

# CST8130: Data Structures

## Lab 2: Using `qsort()`

### Introduction

I have supplied code that inputs data covering U.S. earthquake incidents for over 300 years. The records have a number of fields.

Your task will involve sorting the data in at least four different sequences, using fields of differing types (at least one *int* field, one *double* field and one *string* field).

The original data is read into memory in a dynamically allocated array of *EarthQuakeRecord* objects. Once all the data is read, this array will remain unchanged throughout the program's execution. You will generate four different indices to give four different views of the original data. Each element in these parallel arrays will contain a pointer to one of the original *EarthQuakeRecord* objects. These parallel arrays will be sorted based on some data value in the *EarthQuakeRecord* objects. Thus, the pointers (that is, the addresses) will be sorted to offer an alternate sequence.

### Detailed Steps: Build Program

Begin by downloading the code and data file. Build a project with the code; decompress the data file; try running the program. The program will read all records (over 157,000), but do nothing with them. After compiling the program, run it under the debugger, set breakpoints to inspect memory as the records are being input. You should be able to verify that the initial program works as expected. (Note: This first step will really be the first step in your test plan. That is, you will be working through a plan to test a stage of development.)

### Detailed Steps: Draw Memory Map

*Original Map:* You must understand the organization of memory before proceeding. Sketch a picture as memory is allocated in the original code. The original class *EarthQuakeDataSet* had three data members:

- *int nMaxRecords:* Stores a value entered by the user. It is used to size the dynamically allocated array. It also defines the maximum number of

records to read from the file and store in the newly created array of *EarthQuakeRecord* objects.

- *int nNumRecords:* Stores the count of the number of records successfully read from the file.
- *EarthQuakeRecord\* pEarthQuakeRecords:* Stores the address of the dynamically allocated array of *EarthQuakeRecord* objects.

*Planned Memory Map:* Of course, you'll keep the original three data members. Ultimately, you'll add four new data members to manage the four new indices. I would suggest that you sketch your enhanced memory map showing all four to-be-added data members, and all four to-be-created arrays of pointers.

*But here's an important point.* When you actually begin to build a solution in program code. Work with only one new index array at a time. Once you complete the code for the first one, the following three will be much easier.

So, to summarize:

- Plan for all four in your memory map, to give you the big picture.
- Implement only one at a time in code.

### Detailed Steps: Write Your Problem Statement

At this point (with your planned memory map sketched out), you can clarify the problem statement. What do you expect to achieve, and how do you expect to achieve it? We want to use *qsort()* in order to achieve the excellent Big-O of  $O(n \log_2 n)$ . But we want to minimize data movement, so we'll sort arrays of *pointers-to-EarthQuakeRecord* objects, rather than arrays of *EarthQuakeRecord* objects. Sorting pointers means moving 4 bytes per record, as opposed to about 100 bytes per record. It also means we can have four parallel sorted views of the data without having to re-sort again and again. Once the four parallel index arrays have each been sorted once, they will be available for unlimited use without having to be re-sorted again.

Once the data has been read and the four indices have been built and sorted, you'll need to have a mechanism to display relevant data. You can create a single display routine that can work with any of the four parallel indexing arrays of pointers. You'll probably also need some menu to allow the user to select any of the four views (to display); the menu should also allow the option of exiting the program.

### Detailed Steps: Display Output

You need a method to validate the success of subsequent sorting operations. I would suggest creating a function that will display three subsets of data: perhaps the first 10 records, 10 records in the middle and the last 10 records. This will provide an efficient method of reviewing the result of your sort operations. (It sure beats trying to inspect all 157,000 records).

### Detailed Steps: Create Four Duplicate Arrays of Pointers

The original objects will remain in their original order in the original array. Ultimately, you'll be sorting parallel arrays of pointers to *EarthQuakeRecord* objects; and the sequence of the original array will remain unchanged. Since I've asked you to sort in four different sequences, you'll need four new data members in the *EarthQuakeDataSet* class: each will contain an address of a to-be-created array of pointers.

