Arrays of Strings

We can make an array of strings to group multiple strings together:

```
char *stringArray[5];   // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```
char *stringArray[] = {
     "Hello",
     "Hi",
     "Hey there"
};
```

Pointers

- A *pointer* is a variable that stores a memory address.
- Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
- One (8 byte) pointer can refer to any size memory location!
- Pointers are also essential for allocating memory on the heap, which we will cover later.
- Pointers also let us refer to memory generically, which we will cover later.

Looking Closely at C

- All parameters in C are "pass by value." For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.
- This means whenever we pass something as a parameter, we pass a copy.
- If we want to modify a parameter value in the function we call and have the changes persist afterwards, we can pass the location of the value instead of the value itself. This way we make a copy of the *address* instead of a copy of the *value*.

Pointers Summary

- If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**. This makes a copy of the data.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify. This makes a copy of the data's location.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

Strings In Memory

1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.
4. If we create a new string with new characters as a **`char *`**, we cannot modify its characters because its memory lives in the data segment.
5. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
6. Adding an offset to a C string gives us a substring that many places past the first character.
7. If we change characters in a string parameter, these changes will persist outside of the function.

Memory Locations

Q: Is there a way to check in code whether a string's characters are modifiable?

A: No. This is something you can only tell by looking at the code itself and how the string was created.

Q: So then if I am writing a string function that modifies a string, how can I tell if the string passed in is modifiable?

A: You can't! This is something you instead state as an assumption in your function documentation. If someone calls your function with a read-only string, it will crash, but that's not your function's fault :-)

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```
Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```
Segmentation fault (string literal)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```
Prints eulo3

4.
```
char *ptr = "Hello4";
char *str = ptr;
```
Segmentation fault (string literal)