

# CS107, Lecture 8

## C Strings, Valgrind and Pointers

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



masks recommended

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

# 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 (previously)
- 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 how strings are represented as pointers and how that helps us better understand their behavior
- Learn about pointers and how they help us access data without making copies
- Become familiar with using memory diagrams to understand code behavior

# Lecture Plan

- Buffer Overflows, Security and Valgrind
- Review: Pointers
- Strings and Pointers

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

# Lecture Plan

- **Buffer Overflows, Security and Valgrind**
- Review: Pointers
- Strings and Pointers

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

# 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

```
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

name				secret			
'a'	'b'	'c'	'd'	'e'	'f'	'g'	'\0'

# Buffer Overflow Impacts

- AOL instant messenger buffer overflow: allowed remote attackers to execute code: <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://www.zdnet.com/article/the-morris-worm-internet-malware-turns-25/>



# 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 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](https://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**



# Memory Safe Systems Programming

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

- Rust (Mozilla)
- Go (Google)
- Project Verona (Microsoft)

# Association for Computing Machinery (ACM) Code of Ethics

## ACM Code of Ethics and Professional Conduct

### 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 applying the principle.

<https://www.acm.org/code-of-ethics>

#### On This Page

Preamble

1. GENERAL PRINCIPLES

1.1 Contribute to society and the well-being, while avoiding harm to others

1.2 Avoid harm to others

1.3 Be honest and trustworthy

1.4 Be fair and avoid discrimination

1.5 Respect the privacy and confidentiality of others

# ACM Code of Ethics on Security

## 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.

# 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](https://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

# Lecture Plan

- Buffer Overflows, Security and Valgrind
- **Review: Pointers**
- Strings and Pointers

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

# Strings and Pointers

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[]**
- 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).

# Pointers and Memory

A *pointer* is a variable that stores a memory address.

- Memory is a big array of bytes, and each byte has a unique numeric index that is commonly written in hexadecimal. A pointer stores one of these “indexes”.
- 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.
- Pointers are also essential for allocating memory on the heap, and to refer to memory generically, both of which we will cover later.

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



# Pass By Value

When you pass a value as a parameter, C passes a copy of that value.

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

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

main()



STACK	
Address	Value
x    0x1f0	...
	2
	...

# Pass By Value

When you pass a value as a parameter, C passes a copy of that value.

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

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

main()



STACK		
Address		Value
		...
x	0x1f0	2
		...

# Pass By Value

When you pass a value as a parameter, C passes a copy of that value.

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

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

main()  
myFunc()

STACK		
Address		Value
	x	0x1f0
		2
		...
	val	0x10
		2
		...

# Pass By Value

When you pass a value as a parameter, C passes a copy of that value.

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

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

main()  
myFunc()

STACK		
Address		Value
	x	0x1f0
		2
		...
	val	0x10
		2
		...

# Pass By Value

When you pass a value as a parameter, C passes a copy of that value.

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

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

main()  
myFunc()

STACK		
Address		Value
	x	0x1f0
		2
		...
	val	0x10
		3
		...

# Pass By Value

When you pass a value as a parameter, C passes a copy of that value.

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

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

main()



STACK	
Address	Value
x	...
	2
	...

# Pointers

Pointers allow us to pass around the *location* of data so that the original data can be modified in other functions.

**Example:** I want to write a function *myFunc* that can change the value of an existing integer to be 3.

```
int main(int argc, char *argv[]) {  
    int x = 2;  
    myFunc(???);  
    printf("%d", x);    // want to print 3  
    ...  
}
```

# Pointers

