How I built a simple database with the parts we have already created.

An essay on solving a problem from the concept and a pile of parts.

I reviewed the first nine lessons and thought of a way to tie it all together. A simple database you can use for every day or for one of your clubs. This database should have functions for input, for editing, for deletion, for saving, for searching, and for printing reports. You should also be able to merge lists. You need a structure for each record you want to create. You learned that in lesson 3 so we'll grab a few parts from that tool kit. You created most of the records during the lessons as random numbers or random choices. Now you will have to create an input function. Editing a record is closely related to creating a new one. But, instead of creating the record, you retrieve it from memory, or your disk drive. Once it is in memory, you can simply overwrite the fields which need fixing with parts of your input routine. Here is a good place to factor functions, so you can reuse them later.

You have also learned how to search lists and print them as reports. Once again, we can find some reusable functions between the print and save routines. If you are printing a full report, expressing each field, you are almost there. The only difference between printing to the screen, and saving to a disk file, is the use of printf() for the former and fprintf() for the latter. So if you are careful, you can reuse quite a bit of code between the two functions by simple copying and pasting.

Work you need to add: input/edit, read/save/print, and merge if you like. You have most of the framework already available from previous lessons. Grab the necessary tools out of each of the lessons, put them into a new working folder, and build your makefile. Within a few minutes you'll have a few randomly built records to use for testing. Then start putting more parts together, reworking them to fit more functions.

It is time to start thinking about design details. Where am I going to store my database? Should the command-line interface to it be available across the file system? On Windows, you can edit the PATH command by adding the executable's path. What kind of lists are you going to save as databases? If you write your code cleverly, you can open records from any of a variety of databases you create, and write reports from queries on their unions or intersections. Read about relational databases for a bit more depth on that idea. Normal databases keep only one fixed type of record per list. I am hypothesizing a hybrid system. You create and use the single record format, but allow for the query language you create to open more types of records simultaneously. So the search step grabs stuff from any list which fits the if/then criteria, puts them together into a temporary record, and then prints the report.

Here's an example. You have a list of your friends' names and birthdays, as well as other data on one list. You have your friends' addresses and other data on another list. On a third list you have the collection of all the members of all the clubs you are in. You created that one from all your club lists. Using those three lists, write a query which asks for all of your friends who are in one of your clubs who is having a birthday this month, so you can create an address list of them for birthday cards. Thus you need the friends' birthday list, the club list, and your friends' address list. Basically you search the friends' birthday list for the month in question. Then take that list of friends' names and search the club

list. Only matching names get kept in the list. Now take that list and search the address list with it. After three searches you have your final list. Think of a way to automate some of that work.

Now you have some idea of your options. Keeping an extensive variety of data, on a vast number of individuals, is a very hard list to maintain. Or you can keep a variety of simpler lists, where you match a few fields to display unions or intersections of them. Using sequential searching is a quick and dirty way to create a relational database.

For now, let's think about the single list database system. However, it is always good to have some future plans in mind while you are creating the underlying structure. I think the best place to show what I mean is by creating a standard way of opening and saving all records. Save the description of a single record as the first section of the saved file for any database list. The reading program will read the description section data and be ready to process the following N records. It seems to me that descriptive segment would be much like the definition file of a struct. Such as:

```
struct db     // create an employee record
   {
   int id;
   int age;
   float salary;
   };
```

My disk saving function will write "db, int id, int age, float salary" followed by N, the number of records of this type. That way the reading program will know it needs to allocate N chunks of memory each struct db in size. Once it allocates the memory space, then N records of type db are read from disk and written into your newly allocated db space. By saving the format of each type of record you have created a more general, and more useful function. It won't just read one kind of database record but all of them. You will need to work on any symbols or formats you want to use for your description segment. If they are all structs, then the first token before the first comma is the name of the struct type.

This is a good time to think about parsing, strings, tokens, and delimiters. Parsing is the act of cutting up strings between delimiters into tokens. You have your list: "db, int id, int age, float salary" now you can start parsing. Store everything into a new string, character by character until you hit the delimiter character ',' . Once you reach that comma, store your new string (first token) into a more permanent residence, like structType = first token. The comma is not copied to the token, it is only used as a marker to show the end of a token. I keep saying token but have yet to tell you what it is. A token is a chunk of code, a word of text, a single command, a number, etc. In computer code you have three kinds of tokens: numbers, commands, and addresses. Numbers can represent most anything from text to graphics to actual numbers. In the latter case you need to make sure you know what kind of number you're talking about: integer, floating point, etc. They look different and have different memory requirements.

Now you are walking down the data string, reading from the disk file, clipping off tokens comma by comma. Once you get to the end of the file you have a list of tokens to deal with. You'll write some code which reads each token, and does what it is told to do. In my simple case, with a token string of "db, int id, int age, float salary", I have to interpret first the name of the struct (db), then a number of

type int and name id followed by another number of type int and name age. Next, we read a number of type float and name salary. The parsing code (also called a tokenizer) took “db, int id, int age, float salary” and broke it into “db”, “int”, “id”, “int”, “age”, “float”, “salary”. Your interpreter code will need to recreate the form:

```
struct db     // create an employee record
   {
   int id;
   int age;
   float salary;
   };
```

from those seven tokens. It will need to know the first token is the name of a struct. It will need to know that both int and float are required to have names immediately following both. Just a few rules to create to interpret structs, ints, and floats. Another one or two rules for interpreting strings and you’ll be fine.

As long as the number of types of descriptive tokens held in our data files is not too large, then the code to parse and interpret them can be written by hand. Otherwise, there are tools to write parsers automatically. Lex comes to mind. But let’s keep things simple, with a token list of only ints, floats, doubles, strings, and maybe structs. But I digress, let us stick to simple for now. I just wanted to introduce where this could go to explain some of the underlying framework preparation for the next level.

I could write this simply, but rigidly, by reading only a fixed struct db format. Then I would have to rewrite each part, for each list, making more work for myself, and more places for errors to creep in. By generalizing here I can save work later. Reading and writing generic records is only a matter of adding a descriptive segment to the data file and interpreting it correctly when reading.

I need to think about what types of things I want to save, and what kind of lists I want to generate, or clip from the web. Yes, clip. There are a lot of tables of data ready to copy and paste somewhere. You can paste it to an editor, or better yet into a spreadsheet. With a spreadsheet you can import tabular data and make comma separated value (CSV) files out of them. This trick does save a lot of time building tables. If the input data is available, I can find a way to massage it into a more usable format. Now all you need to do to make that data file fit your generic format, is add the descriptive section at the beginning.

