CS107, Lecture 5
More C Strings

Reading: K&R (1.6, 5.5, Appendix B3) or Essential C section 3
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 (last time)
• Helps us better understand buffer overflows, a common bug (last time)
• Introduces us to pointers, because strings can be pointers (this 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

• Understand how to use the built-in string functions for common string tasks
• Learn more about the risks of buffer overflows and how to mitigate them
• Understand how strings are represented as pointers and how that helps us better understand their behavior
Lecture Plan

- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows, Security and Valgrind
- Pointers
- Strings in Memory

```bash
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```
Lecture Plan

• **Recap: Strings so far**
• Searching in Strings
• **Practice: Password Verification**
• Buffer Overflows, Security and Valgrind
• Pointers
• Strings in Memory

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```
C strings are arrays of characters ending with a null-terminating character '\0'.

<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>

String operations such as strlen use the null-terminating character to find the end of the string.

Side note: use strlen to get the length of a string. Don’t use sizeof!
## 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 n characters.</td>
</tr>
<tr>
<td><code>strncmp(str1, str2, n)</code></td>
<td></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></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 n chars, and does not add null-terminating char.</td>
</tr>
<tr>
<td><code>strncpy(dst, src, n)</code></td>
<td></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 n characters. <code>Always</code> adds a null-terminating character.</td>
</tr>
<tr>
<td><code>strncat(dst, src, n)</code></td>
<td></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>
<tr>
<td><code>strcspn(str, reject)</code></td>
<td></td>
</tr>
</tbody>
</table>
Substrings

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

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

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

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
```
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
BAILE
BAILEY
AILEY
ILEY
LEY
EY
Y
```
Practice: String Diamond

string_diamond.c
Lecture Plan

• **Recap:** Strings so far
• **Searching in Strings**
• **Practice:** Password Verification
• Buffer Overflows, Security and and Valgrind
• Pointers
• Strings in Memory

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```
Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```c
char bailey[7];
strcpy(bailey, "Bailey");
char *letterI = strchr(bailey, 'i');
printf("%s\n", bailey); // Bailey
printf("%s\n", letterI); // iley
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the first one. Use `strchr` to obtain a pointer to the last occurrence.
**Searching For Strings**

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```c
char bailey[11];
strcpy(bailey, "Bailey Dog");
char *substr = strstr(bailey, "Dog");
printf("%s\n", bailey); // Bailey Dog
printf("%s\n", substr); // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the \textit{first} one.
String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```c
char bailey[10];
strcpy(bailey, "Bailey Dog");
int spanLength = strspn(bailey, "aBeoi");       // 3
```

“How many places can we go in the first string before I encounter a character *not in* the second string?”
String Spans

strcspn (c = “complement”) returns the length of the initial part of the first string which contains only characters not in the second string.

char bailey[10];
strcpy(bailey, "Bailey Dog");
int spanLength = strcspn(bailey, "driso"); // 2

“How many places can we go in the first string before I encounter a character in the second string?”
When we pass a string as a parameter, it is passed as a char *. We can still operate on the string the same way as with a char[]. (We’ll see why today!).

```c
int doSomething(char *str) {
    char secondChar = str[1];
    ...
}
```

// can also write this, but it is really a pointer
int doSomething(char str[]) { ...}
Arrays of Strings

We can make an array of strings to group multiple strings together:

```c
char *stringArray[5];  // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```c
char *stringArray[] = {
  "Hello",
  "Hi",
  "Hey there"
};
```
Arrays of Strings

We can access each string using bracket syntax:

```c
printf("%s\n", stringArray[0]);  // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the first element of the array*. This is what `argv` is in `main`! This means we write the parameter type as:

```c
void myFunction(char **stringArray) {
```

// equivalent to this, but it is really a double pointer
void myFunction(char *stringArray[]) {
```
Practice: Password Verification

Write a function `verifyPassword` that accepts a candidate password and certain password criteria and returns whether the password is valid.

```c
bool verifyPassword(char *password, char *validChars, char *badSubstrings[], int numBadSubstrings);
```

`password` is valid if it contains only letters in `validChars`, and does not contain any substrings in `badSubstrings`. 
bool verifyPassword(char *password, char *validChars, char *badSubstrings[], int numBadSubstrings);

