CS107, Lecture 6

C Strings

Reading: K&R (1.9, 5.5, Appendix B3) or Essential C section 3
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 (this time)
- Introduces us to pointers, because strings can be pointers (next time)

**assign2:** implement 2 functions a 1 program using those functions to find the location of different built-in commands in the filesystem. You’ll write functions to extract a list of possible locations and tokenize that list of locations.
Learning Goals

• Learn how strings are represented in C; as an array of null-terminated characters.
• Understand how to use the built-in string functions for common string tasks
• Learn about buffer overflow and what might cause it
Goal: 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("BAILEY")` should print:

  B
  BA
  BAI
  BAIL
  BAILE
  BAILEY
  AILEY
  ILEY
  LEY
  EY
  Y
• Characters
• Strings
• Common String Operations
  • Comparing
  • Copying
  • Concatenating
  • Substrings
Lecture Plan

• Characters
• Strings
• Common String Operations
  • Comparing
  • Copying
  • Concatenating
  • Substrings
A **char** is a variable type that represents a single character or “glyph”.

```c
char letterA = 'A';
char plus = '+';
char zero = '0';
char space = ' ';
char newLine = '\n';
char tab = '\t';
char singleQuote = '\'';
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 uppercaseA = 'A';    // Actually 65
char lowercaseA = 'a';     // Actually 97
char zeroDigit = '0';      // Actually 48
```
We can take advantage of C representing each char as an integer:

```c
bool areEqual = 'A' == 'A';    // true
bool earlierLetter = 'f' < 'c'; // false
char uppercaseB = 'A' + 1;
int diff = 'c' - 'a';          // 2
int numLettersInAlphabet = 'z' - 'a' + 1;    // or
int numLettersInAlphabet = 'Z' - 'A' + 1;
```
We can take advantage of C representing each char as an integer:

```c
// prints out every lowercase character
for (char ch = 'a'; ch <= 'z'; ch++) {
    printf("%c", ch);
}
```
### 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
Lecture Plan

• Characters
• Strings
• Common String Operations
  • Comparing
  • Copying
  • Concatenating
  • Substrings
C Strings

C has no dedicated variable type for strings. Instead, a string is represented as an array of characters with a special ending sentinel value.

"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; you always need to allocate one extra space in an array for it. '\0' is the character with numerical value 0.
C Strings

char myString[6];
myString[0] = 'H';
myString[1] = 'e';
myString[2] = 'l';
...
myString[5] = '\0';
Strings are **not** objects. They do not embed additional information (e.g., string length). We must calculate this!

<table>
<thead>
<tr>
<th>index</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>value</td>
<td>'H'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>','</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>

We can use the provided `strlen` function to calculate string length. The null-terminating character does *not* count towards the 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.
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 doSomething(char *str) {
    ...
}
```

```c
cchar myString[6];
...
doSOMething(myString);
```
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 doSomething(char *str) {
    ...
    str[0] = 'c'; // modifies original string!
    printf("%s\n", str); // prints cello
}
```

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

```c
char myString[6];
...
// e.g. this string is "Hello"
doSomething(myString);
```
Lecture Plan

• Characters
• Strings

• Common String Operations
  • Comparing
  • Copying
  • Concatenating
  • Substrings
### 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>, <code>strncmp(str1, str2, n)</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>strrchr(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>, <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 <code>n</code> chars, and does not add null-terminating char.</td>
</tr>
<tr>
<td><code>strcat(dst, src)</code>, <code>strncat(dst, src, n)</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. Always adds a null-terminating character.</td>
</tr>
<tr>
<td><code>strspn(str, accept)</code>, <code>strcspn(str, reject)</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>
# 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>strnstrmp</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. Assumess 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>strncat</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>

Many string functions assume **valid string input**, i.e., ends in a null terminator.
Comparing Strings

We cannot compare C strings using comparison operators like ==, < or >. This compares addresses!

// e.g. str1 = 0x7f42, str2 = 0x654d
void doSomething(char *str1, char *str2) {
    if (str1 > str2) { ... // compares 0x7f42 > 0x654d!
Instead, use strcmp.
The string library: `strcmp`

`strcmp(str1, str2)`: compares two strings.

