CS107, Lecture 8
C Strings and Valgrind

Reading: K&R (5.2-5.5) or Essential C section 6

CS107 Topic 2

How can a computer represent and manipulate more complex data like text?

Why is answering this question important?

- Shows us how strings are represented in C and other languages (previously)
- Helps us better understand buffer overflows, a common bug (this 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

- Learn more about the risks of buffer overflows and how to mitigate them
- Understand more about how strings are represented in memory and how that helps us better understand their behavior
- Become familiar with using memory diagrams to understand code behavior

Lecture Plan

- **Recap**: C Strings and Buffer Overflows
- Buffer Overflows, Security and Valgrind
- Debugging and Testing
- More about C Strings

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

Lecture Plan

- **Recap: C Strings and Buffer Overflows**
- Buffer Overflows, Security and Valgrind
- Debugging and Testing
- More about C Strings

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

Common `string.h` Functions

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

Buffer Overflow Impacts

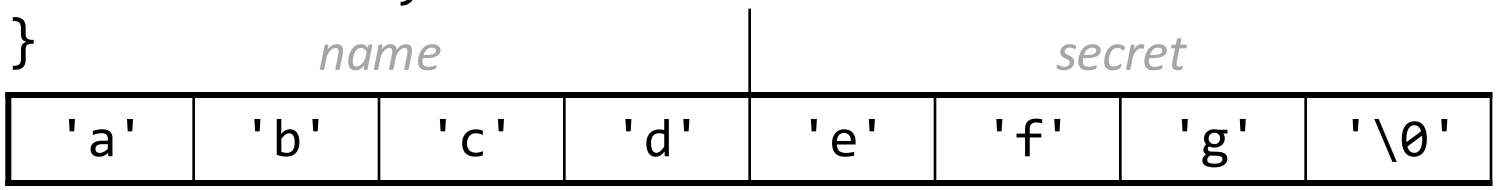
Buffer overflows are not merely functionality bugs; they can cause a range of unintended behavior:

- Let the user access memory they shouldn't be able to access
- Let the user modify memory they shouldn't be able to access
  - Change a value that is used later in the program
  - User changes the program to execute their own custom instructions instead
  - 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 our code.

Buffer Overflow Example: ./buf

```
// first argument is the name, second is the password
int main(int argc, char *argv[]) {
    char secret[4] = "123";
    // assume secret comes right after name in memory
    // (this is not always true)
    char name[4];
    strcpy(name, argv[1]);

    if (!strcmp(secret, argv[2])) {
        printf("You're in!\n");
    }
    return 0;
}
```



Which of these arguments would cause the program to print “You’re in!”?

A. **./buf abcdefg efg**
B. ./buf abcd abcd
C. ./buf a a
D. ./buf abcdefgh abcd

Lecture Plan

- **Recap**: C Strings and Buffer Overflows
- **Buffer Overflows, Security and Valgrind**
- Debugging and Testing
- More about C Strings

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

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.
  - Valgrind can detect some, but not all, stack-memory-related issues

Demo: Memory Errors

`memory_errors.c`

Debugging Tools Summary

GDB

```
gdb myprogram
…
(gdb) run [args]
```

- Pause, step through, and print out values during program execution
- Helpful to track down bugs like crashes, infinite loops, functionality discrepancies, etc.

Valgrind

```
valgrind myprogram [args]
```

- Observes program execution without interrupting it, prints findings.
- Helpful (to a certain extent) to track down memory-related issues like buffer overflows, reading uninitialized memory, etc.

Lecture Plan

- **Recap:** C Strings and Buffer Overflows
- Buffer Overflows, Security and Valgrind
- **Debugging and Testing**
- More about C Strings

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

Debugging

GDB and Valgrind will be essential tools throughout the quarter to find and squash bugs.

- Emphasis on both tools and the debugging process: http://cs107.stanford.edu/resources/debugging.html
  - We'll be relying on this checklist going forward for debugging – use the tips here as you work and come ask questions!
- Debugging can certainly be frustrating sometimes! Checklist emphasizes a methodical and systematic process that will save time in the long run.
- Debugging also where we learn a lot about how our code works

How do you find bugs? **Testing!**

Testing

Testing helps surface bugs and track progress in implementing a program.

Opaque-box testing: writing tests only considering the specification of what the program should do, without considering the implementation details/actual code

Clear-box testing: writing tests relying on knowledge of the design internals and code paths such as internal special cases, code structure, etc.

http://cs107.stanford.edu/testing.html

- *Test case size*: small tests help early on to catch bugs; large tests help later to stress test the system.

