Section 1 Lesson 3

Section 1 lesson 3 linked

I Discuss new data structures, algorithms, or language features

Linked lists are a commonly used data structure. It can be modified in many ways. In this lesson we create data structures with pointers. These are used to create a linked list. Then a some fiddly bits, like seeding the random number generator with srand(), casting variables, plus more about pointers. You will learn linked list idioms for: pointers, links, accessing data in a node, walking the list, and allocating memory for our hybrid nodes.

Normally, you use functions to add each node to a linked list. I wrote this app so you could see all the steps in order. Calling functions does make the code cleaner, but much less clear if you don't already know what's going on. Functions will be used in the future to make linked lists, queues, graphs, and trees. But, if you dig into the functions, you will see these very lines of code. However, you will soon see that the data structure, and the data it holds, don't need to be tightly connected in one struct. You can use two structs in your definition: one for the data structure and the other for the data. This makes your pointer work a bit more complex but the app will run faster because the data never moves.

II Describe functions and modules

This application is written using inline code; there are no functions. Each chunk you see here will become its own function in future lessons. Here, you will learn how to define a hybrid structure which holds both the data, and the data structure. Each struct will be part of a singly linked list with a *next pointer. We also declare a few handy pointers to use with our list.

Randomize, so you obtain different age and salary values for each program run. If you did not include line 25, you would get the very same sequence of ‘random’ numbers each time you ran the code. To prevent that we make sure to seed our random number generator from the system clock. Line 25 is needed to #include <time.h>, the library holding the time() functions.

First, we allocate memory space for a node of the list, fill it with randomly generated data, then create 27 more records in the growing list. Set our employee pointer to the head of our linked list. Walk the list record by record, printing out the three data fields, in a while loop. Create a few variables to collect data on the list. Walk the list again, summing ages and salaries, while keeping track of how many are employed. Print the statistics in a formatted report.

III Compile and run the code --- expanded into prose instead of bullets for this lesson

- Open a terminal window (in Linux press Ctrl Alt and T simultaneously) (in Windows 7 - Start | All Programs | Accessories | Command Prompt - I usually drag it to the Taskbar for ease of use)

During the course of these lessons you will be working from inside a command window. No matter if you are using Windows or Linux, I want you to see how to write code without any fancy tools. Writing command-line applications allows you to create the solution to your problem quickly. If you need to massage data from one format into another, there is no need for graphics. Create the tool you need for the translation and continue with your work. Input/output functions are simple, as is most of the code. When we move into the graphics portion of this course, you will still use command-line tools, but when the program runs it will have a graphic display. However, underneath all that flashy exterior you have all the same code. The graphic user interface (GUI) adds a layer of complexity on top of what you already have created.

I think using command-line tools and makefiles helps us learn more about the hardware and how we can write more efficient code. I am not starting with object oriented code but rather with plain vanilla C. This will get you into the habit of thinking about how much memory you have and how you will allocate it from either the stack or heap.

- Navigate to … S1L3 using cd (change directory in Linux or in Windows using MinGW)

Navigation may seem more difficult when you need to type the path name instead of simply clicking on it in a window. But you can replace a lot of that typing by using the * key, liberally. For instance – I am currently on my server Hermes in a folder called BD_solutions, where all of the lessons and essays live, working on this text file. The full path name is "kevin@Hermes:/srv/samba/share/BD_solutions/lessons/1 tools/S1L3_linked/Section 1 lesson 3 linked.odt". To get to this file I logged into Hermes and typed **cd /srv/samba/share/*ions** There I saved 7 key strokes by substituting * for BD_solut. If I want to navigate a bit deeper, to find this file, I would type **cd *ons/1*/*3*/** to get there. From this … "lessons/1 tools/S1L3_linked/Section 1 lesson 3 linked.odt" to this *ons/1*/*3*/ for 18 fewer key strokes. Notice I quoted the string in the previous sentence. This is because Linux breaks instructions into space delimited chunks. So, Section 1 lesson 3 linked.odt is not one thing but five! By adding the quotation marks around the entire string the operating system will 'see' it as one chunk. This is one reason to eliminate spaces from file and folder names, and another reason to use the * key liberally to save typing. By using the * for a wildcard navigation will not be as difficult as it at had first appeared.

