CS107, Lecture 5
More C Strings

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

Lecture Plan

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

Lecture Plan

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

C Strings

C strings are arrays of characters ending with a **null-terminating character** '\0'.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	'H'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

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

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 <u>does not</u> 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. <u>Always</u> adds a null-terminating character.
strspn(*str, accept*), strcspn(*str, reject*)	**strspn** returns the length of the initial part of *str* which contains <u>only</u> characters in *accept*. **strcspn** returns the length of the initial part of *str* which does <u>not</u> contain any characters in *reject*.

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


```
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 str**r**chr to obtain a pointer to the *last* occurrence.

Searching For Strings

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

```
char daisy[10];
strcpy(daisy, "Daisy Dog");
char *substr = strstr(daisy, "Dog");
printf("%s\n", daisy);           // Daisy Dog
printf("%s\n", substr);          // Dog
```

If there are multiple occurrences of the string, strstr returns a pointer to the *first* one.

String Spans

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

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

"How many places can we go in the first string before I encounter a character <u>not in</u> the second string?"

String Spans

strcspn (c = "complement") returns the *length* of the initial part of the first string which contains only characters <u>not in</u> the second string.

```
char daisy[10];
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 <u>in</u> 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!*).

```
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:

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

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

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

Arrays of Strings

We can access each string using bracket syntax:

```
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:

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

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

Practice: Password Verification

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

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

password is <u>valid</u> if it contains only letters in **validChars**, and does not contain any substrings in **badSubstrings**.

Practice: Password Verification

```
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

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

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

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

Pointers

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

Memory

- 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.

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

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.

Address	Value
	…
261	'\0'
260	'e'
259	'l'
258	'p'
257	'p'
256	'a'
	…

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

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

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

Looking Ahead to C

- 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*.

```c
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!
    ...
}
```

Pointers

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!
     ...
}
```

STACK

main

Pointers

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!
    ...
}
```

STACK

main

x 2

Pointers

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!
      ...
}
```
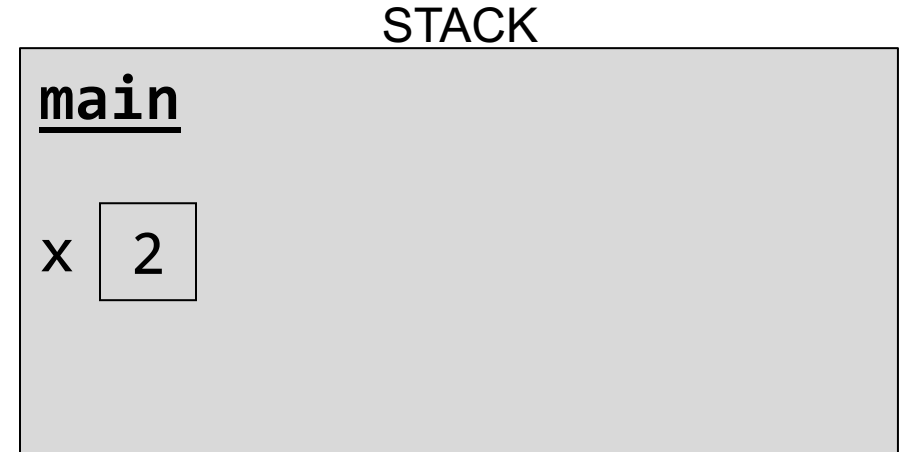
main

x 2

Pointers

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!
    ...
}
```
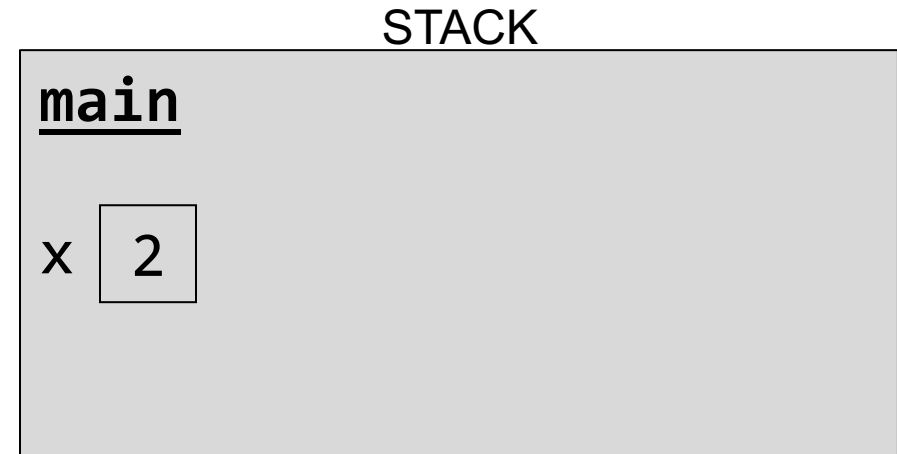
STACK

main

x 2

myFunc

intPtr

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!
    ...
}
```

STACK

main

x   2

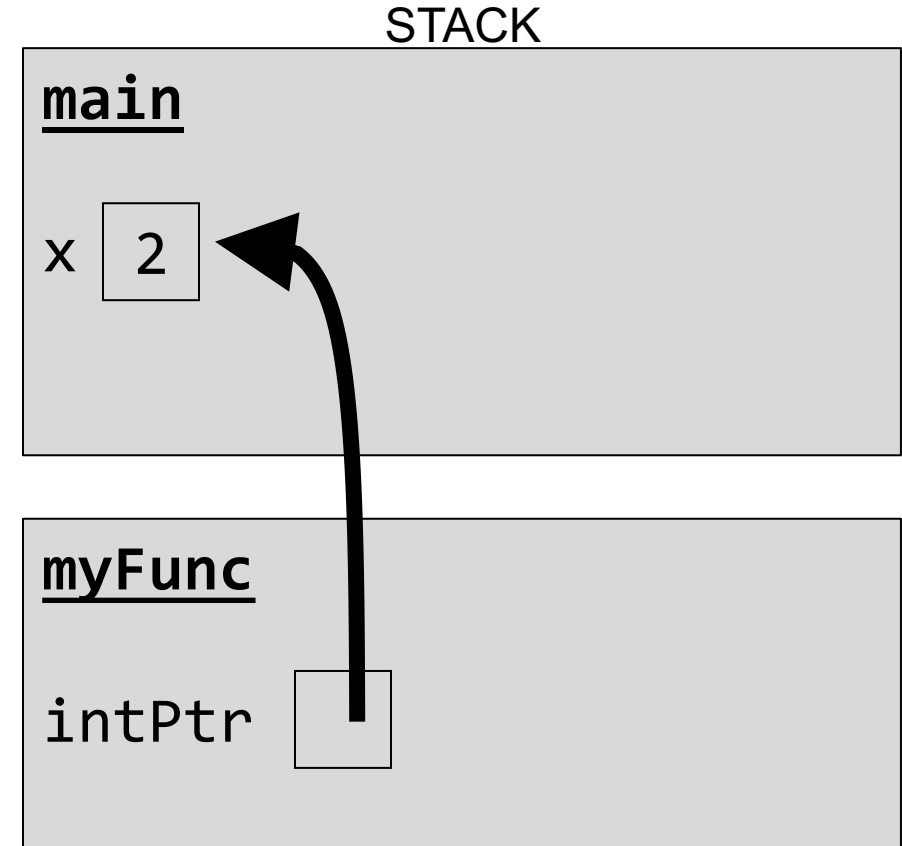**myFunc**

intPtr

Pointers

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!
    ...
}
```

main

x 3

myFunc

intPtr

Pointers

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!
    ...
}
```
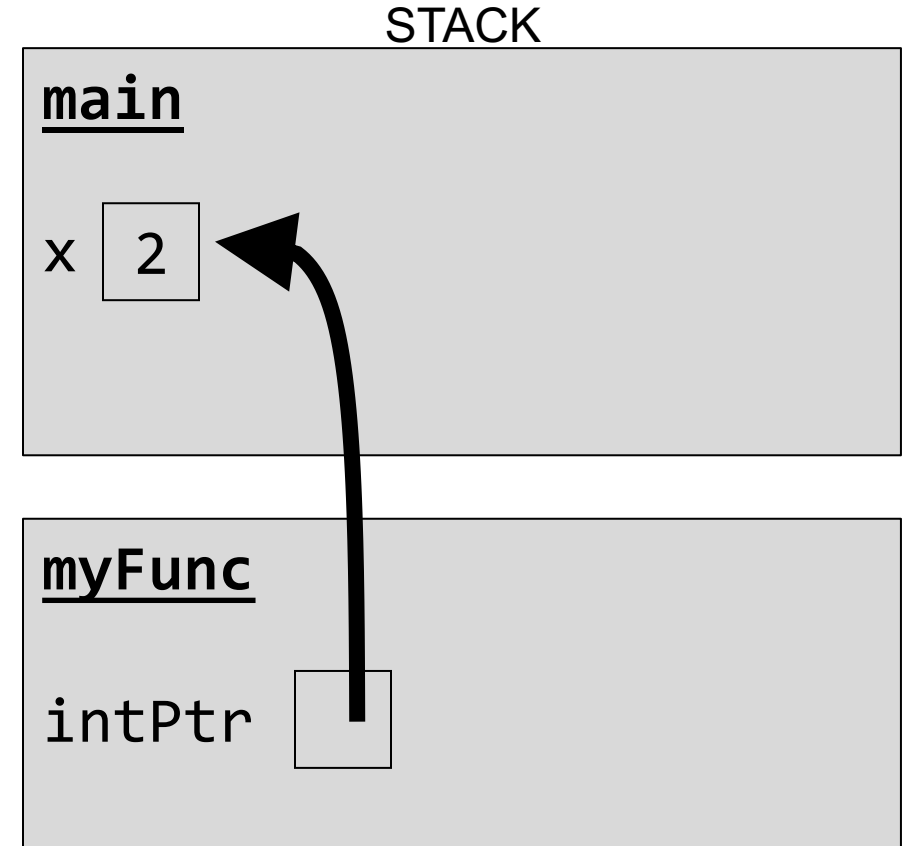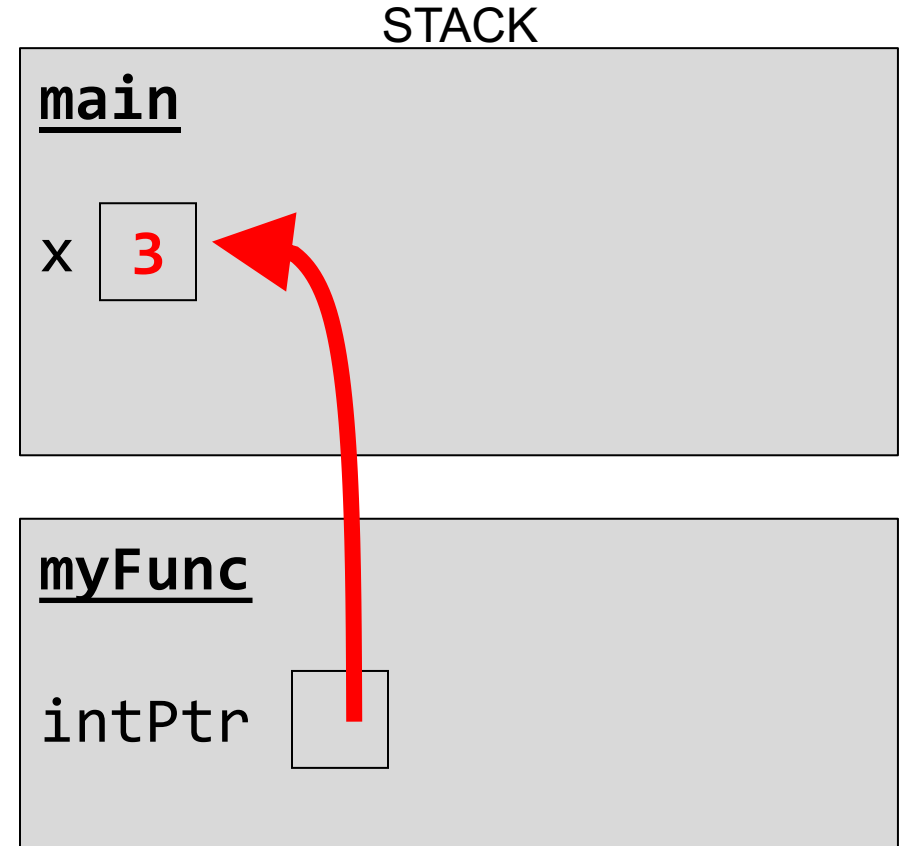
main

