 (assumes year is 2021) by converting the month name and number into a number from 1 to 365.

The months of 2021, 2022, 2023 have the following days: Jan=31; Feb=28; Mar=31; Apr=30; May=31; Jun=30; Jul=31; Aug=31; Sep=30; Oct=31; Nov=30; Dec=31. One way to translate a month name and month number into the int day\_of\_year is to use a static const array daysInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Helpful is an array: string monthsOfYear[12]; This can be initialized to: {"january", "february", "march", "april", "may", "june", "july", "august", "september", "october", "november", "december"}; To keep is simple, a 3-letter month name will work: {"jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec"};

An enum type, such as: enum month\_t {jan, feb, mar, …} can make your code more self-documenting. For simplicity, you can use 3 letters for the month, although you can use full month names if you like. You can avoid uppercase, lowercase when searching for the month for greater flexibility.

Also useful are weekday names:   
const string Day::daysOfWeek[7] = {"sunday", "monday", "tuesday", "wednesday", "thursday", "friday", "saturday"}; or  
const string Day::daysOfWeek[7] = {"sun", "mon", "tue", "wed", "thr", "fri", "sat"};

**Use static member variables to store data common to all instances.** The YEAR, daysInMonth array, monthsOfYear array, and daysOfWeek array are common to all instances of class Day. You cannot initialize static data in the declaration of a class. (It may depends on the version of C++ and whether the data is const.) Initialize static data AFTER the declaration of the class, but BEFORE any instances of the class are created. See program 11-3 as an example of a program that uses static data. **Use "const static" for data which is constant and static. These arrays should be public and static and const. They should not be duplicated for each instance.** They can be accessed using the scope resolution operator, as in: Day:: . Because they are const, outside client / users cannot mess them up.

C++11/14/17 syntax allows you to initialize non-static data members when you declare them in your class. This is not available in earlier compilers.

You can use the Day constructors like this:  
Day first; // creates a Day with value of 1 which is Fri, Jan 1, 2021  
Day halfway(365/2); // creates an instance of class Day, called halfway, initialized to the day number 182  
Day birthday("jul", 4) // creates an instance of class Day, called birthday, value set to day of year for July 4, 2021

halfway.print(); // displays in “Weekday Month dd, yyyy” format. This will output something like: "Wed, June 30, 2021".

Also **write four overloaded operators (++day, --day, day++, day--)** so day of year can be incremented and decremented (may jump to the next or previous day). Testing example:  
Day cincoDeMayo("may", 5); // creates a day initialized to the number of the day for Wed, May 5, 2021.  
Day restDay = ++cincoDeMayo // increment to next day  
restDay.print(); // displays: "Thr, May 6, 2021"

To display the days to the screen, you need to convert the number into a more useful format for the user. For example, the day value 1 should be displayed as: Fri, Jan 1, 2021. The day value 365 should be displayed as Fri, Jan 31, 2021. The day of the week is easy. You know they are Sun, Mon, Tue, .. Sat, repeat. Given the day value number, divide by 7 get the remainder and adjust as needed. Knowing that Jan 1, 2021 was a Friday, all other days of the week can easily be determined. Similarly, for the year. For days past 365, the year increases by 1 for every 365 days past. The BASE\_YEAR is 2021. If the day of year is divided by 365 (and adjusted) you can get years after 2021: 2022, 2023, ... Do not worry about 2024, which is a leap year. Leap years take more work – a bit tedious.

In a loop, you can reduce the int day-of-year by month, until you have month number and day number. Needed is an array: string monthsOfYear[12]; This can be initialized to: {"january", "february", "march", "april", "may", "june", "july", "august", "september", "october", "november", "december"}; To keep is simple, a 3-letter month name will work: {"jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec"};

See program 11-11 for code that overloads operator++, operator--.

**CAUTION! Do not allow your day\_of\_year to become invalid! Day of the year below 1 is not valid! Decrementing day=1 should not allow day 0, -1, -2, -3, ... What to do? Leave it at the minimum supported day: 1**

**For testing:** Provide a menu-driving testing loop in main(). Allow the user to create days using the 3 constructors. Allow the users to see a single day, a week, or a month of days. Verify that valid data is accepted and invalid data does not crash the program or create invalid days. **Test all constructors, operators, getters, setters in class Day.**

**Invalid data should be rejected by the constructors.** If invalid data is detected in a constructor, it is common to “throw an exception”, which can exit the program. We don’t cover error-handling with exceptions (a big topic) in this class. To deal with invalid day (invalid month, day combination), set the day to a valid default. This could be simple, like January 1, or more intelligent, such as setting it to the closest valid date: Feb 31 becomes Feb 28; Jun -5 becomes Jun 1.

Test the four operators (++day, --day, day++, day--) by showing the previous five days and the following five days in expanded format. Example:

{

cout<<" start with most recently entered day: "<<day.to\_string()<<endl;

cout<<" (use +/-5 days with ++day, day++, --day, day--):\n";

const int RANGE=5;

for (int i=0; i<RANGE; ++i) {cout<<" "; (++day).print();} cout<<endl;

for (int i=0; i<RANGE; ++i) {cout<<" "; (day++).print();} cout<<endl;

for (int i=0; i<RANGE; ++i) {cout<<" "; (--day).print();} cout<<endl;

for (int i=0; i<RANGE; ++i) {cout<<" "; (day--).print();} cout<<endl;

}

cout<<" back to beginning: "; day.print(); cout<<endl;

**Extra credit:** (5%): Overload the << operator so when the class is sent to cout (ostream) it prints out the date in expanded format. Example:  
cout << cincoDeMayo; // prints out: "Wed, May 5 2021"  
Can use cout << testDay; as well as: testDay.print(); This makes it easier to output a Day. See code on page 719 (7ed) or page 737 (8ed) or page 745 (9ed) or page 759 (10ed). Hint: Use setw, stringstream, as desired.

**See the attached file: Lab11\_test\_output\_sample.txt for ideas on how to test your code.** If you don’t have time to handle multiple possibilities (such as: month name uppercase, lower-case, 3-letter month name, full month name, abbreviated, etc.) that is OK – just state your assumptions in comments in the code. In the test option, I included a few test cases to be sure I tested all class methods. Test cases of the various option are up to you. It is a good place to test both valid and invalid data. The various test cases should run without further user input in the t)esting option.

It is nice to customize the calendar application with your name in the introduction. See sample test output.