



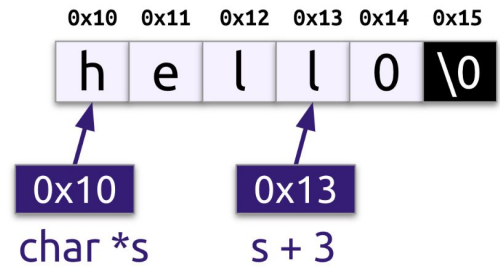
C Strings



A C string is an **array of characters** (`char []`) with a **null-terminating character** `'\0'` at the end. **It is vital that a string be properly null-terminated.**

- For common operations (e.g., comparing/concatenating/searching strings), we can use string library functions in `<string.h>`

- C strings can be represented as `char*`: **pointers to characters**, and we can move those pointers to advance through chars.



- When passing strings as parameters, we always pass them as `char*` (a pointer to the first char).** Thus, chars changed inside a function will persist outside of it.

In summary, we usually find it more convenient and powerful to use a `char*` / string pointer to access, share, and manipulate the data inside an array of characters.

Stack vs Read-Only Data Segment



- The characters of `myArr` live on the stack → **modifiable** .
- The characters of `myPtr` live on read-only data segment → **unmodifiable** .

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char myArr[6] = "Hello";
    char *myPtr = "Hello";

    myArr[0] = 'J';
    printf("%s\n", myArr); // Prints "Jello"

    myPtr[0] = 'J'; // Segmentation fault!!
    printf("%s\n", myPtr);
    return 0;
}
```



Stack

J e l l o \0
Array of chars!

Data Segment

H e l l o \0
String literal

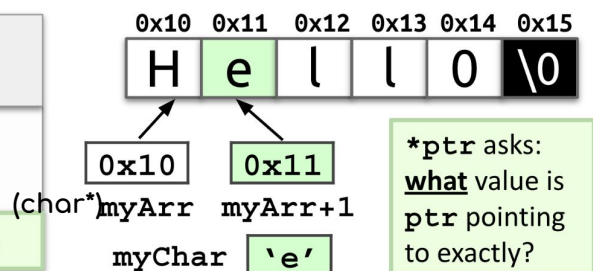
Dereferencing *pointer

Referencing &variable



Exercise! Rewrite the second line using pointer arithmetic

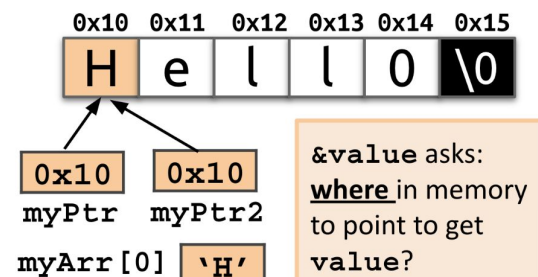
```
char myArr[6] = "Hello";
char myChar = myArr[1]; // e
char myChar = *(myArr + 1);
```



Exercise! What's the difference?

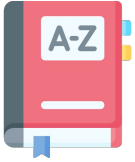
```
char myArr[6] = "Hello";
char* myPtr = myArr;
char* myPtr2 = &(myArr[0]);
```

Answer: None! They both store a pointer to first element of `myArr`!





Key String Functions

**int strcmp(str1, str2)**Returns **0** if str1 and str2 are identical
Returns **negative** if str1 comes first in the dictionary, **positive** if str2 comes first

returns

str1	black	str2	pink	returns	<0
str1	pink	str2	pink	returns	0

size_t strlen(str)Returns length of string, **not** including the null-terminating `\0`

returns

str	n y a n c a t \0	returns	7
-----	------------------	---------	---

char *strncpy(dest, src, n)Copies **n bytes** from source into destination (1 char = 1 byte), but doesn't guarantee result will be null-terminated unless the `\0` is copied over

src n y a n c a t \0 n 4

returns char* → dest n y a n

More String Functions!

**char *strstr(str1, str2)**Returns pointer to **first occurrence** of str2 in str1, or **NULL** if not found.

str1 a b e n t w o b e n s \0

Returns char* → b e n \0

size_t strspn(str1, str2)Returns **length** of initial portion of str1 that only contains characters from str2

str1 = + = f l o m o = l a g \0

Returns 3

str2 = + \0

size_t strcspn(str1, str2)Returns **length** of initial portion of str1 that **doesn't** contain characters from str2

str1 y a g a = + = = g a t e s \0

Returns 4

str2 = + \0

strtok() library function

**char *musl_strtok(char *s, const char *delimiters)**

- Splits **string s** into substrings or "tokens" that are separated by delimiters
- Returns **one token per call** to strtok
- Maintains state across calls -- **calls are not independent!**

str: "web.stanford.edu/class/archive/cs/cs107"

delimiters: "/"

The tokens would be:

"web.stanford.edu" "class" "archive" "cs" "cs107"

To do this, instead of creating new strings, strtok chops the existing string by overriding **delimiters** with **null-terminators** when needed, e.g., after everything,

... c l a s s / a r c h i v e / c s / ...

... c l a s s \0 a r c h i v e \0 c s \0 ...



You'll write a better version in Assign2 :)