x 3

Pointers

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!
    ...
}
```

STACK

main

x   3

Pointers

<u>A pointer is a variable that stores a memory address.</u>

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

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

STACK

Address	Value
	…
x 0x1f0	2
	…

main()

Pointers

<u>A pointer is a variable that stores a memory address.</u>

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

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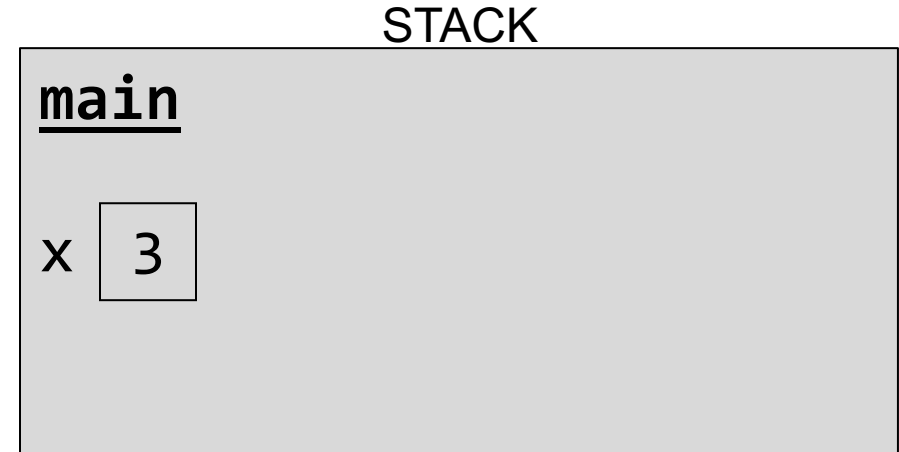
STACK

Address   Value

…

main()   x   0x1f0   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!
    ...
}
```
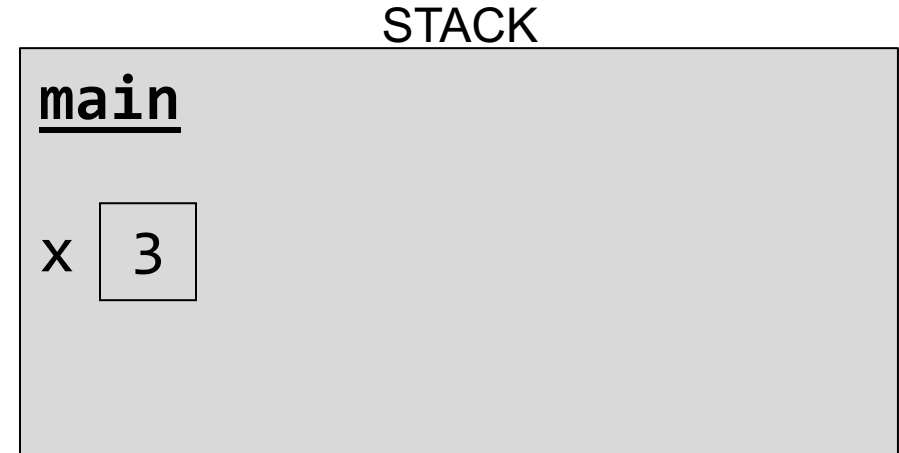
STACK

Address   Value

	…
x 0x1f0	2
	…
intPtr 0x10	0x1f0
	…

main()

myFunc()

<u>A pointer is a variable that stores a memory address.</u>

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

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

STACK

Address   Value

main()   x   0x1f0   2

myFunc()   intPtr 0x10   0x1f0

38

Pointers

<u>A pointer is a variable that stores a memory address.</u>

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

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

STACK

Address   Value

…

main()   x 0x1f0   3

…

myFunc() intPtr 0x10 0x1f0

…

<u>A pointer is a variable that stores a memory address.</u>

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

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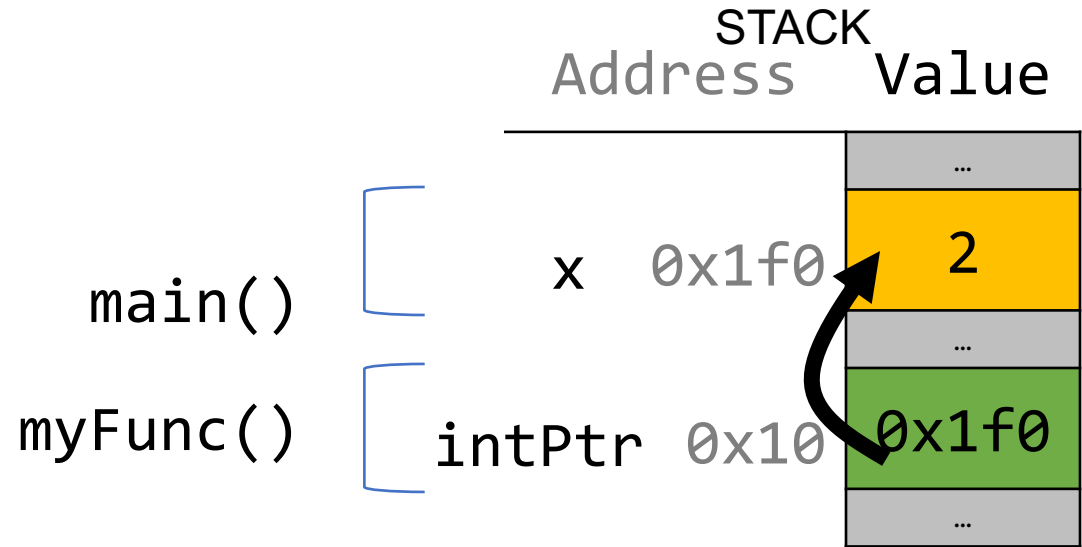
STACK

Address	Value
	…
x 0x1f0	3
	…

main()

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!
      ...
}
```
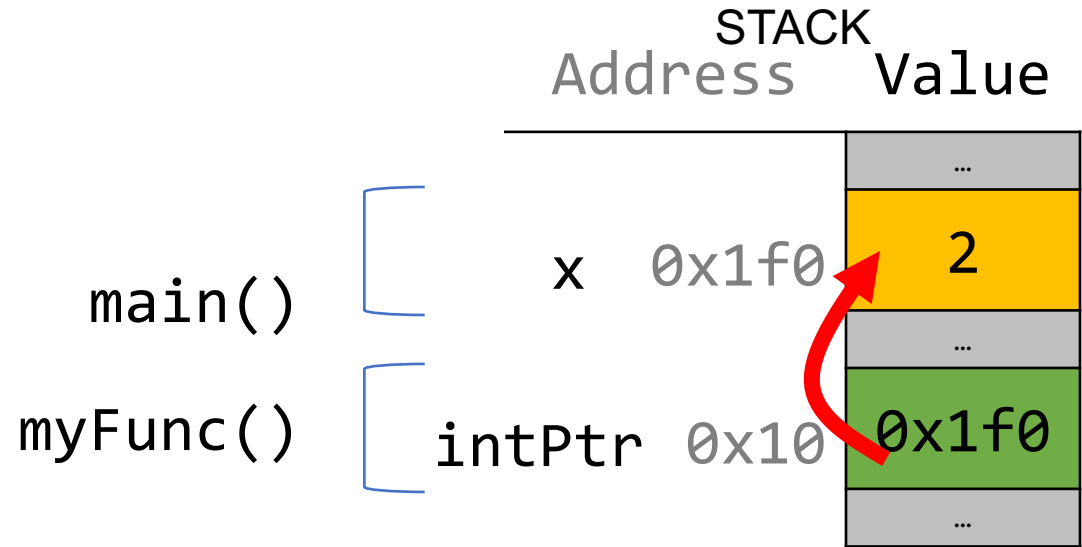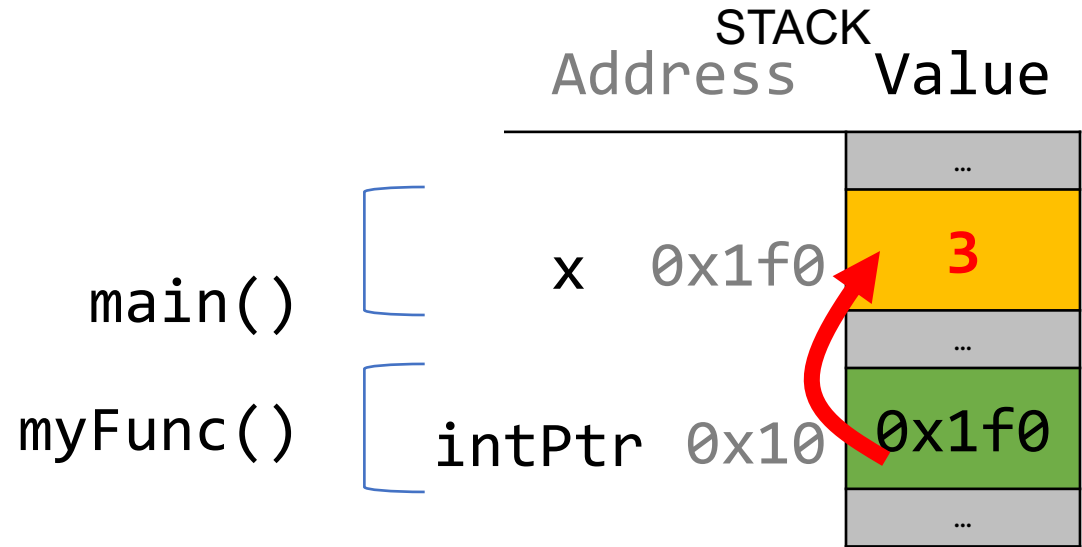
STACK

Address	Value
	…
x 0x1f0	3
	…

main()

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

Address	Value
	…

main()   x 0x1f0 | 2 |

…

Pointers

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!
    ...
}
```

