1. String Definition

Strings are a sequence of values. Most of the time, they are thought of as a sequence of characters. For the purposes of implementation in C an array of characters is used, and expected by most libraries that take strings as input.

Since arrays are pointers in C. Passing a string to a string function does not require copying over all the memory, but instead it will pass in the pointer (the address). The address can be a lot smaller than an actual string, especially when dealing with genomes, which could contain billions of base pairs.

NULL Terminator

Every string function needs to know how long the string is. In C an extra character (the NULL terminator) is appended to the end of a string to denote the end. Keep in mind that characters can be thought of as values in the range of 0 to 255 inclusive. The NULL terminator uses the value of 0. The NULL terminator can also be represented using the character literal `'\0'`.

Since the NULL terminator uses the value of 0, it is the only character that evaluates to false in C, which makes code shorter sometimes when checking for NULL terminators.

This addition of the NULL terminator does mean that typically more memory than characters in the string are required when allocating the memory.

The string "Apple" which has 5 characters, must be stored in an array with **at least** 6 characters. Failure to do so can result in array out of bounds errors when trying to store "Apple" using built in string functions. My recommendation is that you always add 1 to the allocation for strings. Here is a picture of how the values in a string storing "Apple" would look.

Index	0	1	2	3	4	5	6
Value	'A'	'p'	'p'	'l'	'e'	0	??

For example, if you need an array for string of up to length 20, then do the following

```
#define SIZE 20

int main()
{
    char word[SIZE + 1];

    // Read in the word and process it here...

    return 0;
}
```

2. String Functions

There are several functions that are part of the c standard library. These functions can be included in your program by using the <string.h> header file. Since strings are arrays, and arrays are pointers, each string function deals with character pointers. Below are a list of some of the functions in <string.h> of which you should be aware.

`strlen`

Probably the most simple string function is the strlen. It takes as an argument the pointer to (the address of) the first character of the target string, and it returns the number of characters that occurs before the first NULL terminator. Usage of the strlen can be seen below

```
#include <string.h>

int main()
{
    // Find the length of the word "Apple"
    // Note 5 will be stored in integer below
    int lenOfWord = strlen("Apple");

    // Exit the program
    return 0;
}
```

What is interesting about this is that the number of characters before the first NULL terminator is also the same as the ***index*** of the first NULL terminator. Below is an example of how strlen can be implemented.

```
int myLen(char * str)
{
    // Find the index of the first NULL terminator
    int index = 0;
    while (str[index] != '\0')
    {
        index++;
    }

    // Return the index of the first NULL terminator
    return index;
}
```

Note that since any character in the array could be a NULL terminator, the strlen function **needs** to look as all possible characters up until the first NULL terminator. There is a common mistake beginning programmers make that causes some programs to be needlessly slow when processing strings that are not changing.

The below example code is a program that is slow.

```
#define SIZE 1000000

int main() {
    // Make a string with A LOT of A/B's
    char str[SIZE + 1];
    for (int i = 0; i < SIZE; i++)
        str[i] = 'A' + (rand()%2);
    str[SIZE] = '\0';

    // Count the number of 'A'
    int count = 0;
    for (int i = 0; i < strlen(str); i++)
        if (str[i] == 'A')
            count++;

    return 0;
}
```

Below is a faster version of the above program.

```
#define SIZE 1000000

int main() {
    // Make a string with A LOT of A/B's
    char str[SIZE + 1];
    for (int i = 0; i < SIZE; i++)
        str[i] = 'A' + (rand()%2);
    str[SIZE] = '\0';

    // Count the number of 'A'
    int len = strlen(str);
    int count = 0;
    for (int i = 0; i < len; i++)
        if (str[i] == 'A')
            count++;

    return 0;
}
```

Rather than recomputing the string length everytime (which does not change). We compute it once and use the store value. Here is a third version that is arguably faster.

```
#define SIZE 1000000

int main() {
    // Make a string with A LOT of A/B's
    char str[SIZE + 1];
    for (int i = 0; i < SIZE; i++)
        str[i] = 'A' + (rand()%2);
    str[SIZE] = '\0';

    // Count the number of 'A'
    int count = 0;
    for (int i = 0; str[i] != '\0'; i++)
        if (str[i] == 'A')
            count++;

    return 0;
}
```

strcpy

The next string function discussed in these notes is the string copy function. strcpy copies the contents of one string into another. One of the strings is overwritten with the new string. The function takes as input 2 string pointers. The function returns the pointer to the string written to. **Author Note: I almost never use the return value from this function**

