Introduction to C - Programming Assignment #3

Due date: October 14, 2011, 11:59pm

Objectives

1. To review using loops for repetition.
2. To learn how to read (large amounts of) information from files.
3. To learn how to use arrays to store many pieces of information.

Problem A: Weather Planning (weather.c)

Arup plans on having an outdoor wedding. However, as everyone knows, outdoor weddings are contingent upon the weather. You've decided that you'd like to help Arup and others planning an outdoor wedding by writing a computer program that gives historical data about the average temperature on the day and month of the planned wedding.

You will be reading files from the following website that contains average daily temperature data for many cities:

http://academic.udayton.edu/kissock/http/Weather/default.htm

Your program will ask the user for their wedding day (month, day) and the file that stores the city in which they will get married. (For example, the file name for Orlando is FLORLAND.txt.)

Based on this information, output the following pieces of information:

a) The average temperature on the given wedding day in each of the years listed in the file.

b) The average temperature on the given wedding month in the years listed in the file.

Finally, if this temperature is in between 65 and 75 degrees Fahrenheit, print out your recommendation that the date and city planned are viable for an outdoor wedding. If it isn't recommend to move the wedding indoors!

How to read in a string from the user

A string variable is declared as follows:

```
char filename[20];
```

A string is a special type of character array. The string above can store any string of characters of length 19 or less. This should be plenty for this assignment.
To read in a string from the user into the variable filename do the following:

```
scanf("%s", filename);
```

You'll notice that the percent code for strings is %s and that there's no ampersand in front of filename. This is because filename (and all arrays) are actually memory addresses, so what the user enters is read in naturally to the correct memory locations.

Now, once this variable stores the name of a file, we can open the file as follows:

```
FILE* ifp = fopen(filename, "r");
```

No double quotes are needed because we now have a string variable, as opposed to a string literal. filename will evaluate to whatever the user entered, and that file will get opened.

Input Specification

Each input file contains one day of data per line. Each line contains four pieces of information: month (1 – 12), day (1 – 31), year(1995 – 2011) and temperature. The first three values are integers and the last is a real number representing the average temperature in Fahrenheit on the specified day. If no temperature was recorded, -99 will be listed instead. Do NOT count these values are real temperatures and simply skip over them!!!

You are guaranteed that each day will have at least one valid temperature reading. (Thus, no matter what day you're getting married, there will be at least one recording for that day stored in the file.)

Output Specification

The first line of output should be of the following format:

```
The average temperature on your wedding day is X degrees F.
```

The second line of output should be of the following format:

```
The average temperature on your wedding month is Y degrees F.
```

where both X and Y are printed out to two decimal places.

If **both of these values** are in the range [60, 75], then output the following line as the third line of output:

```
The weather looks good for an outdoor wedding!
```

Otherwise, make the third line of output:

```
It's probably best to move the wedding indoors, sorry.
```

Output Sample

Below is one sample output of running the program. **Note that this sample is NOT a comprehensive test.** You should test your program with different data than is shown here based on the specifications given above. In the sample run below, for clarity and ease of reading, the user input is given in *italics* while the program output is in **bold**. (Note: When you actually run your program no bold or italics should appear at all. These are simply used in this description for clarity's sake.)

Sample Run

What is the month and day of your wedding?
3 3
What file stores your city's temperature data?
FLORLAND.txt

Sample Output

```
The average temperature on your wedding day is 64.15 degrees F.
The average temperature on your wedding month is 66.46 degrees F.
The weather looks good for an outdoor wedding!
```

Note: These files are continually updated. This output is for the file with the last day of data on 9/14/2011. If you download the file later, you may get slightly different values. Also, an input of 5 followed by 22 on the first question represents May 22nd.

Problem B: Scheduling Problems (schedule.c)

Since Arup enjoys many different activities, he often overbooks himself. In the past, this wasn't a problem. Either he would simply go to parts of two events that coincided, or he'd simply call the person hosting one of the two events telling them he couldn't make it. Normally, this didn't cause a problem because none of his appointments were ones that he absolutely had to keep. But now that he's getting married, his fiancée occasionally has very important tasks for him that he can't simply call to get out of, if he has double-booked.

In this problem you will help Arup identify schedules that contain double booking and schedules that don't. Though this problem can be solved in many ways, you'll be asked to solve it in a specific manner.

You'll be given a set of responsibilities/events Arup has over the course of a week. Your goal will simply be to determine if there are any overlapping events at all.