- returns 0 if identical
- <0 if `str1` comes before `str2` in alphabet
- >0 if `str1` comes after `str2` in alphabet.

```c
int compResult = strcmp(str1, str2);
if (compResult == 0) {
    // equal
} else if (compResult < 0) {
    // str1 comes before str2
} else {
    // str1 comes after str2
}
```
We cannot copy C strings using =. This copies addresses!

// e.g. param1 = 0x7f42, param2 = 0x654d
void doSomething(char *param1, char *param2) {
    param1 = param2; // copies 0x654d. Points to same string!
    param2[0] = 'H'; // modifies the one original string!
}

Instead, use `strcpy`. 
**The string library: strcpy**

`strcpy(dst, src)`: copies the contents of `src` into the string `dst`, including the null terminator.

```c
char str1[6];
strcpy(str1, "hello");

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

printf("%s", str1);  // hello
printf("%s", str2);  // cello
```
Copying Strings - strcpy

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!
Copying Strings – Buffer Overflows

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

![Diagram showing buffer overflows with characters and indices]
Copying Strings – Buffer Overflows

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

![Diagram showing buffer overflow](https://via.placeholder.com/150)
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1); // not enough space - overwrites other 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!
```
Copying Strings – Buffer Overflows

```c
#include <stdio.h>

int main()
{
    char str1[14];
    strcpy(str1, "hello, world!");
    char str2[6];
    strcpy(str2, str1); // not enough space - overwrites other memory!
    return 0;
}
```

![Diagram of char arrays str1 and str2 showing buffer overflow](image-url)
Copying Strings – Buffer Overflows

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

---

```
char str1[14];
strncpy(str1, "hello, world!", 13);
char str2[6];
strncpy(str2, str1, 6); // ensures str2 does not overwrite other program memory
```

---

In the example above, `strncpy` is used to copy the string from `str1` to `str2` without overflowing the buffer. This prevents the content from overwriting other program memory.
Copying Strings – Buffer Overflows

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

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
\text{str1} & \text{str2} & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{str1} & \text{str2} & \\
\hline
'h' & 'e' & 'l' & 'l' & 'o' & ',' & 'w' & 'o' & 'r' & 'l' & 'd' & '!' & '\0' \\
'h' & 'e' & 'l' & 'l' & 'o' & ',' & \\
\end{array}
\]

- 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!
```

![Diagram showing buffer overflow](image-url)
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>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>'h'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'e'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>','</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'w'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'r'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'d'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'!'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'\0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>str2</th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'e'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>','</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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!
```

<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>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<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>

<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 Strings – Buffer Overflows

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

![Diagram showing buffer overflow](image-url)
Copying Strings – Buffer Overflows

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

```
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  
  'h' 'e' 'l' 'l' 'o' ',' ' ' 'w' 'o' 'r' 'l' 'd'
```

The `strncpy()` function can be used to copy strings without overflow by specifying the maximum number of characters to copy: `strncpy(str2, str1, 6);`
Copying Strings – Buffer Overflows

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

![Diagram showing buffer overflow and memory overwrite](image-url)
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>'w'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</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>'w'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</td>
<td>'!'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
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>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></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></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>str2</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>'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></td>
<td>'\0'</td>
</tr>
</tbody>
</table>
**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` will *not be null terminated*!

```c
// copying "hello"
char str2[5];
strncpy(str2, "hello, world!", 5); // doesn't copy '\0'!
```

If there is no null-terminating character, we may not be able to tell where the end of the string is anymore. E.g. `strlen` may continue reading into some other memory in search of '\0'!
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
Copying Strings - `strncpy`

```c
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```
Copying Strings - `strncpy`

```c
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```
Copying Strings - `strncpy`

```c
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
Copying Strings - `strncpy`

char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
Copying Strings - strncpy

char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
Copying Strings - strncpy

```c
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

![Character representation](image)
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
Copying Strings - strncpy

```c
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```
If necessary, we can add a null-terminating character ourselves.

```c
// copying "hello"
char str2[6]; // room for string and '\0'
strncpy(str2, "hello, world!", 5); // doesn't copy '\0'!
str2[5] = '\0'; // add null-terminating char
```
Important note: C doesn’t automatically initialize variables or values to a default value.

```c
int x;     // contains garbage value
char str[6]; // contains garbage characters
```
char str1[14];
strncpy(str1, "hello there", 5);
### Copying Strings - `strncpy`

```c
char str1[14];
strncpy(str1, "hello there", 5);
```