While it might seem I am getting way ahead of myself, I am in fact preparing a way to do less work, more correctly. Thinking has a way of keeping you from working too hard. Why should I do something twice when I can think about doing it right the first time, and then doing it. Hmmm… soapbox … whew. OK then. We have the parts basic to all programming: input, processing, and output. Each of these parts is quite different for any given application, but the thinking is the same. You get data. You work on that data. You write a report about that data.

In this case – a simple database system for small scale purposes – has each of those parts. You can add input by creating new records manually or by loading them from a disk file. You can process your records by sorting them on various attribute values. You can create output as stored files, or as printed

reports, or as either textual or graphical displays. I propose we stick with simple, and keep fewer than ten thousand records. At first a simple, single record type from a single file system. Multiple files, with a variety of records, can come later. Use the method I have described to read and write generic record lists. Then you'll be ready for more complex programming while you are allowing yourself to create multiple lists. By creating them as you go, you will be ready by the time you have conquered multi-file queries. Work simply but dream big.

Let's start grabbing parts and get to building. I think you might like to follow along as I take the idea I described above and beat it into code. I hope to use many of the parts from the first sequence of lessons. I will have to adapt them, and add new functions, but you will see how a lot of it is already in your tool kit. Grab structs from S1L2 or later. Linked lists from S1L3 and later. Sorting from S1L4 & S1L5. Trees from lesson S1L6. Searching from lesson S1L7. And for any string manipulation I'll grab parts of lesson S1L8.

- Make records as structs
- * Input/output random routines for now. Need work.
- Reports – printf() which leads to * read and * save functions.
- Sorting with trees or functions.
- Searching with indexed arrays
- * Delete is part of linked lists but has not been used to my knowledge. (S2L8 Orbit's prune routine?) At least not during the first eight lessons. However, the code has been around the block more than once and knows the way.

Let's go back through that list and add an asterisk for any * future work. Make merging two dissimilar lists an assignment :) Hee hee hee. Another good assignment is the multiple list search routine I described. Sounds like fun. Interesting to see how it fits into this small format database.

The first thing is to start creating lists and saving them to disk. Then read them back. Once I can read and write my lists, I can sort and search them. When that works it is a close step to writing reports. Editing and deleting can come later.

I will start with a fixed format structure for the first steps. Then once I get the structure set up I will switch to using generic format structure lists. Just need to work on the parsing code for when I scan the description segment string.

Need a tokenizer to spit out – names, types, then interpret each of them into

```
struct db     // create an employee record
   {
   int id;
   int age;
   float salary;
   };
```

Once you have created your struct you can then read the next line of the list file, N, which tells you how many of these kind of structs to read in next. Or, should we save these as comma separated values

on the disk? So instead of record0, record1, record2, record… you will see id, age, salary, id, age, salary, id, age, salary, … and so on.

The file will look like this if you open it in a text editor (the comments are not in the data file but are here for reference)

```
db          // name of struct as a string
int, id     // type, name pairs
int, age
float, salary
            // need way to designate end of struct.
            // look for } ??
            // or better yet use ; as the delimiter of chunks
N           // number of records of the prescribed structure
id, age, salary
.
.
.
id, age, salary
```

Just writing down what I needed to do, helps me figure out how to do it. Such as the need for a terminating character for any segment. Now, let's describe which segments and why. Here we have the record description segment as comma delimited pairs on sequential lines. Then comes the number N (do we terminate this with a ; too?) After that the list of records which you could grab by looping on N. {*nota bene* – no comma need be used, all that is required is white space}

From this point forward I am writing in a journal style, showing day by day progress. Examine the files in the snapshots folder to see the code at various points during its development.

2:02 PM November 12, 2018

I built the framework of database.c, database.h, and the makefile. The makefile is very generic, the header file nearly blank, but the main file has copies of all of the code from lessons 3, 5, and 6 as comments. I also added some new code. A list of #include statements and main() function which holds only comments. It is time to start pulling out the parts I need and putting them into database.h or database.c I will try to keep all of the parts in those two files and document the process of creating our finished app here, in this report. I want you to see all the thought processes I go through, and my decisions along the way. I am also going to show how I plan ahead for future work at this stage, to save effort and prevent more errors.

My first goal is to get a linked list of records printed to the screen. I'll stick with randomly crafted records for now. Then I will add input, edit, delete, and save since they are somewhat related, especially the user interface (UI) parts. I created the cruft.c file to hold parts as I cull them from database.c They'll be around if I need them again, but I am taking the parts I need and moving the others. A little fiddling, a little fixing of the makefile. OK then, it compiles, and prints a report. Onward!

5:19 PM November 12, 2018 .

Right now the list is an array. Make it a doubly linked list. Then add input and delete node functions. Moved the modules from sortLink.c around then added a rudimentary UI. This UI is very old school, simply a menu printed on the screen, where you the user is asked for task choices. The tasks we want to use on a regular basis are: add a new record, delete an existing record, print the whole list. Once these are working I'll add functions to edit an existing record, sort the list on a chosen attribute, search for a single record, search for records which satisfy certain criteria, print only those records, and save all records (with some code to read all records on start up). For now I will stick with a specific data structure, or I'll rewrite it to fit the list I want to create. Writing a few functions to read lists and create new databases should be explored. Simple for now. Bug free and reusable. Do you need to store, modify, and query a few hundred records? Then this simple database should work for your list maintenance.

Currently writing addRecord() routine to be called from your menu choice # 1. I can now addRecord()s with the clunky UI. I will add features before I clean it up. Once I build a few parts there should be some common routines to factor out and clean up. The user interface portion of the application is where the user will spend all of their time. Once you get three or four of the choices working you should have enough code to start factoring out a better user interface. Remember how life used to be, before the Mac, with its graphic user interface came along? Back when UIs were built from characters, reverse characters, and maybe even color if your machine had it. Using those funky characters available in the character generator kits was common, but I'm going to stick with plain and simple alphanumerics. Simple, fast, reliable, extensible, and maintainable.

