Section 1 lesson 2

This lesson introduces our first data structure, the array. Arrays are indexed lists of contiguous memory locations. You can access the array's contents, either by index value, or by counting offsets. We will use this data structure to store a simple list of data from each of our employees: id, age, and salary.

I Discuss new data structures, algorithms, or language features

My boss wanted to find out how age correlated with salary in our company but did not need any names. struct is a keyword in C. Using it allowed us to group variables into a convenient form. We used this record structure:

```
struct db
   {
   int id;
   int age;
   float salary;
   };
```

Each record has two integer values, id and age, and one floating point value of salary.

```
struct db employee[10];
```

Read the above line of code from right to left and it tells us we have an array called employee which is filled with db structs. In this case we have ten structs in our array or ten employee records with id, age, and salary.

```
struct db
   {
   variable type variable name;
   variable2 type variable2 name;
   etc.
   };
```

groups our variables in a record called db which in this case means database. I used db as a mnemonic any sensible name will be fine. Naming variables, functions, or files is a story all its own.

Because this is a one time situation the data is added only once and not changed. If this is all you intend to do then an array is a fine data structure. While searching an array is easy, it does not allow easy sorting or deletions. In future lessons other data structures are explored which allow us to sort, add new records, or delete outdated records easily.

II Describe functions and modules WHY?

Code files are written with a fairly standard structure.

Comment lines include the file's name, a by line, and a description of the code

#include statements tell the compiler where to look for any function not already included in the language. Some of these libraries are standard while others are custom crafted by the programmer.

Function declarations tell the compiler to remember the name and format for later definition.

Variable declarations tell the compiler to put these names into it's dictionary so it can find them when it scans the code.

The main() function is where the operating system and your code meet. main() is the standard entry point for command-line applications. In our case, main() takes no input and returns an integer error message when it closes. You want to exit a program with no errors, which is designated by a zero.

Function definitions can follow the main() function or precede it. Where they are located is a matter of style. If you declare your functions prior to defining main() the compiler will not complain. If you define your functions before main() then there is no need to declare them first. What matters is if the compiler scans a function not already listed in its dictionary there will be a problem. When you declare a function you are telling the compiler its name, input type, and output type. A function definition starts with the declaration line and continues with all of the code you need to perform the function's work.

I normally have main() at the end of my file and create function definitions above it as my program progresses. But, later in the development of the application, I'll write function declarations for each function I've thoroughly debugged and move the definition to after main(). When I am done debugging the code all of the function definitions will be after main(). In this way I keep what I am working on at the time closer to the top of the file. As long as the compiler can fill its dictionary before it scans a variable or function in your code life is good.

We #include three libraries: stdio.h, stdlib.h, and time.h in structures.c. Standard I/O gives us printf(), the function we need to print to the screen. The standard library lets us use srand() and rand() which seeds the random number generator and lets us obtain random numbers respectively. The time library is used for the time() function which we use to seed srand(). If you don't seed a random number generator with a different number each time the program runs the "random" numbers you receive will be identical.

Next we come to a structure definition – struct db. struct is the keyword used to start a structure definition. In this case I am calling my new structure db for database. What I am creating is the structure for one record in the database I will build. There are three fields of data in my db record – id, age, and salary. The id and age are both integer numbers (natural numbers) while the salary is a floating point number so you can pay in dollars and cents. Line 20 puts this all together into our employee array.

main() is the only function in structures.c which is not contained either in the base language or in one of the three libraries we #included.

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 … S1L2 using cd (change directory in Linux or Windows using MinGW)
- Type – **make** which will link and compile the application called structures (in Linux, in Windows it is called structures.exe)
- Type - **./structures** (in Linux) or **structures** (in Windows) to run the application
- Read structures.c in your editor to find how the code creates the application's output

IV Describe code line by line HOW?

I will describe structures.c so you can understand why I wrote each line, and how they all work together to produce our application. Lines 1 through 8 are called a comment block. This is the common format with /* comments */ - the file name, my by line with a copyright sign, and a description of the code. The copyright is there only for form as I do expect you will take this code and rewrite it. And then rewrite it again for something else. It is simply a template you can build on. That holds true for all the code in all of the lessons in this course.

