ECE118 LAB SIX

In this lab you will use clean structured design to solve a problem that is normally considered to be very difficult, and find that it is in fact surprisingly easy. *Look before you leap*: think about how your program is going to be organised, don't just start typing. A rational design will give you a working program quite easily; an unplanned design will not.

You will also be programming under Unix instead of Windows. This means that no graphical operations will be available, only plain text.

The assignment is to create a nicely formatted calendar for any month of any year.

Remember: the use of variables is still forbidden.

1. *Length of a Month*

Design a function that takes two parameters: year and month, and returns an integer indicating how many days long that month is. January 2016 was 31 days long, February 2015 was 28 days long, February 2016 is 29 days long, and so on. Remember that for February, leap years must be taken into account.

For this first part, we are only interested in the 21st century. Between the years 2000 and 2099 the leap year rule is very simple: a year is a leap year if it is divisible by four.

Incorporate your function into a simple program that allows you to test it conveniently, and then test it conveniently. Each stage of this lab assignment depends on the previous stage, so you won't do any good by going ahead with an incorrect function.

2. *Day of the Year.*

Design a function that takes three parameters: year, month, and day, and returns an integer indicating which day of the year that date is. For example, the 1st of January is day 1 of the year, the 2nd of January is day 2, the 1st of February is day 32, and so on.

Incorporate your function into a simple program that allows you to test it conveniently, and then test it thoroughly.

3. *Day of the Century.*

Now make a function that tells you what day of the century it is. Forget about foolish arguments about whether the century starts in 2000 or 2001. If you take 1st January 2000 as day 1, everything works out nicely. So 31st December 2000 was day 366, and 1st January 2001 was day 367, and so on. You still only need to be concerned with this century, 2000 to 2099.

4. *Day of Forever.*

You knew this part was coming. Now we want a function that again takes three parameters, representing year, month, and day, but this time, the year could be any positive number. This raises two issues: where to start counting (i.e. what date shall we choose to be day number 1?), and how to handle leap years.

Although pedantic folk will argue that there is no such thing as the year 0, pretending that there was makes for a very simple solution. Day 1 will be 1st January of the year 0 regardless of whether or not that date ever existed. It makes the counting easy.

The true rules for leap years are slightly more complex than just divisibility by four. The exact rules are given on the last page if you don't already know them, but in summary:

> Any year that is divisible by 400 *is* a leap year,
> any other year that is divisible by 100 is *not* a leap year,
> any other year that is divisible by 4 *is* a leap year, and
> any other year is *not* a leap year.

Here are some pre-calculated samples to help with testing:

1st January 2000 was day number 730486
1st January 1900 was day number 693962
4th July 1776 was day number 648857
4th to 10th October 2016 are days 736607 to 736613
27th November 2737 will be the millionth day
1st January of the year 10 A.D. was day 3654

Give some serious thought to testing. If you are getting the wrong number for a date, try some very close dates, and you are likely to spot a pattern in the error that will give you a big clue about where your program may be wrong.

5. *Day of the Week.*

Now make a function that takes year, month, and day as parameters, and tells you what day of the week that date was. **Let us consider here the work week where the week starts on a Monday.** Make your function return the answer as an int, using 0 for Monday, 1 for Tuesday, ..., and 6 for Sunday. That will make the next step slightly easier. This is an easy task if you think of the modulo `%` operator and remember how many days there are in a week.

Here are some pre-calculated samples to help with testing:

1st January 2000 was a Saturday
1st January 1900 was a Monday
4th July 1776 was a Thursday
4th and 10th October 2016 are Tuesday and Monday, as you probably know.
27th November 2737 will be a Saturday
1st January of the year 10 A.D. would have been a Friday.

The lab guys will show you the unix "`cal`" command which will help with testing.

6. *A Calendar for a Month.*

Use that function in a program that allows the user to enter two integers, representing year and month, and then prints a correctly formatted calendar for that day and month. The columns should be properly aligned (right justified), and use two letter abbreviations for the days of the week, starting with **Monday**.

Like this, which would come from an input of `2018 10`:

```
    October 2018
--------------------
Mo Tu We Th Fr Sa Su
--------------------
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
--------------------
```

You certainly know how to print out a list of numbers starting from 1. To make those numbers come out looking like a calendar, you need to work out how many spaces to print before the "1", and how to tell when it is time to start a new line. Take a little care to get the alignment of one and two digit numbers right.

And no taking the easy way out! The week must start on a Monday, and we want two-letter abbreviations for the days so that it is nicely spaced out. Turn in a screen shot showing your programs results and the results of the unix `cal` command for the same month, on the same screen.

Getting the month and year properly centred at the top is easy if you remember how to find the length of a string.

7. *A solid Product.*

Make sure that your calendar works for any year, not just in the 21st Century.

8. For A Little Extra Credit.

Write a function that works out how many Wednesdays any given year has.

9. For Extra Credit Only.

Transfer your program to a windows computer, and convert it to use the graphics library, so that the calendar displayed will look even nicer.

Only For Reference:

Rules for Leap Years

Under the Gregorian calendar system, which is what we use now, the rules for working out whether a particular year is a leap year or not are

> If the year number is divisible by 4, it is normally a leap year,
> except that if it is divisible by 100, it is not a leap year after all,
> except that if it is divisible by 400, it really is a leap year again.

So, the years...

1600, 2000, 2400, 2800 are leap years
1800, 1900, 2100, 2200 are just ordinary years
1904, 1908, 2004, 2008 are leap years
1901, 1999, 2001, 2009 are ordinary years.

The Gregorian calendar was only introduced in the English-speaking world and all the British colonies on 14th September 1752. Before that, the Julian calendar had been in use since roman times. Under the Julian calendar, a leap year is simply any year whose number is divisible by 4. The major European countries had switched to the Gregorian calendar in 1582, so there was a long period of international confusion.

Florida was a Spanish colony until 1763, British from 1763 to 1784, Spanish again from 1784 to 1810, independent from 1810 to 1811, then Spanish again, until it was finally taken over by the United States in 1821. There isn't much to be gained by trying to take all of those changes into account. Throughout the United States, the date for the change is taken to have been 14th September 1752 even in places where it wasn't really.

1752 was a very confusing and tempestuous year. All of a sudden 11 leap years that had contributed an extra 29th of February didn't count as leap years any more, and those 11 days had to be given back. The chosen solution was that the 3rd to 13th days of September just didn't happen that year. This is the correct calendar for the period:

August 1752

Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

September 1752

Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

October 1752

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

You do not need to make your calendar program take account of the Julian period in any way. You may just pretend that the current system has always been in place.