Section 1 Lesson 8

Section 1 lesson 8 strings

I Discuss new data structures, algorithms, or language features

There are no function definitions or new structures in this lesson. Just various ways of examining and manipulating strings by using standard string functions from <string.h>. I am finally going to tackle pointers and strings. Most texts introduce pointers by using them on strings of characters. I chose to use pointers to access large chunks of data first. I wanted to show how memory structures use memory which you can directly access using memory addresses. Now that you know how to do that, you can learn how to manipulate the smallest bits of memory.

II Describe functions and modules

In main() you declare a file pointer and a number of working character pointers. Open the file 'The Raven.txt' for reading and set the file pointer fp at its beginning. Find the end of the file using the C function fseek(). Determine the file's size by finding where fp is pointing at the end of the last step. Set the file pointer at the beginning again, by using the function rewind(). Use the size of the file to allocate the right amount of space in heap memory. Then read the entire text file into that memory space one character at a time. Close the file with fclose(fp). Repeat this series of steps with files: 'The Memoirs of Sherlock Holmes.txt', 'Huck Finn.txt', and 'Tom Sawyer.txt'.

Now that we have all four files read into memory, we can test how various string functions are used to manipulate data. First we allocate space in memory, then we fill it with C's string terminator '\0'. By doing this, no matter how many characters are added into the space, there will be no trailing characters represented in the output. We use the string function strchr() to look for the first 'x' found in 'The Memoirs of Sherlock Holmes.txt'. strncpy() copies from the file in memory starting at where you found the first 'x', then puts it and the next 19 characters, into your allocated space – dest. Then we print what's in dest and free up the memory with puts(dest) and free(dest). We repeat these steps, but look for the final 'x' instead of the first 'x', with ret = strrchr(holmes, 'x'). We then print everything in the file 'holmes' from the last 'x' found until the end of the file with puts(ret).

Next, we create an array of pointers to characters and fill the array with 10 breeds of dogs. Then we use the function strcmp() to determine if 'Airedale' is lexicographically (alphabetically) less than 'Dalmatian'. Remember to start counting with 0 not 1 (you are not crazy). After that, we print out some of the breeds with puts(dogs[9]), etc. Then we print them all from an indexed loop. I generated an interesting error while writing this showing that stacks are not heaps. I wrote the line free(dogs) which generated the error. You can only free memory you have allocated from the heap not space

compiled from stack memory. Uncomment the line free(dogs); to read the error message. The moral is you should leash your dogs.

Now we're going to find out where 'Huck' was first mentioned in the book 'Tom Sawyer.txt'. We ask the user for a word to find in a file. We allocate 35 characters worth of memory space and set the pointer, word, to its start. In this case we are again searching in 'Tom Sawyer.txt'. If we input 'Huck' we could see how many times he is mentioned in the book Tom Sawyer. Or enter any word you like and see how often Mark Twain used it in his story.

Now that we have our search routine working, we can modify it to find the frequency of characters in a given file. In this case, we are searching through the string 'The Memoirs of Sherlock Holmes.txt' for the character input by the user, and calculating its frequency. This routine could easily be modified to determine the frequency of all characters in the file with output as a histogram. Choose text files from different periods in history and see how letter frequency has changed over time. To exit this loop type '!'.

Free up all of our allocated memory and exit the program with a return 0 to show all is well with this run of string.exe

III Compile and run the code

- Open a terminal window
- Navigate to … S1L8_string using cd (change directory in Linux or Windows using MinGW)
- Type **make** which will link and compile the application called string (in Linux, in Windows it is called string.exe)
- Type **./string** (in Linux) or **string** (in Windows) to run the application
- Read string.c in your editor to find how the code creates the application's output

IV Describe code line by line

Starting with line 12 we have our usual #includes plus a new one for strings. Notice, we did not randomize the number generator so we did not need time.h. There is no need for random numbers in this source file.

