CS107, Lecture 6
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/3538920
CS107 Topic 2: How can a computer represent and manipulate more complex data like text?
How can a computer represent and manipulate more complex data like text?

Why is answering this question important?
• Shows us how strings are represented in C and other languages (this time)
• Helps us better understand buffer overflows, a common bug (next time)
• Reintroduces us to pointers, because strings can be pointers (next Wednesday)
A `char` is a variable type that represents a single character or "glyph".

```cpp
char letter = 'A';
char plus = '+';
char zero = '0';
char space = ' ';
char newline = '\n';
char tab = '\t';
char single_quote = '\'';
char backslash = '\\';
```
Under the hood, C represents each `char` as an `integer` (its "ASCII value").

- Uppercase letters are sequentially numbered
- Lowercase letters are sequentially numbered
- Digits are sequentially numbered
- Lowercase letters are 32 more than their uppercase equivalents (bit flip!)

```c
char upper = 'A'; // Actually 65
char lower = 'a'; // Actually 97
char zero = '0'; // Actually 48
```
# Common `ctype.h` Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isalpha(ch)</code></td>
<td>true if <code>ch</code> is 'a' through 'z' or 'A' through 'Z'</td>
</tr>
<tr>
<td><code>islower(ch)</code></td>
<td>true if <code>ch</code> is 'a' through 'z'</td>
</tr>
<tr>
<td><code>isupper(ch)</code></td>
<td>true if <code>ch</code> is 'A' through 'Z'</td>
</tr>
<tr>
<td><code>isspace(ch)</code></td>
<td>true if <code>ch</code> is a space, tab, new line, etc.</td>
</tr>
<tr>
<td><code>isdigit(ch)</code></td>
<td>true if <code>ch</code> is '0' through '9'</td>
</tr>
<tr>
<td><code>toupper(ch)</code></td>
<td>returns uppercase equivalent of a letter</td>
</tr>
<tr>
<td><code>tolower(ch)</code></td>
<td>returns lowercase equivalent of a letter</td>
</tr>
</tbody>
</table>

Remember: these **return** a char; they cannot modify an existing char!
More documentation with `man isalpha`, `man tolower`
bool isLetter = isalpha('A');   // true
bool capital = isupper('f');    // false
char uppercaseB = toupper('b');
bool isADigit = isdigit('4');   // true
C has no dedicated variable type for strings. Instead, a string is represented as an array of characters with a sentinel value marking its end.

"Hello"

\[
\begin{array}{c|c|c|c|c|c|c}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 \\
\text{char} & 'H' & 'e' & 'l' & 'l' & 'o' & '\0' \\
\end{array}
\]

'\0' is the null-terminating character, and you always need one extra space in an array for it.
C strings are not objects. (In fact, nothing in C is an object.) If we want to compute the length of the string, we must calculate it ourselves.

We call the built-in `strlen` function to calculate string length. The null-terminating character doesn’t contribute to a C string’s length.

```
int length = strlen(myStr);       // e.g., 13
```

**Caution:** `strlen` is $O(N)$ because it must scan the entire string! We should save the value if we plan to refer to the length later.
When we pass a string as a parameter, it is passed as a `char *`. C passes the location of the first character rather than a copy of the whole array.

```c
int foo(char *str) {
    ...
}

char string[6];
...
foo(string); // equivalently foo(&string[0])
```
When we pass a string as a parameter, it is passed as a `char *`. C passes the location of the first character rather than a copy of the whole array.

```c
int foo(char *str) {
    ...
    str[0] = 'c';       // modifies original string!
    printf("%s\n", str);  // prints cello
}
```

