Instructions

There are two deliverables for this assignment: a C source file named `strings.c` and a header file named `strings.h`. Please commit these files to GitHub and submit via Gradescope. Only one member of each pair needs to do this, though do remember to add the other person's name to the submission. You are expected to practice pair programming, where both partners are working together on a shared screen to complete this assignment. See the explanation pair programming in the syllabus for more-detailed guidelines.

Implementing a C String Library

On this assignment, you will implement your own version of some core functions in C's string processing library. Recall that in C, strings are just blocks of memory (e.g., static arrays, or a dynamically allocated block of bytes) that contain one character after another, terminated by a special `\0` character. Performing any non-trivial operation on a string (like say concatenation or comparison) requires calls to library functions. Since strings are such fundamental building blocks, it's extremely likely that you'll find yourself using these functions frequently. A great way to learn about how they work is to implement them yourself.

Here is the interface of the library (`strings.c`) you'll be implementing:

- `int mystrlen(const char*)`
- `int mystrcmp(const char*, const char*)`
- `char* mystrcpy(char*, const char*)`
- `char* mystrcat(char*, const char*)`
- `char* mystrchr(const char*, int)`

All of these functions should behave identically (including edge cases) to their standard C library counterparts (which have the same names, except for the `my` prefix). You may assume that your implementation will not be tested on inputs where the behavior of the original functions is undefined. To learn about the specification for a particular function, you can find its documentation pages on the web—or directly from inside a Linux terminal. For example, typing:

```
man strchr
```

at the prompt will bring up the manual pages for `strchr`. You can scroll with the arrow keys, use space-bar to advance to the next page, or press `q` to quit and return to the prompt. The `man` command is very handy—you can bring up manual pages for most of the common C functions, shell commands, and other UNIX programs, without ever leaving the terminal environment.

To test the correctness of your library, use a *separate* tester program. Your `strings.c` program *may not contain a main function.* Instead, use the `tester.c` program for testing and debugging. A few examples have been provided, but you should add many, many more tests! The reason for this requirement is that the `strings.c` file is meant to be a library, i.e., a pure collection of functions that are meant to be used by other programs. Notice how the `tester.c` program imports the library:

```
#include "strings.h"
```

The quotes indicate that `strings.h` is a user-defined library located in the current folder, as opposed to built-in libraries that are included with angle brackets. You should compare the behavior of your string manipulation functions to those in the standard C library to verify correctness. Your tester file will not be graded, but is nonetheless a critical part of the assignment!

Function Declarations, Libraries, and Header Files

We've seen throughout the semester that the C compiler needs to know about the functions you will be calling before you call them. In simple programs, this can be accomplished either by defining our functions before they are called in main (or other functions), or by declaring them at the beginning of the program, and defining them later. However, as our programs get more involved, we will naturally want to divide their source code among multiple files, so the question arises: how can we call functions that are defined in another source file?

The first step in solving this in C is to include multiple source files in our compilation, for example:

```
gcc library.c program.c
```

but this can still run into problems, because linking the files together is the last stage of compilation, so when compiling each file individually, `gcc` doesn't know about functions defined in other source files. The way around this is to separate the function *declarations* into a header file with the extension `.h`, and the *definitions* into a source file with the extension `.c`. Then the header file can be included in both the library and the program.

When we use a `#include` statement, the contents of the header file are literally pasted into our program by the compiler's pre-processing step. This means that if we `#include library.h` at the top of our `program.c` source code, it's as though we had typed out the prototypes for all of the functions declared in `library.h`, so the compiler will happily allow us to call those functions within `program.c`.

Aside: Here's a different way to think of the header file and the library file. The header file specifies an *interface* — it lists the prototypes of the functions that are exported by the library file, that can be used by any consumer of the library. If well-documented, then this is all the user needs to use the library — think of how you use functions like `printf`. You don't necessarily know the details of how `printf` is implemented, but you understand its interface, i.e., what inputs it expects and how it behaves. The library file's job is to provide the *implementation* for each of the functions declared in the interface. Note that the library file may contain additional functions, whose prototypes do *not* appear in the header file. These are functions that are not meant to be exported, but are used internally by the library (for example, helper functions).

Other Notes and Constraints

- Pay careful attention to the names! Your library file should be named `strings.c` (note that the word is pluralized) and the header file, `strings.h`. Similarly, be careful when naming the various functions.
- This should be obvious: you are *not* to use any of the functions from the standard C library (i.e., from `string.h`) to implement your solutions.
- Avoid copy-pasting code. If you find yourself needing to perform the same (non-trivial) operation in more than one function, then delegate that work to a helper function. This helper function should not be exported via the header file, i.e., you should not create a prototype for this helper in `strings.h`. It should stay private to your library.
- That said, it's possible that the functionality provided by one of the functions that your library *does* export could be useful in implementing a different function — for example, you may decide to use a call to `mystrlen` when implementing `mystrcat`. This

is ok — you might want to carefully think about the order in which you implement the various functions to maximize these opportunities for code reuse.

- In many functions, you may need to test for the end-of-string character. There are various ways to perform this test, but the clearest is to compare against the `'\0'` character with `==`.
- Perform incremental testing — there's no reason to wait until you've written every function to test them all. Test each function as it's written!
- Your library functions should not print any unnecessary output to the screen.
- You can use CNTRL-C to kill your program if it appears to be stuck in an infinite loop.

Code Style Guidelines

Your code should abide by the guidelines outlined in the `250_style_guide` document that is included in this repository. The following (non-exhaustive) checklist may be handy when preparing your code for submission.

Checklist

- Correctness and Defensive Programming
  - ☐ The file `strings.c` implements the functions specified in the assignment and `strings.h` exports the specified interface.
  - ☐ The `strings.c` file does not contain a `main` function.
  - ☐ The source file compiles with no error messages when using the `-Wall` flag.
  - ☐ All of the specified functions run without generating any `valgrind` errors or `valgrind` warnings about memory leaks.
  - ☐ The function implementations check the return values of all calls to the library functions listed in the style guide.
- Documentation
  - ☐ A comment header at the top of your file has a brief description of your library, the names of you and your partner, and the time you spent on the assignment.
  - ☐ Every function has its own comment header in the `.h` file that briefly describes its purpose, parameters, and return values.
  - ☐ Comments inside functions describe the sub-tasks solved by the function, when appropriate.

- Readability
  - ☐ Variable and function names are meaningful.
  - ☐ Local variables are used to the maximum extent possible, without resorting to unnecessary global variables.
  - ☐ The code is indented neatly and consistently.
  - ☐ Long lines of code are wrapped to the next line if they interfere with readability.
  - ☐ "Magic numbers" and other constants have been eliminated using the `#define` directive.
  - ☐ Boolean expressions are used in the simplest form possible.
- Structure and Parsimony
  - ☐ The code is free of unnecessary complications and redundant fragments.
  - ☐ The library displays evidence of thoughtful design, with appropriate decomposition and helper functions.

Submission

Submit the files `strings.c` and `strings.h` by pushing to GitHub and uploading to Gradescope. Only one member of your team needs to upload to Gradescope, though you should remember to attach your partner's name to the submission. Note that not following the given specifications — misnaming your files, misnaming your functions, having your functions accept an incorrect number of parameters or parameters in the incorrect order, not returning the correct value in your functions, etc. — will result in significant autograder failures, so read the handout again carefully if this happens.