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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strlen(str)</code></td>
<td>returns the # of chars in a C string (before null-terminating character).</td>
</tr>
<tr>
<td><code>strcmp(str1, str2)</code></td>
<td>compares two strings; returns 0 if identical, &lt;0 if <code>str1</code> comes before <code>str2</code> in alphabet, &gt;0 if <code>str1</code> comes after <code>str2</code> in alphabet. <code>strn cmp</code> stops comparing after at most n characters.</td>
</tr>
<tr>
<td><code>strncmp(str1, str2, n)</code></td>
<td>character search: returns a pointer to the first occurrence of <code>ch</code> in <code>str</code>, or <code>NULL</code> if <code>ch</code> was not found in <code>str</code>. <code>strncmp</code> find the last occurrence.</td>
</tr>
<tr>
<td><code>strchr(str, ch)</code></td>
<td>returns a pointer to the start of the first occurrence of <code>haystack</code> in <code>haystack</code>, or <code>NULL</code> if <code>haystack</code> was not found in <code>haystack</code>.</td>
</tr>
<tr>
<td><code>strncpy(dst, src, n)</code></td>
<td>copies characters in <code>src</code> to <code>dst</code>, including null-terminating character. Assumes enough space in <code>dst</code>. Strings must not overlap. <code>strncpy</code> stops after at most n chars, and does not add null-terminating char.</td>
</tr>
<tr>
<td><code>strcat(dst, src)</code></td>
<td>concatenate <code>src</code> onto the end of <code>dst</code>. <code>strncat</code> stops concatenating after at most n characters. Always adds a null-terminating character.</td>
</tr>
<tr>
<td><code>strspn(str, accept)</code></td>
<td><code>strspn</code> returns the length of the initial part of <code>str</code> which contains only characters in <code>accept</code>. <code>strcspn</code> returns the length of the initial part of <code>str</code> which does not contain any characters in <code>reject</code>.</td>
</tr>
</tbody>
</table>
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.

```c
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 strrchr 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.

```c
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.

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

```c
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[]`.

```c
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:

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

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

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

We can access each string using bracket syntax:

```c
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:

```c
void func(char **array) {
    // equivalent to this, but it is really a double pointer
    void func(char *array[]) {
```
Write a function `verifyPassword` that accepts a candidate password and certain password criteria and returns whether the password is valid.

```c
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*. 
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*.

```c
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

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

![Buffer Overflow Diagram]

- `str1`: "hello, world!", size 14
- `str2`: Not enough space, overwrites other memory.