Aim to write tests throughout the development of a program.

- **Before writing the program:** documents intended behavior, avoids code assumptions
- **During writing the program:** add additional test cases as needed

Goal: incremental development, testing at each step.

Lecture Plan

- **Recap**: C Strings and Buffer Overflows
- Buffer Overflows, Security and Valgrind
- Debugging and Testing
- **More about C Strings**

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

More about C Strings

C strings can be represented as **char[]** or **char ***.

- When we create **char[]**, we are creating space for characters
- When we create **char ***, we are creating space for an address of character(s)
- Strings are implicitly converted to char * when passed as parameters
  - E.g. all string functions take char * parameters, but accept char[]
- Both **char[]** and **char *** work as strings - both let us access characters using **[]**, and both can be passed as parameters to string functions since they are passed as a **char *** when passed in anyway.
- A char * is technically a pointer to a **single character**. But we commonly use char * as string by having the character it points to be followed by more characters and ultimately a null terminator. But a char * could also just point to a single character (not a string).

char * vs. char[]

char *	char[]
8 bytes big, stores an address. Create one when you want to create a string without creating new characters.	Any # of bytes big, stores characters. Create one when you want to create a string with new space for its characters.

char * vs. char[]

char *	char[]
8 bytes big, stores an address. Create one when you want to create a string without creating new characters.	Any # of bytes big, stores characters. Create one when you want to create a string with new space for its characters.
We can use [] to get individual characters	We can use [] to get individual characters

char * vs. char[]

char *	char[]
8 bytes big, stores an address. Create one when you want to create a string without creating new characters.	Any # of bytes big, stores characters. Create one when you want to create a string with new space for its characters.
We can use [] to get individual characters	We can use [] to get individual characters
We can use it with string functions like **strlen** – string functions take char *s as parameters.	We can use it with string functions like **strlen** – will be passed as a char *.

char * vs. char[]

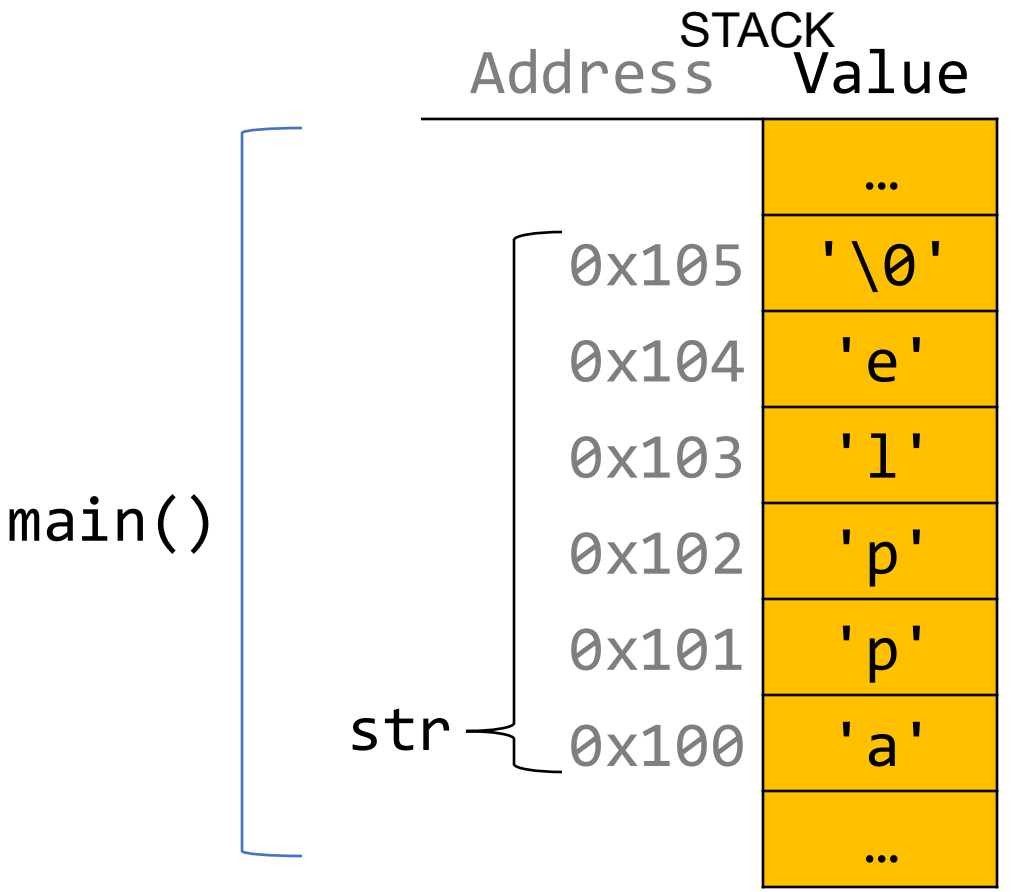
char *	char[]
8 bytes big, stores an address. Create one when you want to create a string without creating new characters.	Any # of bytes big, stores characters. Create one when you want to create a string with new space for its characters.
We can use [] to get individual characters and add to it to get a pointer further in the string.	We can use [] to get individual characters and add to it to get a pointer further in the string.
We can use it with string functions like **strlen** – string functions take char *s as parameters.	We can use it with string functions like **strlen** – will be passed as a char *.