The input to the function is first the string being overwritten and second the string from which the characters come. Alternatively, you can think of it as the destination string before the source string. Mnemonically, I remember "`strcpy(a,b)`" as the string equivalent of "`a = b;`". Keep in mind that string assignment typically does not work the way you will probably intend in C. The string copy function is how string assignments should be performed. Below is an example of strcpy,

```
#include <string.h>

int main() {
    // Store the word "Apple" into str
    char str[5 + 1];
    strcpy(str, "Apple");

    // Exit the program
    return 0;
}
```

Author's Note: The src and dest should not overlap (UB). Beware of array out of bounds (UB)

Below is an example of how strcpy could be implemented.

```
char * myCpy(char * dest, char * src) {
    // Store the contents of the src string upto the NULL terminator
    int index = 0;
    while (src[index] != '\0') {
        dest[index] = src[index];
        index++;
    }

    // NULL terminate the destination string
    dest[index] = '\0';

    // Return the location of the destination string
    return dest;
}
```

`strcat`

The cat in strcat is short for concatenate. The function like strcpy takes in the destination and source string pointers (in that order). The function also returns the resulting destination string. The contents of the second string are written to the end of the first string.

Mnemonically, "`strcat(a, b)`" can be thought of as "`a += b;`". Again since strings are pointers. You can/should not use standard incremental operators (e.g. +=) on them. Below is an example of strcat.

```
#include <string.h>

int main() {
    // Store the word "Apple" into filling and "Pie" into pastry
    char filling[8 + 1];
    char pastry[3 + 1];
    strcpy(filling, "Apple");
    strcpy(pastry, "Pie");

    // Make an "ApplePie"
    strcat(filling, pastry);

    // Exit the program
    return 0;
}
```

Author Notes: dest and src should not overlap (UB). Beware of array out of bounds (UB). Don't assume that uninitialized memory will be "empty strings" when using strcat (UB)

On the next page is an example of how strcat could be implemented.

```
char * myCat(char * dest, char * src) {
    // Find the end of the destination string
    char * newDest = dest;
    while (newDest[0] != '\0')
        newDest++; // Mild pointer abuse

    // Copy all the source to the end of the new dest
    int index = 0;
    while (src[index] != '\0') {
        newDest[index] = src[index];
        index++;
    }

    // Terminate the resulting string
    newDest[index] = '\0';

    // Return the location of the destination string.
    return dest;
}
```

strcmp

The strcmp function is useful for finding the lexicographical ordering between 2 strings. Lexicographical is not the same as alphabetical. The value of the characters is used for sorting, which means that `'Z'` comes before `'a'`. The value returned from strcmp is guaranteed to be:

- 0 if the two strings are identical.
- A positive value if the first differing characters in the 2 strings is greater for the first argument.
- A negative value if the first differing characters in the 2 strings is greater for the second argument.

Author's Note: Don't check that the return is 1, 0, or -1, because it could be any positive value not just 1. Similarly the negative value might not be -1.

Strcmp cannot directly be used to check if a string is a prefix of another string. A prefix of some original string is a string that is composed of some contiguous substring of characters starting at the beginning of the original string. A substring of some original string is a contiguous subsequence of characters that occurs in the original string.

Mnemonically, "`strcmp(a,b)`" is similar to "`a-b`". Once again you should not try to actually subtract strings in C, instead use the strcmp method.

On the next page is an example of using strcmp.

```
#include <string.h>

int main() {
    strcmp("apple", "Banana"); // Some positive value (b/c 'a' >
'B')
    strcmp("apple", "Apple"); // Some positive value (b/c 'a' > 'A')
    strcmp("apple", "apple"); // 0 (b/c equal strings)
    strcmp("apple", "banana"); // Some negative value (b/c 'a' <
'b')
    strcmp("app", "apple"); // Some negative value (b/c '\0' < 'l')
    strcmp("apple", "app"); // Some positive value (b/c 'l' > '\0')

    // Exit the program
    return 0;
}
```

Below is a possible implementation of strcmp

```
int myCmp(char * first, char * second) {
    int index = 0;

    // Find the first difference or the end
    while (first[index] == second[index] && first[index] != '\0')
        index++;

    // Check if the strings were equal
    if (first[index] == second[index])
        return 0;

    // Different strings; return the difference
    return first[index] - second[index];
}
```

`strncmp` (not on Foundation Exam)

The strncmp function is like string compare where only a limited number of characters is compared. The limited comparison can enable checking if a string is a prefix or check how long of a prefix 2 strings have in common. The arguments for the function are 2 strings and the number of characters to compare. The return value is similar to the return of strcmp, with the exception that no characters are checked beyond the specified limit.