I want you to use the code I provide as a learning tool and then as a base upon which you can build something else. If you give me credit in your code that would be great. But don't worry about breaking any copyright law. It is like your old hammer. The one with its third new handle and one new head. Yes, it is the same hammer but over time it truly has become yours. I want you to take my code apart, learn from it, critique it, and put it back together in better ways. By giving you working code of some complexity you will see how a problem is solved, how you write data structures, how you use algorithms, and give you a starter toolkit for building your own projects, to solve new problems.

After the comment block we include three libraries. Lines 9 through 11 expand the compiler's dictionary with many new choices. But only a few of the available functions will get linked into our application. From stdio.h we use printf(). From stdlib.h we link srand() and rand(). And from time.h we use time(). As you can see even though we included a large chunk of code with each library we

only used what we needed for the final executable file. This helps us write compact and speedy code. I like C :)

Next we create the structure for our employee records. Lines 13 through 18 create one database record, db, with three bits of data for each: id, age, and salary. Line 20 builds an array of 10 of these structures so we can experiment with them. Line 20 is best read from the right. There are 10 employee records of type struct db. struct and db work together, struct tells the compiler to expect a label, in our case db. You will see this standard idiom of C often.

I declared the structure globally because this is a simple app and it makes for cleaner looking code. I wrote it in a more expanded way so you can see all the parts. C can be quite obscure if you use all its tricks. I tend to write code which is very readable even if it may be slightly slower.

How it is written now:

```
struct db
   {
   int id;
   int age;
   float salary;
   };

db employee[10];  // create a new database array called employee
```

I normally write it like this:

```
struct db
    {
    int id;
    int age;
    float salary;
    }  employee[10];  // create a new database array called employee
```

Next we enter main() and declare a pointer to structures of type db. Then we point it at the first employee in the array.

```
struct db *ep;
ep = employee;   // ep now points at employee[0]
```

In C the name of the array is also its base address (line 27). All you need to do is offset from this base. The first database record lives at the base memory address of employee. The second database record lives at memory address employee + the size of one database record. Thus, record n would live at employee + n * size of db. Arrays allow you to perform this type of memory addressing directly.

We seed the random number generator on line 30 by setting it to the current time with time(NULL) srand(). Lines 34 through 36 populate the fields of the first employee record. Notice each part of line 35

```
employee[0].age = rand() % 40 + 20;
```

employee[0].age chooses the structure's member, age from the 0th record of our employee database. rand() provides a pseudo-random number between 0 and 65535. The percent sign is the modulo operator in C. % takes our random number and divides it by 40 and returns the remainder while discarding the quotient. That way, our random number spans a large quantity but provides us with the range we need. By modding it with 40 you get a range between 0 and 40. I offset by 20 years so we don't have any child employees, giving us a random number between 20 and 60.

Structs are very handy. They also were the beginning of the object oriented notion. Structs are more convenient for me since I like avoiding all the gotchas of C++. I do use some C++ conventions though. Such as the for (int i=1; i<10; i++) construct. Notice each time I am declaring a new int i. The i within that function is independent of other i's in the system. If I had declared int i; at the top of main this would work too and be strict C. This concept is called scope. Each variable i remains active only as long as you are inside that for loop. This construct is cleaner in many ways. The // comments are also not strict C, they come from C++.

You'll see two lines which may look strange to you:

```
srand((unsigned) time(NULL));    &
employee[i].salary = (float) (rand() % 40000 + 20000);
```

I have used what is called 'casting' in both cases. In the first case I take the output of time() and change it to an unsigned variable, which srand() requires. In the second case I take the random number and change it into a floating point variable, which is what employee[i].salary wants. This is necessary because C is a typed language. Knowing the size requirements of the variables is important, not only to the compiler, but also to your use of memory. I like strict typing because I always know how big things are. That really helps when you're working on a microcontroller with only 10k of RAM.