STACK

Address		Value
		…

main()

x 0x1f0   2

…

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!
    ...
}
```

STACK

Address	Value
	…
x 0x1f0	2
	…
val 0x10	2
	…

main()

myFunc()

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!
    ...
}
```

STACK

Address   Value

main()

x	0x1f0	2

myFunc()

val	0x10	2

46

Pointers

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!
    ...
}
```

STACK

Address Value

		…
x	0x1f0	2
		…
val	0x10	3
		…

main()

myFunc()

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!
    ...
}
```

STACK

Address	Value
	…
x 0x1f0	2
	…

main()

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!
    ...
}
```

STACK

Address	Value
	…
x 0x1f0	2
	…

main()

Lecture Plan

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```
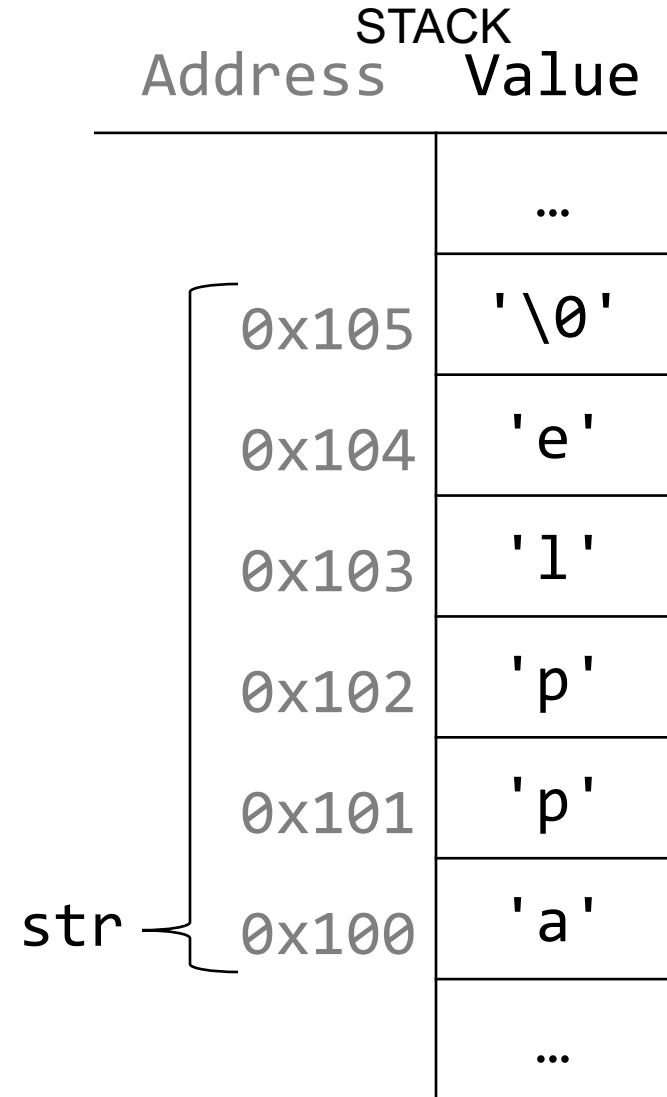
Strings In Memory

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.

```
char str[6];
strcpy(str, "apple");
```

STACK

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

str

53

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.

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


str = str2;    // not allowed!
```
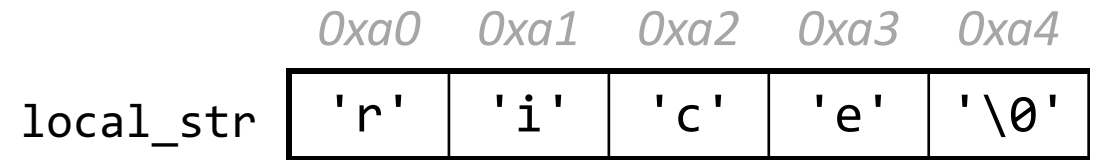
An array's size cannot be changed once we create it; we must create another new array instead.

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 *`.

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

```
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;
}
```

str ?

	0xa0	0xa1	0xa2	0xa3	0xa4
local_str	'r'	'i'	'c'	'e'	'\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 parameter `str` is being represented?

```
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;
}
```

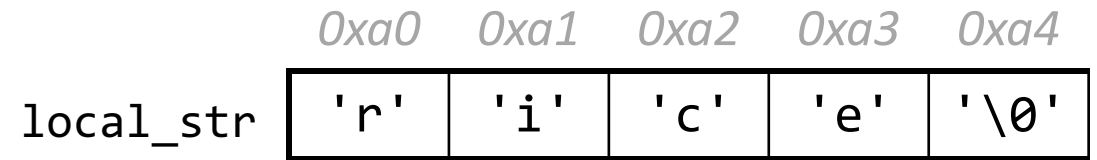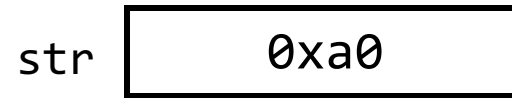| str	0xa0

	0xa0	0xa1	0xa2	0xa3	0xa4
local_str	'r'	'i'	'c'	'e'	'\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?

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

str

?

	0xa0	0xa1	0xa2	0xa3	0xa4
local_str	'r'	'i'	'c'	'e'	'\0'

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?

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

str | 0xa0 |

       0xa0 0xa1 0xa2 0xa3 0xa4
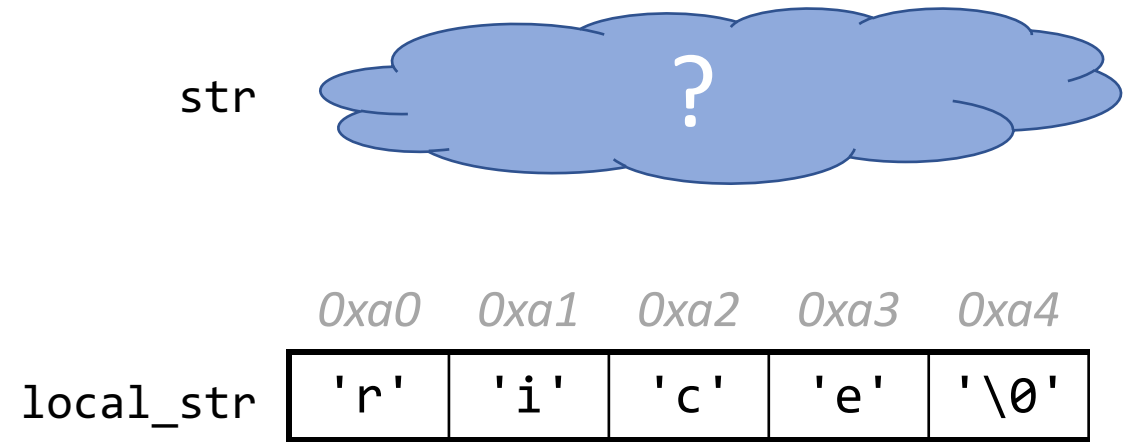
local_str | 'r' | 'i' | 'c' | 'e' | '\0' |

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?

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

str ?

