Mutable Strings | Coursera

coursera.org/learn/pointers-arrays-recursion/supplement/NcaMa/mutable-strings

Mutable Strings

Mutable Strings

When we want to modify a string, we need the string to reside in writeable memory, such as the frame of a function or memory that is dynamically allocated by *malloc* (which you will learn about in Course 4). To make space for a string in a function's frame, we need to declare an array of chars with sufficient space to hold all of its characters, plus its null terminator.

One way we can declare and initialize our array of characters is like this:

```
1
char str[] = "Hello World\n";
```

This code behaves exactly as if we wrote:

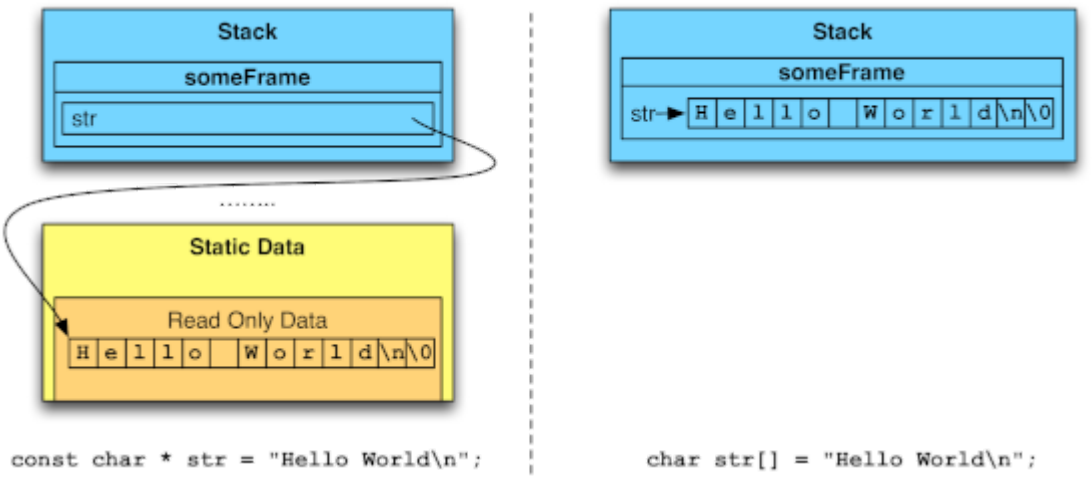
```
1
2
3
char str[] = {'H', 'e', 'l', 'l', 'o', ' ',
              'W', 'o', 'r', 'l'  'd', '\n', '\0'};
```

That is, it declares a variable str which is an array of 13 characters (remember that the size of an array may be implicit if we provide an initializer from which the compiler can determine the size), and initializes it by copying the characters of the string "Hello World\n"(including the null terminator) into that array. Being slightly more explicit, one could think of this code doing:

```
15

12

13

14

9

10

11

7

8

1

2

3

4

5

6


str[10] = 'd';

str[11] = '\n';

str[12] = '\0';

str[7]  = 'o';

str[8]  = 'r';

str[9]  = 'l';

str[5]  = ' ';

str[6]  = 'W';

char str[13];

str[0]  = 'H';

str[1]  = 'e';
```

```
str[2]  = 'l';

str[3]  = 'l';

str[4]  = 'o';
```

The figure below illustrates the difference between declaring *str* as **const char** * *str* versus **char** *str*[]. The figure shows the difference between a string declared as a pointer to a literal (left) and as an array, initialized by a literal (right).



We can declare the array *str* with an explicit size, but we must be careful—if we do not include enough space for the null terminator (*i.e.*, we declare it **char** *str*[*12*]="Hello World\n";), the compiler will not complain. Instead, it will initialize the character array exactly as we have requested, but there will be no '\0' placed at the end. The compiler allows this behavior since it makes for a perfectly valid array of characters, even though it is not a valid *string*. If we only compute on the array in such ways that it only accesses those 12 characters, our program is fine. However, if we use that array for anything (*e.g.*, pass it to *printf* or any of the string library functions that you will learn about soon) that expects an actual string (*i.e.*, one with a null terminator on the end), then the array will be accessed past its bounds.

Failing to terminate the string may not always appear in testing—you might “get lucky” and have the next byte of memory already be 0 anyways. While this may seem nice—your program “works”—it is actually quite a dangerous sort of problem. You may test your

program a thousand times and not see any errors, then deploy it and have it crash or produce incorrect results. We strongly recommend the use of tools such as *valgrind* which are capable of detecting this sort of error.

It is perfectly fine, however, to request *more* space than is required for your string. For example, **char** *str*[*100*] = "Hello World\n"; is entirely legitimate. We may wish to request extra space in this fashion if we plan to add to the string, making it longer. Of course, whenever we do so, we must be sure that we have enough space for whatever we may want our string to hold. (Remember that the programmer is responsible for keeping track of the size of her arrays. There is no way to inspect an array and derive its size.)