I modified this code to make it work with pointers. This is why I added lines 24 through 27, creating a struct db pointer and aiming it at employee[0]. I also added another loop (lines 52 - 57). It does the same thing as the previous loop, stepping a pointer ten times instead of the previous code where we had an index into the employee array of structures. The pointer ep moves from one employee record to the next with the ep++ operator. Since ep is declared as a struct db pointer it knows exactly how far to move in memory with each step. Instead of typing employee[i].id, you can use the pointer notation ep→id. Notice how much easier it is to read. It is also easier to type. Remember: laziness is a virtue.

I used formatted output. Look at the printf statements, on lines 49 & 55, especially at %2d , %2d and %6.2f Those are the formatting strings, d for integers and f for floating point numbers. %2d takes two spaces to hold an int. In this case it is an id of 2 characters or less. Same with the %2d for age. %6.2f is used to format the salary. 6 characters total with 2 after the decimal point. Remember, a leading minus sign and the decimal point each use one of those character spaces. That's enough for now, printf can do much more.

V Describe makefile line by line

Each rule in a make file has the following syntax:

```
<target>: <list of dependencies>
<tab>Commands to execute
```

Using make allows you to create complex schemes of rules which require only one command to run. A makefile is a list of rules in reverse order. The first line (rule) is how you make the final executable file. Those following it build the parts necessary to satisfy the dependencies of the top rule. I will use // for comments describing what each line of the makefile means. A line in the makefile which has # in its very first spot is a comment. Using # anywhere else is not correct.

```
CC=g++   // CC is a macro for compiler choice.
         // replace $(CC) with g++ anywhere it appears

structures: structures.o  // the application structures requires structures.o
   $(CC) -Wall -o structures structures.o
                // g++ create the application structures with all warnings on

structures.o: structures.c     // structures.o requires structures.c
   $(CC) -Wall -c structures.c
                // g++ compile structures.o with all warnings set

clean:           // this rule is used to clean up object files and executable
   rm structures.o             // allows us to create a clean build
   rm structures.exe
```

VI Deeper

Pointers

Most lessons in C use strings as your introduction into pointers. I find this is confusing. That is why I will use data structures to introduce pointers. It fits my idea of a pointer to a chunk of memory instead of a single point. I will cover strings once the basic ideas of pointers have been introduced and reinforced. The concept of pointing at chunks of memory instead of individual memory slots better illustrates what is going on. As you can see in the strings lesson there are a number of ways to do any given thing which makes it a less effective teaching method. Complexity should be introduced gradually not dumped on the student all at once.

Pointers are addresses. Each of them points at a memory location. You begin by aiming your pointer at the first chunk of data. Depending on your pointer type, you could be pointing at a single character or a

number of associated fields in a record. You can use pointers to move from address to address. If you increment a pointer it will point at the beginning of the next data record. When you want to look at the data at an address you **dereference** the pointer by adding the preceding asterisk, *p, which gives you the data at that location. The pointer p will get you to the right door, you need to open it, with the * operator, to see what is inside.

When you want to point at the next record you increment your pointer p++. This automatically adds the size of your defined chunk of memory (pointer data type) to the address p to give you the beginning of the next record (structure, list, etc.). Without the pointer mechanism you would have to keep track of these sizes and offset your addresses accordingly. By using the pointer, you make this much easier and transparent.

Pointers are also used when you want to allocate large chunks of memory from the operating system. If you declare the array myArray[40], you are statically requesting the memory from your limited amount of stack space at compile time. Using malloc() and free() instead lets you obtain heap memory space dynamically during run time. There is a limited amount of stack space available while there is a vast (4 GB?) amount of heap space. If you are not careful you can use up either one or both types of memory. When memory was expensive you wrote code which used less of it, solving smaller problems. Now heap space is very large while stack space is still rather limited. I will show techniques using either type of memory during these first set of lessons.

I added these two lines to main() to declare the pointer ep then aim it at the first employee.

```
struct db *ep;
ep = employee;   // ep now points at employee[0]
```

In C the name of the array is also its base address. Now all you need to do is offset from the base. That is why we programmers count, "zero, one, two, three, ..." it is not because we are crazy.