Because you are only working with one record format per list, you will tend to create all encompassing structures, to hold ALL the things you want to archive. Start thinking about how to open and query two dissimilar lists as soon as possible. Once you get the query system working for that case, your design goals can expand quite a great deal. Say the case where you have a list with employee id, first name, last name, address, telephone number and you have another list with employee id, age, salary, and seniority. First, get to where you can pair items from either list using the employee id attribute. That will give you a report including a union of the two lists. To make things more complex, now print a report of the union of both lists, where the age of the employee is above 45 AND the salary is greater than or equal to $70,000. My first thought on how to do this would be to print a report with the union of both lists, but only print those records which have the age greater than 45 AND have a salary greater than or equal to $70,000. Implementing the query system should be interesting. Write down and manually solve a number of cases to see if you can generalize a procedure.

When you get that solved you'll have a powerful system. By combining only two lists you can accomplish good and useful things. Generalizing storage, query, and report generation will open up new paths, for new techniques. Start simple, but aim for a means to query two lists at a time. After that works think about three or more lists in one query. Look at SQL for ideas and methods. Don't reinvent the wheel, even if you have to craft one yourself. Copy and paste is your friend. Being able to modify running code to do something else is the goal for this course. Now, work on sorting by the criterion of choice. Generalize the procedure as much as possible to lessen errors.

Delete one node based on id choice etc. Then automate deletions of every record which satisfies a given query. This reminds me to think about list creation from a query. Or creating a new (temporary?) list from the union of two other lists. Or the union of those lists, satisfying some query. Time to look at query languages to see if we can glean any notes from them. SELECT, FROM, and WHERE of SQL is a good system to emulate. Get some examples and see if you can rough out routines to duplicate their function.

I worked on adding records by hand, saving them, and then retrieving them. I am about to make sort() work on more fields. I also created a print() function to show the entire list. It is time to take a snapshot of the project as it exists right now so you can see how things are progressing, and how they will change in the future.

{.../1 tools/S1LFP_database/snapshots/database November 13.c}

I have added parts from S1L7 search, S1L8 string, and S1L13 queue. Once I get the menu structure working with adding, deleting, reading, writing, and printing I will start thinking about generalizing things, to make life simpler. Moving code and variables to database.h would clean things up too. For now I want to be able find them with a quick search or scan. I want to start adding fields to the structure which require strings. Names, addresses, telephone numbers, dates – basically anything that is not an integer, float, or double needs string manipulation routines.

I made a copy of database.c as a snapshot in time. I plan to take a few more as I progress so you can see the path I followed to get to my goal. Hopefully, I can get all the parts to fit. I am currently trying to figure out how to sort on a key instead of hard wiring each field. Once that works I can switch > and <= to reverse the order in my report.

There are more modules written which can be called separately. Time to use a forever loop and start implementing the menu structure. Then you can run multiple tests between each edit compile loop. Added more to the menu structure and filled in a few stubs. Currently the while loop displays the menu and then runs the choice (if it's ready yet). Really helps testing. Now to add input for each choice to add versatility. Need to figure out how to choose sort field for a report. Need to generate reports with your choice of fields in your order. And then only those records which fit the query.

Need to get strings working so you can have a more realistic set of fields for your list's records. If they fit what you need in your life you will actually use the application you are creating.

Things I have added which were not in the previous lessons. scanf(), menus, strings in structs, ???

A database system is easy to explain. It is an external list or lists of data, a set of tools to manipulate those data and create reports from them. An external list only means it is held on disk as a file. You don't really need to pull all of the data into memory, but for this simple system I'll read all of a list or lists, into memory. Memory is cheap and processors are fast. When those facts were not true my design criteria were different. Sorting a list externally is no fun. I have done it across eight floppy drives and it took a lot of time, and a lot of disk handling. It was a pain.

Because the system is a set of tools, working on a set of data, I can write the tools as functions. As you may have noticed when I took the first snapshot I was just starting to form function modules and call

them inside main. Now I have a loop based on the menu. You call a function, it does its work on the list(s), and then asks for the next function. Up to the point where you choose exit and you drop out of the loop and the app. I need to add a function which creates a new list from the old list and a query. If I can do that once then I can chain my queries into something complex. If I can create a generic list then I can span my queries across different kinds of lists. But stick to simple for now, and let the ideas grow as you write code.

There is the problem of writing sort routines for every field in a record. I could write each case individually but that causes an explosion of cases when I change my record's format. If I think of a way to generalize I can have two sort routines: one for numbers the other for text. So we can tell the sort routine - 'sort on field 3 which happens to be text' or 'sort on field 12 which is a number'. The field type and field name are part of the generic record header maintained on the disk file. So it looks like it is forming up to be an array of fields[] so you can call field[6] or field[7] of type field[4].type with a name of field[4].name Hmmm… now try typing that a bit more formally to start dragging it from pseudo code to C.

Once the input/create functions are written, and you have a few records to examine, you can build the menu loop. After you get the first few functions running it is easy to write more. In fact, this is a good way to write larger programs by yourself. You can spend an hour or two writing a new function. Test it and leave the system bug free when you're done working for the day. The next day you can type make and get a clean build. It is always good to start your day with a clean build so you know you have a stable system. If you find you cannot get rid of a bug before you're done for the day make sure to comment it for the next day. Then make sure to comment out any reference to the new code in the stable build.

Fix the bugs the next day, before the code gets stale in your memory. If you need to let your code rest for a few days, it is always best to leave it bug free, with each new function working properly. This style of coding can create a number of global variables. It can also lead to functions which were never anticipated in the initial design. When you get too many global variables it is easy to induce interactions (coupling) and errors. This is a good time to take a look at the overall design. Have you outgrown your initial framework? How much of your code can be moved intact to a new framework? Did you discover a better way of doing things which requires a major rewrite? Balance the costs with the benefits. You may find it better to live with the rough edges, than spend the time rebuilding. But, it is good practice to craft a new design from an overgrown project. You may find a simpler, more efficient way to do things which is worth the effort to achieve. You have already written the tools all you need to do is rearrange and generalize them.

The menu loop structure works for most input, process, output type programs. You have your data file with your tools to manipulate it and write reports. Fairly standard, you should be able to boilerplate the code in a day or two. When you use a command-line menu system you can write very compact code. Any time you can do that you have saved time in a number of ways. You don't need to write as many lines of code to have a working system. You also provide less space for bugs to appear. Smaller code is also easier to read so you can keep all the paths in your head, ready to follow each of them. Tracing program flow as you read your code is why you write it plainly. Plus, if you really need to have a graphic user interface, you already have the application code written and tested. All you need to do

now is write the menus with graphics code. A series of copy and paste commands and you're most of the way there.

