CS107, Lecture 6
More Pointers and Arrays

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

CS107 Topic 2

<u>How can a computer represent and manipulate more complex data like text?</u>

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

- 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

- Pointers and Parameters
- Strings in Memory
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

Lecture Plan

- **Pointers and Parameters**
- Strings in Memory
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

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 represent 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'
	…

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

<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!
      ...
}
```

Pointers

A pointer is a variable that stores a memory address.

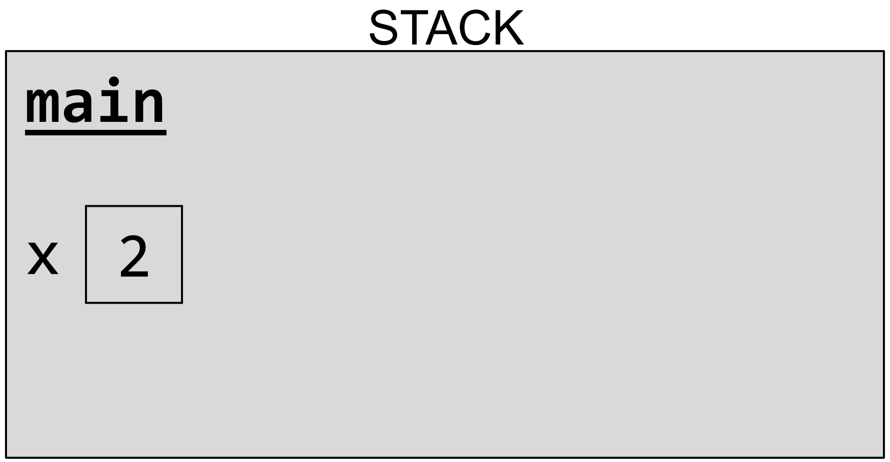
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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

STACK

Pointers

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

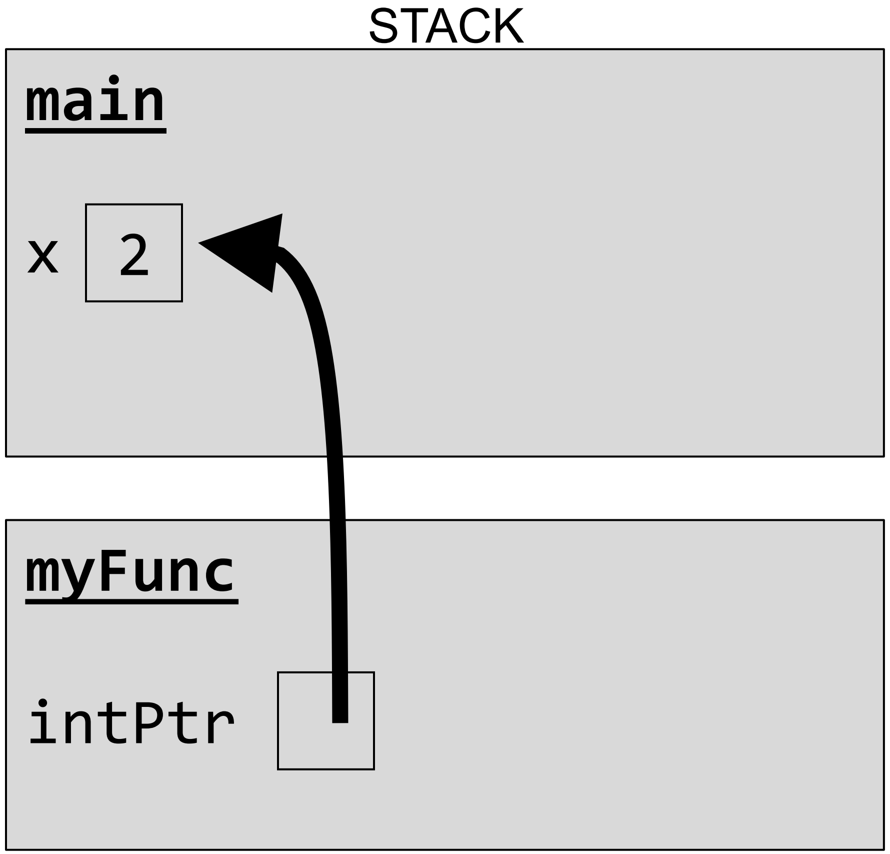
```
void myFunc(int *intPtr) {
	*intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