On line 18 we create a file pointer. Same as any other pointer except it points to a place not in memory but on your disk, or in stream, or on a USB drive, or etc. We use the file pointer fp on line 24 when we

open our first text file for reading ("r"). Then, because we don't know how long it is, we measure it with fseek(to the end of the file) and ftell(fp). Once we know that datum we store it in size, and rewind the file pointer back to the beginning of the file. Next, we allocate the memory space necessary for our new text file, and load the file into that memory space.

Line 32 is an idiom you will need to decipher. I could have written this any number of ways with added variables. By doing it this way I don't need any external variables. In fact this is how the compiler thinks. If you write the code any other way the compiler needs to massage it into this form. So, bit by bit, I'll explain it to you.

The first thing line 32 does is get a character from your file with fgetc(fp). The character returned by that statement is written into raven[i]. This character is now compared to EOF. If it is the end of the file you're done. Otherwise raven[i++] increments to the next slot and gets the next character with fgetc(fp). Notice the ++ is placed after the i. This means the increment takes place after the entire line of code has executed. The open set of braces { } is needed, semantically, so the compiler does not scream. Nothing really happens within the braces they are there to look pretty, nothing more. Remember that idiom for later, it is useful, saves variables, and is faster because the compiler "likes" it.

Line 34 prints the entire text file, now in memory, onto the screen. puts(raven) is the put string statement to match the putc() statement for characters alone. It works well for simple printing statements. Only rudimentary formatting is available with puts() so keep your statements simple. Next we close the open disk file with an fclose();

We continue onward, opening three more text files. The amount of memory you have determines the size of your text files. If you ask for too much you'll start using virtual memory, which thrashes the main drive. You notice because your response time drops precipitously. Memory is cheap, buy more if you need it.

Now we come to the string manipulation part of this code. Choose either line 80 or line 81 to define dest. It really does not matter which you choose they work equivalently. As ever, you need to allocate any space you want to use. Line 82 grabs the memory for 25 chars. I probably could have written

```
dest = (char *) malloc(25);
```

and gotten away with it but I added the sizeof(char) casting to tell the compiler the size requirement explicitly.

Line 84 scans the string holmes for the letter x. ret is assigned a pointer to the FIRST time x occurs in the file. Now we copy 20 characters from holmes, starting at ret, then write them into dest. Then we print some whitespace, and the string we just built. Next, we free up the memory for dest with free(dest), to be polite.

On line 92 we scan holmes for 'x' again only this time we find it from the end instead of the beginning. Notice the function strrchr() is used not strchr(). In this case we print from our newly found 'x' to the end of the file.

On line 101 we create an array of dog breeds. Notice the const char instead of just char. const means it is a constant or fixed string. The compiler won't let you change dogs[] anywhere in the file. I am pretty sure const is not strict C, but from C++. I only use it when I really mean it. Normally just using char *dogs[] = {...} is enough to tell me that dogs is fixed in time and space. I won't bother futzing with it again. It is just the idiom for getting a number of strings ready for later, and available via pointers. Trying to add the strings later just opens up a large can of worms. If you don't really need to do it just don't, and keep your life simpler.

On line 105 we perform a string comparison. The results are in lexicographical order (alphabetic). rt is an integer showing < = or > for a result (-1, 0, 1 are the output choices). I'm simply comparing ‘Airedale’ with ‘Dalmatian’.

The commented out line (123) is left in, even though it is an error. If you uncomment it and try to compile the file you'll get an interesting error. It is instructive to read it. What it means is dogs[] can't be freed because it never was located in heap memory. Since you allocated it at compile time you are using stack memory. And as we all know - stacks are not heap. But, a well trained dog should be allowed to walk freely.

Lines 111 through 117 use puts() to display a few of the breeds.

Lines 119 through 121 show them all.

Now an interesting function – strstr(), which finds a string within a string. In this case, we're looking for 'Huck' inside of the Tom Sawyer string. I loaded large files because it gets more interesting. Small files are too predictable and boring. Boredom is the little death.