	0xa0	0xa1	0xa2	0xa3	0xa4
local_str	'r'	'i'	'c'	'e'	'\0'

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

char * Variables

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

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```
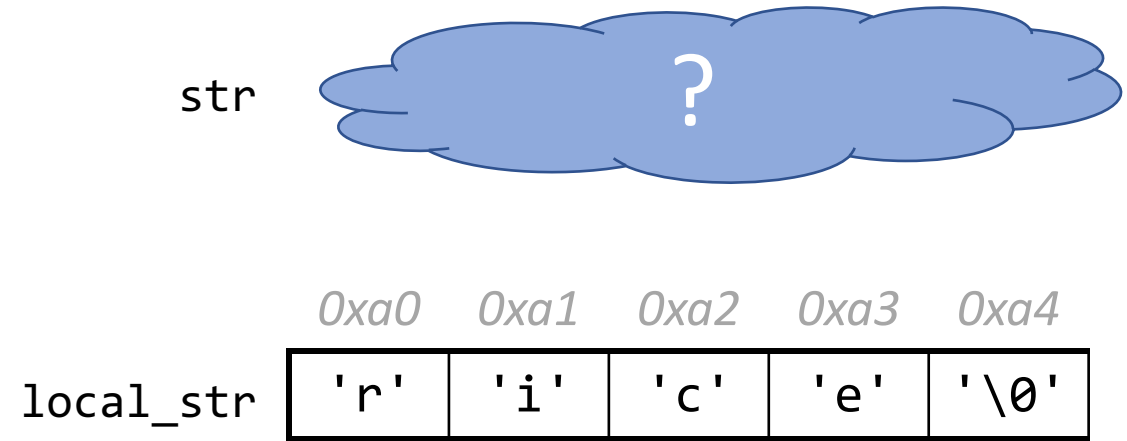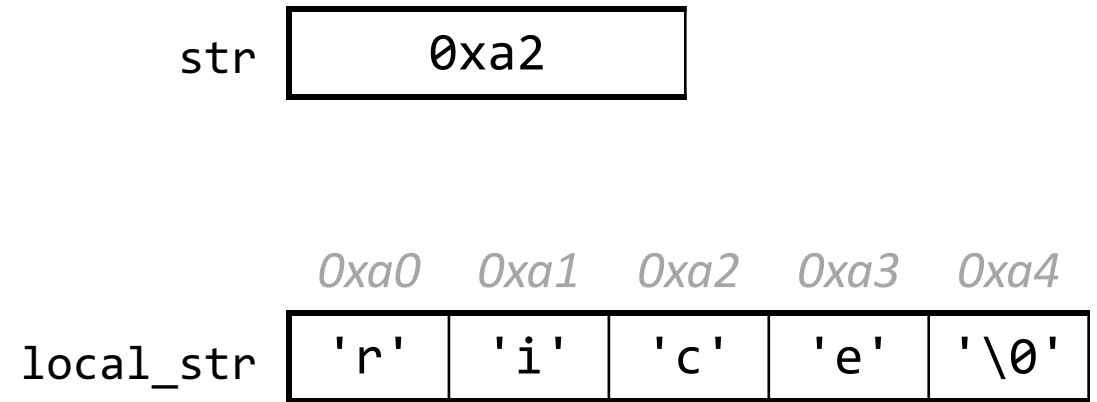
str | 0xa2 |

| | 0xa0 | 0xa1 | 0xa2 | 0xa3 | 0xa4 |
| local_str | 'r' | 'i' | 'c' | 'e' | '\0' |

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.

char *

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!";
char *empty = "";


myString[0] = 'h';                  // crashes!
printf("%s", myString);             // Hello, world!
```

char *

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*.

This applies only to creating *new* strings with char *. This does **not** apply for making a char * that points to an existing stack string.
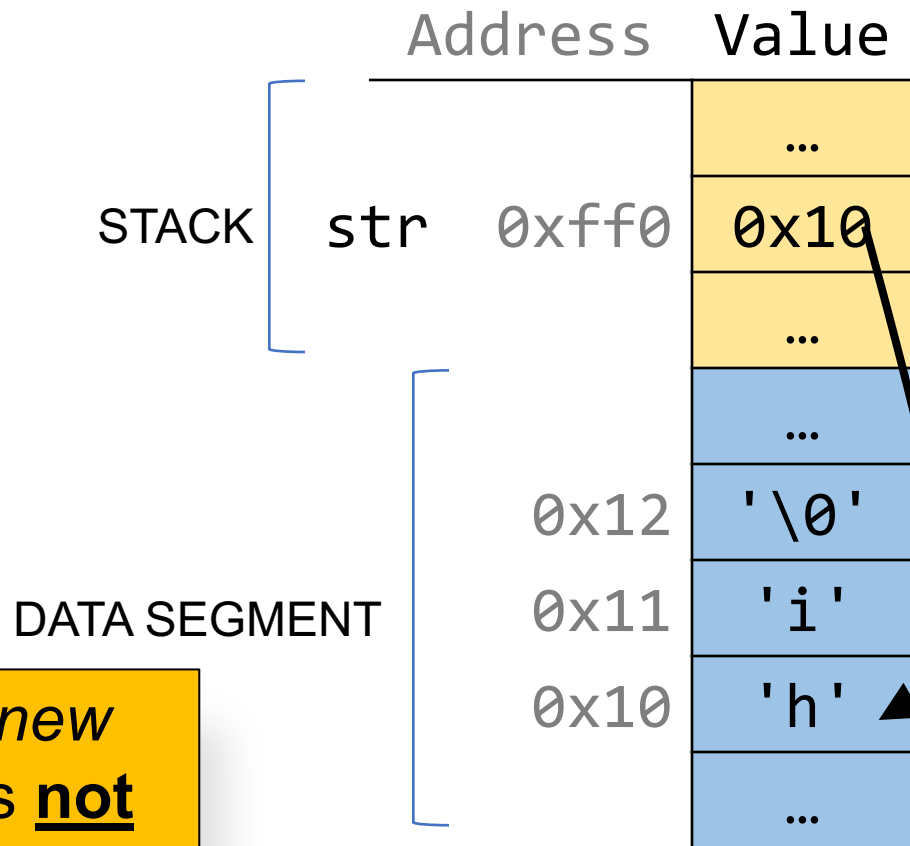
Address	Value
	…
STACK str 0xff0	0x10
	…
	…
0x12	'\0'
0x11	'i'
DATA SEGMENT 0x10	'h'
	…

Memory Locations

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

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

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

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

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

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**?

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

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

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

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

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

Memory Locations

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

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

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

Memory Locations

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.

char *

A **char** * variable refers to a single character. We can reassign an existing **char** * pointer to be equal to another **char** * pointer.

```
char *str = "apple";          // e.g. 0xfff0
char *str2 = "apple 2";       // e.g. 0xfe0
str = str2;    // ok!  Both store address 0xfe0
```

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;
    ...
}
```

STACK

Address	Value
	…
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
0xf8	0x100
	…

main()

str

ptr

75
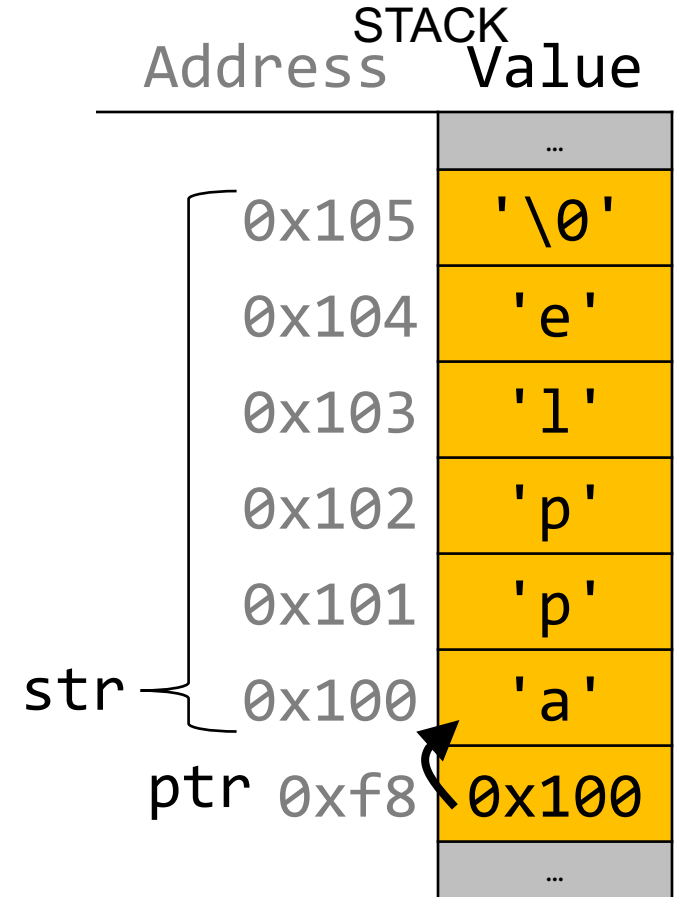
We can also make a pointer equal to an array;
it will point to the first element in that array.

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

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

    // confusingly equivalent, avoid
    char *ptr = &str;
    ...
}
```

STACK

Address	Value
	...
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
0xf8	0x100
	...

main()

str

ptr
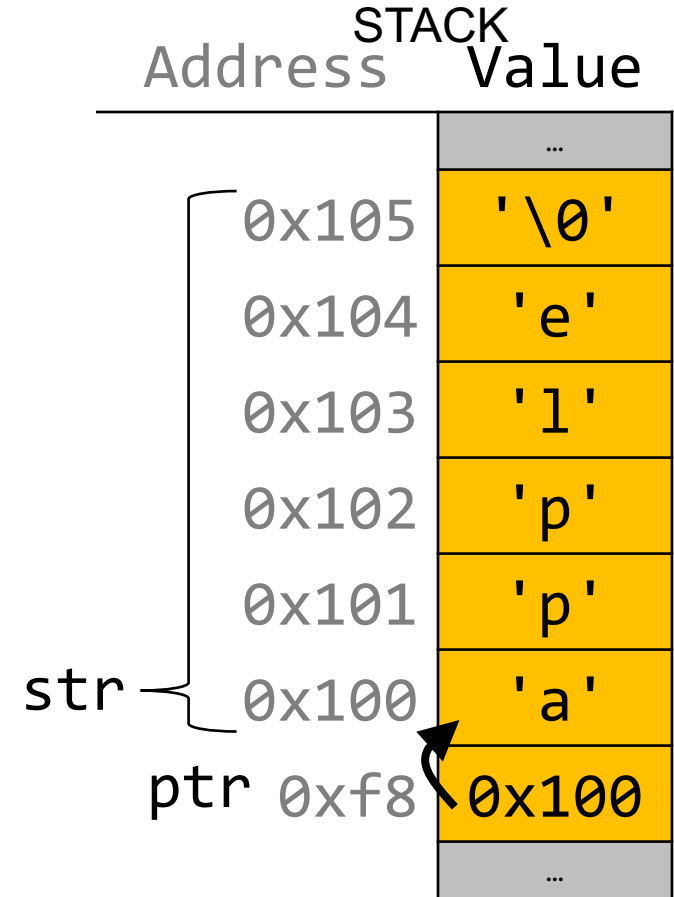
String Behavior #6: Adding an offset to a C string gives us a substring that many places past the first character.

Pointer Arithmetic

When we do pointer arithmetic, we are adjusting the pointer by a certain *number of places* (e.g. characters).

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

printf("%s", str);      // apple
printf("%s", str2);     // pple
printf("%s", str3);     // le
```

TEXT SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

78

char *

When we use bracket notation with a pointer, we are performing *pointer arithmetic and dereferencing*:

```
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'
```

TEXT SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.

