

Chapter 11: Reading in a Meteorological Data File

Each hour, meteorological surface observations are taken at hundreds of weather stations around the country. These observations are coded into the METAR format and broadcast across the internet to universities that use a computer system called the Local Data Manager (or “ldm”) to receive the data. Creighton is one such university; we receive several thousand METAR surface reports each day. The reports come in grouped such that each hour’s observations are together in one file for the whole country.

It would be great to be able to write computer programs that could work with the METAR data files. However, METAR is a fairly complicated coding system. While most METAR observations are fairly straightforward, there are all kinds of special codes and comments that can be added to a METAR, making it extremely difficult to write a computer program that can “parse” through a file of METAR observations and interpret these observations correctly.

Fortunately, we don’t have to do this, because others have done this for us already. There are several such programs available, and here at Creighton we use a program called “sacvt”, which is a part of the WXP software package. The sacvt program knows how to read METAR codes (and several other types of surface data codes), and converts the observations into a format that is more “regular”—meaning that there are no exceptions to the rules (unlike in METAR), no “special” codes (unlike in METAR) and no critical comments (again, unlike in METAR).

The sacvt program spits out a new file every hour, filled with the decoded observations. Your job, in this assignment, is to read through the current *_sao.wxp file, file a specific station, and print the current weather report for that station. Here is specifically how I want your program to behave:

1. Prompt the user for a 4-digit station identifier, such as “KOMA”.
2. Open the current decoded observations file.
3. Find the observation for the requested station.
4. Report the current temperature, dewpoint, wind speed and direction, and mean sea level pressure for that station.

The Format of *_sao.wxp Files

To understand how the decoded surface data files are structure, we will examine the first 24 lines of a typical file:

```
1  WXPSFC
2  0600Z 12 NOV 03
3  MHAM 752 752 0000 -99 -99 -99 25F,30F,700 - @0600 #CB LTNG NW $
4  MHCH -99 -99 -999 -99 -99 -99 -99M - @0600 $
5  MHTG 680 662 0306 7 -99 2.49 26B,80B !RA @0600 #RED 4 KMS TDS RA $
6  KU28 381 315 0503 -99 263 25.0 250S - @0550 P2 14 x51.3 #NOSPECI $
7  KENL 644 608 2408 987 -99 4.00 48S,60S,70B !BR @0546 #AO2 LTG DSNT E $
8  K1F9 716 626 2112 998 -99 10.0 -99C - @0546 G 20 #AO2 $
9  K11R 662 662 0000 9 -99 5.0 4S !BR @0545 #AO2 $
10 KGLE 698 644 1809 999 -99 10.0 -99C - @0545 #AO2 $
```

```

11 KJWY 698 644 2008 2 -99 10.0 -99C - @0545 #AO2 $
12 K1H2 644 608 2406 987 -99 5.0 29S !BR @0545 #AO2 LTG DSNT S $
13 KAAA 572 536 0000 984 -99 1.50 30 !BR @0545 #AO2 $
14 KASW 572 536 0000 985 -99 1.25 30 !BR @0545 #AO2 $
15 KBKS 698 680 1303 5 -99 3.00 -99M - @0545 #AO1 SKY OBSCURED $
16 KBPG 590 554 1708 9 -99 4.00 -99C - @0545 #AO1 $
17 KBWD 608 608 0000 6 -99 8.0 -99C - @0545 $
18 KBYY 680 680 1204 8 -99 4.00 33S !BR @0545 #AO2 $
19 KC09 464 464 0903 982 -99 0.25 10 !FG @0545 #AO2 $
20 KTIP 572 572 0000 985 -99 0.25 10 !FG @0546 #AO2 $
21 K4HV 421 320 0000 -99 -99 10.0 120B - @0550 P1 14 n42.1 #NOSPECI $
22 KEBG 680 644 1104 6 -99 2.50 -99M !BR @0545 #AO2 SKY OBSCURED $
23 KDKB 410 374 0000 978 -99 0.25 1B,70 !FG @0545 #AO2 $
24 KDNV 590 572 2803 982 -99 0.50 20 !FG @0545 #AO2 LTG DSNT SE $

```

The first line of any surface data file decoded by sacvt will always read “WXPSFC”. This is just a “header” that other WXP programs use to verify that the file contains surface observations. A computer science professor would tell you to your program always read this line of the file to verify that the file contains surface observations, but I’m not going to worry about that kind of detail. Rather, just use your SkipToEndOfLine function to ignore this line.

The second line of the surface data file will always be the Greenwich time and date of the observations that follow. It would be harmless to just skip this line as well, but I think that this is an interesting and important piece of information: you will probably want to print the time and date of the observations on your weather maps, so don’t throw this information away! Rather, read it into a 15-character string, which I’ll call “timecode” but you can call anything you want. Reading this particular string is a bit tricky, since the string contains spaces (which, as you will recall, are “tokens”). If you tried to read this string with a simple fscanf (fin, “%s”, &timecode) command, all you would get would be the first few characters, ending at the first space. Therefore, you will need to use the fairly specialized fscanf command shown below:

```
fscanf (fin, “%[^\n]s\n”, &timecode);
```

where timecode is declared by this statement:

```
char timecode[15];
```

Starting at the third line of the file, each line contains one decoded METAR observation. For the purposes of this course, only the first 6 values are important:

1. The station identifier
2. The temperature code
3. The dewpoint code
4. The wind code
5. The altimeter setting code
6. The mean sea level pressure code

Anything after these first six codes is not going to be used in this class. Therefore, it should be ignored—use your `SkipToEndOfLine` function to scan past this useless information once you have read the information you need on each line.