Need to smooth out the menu structure and choice flow. I could enforce a save() on closing the application by just adding save() near the end. I could also enforce read() by adding it before the menu is called. I could leave both of these functions strictly up to the user. By writing the code in functions like these it is easy to use them in various ways; just string the beads in different patterns.

I have mentioned the term factoring more than once. I had better take the time to truly explain what I mean and how it is done. Imagine you are in the middle of creating an application like the database system we are working on right now. You are writing function after function which are nearly the same in form. You realize with only a little modification you can pull out a chunk of code and generalize it for regular use. Look at the print() function for instance. It is currently written to start at the head of the list and write every field until you reach the end of the file. What you would like to do is write reports which are sorted on various fields (yes, more than one sort at a time) or for a range of records. You need to find all the places in database.c where you want to print a single record, all the records, or some subset of the records and craft a function to use in all those cases. Then every place where you have many lines of code you can replace them by adding your factored function newPrint().

After working on database.c, I realized I was limiting myself if I could only use a struct as my record. Because a struct cannot be created after compile time I had to work around the problem. I mentioned generic lists previously, I explored the idea a little deeper by creating generic fields. My thinking was that if I could build a field which could be stored on disk, retrieved, and edited all under programmatic control I could then create records, and my generic list.

I wrote the code in the generic folder to test my ideas on a generic field. I am only going to use three data types in my generic field: integer, double, and string. Each variable has a name, a type, and a value. The line int x = 15; declares the variable named x as an integer and assigned it the value 15. Once I saw this I wrote my generic field. Until I got to the value part that is. How do I represent three different data types' values in the same slot. Then I remembered unions. I knew they could hold either int or double but how about a pointer? Well the best way to find out what your compiler permits is to write some code to test it. I wrote that code and the code necessary to build the the generic fields.

Now I can start writing a record as an array of fields – field[3].type or field[1].value. This allows me to sort on any field by passing an index to my field array. Next some functions to save and retrieve generic lists and the functions to build one manually, automatically, or semi-automatically.

I can complete the database program the way it is now. With one list, hard wired structs for records, and a number of sorting routines. All the parts necessary for a limited, problem defined, database. But with the code exploration I did in genList.c I could write a system which uses generic lists. That opens the world up for relational database methods. Open a few generic lists which are linked by a key field. Write a query on the chosen lists and create a report. Save that report onto a new generic list or display it in easily readable, scrollable chunks. Sure the extra work necessary is outside of the initial bounds of this exercise: use parts from lessons S1L2 through S1L8 to create a simple database.

Now what to do? I think the best thing is to do what I set out to do – write the simple database from those example lessons. THEN continue on with a rewrite to use generic lists instead of simple, hardwired ones. By keeping good notes along the way, during the completion of the initial task, and through the rewrite into the final task, you will be better served. I'll keep taking snapshots of the code every so often so you can scan through it with the diff command and see how the design is changing. The final task of crafting a generic list database is not overly complex, but it is more so than the code in lessons S1L1 through S1L8. It may be a good segue into the next section of lessons; especially if you choose to continue on with lesson S3L1 in 3 Advanced Tools. Lessons in 2 graphics – S2L0 through S2L9 are simpler but they all use a graphic user interface (GUI) instead of a command-line. The lessons in 3 Advanced Tools are more complex and explore problem space you may never attempt. These are not your run of the mill programs which take 95% of the time of programmers. Sigh. They are fun, and they do stretch your mind. Cool. If you really want mind expansion wait until section 4 Advanced Graphics. Complex algorithms, fancy databases, and lots of colorful, active displays. Transcendentally cool.

I am not sure if describing my design process is helpful or not. Because most of the time I am exploring, more than solving any one problem. However, I am trying to cover a great number of problem types so they are in your memory. Once I know what type of program I am asked to write I can start scanning my memory for examples. Or parts of programs I have written which may be applicable. When I have written my notes, and collected my thoughts, I start gathering those parts into a single file. I like to begin my projects with only a few files. The application code and the makefile at best. But I may need one or two more for a graphics app. Collect the parts then start roughing out my notes into pseudo code for the initial structure. Get to a command loop then compile. It doesn't need to do anything except loop through commands which do nothing. For a command-line driven app write a function to print some data. For GUI code, write a function to draw something to the display. I usually include a command line window with my graphics apps, so I can print out data easily from any point in the code. Really helps for debugging purposes and sanity checks.

When I write a function I add test routines and structures around it so I can test the function on various input data. Sometime I write an external framework to test the function as I am developing it. Write a function to supply the new function with input. Write a routine to display the function's output. Write a loop around the function to test it repeatedly with randomly generated input. Make sure to test with incorrect as well as properly formatted data. How does your input scanner handle negative numbers? Numbers with a dollar sign or a comma in them? **Don't annoy the user.** They're using your code, treat them well. Error trap so you can recover gracefully. Don't just let the system crash. Try to parse out what they meant or simply ask them to re-enter the data with a polite note explaining why their first try was unsuccessful.

I will continue adding notes on my design decisions along the way. If I do this for a few applications you will start to see my methods. These methods are good for projects of less than ten thousand lines of code. This is about the limit of what one person can keep under control. If a project gets too large it may need more people. That requires a great deal more complexity. How do you separate the tasks between programmers? How do you put the parts together? How do you keep track of variable names? You will need a versioning system to make sure program changes are done in an atomic way.

When there is a case of merging two chunks of code covering the same function you'll need to sit down and talk it out. However, large projects are often groups of programs around the same data set. Each programmer can have a little more freedom but it is best to keep your format and style consistent across the project. A good editor with an instant format translation tool is very, very handy. Once you develop a format preference you will work best in it and it alone. So formatting should be at the preference of the programmer. When it is easy to reformat the code, then use whichever format makes you happy, and reformat to company standards when you check it in to the versioning software.

{.../1 tools/S1LFP_database/snapshots/database November 15.c}

December 4, 2018 After figuring out how to create a generic field for a more robust database I started research for a few new projects. I want to use NVIDIA's CUDA toolkit so I can access and use my GPU as a parallel processor. Not trivial. But I took a break from that frustration to get back to writing the simple database tool. I will continue adding features to the menu structure and debug them as I go. I just worked on reading and writing functions, as well as a bit of formatting in the report module.

