Section 1 Lesson 4

Section 1 lesson 4 sort

I Discuss new data structures, algorithms, or language features

We will use an array of structures of records to store employee data in sort.c Then use the selection sort on the array. The selection sort has an order O(n^2) time complexity. If you try to use this algorithm to sort a large number of records it will be very slow. Storing your records in an array also causes problems. You have to copy and move each field of each record during the sort. This lesson will demonstrate why you don't use selection sort for a large number of records and how arrays are not very flexible. Arrays have far better uses. They are great when you need a stack, or a queue, and are excellent for searching.

II Describe functions and modules why

Working from a previous lesson, structures.c, I added a selection sort routine. I put the code inline to let you see the algorithm as simply as possible. The swap step is more complex because setting one struct equal to another struct does not copy the underlying data. You need to do that task manually. This is called the shallow copy problem.

I can show you the additions I made in a more comfortable setting by using the same code I have previously discussed. The best way to learn coding techniques is to read working code. I can describe the data structures and algorithms but, until you see them expressed as code, they will not seem real. You can't make them yours until you have read the code, rewritten the code, and fixed any errors you may have introduced. Then you will know each part much better than if you had simply read about it. Breaking the code and fixing it again helps me learn quickly. Especially if working code is my starting point.

Please tell me if the code I have given you does not compile. I am trying to make sure it runs well before I ship it off. I am giving you working code so you don't have to waste your time creating it yourself; that comes later. I do want you to think about factoring my code and finding the parts which should be functions. In this case, I have put everything in the main() function so I don't have to pass arrays. I also make sure to write in a more readable style so you can understand each step. Remember to stress the code with large numbers. See where it breaks then think about how to fix it. I want you to think in broad strokes for now. How do the parts work together compared to how does each part work. That will come with time as you assimilate idioms of programming style. These idioms cross between languages, so learning one language helps you learn the next one more easily.

As you learn C/C++ from these lessons you are preparing yourself for python or Perl or most any other procedural language. While the syntax may differ, a few days with a book and a compiler and you’ll be fine. The data structures and algorithms are what really matter. Study any given problem and list the data you’ll need to process. Examine that data for a while to determine which data structures allow you to access the data easily and manipulate it most efficiently. Those data structures will define which algorithms you can use in your application. However, you may want to use certain algorithms because they better fit your data usage. Then, the algorithm itself would choose which data structure fits best. Either choice depends on the structure of the data. A little time finding those patterns makes your choice easy and implementation of your application will flow smoothly.

Programs = Data + Data Structures + Algorithms.   Once you know the data you will be working with the next two parts follow. So study your data first. You will be able to make the most efficient program to solve the problem, once you have examined your choices of data structures, and the algorithms which work with each of them. This lesson is about sorting. Specifically the selection sort. It scans through the array looking for the smallest for your given criterion. When the sort finds the smallest record the function swaps it with the first spot in the array. Then it looks for the second smallest record and puts that one into spot number 2. This is a very simple, easy to understand sorting routine. However, it is much slower than what is possible. Normally, I call the built in quicksort routine if I want something sorted. But, then, I had to learn all of these sorting routines in my algorithm class :) I want you to see how things are moved around in an array of structs, and why you DON'T want to design things this way unless you absolutely have to, or the number of records being sorted is very small (less than 100 for instance).

The next lesson will be sorting a linked list instead of an array. By comparing it to this lesson you will see why pointers make life easier, your programs more compact, and your programs run faster. Learning what does not work, and why, is as important as knowing which methods work best. You learn when to choose which data structure and which algorithms work best for your data set.

III Compile and run the code

- Open a terminal window (in Linux press Ctrl Alt and T simultaneously) (in Windows 7 - Start | All Programs | Accessories | Command Prompt - but I usually drag it to the Taskbar for ease of use)
- Navigate to … S1L4 using cd (change directory in Linux or Windows using MinGW)
- Type – **make** which will link and compile the application called sort in Linux, in Windows it is called sort.exe
- Type - **./sort** (in Linux) or **sort** (in Windows) to run the application
- Read sort.c in your editor to find how the code creates the application's output

My former working directory for this lesson was:

kevin@Hera:/mnt/share/BD_solutions/lessons/1 tools/S1L4_sort

Your file location will start differently and for Windows machines the slants flip over.

