Assignment 2: Whitespace Trimmer

CSE30: Computer Organization and Systems Fall 2021

Instructors: Bryan Chin and Keith Muller

Due: Sunday, October 10th, 2021 @ 5:00PM

Please read over the entire assignment before starting to get a sense of what you will need to get done in the next week. REMEMBER: Everyone procrastinates but it is important to know that you are procrastinating and still leave yourself enough time to finish. Start early, start often. **You MUST run the assignment on the pi cluster. You HAVE to SSH: You will not be able to compile or run the assignment otherwise.**

NOTE: Please read the FAQ and search the existing questions on Edstem before asking for help. This reduces the load on the teaching staff, and clutter/duplicate questions on EdStem.

Table of Contents

Learning Goals

- Learn to use GDB to debug programs
    - Common gcc flags

- Learn to implement a more complex C program
    - Use file I/O functions
    - Process input from file streams
    - Create helper functions

Part 1: Debugging with GDB [5 points]

GDB is a very important tool that will save you hours of time debugging, and being able to debug code on your own using the proper toolset is invaluable. In this section you will use GDB to determine what part of a file is causing issues.

What is GDB?

GDB is the GNU debugger. It allows you to see what is going on inside another program while it executes or the moment the program crashes. (https://www.gnu.org/software/gdb/)

Why use GDB? Can't I Just Use Print Statements?

You can and should use print statements in some cases BUT gdb allows you to take it a step further. You can execute a program step by step, print intermediate values, and set breakpoints to pause code execution at a specific line and do sanity checks at any point.

Compiling

Compile your program with gcc and add the `-g` option to add debugging information..

Example:

```
$ gcc -g -std=c11 -Wall -Wextra -o <exe file> <src files>
```

Argument	Description
`gcc`	Runs the GNU Compiler Collection (GCC)
`-g`	Flag to produce debugging information
`-std=c11`	Use the C11 language standard
`-Wall` (optional)	This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid
`-Wextra` (optional)	This enables some extra warning flags on top of `-Wall`
`-o <exe file>` (optional)	Name of the output executable (will be `a.out` if unspecified)
`<src files>`	Name of the source code files (just `loop.c` in our case)

Running GDB

```
$ gdb ./a.out
```

You will be presented with a prompt at which point you can begin entering GDB commands like `break`, `run`, `where`, `n`, `p <value>`, `q`, etc. Try it out!

Clone the following copy of the starter code on your CSE30 account and compile it with debugging symbols: https://github.com/cse30-fa21/A2-gdb. **You must do this work within the pi-cluster environment.**

The code sums all the values of the characters in a string. This can be used, very crudely, to detect errors in a string. Use GDB to pause the loop and tell us what the value of `sumIt` is **after** the body of the loop has executed 500 times.

GDB Commands

https://gcc.gnu.org/onlinedocs/gcc-3.3.6/gnat_ug_unx/Introduction-to-GDB-Commands.html

Argument	Description
`run`	executes the program from the beginning
`break`	set a breakpoint at a given line number or function
`next`	runs the next line of program (doesn't step into function)
`step`	runs the next line of program (steps into function)
`continue` (or `c`)	continues executing program until the next breakpoint
`print` (or `p`)	print the value of a given expression
`p/x`	print the value of a given expression in hex
`where`	shows the next statement that will be executed
`up` `down`	examine the caller/callee in the call stack
`info break`	list all the currently set breakpoints
`condition # <cond>`	make a breakpoint active when condition is true e.g. `condition 1 (x==5)` sets breakpoint #1 to be active only when `x` is 5.
`quit` (or `q`)	used to quit out of gdb

Submitting gdb.txt

Include a `gdb.txt` file in your submission along with the solution to part 2 on Gradescope. Cut and paste your GDB session (as shown in the terminal) into a text file called `gdb.txt`.The txt file should contain these interactions with GDB:

1. Show a listing of the program from within GDB by using the `list` command.
2. Show us how you set a breakpoint using condition breakpoints for the loop iteration where `i` is 500.
3. Print from GDB the value of `sumIt` **in hexadecimal** after the `sumIt` line has executed 500 times.