```
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
```

If **declaration**: "pointer"

ex: int \* is "pointer to an int"

\*

If **operation**: "dereference/the value at address"

ex: \*num is "the value at address num"



# Pointers

A pointer lets us pass where a particular instance of data is, so it can be modified.

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

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

main()



STACK	
Address	Value
x    0x1f0	...
	2
	...

# Pointers

A pointer lets us pass where a particular instance of data is, so it can be modified.

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

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

main()



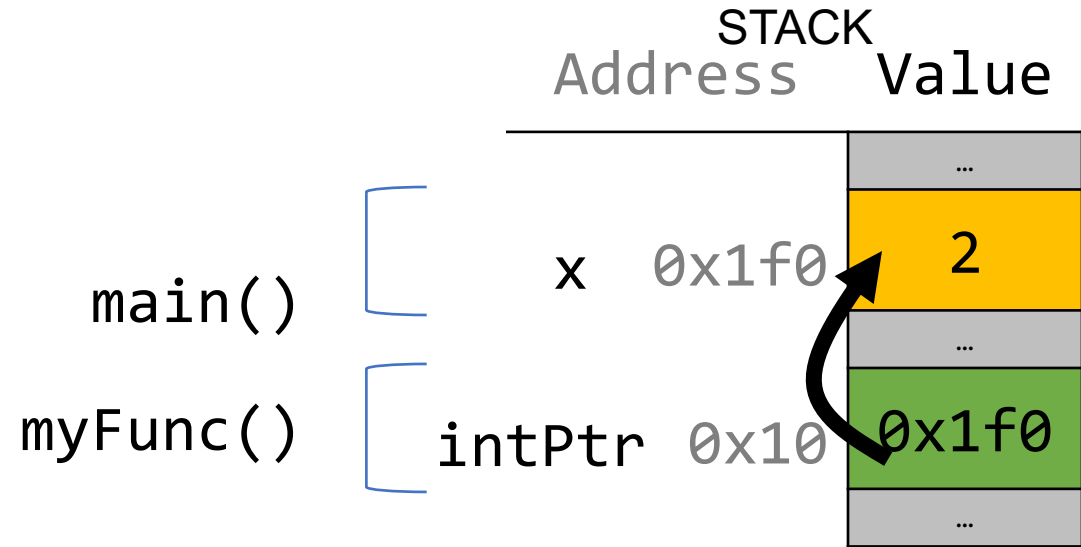
STACK	
Address	Value
x    0x1f0	...
	2
	...

# Pointers

A pointer lets us pass where a particular instance of data is, so it can be modified.

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

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

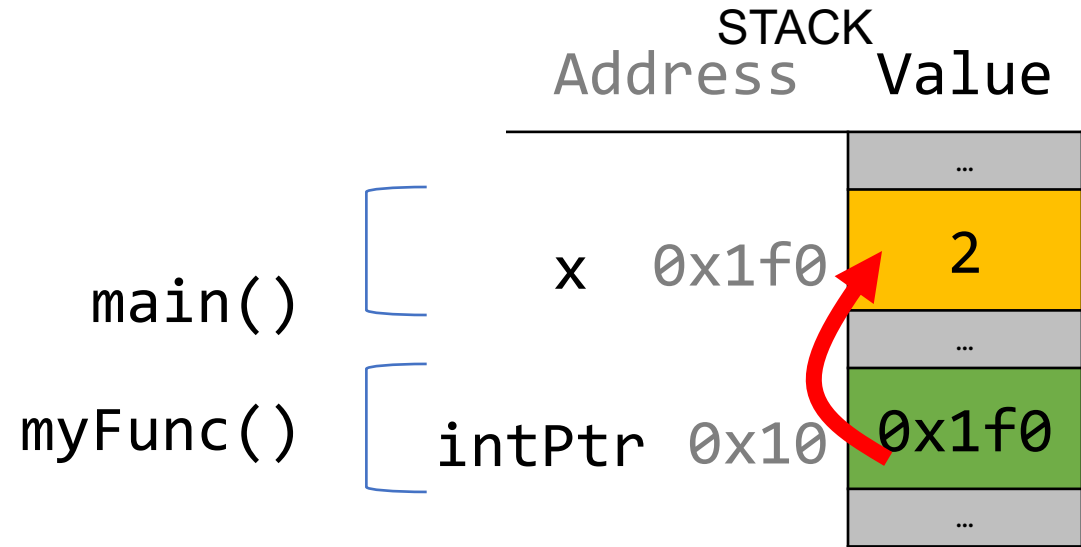


# Pointers

A pointer lets us pass where a particular instance of data is, so it can be modified.

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

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

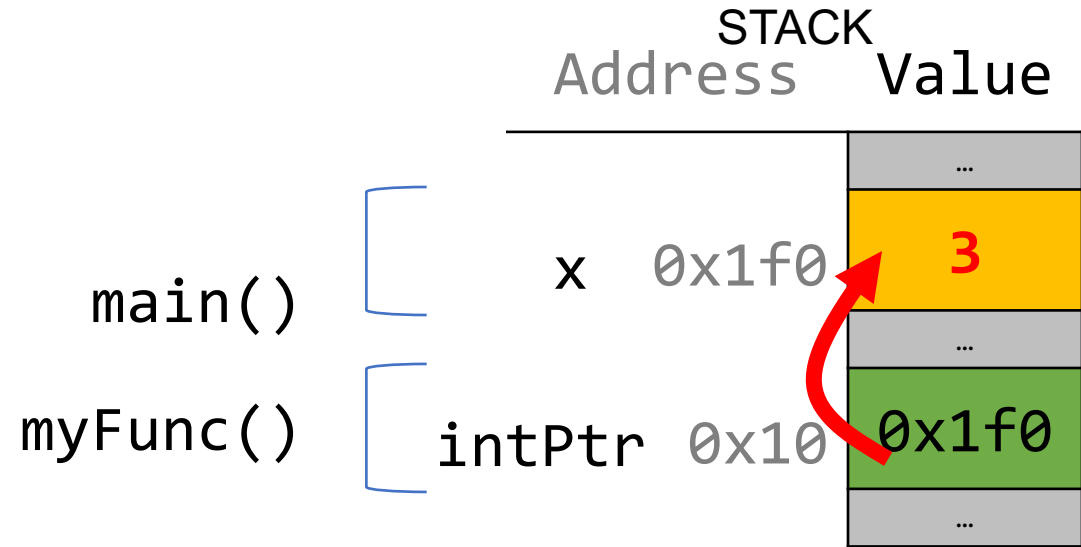


# Pointers

A pointer lets us pass where a particular instance of data is, so it can be modified.

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

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



# Pointers

A pointer lets us pass where a particular instance of data is, so it can be modified.

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

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

main()



STACK	
Address	Value
x    0x1f0	...
	3
	...

# C Parameters

- If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.
- If you are modifying a specific instance of some value, pass the *location* of what you would like to modify and dereference that location to access what's there.

Do I care about modifying *this* instance of my data? If so, I need to pass where that instance lives, as a parameter, so it can be modified.

# Pointers Practice