Hermes:/srv/samba/share/BD_solutions is the location of my version of the code checked out from the Subversion repository. Subversion is a versioning tool which keeps track of all the changes in all the files in this course. Each time I modify any file in the course I type **svn commit -m "comment here"** to log the change to the repository. Since I am working across a network, on multiple computers, this keeps track of versions. Each computer is updated by typing **svn update** when I start using it. I can edit any file from that computer and know I am not rewriting something I have already written. I checked the course out from the repository onto my laptop and onto one of the servers. I can edit either set of files as long as I commit to the repository, and update from the repository in a logical manner.

Using a terminal window allows you to work from any Windows or Linux box as soon as you sit down in front of it. There are some advantages to using the file system at this level. You can see all the files on a Windows system, even those hidden in the graphic file system. We have seen how to navigate into a folder but how do we go in the other direction? Type **cd ../** and you will move level higher. Type **cd ../../..** and be three levels less deep. I find I run my Linux boxes more from the command line than I do with a GUI. A GUI is pretty, and has its uses, but when I am writing code I find the command-line fits me just right.

- Type – **make** which will link and compile the application called linked (in Linux, in Windows it is called linked.exe)

When you use an IDE, like Microsoft Visual Studio (MSVS), you normally start from some sort of wizard which creates all the ancillary files necessary to create the executable. MSVS creates a few folders and fills them with files which tell the compiler how to build the application. You have to navigate quite a few menu structures to fiddle with all the link and compile settings. This goes against my curious nature. I want to know what everything is doing, and why. That is the major reason I prefer using makefiles. By writing a few lines of rules I know all of the parts and all of the settings needed to create my executable application.

The makefile for S1L3 has two rules to build the app and another one to clean up afterward. I use one make macro, CC=g++ which is later used in $(CC) This macro picks the g++ compiler. g++ is a superset of the gcc compiler which allows you to compile object oriented code. While this app is not written with object oriented methods, using g++ lets me use a few new features. For instance, I use // for commenting a line. This is not legal in strict C. I also use the idiom, for (int i=0; i<35; i++) which again, is not legal in strict C. Occasionally I will define a variable in between functions. This lets me associate those variables with that function in preparation for moving them all to a header file. Again, this is not legal in strict C. If you stay away from these features you can use gcc with no problems whatsoever. But I like them, so I use g++ more often than not.

The rules of a makefile start with what you are going to create, so on line 5 you are creating the file named linked. After the colon comes a list of files required to satisfy this rule. The next line, the actual recipe to build things, starts with a tab. Spaces won't work, and will be flagged as an error. This is my leading error when writing a makefile. It must be a tab with no spaces at all. Now, let's read line 6. Tab then $(CC) -o tells make to link the application and name it linked. The '-o linked' tells make the name of the application I want to create, and that g++ is linking, not compiling. However, to run this rule you need to have the requirement linked.o which leads us to the second rule in our makefile.

Once again the file being created is named, followed by a list of the requirements for this rule. In this case we need the source file linked.c The recipe to build it says use g++ to compile linked.c into linked.o Once this file has been created, make can run the first rule to create the executable file.

I included the rule for cleaning. When I type **make clean** both the object file and the executable will get erased This makefile has two other recipes to create a debug file and another for profiling the code. More on these later.

- Type - **./linked** (in Linux) or **linked** (in Windows) to run the application

I want you to be able to run these applications from Linux or Windows operating systems. That is why I am providing rules for both, and the tools to work under either OS. When you type **make** under Windows you will get linked.exe. But if you compile under Linux the executable will be named linked, without the .exe extension. With Windows, the OS recognizes any .exe file as executable. Under Linux designating a file as executable is part of its permissions. Don't worry, make knows what it is doing under either OS, and gives you the right output.

When Linux looks through its path for executable code, it won't find linked anywhere. You need to type **./linked** which tells the OS to look in the current working directory. You have to specify the path for executables if they're outside your $PATH variable, that is why typing **linked** isn't enough. Remember how we used ../ to move up one level in the file system? When we type **./** we are referencing the present level in the file system. You are telling the operating system to look for the executable 'linked' located in the present level. If you do not add the ./ before your new executable file, the operating system will scan the environmental variable PATH for linked. It won't find it because your present location is not listed in PATH. By typing ./ first you are telling the OS where to look. This is one of the ways Windows and Linux file systems are a little different.