We will grade your `gdb.txt` looking for the proper value of `sumIt`.

For example, your `gdb.txt` (containing your GDB session) should follow this general format:

1. List the program using the **list** command

   `list`

2. Use the break command to break on the line : "`sumIt += someStr[i % len];`"

   `break` `some_line_number`

3. Set the breakpoint condition so that the break happens when i is 500

4. Run the program in gdb using the **run** command

   `run`

5. Print the value of `sumIt` in hex.

Part 2: Whitespace Trimmer [45 points]

Assignment Overview

For this assignment, we will be delving into some trivia about code development in Minecraft, specifically Mojang's workflow in fixing bugs. There exists a bug in the Minecraft: Java Edition that was originally opened about 7 years ago. [This bug is one where whitespace is undesirably trimmed from commands executed in-game]. What is funny about this bug is that though it has been fully investigated and already has a working code solution for it, the developers have somehow forgotten to properly merge it into the code base for these entire 7 years (because the bug is still in the game!!!). Software developers are more flawed than it seems...

A screenshot of the bug in action:

For this assignment, we will be reimplementing the function that caused the bug in C code, specifically the Java function [normalizeSpace() in the Java StringUtils library]. Don't worry about understanding the Java function, since we have written instructions that we want you to follow below. This is a class on C, not Java after all. Additionally, your programs will have to be able to handle files, so there will be some significant differences.

Getting Started

Developing Your Program

For this class, you **MUST** compile and run your programs on the **pi-cluster**. To access the server, you will need your `cs30fa21cxx` student account, and know how to SSH into the cluster.

Need help or instructions? See [Edstem FAQ] or watch [CSE30 Development Tools Tutorial]
(Do NOT wait until Friday to try this. There will be limited or no ETS support on the weekends!)

We've provided you with the starter code at the following link:
https://github.com/cse30-fa21/A2-starter

1. Download to get the `Makefile`, `README`, and template `trimmer.c`.
2. Fill out the fields in the `README` before turning in.
3. Open your favorite text editor or IDE of choice and begin working on `trimmer.c`

Running Your Program

We've provided you with a `Makefile` so compiling your program should be easy!

Additionally, we've placed the reference solution binary at:
`/home/linux/ieng6/cs30fa21c/public/bin/trimmer-a2-ref`.
You can directly use `trimmer-a2-ref` as a command.

Makefile: The `Makefile` provided will create a `trimmer` executable from the source files provided. Compile it by typing `make` into the terminal. By default, it will run with warnings turned on. To compile without warnings, run `make WARN=` instead. Run `make clean` to remove files generated by `make`.

How the Program Works

You will be given two files as command line arguments. Read from the first file, remove extra whitespaces from it, and write the result to the second file. The executable will be called like so:

```
./trimmer <input filename> <output filename>
```

Once you open a file, you need to remove all whitespaces and replace each of them with a single space (`' '`). Whitespaces include the following:

- Space (`' '`)
- Formfeed (`'\f'`)
- Carriage return (`'\r'`)
- Tab (`'\t'`)
- Vertical tab (`'\v'`)

We suggest that you write a helper function `int is_space(char c)` to account for all these types of whitespaces. This is a function that takes a `char` as a parameter, and returns `1` if the character meets the criteria above, and `0` otherwise. In order to use `is_space()` in `main()`, you must declare it before `main()`.

Special characters

Newline characters (`\n`) are **not** considered whitespace. When you encounter a newline, preserve it and eliminate **all** whitespaces you encounter before it. (See the third example in the table below.)

The null terminator (`\0`) will **not** appear in the file, nor will other non-visible characters not mentioned here (e.g. character 0x02 `STX` "Start of Text").

Entering special characters

To enter special non-visible characters in Vim, enter CTRL+V while in Insert Mode, and then type the 3-digit decimal code for it. For example, the carriage return `'\r'` is `013`. You will see a `^` under your cursor when you enter CTRL+V.

