Learning Outcomes

After completing this program, students will be able to

- **Write** a *library* of functions in a separate assembly file.
- **Manipulate** C-style strings (character arrays).

Note: For *this* assignment you may make use of the **`mult`** instruction it MIPS.

Overview

Students will write a program that prompts the user for a *line* of text until the user enters `"done"`. Each line will be broken up into "words" on *space* characters with each word printed on a separate line.

Think for a minute: how would you write this program in Java using the `length`, `indexOf`, `substring` and assignment to another string, `word`. The input string is split into parts separated by single space characters until the line is empty.

0

Procedure

1. Read this entire assignment. Spend a few minutes thinking about what you are going to do.

String Library

2. You will write (at least) two assembly files. One, `main.s`, will contain the `main` method along with the **`.data`** segment and program specific functions.

   The other file, `string.s`, contains all of the standard routines in the string library. Your library must include *all* of the following **global** functions; the global functions must adhere, **exactly**, to the given interfaces. If you discover a better signature or interface for a function you must give it a *different* name and then *implement* the global functions using your great new function.

   Terminology: a *buffer* and a *string* are both passed as the address of (pointer at) a character array (`char *`). A *string* is always `NUL`-terminated (ends with `'\0'`); a *buffer* is not, necessarily, `NUL`-terminated and can be space set aside to hold a new string.

   `char * chomp(char * str)`
   `str` – a *non-null* **string**
   "Chomps" off all *trailing* tabs, carriage returns, line feeds, and the like. This is done by trimming *all* characters with ASCII codes less than 32 (the space character, `' '`, is ASCII 32). Trimming is halted when the string is *empty* or a character at or above code 32 is encountered.
   Returns `str`.

   `int strlen(char * str)`
   `str` – a *non-null* **string**
   Counts the number of characters (bytes) before the `NUL`.
   Returns the *length* of `str`.

   `char * strcpy(char * destination, char * source)`
   `destination` – a *non-null* **buffer**
   `source` – a *non-null* **string**
   Copies the string `source` into the memory pointed to by `destination`, including the `NUL`.
   Returns `destination`.

`char * strcat(char * destination, char * source)`
`destination` – a *non-null* **string**
`source` – a *non-null* **string**
Copies the string `source` onto the end of the string `destination`, including the `NUL`.
Returns `destination`.

`char * strncpy(char * destination, char * source, int n)`
`destination` – a *non-null* **buffer**
`source` – a *non-null* **string**
`n` – the *maximum* number of characters to copy
Copies the string `source` into the memory pointed to by `destination`, including the `NUL` if `strlen(source)` is less than `n` and just `n` characters otherwise (*Correct:* might **not** copy in a `NUL`).
Returns `destination`.

`char * strncat(char * destination, char * source, int n)`
`destination` – a *non-null* **string**
`source` – a *non-null* **string**
`n` – the *maximum* number of characters to copy
Copies the string `source` onto the end of `destination`, including the `NUL` if `strlen(source)` is less than `n` and just `n` characters otherwise (*Correct:* might **not** copy in a `NUL`).
Returns `destination`.

`int strcmp(char * left, char * right)`
`left` – a *non-null* **string**
`right` – a *non-null* **string**
Compare `left` to `right`, character by character returning an int reflecting the relationship between them.
Returns **negative** value if `left` < `right` (lexicagraphically), **zero** if they are the same, and a **positive** value if `left` > `right`.

`char * strchar(char * str, char ch)`
`str` – a *non-null* **string**
`ch` – a character to search for
Scan across `str` for a match for `ch`.
Returns address of first occurrence of `ch` in `str` or, if none is found, the `NULL` pointer.
(**`NULL`** != **`NUL`**)

`char * substring(char * destination, char * source, int start, int n)`
`destination` – a *non-null* **buffer**
`source` – a *non-null* **string**
`start` – the *offset* in `source` from which to start copying
`n` – the *maximum* number of characters to copy
Copies the string from `source[start]` to `destination` until `n` characters or the end of `source` is reached. If `start` is out of bounds, copy zero characters. If fewer than `n` bytes were copied, append a `NUL`.
Returns `destination`.

3. The **main** program that uses the string library follows a really simple CS I program: loop, prompting the user for a *line* of text and breaking the line into individual words, printing one word per line, until the input is the sentinel value.

   In a C/Java-like syntax:

```
1  while (true) {
2    char *  line = promptReadAndChomp("Next line: ", line, 100);
3    char * word;
4
5    if (strcmp(line, "done") == 0)
```

```
6       break;
7
8     char * space; // location of leftmost ' ' in line
9     while ((space = strchr(line, ' ')) != NULL) {
10      word = substring(word, line, 0, space - line);
11      line = substring(line, line, (space - line) + 1, 100);
12      println(word);
13    }
14    if (strlen(line) > 0)
15      println(line); // whatever is leftover
16  }
```

Testing

4. You should build small test programs for each routine in the string library. To have MARS do the right thing:

   - Start with your test program, `testChomp.s` in the same folder as `strlib.s`. Write a program that, say, reads a line and chomps the end off of it.
   - Test the library routine by assembling the test program and giving it some input (or build data in RAM if that makes more sense to you). (Make sure, with `chomp` in particular, to have more than just `\n` at the end of the string; remember that `\t` should get chomped, too.)
   - After you are happy that the test passes, make a sibling directory for your main, `strlib.s`, directory. Maybe call it `testChomp`. Move `testChomp.s` to that folder so that you can put a different `main` file in with the library.
   - **Notice** this is a great time to check the working code into `git`, too.
   - If, in the future, you seem to have problems with `chomp`, you could copy the current state of `strlib.s` into `testChomp/` and run `testChomp.s` against the newest library.

     Don't forget to backport changes in `testChomp/strlib.s` to the main copy. When done with this testing, you should make sure the backporting is done and *delete* the copy of the library. Really. Use the **DRY** Principle — **D**o not **R**epeat **Y**ourself: have a *canonical* place for everything and *only* keep it there.

   This test pattern is not part of the grading rubric (would probably garner *Aesthetic* points). Your code should work, be clean, and be well documented. Building working components is a good way to get to working code.

Submit through Gitea

Check your working code into `git`. When you have one working "feature" in your program, checking it in to `git` is a *good thing*. Protect yourself from making things *worse*.

Use a `.gitignore` file to exclude any garbage files your IDE produces from the repository. They will cost you points.

You have an account on the departmental **Gitea** server. Submit your work in your shared organization in a repo named `pAddingSomeNumbers`