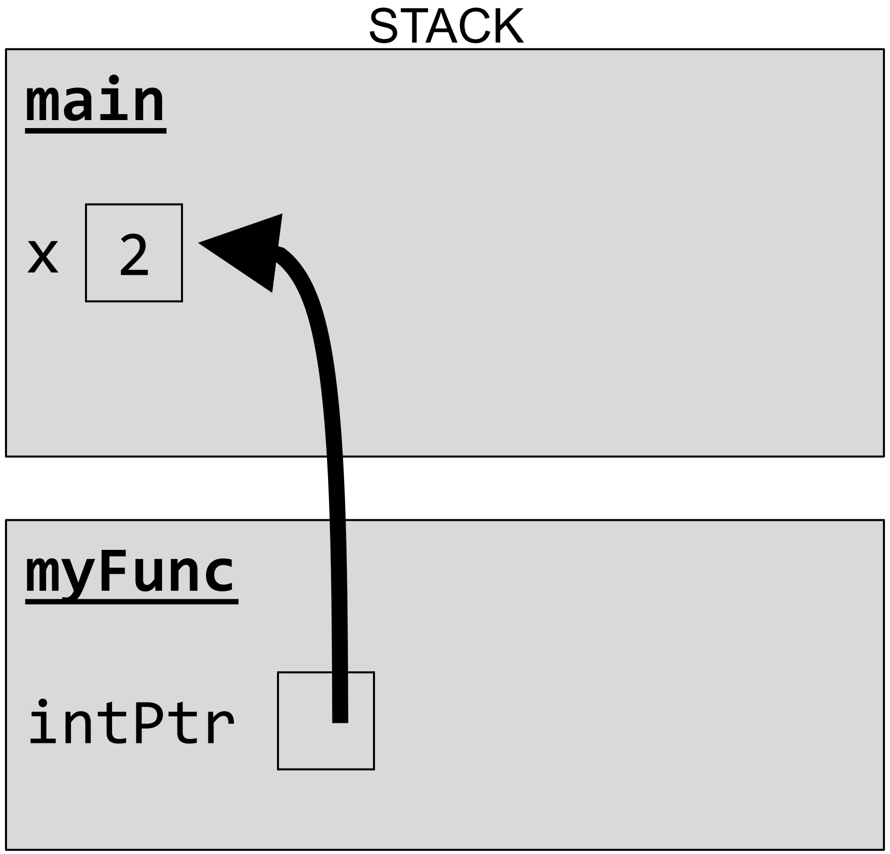
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

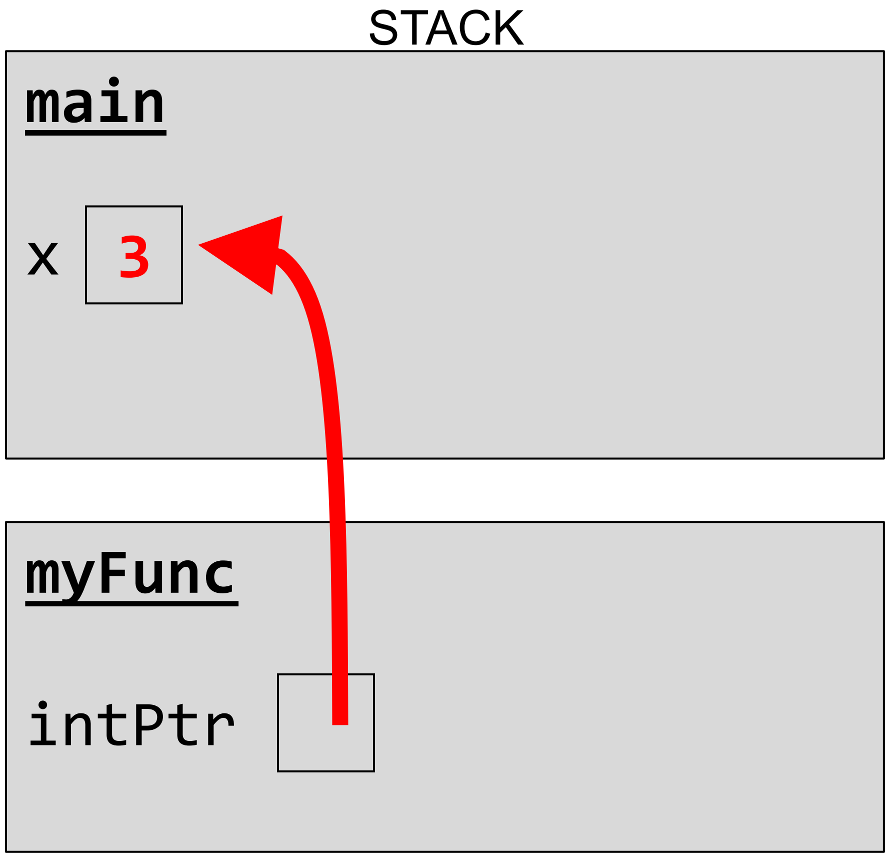
```
void myFunc(int *intPtr) {
	*intPtr = 3;
}

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



Pointers

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

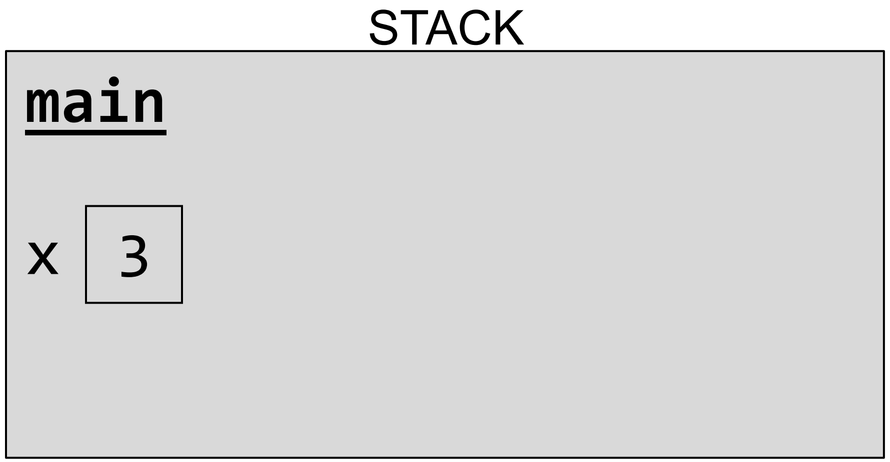
```
void myFunc(int *intPtr) {
	*intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

