CS107 Winter 2020, Lecture 4
C Strings

reading:
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?
Plan For Today

• Characters
• Strings
• Common string operations: String length and comparing strings
• Strings, memory, and pointers, part 1
• **Break**: Announcements
• Common string operations: Copying strings
• Strings as function parameters
Warmup exercise

What do the following 8-bit binary numbers represent? (select all that apply)

10010101
A. Unsigned integer, 149
B. Two’s complement signed integer, -21
C. Two’s complement signed integer, -107
D. C char variable
E. None/other

01100011
A. Unsigned integer, 99
B. Two’s complement signed integer, 99
C. C char variable, 'c'
D. None/other
Plan For Today

• Characters
• Strings
• Common string operations: String length and comparing strings
• Strings, memory, and pointers, part 1
• **Break**: Announcements
• Common string operations: Copying strings
• Strings as function parameters
A **char** is a 1-byte 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 `char` as an 8-bit unsigned integer ("ASCII value").

- Uppercase/lowercase letters, digits are sequentially numbered.
- Lowercase letters are 32 more than their uppercase equivalents.

```c
char uppercaseA = 'A'; // Actually 65
char lowercaseA = 'a'; // Actually 97
char zeroDigit = '0'; // Actually 48

// 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>isalpha(ch)</td>
<td>true if ch is 'a' through 'z' or 'A' through 'Z'</td>
</tr>
<tr>
<td>islower(ch)</td>
<td>true if ch is 'a' through 'z'</td>
</tr>
<tr>
<td>isupper(ch)</td>
<td>true if ch is 'A' through 'Z'</td>
</tr>
<tr>
<td>isspace(ch)</td>
<td>true if ch is a space, tab, new line, etc.</td>
</tr>
<tr>
<td>isdigit(ch)</td>
<td>true if ch is '0' through '9'</td>
</tr>
<tr>
<td>toupper(ch)</td>
<td>returns uppercase equivalent of a letter</td>
</tr>
<tr>
<td>tolower(ch)</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`
Code study: `ctype.h` implementations

```
int my_isdigit(int ch) {
    return ch >= '0' && ch <= '9';
}
```

Standard ASCII maps 0x00 - 0x7f to letters, digits, and punctuation

- `man ascii` to display table
- 8th bit used as parity/error check in some situations
- No consensus for characters mapped to 0x80-0xff

**Small note:** bool (e.g., true and false) is not a built-in C type. Import `stdbool.h` if you want to use bools.
```c
int my_isdigit(int ch) {
    return ch >= '0' && ch <= '9';
}
```

Contains a bonus example. Test your bitwise/ASCII skills!
Plan For Today

- Characters
- Strings
  - Common string operations: String length and comparing strings
  - Strings, memory, and pointers, part 1
  - **Break**: Announcements
  - Common string operations: Copying strings
  - Strings as function parameters
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. 

'\0' is the null-terminating character; you always need to allocate one extra space in an array for it.
C Strings, initialization

3 equivalent ways to create a string in local memory (on the stack):

"Hello"  
|   | 'H' | 'e' | 'l' | 'l' | 'o' | '\0' |
---|-----|-----|-----|-----|-----|------|

1. char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};

2. char str[6];  
str[0] = 'H';  
str[1] = 'e';  
str[2] = 'l';  
str[3] = 'l';  
str[4] = 'o';  
str[5] = '\0';  
Create an empty 6-character array, then set each element.

3. char str[] = "Hello";  
Shorthand: Populate array, auto-add null terminator
Plan For Today

• Characters
• Strings
• Common string operations: String length and comparing strings
• Strings, memory, and pointers, part 1
• Break: Announcements
• Common string operations: Copying strings
• Strings as function parameters
# 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), 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> finds 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), 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 (might not include null-terminating character).</td>
</tr>
<tr>
<td><code>strcat(dst, src), 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), 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>strncmp</code> stops comparing after at most <code>n</code> characters.</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 <code>n</code> chars (might not include null-terminating character).</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 (might not include null-terminating character).</td>
</tr>
<tr>
<td><code>strspn(str, accept)</code></td>
<td>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>

---

*we’ll cover these this lecture*
The string library: strlen

strlen: returns the # of chars in a C string.

```c
char str[] = "Hello";
int length = strlen(str); // 5
```

- The null-terminating character '\0' does not count towards the length.
- strlen is O(N) because it counts the characters until the null terminator.

**Important:** Strings are not objects.
- They do not embed additional information (e.g., string length).
- Many string functions assume valid string input; i.e., ends in a null terminator.
What is printed out by the following program?

```c
int main(int argc, char *argv[]) {
    char str[] = "Hi earth";
    str[2] = '\0';
    printf("str = %s, len = %ld\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
1. Valid strings are null-terminated.
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.

