ELEC 278 – Data Structures and Algorithms

Lab 1: Getting Started with Data Structures and Algorithms

Start: September 6, 2022 Due: Day 7 of Lab Day

Contents

1. Basic Lab Information

You should review the Lab Overview document prior to reading the material for this lab exercise.

If you have already installed a C compiler on your (lab) computer (or have one available), then you can continue with the lab. If you have not installed the C programming tools, please stop working on this lab, and review Lab Overview document Appendix A. You may also wish to review Tutorial 1 before you attempt this lab.

You should also review Appendix B of the Lab Overview document. This contains important information about how to submit your assignment.

2. Lab Resources

Specific to this lab:

1. Lab Instruction sheet (this document)
2. A zip file containing source code (in a folder called Lab01Src). Check the file "PACKINGLIST.txt" for a list of files that should be included.

Learner provided:

1. Programming environment selected by the learner. The Lab Overview document provides a list of possible C programming tools that you may use.

This lab has a lot of code for you to read. The purpose of the example code is to reinforce what you have seen in the class material. It also provides you with code to cut-and-paste into the code you write to solve the assigned problems. This lab has multiple programming problems to solve.

3. Lab Objective

The goal of this lab is to explore some of the basic C programming concepts that will be useful for all the other labs in this course.

You will note that some of the lab sections ask you to read programs, with the goal of understanding how the code works. These programs illustrate some of the basic ideas discussed in the lecture part of the course. The programs in these sections are heavily commented – the intent is that you read and understand the code in the same way that you would read and understand solved problems in other courses. Besides reading the source code, you should compile and run the programs. I would suggest that you try to have the source code open at the same time as you run your code – so that you can correlate the code with the program's behavior.

3.1 Topics covered

- Arrays, pointers, using pointer as if it were an array name.
- Dynamic memory allocation – malloc() and free().
- Reading files – whole lines as strings, individual numbers, using scanf() to read data, flushing input after using scanf()

- Using command-line arguments.
- Structures and linked lists.

4. Lab Steps

There are many steps in this first lab. Some of you may find it easier if you tackle all steps before the lab and ask the questions you have in the lab. You can nibble on pieces over the two-week period – or have a large feast the night before the lab is due. Figure out which technique works for you.

4.1 Part 1 - Programs Reading Text Files

There are three programs – **readtextv1.c**, **readtextv2.c** and **readtextv3.c** – provided. These are intended to illustrate several C programming concepts. There are also several text files – **textfile.txt**, **textfile1.txt**, **textfile2.txt**, etc. (There is also a copy of the program – **walkptrn.c** - that was used to create **textfile4.txt** – provided for those who are curious.)

Readtextv1.c is a simple program that shows how to read a file. It **opens** a file whose name is coded into the program, and then **reads** the file, line by line, printing each line as it goes. Review the code and make sure you understand what it is doing. You could create your own plain text file and modify the program to read your file.

Aside: Warning about file names

Pay attention to the notice at the top of **readtextv1.c**, explaining about the how to deal with different ways that compilers and IDEs store and run programs for you. The following diagrams may help to explain the issue.

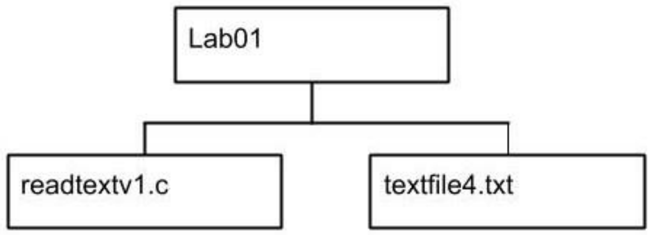

Figure 1. Source file and text file in same folder (directory).

This is where you start, with your text file and your source file in the same folder.