My new working directory:
kevin@Hermes:/home/kevin/files/share/BD_2nd_Edition/1_tools/S1L4_sort

IV Describe code line by line how

OK, from the top.

```
#include <stdio.h>   -- needed for the printf statements
#include <stdlib.h>  -- needed for rand() & srand()
#include <time.h>  --  used to seed random number generator with current time
```

Next we create our database record template.

```
struct db         // create database structure of employee data
   {
   int id;            // an id to be used as a key field
   int age;         // integer holder for age data
   float salary;  // floating point holder for salaries
   };
```

In main() we declare our variables and set a few of them. Two of these new variables, age and salary mirror fields in our database records. I use them to sum the age and salaries as I scan the database. Summed salaries gives you the total payroll while salary/count will give you the average salary. I mirror the record field names with these new variables to make the code more readable and to cause fewer bugs.

Next we create our database with

```
db employee[count];
```

Instantiate is a word used to describe both the allocation of the memory needed for a record, and the filling of that record with data. Thus, we instantiate the founder's data record. In this case the memory is allocated from stack space at compile time by creating the array on line 27. It is filled like this:

```
employee[0].id = 42;
employee[0].age = rand() % 40 + 20;   // minimum age is 20
employee[0].salary = 25080.15;          // our frugal founder
```

Next comes a loop adding the data of the rest of your employees. Notice you can change the number of employees at the top of main in the int count = 19; statement on line 23.

Scan through the unsorted list printing the data in each record.

```
for (int i=0; i<count; i++)
  {
  printf(...);
  }
```

Now comes the selection sort routine, set to sort on age held in employee[p].age Our outer loop keeps track of how far we have sorted through the array of records with the index q. The inner loop determines which record is being compared and possibly swapped. This is indexed by p. Keeping track of p and q is most of the work of this algorithm. The rest is done in the swap step. This, too, should be a function but I left it inline for clarity.

A swap function takes record A and swaps it with record B. However, we need a third memory space for temporary use. We create the space with struct db temp; on line 25.

```
temp = employee[q]              set temp equal to the receiving record
employee[q] = employee[pMin]    swap from sending record to receiving record
employee[pMin] = temp           set sending record to temp for the swap
```

I wrote lines 72 through 88 for demonstration purposes. Line 72 would normally call a subroutine named swap() which would exchange the data between the two records. In this case, I am setting a flag, pMin, to be used a few lines later to trigger our inline swap routine. This is one of the more simple sort routines, which means it is easy to read and understand. The more complex sorting routines take much longer to understand but do essentially the same thing: sort. I chose selection sort, not for its efficacy but rather, for its clarity. There will be more sorting lessons in the future to expand your growing toolkit.

Next, print out the employee database in age order with lines 92 through 98. Here is where we use the variables age and salary to sum the data as we count through the list. Notice how many of the comment lines resemble pseudo code. When I was laying out the program I used those lines as pseudo code, then as comments when I wrote the code for each section. I wanted to keep straight what was going where when I expanded them out to copy each datum of each record.

Sort again, only this time use salary as your criterion with lines 107 through 139. Don’t worry, the swap routine will be its own function in later lessons. But, for now, we inline them so you can follow the program flow explicitly. Once again, we use the comment lines as sanity checks to make sure we are moving the right things into the right places. This time we are sorting on employee salary so line 116 is not the same as line 63 where we were performing our comparison on the age field. Other than that one place this is copy and paste from the previous sort.

Lastly, you print out the employee database in salary order. I have also added averaging and summing. I played fast and loose with those however. When you divide an integer by an integer you get an integer back but one which has a rounding error (usually). When you divide a float by an int you get a float but you really should cast the int into a float to be formally correct and satisfy all the finicky rules.

For example:

```
printf("\n The average age of employees is %2d\n", age / (float) count);
```

The cast, (float), lets you perform that division a little more accurately. To be completely accurate you would say

```
printf("\n...%2d\n",   (int) ((float) age / (float) count));
```

Which casts both ints to floats, performs the division, and then converts the result back to integer. Yes, all those parentheses need to be there :) Do you need absolute accuracy, or is close good enough? Usually, the former case is true, but sometimes you just want a ball park estimate to save time. Remember to set the compiler flags correctly to produce the most warnings. When you can compile and link with no warnings it is a good thing. Those warnings hint at subtle, underlying errors which may not show up at run time. But they normally show up when the client starts poking around :) So set the warnings to all and clear any you get as you edit.

