CS 2110 Homework 7
Intro to C

Farzam, Camille, Sid, Matthew So, Josh, Sean

Spring 2020

Contents

1 Overview

1.1 Purpose

The purpose of this assignment is to introduce you to basic C programming and the Linux command line, building on your knowledge of Assembly. This assignment will familiarize you with C syntax and how to compile, run, and debug C programs. You will become familiar with how to work with strings, arrays, pointers, and structs in C, as well as with functions and command-line arguments from the Linux shell. You will understand the relationship in C between arrays and pointers, including pointer arithmetic. (Think about how arrays are stored in memory in Assembly Language.) You will also become familiar with the Linux command line, including how to use a makefile to compile your program and how to redirect standard output to a file instead of to your screen.

1.2 Task

You will write a C program that accepts command-line arguments that represent basic arithmetic expressions (i.e. "3+4-0xA"), parses these strings, computes the result of each expression, and stores the argument strings, resulting integers, and error codes into a struct. Each struct (for each command line argument) will be located in an array of structs. The string expressions from the command line will represent integers (either decimal or hex) and operations (addition or subtraction).

You will write C code in two C files, my_string.c and hw7.c. In my_string.c, you will write your own implementations of the C library functions strlen(), strncmp(), and strncpy(). In hw7.c, you will implement the functionality to parse the command line strings, populate the array of structs, and print the debugging output. Please see the Detailed Instructions below.

Usually placing those definitions in .h files would be good practice, but for this assignment you are not turning them in, and so those declarations would be lost when submitting.

At the end of this assignment, you will "pipe" the standard output of your program (using the -d flag as an argument) to a file named 'out.txt'. For instructions on this, see the Piping Output section.

Take a look at the sections on Makefiles and Testing and Example Output for more info on how to compile and test your program.

You may find it helpful to draw diagrams of memory locations, as you did with Assembly programming. How are arrays represented in memory? How can you use pointers to find the address of array[i]? How are arguments passed to a function and results returned using the stack frame? Remember, C functions are pass by value (push copies of the arguments on the stack), just like subroutines in Assembly.

1.3 Criteria

Your C code must compile without errors or warnings, using the provided makefile. Your array of structs should be populated correctly at the end of the program, based on test cases (including command line arguments that contain errors). Your helper functions in my_string.c must all be implemented correctly (producing the same behavior for test cases as the equivalent library functions from string.h). Your text output in out.txt should match the expected text output.

2 Detailed Instructions

2.1 Command-Line Arguments

When you write a C program, you can work with arguments you receive from the terminal through two parameters you receive in the main function, argc and argv.

argc: The number of command line arguments you receive.
argv: An array of your command line arguments as null-terminated string literals.

Note: the zeroth argument to your program is always going to be the name of the program itself.

In this homework, all mathematical expressions are passed in as command line arguments through a wrapper function called "my_main" (see main.c) surrounded by quotation marks. In addition to these arguments, there is a debug flag (i.e. "-d") that can only be passed in as the first argument to ./hw7 (along with expressions) which should call the provided printArrayDebug() method in order to print the array information.

You need to finish the implementation of the my_main method. We recommend at least setting up the necessary validation before moving on. Keep in mind the return value of main depending on the **status of termination, which is different from error codes explained below.**

2.2 "my_string.c" functions

A major part of this homework is to implement 3 of the very popular C string libraries: my_strlen, my_strncmp, and my_strncpy, **using only pointer notation**. We highly recommend implementing these methods first so you are able to use these functions as you move on with this assignment.

In order to understand the functionalities of these three libraries, you need to take a look at their man page (i.e. manual page). See the section on Man Pages for more info.

- Note 1: **You are NOT allowed to use array notation in this file. All methods should be implemented using pointers only!** Think about how arrays and pointers correlate with each other in C.
- Note 2: You are **NOT** allowed to use any of the standard C string libraries (e.g. <string.h>).
- Note 3: Although expressions passed in to this program have a min/max length, the string functions should not have a boundary on any arguments passed in.
- Side-Note: For my_strncmp you do not need to return a specific number (as long as it follows the description in the man page).

2.3 Type Conversions and String Parsing

Similar to HW1, you will need to handle the conversion from both a decimalString and hexString to integer. For hexadecimal values (preceded by "0x"), you should check the validity of the hex value **parseExpression method** before continuing your calculations. Examples of invalid hex representations include: 0xx3A, xF, and 0xFH. (Leading zeros are acceptable).