Example:

char *invalidSubstrings[] = { "1234" };

bool valid1 = verifyPassword("1572", "0123456789", invalidSubstrings, 1);  // true
bool valid2 = verifyPassword("141234", "0123456789", invalidSubstrings, 1);  // false
Practice: Password Verification

verify_password.c
Lecture Plan

• **Recap**: Strings so far
• Searching in Strings
• **Practice**: Password Verification
• **Buffer Overflows, Security and Valgrind**
• Pointers
• Strings in Memory
Recall: Buffer Overflows

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!
Recall: Buffer Overflows

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

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

str2: 0 1 2 3 4 5
    'h' 'e' 'l' 'l' 'o' ','
```
Buffer Overflow Impacts

Buffer overflows are not merely functionality bugs; they can cause a range of unintended behavior:

- Access memory you shouldn’t be able to access
- Modify memory you shouldn’t be able to access
  - Change a value that is used later in the program
  - Change the program to execute your instructions instead of its own
- And more...

It’s our job as programmers to find and fix buffer overflows and other bugs not just for the functional correctness of our programs, but to protect people who use and interact with my code.
Buffer Overflow Impacts

• AOL instant messenger buffer overflow: allowed remote attackers to execute code: [https://www.cvedetails.com/cve/CVE-2002-0362/](https://www.cvedetails.com/cve/CVE-2002-0362/)

• Morris Worm: first internet worm to gain widespread attention; exploited buffer overflow in Unix command called ”finger”: [https://en.wikipedia.org/wiki/Morris_worm](https://en.wikipedia.org/wiki/Morris_worm)
How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

• Constant vigilance while programming (checking arrays and where they are modified)

• Carefully reading documentation

• Thorough testing to uncover issues before release

• Thorough documentation to document assumptions in your code

• (Where possible) use of tools that reduce the possibility for buffer overflows
There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

• Constant vigilance while programming (checking arrays and where they are modified)

• **Carefully reading documentation**

• Thorough testing to uncover issues before release

• Thorough documentation to document assumptions in your code

• (Where possible) use of tools that reduce the possibility for buffer overflows
How can we fix buffer overflows?

MAN page for gets():

“Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.”
How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

• Constant vigilance while programming (checking arrays and where they are modified)

• Carefully reading documentation

• **Thorough testing to uncover issues before release**

• Thorough documentation to document assumptions in your code

• (Where possible) use of tools that reduce the possibility for buffer overflows
How Can We Fix Overflows?

- **Valgrind**: Your Greatest Ally
- Write your own tests
- Consider writing tests *before* writing the main program

🌟 [cs107.stanford.edu/testing.html](cs107.stanford.edu/testing.html) 🌟
How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

• Constant vigilance while programming (checking arrays and where they are modified)
• Carefully reading documentation
• Thorough testing to uncover issues before release
• **Thorough documentation to document assumptions in your code**
• (Where possible) use of tools that reduce the possibility for buffer overflows
How Can We Fix Overflows?

Documentation & MAN Pages (Written by Others)

“The strcpy() function copies the string pointed to by src, including the terminating null byte (‘\0’), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. Beware of buffer overruns! (See BUGS.) …

BUGS

If the destination string of a strcpy() is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there’s enough space. This may be unnecessary if you can show that overflow is impossible, but be careful: programs can get changed over time, in ways that may make the impossible possible.”
How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

• Constant vigilance while programming (checking arrays and where they are modified)
• Carefully reading documentation
• Thorough testing to uncover issues before release
• Thorough documentation to document assumptions in your code
• *(Where possible) use of tools that reduce the possibility for buffer overflows*
Idea 5: Choose your Tools & Languages Carefully

Existing code bases or requirements for a project may dictate what tools you use. Knowing C is crucial – it is and will remain widely used.

When you are choosing tools for systems programming, consider languages that can help guard against programmer error.

- Rust (Mozilla)
- Go (Google)
- Project Verona (Microsoft)
ACM Code of Ethics and Professional Conduct

Preamble

Computing professionals’ actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good. The ACM Code of Ethics and Professional Conduct ("the Code") expresses the conscience of the profession.

The Code is designed to inspire and guide the ethical conduct of all computing professionals, including current and aspiring practitioners, instructors, students, influencers, and anyone who uses computing technology in an impactful way. Additionally, the Code serves as a basis for remediation when violations occur. The Code includes principles formulated as statements of responsibility, based on the understanding that the public good is always the primary consideration. Each principle is supplemented by guidelines, which provide explanations to assist computing professionals in understanding and
2.9 Design and implement systems that are robustly and usably secure.

Breaches of computer security cause harm. Robust security should be a primary consideration when designing and implementing systems. Computing professionals should perform due diligence to ensure the system functions as intended, and take appropriate action to secure resources against accidental and intentional misuse, modification, and denial of service. As threats can arise and change after a system is deployed, computing professionals should integrate mitigation techniques and policies, such as monitoring, patching, and vulnerability reporting. Computing professionals should also take steps to ensure parties affected by data breaches are notified in a timely and clear manner, providing appropriate guidance and remediation.

To ensure the system achieves its intended purpose, security features should be designed to be as intuitive and easy to use as possible. Computing professionals should discourage security precautions that are too confusing, are situationally inappropriate, or otherwise inhibit legitimate use.

In cases where misuse or harm are predictable or unavoidable, the best option may be to not implement the system.
How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

• Constant vigilance while programming (checking arrays and where they are modified)
• Carefully reading documentation
• Thorough testing to uncover issues before release
• Thorough documentation to document assumptions in your code
• (Where possible) use of tools that reduce the possibility for buffer overflows
Buffer Overflows

• We must always ensure that memory operations we perform don’t improperly read or write memory.
  • E.g. don’t copy a string into a space that is too small!
  • E.g. don’t ask for the string length of an uninitialized string!

• The **Valgrind** tool may be able to help track down memory-related issues.
  • See cs107.stanford.edu/resources/valgrind
  • We’ll talk about Valgrind more when we talk about dynamically-allocated memory.
Demo: Memory Errors

memory_errors.c
Lecture Plan

• Recap: Strings so far
• Searching in Strings
• Practice: Password Verification
• Buffer Overflows, Security and Valgrind
• Pointers
• Strings in Memory

cp -r /afs/ir/class/cs107/lecture-code/lect5 .
Pointers

- A *pointer* is a variable that stores a memory address.
- Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
- One (8 byte) pointer can refer to any size memory location!
- Pointers are also essential for allocating memory on the heap, which we will cover later.
- Pointers also let us refer to memory generically, which we will cover later.
• Memory is a big array of bytes.
• Each byte has a unique numeric index that is commonly written in hexadecimal.
• A pointer stores one of these memory addresses.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x105</td>
<td>\0</td>
</tr>
<tr>
<td>0x104</td>
<td>e</td>
</tr>
<tr>
<td>0x103</td>
<td>l</td>
</tr>
<tr>
<td>0x102</td>
<td>p</td>
</tr>
<tr>
<td>0x101</td>
<td>p</td>
</tr>
<tr>
<td>0x100</td>
<td>a</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
• Memory is a big array of bytes.
• Each byte has a unique numeric index that is commonly written in hexadecimal.
• A pointer stores one of these memory addresses.
Looking Back at C++

How would we write a program with a function that takes in an int and modifies it? We might use pass by reference.