Examples

Here are some input and output examples. Your terminal will likely not show `\n` characters. Line breaks will automatically be inserted instead. If you want to see where the end of a line is, you can use the command `cat -e <filename>` to display a `$` character at the end of a line. To differentiate between whitespaces, use `hexdump -C <filename>` or `hd <filename>` for short, for a hex view of the file data.

Input File	**Output File**
`"I  love  Minecraft  \n"`	`"I love Minecraft\n"`
`"I\r  love  \v  \tMinecraft"`	`"I love Minecraft"`

`"I love Minecraft  \n"`
`"I  love  Minecraft  \n`
`"I  love Minecraft  \f"`
	`"I love Minecraft\n"`
	`"I love Minecraft\n`
	`"I love Minecraft "`
	(Note: There is a space at the end of the last line)

You **must** write your input files on the pi-cluster with Vim. Windows machines have end-of-line whitespace differences that cause problems (see last item on the Edstem FAQ).

Program Implementation

You will be given two filenames as command line arguments, `argv[1]` and `argv[2]`, to the `trimmer` executable. You will use `fopen` to open `argv[1]` for reading, and `fopen` to open `argv[2]` for writing. The function header for `fopen` is as follows:

```
FILE *fopen(const char *filename, const char *mode)
```

The filenames will be given to you through the command line arguments, and you must use `argv[1]` or `argv[2]`, depending on which file you are accessing. `const char *mode` tells you the access mode you are opening the file in. In this question, you will need to open a file for reading and another for writing. The various access modes are given in the table below.

Mode	Description
`"r"`	Opens a file for reading. The file must exist.
`"w"`	Creates a file for writing. If the file exists, it is overwritten.
`"a"`	Appends to an existing file, or creates a file and appends to it.
`"r+"`	Opens a file for reading and writing. The file must exist.
`"w+"`	Creates a file for reading and writing.
`"a+"`	Opens a file for reading and appending.

You can find more about `fopen()` here or `man 3 fopen`.

We suggest using `fgets()` to read one line of the input file at a time. Whenever you see a whitespace, replace it with one space (`' '`). Scan the array read by `fgets()`. Write to the output file every non-whitespace character. When you see whitespace of any length, make sure to write a single space instead. To write a character to a file, you can use `fputc()`. Solutions using `fgetc()` instead of `fgets()` will also be accepted.

You are guaranteed the following:

- No line will start with whitespace(s).
- There are no degenerate input files - all files contain only ASCII characters. **The only non-visible characters that will appear are:** **`' ', '\f', '\r', '\t', '\v', '\n'.`**
- We always give exactly two command line arguments to the program: the input filename, and the output filename.
- No line will be longer than 255 characters, excluding the null terminator.

Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled “A2: Whitespace Trimmer”. You will submit the following files:

   ```
   trimmer.c
   gdb.txt
   README.md
   ```

   You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint.

Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all of the files and upload the `.zip` to the assignment. Ensure that the files you submit are not in a nested folder.

2. After submitting, the autograder will run a few tests:
    a. Check that all required files were submitted.
    b. Check that `trimmer.c` compiles.
    c. Runs some sanity tests on the resulting `trimmer` executable.

Grading Breakdown [50 pts]

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade. Also, throughout this course, make sure to write your own test cases. ***It is bad practice to rely on the minimal public autograder tests.***

The assignment will be graded out of 50 points, with 5 points allocated to the GDB assignment (part 1), and 45 points allocated to the main Whitespace Trimmer program (part 2). Each test case will be equal to 5 points, with a total of 9 test cases. (There are 4 public test cases.) Make sure your assignment compiles correctly through the provided `Makefile` on the pi-cluster. **Any assignment that does not compile will receive 0 credit for part 2.**

Writeup version

1.0 Initial release.
1.0.1 Sections added regarding special characters, how to type them, and which ones will not be used.