CS107 Spring 2019, Lecture 5
More C Strings

Reading: K&R (1.6, 5.5, Appendix B3) or Essential C section 3
Plan For Today

• Recap: String Operations
• Demo: Buffer Overflow and Valgrind
• Arrays of Strings
• Practice: Password Verification
• Pointers
• Announcements
• Strings in Memory
• Pointers to Strings
Plan For Today

- **Recap**: String Operations
- **Demo**: Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice**: Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings
C strings are arrays of characters, ending with a null-terminating character '\0'.

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'H'</td>
</tr>
<tr>
<td>1</td>
<td>'e'</td>
</tr>
<tr>
<td>2</td>
<td>'l'</td>
</tr>
<tr>
<td>3</td>
<td>'l'</td>
</tr>
<tr>
<td>4</td>
<td>'o'</td>
</tr>
<tr>
<td>5</td>
<td>','</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>'w'</td>
</tr>
<tr>
<td>8</td>
<td>'o'</td>
</tr>
<tr>
<td>9</td>
<td>'r'</td>
</tr>
<tr>
<td>10</td>
<td>'l'</td>
</tr>
<tr>
<td>11</td>
<td>'d'</td>
</tr>
<tr>
<td>12</td>
<td>'!'</td>
</tr>
<tr>
<td>13</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

String operations such as strlen use the null-terminating character to find the end of the string.
# Common `string.h` Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strlen(str)</code></td>
<td>returns the # of chars in a C string (before null-terminating character).</td>
</tr>
<tr>
<td><code>strcmp(str1, str2)</code></td>
<td>compares two strings; returns 0 if identical, &lt;0 if <code>str1</code> comes before <code>str2</code> in alphabet, &gt;0 if <code>str1</code> comes after <code>str2</code> in alphabet. <code>strncpy</code> stops comparing after at most <code>n</code> characters.</td>
</tr>
<tr>
<td><code>strncpy(str1, str2, n)</code></td>
<td></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 <code>n</code> 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 <code>n</code> characters. Always 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>
C Strings As Parameters

When you pass a string as a parameter, it is passed as a `char *`. You can still operate on the string the same way as with a `char[]`. *(We’ll see how today!)*

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

// can also write this, but it is really a pointer
int doSomething(char str[]) { ...
```
Buffer Overflows

• It is your responsibility to ensure that memory operations you 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
• **Recap:** String Operations
• **Demo:** Buffer Overflow and Valgrind
• **Arrays of Strings**
• **Practice:** Password Verification
• **Pointers**
• **Announcements**
• **Strings in Memory**
• **Pointers to Strings**
Arrays of Strings

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

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

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

```c
char *stringArray[] = {
    "my string 1",
    "my string 2",
    "my string 3"
};
```
Arrays of Strings

You can access each string using bracket syntax:

\[
\text{printf("%s\n", stringArray[0]);  // print out first string}
\]

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

```c
void myFunction(char **stringArray) {
    // equivalent to this, but it is really a double pointer
    void myFunction(char *stringArray[]) {
```
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 valid = verifyPassword("1572", "0123456789", invalidSubstrings, 1); // true
bool valid = verifyPassword("141234", "0123456789", invalidSubstrings, 1); // false
Practice: Password Verification
Plan For Today

• **Recap:** String Operations
• **Demo:** Buffer Overflow and Valgrind
• Arrays of Strings
• **Practice:** Password Verification
• **Pointers**
• Announcements
• Strings in Memory
• Pointers to Strings
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.
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!
    ...
}
```

STACK

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f0</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
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!
    ...
}
```

- **Value**: 2
- **Address**: 0x1f0
- **Stack Frame**: `main`: `int x = 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!
    ...
}
```
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!
    ...
}
```
Without pointers, we would make copies.

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!
    ...
}
```

---

**STACK**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f0</td>
<td>2</td>
</tr>
<tr>
<td>0x10</td>
<td>2</td>
</tr>
</tbody>
</table>

---

**Pointers**
Points

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!
    ...
}
```

STACK

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>3</td>
</tr>
<tr>
<td>0x1f0</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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!
    ...
}
```
Plan For Today

• **Recap:** String Operations
• **Demo:** Buffer Overflow and Valgrind
• Arrays of Strings
• **Practice:** Password Verification
• Pointers

• **Announcements**
• Strings in Memory
• Pointers to Strings
Announcements

• Assignment 0 grades released this afternoon
• Assignment 1 due Monday 4/15 11:59PM PST
  • Grace period until Wed. 4/17 11:59PM PST
• Lab 2: C strings practice
• Assignment 2 released at Assignment 1 due date
  • Due Mon. 4/22 11:59PM PST, grace period until Wed. 4/24 11:59PM PST
  • Programs using C strings
Plan For Today

• Recap: String Operations
• Demo: Buffer Overflow and Valgrind
• Arrays of Strings
• Practice: Password Verification
• Pointers
• Announcements
• Strings in Memory
• Pointers to Strings
Character Arrays

When you declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array.

```c
char str[6] = "apple";
```

An array variable refers to an entire block of memory. You cannot reassign an existing array to be equal to a new array.

```c
char str[6] = "apple";
char str2[8] = "apple 2";
str = str2;   // not allowed!
```

An array’s size cannot be changed once you create it; you must create another new array instead.
There is another convenient way to create a string if you do not need to modify it later. You 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 you 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”. You *cannot modify memory in this segment*.

char *str = "hi";

The pointer variable (e.g. str) refers to the *address of the first character of the string in the data segment*. 
A `char *` variable refers to a single character. You 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
```
You 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] = "apple";
    char *ptr = str;
    ...
}
```
Arrays and Pointers

You 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] = "apple";
    char *ptr = str;
    // equivalent
    char *ptr = &str[0];
    // confusingly equivalent, avoid
    char *ptr = &str;
    ...
}
```
• A char array is not a pointer; it refers to the entire array contents. In fact, sizeof returns the size of the entire array!