In particular, you'll be given several weekly schedules. Each weekly schedule will be a list of events. Each listing of events will contain the day (0 – 6, where Sunday = 0, Saturday = 6), a start hour (0 – 23, where 0 is midnight, 23 is 11 pm), and an end hour (1 – 24, where 1 is 1am and 24 is midnight of the following day.) It is guaranteed that the end hour is greater than the start hour, so each event has a duration that is a positive integer number of hours.

Restrictions for Solution

In order to solve this problem use an integer array of size 168 (the number of hours in a week). Each slot in the array stands for one hour in the week. Assume that the week starts on Sunday. Thus, indexes 0 through 23 represent the 24 hours on Sunday. For example, index 0 represents midnight to 1 am and index 20 represents 8pm through 9pm. Similarly, indexes 24 through 47 represent Monday. The array should store all 0s (false) to indicate that the entire schedule is free before any of the data is read in.

When you read in each event for the week, you must go to the appropriate slots in the array for that time period and see IF they are already scheduled, which would indicate a conflict. If they are, you know a conflict exists. If not, set each of these indexes to 1 (true) to indicate that the time period is now taken. For example, if an event runs from 5pm – 8pm on Tuesday, then indexes 65, 66 and 67 should be set to 1.

Input Specification (schedule.txt)

The first line of the input file will contain a single positive integer, *n*, representing the number of schedules in the input file to check. The schedules will follow. The first line of each schedule will contain a positive integer, $E\ (< 30)$, representing the number of events scheduled for the week. On each of the following E lines a single event will be listed for that schedule. In particular, each of these lines will have three integers: $d\ (0 \le d < 7)$, $s\ (0 \le s < 24)$, and $e\ (s < e \le 24)$, representing the day, start hour and end hour of the event, respectively.

Output Specification

For each schedule, output a single line of the following form IF the schedule has no conflicts:

```
Schedule #k: Good job, no conflicts!
```

If there is at least one conflict, output a single line of the following form:

```
Schedule #k: Sorry, you double booked yourself again.
```

where $k\ (\leq n)$ represents the schedule number, starting at 1.

Note: PLEASE FOLLOW THIS SPECIFICATION EXACTLY. IT WILL MAKE GRADING GO MORE SMOOTHLY!!!

Sample Input File

```
2
10
0 8 12
1 8 12
2 8 12
3 8 12
4 8 12
5 8 12
6 8 12
3 12 15
4 20 22
5 18 20
3
1 13 18
2 8 17
1 17 20
```

Corresponding Output

```
Schedule #1: Good job, no conflicts!
Schedule #2: Sorry, you double booked yourself again.
```

Problem C: Resolving Scheduling Problems (schedule2.c)

Ultimately, the only way to resolve the scheduling problems is to put different priorities on each activity and schedule activities from most important to least important. If a less important activity conflicts with a previously scheduled activity, you simply don't do it.

In this program, you'll take the same input as the previous problem, assuming that the order of events in the file is in the order of importance of those events. Thus, you'll put in the schedule every event that fully fits at the time you try to schedule it. Once an event is placed in the schedule, no event that conflicts with it may be placed in the schedule.Once the schedule has been processed, you'll calculate the total number of scheduled hours (out of 168) for that week.

Output Specification

For each schedule, output a single line of the following form:

```
Schedule #k contains X hours of scheduled activity.
```

where X is a positive integer less than or equal to 168, representing the number of hours that are scheduled for the given week, and $k (\leq n)$ is the schedule number starting at 1.

Corresponding Output (for original input file in part B)

```
Schedule #1 contains 35 hours of scheduled activity.
Schedule #2 contains 14 hours of scheduled activity.
```

Deliverables

Three source files: *weather.c*, for your solution to problem A, *schedule.c* for your solution to problem B and *schedule2.c* for your solution to problem C. All files are to be submitted over WebCourses.

Restrictions

Although you may use other compilers, your program must compile in gcc and run in the Code::Blocks environment. Each of your three programs should include a header comment with the following information: your name, course number, section number, assignment title, and date. Also, make sure you include comments throughout your code describing the major steps in solving the problem.

Grading Details

Your programs will be graded upon the following criteria:

1) Your correctness

2) Your programming style and use of white space. Even if you have a plan and your program works perfectly, if your programming style is poor or your use of white space is poor, you could get 10% or 15% deducted from your grade.

3) Compatibility to gcc in Code::Blocks. If your program does not compile in this environment, you will get a **sizable** deduction from your grade.