We still use a `char *` the same way we use a `char[]`.

```c
char string[6];
...  // code to build string to be "Hello"
foo(string);
```
## 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>strncmp</code> stops comparing after at most <code>n</code> characters.</td>
</tr>
<tr>
<td><code>strchr(str, ch)</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>strrchr</code> find the last occurrence.</td>
</tr>
<tr>
<td><code>strstr(haystack, needle)</code></td>
<td>string search: returns a pointer to the start of the first occurrence of <code>needle</code> in <code>haystack</code>, or <code>NULL</code> if <code>needle</code> was not found in <code>haystack</code>.</td>
</tr>
<tr>
<td><code>strcpy(dst, src)</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 <code>n</code> chars, and <strong>does not</strong> 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 <code>n</code> characters. <strong>Always</strong> adds a null-terminating character.</td>
</tr>
<tr>
<td><code>strspn(str, accept)</code></td>
<td><strong>strspn</strong> returns the length of the initial part of <code>str</code> which contains only characters in <code>accept</code>. <strong>strcspn</strong> returns the length of the initial part of <code>str</code> which does <strong>not</strong> contain any characters in <code>reject</code>.</td>
</tr>
</tbody>
</table>
The string library: `strcmp`

`strcmp(str1, str2)` compares two strings (note: `==`, `<`, etc. don’t work)

- returns 0 if both strings are identical
- `< 0` if `str1` is lexicographically smaller than `str2`
- `> 0` if `str1` is lexicographically larger than `str2`

```c
int cmp = strcmp(str1, str2);
if (cmp == 0) {
    // equal
} else if (cmp < 0) {
    // str1 comes before str2
} else {
    // str1 comes after str2
}
```
The string library: `strcpy`

`strcpy(dst, src)`: copies the contents of `src` into the string `dst`, including the null terminator. *(Note that you can’t copy a C string using =.)*

```c
char str1[6]; // include space for '\0'
strcpy(str1, "hello");

char str2[6];
strcpy(str2, str1);
str2[0] = 'c';

printf("%s", str1); // hello
printf("%s", str2); // cello
```
char str1[6];
strcpy(str1, "hello");

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

```
\begin{array}{cccccccccccccc}
\text{str1} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
'h' & 'e' & 'l' & 'l' & 'o' & ',' & '' & 'w' & 'o' & 'r' & 'l' & 'd' & '!' & '\0'
\end{array}
```

```
\begin{array}{ccccccc}
\text{str2} & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\end{array}
```

- other program memory -
Copying Strings – Buffer Overflows

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>str1</td>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>','</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'\0'</td>
</tr>
<tr>
<td>str2</td>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

str2  0  1  2  3  4  5
  \h\' 'e' 'l' 'l' 'o' ','
```

- other program memory -

 максимально
Copying Strings – Buffer Overflows

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

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>
| 'h' | 'e' | 'l' | 'l' | 'o' | ',' | ' ' | 'w' | 'o' | 'r' | 'l' | 'd' | '!' | '\0'

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>','</td>
</tr>
</tbody>
</table>
```

Copying the string "hello, world!" into `str1` and then copying `str1` into `str2` results in an overflow, as `str2` does not have enough space to hold the entire string. The extra characters are placed in other memory locations, as indicated by the red arrow and sad face.
**Copying Strings - strncpy**

`strncpy(dst, src, n)`: copies at most the first `n` bytes from `src` into the string `dst`. If there is no null-terminating character in these bytes, then `dst` won’t get a null terminator either.

```c
// copying "automata"
char str[8];
strncpy(str, "automata", 8);  // doesn’t write a '\0'!
```

When we fail to terminate a character array with a '\0' but treat it as a C string anyway, we can’t expect C string functions to work properly, e.g., `strlen` may continue reading beyond the bounds of `str` in search of '\0'!
String Copying Exercise

What value should go in the blank at right?

A. 4
B. 5
C. 6
D. 12
E. strlen("hello")
F. Something else

```c
char str[______];
strcpy(str, "hello");
```
What is printed out by the following program

```c
int main(int argc, char *argv[]) {
    char str[9];
    strcpy(str, "Hi earth");
    str[2] = '\0';
    printf("str = %s, len = %zu\n", str, strlen(str));
    return 0;
}
```

A. str = Hi, len = 8
B. str = Hi, len = 2
C. str = Hi earth, len = 8
D. str = Hi earth, len = 2
E. None/other
The string library: \texttt{str(n)cat}

\texttt{strcat(dst, src)}: concatenates the contents of \texttt{src} into the string \texttt{dst}.

