ECE 118 LAB NINE

This lab is all about data processing. You will be reading information from a file that contains thousands of individual pieces of data. There is far too much for a person to deal with in its numeric form, but you will display it in a way that is very easy to understand.

On the class web site, associated with this lab, there are 76 data files, each containing a whole year of weather observations from a different location in the U.S. Choose one of them and down-load it to the computer you are using.

The files all have one line of data for each day of a particular year, and each line consists of exactly nine numbers. But beware: this is real data from real meteorological stations. Sometimes their equipment isn't in perfect working order, so some days might not appear.

As an example of the data, here are three lines taken from near the end of the file for Mount Washington, NH.

```
2003 12 16 -17.7 -7.7 0.0 99 3 34.4
2003 12 17 -3.8 -1.1 1.1 99 -1 24.5
2003 12 18 -13.3 -8.3 0.0 130 53 36.6
```

The first three numbers on each line give the year, month, and date of the observations. The remaining numbers are:

4. The Minimum Temperature recorded on that day (Fahrenheit)
5. The Average Temperature recorded during that day (Fahrenheit)
6. The Maximum Temperature recorded on that day (Fahrenheit)
7. The Maximum Wind speed observed (miles per hour)
8. Total depth of snow-fall
9. Total amount of Precipitation, incl. snow melted

At least, I *think* that's what the columns represent. I may be wrong.
If you were wondering, their rain meter was broken on 17th December. That's what `-1` indicates in the last three items. Also, 99 appears a suspicious number of times as the wind speed, so that must also be an error, but there isn't much to be done about that.

1. *Make sure you can read the file*

Write a simple program that reads the whole file, and gives you just enough information to verify that it is working properly. Perhaps you could just print out the average temperature for every day, and check that it does generally get warmer in the first half of the year and colder at in the second.

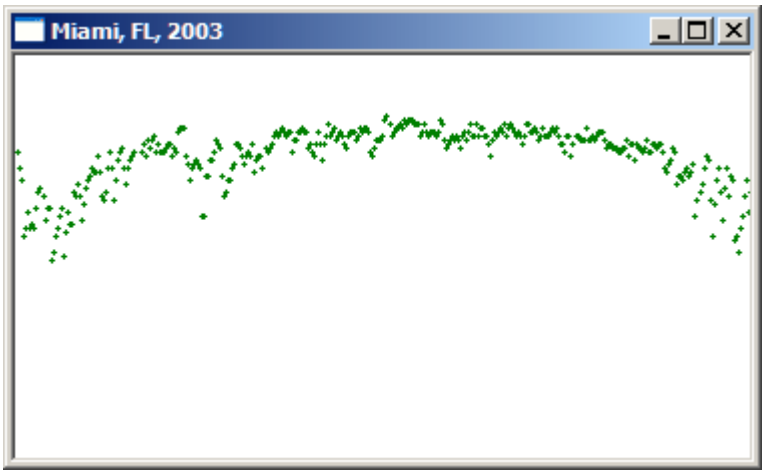
As a reminder, here is a little snippet of code that would open a file, read three numbers from it, then close it again.

```
ifstream fin("file.name");
if (fin.fail())
{ cout << "Can't read the file\n";
  Return; }
int a, b, c;
fin >> a >> b >> c;
if (fin.fail())
```

```
   cout << "Not three ints at the beginning of the file!\n";
else
   cout << "The average is" << (a + b + c) / 3 << "\n";
fin.close();
```

2. *Turn the numbers into a graph*

Adapt your program so that it displays all the average temperatures as points on a graph. Pick a reasonably large window and pen size so that it will look OK. The vertical position is of course just the average temperature multiplied by some suitable scaling factor, but what about the horizontal position?



Make it easy on yourself, just for now. If you pretend that every month is 31 days long, then `(month * 31 + day)` gives a simple basis for the x-coordinate, which you can scale or shift as desired.

The picture above was produced from the Miami FL file. If you want to set the caption of the window like I did, there is a handy little function called `set_caption`. It takes a string parameter.