```
void makeUpper(char *ptr) {  
    __1__ = toupper(__2__);  
}  
  
int main(int argc, char *argv[]) {  
    char ch = 'h';  
  
    // want to modify ch to be capital  
    makeUpper(__3__);  
    printf("%c\n", ch); // should print 'H'  
    return 0;  
}
```

**What should go in each of the blanks** so that this code correctly modifies **ch** to be capitalized?

**Respond with your thoughts on PollEv:** [pollev.com/cs107](https://pollev.com/cs107) or text CS107 to 22333 once to join.





**What should go in each of the blanks so that this code correctly modifies ch to be capitalized?**

1: \*ptr, 2: ptr, 3: &ptr

1: ptr, 2: \*ptr, 3: &ptr

1: \*ptr, 2: \*ptr, 3: &ptr

1: &ptr, 2: \*ptr, 3: \*ptr

# Pointers Practice

```
void makeUpper(char *ptr) {  
    *ptr = toupper(*ptr);  
}  
  
int main(int argc, char *argv[]) {  
    char ch = 'h';  
  
    // want to modify ch to be capital  
    makeUpper(&ch);  
    printf("%c\n", ch); // should print 'H'  
    return 0;  
}
```

**What should go in each of the blanks** so that this code correctly modifies **ch** to be capitalized?



# Lecture Plan

- Buffer Overflows, Security and Valgrind
- Review: Pointers
- **Strings and Pointers**

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

# 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");  
    ...  
}
```

main()