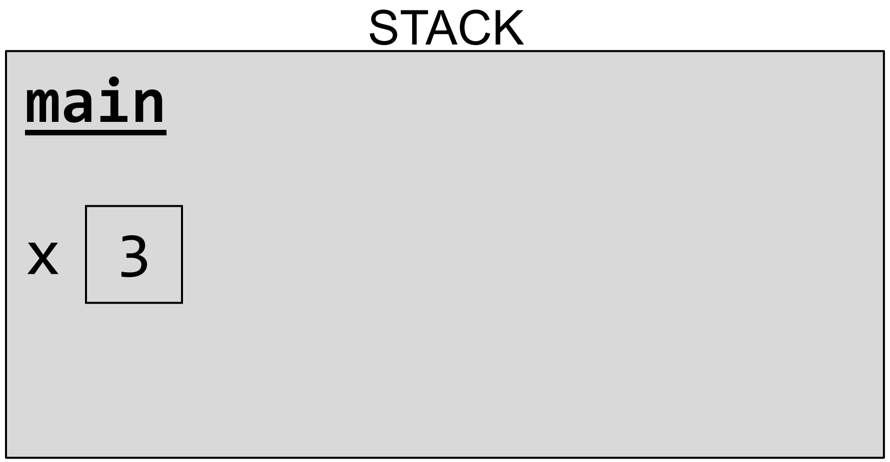
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