- Note 1: Similar to HW1, conversion methods will only be tested with valid inputs, but practice safe C programming early on.
- Note 2: We will only test your code with upper case hex letters, however other invalid hex values should set the corresponding error message for the respective array index.

For parsing the string expressions passed in via command line, there are certain error cases you must handle, such as an invalid hexadecimal value.

You are NOT expected to handle the following:

- Input that contains any non-integer characters (other than those necessary for hex representation, like "." or "A"). This means you may assume that decimal integers in the expression are correctly formed, though as specified above you must check the formatting of hexadecimal integers.
- Input that has consecutive operators (i.e. "4+-3")
- Input that has trailing operators (i.e. "4+-3+" or even "+4-3")

2.4 Populating The Array

After a string expression is parsed, the results should be stored in a struct in the array of structs. The array of structs is declared in hw7.c under the name *arr*. Remember, you can access a structs field using the 'dot' operator as shown:

```
struct_name.field
```

Look in the hw7.h file provided to see the 'expression' struct declaration, and learn the correct field names and types for this struct.

Depending on the expression string argument, the buffer field (think what function might be useful for this task) will either be filled with the correct expression string or an error message, defined as the ERROR_MSG macro (see errcodes.h). The default value for this field is empty.

The result value will depend on the expression's result. If the expression string is empty or invalid, the default value for this result is zero.

The error code will also depend on the parsing of the expression string. See the Error Codes section below for their usage. If the expression string is empty, the value of this field is set to zero.

The array should be populated in the same order as argv.

Note: We will only test your expression parser with simple addition/subtraction operations.

2.5 Error Codes

All error codes are predefined in errcodes.h.

If an inputted operation string is longer than the maximum expression length (provided in a macro), or if one of the arguments is incorrectly formatted (i.e. invalid hex number), the corresponding struct in the array should be populated with the corresponding error code, the result should be set to zero, and the expression string set to error message string (specified in the ERROR_MSG macro).

Note:

2.6 Piping Output

Note: This should only be done AFTER YOU HAVE COMPLETED THE ASSIGNMENT.

Once you have implemented your parser and functionality for the "-d" flag, you are able to print a visual representation of your array to the console. As the final part of your assignment, we ask that you pipe the output of this program, along with a special set of arguments, as run with the "-d" flag, to a text file as part of your submission.

To generate your unique expression, run the provided generator, replacing ⟨gburdell3⟩ with your GT username.

```
$ ./gen <gburdell3>
```

This should generate a file named **exp.txt**. This file contains a unique expression that we want you to pipe as arguments to your hw7 program!

How the heck do we do that?

We'll start by learning a few basic Linux commands necessary for this task:

The 'cat' command concatenates the contents of the file to a destination of your choice. The default destination is the command line.

```
$ cat filename.extension
```

Try running cat on the **exp.txt** file generated in the previous step.

The redirect command, '>' or '>>' redirects the output of some other command to a destination of your choice. The '>' command will over-write the destination file, and the '>>' will concatenate to the end of said file. If the file does not exist, this command will generate it.

```
$ cmd > filename.extension
$ cmd >> filename.extension
```

The command $() is the substitution command; it substitutes this argument by the output of the command between the parentheses.

```
$ cmd1 $(cmd2)
```

In the above example, 'cmd1' uses the result of 'cmd2' as an argument.

Now to connect everything, you may use the command below to pipe your program's output to gtUsername_out.txt, or come up with your own. Be creative! There are many ways to do this.

```
$ ./hw7 -d $(cat exp.txt) > gtUsername_out.txt
```

The above is simply taking the standard-out of your program (what you print to the command line, or stdout in C) and writing it to a file instead of your terminal window. The command line arguments passed in are the contents of the exp.txt generated file.

Linux Rocks !!!

More Basic Linux Commands

2.7 Makefiles and Testing

Makefile is a common utility for executing a set of directives. In all of our C assignments (and also in production level C codes), a Makefile is used to compile C programs with a given set of flags. We already provided you a Makefile for this homework and although you are not being tested on it, we highly recommend that you take a look at this file and understand the GCC commands and flags used to understand how to compile C programs. If you are interested, you can also find more information regarding Makefiles here.