```cpp
void myFunc(int& num) {
    num = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 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.

This means whenever we pass something as a parameter, we pass a copy.

If we want to modify a parameter value in the function we call and have the changes persist afterwards, we can pass the location of the value instead of the value itself. This way we make a copy of the address instead of a copy of the value.
int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;

// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xPtr);  // prints 2
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x); // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);     // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);    // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);    // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);       // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);   // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...  
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);    // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x); // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);    // 3!
    ...
}"
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
• If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**. This makes a copy of the data.

• If you are modifying a specific instance of some value, **pass the location** of what you would like to modify. This makes a copy of the data’s location.

• If a function takes an address (pointer) as a parameter, it can **go to** that address if it needs the actual value.
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);        // 2!
    ...
}
```
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);    // 2!
    ...
}
```
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x); // 2!
    ...
}
```
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```
Lecture Plan

• **Recap:** Strings so far
• Searching in Strings
• **Practice:** Password Verification
• Buffer Overflows, Security and Valgrind
• Pointers
• **Strings in Memory**
Strings In Memory

1. If we create a string as a `char[]`, we can modify its characters because its memory lives in our stack space.

2. We cannot set a `char[]` equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.

3. If we pass a `char[]` as a parameter, set something equal to it, or perform arithmetic with it, it’s automatically converted to a `char *`.

4. If we create a new string with new characters as a `char *`, we cannot modify its characters because its memory lives in the data segment.

5. We can set a `char *` equal to another value, because it is a reassign-able pointer.

6. Adding an offset to a C string gives us a substring that many places past the first character.

7. If we change characters in a string parameter, these changes will persist outside of the function.
String Behavior #1: If we create a string as a `char[]`, we can modify its characters because its memory lives in our stack space.
When we declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array. We can modify what is on the stack.