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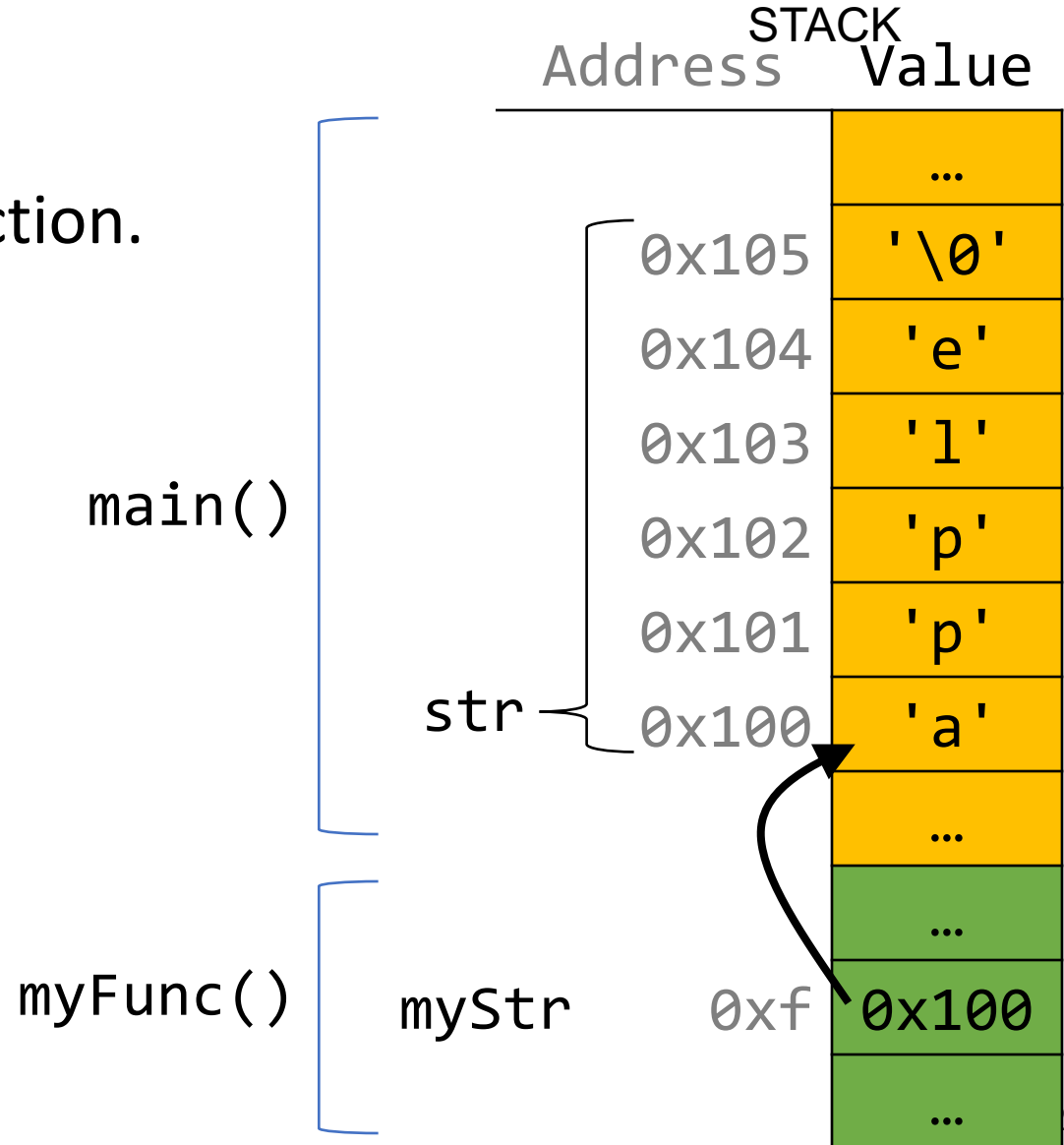
```
void myFunc(char *myStr) {
    ...
}

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

STACK

Address	Value
	…
str 0xfff0	0x10
	…
	…
myStr 0xff0	0x10
	…

main()

myFunc()

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);
    ...
}
```

STACK

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

main()

str
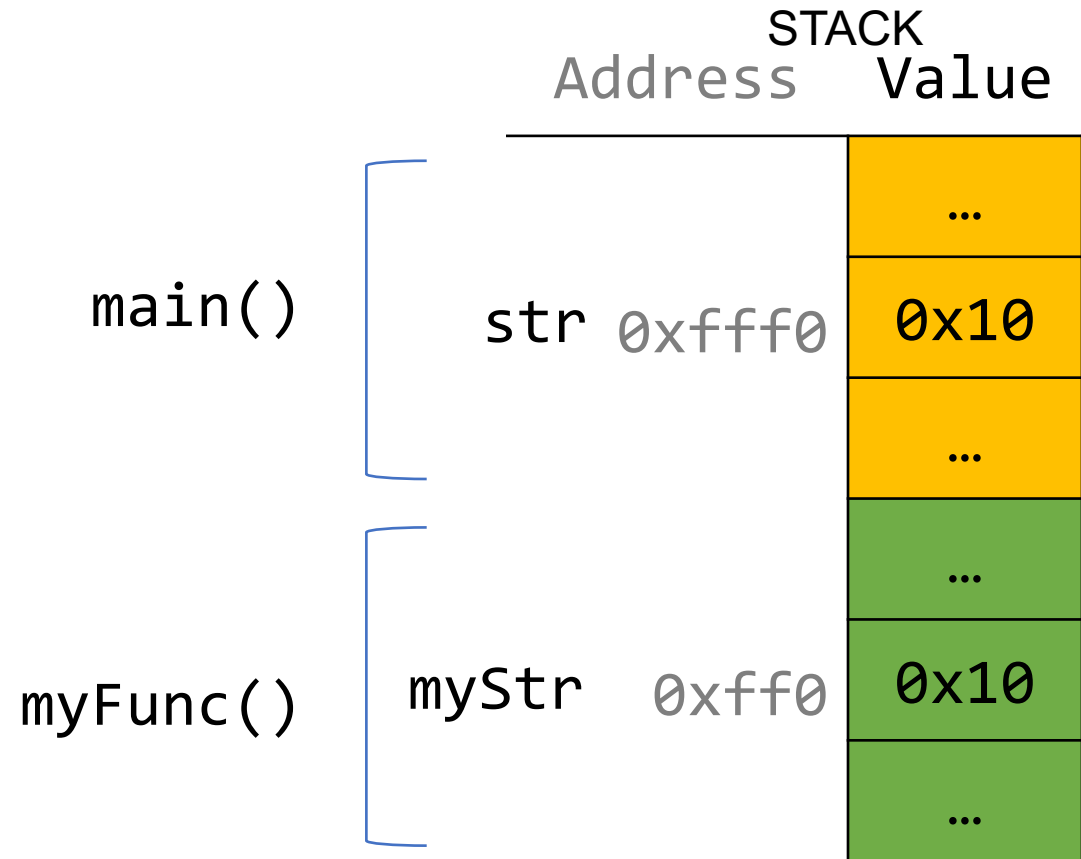
myFunc()