The Station Identifier

The first entry on each line of the decoded METAR files is the station identifier. This entry is a string of characters. All of the identifiers shown in this example are four digits long, but some reports do come in from stations that use 5-character identifiers (these are mostly buoys and ships). Therefore your variable that stores the station identifier should be declared to be five characters long:

```
char id[5];
```

The station identifier is an example of a “string”—a collection of characters that doesn’t represent a number but rather a word. Working with strings is a little different than working with other variables. For example, to set string A equal to string B, use this code:

```
strcpy (A,B);
```

(The `strcpy` function stands for “string copy”). To check if two strings are equal, use this code:

```
if (!strcmp(A,B)) {  
    printf ("The two strings are the same.\n");  
}
```

Here, `!strcmp` will return “true” if A and B are the same; otherwise, it will return false.

The Temperature and Dewpoint Codes

The temperature and dewpoint for the station are coded as integers. The values are expressed in tenths of a degree Fahrenheit. Therefore, a code of “754” means that the temperature is 75.4°F. You will need to read these values in as integer codes, and then calculate the correct temperature and dewpoint from these codes. Here’s a hint about how to do this: you can convert an integer to a float with an equal value by “type casting” it as a float:

```
float x1;  
int x2;  
  
x1= 100;  
x2 = (float) x1;
```

In this little example, x1 is an integer equal to 100. The float variable x2 is then equal to the floating point number 100.0, thanks to the “(float)” cast.

The Wind Code

The wind is stored as a 4-digit integer. The first two digits are the wind direction in tens of degrees, and the last two digits are the wind speed in knots. Therefore, a wind code of “0817” means that the wind is from 80° (i.e., just north of east) at 17 knots.

Decoding the wind observation is a little bit tricky, so I will give you this decoding function:

```
void DecodeWind (int windcode, float *speed, float *dir) {  
  
    *speed = (float) (windcode % 100);  
    *dir = (float) (windcode - (windcode % 100) ) / 10.0;  
  
}
```

The Altimeter Setting Code

Altimeter setting is an important meteorological observation for pilots, but not something the meteorologists themselves use very often. Therefore, your program should read in the altimeter setting code for each observation, but we won’t decode this observation for this class.

The Mean Sea Level Pressure Code

The Mean Sea Level Pressure (MSLP) for a station is encoded in the *_sao.wxp file just as it is on a plot of station models. The MSLP is written as a three digit number, and the meteorologist is expected to know that you need to add either a “9” or a “10” in front of the code, and move the decimal point one place to the left to obtain the MSLP in millibars. Therefore, a code of “985” means that the MSLP is 998.5 mb, while a code of “056” means that the MSLP is 1005.6 mb.

How do you know (more to the point, how does your computer program know) whether to add a “9” or a “10” to the observation? In your Introduction to Atmospheric Sciences Lab, you probably learned that you add either a “9” or a “10”—whichever makes the MSLP closer to 1000mb. For example, a code of “985” could mean either 998.5 mb or 1098.5 mb. Since 1098.5 mb is unrealistically high, you know to decode this as 998.5 mb. In your computer program, you can do this by checking whether the original MSLP code is greater than or less than 500. If the code is less than 500, you are going to add a “10” in front of the number; otherwise, you add a “9”.

Reading All of the Observations

Your *_sao.wxp file will contain thousands of observations. The number of observations will vary slightly from hour to hour, as some stations don’t report every hour. Therefore, your program will need to know how to read to the end of the file and

then stop without generating any errors. This will be a job for the “while (!feof(fin))” loop structure.

The feof Function

When you read a file, it is important to not try to read beyond the end of the file. Any number of strange and spectacular errors can occur if you try to read beyond the end of the file. To prevent this, we normally know in advance how much data is in the file, and we just read *that much* information. For example, perhaps you know that a given file contains 100 observations. In this case, you would fopen the file, read 100 values, and fclose the file, with no danger of reading past the end of the file.

But often you don’t know how much information is going to be in a file. These decoded METAR files are a good example—you don’t know how much information is in any given hour’s file, and the number will almost certainly be different for the next hour. Therefore, we commonly use a function called feof (for “file—end of file?”) in C. Consider the following code:

```
while (!feof(fin)) {  
    fscanf (fin, "%s %d %d", &id, &tcode, &tdcode);  
    fscanf (fin, "%*[^\\n]\\n");  
}
```

The “while” statement means that the computer should everything between { and } as long as whatever is inside the parentheses is true. The feof(fin) function returns “true” if you are at or beyond the end of the file; otherwise, it returns “false”. The exclamation point before the feof means “not”. So “while (!feof(fin))” literally translates as “while not the end of the file fin”. As long as you are not at the end of the file, execution of the loop will continue. If you are at or beyond the end of the file, execution will move on from the end of the loop.

Assignment 8: Reading meteorological observations

Prompt the user for a four-digit station identifier. Then read through the current observations until you find the report from this station. Print out the current temperature and dewpoint, wind speed and direction, and mean sea level pressure for this station.

Completing the assignment shown is worth 85%. Higher grades will be earned by students who impress me.

You can impress me on this assignment in many ways:

- After printing the output, give the user a chance to either enter another station identifier or end the program.
- Use your meteorological functions in `metcalc.c` to compute some derived variables for the weather at the current station for the current hour.
- Make the program end gracefully if the station requested doesn't exist.

This assignment is worth 10 points.