1 Introduction

In this homework, you will be implementing an arraylist and debugging code for a "Random Message Generator."

Before you begin, we **heavily** recommend using the Docker to test your code locally. You are responsible for making sure your code works on the Docker container we gave you, as C may work slightly differently on different OSes. A guide for installing the docker image is on Canvas.

The files we provided include:

- `main.c`: main functions that generate the random message
- `main.h`: header file of the main function
- `arraylist.c`: functions that are used to manipulate the ArrayList (used by main.c)
- `arraylist.h`: header file of ArrayList data structure
- `arraylist_test.h`: header file for the tests for the arraylist data structure.
- `arraylist_test.c`: functions for testing the the arraylist data structure
- `Makefile`: A script that can be used for compiling the code (optional).

You will be modifying `arraylist.c`, `main.c`, and `main.h` for this homework. Before completing the various parts of this homework, read all the relevant files, and fill in your name and GTID at the top.

2 Part 1: Implement taking in parameters from command lines

First, let's knock out taking in arguments from the command line. There will be two command line arguments for this program. This code should be written in the main method in the main.c file. Look for inline comments that will describe where you should implement your code. The arguments are:

- `t` triggers the program to run the unit tests on the arraylist rather than generating a random message.
- `l` sets the length of the message that the program will generate.

We highly recommend using the `getopt()` function. The method head looks like this:

```
getopt(int argc, char *const argv[], const char *optstring)
```

where `argc` is the argument count, `argv` is argument array, and `optstring` is the specified list of characters, each representing a single character option. If the character in `opstring` is followed by ':', the corresponding option will take in an argument, and that argument will be pointed by `optarg`.

This function will return the next option character (if one is found) from `argv` that matches a character in `optstring`. If no character in `argv` is in `optstring` or there is a missing argument, it will return '?.' If we set the first character of `optstring` to be ':' and we encounter a missing argument situation, it will return ':' instead. Otherwise, it will return -1 which indicates that all option characters have been parsed.

2.1 Deliverable 2: GDB Snapshots

Once we implement taking in the command line arguments we want to test to ensure the program is taking in arguments correctly. Compile `main.c` and use GDB to test it. Don't specify any other files when compiling with gcc. Take a screenshot of your GDB program showing that you got the values of the variables through GDB and submit that with your code on Gradescope. You can pass in any value for `l` as long as $1 < \texttt{l} \leq 15$. One of your GDB tests should show that the local variable *length* has been correctly assigned to what you passed in for l, and the other tests should show that the local variable *tests* = 1. A guide to using GDB is located in Section 4 of this PDF.
Your screenshot should include the following:

1. Setting the breakpoint at the appropriate line to print the values stored in variables *length* and *tests*
2. Running the program in gdb with appropriate arguments
3. Printing the values of *length* and *tests*
4. Delete the breakpoint you set above.

If you can't fit all of this into one screenshot, you may upload multiple.

2.2 Compiling and Running

In the future, we will provide you with a Makefile for compiling your programs. However, for this assignment, we want you to be aware of how C programs are actually compiled. To compile a program you'll use the gcc compiler. To use the gcc compiler use the following format:

- gcc PATH/TO/C/FILE PATH/TO/C/FILE ... -o PROGRAM_NAME

Or if you're compiling to debug with gdb:

- gcc -g PATH/TO/C/FILE PATH/TO/C/FILE ... -o PROGRAM_NAME

Replace the capitalized words with the appropriate file's paths. The program name can be anything you would like, since we will compile your programs on our own during grading. However, we will call the program Word_Generator if you would like to use that name. You will need to compile all the .c files that will be used in your program. In this case, that means you will need to compile all the **.c** files in the **/src** directory. If that seems like a lot to type whenever you want to compile you can use the **Makefile** provided. Essentially, add your gcc command to the line under "final:" and now typing make should compile your code. Please note that the line after "final:" needs to be indented. Alternatively, you can compile all the **.c** in your folder with `gcc *.c`, making sure you're in the folder of the files you are trying to compile. The Makefile is not turned in and is optional. Makefiles in the future will be provided and will have more functionality.

To run the program you will simply need to type the path to your program. One thing to note is that you will have to put "**./**" before the filename if you are in the same directory as your program. For example, if you use the name Word_Generator:

- `From same directory`: ./Word_Generator -l 10
- `From different directory`: ./PATH/.../Word_Generator -t

3 Part 2: Implement the ArrayList methods

Next, implement the functions for the ArrayList data structure in `arraylist.c`:

- `create_arraylist`: Taking the capacity of the arraylist as an input, create an ArrayList using the backing array with type `** char`. Remember to use `malloc()` to allocate space for the storage of the ArrayList. The function should return a pointer to the ArrayList you created.
- `add_at_index`: Add a word to ArrayList at a specific index. If the length of the ArrayList is going to exceed the capacity, resize the ArrayList to twice its original capacity. You can call `resize()`, which is also a function you will implement, to do this.
- `append`: Add a word to the end of the ArrayList. Remember to resize as required in `add_at_index`.
- `remove_from_index`: Remove a word from the ArrayList at a certain index and return the word removed. Make sure to maintain the ArrayList contiguous.
- `resize`: Resize the backing array to hold twice its original capacity. It should now support `arraylist->capacity` $*$ 2 elements.
- `destroy`: Destroy the ArrayList by freeing the ArrayList itself and its backing array.

