Exercise 12 – Sorting and Searching

Objective

The objective of this session to use the standard functions `qsort()` and `bsearch()` firstly to sort integers then to sort pointers.

Reference Material

This session is based entirely on the Sorting and Searching chapter. This practical session is located in the following directory:

	Microsoft Windows	*Linux*
Directory:	**c:\qacadv\sortsrch**	**~/qacadv/sortsrch**
Solution directory:	**c:\qacadv\sortsrch\Solution**	**~/qacadv/sortsrch/Solution**

Overview

There are five questions in this session.

The first question uses `qsort()` to sort an array of integers. The second question extends this by first sorting and then searching through the array for a fixed values.

The third question extends the "lottery" program from earlier in the course, so that the ball numbers are displayed in sorted order.

The fourth question returns to the database of elements, writing a program that sorts the elements into different orders.

The final question takes a first look at the problem of sorting words in a file. This problem will be revisited in two future labs on linked lists and trees.

Practical Outline

1. **On Microsoft Windows** open **qsort.sln, on Linux** change your current working directory

   Examine the program in **qsort.c**. You will find a rather large array of random numbers to be sorted. You must write a comparison routine then add a call to `qsort()`. The `ASIZE` macro from the arrays chapter has been added to help determine the number of elements in an array.

   You will see a checking routine, which checks that the array really is sorted.

2. **On Microsoft Windows** open **bsearch.sln, on Linux** stay in the same working directory

   Examine the program in **bsearch.c**. This will require the routines you wrote for question 1. Having sorted the array, the program searches for specific values. You must also write the `lookFor()` function which is called several times. As you will see the routine must return the index position of the number within the array. Thus if you are looking for 37 and that occurs in the 3rd location in the array you should return 2 (counting from zero).

   If the number does not occur within the array you should return -1.

3. **On Microsoft Windows** open **lottery.sln, on Linux** stay in the same working directory

   Examine the program in **lottery.c**. This is the solution for the **lottery.c** program from the previous practical (use your own version if you prefer). Modify the program, so that it sorts the numbers using `qsort` before displaying them.

4. **On Microsoft Windows** open **sort_db.sln, on Linux** stay in the same working directory

   Examine the program in **sort_db.c**. This program uses the "optimised" elements database previously seen in the Structures practicals. There is a `readIn()` function which reads all the elements into an array. There are three arrays of *pointers* which are initialised with the addresses of the structures. It is these arrays of pointers which you must sort.

   There is an **elements.rec** file built from **elements.txt**. If you examine this text file, you will see that the elements appear in no particular order. Write a comparison routine to compare two elements of the pointers array. Remember that the routine will have parameters of type *pointer to pointer* to element structure. Why? Well, when sorting `int`s your compare routine was passed pointers to `int`s. Thus when sorting pointers to elements your compare routine will be passed pointers to pointers to elements.

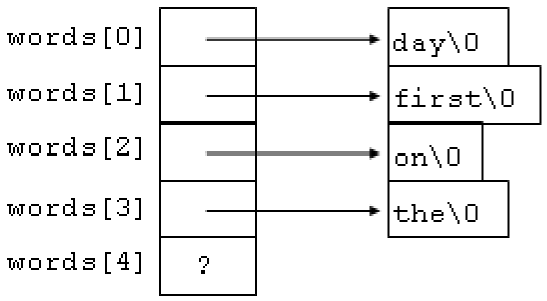
   To sort the `nameOrder` array your comparison routine will need to concentrate only on the `name` data member. You can use the standard `strcmp()` function to compare the two names. To sort the `rmmOrder` array you will need to concentrate only on the `rmm` data member. Similarly to sort `meltOrder` you will need the `mp` data member.

   A `displayRecords()` function has already been written to display the arrays of pointers as elements on the screen.

5. **On Microsoft Windows** open **freq.sln, on Linux** stay in the same working directory

Examine the program in **freq.c**. The `main()` function prompts for a filename, which it opens for reading before calling `process()`. `process()` repeatedly calls `getNextWord()` to break the file into separate words.

Your task is to sort these words. You should do this by building an array of pointers to characters. Start off by making this array a fixed size, say 5000 words. Each word should be copied into dynamic memory before storing in the array. As an example, imagine a file is opened containing the words “on the first day”. The array of pointers would look like:



The thing to remember here is to keep the array sorted so that it may be quickly searched for duplicate words (which do not need to be stored). New words should be added onto the end of the array. Thus if the next word were “aardvark” a pointer to it would be placed in element `words[4]`. Having done this the 5 pointers then need to be resorted. Take care not to attempt to sort more pointers than are stored in the array, otherwise you will be sorting random values into the array.

There are some files for you to sort. **short.txt** contains the first few lines of the much longer file **long.txt**. The file **revord.txt** contains a file with words in exactly the reverse order.

When you have your program working you might like to introduce the timing routines from the arrays practical.

Why do you think sorting the file **long.txt** should take so long? What special property of the `words` array might make it difficult for `qsort()` to cope with?