

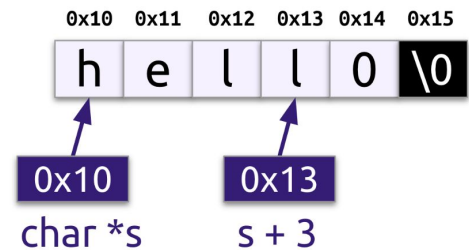


C Strings



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.

- Characters of strings live either on the **stack** or the **data segment (read-only)**
- Stack strings**, e.g., `char str[6] = "Hello";` are modifiable:
 - `str[0] = 'J';` would successfully change the string to "Jello" 
- Read-only string literals**, e.g., `char* str = "Hello";` are **NOT** modifiable
 - `str[0] = 'J';` would result in a segmentation fault / crashing 

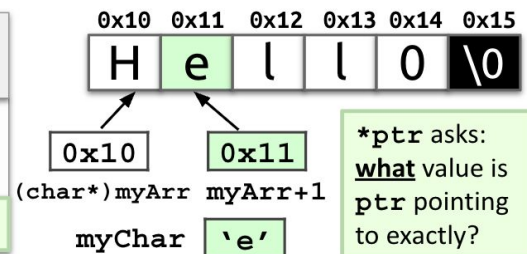
Dereferencing
*pointer

Referencing
&variable



 **Exercise!** Rewrite the second line using pointer arithmetic

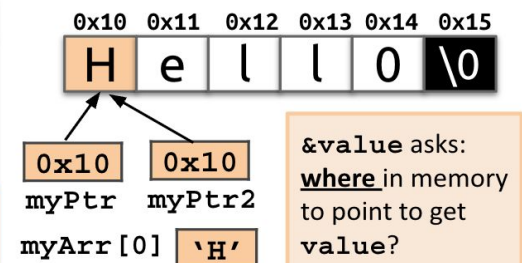
```
char myArr[6] = "Hello";
char myChar = myArr[1]; // e
char myChar = *(myArr + 1);
```



 **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!

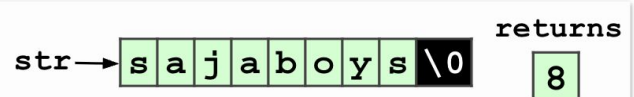


Hot 100 String Library Functions



`size_t strlen(str)`

Returns length of string, not including the null-terminating `\0`



`int strcmp(str1, str2)`

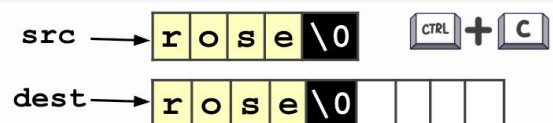
Returns 0 if str1 and str2 are identical
Returns **negative** if str1 comes first in the dictionary, **positive** if str2 comes first.



Note: The variant function `strncmp(str1, str2, n)` will compare only up to `n` chars!

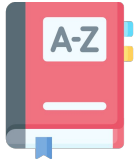
`char* strcpy(dest, src)`

Copies the characters of a source string (including `\0`) to a destination buffer.





More String Functions!



char* strncpy(dest, src, n)

Copies up to **n** bytes from source into destination.
No guarantee result will be null-terminated unless the **\0** is copied over (which also ends the copying)

src → r o s e b u d \0
dest → r o s e n 4

char* strcat(dest, src)

Ben's favorite function, because cats .
Appends a copy of destination string to source, overriding the source's **\0** char, and tacking on a new **\0** at the end.

src → b e a r \0
dest → p o o h \0
dest updated! → p o o h b e a r \0

char* strchr(str, ch)

Returns pointer to first occurrence of character **ch** in **str**, NULL if not found

str1 → b b y @ s t e d u \0
Returns char* ch @

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* str2 → 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 → l l a g a = + = c o d a \0
Returns 5 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 ...