```c
char str[6];
strcpy(str, "apple");
```
String Behavior #2: We cannot set a `char[]` equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
An array variable refers to an entire block of memory. We cannot reassign an existing array to be equal to a new array.

```c
char str[6];
strcpy(str, "apple");
char str2[8];
strcpy(str2, "apple 2");

str = str2;  // not allowed!
```

An array’s size cannot be changed once we create it; we must create another new array instead.
String Behavior #3: If we pass a char[] as a parameter, set something equal to it, or perform arithmetic with it, it’s automatically converted to a char *.
String Parameters

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

```c
void fun_times(char *str) {
    ...
}

int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(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`
String Parameters

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

```c
void fun_times(char *str) {
    ...
}

int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    fun_times(local_str);
    return 0;
}
```

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

How do you think the local variable str is being represented?

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = 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 local variable `str` is being represented?

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = 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 local variable `str` is being represented?

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```

A. A copy of part of the array `local_str`
B. A pointer containing an address to the third element in `local_str`
char * Variables

How do you think the local variable str is being represented?

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```

A. A copy of part of the array local_str
B. A pointer containing an address to the third element in local_str
All string functions take char * parameters – they accept char[], but they are implicitly converted to char * before being passed.

- strlen(char *str)
- strcmp(char *str1, char *str2)
- ...

- char * is still a string in all the core ways a char[] is
  - Access/modify characters using bracket notation
  - Print it out
  - Use string functions
  - But under the hood they are represented differently!

- **Takeaway:** We create strings as char[], pass them around as char *
String Behavior #4: If we create a new string with new characters as a `char *`, we cannot modify its characters because its memory lives in the data segment.
There is another convenient way to create a string if we do not need to modify it later. We can create a `char *` and set it directly equal to a string literal.

```c
char *myString = "Hello, world!";
char *empty = "";

myString[0] = 'h'; // crashes!
printf("%s", myString); // Hello, world!
```
When we declare a char pointer equal to a string literal, the characters are *not* stored on the stack. Instead, they are stored in a special area of memory called the “data segment”. We *cannot* modify memory in this segment.

```c
char *str = "hi";
```

The pointer variable (e.g. `str`) refers to the *address of the first character of the string in the data segment*.

This applies only to creating *new* strings with char *. This does *not* apply for making a char * that points to an existing stack string.
For each code snippet below, can we modify the characters in `myStr`?

```c
char myStr[6];
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
For each code snippet below, can we modify the characters in `myStr`?

```c
char *myStr = "Hi";
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
For each code snippet below, can we modify the characters in `myStr`?

```c
char buf[6];
strcpy(buf, "Hi");
char *myStr = buf;
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
For each code snippet below, can we modify the characters in `myStr`?

```c
char *otherStr = "Hi";
char *myStr = otherStr;
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
Memory Locations

For each code snippet below, can we modify the characters in `myStr`?

```c
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char buf[6];
    strcpy(buf, "Hi");
    myFunc(buf);
    return 0;
}
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
Q: Is there a way to check in code whether a string’s characters are modifiable?
A: No. This is something you can only tell by looking at the code itself and how the string was created.

Q: So then if I am writing a string function that modifies a string, how can I tell if the string passed in is modifiable?
A: You can’t! This is something you instead state as an assumption in your function documentation. If someone calls your function with a read-only string, it will crash, but that’s not your function’s fault :-)
String Behavior #5: We can set a char * equal to another value, because it is a reassign-able pointer.
A `char *` variable refers to a single character. We can reassign an existing `char *` pointer to be equal to another `char *` pointer.