To test your code manually, compile your code using make and run the resulting object file with command-line arguments of your choice.

Keep in mind that you should run all commands inside the Docker terminal. We highly recommend running the usual script as follows to immediately get a terminal:

```
./cs2110docker.sh -it
```

If you use your own Linux distribution/VM, make sure you have the check unit test framework installed, however, keep in mind that your code will be tested on Docker.

Below is an example of manually testing your code.

```
make clean
make hw7
./hw7 -d "3+0x40-7+5" "0xA6+5000-45‘‘."0xHH+3000"
```

Remember the "-d" debug flag you have implemented; use this to print the populated array and check that the output is as expected.

To run the autograder, see the Autograder section.

2.8 Example Output

Note: In the following example commands, the '$' character demarcates the shell or command line, and should not be copied when testing these commands yourself.

For the following command, the array of structs is populated as shown in the debug output:

```
$ ./hw7 -d "3+0x40-7+5" "0xA6+5000-45" "0xHH+3000" "1+1+1+1+1+2+2+2+2+2+1+1+1+1+1"

Struct info at index 0:
Expression: 3+0x40-7+5
Result: 65
ErrorCode: 2110
--------------------

Struct info at index 1:
Expression: 0xA6+5000-45
Result: 5121
ErrorCode: 2110
--------------------

Struct info at index 2:
Expression: ERROR
Result: 0
ErrorCode: 302
--------------------

Struct info at index 3:
Expression: ERROR
Result: 0
ErrorCode: 301
--------------------
```

```
Struct info at index 4:
Expression:
Result: 0
ErrorCode: 0
--------------------
```

For an invalid number of command line arguments (given a max of 5 expressions), the output is as follows

```
$ ./hw7 -d "3+0x40-7+5" "0xA6+5000-45" "0xHH+3000" "4+3" "5-100" "0x8A+49+50"
PROGRAM ERROR: Too many expressions specified!
```

For -d with insufficient arguments, we expect a similar output

```
$ ./hw7 -d
PROGRAM ERROR: No expression specified!
```

Finally, for no arguments specified, we expect the following message (already implemented):

```
$ ./hw7
USAGE: ./hw7 [-d] "basic math expressions separated by quotation marks"
EXAMPLE: ./hw7 "3+0x40-7+5" "0xA6+5000-45"
EXAMPLE FOR PRINTING OUT DEBUG INFO: ./hw7 -d "3+0x40-7+5" "0xA6+5000-45"
```

3 Useful Tips

3.1 Man Pages

The "man" command in Linux provides "an interface to the on-line reference manuals." This is a great utility for any C and Linux developer for finding out more information about the available functions and libraries. In order to use this, you just need to pass in the function name to this command within a Linux (in our case Docker) terminal.

For instance, entering the command:

```
$ man strlen
```

will print the corresponding man page for the strlen function.

NOTE: You can ignore the subsections after the "RETURN VALUE" (such as ATTRIBUTES, etc) for this homework, however, pay close attention to function descriptions.

3.2 Debugging with GDB and printf

We highly recommend getting use to "printf debugging" in C early on.

Moreover, If you run into a problem when working on your homework, you can use the debugging tool, GDB, to debug your code! Former TA Adam Suskin made a series of tutorial videos which you can find here.

Side Note: Get used to GDB early on as it will come in handy in any C program you will write for the rest of 21ten or even in the future!

When running GDB, if you get to a point where user input is needed, you can supply it just like you normally would. When an error happens, you can get a java-esque stack trace using the backtrace(bt) command. For more info on basic GDB commands, Google "GDB Cheat Sheet."

3.3 Other Basic Linux Commands

```
// Echo the value of whatever is passed in as an argument
$ echo "Hello There :)"

// Print the contents of the file cat.txt to standard output
$ cat out.txt

// Concatenates to the end of the existing out.txt
$ ./hw7 -d "3+0x40-7+5" "0xA6+5000-45" >> out.txt

// Search for a string in a given file in Linux (with line numbers)
$ grep -n "ErrorCode" out.txt
```

4 Checking Your Solution

Important Notes:

1. All non-compiling homework will receive a zero (with all the flags specified in the Makefile/Syllabus).
2. **NOTE: DO NOT MODIFY THE HEADER FILES.**
   You must place any code elements you define (structs, macros, function declarations, etc.) in the C FILES. Usually placing those definitions in .h files would be good practice, but for this assignment you are not turning them in, and so the declarations would be lost when submitting.