I also added another loop. It performs the same task as the previous loop by counting ten times instead of the earlier code, where i was the index into the employee array of structures. Here the pointer ep is moving from one employee record to the next with the ep++ operator. Since ep is declared as a struct db pointer it knows exactly how far to jump the pointer with each step.

When designing algorithms which use pointers you will draw images with pointers to boxes. Adding a record to or deleting a record from a doubly linked list takes six pointer calculations. This is where the fun, the power, and the danger lies. If any of these pointer links are broken you can walk your algorithm off to nowhere. Strange things happen.

With pointers you can create algorithms to mimic trees, meshes, linked lists, queues, stacks, etc. Each structure can contain many internal pointers for these data structures. Pointers make life a lot more fun but they also can be dangerous. With a decent error you can crash everything. If you're really lucky you can blow your code away too. Ask me how I know. While arrays are good for some things the data structures I've listed are much more useful for the every day abuse of data.

Factoring

A function should fit entirely on one screen. It is difficult to follow the logic when my code grows too large. It is time to factor the code.

Factoring locates chunks of the routine which can stand on their own. You take those snippets and craft a function just for them. Then rewrite your original function by calling those new functions instead of including them inline. This makes your code more compact and more readable, allowing you to find logic errors in your coding more readily. Once you have created a library of these routines you can keep them available for future work on other applications. Then only the headers (myLibrary.h, mycology.h) are exposed to the public view. In this manner you can keep your code private while allowing it to be used in other's applications.

When you have factored out a number of commonly repeated routines you need to sort them into logically linked units. Put all your I/O routines into one library, your graphics routines into another, and your math manipulations into a third. Once these libraries are built, and fully debugged, they speed the creation of your next application. I tend to work in one area for a while so I find myself not writing a solution to one problem, but crafting a set of tools to answer questions within the same problem space. I can solve the next problem more rapidly by factoring my code and reusing it.

Finding structure in data sets

Finding and examining structure is important for solving problems and writing software. First determine the underlying structure of the problem's data. Can it be organized into lists? Imagine you have a stack of blank 3x5 cards and a pencil. Can you solve your problem by filling out the cards, sorting them, finding the ones you want, and using them as your new list? Is your list going to remain the same for the life of the application or will it need to change? How often those changes occur will help you determine which algorithms to use with your list data structure.

While arrays are good for some things, here are a few other data structures which are more useful for the every day abuse of data. Arrays have a few problems: they cannot easily grow, they are hard to sort, difficult to append while maintaining list order, and deletions are time consuming. Once you have allocated the memory for an array the size is fixed. Plus, it is allocated from the stack space of the application. There is far less stack memory space than there is heap space so another data structure is normally used for dynamic lists - the singly or doubly linked list. Here, each record has one or more pointers associated with it. They point at locations (addresses) of other records, or they point to NULL which designates the end of the list. When you sort a linked list the data never needs to move; it can remain in the same memory location. Only the pointers are modified so they point to the next record in your newly sorted list. A doubly linked list lets you access the list either from its head or its tail. A linked list can be used as either a stack or a queue. A stack pushes old records down as new ones are added. When the application needs the record from a stack it pulls the one at the top and the next record becomes available. A queue has an input end and an output end. Think of standing in line at

your grocery store. When you're done shopping you pick the shortest queue and then find it is the slowest. Such is life. But eventually you get checked out, exit the queue and go home.

If you need your list ordered (alphabetically or numerically) at all times the tree is a good data structure. Every time you add a new record the algorithm finds the right spot in the tree. Any time you need to delete a record you walk the tree to prune the unnecessary record. After either addition or deletion the tree is in order. Is your problem map oriented, routing of your fleet of trucks for instance? Do you need to find the best routes of least mileage, or most profitable? Then choose the graph data structure.

