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/28214/discussion/1877377
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".

```java
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`
Common `ctype.h` Functions

```c
bool isLetter = isalpha('A');  // true
bool capital = isupper('f');  // false
char uppercaseB = toupper('b');
bool isADigit = isdigit('4');  // true
```
C Strings

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"

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>'H'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

'\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 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.

```c
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.
C Strings As Parameters

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(&str[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>strncpy</code> stops comparing after at most <code>n</code> characters.</td>
</tr>
<tr>
<td><code>strncpy(str1, str2, 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 <code>n</code> 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>strcat</code> stops concatenating after at most <code>n</code> 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>
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
```
Copying Strings - \texttt{strcpy}

\begin{verbatim}
char str1[6];
strcpy(str1, "hello");

char str2[6];
strcpy(str2, str1);
\end{verbatim}
We must make sure there is enough space in the destination to hold the entire copy, \textit{including the null-terminating character.}

\begin{verbatim}
    char str2[6];               // not enough space!
    strcpy(str2, "hello, world!");  // overwrites other memory!
\end{verbatim}

Writing past memory bounds is called a "buffer overflow".
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>str1</th>
<th>str2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'h'</td>
<td>?</td>
</tr>
<tr>
<td>'e'</td>
<td>?</td>
</tr>
<tr>
<td>'l'</td>
<td>?</td>
</tr>
<tr>
<td>'l'</td>
<td>?</td>
</tr>
<tr>
<td>'o'</td>
<td>?</td>
</tr>
<tr>
<td>','</td>
<td>?</td>
</tr>
<tr>
<td>' '</td>
<td>?</td>
</tr>
<tr>
<td>'w'</td>
<td>?</td>
</tr>
<tr>
<td>'o'</td>
<td>?</td>
</tr>
<tr>
<td>'r'</td>
<td>?</td>
</tr>
<tr>
<td>'l'</td>
<td>?</td>
</tr>
<tr>
<td>'d'</td>
<td>?</td>
</tr>
<tr>
<td>'!'</td>
<td>?</td>
</tr>
<tr>
<td>'\0'</td>
<td>?</td>
</tr>
</tbody>
</table>

- 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!
```

![Memory Diagram](attachment://memory_diagram.png)
Copying Strings – Buffer Overflows

```c
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 -
Copying Strings – Buffer Overflows

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

<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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td>,</td>
<td>'</td>
<td>w</td>
<td>o</td>
<td>r</td>
<td>l</td>
<td>d</td>
<td>!</td>
</tr>
</tbody>
</table>

<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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td>,</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>

<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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td>,</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>

😢
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 problem"
char str[7];
strncpy(str, "problem", 7); // 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'!
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: strcat**

`strcat(dst, src)`: concatenates the contents of `src` into the string `dst`.

`strncat(dst, src, n)`: same, but concats at most `n` bytes from `src`.

```c
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!
```

Both `strcat` and `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.)
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>''</td>
<td></td>
</tr>
</tbody>
</table>

The character array `str1` is initialized with the string "hello ". Then it is concatenated with `str2`, which contains "world!", using `strcat()`. The resulting string is then displayed in a table format showing the memory addresses and characters stored at each position.
Concatenating Strings

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

strcat(str1, str2);
Concatenating Strings

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

strcat(str1, str2);
```

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

str2 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 an existing string.

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

char *alias = str; // points to 'H'
```
Substrings

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

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

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

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

```
\begin{tabular}{cccccccc}
  chars & 0xf1 & 0xf2 & 0xf3 & 0xf4 & 0xf5 & 0xf6 & 0xf7 & 0xf8 \\
  \text{\textquotesingle r\textquotesingle} & \text{\textquotesingle a\textquotesingle} & \text{\textquotesingle c\textquotesingle} & \text{\textquotesingle e\textquotesingle} & \text{\textquotesingle c\textquotesingle} & \text{\textquotesingle a\textquotesingle} & \text{\textquotesingle r\textquotesingle} & \text{"\textbackslash 0\textquotescpace"} \\
\end{tabular}
```

```
\begin{tabular}{ccc}
  str1 & 0xee & str2 \\
  \text{\textquotesingle \textbackslash e\textquotesingle} & 0xdf & 0xdf5 \\
\end{tabular}
```
```c
char str[9];
strcpy(str, "potatoes");
char *word = str + 2;
strcpy(word, "mat");
printf("%s\n", str);
```

What is printed?

A. matoes  
B. mattoes  
C. pomat  
D. potatoes  
E. pomitoes  
F. pomidoes

```
buf  0xe0 0xe1 0xe2 0xe3 0xe4 0xe5 0xe6 0xe7 0xe8
word  0xf0
```

The string `potatoes` is copied to `str`. Then `word` is set to `str + 2`, which points to the character 't' in the string. The string 'mat' is copied to `word`. Finally, `printf` prints the entire string `str`. The correct answer is C. pomat.
char str[9];
strcpy(str, "potatoes");
char *word = str + 2;
strncpy(word, "mid", 2);
printf("%s\n", str);

What is printed?
A. matoes  D. tomatoes
B. mattoes  E. pimotoes
C. pomat    F. pomidoes