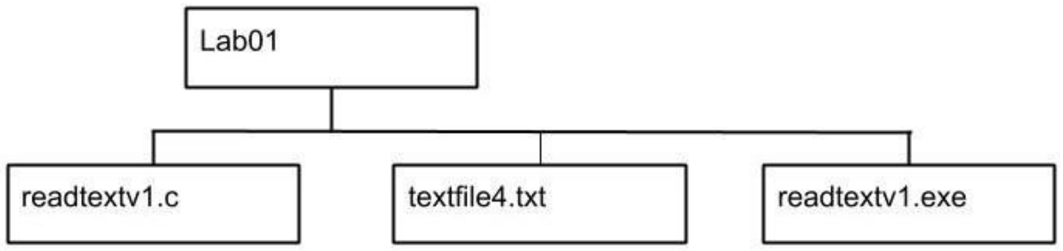

Figure 2. Compiler creates executable in same folder

A simple programming environment (such as clang or Cygwin) creates the executable in the same folder as the source file. When the executable runs, it can find the text file in the current working directory.

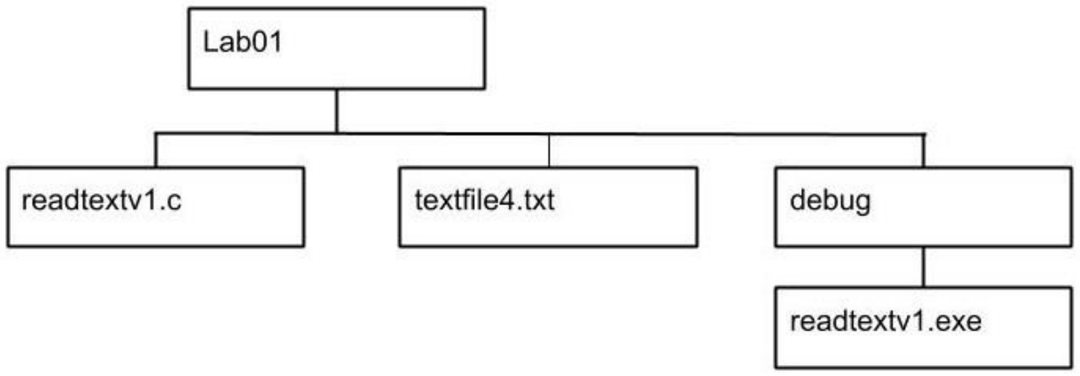

Figure 3. Compiler creates executable in a sub-folder.

Some other programming environments (Microsoft Visual Studio or Qt, for example) create executables in a subfolder. (This way, they can create both a **Debug** version, with lots of extra features to help with debugging and a **Release** or **Production** version meant to be shipped to users.) When the executable is run in programming environments like these, the text file is one level up in the directory tree.

Suggestion:

Modify readtextv1.c so that it asks the user to type in the name of the file to be printed, reads the user's response, and then prints the file that the user named. **DO NOT HAND THIS CODE IN.**

Readtextv2.c is a variation on the first version of the code. This version lets the user specify 1 or more text file names on the command line, and it prints either a warning that the file could not be found, or it prints the contents of the file.

That is, if the program is run from a command line environment (such as cmd.exe in Windows), it could be run as:

```
        readtextv2   textfile1.txt   textfile2.txt
```

and it would print out the contents of **textfile1.txt** then **textfile2.txt**

A substantial portion of the program is identical to **readtextv1.c**. What is new is the code to deal with a command line. Read through the code, and make sure you understand:

1. What a string is in C.
2. What a pointer to a string is and how can it be used.
3. What an array of pointers to strings is useful for.

Readtextv3.c is meant to be more readable than readtextv2.c. Why? Because the code has been grouped into functions that only do one thing. So, there is a function that prints out the contents of a file. There is another function that opens a file, calls the print function to display it, and then closes the file. In this example, it may be hard to see the benefit, but later on, as programs become bigger, it will be an advantage to be able to package up some functionality and then perhaps use the functionality in other programs – by cutting and pasting code (or by building function libraries).

Make sure you understand how **readtextv3.c** works – in particular, make sure you understand how data is passed between the functions.

