CS107, Lecture 4
C Strings

Reading: K&R (1.9, 5.5, Appendix B3) or Essential C section 3
Bits and Bytes Revisited
Data is really stored in binary

int x = 5; // really 0b00...0101 in memory!

char ch = 'A'; // really 0b01000001 in memory!
We know what that binary representation is for integers

```c
int x = 5;   // really 0b00...0101 in memory!

char ch = 'A'; // really 0b01000001 in memory! (65)

int y = -5; // two's complement: 0b111...11011

unsigned long z = ULONG_MAX; // 0b111...111
```
We can manipulate that binary representation with bitwise operators

```c
int x = 5; // in binary it's 0b0000....00101

if (x & 0x4 != 0) {
    printf("x's third bit from the right is on\n");
}

// turn on the 2nd bit from the right
x |= 0x2;

int y = 1;
y <<= x; // y is shifted by 7 places
```
Bitwise OR (|)

| with 1 is useful for turning select bits on.

```java
int x = 5; // 0b101

// Turn on the 2\textsuperscript{nd} bit from the right
x |= 0x2; // 0b111

// Turn on the 9\textsuperscript{th} bit from the right
x |= (1 << 8);
```
Bitwise OR (\(|\))

| is useful for taking the union of bits.

```c
int x = 5;       // 0b00101
int y = 26;      // 0b11010
int z = x | y;   // 0b11111
printf("%d\n", z); // 31
```
& with 0 is useful for turning select bits off.

```c
int x = 5; // 0b101

// Turn off the 3rd bit from the right
x &= -5; // -5 is 0b111...1011
```
Bitwise AND (&)

& is useful for taking the intersection of bits.

```c
int x = 21;    // 0b10101
int y = 27;    // 0b11011
int z = x & y; // 0b10001
printf("%d\n", z); // 17
```
Bitwise XOR (^)

^ with 1 is useful for flipping select bits.

```c
int x = 5; // 0b101

// Flip the 2\textsuperscript{nd} bit from the right
x ^= 2; // 0b111
```
~ is useful for flipping all bits.

```c
int x = 5; // 0b101

// Flip all bits
x = ~x; // 0b11111...1010, which is -6

// Take two’s complement (same as negating)
int y = ~x + 1; // same as -x
```
Bitwise SHIFT (>> or <<)

>> and << are useful for moving bits.

```java
int x = 5; // 0b00101
x <<= 2;   // 0b10100

// Check if 6th bit from the right is on
if (x & (1 << 5)) {...
// or
if ((x >> 5) & 1) {...
```
Number Literal Suffixes

U makes a literal unsigned, and L makes a literal a long.

```java
int w = -5 >> 1; // 0b1111...1101, -5
int x = -5U >> 1; // 0b0111...1101, 2147483645

int y = 1 << 32; // 0! (technically undefined)
int z = 1L << 32; // 4294967296
```
A variable and its binary representation are one and the same

int x = 5; // in binary it’s 0b0000....00101

// turn on the 2nd bit from the right
x |= 0x2;

int y = 1;
y <<= x; // y shifted by 7 places
We can better understand overflow and type conversion behavior

// Overflow
char x = 126;  // 0b01111110
x += 2;  // -128! 0b10000000

// Expansion
long y = x;  // 0b1111...10000000

// Truncation helps us isolate the byte we want
int z = 0xaabbccdd;
char justCs = z >> 8;
We can inspect our program with GDB

- GDB is a **command-line debugger**, a text-based debugger with similar functionality to other debuggers you may have used, such as in Qt Creator
- It lets you put **breakpoints** at specific places in your program to pause there
- It lets you step through execution line by line
- It lets you print out values of variables in various ways (including binary)
- It lets you track down where your program crashed
- And much, much more!

GDB is essential to your success in CS107 this quarter! We’ll be building our familiarity with GDB over the course of the quarter.
gdb on a program

- `gdb myprogram` run gdb on executable
- `b` Set breakpoint on a function (e.g., `b main`) or line (b 42)
- `r 82` Run with provided args
- `n, s, continue` control forward execution (next, step into, continue)
- `p` print variable (p varname) or evaluated expression (p 3L << 10)
  - `p/t, p/x` binary and hex formats.
  - `p/d, p/u, p/c`
- `info` args, locals

**Important**: gdb does not run the current line until you hit “next”
At this point, setting breakpoints/stepping in gdb may seem like overkill for what could otherwise be achieved by copious `printf` statements.

However, gdb is incredibly useful for assign1 (and all assignments):

• A fast “C interpreter”: `p + <expression>`
  • Sandbox/try out ideas around bitshift operators, signed/unsigned types, etc.
  • Can print values out in binary!
  • Once you’re happy, then make changes to your C file

• **Tip**: Open two terminal windows and SSH into myth in both
  • Keep one for emacs, the other for gdb/command-line
  • Easily reference C file line numbers and variables while accessing gdb

• **Tip**: Every time you update your C file, `make` and then rerun gdb.