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

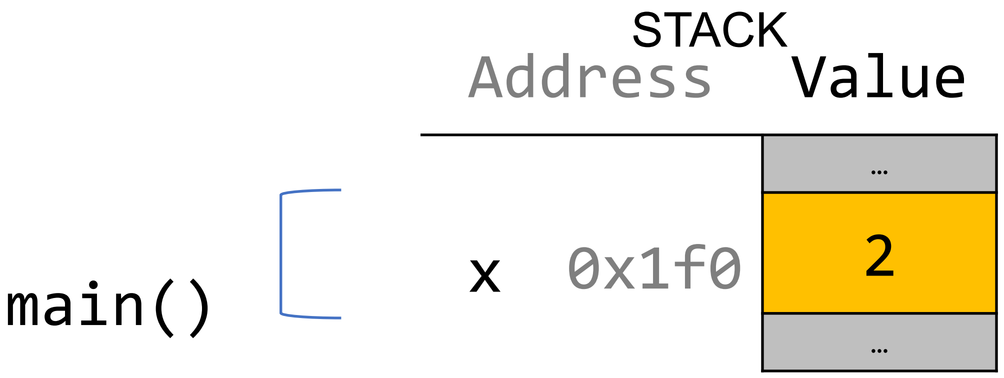
```
void myFunc(int *intPtr) {
	*intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

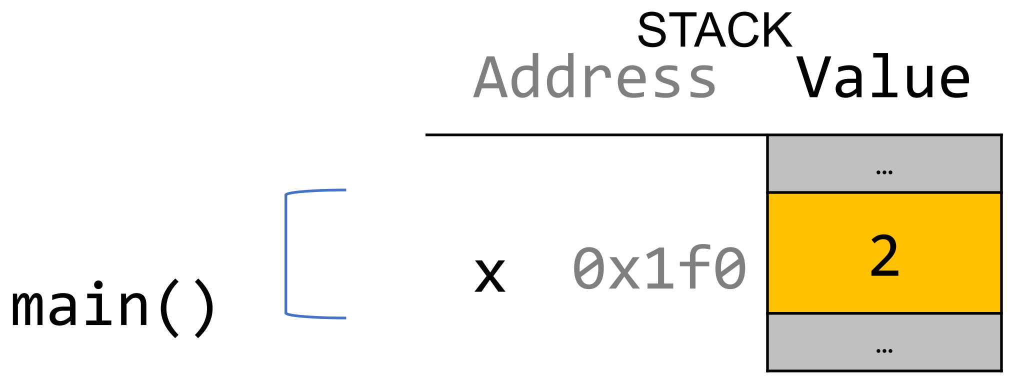
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

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

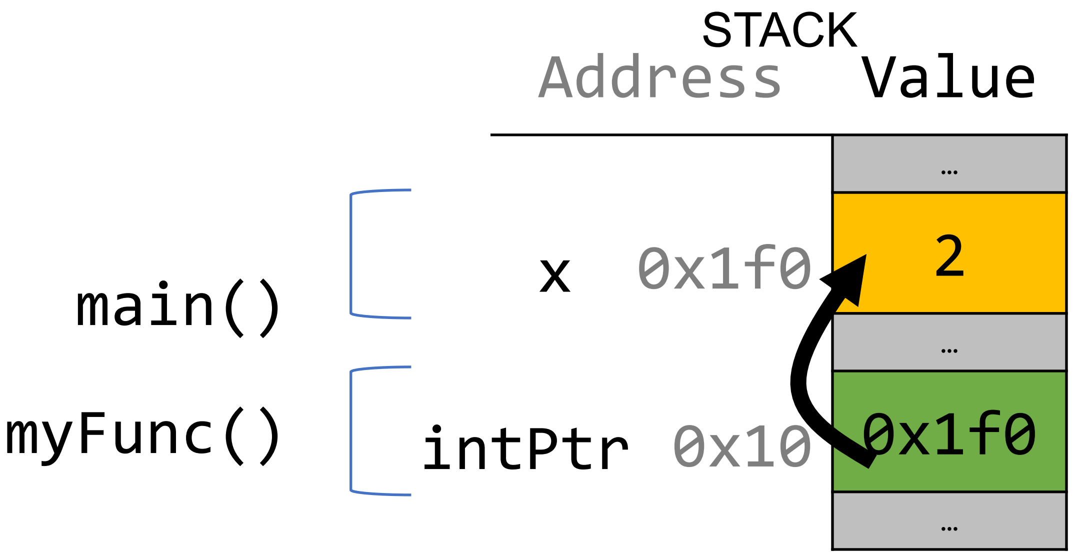
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

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

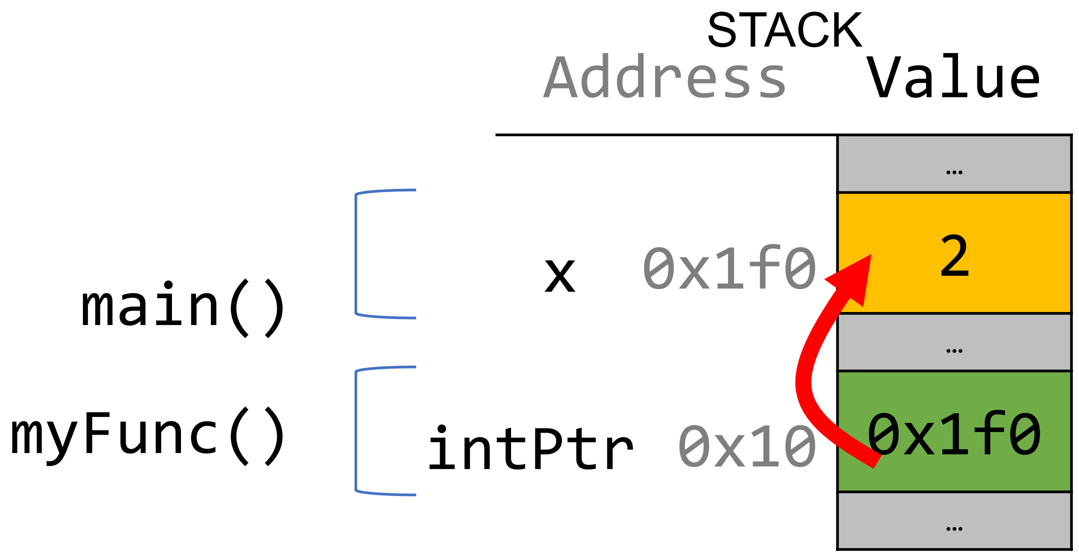
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

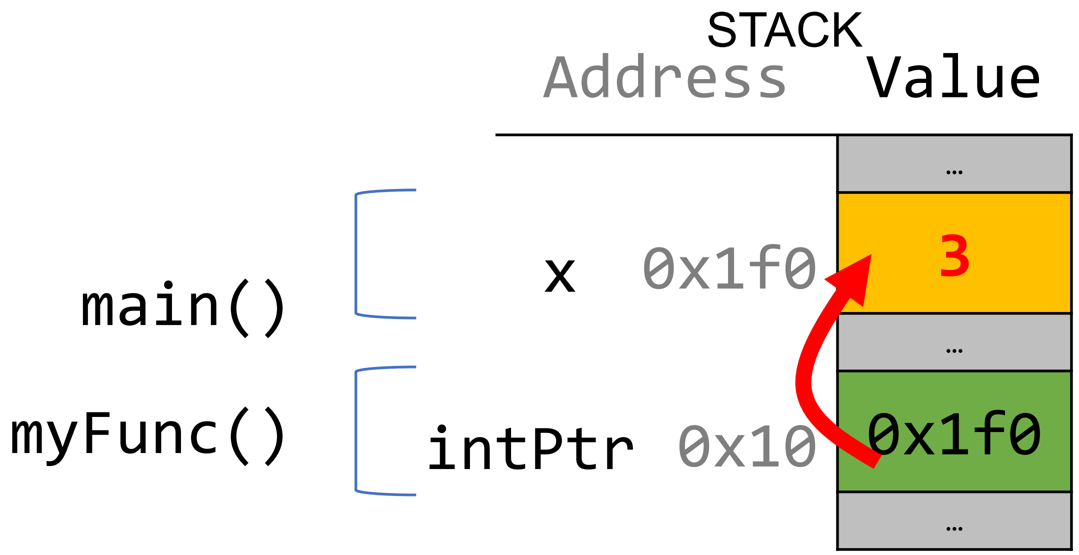
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

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



Pointers

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

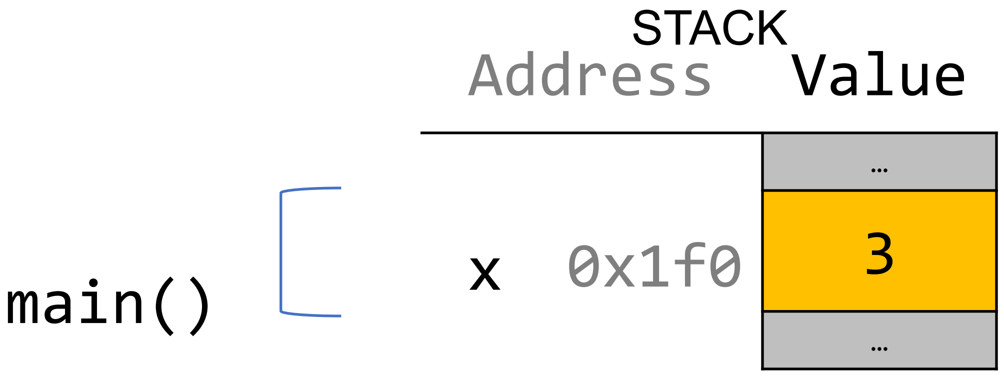
```
void myFunc(int *intPtr) {
	*intPtr = 3;
}