MinGW tools

You can use the command line more effectively now that you have MinGW set up . If you make an command error simply hit the up arrow. That will bring back the line you just typed ready for editing. Use the back arrow (NOT the backspace arrow) to move back to where you need to make corrections. Once you have the error corrected just hit enter, there is no need to be at the end of the line to do so.

This makes life much easier because you can arrow back to the very beginning of the command prompt session. Normally, I am working inside an editor, with the command prompt window alongside it. Screen space is valuable; that is why I am now using three 22" monitors. I am thinking about hooking up another one. But I digress. Back to work flow. Once I am done editing I hit **ctrl-S** to save the file. This works in almost all editors. In the command prompt window I type **make**. Once make completes successfully I type the name of the app to run it. Then I invariably want to make changes. So back to the editor to make the changes. Hit save, then in the command window arrow up to make. After it has compiled and linked the executable file you hit the down arrow to get the name of the app and run it again.

This work flow occurs rapidly, quickly becoming second nature. When you understand the makefile you'll be very far ahead of the crowd. Using a GUI IDE is nice but you miss the details, details which can be important to your project. Optimization is part of the compiler's work. With the profiler, the debugger, and the compile and link messages you gain insight into how to use the compiler's optimization system better. You can optimize for speed or for memory usage. In an embedded project it is also important to optimize for ALL of the different memory storage areas.

Optimization is not trivial. You can get the computer to work on larger problems when you write your code more efficiently. Or you can solve the same size problems more rapidly. I have written code which takes an hour for each cycle to create one frame of video. As you may imagine, this is tedious! Any optimization lets me do more work more rapidly. Or I could just split the program across the network and let a thousand boxes run my code in parallel. Ah, dreams...

Programming is a craft; it is like creating fine cabinetry, fashioning excellent boots, or even designing a winning motor car. You can make something functional quickly but crafting a fine object is like sculpting art. Passion is important as is the day to day work of doing your best on lesser projects.

Hone your skills on even the least of your efforts. Practice is valuable - art is a verb, craft even more so. Writing code to test ideas you have, exploring the problem space of your mind, is sublime. Your computer, or any other you may find along the way, is a tool for you to use to explore problems you wish to solve. All you need is a text editor, imagination, and a compiler. Those can be found freely on the ‘net. Free is good. Code can also be free, but your own additions to it will cost your employer :) Working for free may be noble, but giving your art to one and charging another is not bad, it keeps you under a roof and fed. When you craft code you are not simply creating one thing but a tool to be used by many. You should be paid for it even if only a little. Since it is just code [1010001101011010] it can be sold many times for a pittance, while keeping you under a better roof, eating better food.

VII Assignments

1. Add more fields to the data structure and output more detailed reports.
2. Write reports using a subset of the data – how many employees over $40,000 for instance.
3. Create and use data of your own choosing to fill a new array structure and write new reports.
4. Read information linked below for a different look at structures, arrays, makefiles, and pointers.
5. Read the appendix file “Format notes”

VIII Links

https://www.tutorialspoint.com/cprogramming/c_pointers.htm

https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/

https://fresh2refresh.com/c-programming/c-pointer/

https://www.programiz.com/c-programming/c-pointers

https://www.learn-c.org/en/Pointers

https://www.tutorialspoint.com/cprogramming/c_arrays.htm

https://www.geeksforgeeks.org/arrays-in-c-cpp/

https://www.cs.uic.edu/~jbell/CourseNotes/C_Programming/Arrays.html

https://www.learn-c.org/en/Arrays

https://en.wikipedia.org/wiki/Struct_(C_programming_language)

https://www.geeksforgeeks.org/structures-c/

https://www.tutorialspoint.com/cprogramming/c_structures.htm

https://www.learn-c.org/en/Structures

https://randu.org/tutorials/c/structs.php

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

https://www.cs.oberlin.edu/~kuperman/help/make.html

https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html

https://opensource.com/article/18/8/what-how-makefile

https://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_2.html

https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

http://www.bowdoin.edu/~ltoma/teaching/docs/make.html

http://nuclear.mutantstargoat.com/articles/make/

https://www.tutorialspoint.com/makefile/

http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch10s07.html

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