Demo: `strcmp_ex`

cp -r /afs/ir/class/cs107/samples/lectures/lect4 .
Questions with `strcmp`

```c
char str1[] = "Lettuce";
char str2[] = "Cabbage";
int cmp_result = strcmp(str1, str2);

// Print first string
printf("%s", str1);

// If result is 0, they are the same
if (cmp_result == 0) {
    printf(" is the same as ");
}

// Else if result is less than 0, first string comes before
else if (cmp_result < 0) {
    printf(" comes before ");
}

// Else, first string comes after
else {
    printf(" comes after ");
}

// Print second string
printf("%sn", str2);

// Print result of strcmp
printf("int result of strcmp: %d\n", cmp_result);
```

- Line 3: What is the return value of `strcmp()`?
- Lines 3, 6, 8: **Why can’t we just use ==, >, < to compare strings?**
Plan For Today

• Characters
• Strings
• Common string operations: String length and comparing strings
• Strings, memory, and pointers, part 1
• Break: Announcements
• Common string operations: Copying strings
• Strings as function parameters
Key takeaway #2

1. Valid strings are null-terminated.

2. An array name (and a string name, by extension) is the **address** of the first element.
Pointers and memory, a refresher

* Take CS107 to learn much more!!

** We are currently taking CS107
A **pointer** is a variable that stores a memory address.

```c
int main(int argc, char *argv[]) {
    int x = 2;
    int *xptr = &x;
    *xptr = 3;
    printf("%d\n", x);
    printf("%d\n", *xptr);
    printf("%p\n", xptr);
    return 0;
}
```
A pointer is a variable that stores a memory address.

```c
int main(int argc, char *argv[]) {
    int x = 2;
    int *xptr = &x;
    *xptr = 3;
    printf("%d\n", x); // 3
    printf("%d\n", *xptr); // 3
    printf("%p\n", xptr);
    return 0;
}
```
A pointer is a variable that stores a memory address.

```c
int main(int argc, char *argv[]) {
    int x = 2;
    int *xptr = &x;
    *xptr = 3;
    printf("%d\n", x);    // 3
    printf("%d\n", *xptr); // 3
    printf("%p\n", xptr); // 0x7fffffffffe820
    return 0;
}
```

"%p": Format as address
Memory, in detail

- Memory is a big array of bytes.
- Each byte has a memory address that is commonly written in hexadecimal.
- Local variables are created and pushed onto the stack, which is a part of memory.
- A pointer is a variable that stores a memory address.
- On the myth machines, all pointers are 64-bits, or 8 bytes long.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffef825</td>
<td>…</td>
</tr>
<tr>
<td>0x7fffffffef824</td>
<td>…</td>
</tr>
<tr>
<td>0x7fffffffef823</td>
<td>…</td>
</tr>
<tr>
<td>0x7fffffffef822</td>
<td>…</td>
</tr>
<tr>
<td>0x7fffffffef821</td>
<td>…</td>
</tr>
<tr>
<td>0x7fffffffef820</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
char * and char[]

• An array name (and a string name, by extension) is the address of the first element.

1 char str1[] = "Lettuce";
2 printf("%c", *str1);
3
4 char *cptr = str1;
5 printf("%s\n", cptr);
6 printf("%s\n", str1);
7 printf("%c\n", *cptr);

• Line 2: What is printed?
• Line 4: What is stored in ptr?
• Lines 5-7: What is printed?
• An array name (and a string name, by extension) is the address of the first element.

```c
1 char str1[] = "Lettuce";
2 printf("%c", *str1); // L
3
4 char *cptr = str1;
5 printf("%s\n", cptr); // Lettuce
6 printf("%s\n", str1); // Lettuce
7 printf("%c\n", *cptr); // L
```
Comparing pointers

• An array name (and a string name, by extension) is the address of the first element.

• As a result, it is meaningless in C to compare strings with relational operators:

```
char str1[] = "Cabbage";
char str2[] = "Cabbage";

if (str1 > str2) {
    ...
}
```

C interprets as: compare 0xf0 > 0xe8

Operations on arrays tend to be operations on addresses. Use string library functions when you can!
Answers with `strcmp`

```
1 char str1[] = "Lettuce";
2 char str2[] = "Cabbage";
3 int cmp_result = strcmp(str1, str2);
4
5 printf("%s", str1);
6 if (cmp_result == 0) {
    printf(" is the same as ");
8 } else if (cmp_result < 0) {
9     printf(" comes before ");
10 } else {
11     printf(" comes after ");
12 }
13 printf("%s\n", str2);
14 printf("int result of strcmp: %d\n", cmp_result);
```
Because `char *`s are pointers to characters, we can use them to create substrings of larger strings!