We can set a char * pointer equal to another value later to make it point somewhere different. E.g.
`char *str = buf;`
`// later`
`str = otherBuf;`
	We can't set an array equal to another value after creating it.
	`char buf[6];`
	`strcpy(buf, "Hello");`
	~~`buf = "Hi";`~~ `// not allowed`
	`strcpy(buf, "Hi");  // ok`

char[]

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

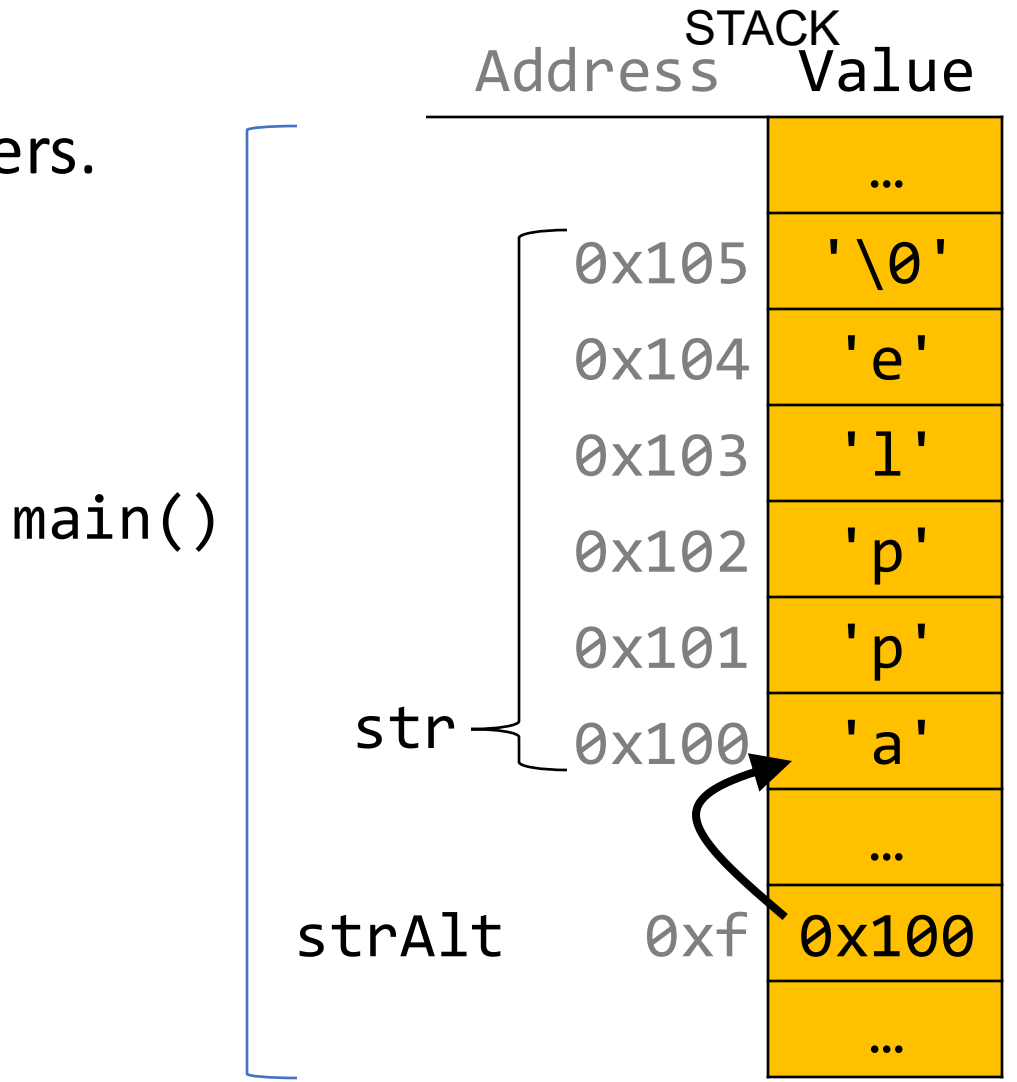
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      ...
```



char *

When we declare a **char** *, we allocate space on the stack to store an address, not actual characters. But we can still generally use **char** * the same as **char[]**.

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



char[] and char * - What is Printed?

