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 datafiles 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, database5.txt, database10.txt, database20.txt, database30.txt, database50.txt, and database100.txt.

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/162/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 date of birth, their social security number, their first name, their last name, their state of residence, and finally their zip code. The six items are separated by spaces, but no item will ever contain a space. Here is a sample from near the beginning of the file:

```
19670607 114680858 Matilda Vincent MI 71734
19471024 114930037 Desdemona Hanover ID 69743
19790110 115550206 Xanadu Perlman ND 12193
19630921 116520629 Alexander Hall SD 94976
19301016 117050976 David Lamprey GA 50895
19650202 119610646 Thomas Porlock IL 83582
19621126 120330928 Cary Cartman NC 90387
19620411 122460462 Bella Oldman SD 94495
19220213 123040628 Elizabeth Watson NC 90369
19230308 123580905 Gustav Hornswaggle MN 37568
19840613 125040813 Godfrey Tumor OR 55645
19580903 125610677 Gustav Trentham IA 73590
19521219 126470499 Justin Oddly MA 33458
19300616 127700250 Ursula Farnes LA 74163
19791114 129540334 Betty Eaton NH 25807
19361114 130020412 Maggie McIntosh NV 16628
19631118 132680826 Raul Kringle NJ 53633
19490427 135040001 Arthropod Gravedigger ID 68521
19561012 135590854 Aloysius Pornman MO 27649
```

As you may have noticed, the date of birth is provided as a single integer, in the format `yyyymmdd`; Betty Eaton was born on the 14th of November 1979. The file `database1.txt` 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 `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 with either first or last name should be accepted. For every matching entry that is found, print out all six 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.

   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 six 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 print the contents of the array after it has finished. Look at the output to make sure that everyone is printed in order of decreasing 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 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 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.

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.