• Key: Incrementing a `char *` by 1 increases the memory address by 1 byte.

```c
1 char chars[] = "racecar";
2 char *str1 = chars;
3 char *str2 = chars + 4;
4 str2[0] = 'f';
5 printf("%s, %s\n", chars, str1);
6 printf("%s\n", str2);
```
Because char *s are pointers to characters, we can use them to create substrings of larger strings!

- Key: Incrementing a char * by 1 increases the memory address by 1 byte.

```c
char chars[] = "racecar";
char *str1 = chars;
char *str2 = chars + 4;
str2[0] = 'f';
printf("%s, %s
", chars, str1);
printf("%s
", str2);
```
Because char *s are pointers to characters, we can use them to create substrings of larger strings!

• Key: Incrementing a char * by 1 increases the memory address by 1 byte.

```c
char 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
```
Questions?
Plan For Today

• Characters
• Strings
• Common string operations: String length and comparing strings
• Strings, memory, and pointers, part 1
• **Break**: Announcements
• Common string operations: Copying strings
• Strings as function parameters
Announcements

• GDB Config File (cs107.stanford.edu/resources/gdb)
• emacs/vim config files also available
• Piazza is an official channel for course communication this quarter
• We hope you enjoyed your first lab!

No class on Monday, 1/20 (Martin Luther King Jr. Day)
• Video lecture posted over the weekend
"Hello, I'd like to hear a C joke"

strcpy(joke, chicken, strlen(chicken));
printf("%s", joke);

Segmentation fault (core dumped)

Note:
This code also just doesn’t compile
Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters
The string library: \texttt{strcpy}

\texttt{strcpy(dst, src)}: copies characters in \texttt{src} to \texttt{dst}, including null terminator.

```c
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);
```

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The string library: `strcpy`

`strcpy(dst, src)`: copies characters in `src` to `dst`, including null terminator.

```c
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);
```

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<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>\0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

buf: 'h' 'e' 'l' 'l' 'o' '\0' ? ?
The string library: \texttt{strcpy}

\texttt{strcpy}(\textit{dst}, \textit{src}): copies characters in \textit{src} to \textit{dst}, including null terminator.

```c
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf); // hello
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>'o'</td>
<td>'\0'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

buf:
The string library: `strncpy`

`strncpy(dst, src, n)` copies at most `n` characters in `src` to `dst`.

```c
char buf[8];
strcpy(buf, "hello");
printf("%s\n", buf);  // hello
char str[] = "Hi";
int n = strlen(str);
strncpy(buf, str, n);
buf[n] = '\0';
printf("%s\n", buf);
```

<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>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'\0'</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>'H'</td>
<td>'i'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The string library: `strncpy`

`strncpy(dst, src, n)` copies at most `n` characters in `src` to `dst`.

```c
1. char buf[8];
2. strcpy(buf, "hello");
3. printf("%s\n", buf); // hello
4. char str[] = "Hi";
5. int n = strlen(str);
6. strncpy(buf, str, n);
7. buf[n] = '\0';
8. printf("%s\n", buf);
```

<table>
<thead>
<tr>
<th>buf:</th>
<th>'h'</th>
<th>'e'</th>
<th>'l'</th>
<th>'l'</th>
<th>'o'</th>
<th>'\0'</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>str:</td>
<td>'H'</td>
<td>'i'</td>
<td>'\0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n:</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The string library: `strncpy`

`strncpy(dst, src, n)` copies at most `n` characters in `src` to `dst`.

```c
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf); // hello
4 char str[] = "Hi";
5 int n = strlen(str);
6 strncpy(buf, str, n);
7 buf[n] = '\0';
8 printf("%s\n", buf);
```

⚠️ `strncpy()` does not automatically put a null terminator at the end of the copy.
The string library: `strncpy`

`strncpy(dst, src, n)`: copies at most `n` characters in `src` to `dst`.

```
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);       // hello
4 char str[] = "Hi";
5 int n = strlen(str);
6 strncpy(buf, str, n);
7 buf[n] = '\0';
8 printf("%s\n", buf);
```

⚠ `strncpy()` does not automatically put a null terminator at the end of the copy.
The string library: `strncpy`

`strncpy(dst, src, n)`: copies at most `n` characters in `src` to `dst`.