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

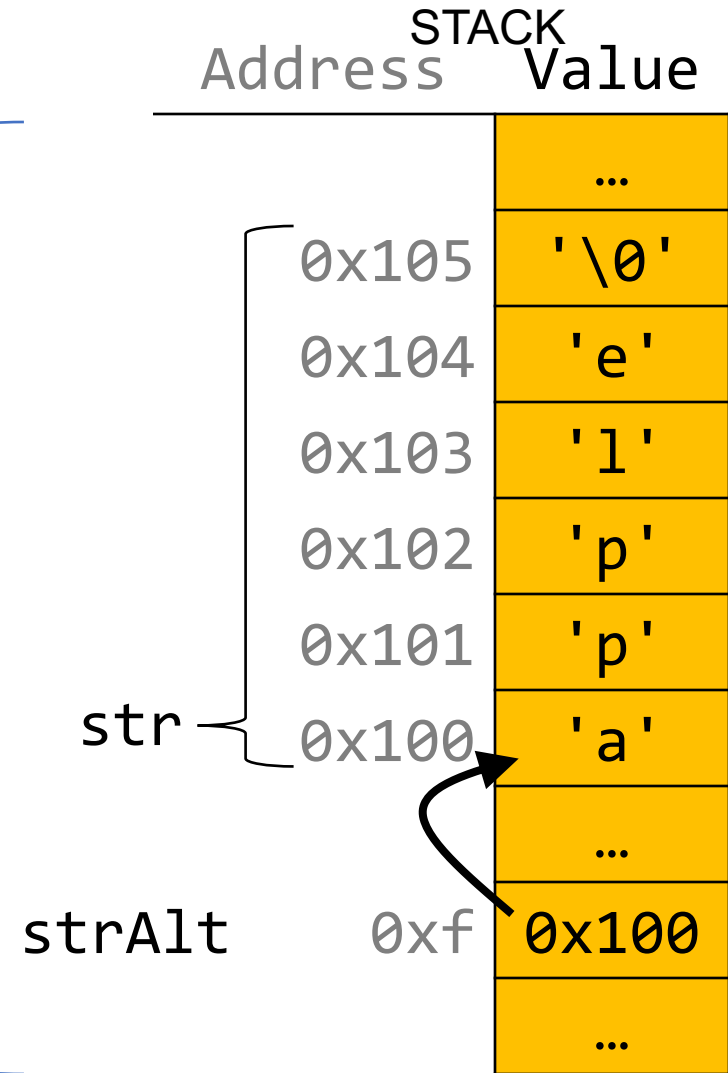
Diagram illustrating the memory layout of the `str` array on the stack. The array is located at addresses 0x100 to 0x105. The values stored are 'a', 'p', 'p', 'l', 'e', and '\0' (null terminator). The label `str` points to the start of the array at address 0x100. A bracket labeled `main()` indicates the function's stack frame.

# 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;  
    ...  
}
```

main()

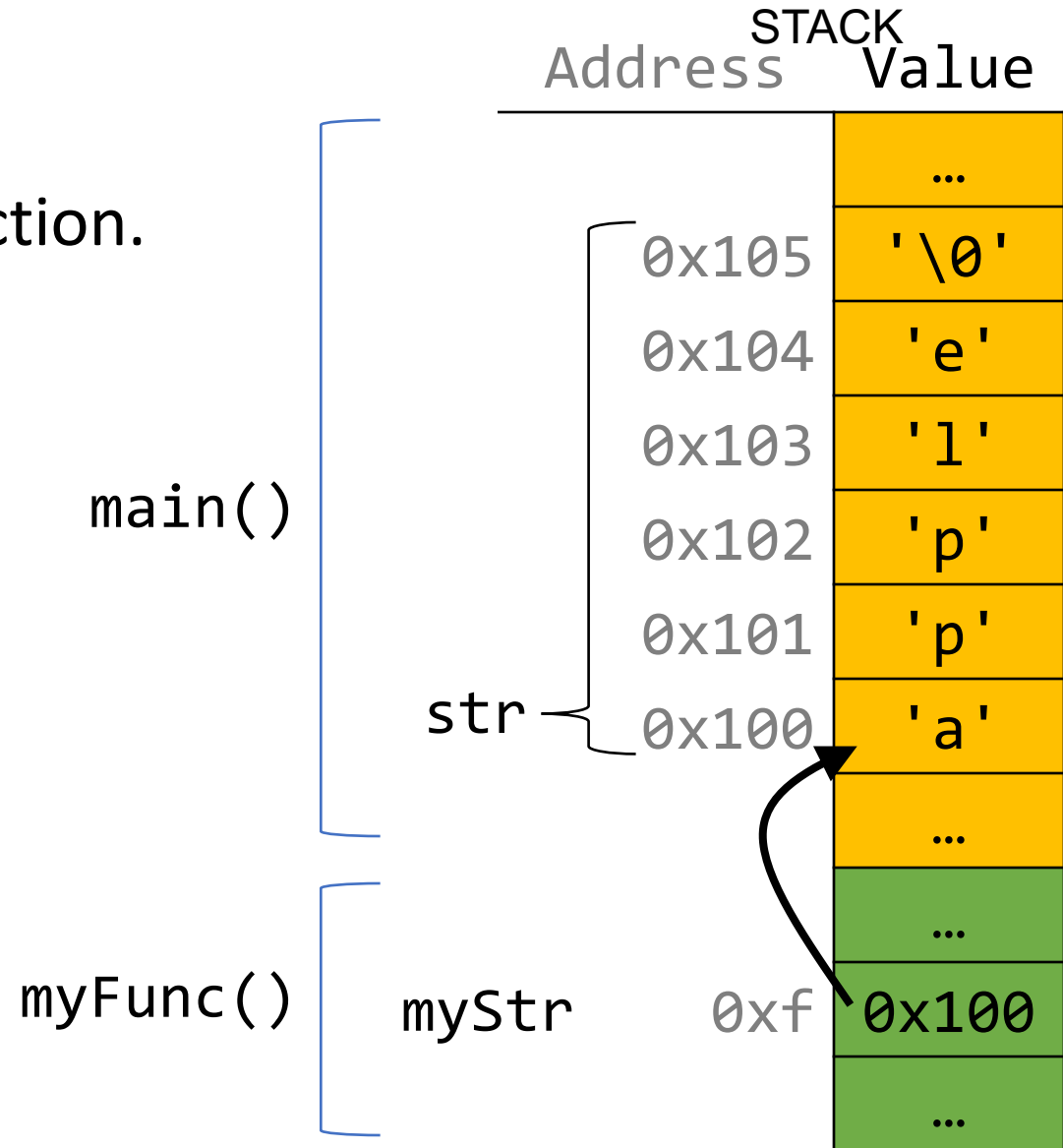


# Strings as Parameters

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) {  
    ...  
}
```