myStr

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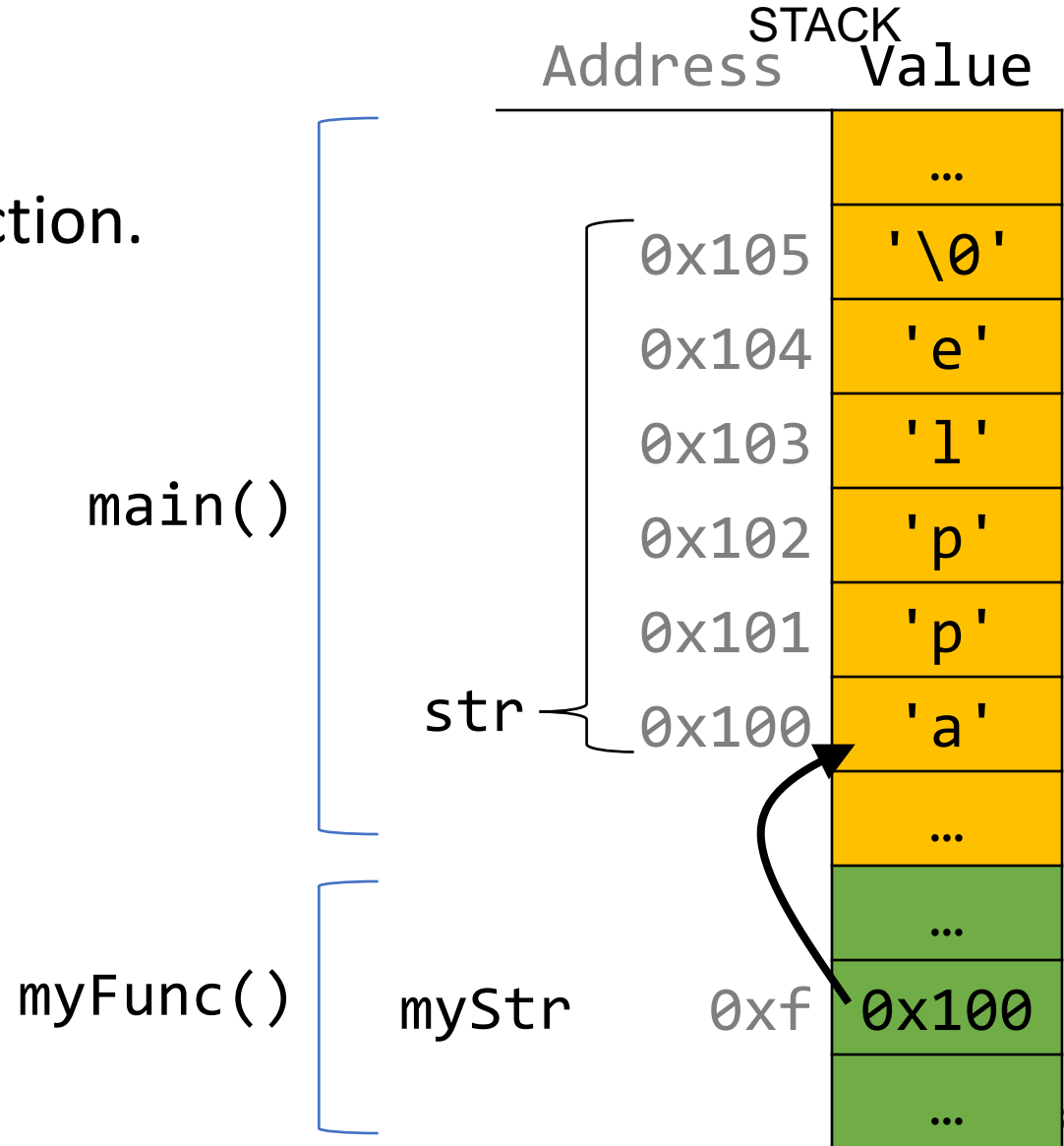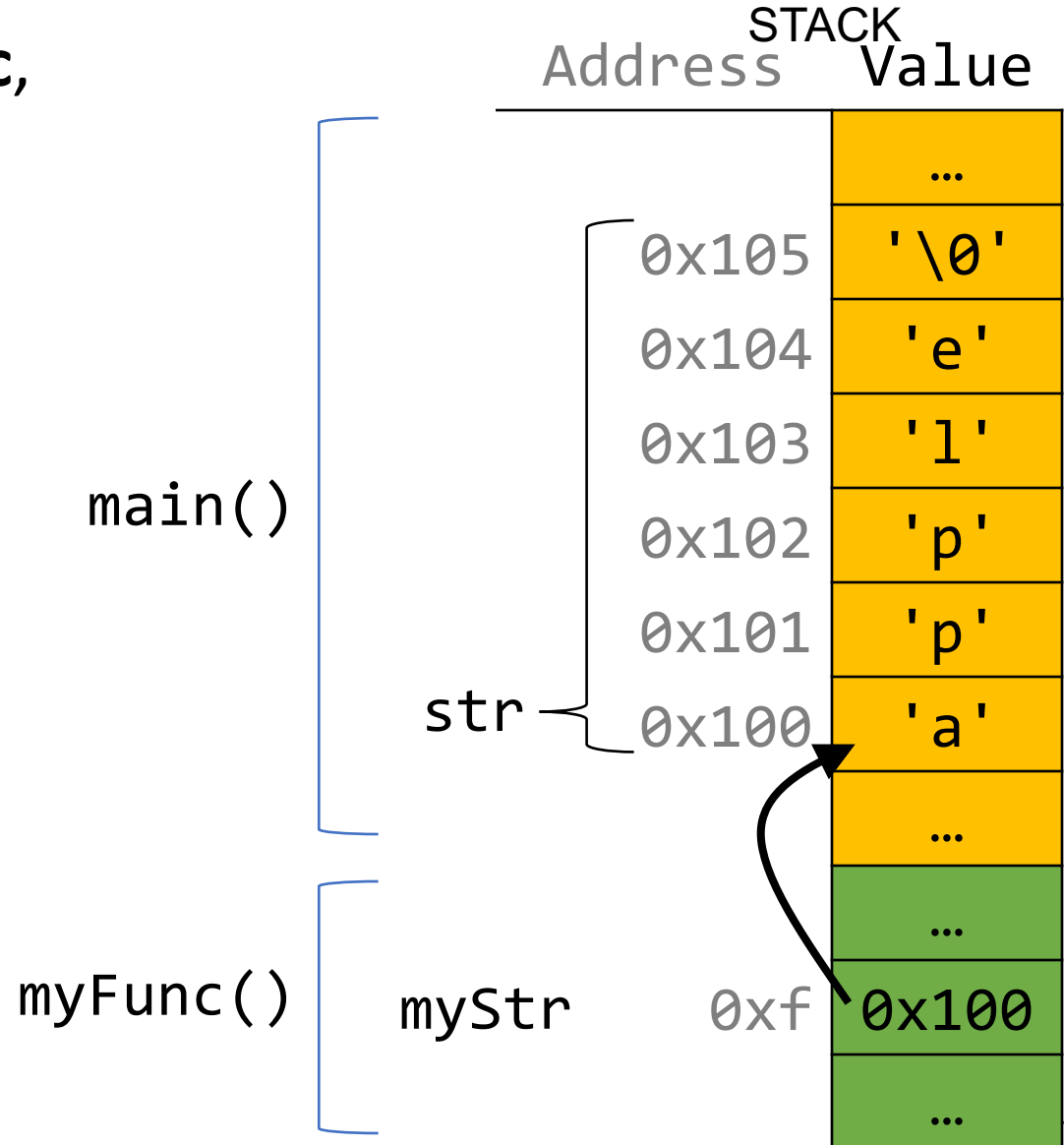
```
void myFunc(char *myStr) {
    ...
}

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


STACK

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

main()

str

myFunc()

myStr

Strings as Parameters

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

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

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

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

main()

str

myFunc()

myStr

Strings as Parameters

This means if we modify characters in **myFunc**, the changes will persist back in **main**!
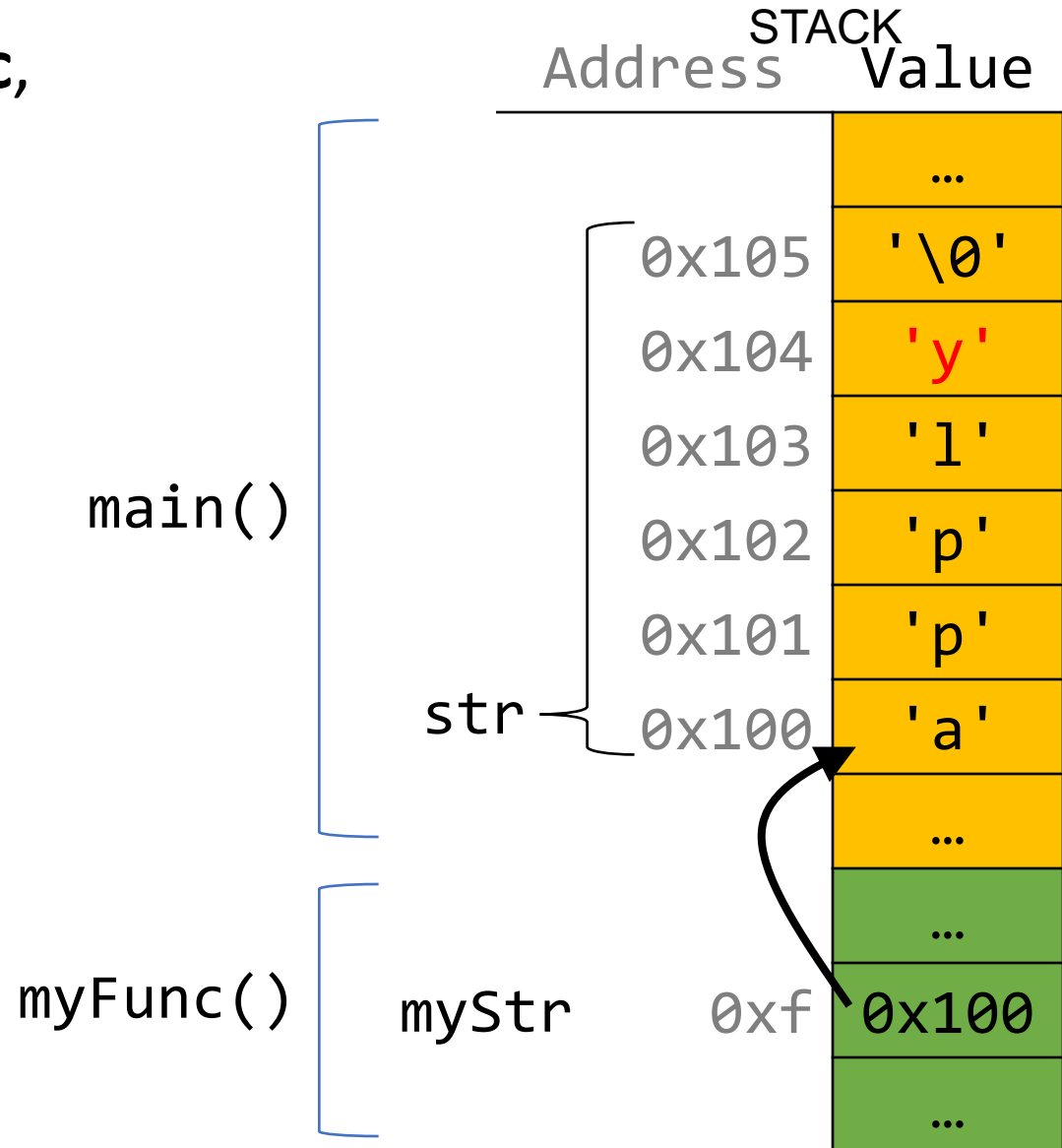
```c
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

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


STACK

Address	Value
	…
0x105	'\0'
0x104	'y'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	…

main()
str

myFunc()
myStr 0xf 0x100

Strings In Memory

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

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

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

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

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

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

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

Lecture Plan

- Searching in Strings   6
- **Practice:** Password Verification   13
- **Demo:** Buffer Overflow and Valgrind   16
- Pointers   19
- Strings in Memory   50
- Security and Overflows   88
- Live Session   110

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

Recall: Integer Overflow in Practice

- PSY – Gangnam Style

- The End of 32-bit Time aka January 13, 2038

- Gandhi in "Civilization"

- Pacman Level 256

- Make sure to reboot Boeing Dreamliners every 248 days

- Comair/Delta airline had to cancel thousands of flights days before Christmas

- Reported vulnerability CVE-2019-3857 in libssh2 may allow a hacker to remotely execute code

