ECE 118 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 not need to down-load them, the files can all be opened directly because you will be working with Unix on rabbit again.

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 "`/home/118/database1.txt`" with a text editor. You will see that it contains data for a number of people. Each line contains exactly seven items: a person's first name, last name, social security number, date of birth (day, month, year), and their secret password. The seven items are separated by spaces, but no item will ever contain a space. Here is a sample from near the beginning of the file:

```
Matilda Vincent 114680858 7 6 1967 2nFeLvBa
Desdemona Hanover 114930037 24 10 1947 wqxEgFJg
Xanadu Perlman 115550206 10 1 1979 KGiXG4gk
Alexander Hall 116520629 21 9 1963 lT8eVxFN
David Lamprey 117050976 16 10 1930 KaPjtehN
Thomas Porlock 119610646 2 2 1965 tEix9RDJ
Cary Cartman 120330928 26 11 1962 lLuRDBZM
Bella Oldman 122460462 11 4 1962 iXQ3ilEQ
Elizabeth Watson 123040628 13 2 1922 jjZDl6gE
Gustav Hornswaggle 123580905 8 3 1923 Ao1y3t5e
Godfrey Tumor 125040813 13 6 1984 4qTwPI8V
Gustav Trentham 125610677 3 9 1958 uNMCWgj4
Justin Oddly 126470499 19 12 1952 n6xw3B1r
Ursula Farnes 127700250 16 6 1930 OQPF9JI2
Betty Eaton 129540334 14 11 1979 ZroC0kni
Maggie McIntosh 130020412 14 11 1936 XXKI1RCe
Raul Kringle 132680826 18 11 1963 OXAHopY0
Arthropod Gravedigger 135040001 27 4 1949 4Cu3b2TY
Aloysius Pornman 135590854 12 10 1956 j7o7gzf1
Alexandra Hanover 136870683 24 12 1952 mOzdv1f9
Tammy Iverson 138350783 18 5 1924 ODG1TtPN
Norbert Pringle 139160698 17 7 1973 T9ESZoE9
Wotan Limburger 139310201 24 12 1973 jS4ph8ht
Bella Fimburg 139480872 5 12 1961 MEu5wz58
Lillian Morlock 139620582 12 6 1929 2zXJQBmx
```

The real file contains exactly 1000 lines of data.

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. Make it be an array of `structs`, don't have seven separate arrays. You will need to define those structs. Make your program close the file, then print out the first and last 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 with either first or last name should be accepted. For every matching entry that is found, print out all seven data items, everything that is known about 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. That isn't good. Create a case insensitive string comparing function and use that instead.

Important: Good clean design is a requirement. Write a separate function that searches the array, do not put all the work in main.

3. *Find the Oldest*

Modify your program so that after closing the file, instead of just printing the first ten items of data, it searches through 1000 of them all to find the oldest person represented. It should print all seven items for the oldest person. In each file, the oldest person has a fairly distinctive name, so you'll have a good clue that you've got it right.

Important: As for part two, good clean design is necessary. Write a separate function that searches the array to find the oldest person, do not put all the work in main.

4. *Promote the Youngest*

For some unfathomable reason, the management wants the youngest person to occupy the first position in the array. Give yourself another function, just like the previous one, that finds the youngest person in the array. Once you know where the youngest person is, you can swap his or her data with the data already occupying the first position in the array. That way nothing is lost. Remember that the first position in an array is numbered zero, not one.

5. *Now Promote the Second Youngest.*

The management has now decided not only that the youngest person must occupy the first position in the array, but also that the second-youngest person must occupy the second position in the array. So, after searching for the youngest 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 youngest person you find (which will necessarily be the second youngest 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-youngest 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 promotion 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 youngest person therein, and swap that youngest with whoever is at position `a`. This makes it very easy to search the remainder of the array to find the second and third youngest.

7. *The Ultimate Demand.*

Now the management make their final demand. You are to repeat the process of moving the nth-youngest 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 increasing age. Of course you are not expected to inspect all 1000 lines, but don't just look at the first few either.

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 in a sensible format. 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 roughly 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()
{ 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) to time how long it takes the computer to sort the array of 1000 data items. 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 1,000 items. How long do you think it would take to sort a database of 2,000 names? 3,000 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 `database1.txt`, but are longer. `Database`*`(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. These are the files that are available:

```
/home/118/database1.txt
/home/118/database2.txt
/home/118/database3.txt
/home/118/database5.txt
/home/118/database10.txt
/home/118/database20.txt
/home/118/database30.txt
/home/118/database50.txt
/home/118/database100.txt
```

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. Include your observed times and conclusions in the document that you submit.