CS107, Lecture 5
More C Strings

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

• Searching in Strings
• **Practice:** Password Verification
• **Demo:** Buffer Overflow and Valgrind
• Pointers
• Strings in Memory
Lecture Plan

• Searching in Strings

• Practice: Password Verification

• Demo: Buffer Overflow and Valgrind

• Pointers

• Strings in Memory
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>'w'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</td>
<td>'!'</td>
<td>'\0'</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</td>
</tr>
<tr>
<td><code>strncmp(str1, str2, n)</code></td>
<td><code>str2</code> in alphabet, &gt;0 if <code>str1</code> comes after <code>str2</code> in alphabet. <code>strncmp</code></td>
</tr>
<tr>
<td></td>
<td>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>,</td>
</tr>
<tr>
<td><code>strrchr(str, ch)</code></td>
<td>or <code>NULL</code> if <code>ch</code> was not found in <code>str</code>. <code>strrchr</code> find the last occurrence.</td>
</tr>
<tr>
<td><code>strstr(haystack, needle)</code></td>
<td>string search: returns a pointer to the start of the first occurrence of</td>
</tr>
<tr>
<td></td>
<td><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.</td>
</tr>
<tr>
<td><code>strncpy(dst, src, n)</code></td>
<td>Assumes enough space in <code>dst</code>. Strings must not overlap. <code>strncpy</code> stops</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td><code>strncat(dst, src, n)</code></td>
<td>after at most <code>n</code> characters. <strong>Always</strong> 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</td>
</tr>
<tr>
<td><code>strcspn(str, reject)</code></td>
<td>characters in <code>accept</code>. <code>strcspn</code> returns the length of the initial part of</td>
</tr>
<tr>
<td></td>
<td><code>str</code> which does <strong>not</strong> contain any characters in <code>reject</code>.</td>
</tr>
</tbody>
</table>
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 daisy[6];
strcpy(daisy, "Daisy");
char *letterA = strchr(daisy, 'a');
printf("%s\n", daisy);  // Daisy
printf("%s\n", letterA);  // aisy
```

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

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

\begin{verbatim}
char daisy[10];
strcpy(daisy, "Daisy Dog");
char *substr = strstr(daisy, "Dog");
printf("%s\n", daisy); // Daisy Dog
printf("%s\n", substr); // Dog
\end{verbatim}

If there are multiple occurrences of the string, \texttt{strstr} returns a pointer to the \textit{first} one.
strspn returns the *length* of the initial part of the first string which contains only characters in the second string.

```c
char daisy[10];
strncpy(daisy, "Daisy Dog");
int spanLength = strspn(daisy, "aDeoi"); // 3
```

“How many places can we go in the first string before I encounter a character *not in* the second string?”
strcpy(daisy, "Daisy Dog");
int spanLength = strcspn(daisy, "driso");  // 2

“How many places can we go in the first string before I encounter a character in the second string?”
C Strings As Parameters

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"
};
```
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[]) {
```
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

• Searching in Strings
• **Practice**: Password Verification
• **Demo**: Buffer Overflow and Valgrind
• Pointers
• Strings in Memory
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

• Searching in Strings
• Practice: Password Verification
• Demo: Buffer Overflow and Valgrind
• Pointers
• Strings in Memory
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.
### Memory

- 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>261</td>
<td>\0</td>
</tr>
<tr>
<td>260</td>
<td>e</td>
</tr>
<tr>
<td>259</td>
<td>l</td>
</tr>
<tr>
<td>258</td>
<td>p</td>
</tr>
<tr>
<td>257</td>
<td>p</td>
</tr>
<tr>
<td>256</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Looking Back at C++

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

\begin{verbatim}
void myFunc(int& num) {
    num = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 3!
    ...
}
\end{verbatim}
• 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 \texttt{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.

```
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.

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.

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!
    ...
}
```
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!
    ...
}
```
Pointers Summary

• 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.
Pointers

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>
</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!
    ...
}
```
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>
</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!
    ...
}
```
Lecture Plan

• Searching in Strings
• **Practice:** Password Verification
• **Demo:** Buffer Overflow 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");
```

<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>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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.
Character Arrays

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?

**A.** 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?

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?

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

A. A copy of part of the array `local_str`
B. A pointer containing an address to the third element in `local_str`
String Parameters

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?
Memory Locations

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?
For each code snippet below, can we modify the characters in \texttt{myStr}?

\begin{verbatim}
void myFunc(char *myStr) {
  ...
}

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

\textbf{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. 0xfee0
str = str2;   // ok! Both store address 0xfee0
```

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;
    ...
}
```
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).

```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>0xff0</td>
<td>'a'</td>
</tr>
<tr>
<td>0xff1</td>
<td>'l'</td>
</tr>
<tr>
<td>0xff2</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff3</td>
<td>'1'</td>
</tr>
<tr>
<td>0xff4</td>
<td>'e'</td>
</tr>
<tr>
<td>0xff5</td>
<td>'\0'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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);
    ...
}
```
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);
    ...
}
```
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) {
    ...
}
```

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

• Searching in Strings
• **Practice:** Password Verification
• **Demo:** Buffer Overflow and Valgrind
• Pointers
• Strings in Memory

**Next time:** Arrays and Pointers
Extra Practice
char* vs char[] exercises

Suppose we use a variable `str` as follows:

```c
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

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

1. `char str[7];
   strcpy(str, "Hello1");`
2. `char *str = "Hello2";`
3. `char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;`
4. `char *ptr = "Hello4";
   char *str = ptr;`
char* vs char[] exercises

Suppose we use a variable `str` as follows:

```c
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

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

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

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

3. `char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;`
   Prints `euIo3`

4. `char *ptr = "Hello4";`
   `char *str = ptr;`
   Segmentation fault (string literal)