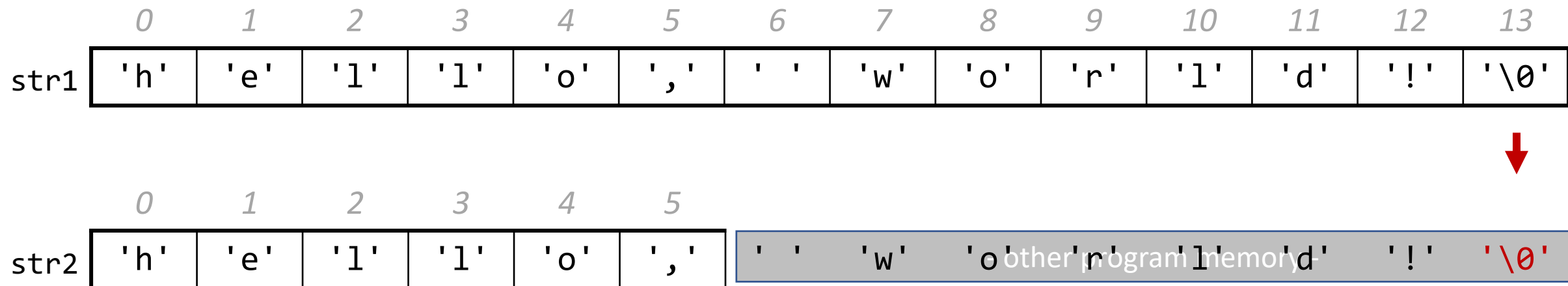- Donkey Kong Kill Screen

Assignment 1: Ariane-5 Case Study
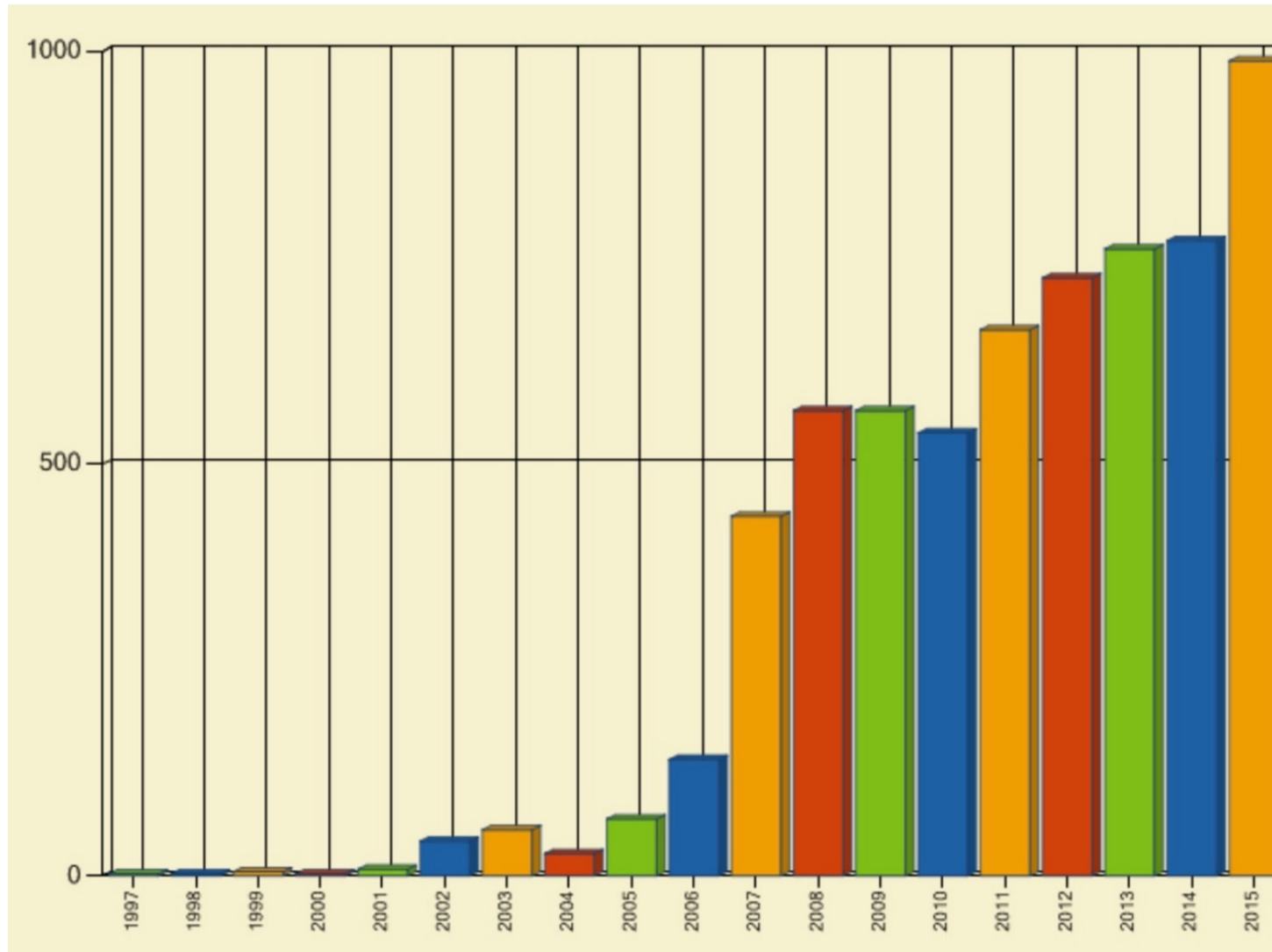
Integer Overflow in the National Vulnerability Database
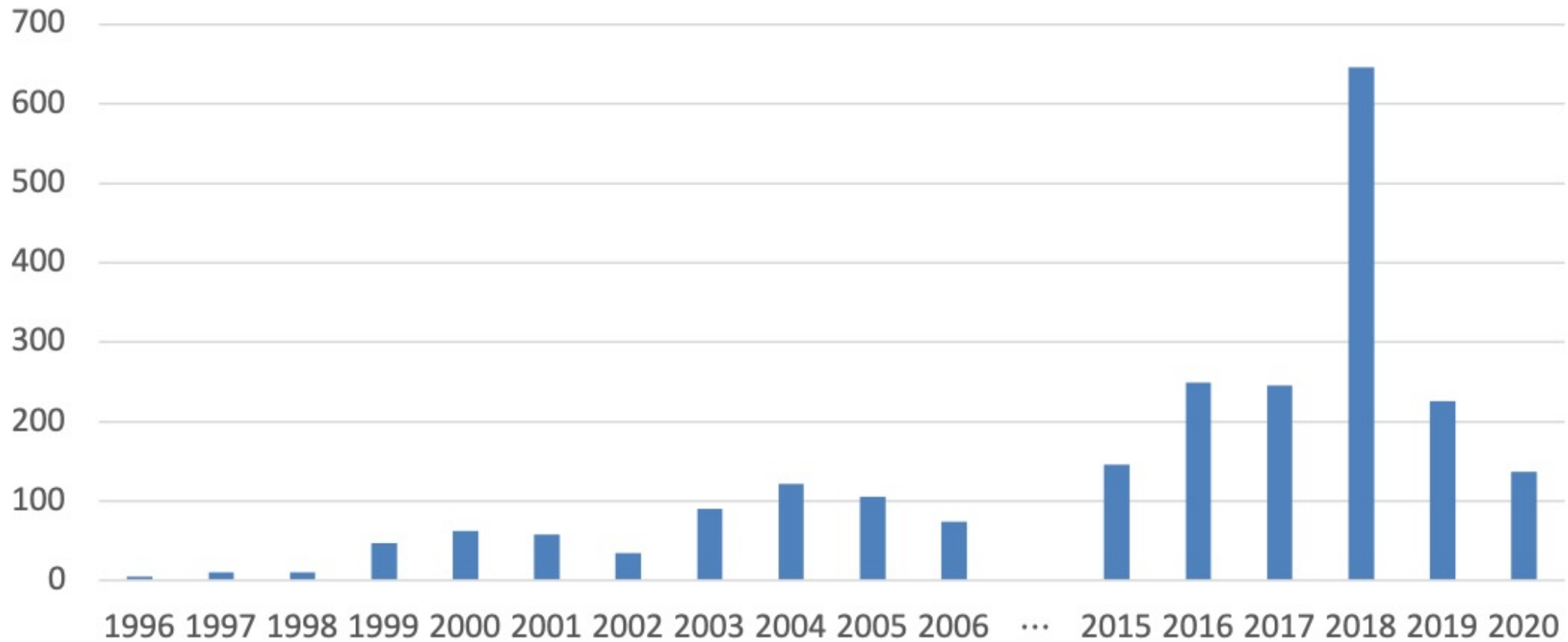
```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

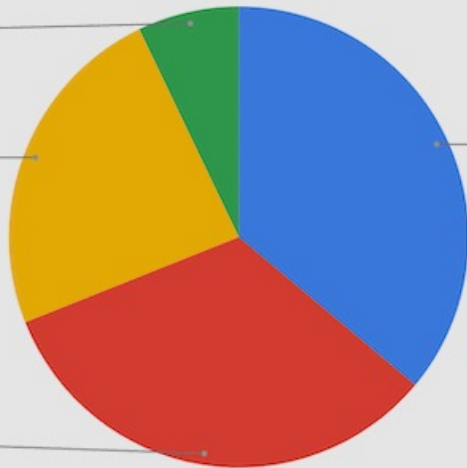# Buffer Overflow Vulnerabilities are Very Common
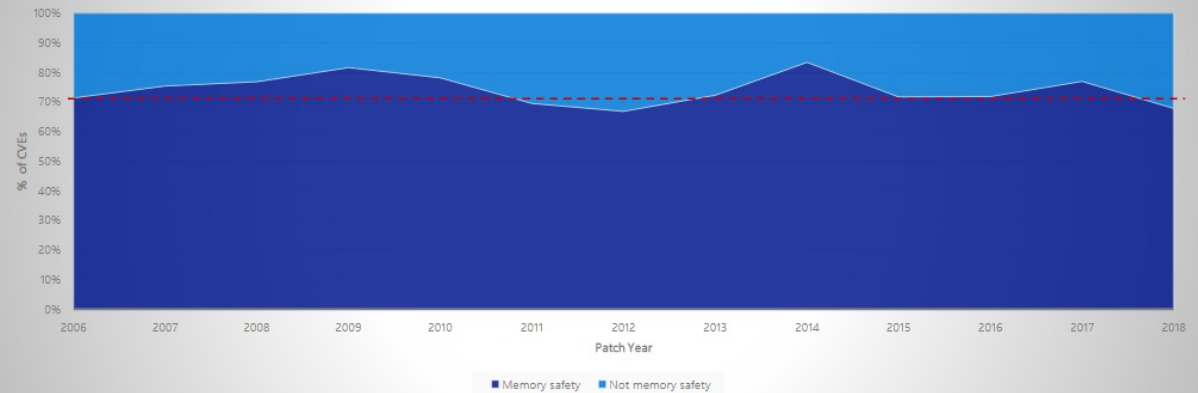
Memory safety issues include buffer overflows ...

How can we fix buffer overflows?

Idea 1: Constant Vigilance (While Programming)

- Check all buffers and trace their possible modifications