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



Pointers

A pointer is a variable that stores a memory address.

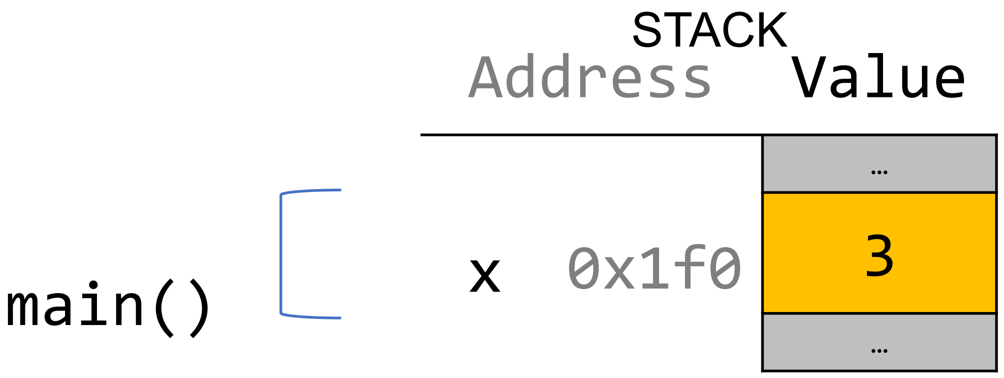
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