Lines 142 through 160 use the string function strstr() to find all the occurrences of the user's chosen word. Line 149 chooses Tom Sawyer as our string. The display gives all the locations in memory where your chosen word begins. Line 150 is of special interest. It shows the compressed idiom

```
while ((ret = strstr(ptr,word)) != NULL) {...}
```

Piece by piece we get the result of looking in string ptr for word. If that result is NULL then we drop out of the loop. Otherwise the result ret is used in the loop. Also notice line 154. At first I got an endless loop when that line read: ptr = ret; I was getting the endless loop because I was finding the very word I had found in the previous iteration of the loop. A moment's thought told me to move past this spot with ptr = ret + sizeof(word); which moves the pointer to just after the word I found.

Now put the string onto the display. In this case I could not use puts() to display the output because the formatting clause was too complex "Found %s %d times\n" So I used the handy printf() function instead.

Lines 169 through 191 are in a bit of flux. There is a strange bug I need to find. scanf() is returning something I don't understand to give me extraneous lines of output. But the answer to a given queried character is correct. I will fix this soon :) Anyhow, the user is prompted for a character for use in scanning the chosen file and counting for character frequency. I was planning on scanning the file for

all upper and lower case letters, to find the frequency of each of them. For now, it finds the frequency of one character at a time. To kill the function you need to type '!' when asked for a character. (I found the bug. Look at line 176 in scanf(). I had to add the preceding space in the format string " %c". Now all is well.)

On line 183, I am accessing the text file holmes using the array notation in a for loop. The array allows me to use an iterative index into the string. Pointer arithmetic would work as well but this is more visual; telling the programmer what is going on a little more clearly.

This frequency code should not be a while loop which gives you an extra line on exit. It should be written as a do loop instead. I use do loops so rarely I need to think of reasons when they are necessary. This is one of them.

I commented out a loop which uses large numbers. Larger than int anyway. In this case it is a long which hold the numbers between 900000 and 1000001. Just a note to myself in any case.

Then, we free all the memory we have used. Not really necessary in this case since, we're exiting the program, but it does show how it is done. It is not necessary, because upon the app closing all the memory used by the app will be reallocated by the OS. It's a matter of do it yourself, or let the OS do it for you. However, it is easy to create memory leaks. When you have arrays of structs with pointers to chunks of memory it is safer to free each part of each structure with a function. Then, at the end, you call that function to make sure memory is freed correctly and thoroughly. While the app is running you may need to free up space, this is how it is done. Freeing the space does not wipe out the data in those memory locations. But it does notify the OS that this memory is ready for reuse. In fact, if you reallocate this memory, the data you put into it previously will still be there. You just overwrite it as you need to. If, instead of using malloc(), you use calloc() your newly allocated memory will be filled with whichever character you tell it. In other words, malloc() does not clear out memory upon allocation but calloc() does. In secure situations this little fiddly bit may prove important.

Line 204 is technically necessary but rarely enforced by the compiler. Because you declare int main(void) you're supposed to return something when it is done. A return value of zero signifies a proper exit. You could add error codes, and then check them upon exit, but we're not quite at that stage yet. Exit codes on functions are far more useful to me than error codes on application exit. The OS can worry about those :)

V Describe makefile line by line

Line 3 sets our compiler macro to g++

Lines 5 & 6 link the executable string from the object file

Lines 8 & 9 compile the object file from the source code

Lines 11 – 13 clean up the files under Windows

Type **make clean** to remove both the object file and the executable

Under Linux line 13 should read: string with no trailing .exe

VI Deeper

I wanted to make sure you were comfortable with pointers and memory locations before I introduced strings. Instead of using pointers to point at structures or objects in memory you will now look at individual memory locations. Characters are the smallest of the data types, each memory location can hold one character. Now that pointers are second hand to you, using them to search through a large text file will be simple. We will use four text files I found on the Gutenberg site. They publish literature as it falls out of copyright. If you like reading prose from the early 20th century, and before, you will probably find what you are looking for. I chose: The Raven, The Memoirs of Sherlock Holmes, Huck Finn, and Tom Sawyer as my source strings. Yes, we read each file in as one string then play with them.