- Read linked.c in your editor to find how the code creates the application's output

Why read the code? The more code you read, and understand, the better you will be able to write your own code. I learn new techniques by doing this. I want you to make reading code a habit. Find it online, in a book, or in these lessons. I often incorporate other people's ideas into my code. If they wrote cleaner code than I do then I emulate them. If there is a new algorithm I don't understand the first thing I will do is find and read example code. Then I will implement their code on my machine, implementing any changes which may be necessary. Once it is working, I will change it more and more until I become familiar with how everything works. After I have rewritten it a few times the code becomes "mine". In some cases I read their code but don't quite understand it. That makes me write my own version of it until I understand how my code works. Reading their code was enough to teach me how to write my own version.

By learning to think in bigger chunks you can read code faster and more thoroughly. This course of lessons provides you with examples of enough different types of programs and algorithms so you can solve your own problems. Compare what you want to do with what I have provided you and pick the parts you need. Reading code will increase your programming vocabulary helping you build more efficient sets of tools. There are many ways to solve a problem but some of them are more elegant, or they take less space in memory, or they run faster using fewer resources.

IV Describe code line by line

Lines 1 through 8 form the standard comment header. I like to include the name of the file, my name, a date, and notes about the contents of the file.

Lines 9 through 11 include our standard files. stdio.h for input/output routines, stdlib.h for rand() and srand(), with time.h for the function time().

Lines 13 through 19 creates a structure called db. I called it db, for database, because I am going to use it to create records in an employee list. Our new structure, db, has four fields. Two integer fields, one floating point field, and one field for a pointer. This is a primitive example of a node structure for a singly linked list. By keeping the data and the pointer in the same node is expedient, but not normal. I wanted to show you both nodes and a linked list. In this case the node contains both the data and the structure for the linked list. In future lessons we will abstract the data away from the structure by creating a separate data node pointed to from the structure node. Don't worry it will become more clear as we progress. Just think of the struct db as one card in your recipe box with the fields representing the contents of the recipe itself.

Line 21 defines four pointers to db structures. head is used as a reference pointer, while ep and the others are used as working pointers which frequently change.

In main we seed the pseudo-random number generator with the current time on line 25; then each random number sequence will be unique, or as close as computers will allow. Next, we allocate memory space one db structure in size, then aim the pointer temp at it.

```
temp = (struct db *) malloc (sizeof(struct db));
```

introduces some new concepts. You are assigning a pointer to a new chunk of memory. malloc() grabs a struct db sized chunk of memory from the heap then points at it with temp. Notice the cast, (struct db *) which changes the void * returned by malloc() into a struct db * pointer, which fits our problem. This is a standard idiom of C/C++.

Now we play with pointers. As I have advised, I set the new node's next pointer to a guard value. This prevents us from walking off of the end of the list.

```
temp->next = NULL;
```

assigns the new node's next pointer a NULL. That means it points to the memory location 00000000x.

Line 30 assigns the company founder’s data to the first node. I chose the id value of 42 to remember Douglas Adams. Now we generate a random age between 20 and 60 years old. This is done by using the rand() function and the modulo operator %. rand() % 40 gives me a random number from 0 to 40 which I offset by 20 to avoid the child labor laws. A random salary is generated for the founder too: 35080.15 + (rand() % 1250). I was unsure of the binding strength difference (M, D, A, S, etc.) between + and % so used parentheses to make sure. An alternate interpretation would add 35080.15 to rand() then perform the modulo 1250 operation. Definitely not what I wanted. Using parenthesis assures you will receive the behavior you expect.

Lastly, we set head equal to temp, which points the reference pointer head at the new list. Since both temp and head are pointers to the same kind of structure an equal sign makes them the point to the same location. Since we will add new nodes at the end of the list head will never change value. That way we can use head as a reference throughout the rest of the file.

Now that we have created and filled the first node, we are ready to add the rest of the company’s employees. The for loop, between lines 37 and 46, does this in an explicit manner detailing each step. Line 39 allocates enough space for another db node then points at that new space with node. Line 40 aims the end of the current list at our new node. Then line 41 makes sure we add the list termination value NULL. That NULL is a guard value. Once you see it you'll know you are at the end of a list, or the bottom of a tree, or the end of whichever data structure you're using. The temp pointer also needs to change so it points at the new node instead of the previous one. This is how we keep track of where to add our new node. You need this extra pointer to locate the end of the list because this is a singly linked list.