3. *A good test.*

Download a few more of the weather data files, and make sure your program works just as well whichever one you choose. Don't keep re-editing your program to change the file name, instead use the special library function:

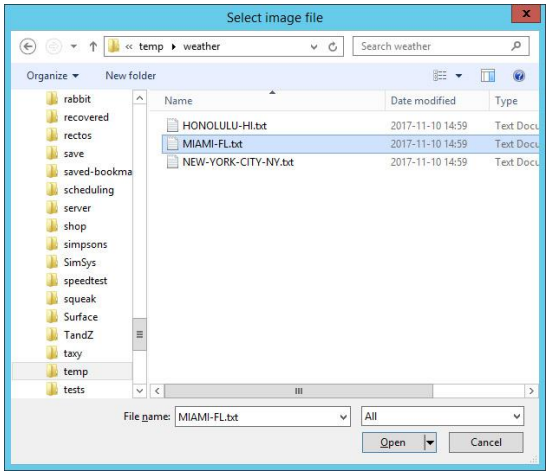
```
string open_file_pop_up("", string dir, string title)
```

The first parameter is an empty string, just a pair of quotes, not even a space between them.

The `dir` parameter is optional (it can be "", which means the current directory); it tells the pop-up which folder to start in.

The `title` parameter provides a title for the dialog pop-up window. I used "Select Image File" in the example below.

It opens up the standard Windows file-choosing dialogue.



The function returns as its result the name of the file you selected, ready to be used to create the `ifstream`.

4. *Get the X coordinate right.*

There aren't really 31 days in every month, and pretending that there are can cause annoying gaps. Fortunately, you recently had an assignment that involved working out what day of the year a particular date is. Make it so that the horizontal position of each dot is correct: the distance between 28th February and 1st March is the same as the distance between 1st March and 2nd March, or any other two consecutive days.

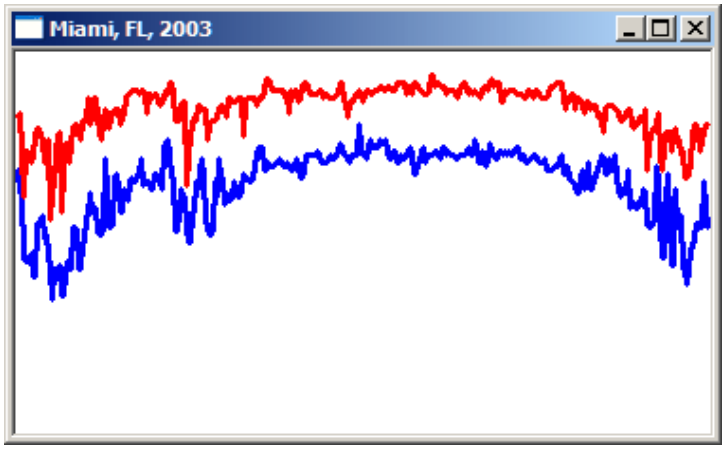
To keep the graph honest, when there are missing days you should not connect them with a line. Make sure that every valid observation is shown, just leave a gap for the missing ones.

5. *Less dottiness*

In the summer, the temperature tends not to change so much from day to day, so the dots almost merge into a solid line. In winter, as you can see, it is harder to see what is going on. There are dots all over the place. Make it into a line graph instead.

6. *More information*

When someone is planning a trip and deciding what to pack, knowing the minimum and maximum temperature is much more useful than just the average. Improve your program so that it shows a red line graph for the daily maximum temperature, and a blue line graph for the daily minimum. On the same axes of course.



This is one of those situations where the restrained use of a few variables can be really useful. Think a little about how to maintain two line graphs at the same time. It isn't difficult, but may not be immediately obvious.

7. *Make it useful*

It's a nice looking graph, but not really very useful. It's hard to tell just where the last week of October is, and nobody has any idea what the actual temperatures are: just how low was that dip in January?

Add some clearly labelled axes: vertical lines showing where each month starts, and horizontal lines for each ten degrees or so. If you draw them in gray, it won't seem too obtrusive.

This is where good clean programming pays off. If you were keeping things reasonably tidy, it will be easy to work out where the lines go, and to shift the graph to make room for the labels.

8. *Last thing before you go*

Make sure you have written a good robust program. Download a file for a different city, and check that your program displays its data correctly. It is important to make sure that your program works correctly even for data files that have a lot of missing data. One of the data files is for a made-up town called Hopeless, Missouri (at least, I

don’t think there’s a real town with that name, but I didn’t check). It was deliberately made with a lot of missing data to help you with testing.

9. Extra Credit

Adaptive scaling would be nice too. All my samples have assumed that temperatures will fit nicely into a 0 to 100 scale. That is untrue for a great many places. Your program could scan the file first to find out what the true range is, then re-read the data and plot it at a custom-made scale.