<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>
</table>
```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);

<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>
<th>12</th>
<th>13</th>
</tr>
</thead>
</table>

hello??J???
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

Respond with your thoughts on PollEv:
pollev.com/cs107 or text CS107 to 22333 once to join.
What is printed out by the example string program?

str = Hi, len = 8
str = Hi, len = 2
str = Hi earth, len = 8
str = Hi earth, len = 2
None/other
We cannot concatenate C strings using +. This adds addresses!

// e.g. param1 = 0x7f, param2 = 0x65
void doSomething(char *param1, char *param2) {
    printf("%s", param1 + param2);  // adds 0x7f and 0x65!
}

Instead, use `strcat`. 
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{lstlisting}[language=C]
char str1[13];       // enough space for strings + '\0'
strcpy(str1, "hello ");
strcat(str1, "world!");  // removes old '\0', adds new '\0' at end
printf("%s", str1);    // hello world!
\end{lstlisting}

Both \texttt{strcat} and \texttt{strncat} remove the old '\0' and add a new one at the end.
Concatenating Strings

```c
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);
```
To omit characters at the end, make a new string that is a partial copy of the original.

```c
// Want just "race"
char str1[8];
strcpy(str1, "racecar");

char str2[5];
strncpy(str2, str1, 4);
str2[4] = '\0';
printf("%s\n", str1);  // racecar
printf("%s\n", str2);  // race
```
Goal: 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("BAILEY")` should print:

```
B
BA
BAI
BAIL
BAILE
BAILEY
AILEY
ILEY
LEY
EY
Y
```

To start: let’s print the top half of the diamond.
Demo: Diamond, Part 1
You can also create a char * variable yourself that points to an address within in an existing string.

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

```c
char *otherStr = myString;  // points to 'H'
```
**Substrings**

char *s (pointers to characters) are strings. We can use them to create substrings of larger strings.

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

![Diagram showing the characters in the `chars` and `str1` variables]
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;
```

![Diagram showing memory layout of chars and str1, str2]
Since C strings are pointers to characters, we can adjust the pointer to omit 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
```

![Diagram of string memory representation]
Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

```c
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
str2[0] = 'f';
printf("%s %s\n", chars, str1);
printf("%s \n", str2);
```
Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

```c
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
str2[0] = 'f';
printf("%s %s\n", chars, str1); // racefar racefar
printf("%s\n", str2); // far
```

![Diagram showing memory layout of C strings with substrings](image)
We can combine pointer arithmetic and copying to make any substrings we’d like.

```c
// Want just "ace"
char str1[8];
strcpy(str1, "racecar");

char str2[4];
strncpy(str2, str1 + 1, 3);
str2[3] = \0;
printf("%s\n", str1);       // racecar
printf("%s\n", str2);       // ace
```
Write a function `diamond` that accepts a string parameter and prints its letters in a "diamond" format as shown below.

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

  B
  BA
  BAI
  BAIL
  BAILE
  BAILEY
  AILEY
  ILEY
  LEY
  EY
  Y

Now let’s implement the second half of the diamond!
Demo: Diamond, Part 2
char * vs. char[]

• char * is an 8-byte pointer – it stores an address of a character
• char[] is an array of characters – it stores the actual characters in a string
• When you pass a char[] as a parameter, it is automatically passed as a char * (pointer to its first character)
• Stay tuned for next lecture for more!
Recap

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings

Next time: more strings

**Lecture 6 takeaway:** C strings are null-terminated arrays of characters. We can manipulate them using string and pointer operations.
Extra Practice
Copycat exercise

**Challenge:** implement `strcat` using other string functions.

```c
char src[9];
strcpy(src, "We Climb");
char dst[200]; // lots of space
strcpy(dst, "The Hill ");
strcat(dst, src);
```

How could we replace a call to `strcat` with a call to `strcpy` instead?
Challenge: implement strcat using other string functions.

```c
char src[9];
strcpy(src, "We Climb");
char dst[200]; // lots of space
strcpy(dst, "The Hill ");

strcat(dst, src); equivalent strcpy(dst + strlen(dst), src);
```
```c
char buf[9];
strcpy(buf, "Potatoes");
printf("%s\n", buf);
char *word = buf + 2;
strncpy(word, "mat", 3);
printf("%s\n", buf);
```

Line 6: What is printed?

A. matoes  
B. mattoes  
C. Pomat  
D. **Potatoes**  
E. Something else  
F. Compile error