Now we are ready to fill the newly linked node with data. temp→id is set equal to our for loop index just for an example. Remember, if you add too many new employees you will run over the founder’s id value at location 42. You will need to keep track of this for the future. But for this simple example we will let it pass. Our employee’s ages use the same algorithm as did the founder but the salaries are generated a little differently. Line 45 has an example of casting. (float) takes the integer value passed to it by ((rand() % 40000) + 60000) and converts the number from an integer into a floating point value. The for loop continues building the list until we have 27 employees in it along with our founder.

Lines 48 through 53 demonstrate a very standard pattern which will be used throughout these lessons. Set your employee pointer at the head of the list with ep = head; These are both node pointers but head is fixed while ep is the pointer which scans the list. I call the latter a working pointer. You will normally see head and tail pointers used in linked lists and root pointers used for trees. They are used as points of reference while the working pointers are mutable.

Once you are pointing at the head of the list with your working pointer ep, you can start scanning with while (ep != NULL). This phrase says you should loop until your pointer equals NULL. Remember, you set your last node’s next pointer to NULL each time you added a node, so this value is set. If you don’t set a pointer variable equal to NULL, or somewhere else that is memorable, it is undefined and can point anywhere. Normally, it points at someplace bad which will crash your application. So ALWAYS initialize your pointers! Inside the while loop we print our employee’s data on line 51 and then “increment” the pointer. Writing ep = ep→next is the standard idiom for pointing at the next node in a linked list. The lines 48, 49, and 52 are the basis you use to ‘walk’ any linked list from head to tail. You will see those three lines repeated quite often throughout this course.

Lines 55 through 57 break C’s strict standard because they are not at the beginning of the function, but appear associated with where they are used, which is a standard practice in C++. Lines 59 through 66 reprise the standard list walking idiom only this time they sum the ages and salaries of the employees. By incrementing count, on line 64, you can use it to calculate averages. This module calculates total payroll, the average paycheck, and the average age of the employees.

## V	Describe makefile line by line

Line 3 tells make to use g++ as the compiler. This is necessary because I broke strict C rules. Lines 54, 55, and 56 of linked.c should be located at the beginning of the main() function. I put them in their present location because they trace the narrative better there than they do in their proper spot. C++ doesn’t mind this at all. That does not mean this is a C++ program, I just used that one feature of the language. If you change line 3 to CC=gcc and run make you can read the error messages it generates. Moving those offending lines should allow you to use gcc without complaints. As these lessons progress, you will use more and more C++ features until you are using it in its entirety in lesson S4L5, ecology.

With lines 5 and 6 we link the object file to create the executable linked (linked.exe under Windows).

The object file is compiled by lines 8 and 9.

Lines 11 through 13 are used for clean up while you are editing this file. rm linked.o removes the object file from the directory. Now, if you call make, the compiler is forced to recreate the object file, which forces the linker to recreate the application, thus giving you a clean build. Line 13 removes the executable file. As an aside – if you are working on these lesson on a Windows box rm linked.exe is the proper form. If you are using a Linux computer this line should read rm linked because the .exe extension is not used. But remember all of the lessons of sections one and two will compile and run under either OS.

A few extra items have been include but are not used yet. Lines 24 and 25 will link and compile your application with debug information included. If you type **make debug** you will produce the application linked4. This then can be run under gdb to debug the code. Since this application is so simple debugging it will be quick.

The lines 28 and 29 produce another application, only this one produces profiling information. Profiling tells you which functions have been called the most times with how much time was spent in each one. Since there is only one function in this application profiling it would be useless. But these are notes for the future. I tend to keep notes for building files inside of their makefiles like I did here. Helps me find them when I need to use them.

VI Deeper

I added a singly linked list data structure by modifying structures.c By using the same code, I can show you the additions I made in a comfortable setting. Once you have read the code, compiled the code, and run it a few times I want you to edit the source file, making changes where you like, then compiling and running it again. Breaking the code and fixing it helps me learn quickly. Especially if I start with working code. Read the errors you generate along the way. Knowing what each error means and how to fix it is valuable.

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 give 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 with many comments so you can understand each step. Make sure 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 exactly. That will come with time as you assimilate these idioms.

In linked.c, lines 39 through 45, you will see I allocated memory space for the node first, aimed its pointers, and pointed to the data afterward. I like to think of it as building the scaffolding then filling it with data. Once all the pointers are in the right place you add the payload.