```c
char *str = "apple";       // e.g. 0xffff0
char *str2 = "apple 2";    // e.g. 0xfe0
str = str2;                // ok! Both store address 0xfe0
```
Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```c
int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    char *ptr = str;
    ...
}
```
Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```c
int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    char *ptr = str;

    // equivalent
    char *ptr = &str[0];

    // confusingly equivalent, avoid
    char *ptr = &str;
    ...
}
```
String Behavior #6: Adding an offset to a C string gives us a substring that many places past the first character.
When we do pointer arithmetic, we are adjusting the pointer by a certain number of places (e.g. characters).

```
char *str = "apple";  // e.g. 0xff0
char *str2 = str + 1;  // e.g. 0xff1
char *str3 = str + 3;  // e.g. 0xff3

printf("%s", str);    // apple
printf("%s", str2);   // pple
printf("%s", str3);   // le
```
When we use bracket notation with a pointer, we are performing *pointer arithmetic and dereferencing*:

```c
char *str = "apple";  // e.g. 0xff0
// both of these add three places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff3.
char thirdLetter = str[3];  // 'l'
char thirdLetter = *(str + 3);  // 'l'
```
String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.
Strings as Parameters

When we pass a `char *` string as a parameter, C makes a *copy* of the address stored in the `char *` and passes it to the function. This means they both refer to the same memory location.

```c
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "apple";
    myFunc(str);
    ...
}
```
Strings as Parameters

When we pass a char array as a parameter, C makes a copy of the address of the first array element and passes it (as a char *) to the function.

```c
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    myFunc(str);
    ...
}
```

Address | Value
---|---
0x105 | '\0'
0x104 | 'e'
0x103 | 'l'
0x102 | 'p'
0x101 | 'p'
0x100 | 'a'
0xf | 0x100
Strings as Parameters

When we pass a char array as a parameter, C makes a copy of the address of the first array element and passes it (as a char *) to the function.

```c
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    // equivalent
    char *strAlt = str;
    myFunc(strAlt);
    ...
}
```

```
Address          Value
0x105            '\0'
0x104            'e'
0x103            'l'
0x102            'p'
0x101            'p'
0x100            'a'
```

```
main()  
str
0x100  'a'
0xf    0x100
...
```
Strings as Parameters

This means if we modify characters in `myFunc`, the changes will persist back in `main`!

```c
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    myFunc(str);
    printf("%s", str);  // apply
    ...
}
```

---

Address  | Value
----------|-------
0x100    | 'a'
0x101    | 'p'
0x102    | 'p'
0x103    | 'l'
0x104    | 'e'
0x105    | ' \0'
Strings as Parameters

This means if we modify characters in `myFunc`, the changes will persist back in `main`!

```c
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    myFunc(str);
    printf("%s", str);  // apply ...
}
```

Address Value

```
0x105 '\0'
0x104 'y'
0x103 'l'
0x102 'p'
0x101 'p'
0x100 'a'
... ...
```

STACK

```
... 0xf 0x100 ...
```
Strings In Memory

1. If we create a string as a `char[]`, we can modify its characters because its memory lives in our stack space.

2. We cannot set a `char[]` equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.

3. If we pass a `char[]` as a parameter, set something equal to it, or perform arithmetic with it, it’s automatically converted to a `char *`.

4. If we create a new string with new characters as a `char *`, we cannot modify its characters because its memory lives in the data segment.

5. We can set a `char *` equal to another value, because it is a reassign-able pointer.

6. Adding an offset to a C string gives us a substring that many places past the first character.

7. If we change characters in a string parameter, these changes will persist outside of the function.
Arrays vs. Pointers

• When you create an array, you are making space for each element in the array.
• When you create a pointer, you are making space for an 8 byte address.
• Arrays ”decay to pointers” when you perform arithmetic or pass as parameters.
• &arr does nothing on arrays, but &ptr on pointers gets its address
• sizeof(arr) gets the size of an array in bytes, but sizeof(ptr) is always 8
• Recap: Strings so far
• Searching in Strings
• **Practice:** Password Verification
• Buffer Overflows, Security and Valgrind
• Pointers
• Strings in Memory

Lecture 5 takeaway: C strings are pointers and arrays; understanding how pointers and arrays work help us better understand C string behavior. C strings are error-prone, and issues like buffer overflows can arise!

cp -r /afs/ir/class/cs107/lecture-code/lect5.
Extra Practice
1. Pointer arithmetic