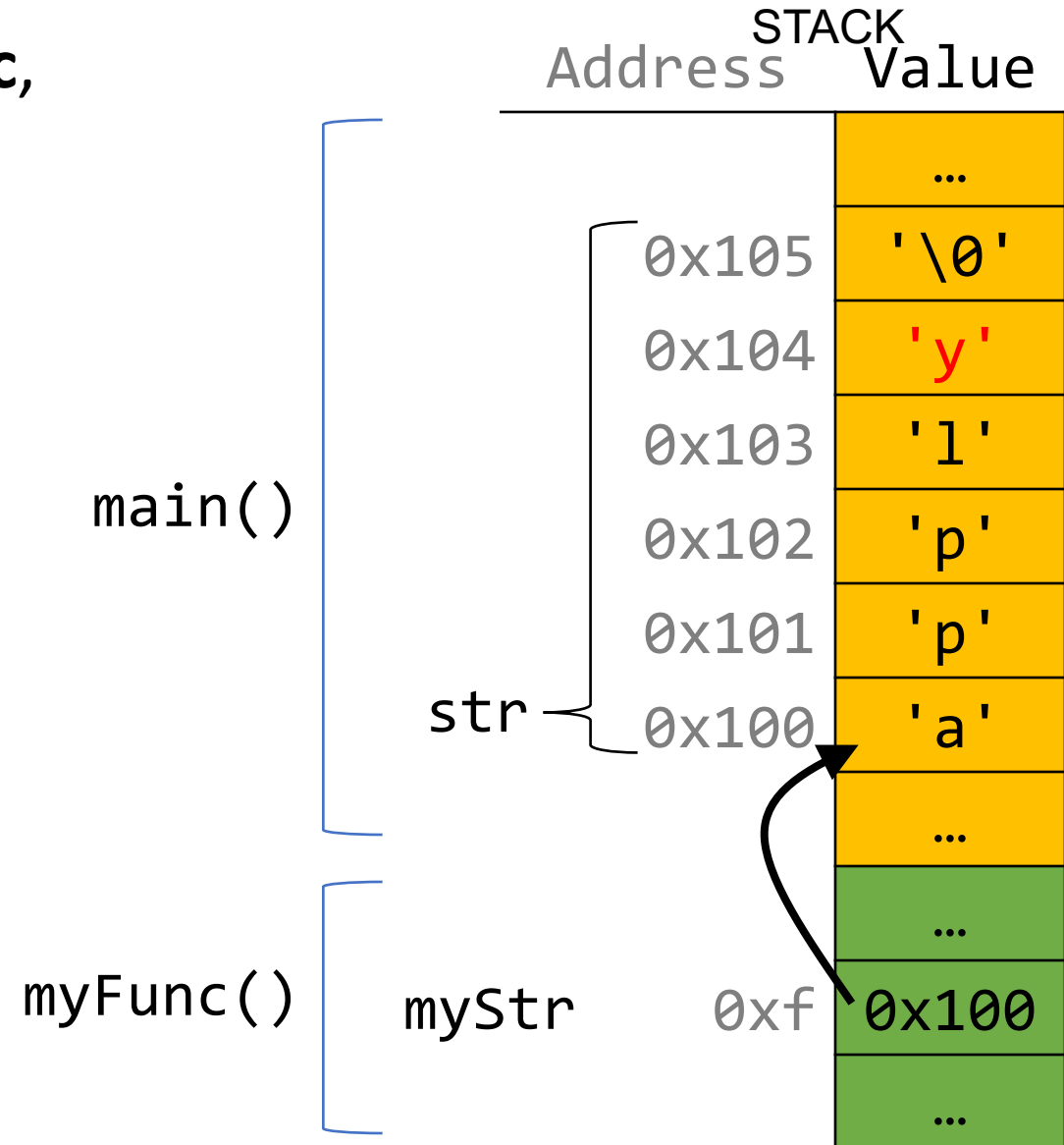
```
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    myFunc(str);  
    ...  
}
```