```
char buf[6];
strcpy(buf, "Hello");
char *ptr = buf;
char *ptr2 = ptr;
char buf2[6];
strcpy(buf2, ptr2);
char *ptr3 = buf2;
ptr3[0] = 'M';
printf("%s\n", buf);
```

Respond on PollEv:
pollev.com/cs107

What would be printed?

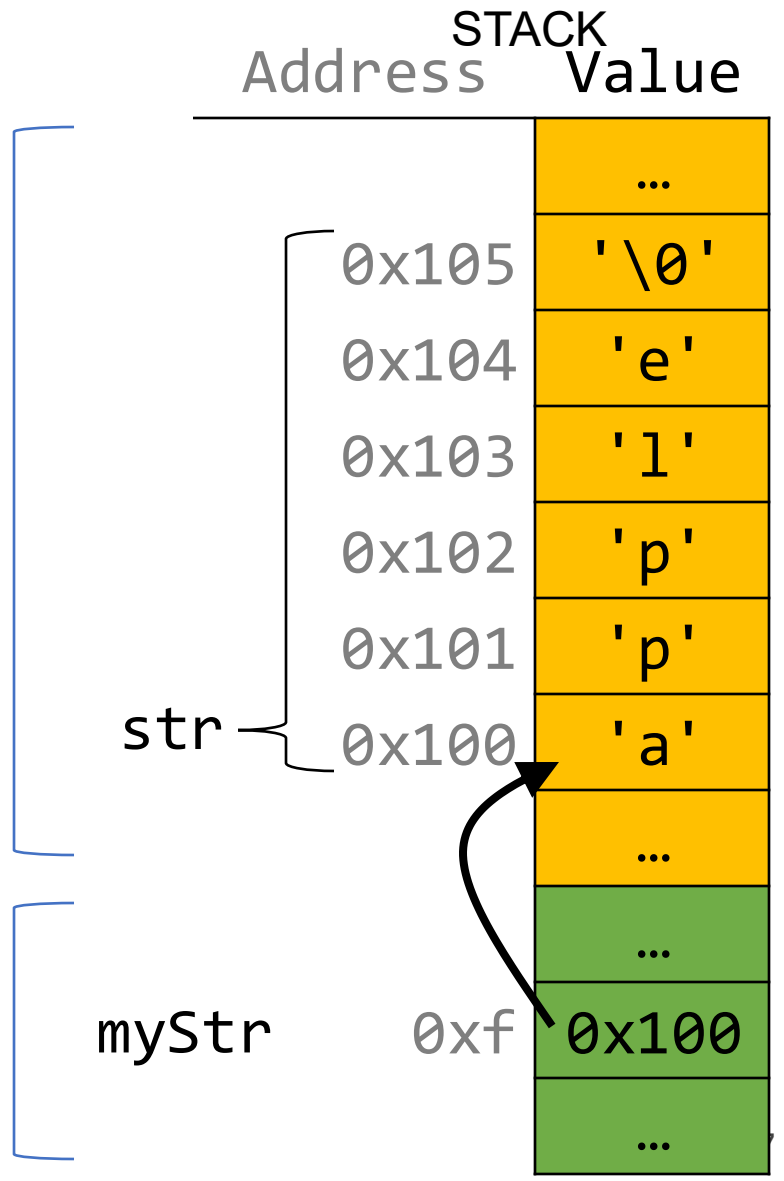
Strings In Memory

1. 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.
2. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
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 change characters in a string parameter, these changes will persist outside of the function.

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.

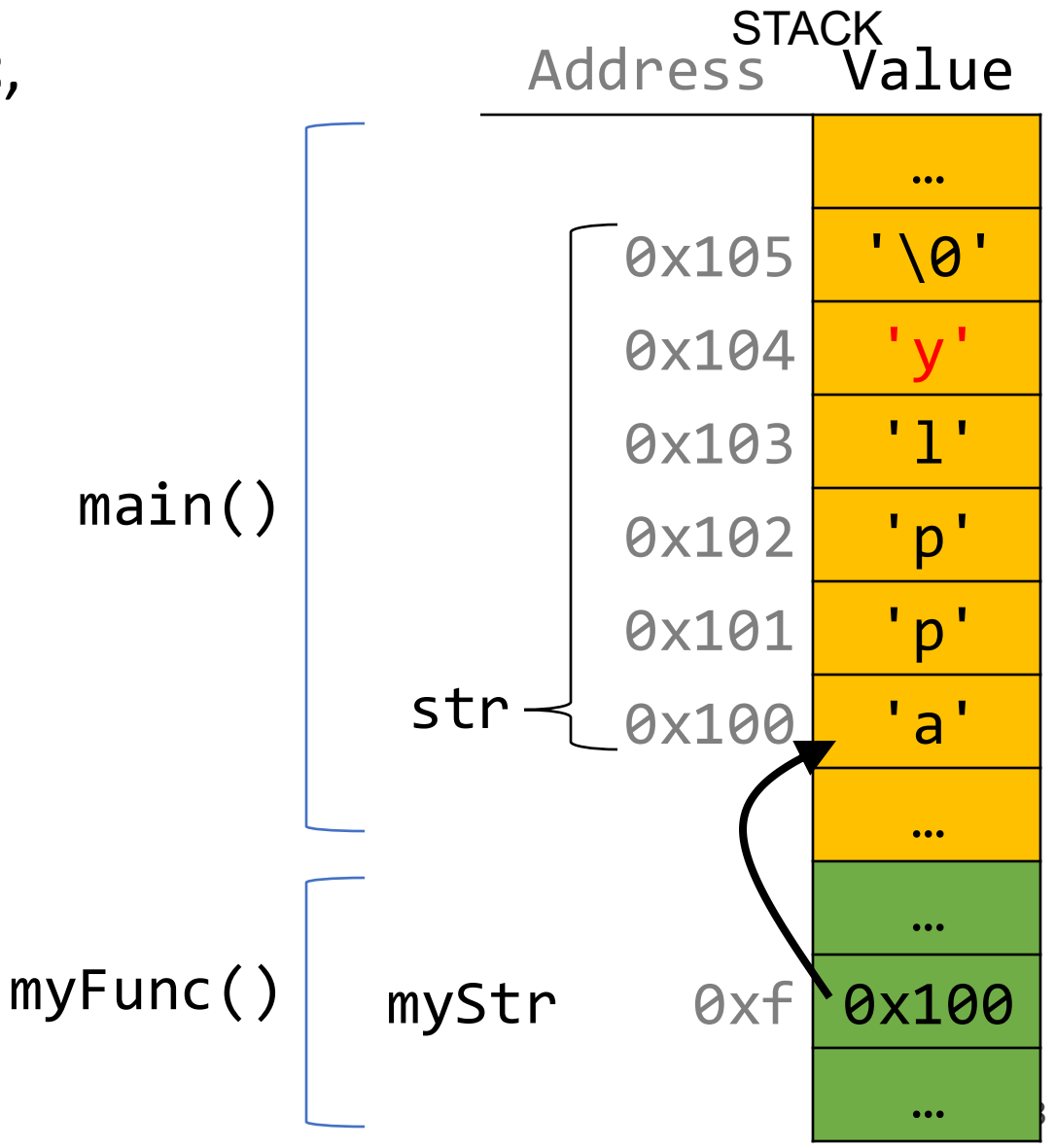
```
void myFunc(char *myStr) {
      ...
}