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

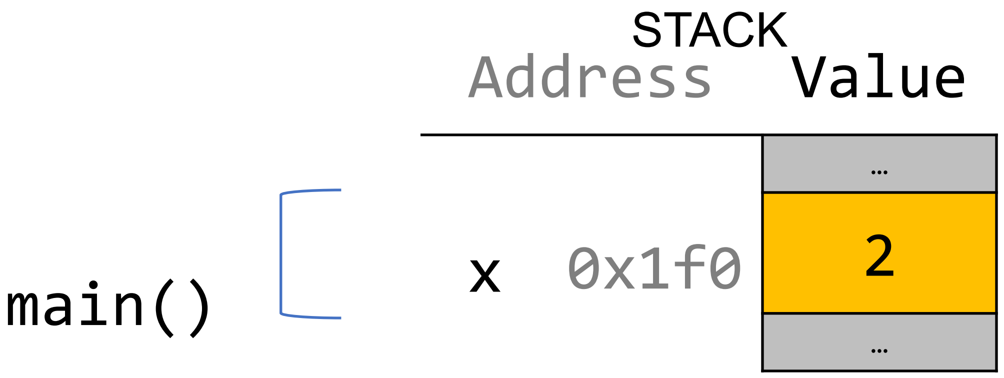
```c
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

Without pointers, we would make copies.

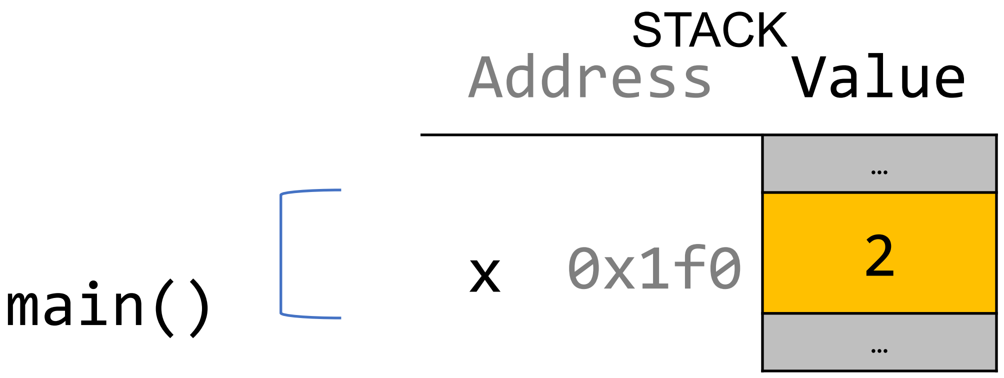
```
void myFunc(int val) {
     val = 3;
}

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



Pointers

Without pointers, we would make copies.