Yes, in C pointers are a big deal. But C was designed around memory and how you can examine memory. Some folks complain about C's odd behaviors, but don't seem to realize how closely it lets you work to the bare metal. More modern languages abstract you away from the metal so you can't really get a feel for what is going on in memory.

Memory deallocation is done differently for each language. It is often called garbage collection. OSes do it too so answering the question best is ‘it depends’. Java and Python garbage collect automatically but have their own little gotchas. In the middle of something, they may decide to garbage collect, instead of doing the programmer's bidding. Bogs things down immensely. In C you are responsible for doing it yourself. In applications which do something then exit it is left up to the OS to recover the memory. But, in a long running application, the programmer needs to be more careful. So it is best to release the memory you have allocated. In C you use malloc() and free(). In C++ use new() and delete().

Always set any 'floating' pointers to NULL. It really helps debugging. Yes, you immediately assign it to your newest node but trust me, consistency really helps. When you are creating a new node you start with all unassigned nodes pointing at NULL. Yes, each node in a linked list is called just that - a node. What is in your node is up to you.

You could rewrite the code in a new order. I do it explicitly so I know exactly what I am doing. Trust me, when you are deleting a node from a doubly-linked list there are a lot of pointers getting changed.

Keeping all that straight takes a bit of drawing and some discipline. Standardizing your methods helps you find any bugs you may have introduced.

Yes, you can eliminate variables by making your argument more complex. But there is a balance here: if adding a new variable helps you better understand the program flow, do it. If you can understand AND REMEMBER how your code works then do it the more complex way. If I do something complex I make sure to add a bunch of comments saying what I did and why.

Search for linked lists online. You'll find they need a lot of pictures to get the concepts across. You need to see those drawings. I usually use a piece of paper to do the work since I did not have those pictures when I learned how to write linked lists or trees. The 'net really does help a lot :)

Here is a good place to discuss linking and libraries. When you see an application with #include <stdio.h> in it you know this app will have some form of input/output statements in it. In this app you also see #include <stdlib.h> as well as the time.h header previously mentioned. The stdlib is where you'll find rand() and srand() defined. As you write code you will need to check each command to see what library it is in. You will learn the basics early. You can always look up a command on the web to see which library it is in. Then you'll know whether you need to add another #include line in your code. Your compiler will remind you with friendly linking error messages. When you see one of them it is time to check which new functions you have called. Comment out code and recompile until you no longer have a link error. Your code may not work but now you know where the offending code is located. That will narrow your search a great deal. Start looking up new or infrequently used commands on the web. Check which library they are in. Normally all is well if you add another #include file. But there are times you will have to add a library to the linking code in your make file. Once the right #include files and the correct libraries are in place your code will compile without link errors. I should remind you that I have done all this work for you in this course of study. I am just presaging what you will need to do in your future.

VII Assignments

1. Change the data struct by adding or deleting fields.
2. Change salary to a double. Make sure to change the associated print statement too.
3. Modify the range of ages and salaries offered by your firm. (Lines 31, 32, 44, 45)
4. Modify the print loop to only display records matching certain criteria.
5. If the age is less than 50 and the salary is over 60,000 then print the whole record.
6. Add a print statement so you can keep track of the address of your employee pointer ep. The printf() statement needs a “%p” to print a pointer. Printf(“%p”, ep); will get you started.

7. Change the struct db by adding a field. Then see how the pointer print statement changes address values by a different step size reflecting the size of your new struct db.
8. Think about how tightly coupled each db struct is with the linked list data structure by including the data and the link in the same node. Why is this an easy way to do something but not the way you would really do it for your working code?

VII Links

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

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

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

https://docs.microsoft.com/en-us/cpp/preprocessor/hash-include-directive-c-cpp?view=vs-2017

https://askubuntu.com/questions/358633/what-does-mean-in-linux-shell

https://askubuntu.com/questions/320632/why-do-i-need-to-type-before-executing-a-program-in-the-current-directory

https://unix.stackexchange.com/questions/77113/what-does-mean

https://www.123-reg.co.uk/support/servers/basic-linux-commands/

http://faculty.ucr.edu/~tgirke/Documents/UNIX/linux_manual.html

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

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

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

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

https://www.geeksforgeeks.org/type-conversion-c/

https://developerinsider.co/type-casting-c-programming/

https://www.cprogramming.com/tutorial/lesson11.html

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

https://www.geeksforgeeks.org/data-structures/linked-list/

https://www.geeksforgeeks.org/linked-list-set-1-introduction/

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

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