String Literals | Coursera

coursera.org/learn/pointers-arrays-recursion/supplement/QuCc7/string-literals

String Literals

String Literals

We have seen string literals so far—a sequence of characters written down in quotations marks, such as

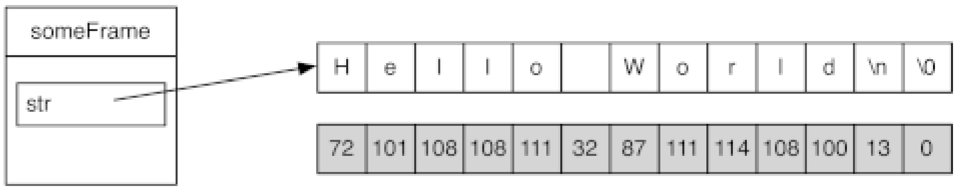
"Hello World\n".

Now, that we understand pointers, we can understand their type: **const char *,** that is, a pointer to characters which cannot be modified (recall that here, the **const** modifies the **chars** that are pointed-to).

That is, if we wanted to have a variable which points to a string literal, we might write:

```
const char * str = "Hello World\n";
```

This code declares and initializes variable *str*, so let us look at a diagram of its effect:

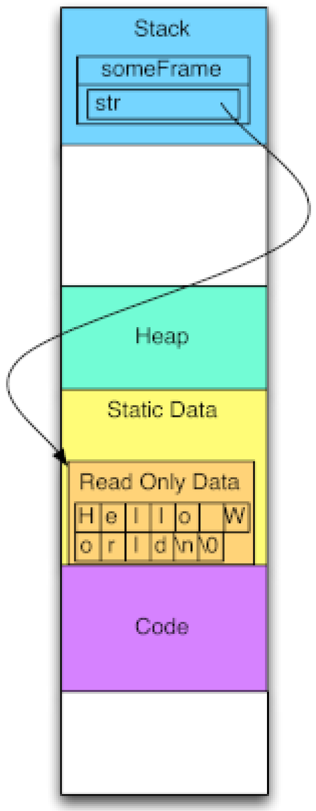


const char * str = "Hello World\n";

The figure above illustrates the effect of such a statement, and the layout of the string in memory. *str* is a pointer, pointing at an array of characters. These characters appear in the order of the string, and are followed by the null terminator character, **'\0'** (note that we do not need to write this character down in the string literal—the compiler adds it for us *for literals only*). Below the conceptual representation of the string, the figure shows its numeric representation—the string is just a sequence of bytes (8-bit numbers) in memory, the last of which has a numeric value of 0 (do not forget: *everything is a number!*).

Notice that we used **const** indicating that we cannot modify the characters pointed to by *str* (that is, assignment to *str[i]* will result in a compiler error). If we forget the **const** modifier, unfortunately, the code will still compile (we can receive a warning for this type of mistake with *-Wwrite-strings*, which is not enabled by default with *-Wall* because many programmers are sloppy about **const** anyways—which is not to say that you should be). However, if we omit **const** and try to modify the string, the program will crash with a *segmentation fault*.

The reason the program will crash if you attempt to modify a string literal is that the data for the string literal is stored into a *read only* portion of the static data section. The figure below shows the variable pointing at the string literal from the figure above in the context of the "picture of memory" you learned about earlier.



Note that string literals are typically placed in a read-only portion of the static data section, as seen above.

The data for the string literal (the actual bytes that make up the string) reside in the read only portion of the static data section for the entire lifetime of the program—from the time that it is loaded into memory until it exits. This data is placed into memory by the *loader*—the portion of the operating system which reads the executable file from the disk and initializes its memory appropriately. The loader knows what to write for the string literals (and where they should go in memory) because the compiler writes information into the executable file describing the contents of the data section. After the loader finishes initializing memory, it marks the read-only portions of the static data section as non-writeable in the *page table*—the structure that the operating system maintains to describe the program's memory to the hardware.

Attempting to write to a read-only portion of memory will behave much like writing to an invalid region of memory—it will cause the hardware to *trap* into the operating system—transferring execution control from your program to the OS kernel (conceptually, the hardware takes the execution arrow out of your program, and puts it into a particular function in the OS, noting where the execution arrow was in case the OS wants to return control to your program). The OS then sees that the program was attempting to access memory in invalid ways, and kills it with a segmentation fault.

If you want to know more about page tables, and how the OS handles memory, we encourage you to take a computer organization course after completing this specialization.

The compiler puts the string literals into a read only region of memory because the literal may get reused, and thus should not be changed. Consider the following code:

```
char * str1 = "Hello";
str1[0] = 'J';  // this would crash, but suppose it did not
…
…
char * str2 = "Hello";
printf("%s\n", str2);
```

Both occurrences of the literal "Hello" evaluate to pointers to the location where the characters of that string is stored. The compiler is free to put the two identical string literals in one location, meaning *str1* and *str2* could point at the same memory. If modifying this memory were allowed, printing *str2* would print "*Jello*", which would be confusing. In a worse case, modifying string literals could pose a wide range of issues, from strange behaviors to security problems. Note that even if the literal appears in only one place in the program, it may get re-used multiple times (inside a loop, in a function that is called more than once, *etc.*)—in such a case, our expectation as a programmer is that the literal will always be what we wrote, and it has not been changed by previous code.