```c
char str[] = "Hello";
int arrayBytes = sizeof(str);  // 6
```

• A char pointer refers to the address of a single character. Since this variable is just a pointer, sizeof returns 8, no matter the total size of the string!

```c
char *str = "Hello";
int stringBytes = sizeof(str);  // 8
```
When you do pointer arithmetic (with either a pointer or an array), you are adjusting the pointer by a certain \textit{number of places} (e.g. characters).

```c
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
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xff0</td>
<td>'a'</td>
</tr>
<tr>
<td>0xff1</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff2</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff3</td>
<td>'l'</td>
</tr>
<tr>
<td>0xff4</td>
<td>'e'</td>
</tr>
<tr>
<td>0xff5</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
Pointer Arithmetic

Pointer arithmetic does not add bytes. Instead, it adds the size of the type it points to.

// nums points to an int array
int *nums = ... // e.g. 0xff0
int *nums2 = nums + 1; // e.g. 0xff4
int *nums3 = nums + 3; // e.g. 0xffc

printf("%d", *nums); // 52
printf("%d", *nums2); // 23
printf("%d", *nums3); // 34

STACK

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>16</td>
</tr>
<tr>
<td>0x1004</td>
<td>1</td>
</tr>
<tr>
<td>0xffc</td>
<td>34</td>
</tr>
<tr>
<td>0xff8</td>
<td>12</td>
</tr>
<tr>
<td>0xff4</td>
<td>23</td>
</tr>
<tr>
<td>0xff0</td>
<td>52</td>
</tr>
</tbody>
</table>

...
When you use bracket notation with a pointer, you are actually *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'
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff5</td>
<td>'\0'</td>
</tr>
<tr>
<td>0xff4</td>
<td>'e'</td>
</tr>
<tr>
<td>0xff3</td>
<td>'l'</td>
</tr>
<tr>
<td>0xff2</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff1</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff0</td>
<td>'a'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
When you 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 you 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] = "apple";
    myFunc(str);
    ...
}
```

![Address and Value Table]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>'a'</td>
</tr>
<tr>
<td>0x101</td>
<td>'p'</td>
</tr>
<tr>
<td>0x102</td>
<td>'p'</td>
</tr>
<tr>
<td>0x103</td>
<td>'l'</td>
</tr>
<tr>
<td>0x104</td>
<td>'e'</td>
</tr>
<tr>
<td>0x105</td>
<td>'0'</td>
</tr>
</tbody>
</table>
When you 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] = "apple";
    // equivalent
    char *arrPtr = str;
    myFunc(arrPtr);
    ...
}
```

Here's a stack diagram showing the layout of the variables and their addresses:

- **str**: `0x100`
- **arrPtr**: `0x101`
- **myFunc()**: `0x102`
- **myStr**: `0x103`
- **argv**: `0x104`
- **argc**: `0x105`
- **main()**: `...`

Each address contains a character:

- `0x100`: ‘a’
- `0x101`: ‘p’
- `0x102`: ‘p’
- `0x103`: ‘l’
- `0x104`: ‘e’
- `0x105`: ‘\0’
This means if you 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] = "apple";
    myFunc(str);
    printf("%s", str);  // apply ...
}
```
This means if you 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] = "apple";
    myFunc(str);
    printf("%s", str); // apply ...
}
```
Strings as Parameters

This also means we can no longer get the full size of the array using `sizeof`, because now it is just a regular `char *` pointer.

```c
void myFunc(char *myStr) {
    int size = sizeof(myStr); // 8
}

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

![Address Value Table]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>'a'</td>
</tr>
<tr>
<td>0x101</td>
<td>'p'</td>
</tr>
<tr>
<td>0x102</td>
<td>'p'</td>
</tr>
<tr>
<td>0x103</td>
<td>'l'</td>
</tr>
<tr>
<td>0x104</td>
<td>'e'</td>
</tr>
<tr>
<td>0x105</td>
<td>' \0'</td>
</tr>
</tbody>
</table>

...
Strings and Memory

These memory behaviors explain why strings behave the way they do:

1. If we make a variable to store a string literal that is a `char[]`, we can modify the characters because its memory lives in our stack space.

2. If we make a variable to store a string literal that is a `char *`, we cannot modify the characters because its memory lives in the data segment.

3. We can set a `char*` equal to another value, because it is just a pointer.

4. 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.

5. If we change characters in a string passed to a function, these changes will persist outside of the function.

6. When we pass a char array as a parameter, we can no longer use `sizeof` to get its full size.
Recap

• **Recap:** String Operations
• **Demo:** Buffer Overflow and Valgrind
• Arrays of Strings
• **Practice:** Password Verification
• Pointers
• **Announcements**
• Strings in Memory

**Next time:** Arrays and Pointers