EEN118 LAB TEN

In this lab, you will be performing some important data-processing operations, specifically sorting a large database file. Sorting data is a very important operation in computing for many reasons. One of those reasons is that it makes the data more accessible to humans once it is printed (imagine trying to use a telephone directory in which the names do not appear in any particular order). Another reason is that it makes the data more quickly searchable by the computer.

There are many large data files to use for this lab, but you will only need the first one until you get on to the advanced parts. They are all available on the class web-site, and are named data1.txt, data2.txt, data3.txt, and so on.

You do not need to download the files when you are working on rabbit, as they are already there if you know where to look. The first one's full file name is

```
/home/www/class/een118/labs/161/data1.txt
```

Important Note

This lab is to be run under a Unix system, not windows. You must also use only the standard C++ and Unix library files. Do not `#include library.h`. Try to remember what the standard includes are, but if you can't remember, the lab guys will remind you.

Look at the file "`data1.txt`" with a text editor. You will see that it contains data about a number of people. Each line contains exactly six items: a person's social security number their date of birth, their first name, their last name, their state of residence, and their bank balance. Here is a sample from the middle of the file:

```
114680858 07061967 Matilda Vincent MI 6050.75
114930037 24101947 Desdemona Hanover ID 2440.47
115550206 10011979 Xanadu Perlman ND 14997.39
116520629 21091963 Alexander Hall SD 756.15
117050976 16101930 David Lamprey GA 142.60
119610646 02021965 Thomas Porlock IL 383.11
120330928 26111962 Cary Cartman NC 9813.81
122460462 11041962 Bella Oldman SD 3186.90
123040628 13021922 Elizabeth Watson NC 18971.91
123580905 08031923 Gustav Hornswaggle MN 4852.70
125040813 13061984 Godfrey Tumor OR 16760.20
125610677 03091958 Gustav Trentham IA 89.12
126470499 19121952 Justin Oddly MA 66038.10
127700250 16061930 Ursula Farnes LA 18551.18
129540334 14111979 Betty Eaton NH 9642.10
130020412 14111936 Maggie McIntosh NV 1542.95
132680826 18111963 Raul Kringle NJ 85179.60
135040001 27041949 Arthropod Gravedigger ID 400.47
135590854 12101956 Aloysius Pornman MO 535.70
136870683 24121952 Alexandra Hanover LA 666.22
138350783 18051924 Tammy Iverson OK 10241.35
139160698 17071973 Norbert Pringle NH 118443.74
139310201 24121973 Wotan Limburger AR 842.13
```

As you may have noticed, the date of birth is provided as a single integer, in the format `ddmmyyyy`, which is not a very common format round these parts, so you'll have to keep it in mind. Matilda Vincent was born on the 7th of June 1967.

The `1` in the filename `dbfile1.txt` indicates that it contains exactly one thousand lines.

1. *Read the Data*

Write a program that creates an array large enough to hold all the data, then reads all the data from the file into that array. Of course, it will have to be an array of `struct`s that you will also need to define. Make your program close the file, then print out the first 10 items of data from the array, so that you can make sure everything was read correctly.

2. *Basic Search*

Make your program ask the user to enter a name. It should then search through the data in the array (don't read the file again), finding any entry with a matching name. Correct matches for either first or last name should be accepted. For *every* matching entry that is found, print out all six data items: the social security number, first and last names, date of birth, state, and bank balance for each matching person.

Remember that if you use the `==` operator to compare strings, the test is case-sensitive. The user (i.e. you) will have to type the name exactly correctly, with capital letters in the right places. If you are feeling adventurous, make a case-insensitive string comparison function.

Important: Good clean design will make this lab much easier. Write a separate function that searches the array, do not put all the work in main.

3. *Find the Extremes*

Modify your program so that after closing the file, instead of printing the first ten items of data, it searches through *all* of them to find the richest and poorest people represented. It should all the data for the people with the highest and lowest bank balances.

Important: As for part two, good clean design will make this lab much easier. Write a separate function that searches the array to find the poorest person, do not put all the work in main. A quick copy of that function with a very small change will give you a function for finding the richest with almost no effort.

4. *Redistribution of Wealth*

For some unfathomable reason, the management wants the poorest person to occupy the first position in the array. Modify your program so that after finding the poorest person, it swaps his or her data with the data already occupying the first position in the array.

5. *Now Promote the Second Oldest*

The management has now decided not only that the poorest person must occupy the first position in the array, but also that the second-poorest person must occupy the second position in the array. So, after searching for the poorest and moving their data to the front of the array, now search the remainder of the array (all except the first element), and move the poorest

person you find (which must be the second poorest of all) into the second position of the array. Make sure you swap data, so that whoever was originally in the second position is not lost.

6. *More of the Same.*

The management are going to keep on adding requirements like this, next putting the third-poorest in the third position, then the fourth, then the fifth. There is no knowing when they will grow out of this petty obsession, so make things easier for yourself. Modify your search function so that it can be told how much of the array to search. That is, give it two `int` parameters (let's call them `a` and `b`); its job is now to search only the portion of the array between position `a` and position `b`, to find the poorest person therein. This makes it very easy to search the remainder of the array to find the second and third poorest.

7. *The Ultimate Demand.*

Now the management make their final demand. You are to repeat the process of moving the nth-poorest person into the nth position 1000 times. (remember, 1000 is the number of data records in the whole file).

This will result in the array being completely sorted. Do it, and check that it worked. Make your program print the contents of the array after it has finished. Look at the output to make sure that everyone is printed in order of their bank balance.

8. *Sorting the File.*

Once you have sorted the contents of the array, it might be a good idea to save the sorted data in a file. Make your program create a new file, and write all the contents of the array into that file. Use a text editor to look at the file and verify that it has the same format as the original file, and all the data is properly sorted.

9. *How Fast Is It?*

It is important to know how long computer operations are going to take when they have to work on a large amount of data. The standard Unix functions that give accurate timing are a little mysterious, so here is a little function that you can just copy and paste into your program. It requires two extra library files to be included, they are:

```
#include <time.h>
#include <sys/resource.h>
```

Here is the function

```
double get_cpu_time()
{ struct rusage ruse;
  getrusage(RUSAGE_SELF, &ruse);
  return ruse.ru_utime.tv_sec+ruse.ru_utime.tv_usec/1000000.0 +
         ruse.ru_stime.tv_sec+ruse.ru_stime.tv_usec/1000000.0; }
```

It returns the time as a double, and is accurate to a couple of milliseconds.

Use this function (twice - think about why) to time how long it takes the computer to sort the entire array. Do not include the time it takes to read the file or the time it takes to write the new file, just the pure sorting time. Note the time that you observe.

Now you know how long it takes to sort a database of 1000 items. How long do you think it would take to sort a database of 2000 names? 3000 names? 10,000 names?

Think about those questions, and work out what you believe the answer is. *Then* find out what the real answer is. The other files have exactly the same format as data1.txt, but are longer. `data`(N)`.txt` contains N thousand data records. If your program was nicely written, it will be a few seconds' work to change the array size and make it read a different file.

See how long it takes to sort these larger files, and compare the results to your predictions. If your predictions weren't substantially correct, make sure you understand why. You have just demonstrated a very important phenomenon of computing.