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



Strings as Parameters

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

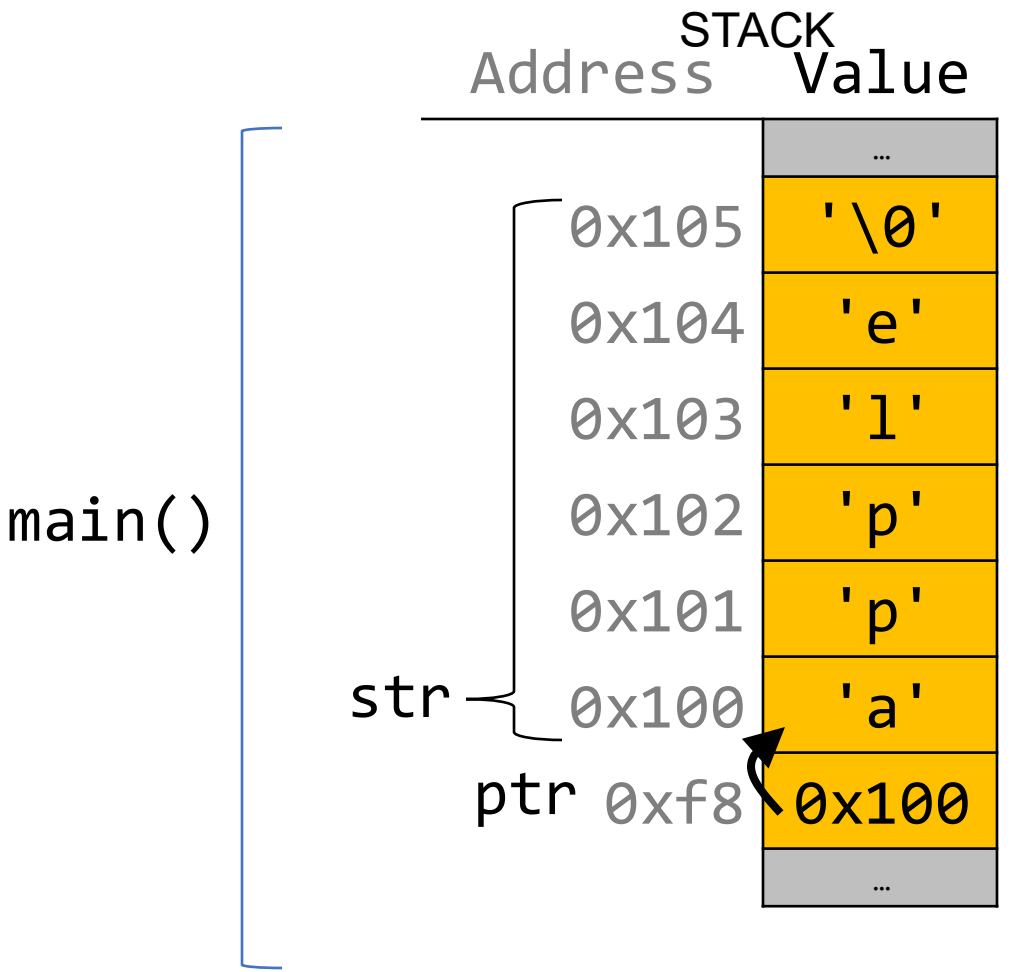
```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

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



Arrays and Pointers

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

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



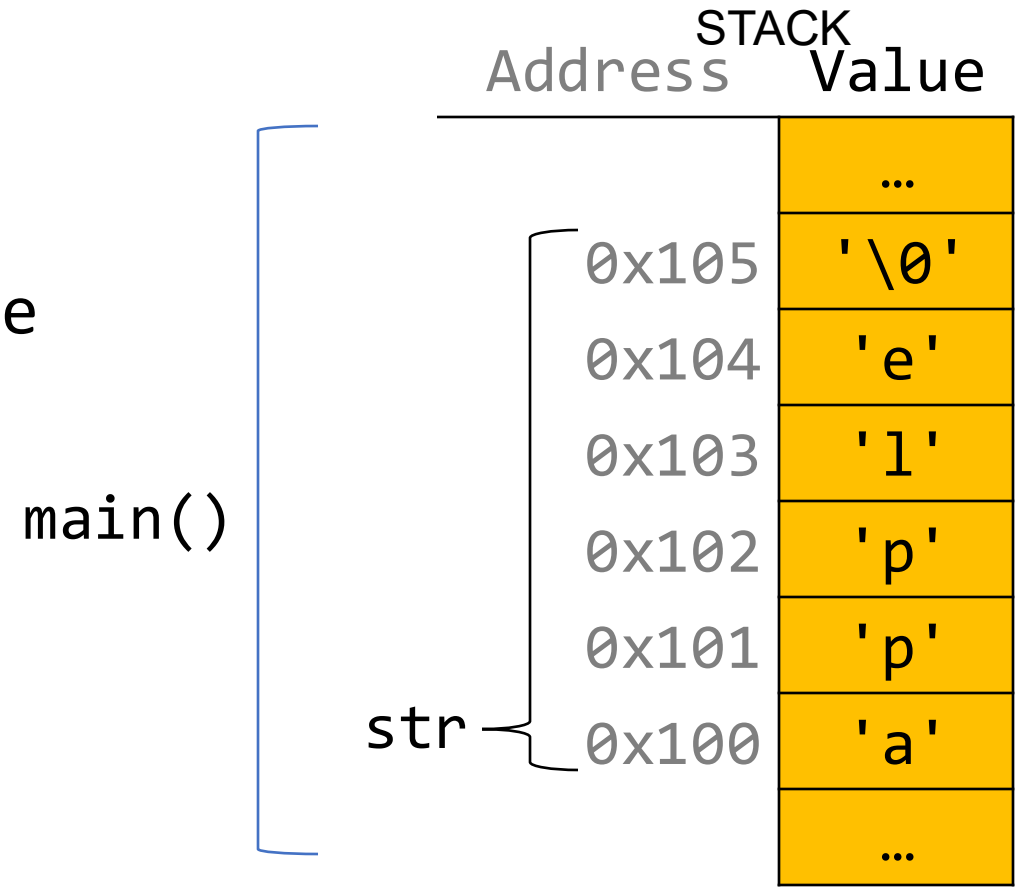
Strings In Memory

1. 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.
2. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