```c
void func(char *str) {
    str[0] = 'S';
    str++;
    *str = 'u';
    str = str + 3;
    str[-2] = 'm';
}

int main(int argc, const char *argv[]) {
    char buf[] = "Monday";
    printf("before func: %s\n", buf);
    func(buf);
    printf("after func: %s\n", buf);
    return 0;
}
```

- Will there be a compile error/segfault?
- If no errors, what is printed?

• Draw memory diagrams!
• **Pointers** store addresses! Make up addresses if it helps your mental model.
### 1. Pointer arithmetic

```c
void func(char *str) {
    str[0] = 'S';
    str++;
    *str = 'u';
    str = str + 3;
    str[-2] = 'm';
}

int main(int argc, const char *argv[]) {
    char buf[] = "Monday";
    printf("before func: %s\n", buf);
    func(buf);
    printf("after func: %s\n", buf);
    return 0;
}
```

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.
What happens if we call `strncpy(buf, str, 5)`?
# 2. Code study: `strncpy`

The `strncpy()` function is similar, except that at most \( n \) bytes of `src` are copied. **Warning:** If there is no null byte among the first \( n \) bytes of `src`, the string placed in `dest` will not be null-terminated.

If the length of `src` is less than \( n \), `strncpy()` writes additional null bytes to `dest` to ensure that a total of \( n \) bytes are written.

A simple implementation of `strncpy()` might be:

```c
1 char *strncpy(char *dest, const char *src, size_t n) {
2     size_t i;
3     for (i = 0; i < n && src[i] != '\0'; i++)
4         dest[i] = src[i];
5     for (i < n; i++)
6         dest[i] = '\0';
7     return dest;
8 }
```

What happens if we call `strncpy(buf, str, 5)`?
### 3. char* vs char[] exercises

Suppose we use a variable `str` as follows:

<table>
<thead>
<tr>
<th></th>
<th>// initialize as below</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>str = str + 1;</td>
</tr>
<tr>
<td>B</td>
<td>str[1] = 'u';</td>
</tr>
<tr>
<td>C</td>
<td>printf(&quot;%s&quot;, str)</td>
</tr>
</tbody>
</table>

For each of the following initializations:
- Will there be a compile error/segfault?
- If no errors, what is printed?

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Code</th>
</tr>
</thead>
</table>
| 1.       | `char str[7];
strncpy(str, "Hello1");`                                           |
| 2.       | `char *str = "Hello2";`                                              |
| 3.       | `char arr[7];
strncpy(arr, "Hello3");
char *str = arr;`                                                          |
| 4.       | `char *ptr = "Hello4";
char *str = ptr;`                                                          |
3. char* vs char[] exercises

Suppose we use a variable `str` as follows:

```c
// initialize as below
A str = str + 1;
B str[1] = 'u';
C printf("%s", str)
```

For each of the following initializations:
- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];
   strcpy(str, "Hello1");`
   - Line A: Compile error (cannot reassign array)

2. `char *str = "Hello2";`
   - Line B: Segmentation fault (string literal)

3. `char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;
   Prints eu0103`

4. `char *ptr = "Hello4";
   char *str = ptr;`
   - Line B: Segmentation fault (string literal)
void tricky_addresses() {
    char buf[] = "Local";
    char *ptr1 = buf;
    char **double_ptr = &ptr1;
    printf("ptr1's value: %p\n", ptr1);
    printf("ptr1's deref: %c\n", *ptr1);
    printf("address: %p\n", &ptr1);
    printf("double_ptr value: %p\n", double_ptr);
    printf("buf's address: %p\n", &buf);

    char *ptr2 = &buf;
    printf("ptr2's value: %s\n", ptr2);
}

What is stored in each variable? (We cover double pointers more in Lecture 6)
4. Bonus: Tricky addresses

```c
void tricky_addresses() {
    char buf[] = "Local";
    char *ptr1 = buf;
    char **double_ptr = &ptr1;
    printf("ptr1's value: %p\n", ptr1);
    printf("ptr1's deref : %c\n", *ptr1);
    printf("address: %p\n", &ptr1);
    printf("double_ptr value: %p\n", double_ptr);
    printf("buf's address: %p\n", &buf);

    char *ptr2 = &buf;
    printf("ptr2's value: %s\n", ptr2);
}
```

While Line 10 raises a compiler warning, functionally it will still work—because pointers are addresses.