I want to make sure the app can add, delete, store, retrieve, and print records. I need to add trapping to the input module so no strange characters get entered into the data set. Instead of a simple scanf() I'll need to work with an sscanf() function so I can process the input from the resulting string. If the data is not correct for that field, just loop back to the field's prompt and start over again.

I installed gdb on Hermes and Hera so I can debug while I am working on code held in its drives when sitting at Hephaestus or Hestia. This allows me to test the code for both Linux and Windows 7. All I need to do is recompile using make. The Windows app has a .exe attached to it while the Linux version does not. Same code works under either OS, you just need to rebuild it to fit. Examine line 9 of the makefile. See the flag -ggdb3? That tells the compiler to build an executable file which includes everything it needs for easy debugging. Once you are sure you are done debugging use line 10 instead; it will optimize your code without the extra baggage required for debugging.

- Type **gdb database** to start debugging.
- **run** – starts the app running
- **break database.c:84** - sets a break point at the first malloc()
- **break addRecord** - breaks at approximately the same location
- after a break you can get going again by typing **continue** or just **c**
- you can single step in two ways
- **step** or **s** - will step line by line
- **next** or **n** - will step function by function
- **print** or **p** - will print out variables such as print head or print myNode or p myNode→next→id
- **watch salary** - will break each time salary changes

- **backtrace** or **bt** - will print a series of stack frames
- **finish** - runs until the end of the current function
- **delete** or **d** - deletes a break point
- **info breakpoints** - shows all break points
- conditional break point **- break database.c:103 if salary < 20000**

```
p myNode→myData→id  or (*myData).id
```

I have a pointer error in my addRecord() function. I'll need to locate it and fix it.

add a randomRecord() doesn't work correctly either. Fix these problems, and learn more about how to use gdb for this very purpose.

I upgraded my eclipse installation on Hephaestus. I set the formatting to Whitesmiths too. Now, a few key strokes {ctl-shift-f} cleans up any mess of code. As long as I don't actually create a project in their system I can continue to use my existing file system. Now, to enter new code and see how much of fight it is to get it correct. Auto completing "", [], {}, etc requires me to move my hands from the keyboard over to the arrow key. That sucks! I can type faster if auto-complete is not set. Need to turn that off. But more testing is in order, to see how I can make the editor help, instead of hinder my work. {*nota bene*: turning off the matching pair auto gizmo really helps! Much faster typing now.}

If I hit menu item 10 I can open a file I built and the hit 4 to print it. This works fine. Examine how this list is put together with gdb, and some break points during the printing routine so I can see the pointer arrangement.

December 5-6 worked on lists. Now database adds a record, prints a list, saves the records, and retrieves the records. Need to work on more complicated reports and deletion. File statistics? Might be fun too.

{.../1 tools/S1LFP_database/snapshots/database December 5.c}

December 6 cleaned up code. Rearranged so main is at the top, with functions afterward. That meant I needed function declarations at the top. I moved some variables and #include statements into database.h I am currently refining the menu. Now to build sub menu structures underneath some of the choices. Add a record or add many records? Need to work on some string code too, so I can add name or other strings to the database. Work on edit, and report, to make them more useful. Add and refine sort routine.

Got sort routine working on sorting low to high. Need to add a choice of direction in the sub menu and pass it to the sort routine.

Took another snapshot of database.c and started tracking database.h now that I have started adding content to it.

December 8 – delete(), report(), and edit() depend on finding a record(s) which fit certain criteria. This requires each record be unique in some way. This got me thinking about how relational databases

work; each file has a unique key. You can use this unique key to create reports with data from a number of files. I already have an id field in my struct because I was going to use them instead names for anonymity. It also meant I was not going to deal with strings :) Well that has changed. I will soon add names and string handling routines for completeness.

Now I need to make sure my ids are each unique. I began with randomly generated data each with a sequentially generated id. Now that I am going to process user generated data I will need to assure unique ids. I will save the highest id # from my list along with all list data and retrieve it when I reload the list. Starting with new, user generated, data will require a mechanism to set sequential id numbers. When I started thinking about all the places where this id will be used, and how it will be generated, I started thinking of flow charts. But that is not the answer – a state diagram would be better.

Our database system was built module by module to work with a list of records. Each module is written to work with the central list in different ways. Some add, some edit, some save. But each case describes a machine state. In CS classes they enjoy formal grammars and state diagrams. Well here is a chance to use them for something worthwhile.

State 0: no list exists, the user is adding the first record, and the id is set at its initial value (42?)

After I had worked with my state diagram for a minute or two I found I was making my life overly complex. The record's id field is only changed when it is added. Other than that it is used for find(), edit(), delete(), and report(). The highest id number is stored with save() and retrieved with read(). The only open case is when there is no list, the system is being run for the first time. So you must add a guard value. I use 42 to honor Douglas Adams. The first id number imprinted on the first record created is 43, or ++id.

It helps to make a few diagrams, figure out which states exist in your system, and figure out how they interact. A few minutes doing so while watching a Die Hard movie helped a great deal. The movie is on pause now that the pieces have fallen into place. The movie helps with free association but makes typing more difficult. Working through the states also helped me clean up my menu. I've grouped them by common tasks. Add, edit, and delete fit together as do print all, print some, and sort. The last three states also fit together. You read a file to begin your session, after the initial list building phase is over, and store it when you are done. After you're done storing you want to exit and move on with life.

Item 11 – find() is not really necessary when you're running the database system. It is a helper function so it does not need to be exposed to the public. Item 27 – ask for a filename is also a helper function for both save() and read(). It does not need to be exposed to the world either. Adding random records should only be used by the programmer, so it too is hidden from view. That leaves a compact menu of nine useful choices which conforms nicely to my keypad. More work needs to be done on the sub menus for edit(), delete(), and report(). sort()'s sub menu is working fine for now. It goes to show you how a little thinking about structure can help you organize the system and further your design. Now back to the movie.

I had to change one place where I had id++ instead of the correct ++id. Because I have already assigned 42 for the founder's id the numbers following are open for use. Also when I read the stored id number I was looking at the largest one used so the preincrement operator allows me to point to the

next open unique id number. Save now stores the highest id number in use, add preincrements that number, and read gives me the highest number used. Everything is in order. Now to keep working on find, delete, and report. This is shaping up to be a useful tool. I will add string I/O and storage for completeness. I already have the names arrays ready for random record construction using strings. I will implement the string fields throughout the system.