\texttt{strncat(dst, src, n)}: same, but concats at most \texttt{n} bytes from \texttt{src}.

\begin{verbatim}
char str1[13];         // enough space for strings + '\0'
strncpy(str1, "hello ");
strcat(str1, "world!");  // removes old '\0', adds new '\0' at end
printf("%s", str1);      // hello world!
\end{verbatim}

Both \texttt{strcat} and \texttt{strncat} remove the old '\0' and add a new one at the end. (Note that we can’t concatenate C strings using + as we can in C++ or Python.)
```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>str1</td>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>' '</td>
<td>'\0'</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>str2</td>
<td>'w'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</td>
<td>'!'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
Concatenating Strings

```c
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

<table>
<thead>
<tr>
<th>str1</th>
<th>str2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'h'</td>
<td>'w'</td>
</tr>
<tr>
<td>'e'</td>
<td>'o'</td>
</tr>
<tr>
<td>'l'</td>
<td>'r'</td>
</tr>
<tr>
<td>'l'</td>
<td>'l'</td>
</tr>
<tr>
<td>'o'</td>
<td>'d'</td>
</tr>
<tr>
<td>' '</td>
<td>'!'</td>
</tr>
<tr>
<td>'w'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
Concatenating Strings

```c
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>h</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td></td>
<td>w</td>
<td>o</td>
<td>r</td>
<td>l</td>
<td>d</td>
<td>!</td>
<td>\0</td>
</tr>
</tbody>
</table>

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| w | o | r | l | d |! | \0 |
You can also create a char * variable yourself that points to an address within in an existing string.

```c
char str[3];
str[0] = 'H';
str[1] = 'i';
str[2] = '\0';

char *alias = str; // points to 'H'
```
Since C strings are pointers to characters, we can adjust the pointer to overlook characters at the beginning.

// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;

```
chars: 0xf1 0xf2 0xf3 0xf4 0xf5 0xf6 0xf7 0xf8
       | 'r'  'a'  'c'  'e'  'c'  'a'  'r'  '\0'
str1: 0xee 0xf1
str2: 0xd2 0xf5
```
Substrings

Since C strings are pointers to characters, we can adjust the pointer to overlook characters at the beginning.

```c
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
printf "%s\n", str1); // racecar
printf "%s\n", str2); // car
```

```
chars     0xf1 0xf2 0xf3 0xf4 0xf5 0xf6 0xf7 0xf8
    'r'  'a'  'c'  'e'  'c'  'a'  'r'  '\0'

str1     0x94 0xe2 0xd2 0xf5
str2     0x94 0xe2 0xf5
```

---

30
```c
char str[9];
strcpy(str, "potatoes");
char *word = str + 2;
strcpy(word, "mat");
printf("%s
", str);
```

What is printed?

A. matoes  
B. mattoes  
C. pomat  
D. potatoes  
E. pomitoes  
F. pomidoes
char str[9];
strcpy(str, "potatoes");
char *word = str + 2;
strncpy(word, "mid", 2);
printf("%s\n", str);

What is printed?
A. matoes  D. potatoes
B. mattoes  E. pomitoes
C. pomat    F. pomidoes

buf: 'P' 'o' 't' 'a' 't' 'o' 'e' 's' '\0'
word: 'm' 'a' 't' 'o' 'e' 's' 't'

0xe0 0xe1 0xe2 0xe3 0xe4 0xe5 0xe6 0xe7 0xe8
0xf0

buf: 'P' 'o' 't' 'a' 't' 'o' 'e' 's' '\0'
word: 'm' 'a' 't' 'o' 'e' 's' 't'

0xe0 0xe1 0xe2 0xe3 0xe4 0xe5 0xe6 0xe7 0xe8
0xf0