Gdb takes practice! But the payoff is tremendous! 😊
CS107 Topic 2: How can a computer represent and manipulate more complex data like text?
CS107 Topic 2

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

By the end of lecture, we’ll be able to 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
• Practice: Diamond
Lecture Plan

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

```java
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>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
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

• **Practice:** Diamond
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.
char myString[6];
myString[0] = 'H';
myString[1] = 'e';
myString[2] = 'l';
... 
myString[5] = '\0';
String Length

Strings are **not** objects. They do not embed additional information (e.g., string length). We must calculate this!

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

char 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);
```
• Characters
• Strings
• Common String Operations
  • Comparing
  • Copying
  • Concatenating
  • Substrings
• Practice: Diamond
## Common string.h Functions

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

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

char str2[6];
strcpy(str2, str1);

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>str1</td>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
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!
```

<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>
</tbody>
</table>
| 'h'| 'e'| 'l'| 'l'| 'o'| ','|   | 'w'| 'o'| 'r'| 'l'| 'd'| '!'| '\0'

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<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>
</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!
```
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'
  ↓
0 1 2 3 4 5
str2 'h' 'e' ? ? ? ?
```

- 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>
</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>
</tr>
<tr>
<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>h</td>
<td>e</td>
<td>l</td>
<td></td>
<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!
```

```
str1: 'h' 'e' 'l' 'l' 'o' ',' 'w' 'o' 'r' 'l' 'd' '!' '\0'
str2: 'h' 'e' 'l' 'l' ? ?
```

- 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!
```
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!
```

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

![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!
```

![Diagram showing buffer overflow](attachment:image.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!
```

<table>
<thead>
<tr>
<th>str1</th>
<th>str2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'h' 'e' 'l' 'l' 'o' ','</td>
<td>'h' 'e' 'l' 'l' 'o' ','</td>
</tr>
</tbody>
</table>

- Other program memory:
  - `\w` other program memory -
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>'\0'</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></td>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>','</td>
</tr>
</tbody>
</table>
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);  // not enough space - overwrites other memory!

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
\text{str1} & 'h' & 'e' & 'l' & 'l' & 'o' & ',' & 'w' & 'o' & 'r' & 'l' & 'd' & '!' & '\0' \\
\text{str2} & 'h' & 'e' & 'l' & 'l' & 'o' & ',' & ' ' & 'w' & 'o' & 'r' & 'l' & 'd' & \text{other program memory}\n\end{array}
\]
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);  // not enough space - overwrites other memory!

```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!
```

![Diagram showing buffer overflow caused by copying a string to a buffer with insufficient space](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></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>
<tr>
<td></td>
<td>'w'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</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></td>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>','</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'w'</td>
<td>'o'</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);
Copying Strings - strncpy

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

- other program memory -
Copying Strings - strncpy

```c
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(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>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- other program memory -
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);
```

<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>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>e</td>
<td>l</td>
<td>l</td>
</tr>
</tbody>
</table>

- other program memory -
Copying Strings - strncpy

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

<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>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>

<table>
<thead>
<tr>
<th>str2</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</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>

- other program memory -
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`

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

```c
strncpy (str2, "hello, world!", 5);
```

<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>

```
str2
```

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

- other program memory -
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);

- other program memory -
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
C Doesn’t Automatically Initialize

Important note: C doesn’t automatically initialize variables or values to a default value.

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

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

char str1[14];
strncpy(str1, "hello there", 5); 
printf("%s\n", str1);
```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>
</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>
<td>13</td>
</tr>
</tbody>
</table>

The string is copied up to the 5th character of the destination string `str1`.

![Diagram of string copying](image)
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
Copying Strings - strncpy

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

The `strncpy` function copies the string "hello there" to `str1`, but only up to 5 characters due to the third argument. The resulting string is "hel".
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
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
```
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

cchar 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  

🤔
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: `str(n)cat`

- **`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'
strcpy(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.
Concatenating Strings

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

```c
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>'w'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</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>
</tr>
<tr>
<td>str2</td>
<td>'w'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</td>
<td>'?'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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);
```
Concatenating Strings

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

strcat(str1, str2);
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';

char *otherStr = myString;  // points to 'H'
```
char *s (pointers to characters) are strings. We can use them to create substrings of larger strings.

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

```
chars: ['r' 'a' 'c' 'e' 'c' 'a' 'r' '\0']
str1: 0xef
str2: 0xf5
```
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);
```

```
\textbf{chars}: ['r', 'a', 'c', 'e', 'f', 'a', 'r', '\0']
\textbf{str1}: 0xf1
\textbf{str2}: 0xf5
```

0xf1 0xf2 0xf3 0xf4 0xf5 0xf6 0xf7 0xf8
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
| 0xf1 | 0xf2 | 0xf3 | 0xf4 | 0xf5 | 0xf6 | 0xf7 | 0xf8 |
\hline
\hline
\hline
\end{tabular}
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); // racefar racefar
printf("%s\n", str2); // far
```
Strings Practice

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  D. Potatoes
B. mattoes  E. Something else
C. Pomat    F. Compile error
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!
You can create char * pointers to point to any character in an existing string and reassign them since they are just pointer variables. You cannot reassign an array.

```c
char myString[6];
strcpy(myString, "Hello");
myString = "Another string";          // not allowed!
---
char *myOtherString = myString;
myOtherString = somethingElse;       // ok
```
Substrings

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
```
Substrings

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

• Practice: 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
```
Bailey!
Practice: Diamond

cp -r /afs(ir/class/cs107/lecture-code/lect4 .
Recap

• Characters
• Strings
• Common String Operations
  • Comparing
  • Copying
  • Concatenating
  • Substrings
• Practice: Diamond

Lecture 4 takeaway: C strings are null-terminated arrays of characters. We can manipulate them using string and pointer operations.

Next time: more strings
Extra Practice
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);
```