On the next page is an example of using strncmp.

```c
#include <string.h>

int main() {
    strncmp("app", "apple", 3); // 0 (b/c first 3 characters are ==)
    strncmp("app", "apple", 1); // 0 (b/c first character is ==)
    strncmp("app", "apple", 4); // negative value (b/c '\0' < 'l')
    strncmp("app", "apple", 20); // negative value (b/c '\0' < 'l')
    strncmp("apple", "apple", 20); // 0 (b/c strings are ==)

    // Exit the program
    return 0;
}
```

`strdup` (not on FE)

When we get to dynamic memory strdup will make more sense. The strdup function is useful if

- You want to reduce the memory used to store
- You need to create a copy of a string dynamically
    - You did not know the size of the string at the time of creating the memory
    - Structs with strings would be a good example for this

The function takes as an argument a character pointer, and returns a dynamically created copy (with a null terminator) that uses the exact size necessary. Strings created this way should be free'd, which means if you are using this function you need to also include <stdlib.h>. An example of strdup can be seen below,

```c
#include <string.h>

int main() {
    // Create a string called hello
    char * str = strdup("hello");

    // Make the first letter capital H
    str[0] = 'H';

    // Print the string
    printf("%s\n", str);

    // Clean up the memory created
    free(str);

    // Exit the program
    return 0;
}
```

Checking for a NULL pointer is a good idea when using strdup.

`strstr` (not on FE)

The strstr function is useful to check if some string (needle) is contained in another (haystack). The 2 strings are passed as arguments: haystack first and needle second. The return value is the start of the first location that the needle is located in the haystack. If the needle is not contained in the haystack, NULL is returned instead. Below is an example of strstr being used.

```
#include <string.h>

int main() {
    // Check if cab is contained in abacaba
    if (NULL != strstr("abacaba", "cab"))
        printf("cab is contained.\n");
    else
        printf("No cab is found.\n");

    // Exit the program
    return 0;
}
```

`strcasecmp` (not on FE)

The strcasecmp is a useful function to compare strings, where you want words that differ by only case (uppercase/lowercase) to be treated as equal. If there is a difference that is not just by case, then the standard lexicographical comparison is used.

3. Reading Strings

Reading strings can be intimidating. You could try to use some method to read a single character at a time to parse your string, but there are more functions that can help parsing depending on your needs.

`scanf`

In my opinion the simplest method to use is scanf using the “%s” format specifier. Using “%s” will read in a string that is delineated by *any* whitespace. This means that if you wanted to read “Travis Meade” into a string, you would need to use “%s” twice since there are two “strings” separated by a space.

The downside to this method is that it is hard to tell if the words are separated by a single space or a tab or newlines. It is for this reason that many programmers opt to use a method that will read a full line. On the next page is an example of reading strings using scanf and “%s”.

```c
#include <string.h>

int main() {
    // Declare the arrays that will hold the input.
    // Hopefully they are large enough
    char filling[5 + 1];
    char pastry[3 + 1];
    char dish[5 + 1 + 3 + 1];

    // Read in "Apple" and "Pie" which are separated by some
    // whitespace
    scanf("%s%s", filling, pastry);

    // Create the dish ("Apple Pie") from the given input
    strcpy(dish, filling);
    strcat(dish, " ");
    strcat(dish, pastry);

    // Print the resulting dish
    printf("We made %s!\n", dish);

    // Exit the program
    return 0;
}
```

`gets` **(DEPRECATED)**

From the description in the man pages "Never use this function." This function is the source of well known bugs in deployed code.

That being said the function will read into a string characters until a newline or End Of File character is reached. The last character read into the buffer (newline/EOF) is replaced with a NULL terminator.

The return value of the gets is the NULL pointer if the read fails (probably already at EOF) and the destination pointer if successful.

`fgets`

fgets more or less stands for file get string. It is not deprecated. It takes 3 arguments:

- First: To where the string is being written (dest)
- Second: The size of the character array
- Third: The file point of the stream from where the word is coming (src)

The method reads a full line upto and including newline characters/EOFs. However, if reading that many characters would overfill the character array. Instead all but the last character will be filled with the contents from the specified file pointer, and the last character would become the NULL terminator. Since fgets also writes into the string the newline character it is most of the time important to make the strings an additional character longer than necessary. Below is an example of the fgets function.

```
#include <string.h>

int main() {
    // We will read "Apple Pie" from the input using fgets
    // Make a string that is long enough
    // 5 for "Apple"
    // 1 for the space
    // 3 for "Pie"
    // 1 for the newline
    // 1 for the NULL terminator
    char line[5 + 1 + 3 + 1 + 1];

    // Read in the line
    fgets(line, 5 + 1 + 3 + 1 + 1, stdin);

    // Print the line read in
    printf("We read in \"%s\".\n", line);

    // Exit the program
    return 0;
}
```

Author's Note: You should probably use MAX_LINE_SIZE + 2 when creating a character array to hold the information you will read in using fgets. It should hopefully be obvious that passing a larger value than the size of the buffer is UB. You should also not try to use sizeof(str) for determining how many characters are in the string.