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 change characters in a string parameter, these changes will persist outside of the function.
5. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
6. We can also create a "read-only string" – we can use it just like any other string, but we cannot modify its characters because its memory lives in the data segment.

char[]

When we declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array. We can change those characters because they live on the stack.

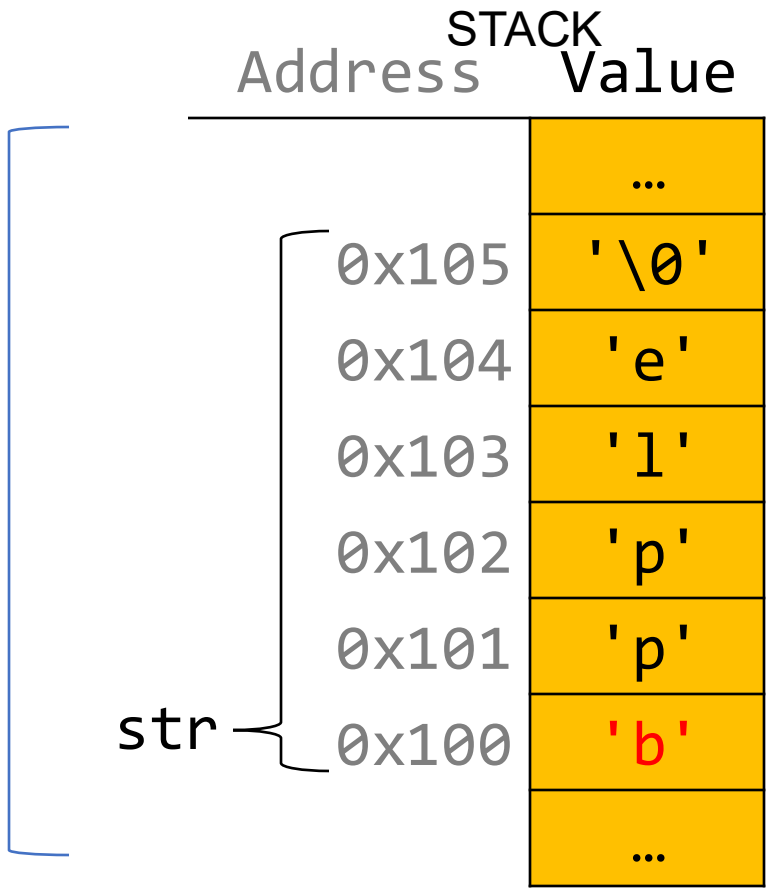
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      ...
```