As you may have noticed the menu and switch statement scheme for arranging modules works very well. Adding new functionality is just a matter of adding another function or two plus adding the necessary sub menus. I developed this method for myself during a stint in the hospital. I had to go in for surgery every few weeks and was on pain medication. By structuring my code into smaller chunks I could write large applications while not feeling very well. A year later I was reviewing the code for my boss when I asked him who wrote it. He gave me a funny look and told me, “You did!” I truly have no memory of writing that application. However, when I read it I found good code, well structured, easy to read and easy to comprehend. I took notes on those methods and you see a little example of it right here in our developing database application.

You start by writing your menu so you have something to display. Then write the main loop which interprets your menu choice. Write the switch statement with all (or as many as you can think of) cases. Put a print statement in each case saying what it is and that it is under construction. Test your app now to see if it compiles cleanly. Parsing input is not simple. If you depend on the user being wise enough to only put numbers into numeric slots and following your prompts explicitly (Hint: provide good prompts!) then you don’t have to trap for input outside the box.

Since I am not writing the full blown database of my dreams I have used scanf() for input. It has an exact template for the input in the command. If you don’t match that template it will glitch. For instance: if you are prompted for a salary amount you would type 98,000 and then enter. At that point your app crashes. All because of that comma. You probably did not even notice I used one because it is so natural to do so. But it breaks the “%6d” format required by the scanf statement. So if I were going to be very ambitious I would used sscanf() instead of scanf() for my inputs. The alternate function captures the input into a string. At that point you can add all the traps you like. Send different formats of input down separate logical paths or loop back to the prompt again after printing an accurate error message.

Input trapping is an art. By using sscanf() you can provide large buffers for your input string and then parse it token by token. One way hackers break in is through your input capture system. If you have a fixed sized array for your input the hacker can overrun that array and then add some instructions. Then all you need are a few words of assembly language to point the computer somewhere else. Done correctly you can gain root access to any system. So it is best we guard against that. Always trap every input you will ever receive from anyone except yourself. Mulder was right - trust no one :)

While I may add some trapping later, I do plan to work on the find function. I want it to find and display on all of my fields: id, age, salary, seniority (add more in the future). I want to be able to search for individuals or groups. The only unique field is id so it is always used by the program for editing or deletion. I want to be able to use <, >, and = as well as range controls to select records for editing, deletion, or for a report. I think this may be a good place to use the alternate pointers I added to each

node: *Tnext and *Tprev. Reminds me, I need an alternate head and tail to the new list. I will add them here but this is a good sign a redesign is necessary. Main lists and alternate lists? How can we use them? How can we generalize the functions around them?

This style of coding, where you have a menu and a message handling loop, is very common. Most applications have repositories of data which your functions manipulate. I am sure there is a name for this design pattern but I don't remember it. Just recognize the pattern and you can use your standard toolbox to build a solution. However, if you have chosen the wrong pattern, you will have quite a number of new tools to use when you build your next version. Don't throw away code you have debugged just because you chose the wrong path. Those tools can be generalized and reused elsewhere. Mine your cruft files often.

I think of writing applications like stringing beads. I design, build, and debug those beads so I have them ready to use. Factor them out of your own code as you see them pop up again and again. Generalize them to make them useful in more places. Once you have the framework and the toolkit making a new application can proceed rapidly. Test each module as you add it using modules you have built and tested.

{.../lessons/1 tools/S1LFP_database/snapshots/database December 7.c}

December 9 – I implemented the find function. Currently it can work with any of the four fields as the key. For id, age, or seniority you can search for a single number. With salary you provide a high and a low value so you can search for a range. I intend to use my new find function inside of delete. Since delete(* here) requires a pointer input you will need to scan the list for an id chosen from your find list. Here is why you wanted id to be a unique identifier for any given record. There can be many records with the same age, salary, or seniority. But there can only be one with your id. Scan the list looking for your chosen id. When you walk to the right record, use the pointer value, and call delete (this Pointer). I'd recommend simply remembering that point and pop out of your while loop. Then call delete(the pointer) afterward just to keep things tidy and prevent memory problems through misdirected pointers. {*nota bene*: break from within a while loop works just fine with no pointer problems. Kevin, you worry too much.}

Our database is almost ready. Once I get delete roughed out I'm going to add a first name field so there will be string manipulation too. A working database useful to the student is what I want to demonstrate. Of course you will have to modify the code to fit your types of records. But I will have built all the tools you need to use it on a day to day basis. This will be a good foundation for developing your own database tools for single record type schemes. If you want to move on the relational databases which use multiple record files you will have to develop them yourself. This lesson is about a single record type list and the tools necessary to build, use, and maintain that list. Do you need your own mailing list system? You can build it yourself with a few hours and the tools I have given you.

Work needs to be done on displaying better reports. Look at the range scheme I built into the find tool. Now expand that to include ANDs, ORs, >=, <=, =, or a range of values. Currently you can sort your list on any of its four present fields, then find on a few criteria. Print the report of only the qualifying records. More work needs to be done here. But now that you have seen how I work module by

module, you can work out those details yourself. I am sure there are places you can refine my code farther. It will be great practice for you to make those changes.

Implementing a delete function was very simple once I built a wrapper function around deleteNode(). I had to rename my function deleteID() because delete conflicted with a C++ operator. What this wrapper function does is search the database for your id choice. Once it finds the right record it passes that pointer to the deleteNode() function which removes the node from the list in memory. Remember the stored list is not changed until you overwrite it. So you can do a lot of sorting, deleting, report creation, and records added in your computer's memory all without changing the data saved on your disk.

This is a good place to think about future design. I put two alternate pointers into each node. You could use them to create a new list while leaving the nodes linked in the original list. If you add a little code to save() and read() you could then save and retrieve those new lists. Or you could compare the two lists to one another. Since you have a unique id embedded within each node you can see if what is in one list is also in the other. It would be a primitive AND mechanism to create more elaborate reports.