Please refer to the comments in *arraylist.h* for further guidelines. Also note that *arraylist.c* is almost blank. In short, you need to implement the functions defined in *arraylist.h*. Think of *arraylist.h* as the interface that your code is expected to comply with. If your code passes the *arraylist_tests* sanity tests and your word generator gives the expected output, then your code should be good. We reserve the right to review the code to ensure that your implementation is not hard coded or incorrect (i.e. implemented a link list instead of an array list).

4 Part 3: Debug the main file

For the last part of the project, you will need to debug the provided code in `main.c` and `main.h`. This code is used to create a "random word generator" that, in short, draws several random words from an array called "Dictionary". There are **3** problematic lines of code that are causing problems located somewhere throughout the provided code. If you get stuck look for the comments for hints! This section is to prepare you for debugging C code on your projects that are much larger and more complex than this code.

Please note, that this code almost works! By eliminating the mistakes causing segmentation faults, the code should run properly. All **3** errors in the code should cause segmentation faults and no other errors. If you are unclear if your fix changes the intended functionality of the code, please refer to the comments.

The intended output from the program is:

- `-l 5`: Message: this tea Half is nothing
- `-l 10`: Message: this tea is nothing more Half than hot leaf juice
- `-l 15`: Message: this tea is nothing more than hot Half leaf juice cs2200 rocks momo secret tunnel

The output shown above should be the **only** output onto stdout (or stderr) that your program makes when the l argument is used. When the t argument is used **only** the outputs of the *arraylist_tests* should be shown on stdout (or stderr). We will not run your program with both the t and l arguments used simultaneously.

If this part seems tedious, often small bugs that are easily overlooked lead to headaches on the projects. Hopefully, this practice will allow you to locate them easier in the future.

4.1 GDB Debugging Tips

If your program is crashing or misbehaving, you can use GDB to locate the bug. GDB is a command line interface that will allow you to set breakpoints, step through your code, see variable values, and identify segfaults. There are tons of online guides, click here for one. (http://condor.depaul.edu/glancast/373class/docs/gdb.html)

First (after compiling with the -g flag) load your binary file into GDB using:

`$ gdb <file name>` or in our case `$ gdb Word_Generator`

Within GDB, you can run your program with the `run` command:

```
$ (gdb) r <command line arguments>
```

Which in our case is either one of these:

```
$ (gdb) r -t
```

```
$ (gdb) r -l <# of words>
```

You may find it useful to set a breakpoint inside the main loop of the simulator to debug specific simulator commands in your implementation. You can do this either by finding the line number inside `main.c` and breaking there:

```
$ (gdb) break main.c:53 !  set breakpoint at call to system_init
$ (gdb) r -l <# of words>
!  (wait for breakpoint)
$ (gdb) s !  step into the function call
```

or by using the actual function name being called from the main loop:

```
$ (gdb) break sim_cmd !  set breakpoint at call to sim_c
$ (gdb) r -l <# of words>
!  (wait for breakpoint)
$ (gdb) s !  step into the function call
```

Sometimes, you may want to examine a large area of memory. To do this in GDB, you can use the x command (short for examine). For example, to examine the first 24 bytes of the frame table, we could do the following:

```
$ (gdb) x/24xb frame_table
0x1004000aa:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000b2:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000ba:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

The format of this command is `x/nfu [memory location]`, where n is the number of items to print, f is a formatting identifier, and u is the specifier for the units you would like to print. b specifies 1 byte, h specifies 2 bytes, w specifies 4 bytes, and g specifies 8 bytes.

If you use the corruption checker, you can set a breakpoint on panic() and use a backtrace to discover the context in which the panic occurred:

```
$ (gdb) break panic
$ (gdb) r -i <# of words>
!  (wait for GDB to stop at the breakpoint)
$ (gdb) backtrace
$ (gdb) frame N ! where N is the frame number you want to examine
```

Lastly, you can use gdb to see the value of a variable without print statements using:

```
$ (gdb) print <variable name>
```

If the program runs to completion, the variable will have a value of 0, so you will need to set a breakpoint at the relevant spot where the value is assigned.

5 Submission

Files you will submit on **Gradescope**:

- `Your GDB snapshots`
- `arraylist.c`
- `main.c`
- `main.h`

Please make sure to **remove any extra code you have added while debugging** outside of what is outlined in this PDF (such as print statements) as this may cause issues with the autograder.