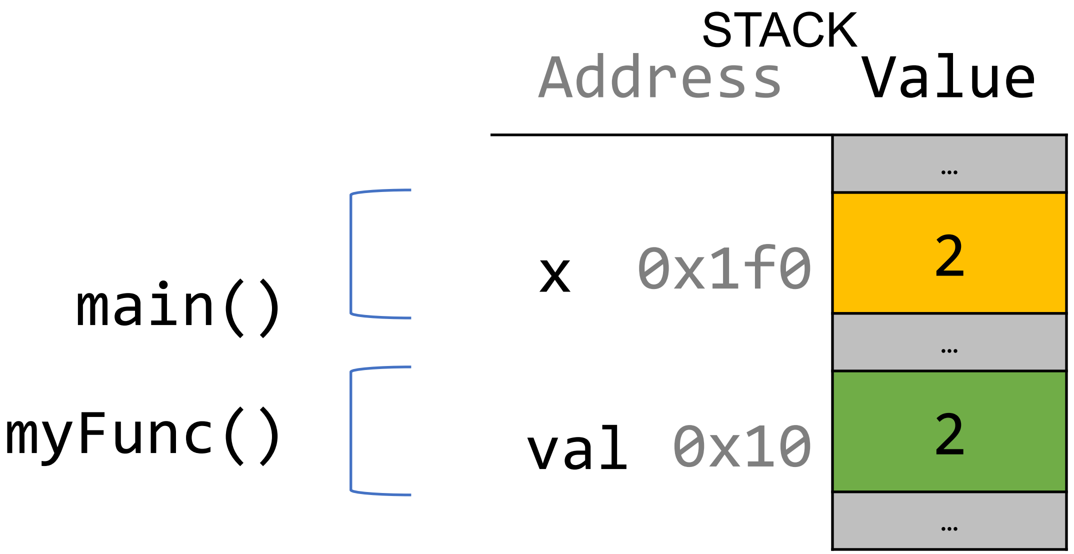
```
void myFunc(int val) {
     val = 3;
}

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



Pointers

Without pointers, we would make copies.

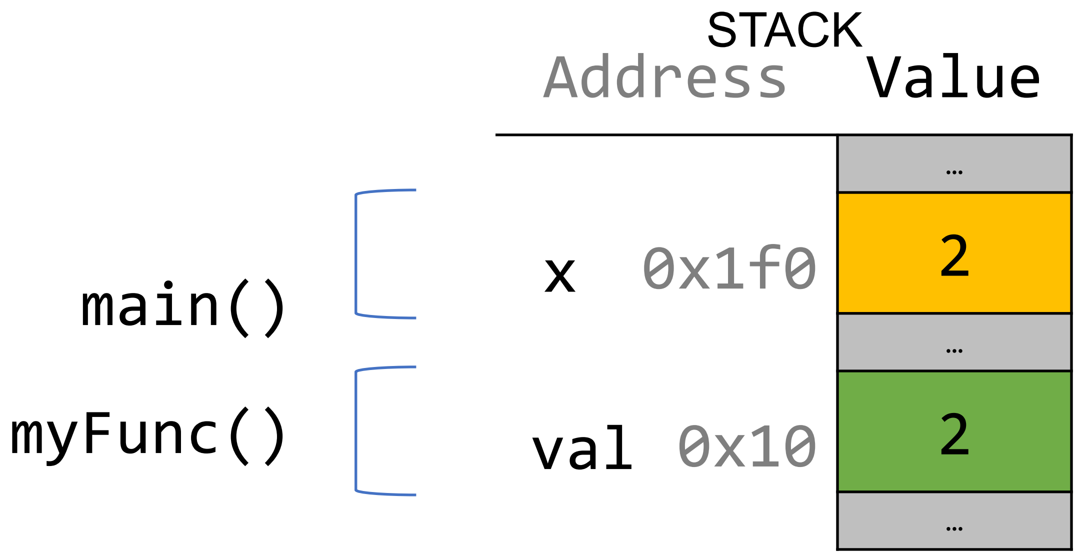
```
void myFunc(int val) {
	val = 3;
}

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



Pointers

Without pointers, we would make copies.

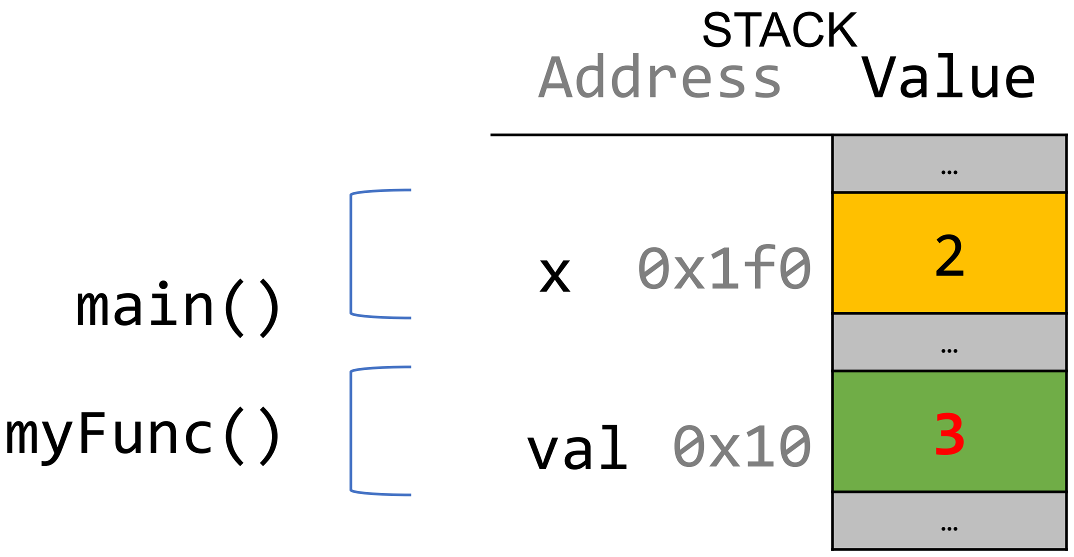
```
void myFunc(int val) {
      val = 3;
}

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



Pointers

Without pointers, we would make copies.

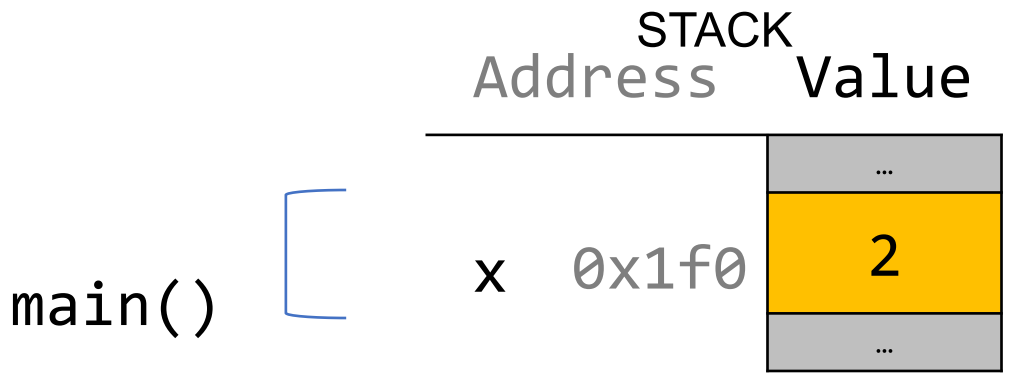
```
void myFunc(int val) {
      val = 3;
}