Strings in C are kind of strange. But like everything else in C, if you know where the data is located in memory you will understand how C works. For example:

```
char myString[] = "My Akita likes to sleep";
```

and

```
char *myString = "My Doberman likes cats";
```

are both strings instantiated at compile time. While

```
char *myString;
```

is a pointer to one single memory location. You need to allocate more memory space if you want to add anything to it. With the previous two declarations you have preallocated the space by including

your data at compile time. Remember, you cannot change these strings after this point. They are constant. Great for titles or headings to your output. Because

```
myString[]   and   *myString
```

are interchangeable you can use either method to access any given character like so:

```
char temp;

temp = myString[5];
```

or

```
temp = myString + 5;
```

In either case myString points at the first memory location of your string. myString[0] is equivalent to myString because the name of an array is the address of the first slot in the array. That is also why C programmers count: 0, 1, 2 ... because they're addressing first the base slot and then the next slots in the array (it is not because we are crazy - at least as far as I know :). The next slot is the base address plus your offset, which in this case is sizeof(char), which is the smallest unit possible. You are simply pointing to an individual character with two different notations. If you dig into the compiler, you will find out which one is used internally. That is the one which will get written into assembly code for the linker. So think of it like you're multiplying A * B or A x B or AB; they each mean the very same thing, they just look different.

C does let you play fast and loose with integer to char conversions. You really should use casting to let the compiler know what you are doing. The warning messages will be far fewer if you do so. If you use Unicode characters you need to be sure to use casting, because you will either get odd results, or the compiler will throw errors at you.

OK, fixed the little bug I was having, and changed to a do loop instead of a while loop. Still a work in progress because I really wanted to get the frequency of all the characters instead of just one at a time. Automation is built into computers, why not use it? I fixed the bug by adding a single space to scanf(" %c", &ch); That single leading space tells scanf() to ignore leading whitespace. Also the &ch is necessary because ch does not designate an address and scanf() needs an address instead of a variable. All of the string functions require pointers to addresses. Now our final string function works the way I had intended it to work.

If you want ,you can change these routines into functions, which will make this file much prettier. Right now I am going to move on to the next section of this course: windowing apps. So you're about 1/4 done. By the way, if you want to see the frequency of all the letters, just type them all in on one scan line. Because the algorithm feeds on only one character at a time it will just keep reading them from the input buffer until it gets done. One by one by one you get all the frequencies of all the letters you entered. This is one of the beauties of Linux, you can chain application calls easily.

VII Assignments

1. Change the search character on line 84 from ‘x’ to some other character.
2. Change the search character on line 92 to another character.
3. Use another text file you’ve found instead of the ones I’ve included.
4. Add a few more breeds of dogs.
5. Change the array indices on lines 105, 107, & 109 to compare other breeds.
6. Change the array indices on lines 112 through 116 to print other breeds.
7. Install a loop around lines 146 to 159. Add an instruction to use ‘-1’ to break the loop.
8. Install a loop structure around lines 172 through 191 with a break instruction.
9. Save the information from the previous assignment and print out frequency of all characters.

VIII Links

- https://www.gutenberg.org
- https://www.cprogramming.com/tutorial/c/lesson9.html
- https://www.tutorialspoint.com/cprogramming/c_strings.htm
- https://www.programiz.com/c-programming/c-strings
- http://www.cplusplus.com/reference/cstring/strchr/
- https://www.tutorialspoint.com/c_standard_library/c_function_strchr.htm
- http://www.cplusplus.com/reference/cstring/strrchr/?kw=strrchr
- http://www.cplusplus.com/reference/cstring/strcmp/
- http://www.cplusplus.com/reference/cstring/strncpy/?kw=strncpy
- http://www.cplusplus.com/reference/cstring/strstr/?kw=strstr
- http://www.cplusplus.com/reference/cstdio/scanf/?kw=scanf
- http://www.cplusplus.com/reference/cstdio/fseek/?kw=fseek
- http://www.cplusplus.com/reference/cstdio/rewind/?kw=rewind