Those two changes, alternate list threading and saving new lists, have most of the parts you need already in place. It is just a matter of hooking them together properly and testing the results. I am very close to meeting the goal I set for myself: to create a usable database system. Very light, very fast, easy to use, easy to modify. Currently it has zero to no error checking. That would be a good exercise for the student :)

I've taken another snapshot of the code so you can see how it evolved with time, how I modified each part and got it running. And a few scenes of error trapping while integrating new functions. What I have been trying to capture in this narrative, is how I work, and how I create the next modules, while I'm testing the ones I just wrote. I wanted to show how to use the toolbox we have created over the first nine lessons. I demonstrated the menu/message loop pattern, and how easy it is to implement and extend. I think it is a good platform to learn how to create an application in a top down manner. Some thought and a few diagrams on a piece of paper help you get started. Figuring out which modules will solve the problem is normally enough to start thinking of what I've already written, and what needs to be added. Fill in the framework with comment lines and printf() statements saying what will happen in the future.

Once you have that framework filled out test each menu choice to see if it responds with the right text. Now it's time to get to work. Write a module to perform one task from your menu. Maybe to read data from a file which gets printed to the screen. You can easily use your editor to type data directly into a file. Only a few lines of text and numbers and save it as myFile.dat (for instance). Work on your read() function until you can retrieve that data and print it to your display for confirmation. Since save() is often a mirror image of read() it is time to work on that code too. They are also related with add a new record function() (whatever it may be called). In this way you can leverage work you've done before to write new modules.

As an example: my deleteNode() function was clipped from my list application's code. I had not used the function in prior lessons but it was in my toolbox, honed and tested. So I copy pasted and wrote the deleteID() wrapper function to call it. I truly did NOTHING else. I made sure I was passing the function the proper type of input and let the function work. I tested my wrapper to make sure it was passing the correct node location but I DID NOT test deleteNode(). I just grabbed it and used it like a hammer from my tool belt. I know how to use the tool but don't need to build a new one for each job. I just grab it and trust my time tested code. I have been known to rewrite old tools but I try very hard not the change their input, their output, or what they do. I only make them more efficient, easier to use, or more useful. Or translate them into a new language.

If you have been following along through the code changes in the project, you will see places where you could factor out some functions. Especially with the menu structures and input capture. Those will need to be modified to trap for input anomalies. But once you develop a good menu pattern it is best to make it its own function. I have tried to keep the menu simple, using single digit choices for all the major choices. Currently 11 is used for find a record but that is only for testing. Find should not be required directly but only as a helper function for report(), deleteID(), or edit() (which may not be implemented). I'm not sure if I want to keep edit() or not. Currently I can delete the node and add a new one. Since there is so little data in each node it is easy to enter it all again. But if you feel motivated and worthy just go ahead and take a whack at writing your own edit code. There are examples throughout the code you can use.

Here were my design ideas going into this final project for section 1:

```
// Tool kit of parts from lessons 5 sortLink, 6 tree, 3 structure
// parts of 7 search, 8 string, and 13 queue too

////////////////////////////////////////////////////////////////////

// need these functions to really feel like a database
// one that can be useful for day to day stuff
// input/output are randomly generated now
// edit is related to input, don't create new but read and overwrite
// Reports yes but they lead to read() and write()
// Sort by your choice of field by function
// search within an indexed array
// delete a node of the linked list
// lists: arrays, linked lists, trees, queue?
```

I put them into my code file as soon as I started writing comments. I referenced them as I worked, gathering parts from previous lessons, and roughing out the menu structure I would need. Follow the snapshots of the code and you will see how the menu was modified making it more readable and usable. I made sure the sections of the menu were common to each other. I also tried to put each section into the order they most likely would be used. 11 will go away soon and become hidden like a few other menu choices. This menu scheme is very easy to use with a number pad. You could implement the /, *, -, and + keys to perform certain tasks if you so desire. They are easy to type while you are using the keypad.

Now to add one string field so I can incorporate string manipulation tools into the system. So far we have getName() (line ~186)

{.../1 tools/S1LFP_database/snapshots/database December 10.c}

December 12 – while working on strings I had an epiphany. Using the form

```
p = (char *) malloc(50);
printf("a %p\n", p);
t = woman[93];
r = p;                    // points at the beginning of your new memory space
while ((*(p++) = *(t++))) ;
   printf("b %p\n", p);  // Indy, they're looking in the wrong place!!!
p = r;                    // reset pointer back to beginning of memory space

printf("|4| %s | %s |\n", p, woman[93]);
```

is destructive to both pointers. After you exit the while loop both p and t no longer point at where they once did. They both point off the end of each string. So you need a third pointer, r, to use as a bookmark. Over the years I had always thought of them as working pointers even though they never change. Then today while I was doing dishes I compared walking a string with walking a list. Then the light bulb event: when I use a linked list I always include a head pointer. A fixed head pointer. Hmmm… kind of like the r pointer I used right here. Wow! That thought clears up a lot. Just remind myself (or you) that a list of structs is very much the same as a list of characters. A structure uses up a lot of space, while a character takes up only one slot. But the ideas of working both lists are the same. I had never thought of my "keeper" pointer as "head" before. Once I thought of it I realized how much sense it made to do the same thing in both places. Just because the character string is not a linked list does not mean I don't need to remember addresses. Woo hoo! I like it when a thought like that saves work and makes a great memory aid. Previously I had to warm up every time I used strings like they were an odd portion of programming arcana. The veil is lifting :)

I had taken a routine directly out of K&R (page 105 2nd ed.) why wasn't it working? Well, it was working. What I was doing was referring to a memory space right after where I wanted to look not at the beginning. No wonder I was seeing strange things or nothing at all. Once I added the print statements telling me the addresses of the pointers I SAW it. Off by 6? Hmmm… strangely coincident with the number of letters in the string we just copied. Then the little grey cells started working.

So the moral to our tale. It pays to have a map, marks of where you are, and where you were. Using fixed reference points to memory locations so you can keep track of them. Heap space is plentiful, learn how to use it well and you'll have faster, more efficient programs.

Now to get back to adding a random name to each member of our stored list. Then save(), then read(), then print(). After that, all the fiddly bits to clean up string handling.

It was the work of about an hour to implement the changes for the name field in save(), read(), and print(). I generated random names and saved them into my normal file myList.dat. Now to add the addRecord() changes, sort() changes, and find() changes. Clean up those things and keep this journal.