4.2 Part 2 – Programs Reading Numbers

Programs provided for this part are **readnumsv1.c** through **readnumsv5.c.**

Readnumsv1.c has a fixed size array that can hold a maximum of 50 integers. The program asks the user how many numbers will be entered, and if the number is less than or equal to 50, it then asks for the numbers. Each number is stored after the previous one in the array. Once all numbers are entered, the program prints them all out. What you need to see from this program is how to read a number using scanf(). What you want to check is that the input can be formatted any way you wish – multiple numbers on one line, each number on its own line, 1 space between, many spaces between – and the program works.

Readnumsv2.c deals with the limitation of the predetermined fixed size array. It asks the size of the data – how many numbers – and then fetches a piece of memory big enough to hold that many numbers. So, the program is only limited by how much memory malloc() can hand out.[1]

You will use malloc() in just about every program you write, so make sure you understand how to use it. You should also make sure you understand what free() does.

There is a program called **makenums.c** provided. What it does is produce output in a form that **readnumsv2.c** is expecting. If you compile the source file to make **makenums.exe** and run it as follows,

```
makenums 240
```

then you will see output that looks like what is in the file **nums.txt**. In fact, **nums.txt** was created by running the **makenums** command as above, but redirecting the output to a file, as follows:

[1] On any computer you are likely to use for this course, malloc() can fetch about 1G or more.

```
makenums  240  > nums.txt
```

This makes it easy to test readnumsv2 – you can run it with its input coming from **nums.txt**:

```
readnumsv2  <  nums.txt
```

That is much easier than typing in 240 numbers.

Readnumsv3.c handles the case where you do not know how many numbers there might be, but you must handle any possibility. Here we use a linked list of structures. (For a general discussion of structures and linked lists, please refer to the class notes and references.) The structure used looks like:

```
struct node   {
        struct node    *pnextnode;
        int            datavalue;
        };
```

For every number read in, a new node structure is created with malloc() and then the new structure is linked to the structures in the existing list. This first version of the linked list code just places each new number at the head of the list – with its next pointer pointing to the list as existed before the new number was added.

Readnumsv4.c is like **readnumsv3.c** except that it adds each new node to the end of the list, not the beginning. Note the differences between the insert routines in version 3 and version 4. Can you explain why they must be different?

Finally, **readnumsv5.c** allows the user to specify, for each number, whether it should be placed at the front or at the end of the list. Read it carefully and note the technique of implementing functionality (such as the creation and initialization of a new node) only once and using it in the two insert routines. Also note that a data file has been supplied – **numdataforv5.txt** – you can use to see the program work. You would use it after you have compiled readnumsv5.c as follows:

```
readnumsv5  <  numdataforv5.txt
```

4.3 Part 3 - Deliverable

The code that you are to hand in is outlined here.

You are to start with the code in **readnumsv5.c**. Add a delete command. The idea is that, in addition to entering F and a number or B and a number, the user could enter the letter D followed by a number. If the number is present in one of the nodes in the linked list, that node will be deleted, and the links repaired so that the linked list still works. As you think about this, make sure you handle the cases where the node to be deleted is the first one in the list, the node is the last one in the list, and the case where the node is somewhere between the first and the last. As an example, if you ran the program with the following input:

```
F  10
F   9
F   8
```

```
D  9
Q  0
```
you would see 8 -> 10 displayed.

<u>Detailed specifications:</u>

- The source file name shall be readnumsv6.c.
- The input shall have the same format as readnumsv5.c, that is, lines containing a letter and a number.
- The output shall match the output that readnumsv5.c produces, including the lines starting with “Received”.
- The first line of your program shall conform to the course standard for headlines, that is, it should be
  // readnumsv6.c – Lab 01 – FirstName, LastName

Hint:
Dealing with Duplicates
The addition of items to the list is done by specifying insert at the front or insert at the back. There is no requirement that the numbers be unique in the linked list. Nothing in the specification says to check for duplicates, and it is fair to assume that duplicates are not prohibited.
That said, how do you handle deleted in a list that may have duplicates? Again, the specification does not say delete all the entries with the specified number. (It does not say delete only the first one, either.) So, you may assume that deleting the first one found when traversing the list is good enough.