```
void printCapitalized(char *str1) {
    // make a copy first
    char buf[BUF_SIZE];
    strcpy(buf, str1);
    ...
}
```

How can we fix buffer overflows?

Idea 1: Constant Vigilance (While Programming)
- Check all buffers and trace their possible modifications
- Scrutinize functions that do not include bounds checking
  - common offenders include strcpy, printf, sprintf

```
int sprintf(char *str, const char *format, ...)
```

How can we fix buffer overflows?

Idea 1: Constant Vigilance (While Programming)

- Check all buffers and trace their possible modifications
- Scrutinize functions that do not include bounds checking
  - Some library functions later deprecated for this reason

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."

Idea 1: Constant Vigilance (While Programming)

- Check all buffers and trace their possible modifications
- Scrutinize functions that do not include bounds checking
- Consider use of strn- functions like strncpy
  - These functions will not write more than the size you specify
  - *However,* they do not ensure that strings are null-terminated
  - Use them, but use with care

How can we fix buffer overflows?

Idea 1:   Constant Vigilance (While Programming)
- Check all buffers and trace their possible modifications
- Scrutinize functions that do not include bounds checking
- Consider use of strn- functions like strncopy

Conclusion: Constant vigilance is necessary but not sufficient in the struggle to identify & fix buffer overflows

How Can We Fix Overflows?

Idea 2: Testing

- Valgrind: Your Greatest Ally

- Write your own tests

- Consider writing tests *before* writing the main program

✨ cs107.stanford.edu/testing.html ✨

How Can We Fix Overflows?

Idea 3: Documentation (Written by You)

- Document your assumptions about what will be stored in the buffer

- This will allow others (including your future self) to write better tests

How Can We Fix Overflows?

Idea 4: 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."

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)

Why Should We Fix Overflows?

- Why should I fix overflows?
  - So that my program doesn't crash

Why Should We Fix Overflows?

- Why should I fix overflows?
  - So that my program doesn't crash
  - So that my program works and accomplishes its goal

Why Should We Fix Overflows?

- Why should I fix overflows?
  - So that my program doesn't crash
  - So that my program works and accomplishes its goal

  - So that I can protect others who use and interact with my code

  - So that I can be a good computer scientist

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

On This Page

Preamble

1. GENERAL ET

1.1 Contribute
well-being, ack
are stakeholder

1.2 Avoid harm

1.3 Be honest a

1.4 Be fair and
discriminate.

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.

Thanks!
Visit my virtual office:

https://calendly.com/kathleencreel

Recap

- Searching in Strings
- **Practice:** Password Verification
- **Demo:** Buffer Overflow and Valgrind
- Pointers
- Strings in Memory
- Security and Overflows

Next time: Arrays and Pointers

Additional Live Session Slides

Plan For Today

- **5 minutes:** post questions or comments on Ed for what we should discuss

- **15 minutes**: open Q&A

- **25 minutes**: extra practice

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

Review

- Parameters in C are passed by value; if we pass something as a parameter, we pass a copy of it

- Therefore, if a function tries to change the value of a parameter itself, this will always change its local copy, not the original:

```
void myFunc(SOME_TYPE param) {
    param = ... // changes local copy
    ...
}

int main(int argc, char *argv[]) {
    SOME_TYPE x = ...
    myFunc(x);
    // x will always be the same here
    ...
}
```

Review

- If we wish to have another function modify some local variable, we must pass *its location* (address). Thus, we pass a copy of its address, not the value.

- The function can then dereference that address and modify what's there, and the changes will persist.

```
void myFunc(SOME_TYPE *param) {
    *param = ... // changes original copy
    ...
}

int main(int argc, char *argv[]) {
    SOME_TYPE x = ...
    myFunc(&x);
    // x will be changed here
    ...
}
```

Arrays vs. Pointers

- When you create an array, you are making space for each element in the array.
- When you create a pointer, you are making space for an 8 byte address.
- Arrays "decay to pointers" when you perform arithmetic or pass as parameters.
- You cannot set an array equal to something after initialization, but you can set a pointer equal to something at any time.
- &arr does nothing on arrays, but &ptr on pointers gets its address
- sizeof(arr) gets the size of an array in bytes, but sizeof(ptr) is always 8

Plan For Today

- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes**: open Q&A
- **25 minutes**: extra practice

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

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```
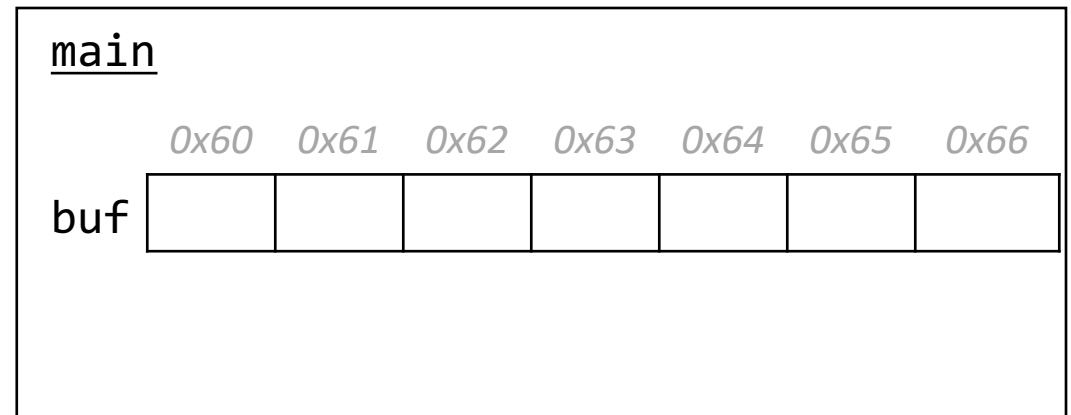
- Will there be a compile error/segfault?
- If no errors, what is printed?

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```

func

str

main

0x60 0x61 0x62 0x63 0x64 0x65 0x66

buf

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

117

2. Code study: strncpy

```
STRCPY(3)                    Linux Programmer's Manual                    STRCPY(3)

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

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

       A simple implementation of strncpy() might be:
```
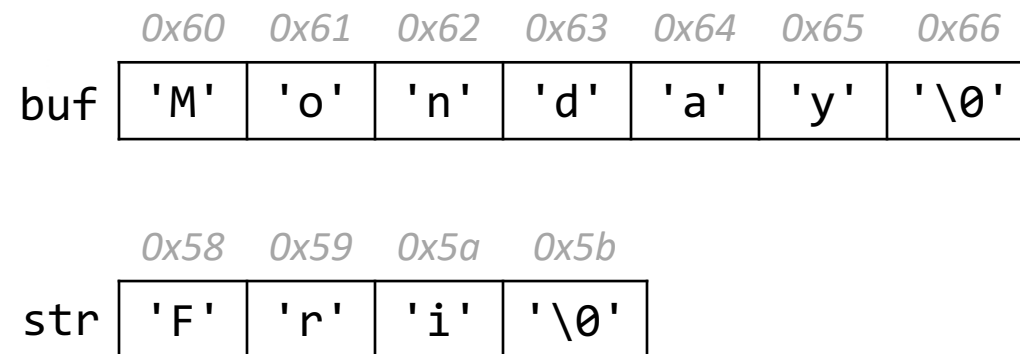
	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

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

What happens if we call `strncpy(buf, str, 5);`? 🤔

STRCPY(3)   Linux Programmer's Manual   STRCPY(3)

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

    If the length of <u>src</u> is less than <u>n</u>, **strncpy**() writes additional null
    bytes to <u>dest</u> to ensure that a total of <u>n</u> bytes are written.

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

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

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

dest [   ]

src [   ]

n [5   ]

i [   ]

What happens if we call `strncpy(buf, str, 5);`?

Suppose we use a variable str as follows:

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

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

1. ```
char str[7];
strcpy(str, "Hello1");
```

2. ```
char *str = "Hello2";
```

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

4. ```
char *ptr = "Hello4";
char *str = ptr;
```

🤔

Suppose we use a variable `str` as follows:

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

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

1.
```
char str[7];
strcpy(str, "Hello1");
```

Line A: Compile error
(cannot reassign array)

2.
```
char *str = "Hello2";
```

Line B: Segmentation fault
(string literal)

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

Prints eulo3

4.
```
char *ptr = "Hello4";
char *str = ptr;
```

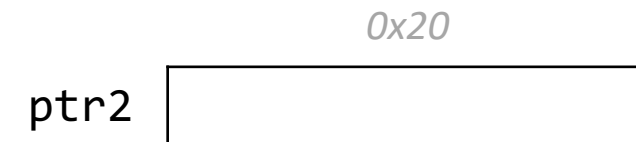Line B: Segmentation fault
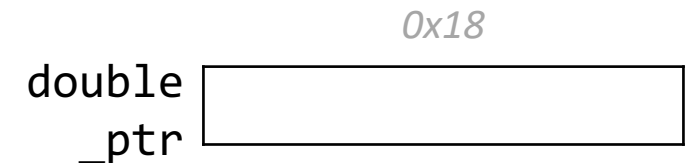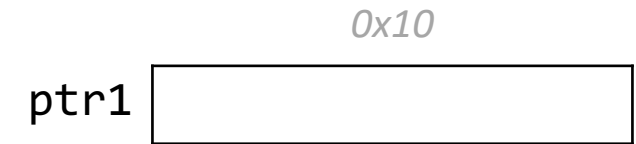(string literal)
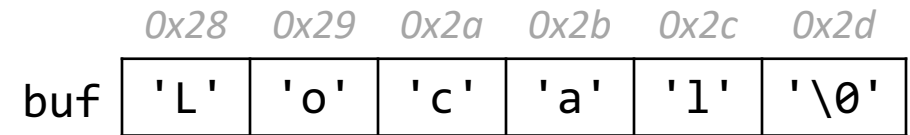
```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:      %p\n", ptr1);
6    printf("ptr1's deref     : %c\n", *ptr1);
7    printf("         address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:     %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:      %s\n", ptr2);
12 }
```

What is stored in each variable? (We cover double pointers more in Lecture 6)

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1's deref    : %c\n", *ptr1);
7    printf("         address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```

	0x28	0x29	0x2a	0x2b	0x2c	0x2d
buf	'L'	'o'	'c'	'a'	'l'	'\0'

0x10

ptr1

0x18

double
_ptr

0x20

ptr2

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

123