char[]

When we declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array. We can change those characters because they live on the stack.

```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      str[0] = 'b';
      ...
```



Read-only Strings

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.

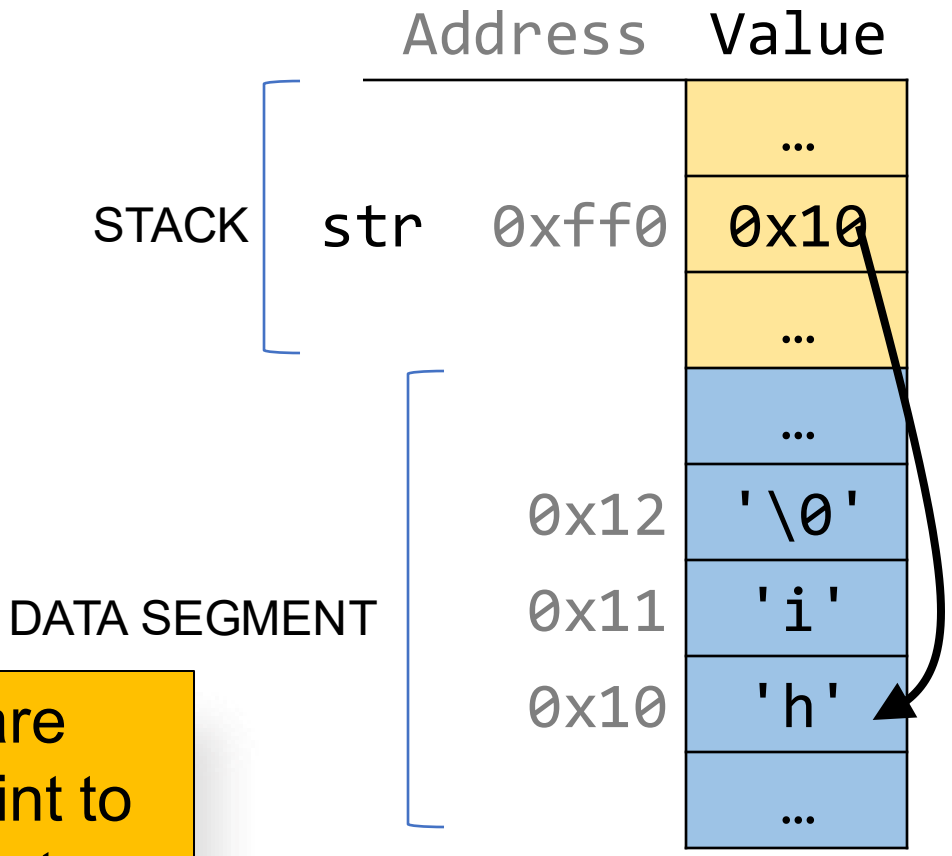
```
char *myString = "Hello, world!";
...
printf("%s", myString);              // Hello, world!
```

Read-only Strings

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*.

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

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



NOTE: not all char * strings are read-only. Only ones that point to characters in the data segment are read-only.

Read-only Strings

Read-only strings are convenient to use, but make sure to not use a read-only string in code that tries to modify its characters – it will crash!

```
char *myString = "Hello, world!";
myString[0] = 'h';  // crashes!
```

There's no way in code to check if a string is read-only; it's up to the programmer to properly use strings to avoid these crashes.

- E.g. don't pass in a read-only string as the **dst** to **strcpy**

```
strcpy(myString, "Hi");  // crashes!
```

Read-only Strings

A string is read-only if it points to characters that live in the data segment, rather than memory we can modify.

```
char *readOnly = "Hi";

char modifiable[6];
strcpy(modifiable, "Hi");

// is ptr read-only?
char *ptr = modifiable;
// no, because it points to characters on the stack
ptr[0] = 'h';  // ok!
```

Strings In Memory

1. 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.
2. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
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 change characters in a string parameter, these changes will persist outside of the function.
5. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
6. We can also create a "read-only string" – we can use it just like any other string, but we cannot modify its characters because its memory lives in the data segment.
7. Adding an offset (some numeric amount) to a C string gives us a substring that many places past the first character.

Adding Offsets to Strings

When you add an offset to a string, you are adjusting it by a certain number of characters.

```
char *str = "apple";
char *str1 = str + 1;
char *str3 = str + 3;

printf("%s", str);
printf("%s", str1);
printf("%s", str3);
```

Recap

- **Recap:** C Strings and Buffer Overflows
- Buffer Overflows, Security and Valgrind
- Debugging and Testing
- More about C Strings

Lecture 8 takeaway: C strings are pointers and arrays. C strings are error-prone, and issues like buffer overflows can arise! Valgrind is a tool that can help detect memory errors.

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```