How big is an *EarthQuakeDataSet* object? Initially, it was 12 bytes: 4 bytes for the three original variables (*nMaxRecords*, *nNumRecords* and *pEarthQuakeRecords*). With the additions, it will grow to 28 bytes (that is, an additional 16 bytes: 4 bytes for each of the four pointers).

This 28-byte object will be allocated on the stack as a result of this code in *main()*.

```
void main()
{
    EarthQuakeDataSet eqdsUSHistoricalData;
    // more code . . .
```

The object *eqdsUSHistoricalData* is allocated on the *stack* because it is created as an automatic variable (remember, the *stack* is a very limited resource, one that can easily be exhausted). All the subsequent dynamic memory allocation (using *new* and *delete*) will result in the use of *heap* memory (a global resource on your

computer, giving access to billions of bytes of memory).

After creating these arrays, copy the addresses of each *EarthQuakeRecord* object into elements in the array. This suggests the need of a *for* loop to walk through the array, capturing and assigning addresses.

### Detailed Steps: Sort one of the Duplicate Arrays of Pointers

Start with just one index. Prove you can perform the sort with the chosen key field, then later, move to the other three indices.

For this first sort, choose the key field that will determine the sort sequence for one of the arrays. Implement the call to *qsort()* and the *Compare()* function to generate the result. If you haven't already done so, make sure you have a correct memory map before attempting this step. You can work out the details of this step only if you understand how memory is organized.

### Detailed Steps: Teach qsort() About your Data

The call to *qsort()* is fairly simple. It's the implementation of the *Compare()* function that requires some thought.

Remember, *qsort()* will demand that the *Compare()* function have the following signature<sup>1</sup>:

```
int Compare(const void* p1, const void* p2)
```

Because the *Compare()* function will likely be embedded in the *EarthQuakeRecord* class, you must suppress the extra, hidden object-oriented *this* argument with the keyword *static*. So, the header for your object-oriented *Compare()* function would really look like this:

```
static int Compare(const void* p1, const void* p2)
```

The call to *qsort()* will likely be in the *EarthQuakeDataSet* class. So, it would look something like this:

```
qsort(
    (void*)pperCityView,
    nNumRecords,
    sizeof(EarthQuakeRecord*),
    EarthQuakeRecord::Compare
);
```

<sup>1</sup> The term *signature* refers to the pattern of arguments and the return value that will be associated with a function.

Note that the *Compare()* function identifier is modified with the *scope-resolution* operator, to tell the compiler that it can find the *Compare()* function in the *EarthQuakeRecord* class.

Once inside the *Compare()* function, you have two phases to complete the syntax.

### **Compare Function: Phase 1: Casting**

Cast the *void\** pointer *p1* to its correct type. It's not really a *void\** pointer (but that's the only way that *qsort()* can handle it); you need to look to your memory map, follow the arrows and correctly identify the type, and apply that type to the casting operation.

### **Compare Function: Phase 2: Dereferencing**

Now that you have the correct data type, you can dereference the pointer to get at the target data. Again, look to your memory map, follow the arrows.

### **Submission Requirements**

You will need to include the following:

- Problem Statement.
- Test Plan that validates the correctness of your work. Here, your test plan should clarify how you intend to use the debugger to inspect memory-resident data. Later you'll use the display function to view data. You may also need to set breakpoints at strategic locations (for example, you can set a breakpoint inside the *Compare()* function and check the pointer values as *qsort()* does its work.
- Source Code (with extensive comments that convince me that you understand your work, in particular, comments in the *Compare* functions you'll need to create to satisfy *qsort()*).
- Memory Maps to clarify how memory will be organized based on your implementation.

Note: No PDL is required for this lab. The *qsort()* algorithm has already been written for you by some other clever programmer. The key to understanding this work is your memory map, and your casting and dereferencing operations.