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 database1.txt, database2.txt, database3.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/database1.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 "`database1.txt`" with a text editor. You will see that it contains data about a number of people. Each line contains exactly five items: a person's social security number, their date of birth, their first name, their last name, and state of residence. The five items are separated by spaces, but no item will ever contain a space. Here is a sample from the middle of the file:

```
114680858 19670607 Matilda Vincent MI
114930037 19471024 Desdemona Hanover ID
115550206 19790110 Xanadu Perlman ND
116520629 19630921 Alexander Hall SD
117050976 19301016 David Lamprey GA
119610646 19650202 Thomas Porlock IL
120330928 19621126 Cary Cartman NC
122460462 19620411 Bella Oldman SD
123040628 19220213 Elizabeth Watson NC
123580905 19230308 Gustav Hornswaggle MN
125040813 19840613 Godfrey Tumor OR
125610677 19580903 Gustav Trentham IA
126470499 19521219 Justin Oddly MA
127700250 19300616 Ursula Farnes LA
129540334 19791114 Betty Eaton NH
130020412 19361114 Maggie McIntosh NV
132680826 19631118 Raul Kringle NJ
135040001 19490427 Arthropod Gravedigger ID
135590854 19561012 Aloysius Pornman MO
```

As you may have noticed, the date of birth is provided as a single integer, in the format `yyyymmdd`; Matilda Vincent was born on the 7th of June 1967. The `1` in the filename `people1.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 five data items: the social security number, first and last names, date of birth, and state of residence 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 youngest and oldest people represented. It should print the social security number, first and last names, date of birth, and state of both the youngest and the oldest person found.

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 youngest 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 oldest with almost no effort.

4. *Promote the Oldest*

For some unfathomable reason, the management wants the oldest person to occupy the first position in the array. Modify your program so that after finding the oldest 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 oldest person must occupy the first position in the array, but also that the second-oldest person must occupy the second position in the array. So, after searching for the oldest 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 oldest person you find (which must be the second oldest 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-oldest 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 oldest person therein. This makes it very easy to search the remainder of the array to find the second and third oldest.

7. *The Ultimate Demand.*

Now the management make their final demand. You are to repeat the process of moving the nth-oldest 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 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 - 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.

10. *Analysis*

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 items? 3000 items? 10,000 items?

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. `databaseN.txt` contains N thousand data records, and N can be 1, 2, 3, 5, 10, 20, 30, 50, or 100. 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 plot a graph of the results. The horizontal axis should be the amount of data, and the vertical axis should be the time it takes.

What do you observe about the graph? What is the mathematical relationship between the number of items and the time taken? Explain why this is true.

Finally, how long would your program take to sort a database of one million items, if we had one?