# 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  
    ...  
}
```



# 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!
```



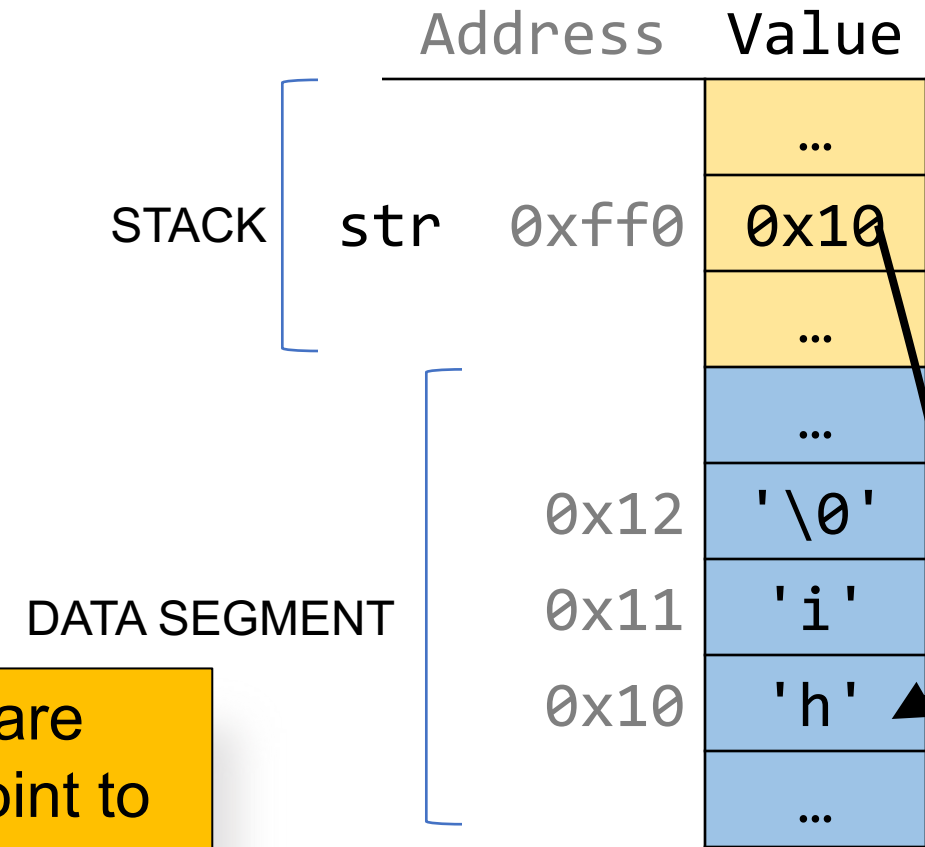
# Read-only Strings

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 **src** 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!
```

# Read-Only Strings

```
/* It's up to programmer using this function to not pass  
 * a read-only string.  
 */  
void myFunc(char *myStr) {  
    myStr[0] = 'h';  
}
```

# Recap

- Buffer Overflows, Security and Valgrind
- Review: Pointers
- Strings and Pointers

**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.