V Describe makefile line by line

Line 3 sets the compiler macro to use g++. I am using g++ instead of gcc so I can use a few C++ idioms. Such as in line 38 of sort.c where I wrote

```
for (int i=1; i<count; i++)
```

i is declared and used within the for loop and then falls out of scope immediately. You cannot do this using standard C. It has become a standard idiom of mine to limit the scope of index variables.

Lines 5 and 6 link the executable file from the object file sort.o

Lines 8 and 9 compile the object file from the source file sort.c

Lines 11 through 13 clean up during debugging. By removing the executable file and the object file the makefile always runs completely. If odd things happen during debugging removing all the object files and running make may solve them.

VI Deeper

This lesson is about how not to store and manipulate a database. With the right data structure many things become simpler. In this case you have the wrong data structure for what you are trying to do, sort the list. Each time you swap during the sort routines you are moving the data in memory. Every single time! With the right structure you won't move the data once it is added to the list. Pointers to structures will be rearranged, not each node's data. I thought it would be best if you saw the work it takes to sort an array of data. If you want to sort a simple one element array, then you can do it this way. But if you are moving data structures around each time you change something you will find the program becomes very sluggish. The next lesson will show you a better, faster, and more compact way to do what we did here in this lesson.

You may not be familiar with the += operator

```
x += age;
```

In longer form this would be x = x + age; But += is easy to type and means the same thing.

There are also -=, *=, /=, and a few more. They are very convenient. |= and &= are great for mask and shift routines. |= is an 'or equal' and &= is the 'and equal' sign. |= is good for setting a single flag (bit) in a command string. I use ^= now and then to toggle flags quickly. It is 'exclusive or equal'.

Oh, another thing. I think the standard way they teach you pointers gets overly complex, way too fast.

Like the *(p+3) as compared to *p + 3 difference. Just remember the ‘My Dear Aunt Sally’ thing and you'll remember to look up binding precedence. The * operator binds more tightly than does the + operator.

If you look at my linked list version of the database in S1L3 you'll see I used

```
employee[i].age
```

The pointer I use, ep, points at the employee record I want then the arrow form ep->age gives me the age without once worrying about pointer arithmetic.

The most common mistake is forgetting about binding precedence. But I avoid it by not using pointer arithmetic. It comes in handy during only the learning of the language; not in its daily use. Except if you are doing stream work. Then you capture the head of the stream and work by offsets. Like when you're getting TCP/IP packets and digging out the payload as well as the routing info.

Folks complain about how they screw up but never think to avoid those pitfalls in the first place. That's what I do in the woods so why shouldn't I do the same when I am writing code?   Complexity for its own sake is kind of sad. Use only the amount of complexity necessary to solve the task elegantly. No more. Yes, add error traps. But don't add fluff code or do really strange things to get the job done. If you have to do that take a break and rethink the problem. Occam's Razor.

Elegance is COOL

VII Assignments

1. Change the code to sort by salary instead of age.
2. Sort by id.
3. Store a rolling sum of salaries then print the average after sorting by salary or id or age.
4. Print array in backwards order for practice.
5. Change sort routine so you don't have to print the array backwards.
6. Use functions from <time.h> to time how long it takes to sort. Time from line 51 to line 92 of sort.c Increase the array's size from where it is to 100 then 1000 and then 10,000 employees. Compare the times to one another. You will see $O(n^2)$ time complexity. Remember this for the next lesson where we sort a linked list instead and get O(n lg n) time complexity.
7. Read "Algorithm Analysis" in the appendices section.

VII Links

https://en.wikipedia.org/wiki/Selection_sort

http://www.algolist.net/Algorithms/Sorting/Selection_sort

https://courses.cs.vt.edu/csonline/Algorithms/Lessons/SelectionSort/

https://www.geeksforgeeks.org/selection-sort/

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

https://www.hackerearth.com/practice/algorithms/sorting/selection-sort/tutorial/

https://www.programmingsimplified.com/c/source-code/c-program-selection-sort

https://www.geeksforgeeks.org/sort-array-large-numbers/

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html

https://www.sanfoundry.com/c-program-sort-array-ascending-order/

To see the algorithm expressed in Python

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html