```c
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);    // hello
4 char str[] = "Hi";
5 int n = strlen(str);
6 strncpy(buf, str, n);
7 buf[n] = '\0';
8 printf("%s\n", buf);    // Hi
```

<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>
</tr>
</thead>
<tbody>
<tr>
<td>'H'</td>
<td>'i'</td>
<td>'\0'</td>
<td>'l'</td>
<td>'o'</td>
<td>'\0'</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'H'</td>
<td>'i'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
```

n: 2
String copying exercise

1. char buf[____ ];
2. strcpy(buf, "Chadwick");
3. printf("%s\n", buf);
4. char *word = buf + 2;
5. strncpy(word, "ris", 3);
6. printf("%s\n", buf);

Line 1: What value should go in the blank?
A. 7  D. 12
B. 8  E. strlen("Chadwick")
C. 9  F. Something else

Line 6: What is printed?
A. risick  D. Chrisick
B. risdwick  E. Something else
C. Chris  F. Compile error
Pitfalls with `strcpy`: Buffer overflow

⚠ `strcpy(dst, src)` does not check if the destination is large enough:

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

<table>
<thead>
<tr>
<th>str1</th>
<th>h</th>
<th>e</th>
<th>l</th>
<th>l</th>
<th>o</th>
<th>,</th>
<th></th>
<th>w</th>
<th>o</th>
<th>r</th>
<th>l</th>
<th>d</th>
<th>!</th>
<th>\0</th>
</tr>
</thead>
</table>

Pitfalls with `strcpy`: Buffer overflow

⚠ `strcpy(dst, src)` does not check if the destination is large enough:

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

Buffer overflow: `strcpy(dst, src)` writes past the available space in `dst`. This is a major security flaw which hackers exploit all the time!
Pitfalls with `strncpy`

✅ `strncpy(dst, src, n)`: always writes up to \(n\) bytes; as the programmer, you can avoid buffer overflow.

⚠ However, if the resulting string is not null-terminated, you could *read* past the end of the \(dst\) buffer!

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

<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>str1</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>
**Pitfalls with strncpy**

✅ `strncpy(dst, src, n)`: always writes up to `n` bytes; as the programmer, you can avoid buffer overflow.

⚠ However, if the resulting string is not null-terminated, you could *read* past the end of the `dst` buffer!

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

<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>str1</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>
**Pitfalls with `strncpy`**

✅ `strncpy(dst, src, n)`: always writes up to `n` bytes; as the programmer, you can avoid buffer overflow.

⚠ However, if the resulting string is not null-terminated, you could _read_ past the end of the `dst` buffer!

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

|--------|-----|-----|-----|-----|-----|---|---|---|---|---|---|---|---|---|

`hello??J??`
Questions?
Plan For Today

• Characters
• Strings
• Common string operations: String length and comparing strings
• Strings, memory, and pointers, part 1
• Break: Announcements
• Common string operations: Copying strings
• Strings as function parameters
1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.

Why did C bother with this representation?

C is a powerful, efficient language that requires a solid understanding of computer memory.

Over the next two weeks, we will hone this understanding with lots of practice!
Key takeaway #3

1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.
3. All parameters in C are “pass by value.” For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.
Pass by value

- When you pass a value as a parameter, C passes a **copy** of that value.

What gets printed out by the following program?

```c
void foo(int a) {
    a++;
    printf("%d\n", a);  // 3
}

int main(int argc, char *argv[]) {
    int x = 2;
    foo(x);
    printf("%d\n", x);  // *2*
    return 0;
}
```
“Pass by reference” with pointers

- When you pass a value as a parameter, C passes a copy of that value. To modify the original variable, use a **pointer** to that variable.

```c
void foo(int *a) {
    *a = *a + 1;     // Equivalent to (*a)++
    printf("%d\n", *a);  // 3
}

int main(int argc, char *argv[]) {
    int x = 2;
    foo(&x);
    printf("%d\n", x);  // *3*
    return 0;
}
```

All C parameters (even pointers) are passed in by value.
Passing in strings

How do you think the parameter `str` is being represented?

```c
void fun_times(char *str) {
    str[0] = 'd';
}
```

```c
int main(int argc, char *argv[]) {
    char local_str[] = "rice";
    fun_times(local_str);
    return 0;
}
```

A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`
Passing in strings

How do you think the parameter \texttt{str} is being represented?

\begin{verbatim}
void fun_times(char *str) {
    str[0] = 'd';
}

int main(int argc, char *argv[]) {
    char local_str[] = "rice";
    fun_times(local_str);
    return 0;
}
\end{verbatim}

\begin{itemize}
    \item[A.] A copy of the array \texttt{local_str}
    \item[B.] A pointer containing an address to the first element in \texttt{local_str}
\end{itemize}
How do you think the parameter `str` is being represented?

```c
void more_fun_times(char str[]) {
    str[0] = 'm';
}

int main(int argc, char *argv[]) {
    char local_str[] = "nice";
    more_fun_times(local_str);
    return 0;
}
```

A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`
How do you think the parameter `str` is being represented?

```c
void more_fun_times(char str[]) {
    str[0] = 'm';
}

int main(int argc, char *argv[]) {
    char local_str[] = "nice";
    more_fun_times(local_str);
    return 0;
}
```

A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`