CS107, Lecture 7
C Strings

Reading: K&R (1.9, 5.5, Appendix B3) or Essential C section 3

Ed Discussion: https://edstem.org/us/courses/46162/discussion/3592722

Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***`str1, str2`***`)`, `strncmp(`***`str1, str2, n`***`)`	compares two strings; returns 0 if identical, <0 if ***`str1`*** comes before ***`str2`*** in alphabet, >0 if ***`str1`*** comes after ***`str2`*** in alphabet. ***`strncmp`*** stops comparing after at most ***`n`*** characters.

`strchr(`***`str, ch`***`)`
`strrchr(`***`str, ch`***`)`
	character search: returns a pointer to the first occurrence of ***`ch`*** in ***`str`***, or ***`NULL`*** if ***`ch`*** was not found in ***`str`***. `strrchr` find the last occurrence.

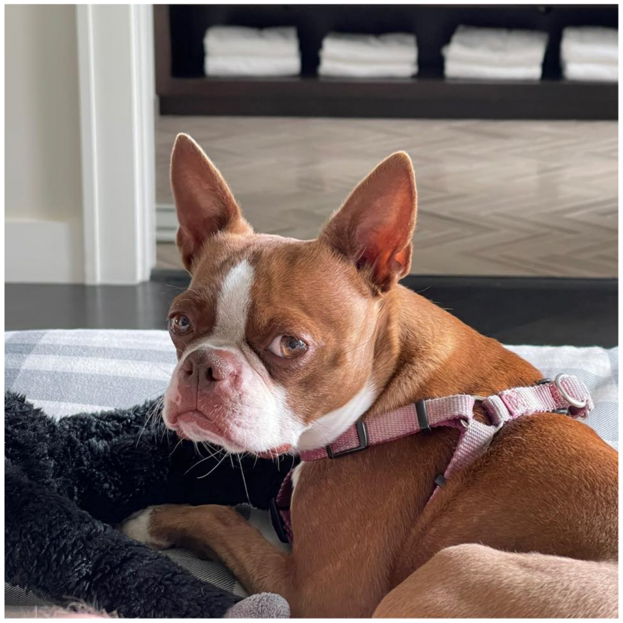
`strstr(`***`haystack, needle`***`)`	string search: returns a pointer to the start of the first occurrence of ***`needle`*** in ***`haystack`***, or ***`NULL`*** if ***`needle`*** was not found in ***`haystack`***.
`strcpy(`***`dst, src`***`)`, `strncpy(`***`dst, src, n`***`)`	copies characters in ***`src`*** to ***`dst`***, including null-terminating character. Assumes enough space in ***`dst`***. Strings must not overlap. **`strncpy`** stops after at most ***`n`*** chars, and does not add null-terminating char.
`strcat(`***`dst, src`***`)`, `strncat(`***`dst, src, n`***`)`	concatenate ***`src`*** onto the end of ***`dst`***. **`strncat`** stops concatenating after at most ***`n`*** characters. Always adds a null-terminating character.
`strspn(`***`str, accept`***`)`, `strcspn(`***`str, reject`***`)`	**`strspn`** returns the length of the initial part of ***`str`*** which contains only characters in ***`accept`***. **`strcspn`** returns the length of the initial part of ***`str`*** which does not contain any characters in ***`reject`***.

String Diamond

Write a function **`diamond`** that accepts a string parameter and prints its letters in a "diamond" format as shown below.

- For example, `diamond("doris")` should print:

```
d
do
dor
dori
doris
 oris
  ris
   is
    s
```



Practice: String Diamond

`string_diamond.c`

Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```
char laureate[15];
strcpy(laureate, "Katalin Kariko");
char *first = strchr(laureate, 'a');
char *last = strrchr(laureate, 'a');
printf("%s\n", laureate);  // Katalin Kariko
printf("%s\n", first);     // atalin Kariko
printf("%s\n", last);      // ariko
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the *first* one. Use `str`**r**`chr` to obtain a pointer to the *last* occurrence.

Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```
char laureate[17];
strcpy(laureate, "Carolyn Bertozzi");
char *zz = strstr(laureate, "zz");
printf("%s\n", laureate);        // Carolyn Bertozzi
printf("%s\n", zz);              // zzi
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char laureate[17];
strcpy(laureate, "Barry Sharpless");
int length = strspn(laureate + 1, "road"); // 3
```

"How many places can we go in the first string before I encounter a character <u>not in</u> the second string?"

String Spans

`strcspn` (c = "complement") returns the *length* of the initial part of the first string which contains only characters not in the second string.

```
char laureate[17];
strcpy(laureate, "Barry Sharpless");
int length = strcspn(laureate + 2, "abcde");    // 6
```

"How many places can we go in the first string before I encounter a character in the second string?"

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char ***. We can still operate on the string the same way as with a char[].

```
int foo(char *str) {
     char ch = str[1];
     ...
}

// can also write this, but it is really a pointer
int foo(char str[]) { ...
```

Arrays of Strings

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

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

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

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

Arrays of Strings

We can access each string using bracket syntax:

```
printf("%s\n", array[0]);    // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the array's first element*. In fact, you're already seen this with **main**'s **argv** parameter! This means we write the parameter type as:

```
void func(char **array) {

// equivalent to this, but it is really a double pointer
void func(char *array[]) {
```

Practice: Password Verification

Write a function **`verifyPassword`** that accepts a candidate password and certain password criteria and returns whether the password is valid.

```
bool verifyPassword(char *password, char *validChars,
                    char *badSubstrings[], size_t count);
```

password is valid if it contains only letters in **validChars** and does not contain any substrings in **badSubstrings**.

Practice: Password Verification

```
bool verifyPassword(char *password, char *validChars,
                    char *badSubstrings[], size_t count);
```

Example:

```
char *invalidSubstrings[] = {"1234" , "4132"};

bool valid1 = verifyPassword("1572", "0123456789",
              invalidSubstrings, 2);  // true
bool valid2 = verifyPassword("141234", "0123456789",
              invalidSubstrings, 2);  // false
```

Practice: Password Verification

`verify_password.c`

Recall: Buffer Overflows

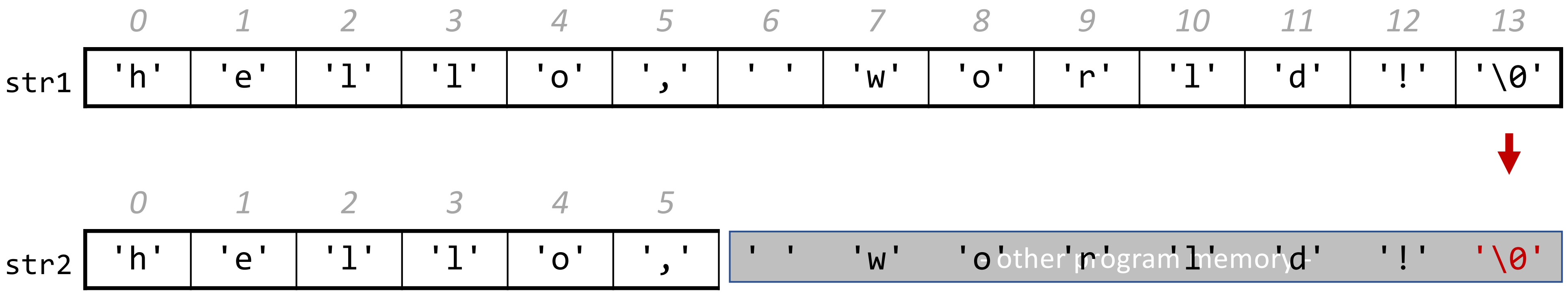
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
char str2[6];                      // not enough space!
strcpy(str2, "hello, world!");    // overwrites other memory!
```

Writing past memory bounds is called a "buffer overflow". It can allow for security vulnerabilities!

Recall: Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```


0
1
2
3
4
5
6
7
8
9
10
11
12
13
str1
'h'
'e'
'l'
'l'
'o'
','
' '
'w'
'o'
'r'
'l'
'd'
'!'
'\0'
str2
- other program memory -