Then I'm done with section one's capstone project: a simple, working database. And now with strings :)

Adding the same printf() statement in five different places showed me I need to factor out my print statements. If I could write a routine to print one record, then pass the ep pointer, I could generalize the print statement. For print all, I would write a wrapper function which walks the list and passes ep to the print(ep) function. Same for find() or print a report. Generalize even further and get print(ep, field1, field2, …) and print only the fields you want. Then a find function could perform as a wrapper accepting values, ranges, equalities or inequalities, ceilings or floors, etc. Find then passes the record pointer, and field choices to print(). Sorting mated with the find() function can make a powerful report generating tool. Now, do you make your reports into savable lists? Creating new lists sounds like it is becoming a task. It might be one for the student though :)

Date fields would be very handy. And they would introduce the tools C has to deal with them. Look in <time.h> for notes.

December 13 – I made another snapshot this morning just before I standardized the print statement. Now there is only one place where you need to make changes for the record display. All you need to do is pass printRecord(ep) the employee pointer and it will print that record in formatted style.

I worked on strings today. Now the system has addRecord with the name field working too. I had to create newData() & newNode() so I could abstract them from the addRecord() routine. I needed to create new nodes at new memory addresses. Previously I had been adding them in an ad hoc manner which broke when I tried adding multiple records including the new name field. Segment faults were the order of the day. Luckily I can use the gdb debugger and see where I'm walking off of a cliff. Factoring out newData() and newNode() helped separate the data from the structure. Then all the pointers worked again. No pointing off to Erehwon. I need to propagate those new routines throughout the code; even for addRandomNode(). Orthogonal code is always best and is the easiest to debug. It, like formatting, help find bugs and keep them from occurring in the first place. Order, style, and elegance is the best way to write code.

Once I reviewed the string function strcmp(str1, str2) I added names to the sorting routine. Only forward at this time, but adding reverse is simple. I should make a record of how long some of these routines take to implement. Once the structure of the menu/message loop pattern is created the modules can almost write themselves. Most are less than an hour of coding, some only minutes. Rearranging modules is just the time it takes to think about it then cut and pasting. That can take 20 minutes to an hour but a lot can change in that time.

However, I find taking breaks and working on other things lets my mind work on this problem in the background. I wake up with ideas ready for my note pad each morning. Those ideas lead to refinements or new modules and new directions. After building the code once you can see how you can rewrite it better and faster. All the modules you created the first time around will get reused; most of them intact.

I think I may be done here. Edit still doesn't work but that can be an exercise for the reader. All the parts are in place in other modules just copy and paste them with a few edits. Reports should be

expanded greatly. But those changes are strongly effected by the fields you are going to use for your solution. I will add a data field now with the tools necessary for it and call it good. Edit this narrative, clean up the code and comments, and call this the capstone lesson for section one – tools.

Oh yes, clean up the menus too. The secondary ones in particular. There also may be some factoring to be done in the whole menu system too.

Started work on <time.h> functions. Have time and date printing beautifully. Now I'm trying sleep() and difftime(). I'll test some of the others tomorrow but I think I have enough for most database things.

{.../1 tools/S1LFP_database/snapshots/database December 13.c}

December 14 – implemented find menu selection 2 use name as key. Once I got the order of comparison correct, the function worked like a charm. Now I can find all six employees named Maria in the present list.

I must get back to thinking about date or time fields and the tools necessary to manipulate them. The seniority field has always been an afterthought to me. How about we change it to date of hire? Then we can winnow out those who have more seniority than years on this earth. Randomly generated input is just that if you don't put constraints on it. The salary function had a lowest value as did age, the randomness was added on top of that base.

Make a list of what you would want a database to do with times and dates. Time and date of hire? Birthday? Vacation days. Time cards. ??? It will be your tool, make it useful for you. If you use your own code in your daily routine you will find yourself polishing the code to make it run better. It is called "eating your own dog food".

As of right now this database system is a fine working tool. If I have a need for it I will change the fields to fit my needs and use it. Remember when you use it you are REQUIRED to have the id field. Everything else can change. But you do need that unique id for some of the modules to perform correctly. Just a few bytes of memory in a computer or on a disk but it is very much worth the space.

As I use the tools I find I like the keypad command of everything. Very fast and convenient. Graphic user interfaces are nice but we forget how easy it is to use command-line interface (CLI) instead. Plus your apps are smaller, faster, and easier to read. If you need to impress folks with a GUI you can always wrap it around an already capable CLI application. Once you have worked your way through section two of these lessons (graphics) you'll be able to do just that.

Now that you have reached the end of the first section of lessons it is time to make a decision. Do I want to tackle much harder algorithms and data structures by moving on to section three? Or do I want to learn the basics of graphics systems by starting section two? It is your choice. I think section three is a little more difficult than section two but parts of section four depend on the advanced tools which are in section three. You can work them both together if you like. But I certainly recommend you have worked through sections one through three before you jump to section four – advanced graphics. It depends on all the sections, and most of the lessons, to understand.

Don’t worry, you can always jump back to any lesson in any section. They are not locked. It is just a matter of how they fit together for you. Some people learn better one way some people the other. I am trying to write these lessons with a number of paths in mind. Plus the appendices offer insight into how some things are done and why others are the best way in certain circumstances.

Section one gave you many fine tools to accomplish some serious work. Just look at the database program written from those lessons. It was a simple matter of putting all the parts together and thinking about how you want to do your work. When you write the code you get to determine how you can use it. It really helps to have tools you have built, so you can use and refine them each day. If you use your tools regularly you understand them better, and can refine them even further.

It is up to you whether to go to section two or section three next. But I am very sure I will see you again when we all meet up in section four – advanced graphics. It is cool. It is fun. It is very colorful. Plus you can mesmerize your friends (or cats). Lots of math, lots of physics, lots of art. See you there.

{.../1 tools/S1LFP_database/snapshots/database December 14.c}

time & gdb

http://www.cplusplus.com/reference/ctime/mktime/

http://www.cplusplus.com/reference/ctime/strftime/

www.cplusplus.com/reference/ctime/difftime/

https://www.thegeekdiary.com/c-library/

https://www.tutorialspoint.com/c_standard_library/time_h.htm

https://www.sitesbay.com/cprogramming/c-read-and-write-character

https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf

union vs struct

https://stackoverflow.com/questions/346536/difference-between-a-structure-and-a-union#346541

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

https://www.cprogramming.com/tutorial/c/lesson9.html

Relational database

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

https://computer.howstuffworks.com/question599.htm

https://www.codecademy.com/articles/what-is-rdbms-sql

From K & R page 33:

Definition refers to the place where the variable is created or assigned storage

Declaration refers to places where the nature of the variable is stated but no storage is allocated.

http://www.cplusplus.com/reference/ctime/strftime/