To run the autograder locally (without GDB):

```
// To clean your working directory (use this instead of manually deleting the .o files
```

```
make clean

// Compile all the required files
make tests

// Run the tester object
./tests
```

This will run all the test cases and print out a percentage, along with details of the **failed test cases**.

Other available commands (after running make tests):

- To run a specific test case (to avoid all printing output/debug messages for all test cases):

  ```
  make run-case TEST=testCaseName
  ```

- To run a test case with gdb:

  ```
  make run-gdb TEST=testCaseName (or no testCase to run all in gdb)
  ```

The output file will **ONLY** be graded on Gradescope.

TA Note: Since C autograders can sometimes print out a lot of info, it might be a good idea to pipe the output to a file and investigate the content of the file instead! Use Gradescope for a cleaner output or run tests individually when debugging as mentioned above.

Many test cases are randomly generated and your code should work every time we run the autograder on it, however, there's no need to submit to Gradescope multiple times once you get the desired grade.

We reserve the right to update the autograder and the test case weights on Gradescope or the local checker as we see fit when grading your solution.

5 Deliverables

Turn in the following files

1. `my_string.c`
2. `hw7.c`
3. `gtUsername_out.txt` (piped and replaced with your gtUsername)

on Gradescope by Thursday, March 12th at 11:55pm.

There will be a 6 hour late period until March 13th at 5:55am. You will receive a 25% deduction during this late period. Anything after that will result in an automatic 0.

Note: Please do not wait until the last minute to run/test your homework, history has proved that last minute turn-ins will result in long queue times for grading on Gradescope. You have been warned.

6 Appendix

6.1 Appendix D: Rules and Regulations

6.1.1 General Rules

1. Starting with the assembly homeworks, any code you write should be meaningfully commented for your benefit. You should comment your code in terms of the algorithm you are implementing; we all know what each line of code does.

2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.

3. Please read the assignment in its entirety before asking questions.

4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

6.1.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to Canvas/Gradescope. You can create an archive by right clicking on files and selecting the appropriate compress option on your system. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.

3. Do not submit compiled files, that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

4. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

6.1.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; neither Canvas/Gradescope, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

6.1.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

6.1.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.

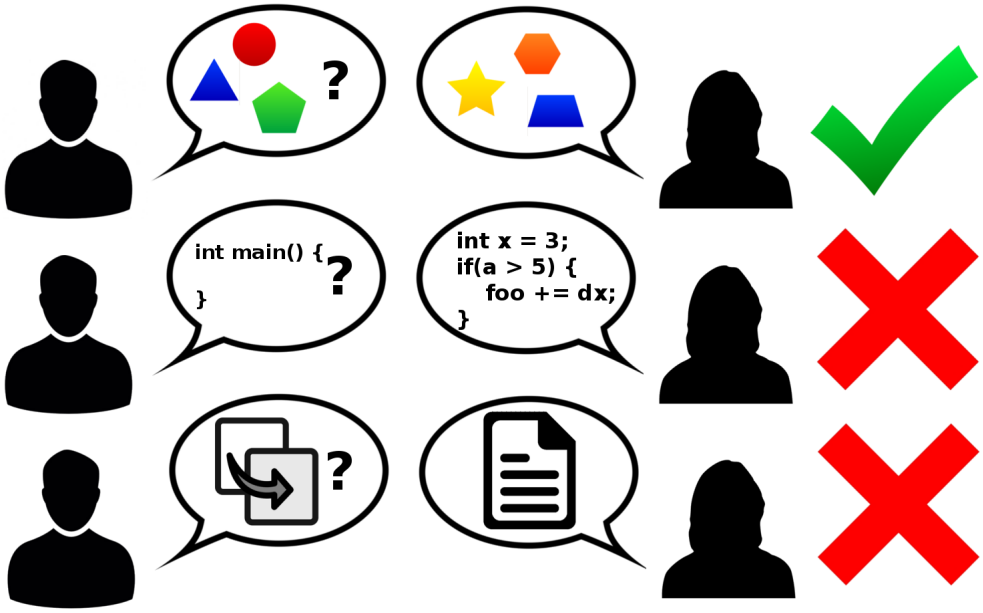

Figure 1: Collaboration rules, explained colorfully