ECE118 LAB ELEVEN

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/www/students/lokesh/labs/data/data1.txt`" with a text editor. You will see that it contains data for a number of people. Each line contains exactly eight items: a person's first name, last name, name, date of birth (year, month, day), the state code of their address, their social security number, and their bank balance. The eight items are separated by spaces, but no item will ever contain a space. Here is a sample from near the beginning of the file:

```
Elephantina Dunstan 1942 11 13 AE 112981687 56092.90
Oswald Bradford 1978 11 25 NH 112989380 76853.34
Carl Ventnor 1944 11 20 GA 112999653 87428.20
Ellery Mozart 1966 03 03 VT 113009041 41196.27
Tom Squire 1973 02 28 RI 113014434 78490.28
Carl Farnesbarnes 1980 04 12 WY 113017336 48296.86
Francis McIntosh 1963 01 23 IN 113023856 77323.85
Camilla Pornman 1972 05 14 OR 113024409 30203.97
Nancy Quire 1982 01 08 MT 113024683 7178.54
Vincent Kelvin 1967 03 14 CT 113030761 29378.97
Carol Ipswitch 1961 12 21 AL 113033037 32971.03
Quincy Pearson 1954 04 18 NH 113039263 44269.07
Ashley Anstey 1980 04 30 ND 113039683 98355.99
Florence Goodman 1940 09 18 NE 113040616 64908.59
Charles Davis 1954 09 14 HI 113043036 88600.91
Nicholas Yapp 1979 10 31 TN 113046720 34131.63
Daphne Holler 1948 03 02 WV 113059571 80998.43
```

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. Of course, it will have to be an array of `structs` 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 with either first or last name should be accepted. For every matching entry that is found, print out all eight 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. You may use .compare function, but even that is also case-sensitive comparison.

Now add a new function that compares names irrespective of the case. Users can input the name as they desire irrespective of the capital or smaller case. Your program should perform a case insensitive search.

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 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 eight 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 will make this lab much easier. 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 a 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 must 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 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 youngest person therein. 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 prints 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.

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 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) 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/www/students/lokesh/labs/data/data1.txt
/home/www/students/lokesh/labs/data/data2.txt
/home/www/students/lokesh/labs/data/data3.txt
/home/www/students/lokesh/labs/data/data5.txt
/home/www/students/lokesh/labs/data/data10.txt
/home/www/students/lokesh/labs/data/data20.txt
/home/www/students/lokesh/labs/data/data30.txt
/home/www/students/lokesh/labs/data/data50.txt
/home/www/students/lokesh/labs/data/data100.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.

9. *Bonus Points*

Now take the input file ***"/home/www/students/lokesh/labs/data/data100.txt"*** , and create a report , which contains the details of person who has the highest bank balance, from each state.

The output should contain as following fields in the format below,

```
STATE   FIRSTNAME   LASTNAME   BANKBALANCE

NY Anne Klein 99925.59

FL Heather Jameson 99892.39

CA Roscoe Epstein 99900.38
```