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



Pointers

Without pointers, we would make copies.

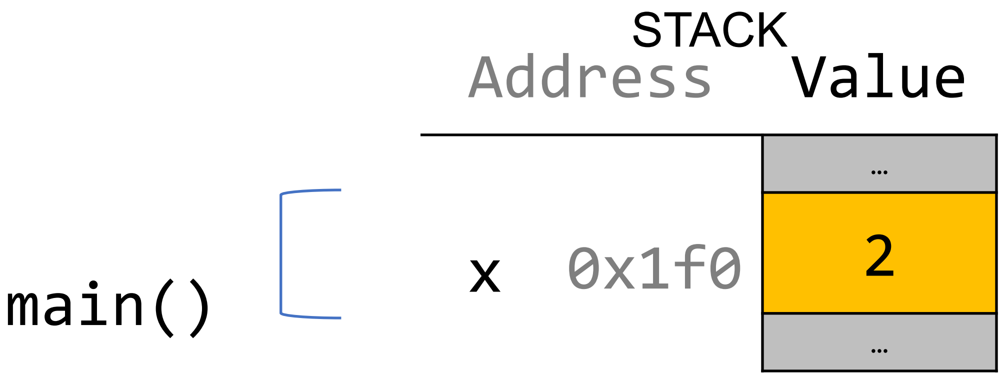
```
void myFunc(int val) {
     val = 3;
}

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



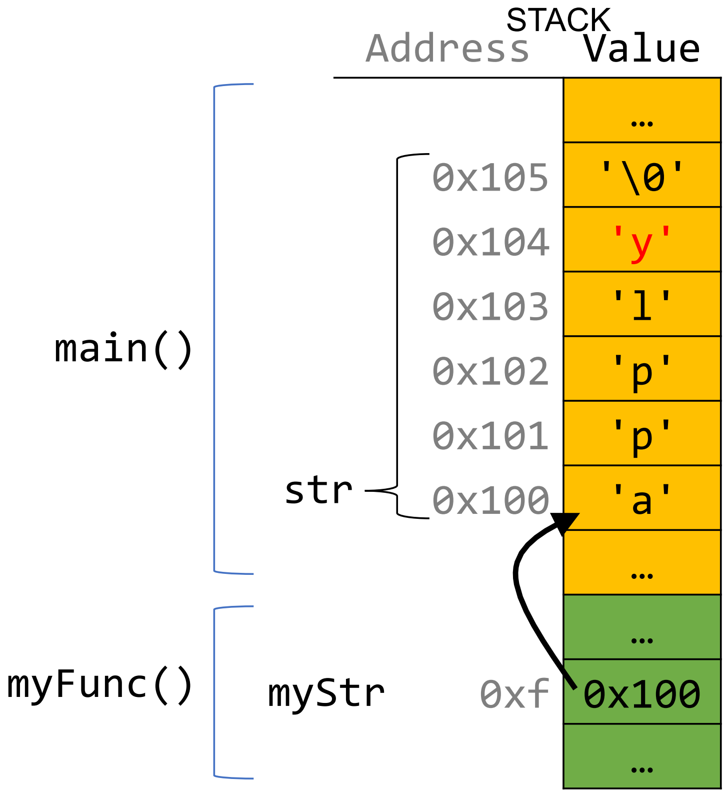
Pointers

Without pointers, we would make copies.

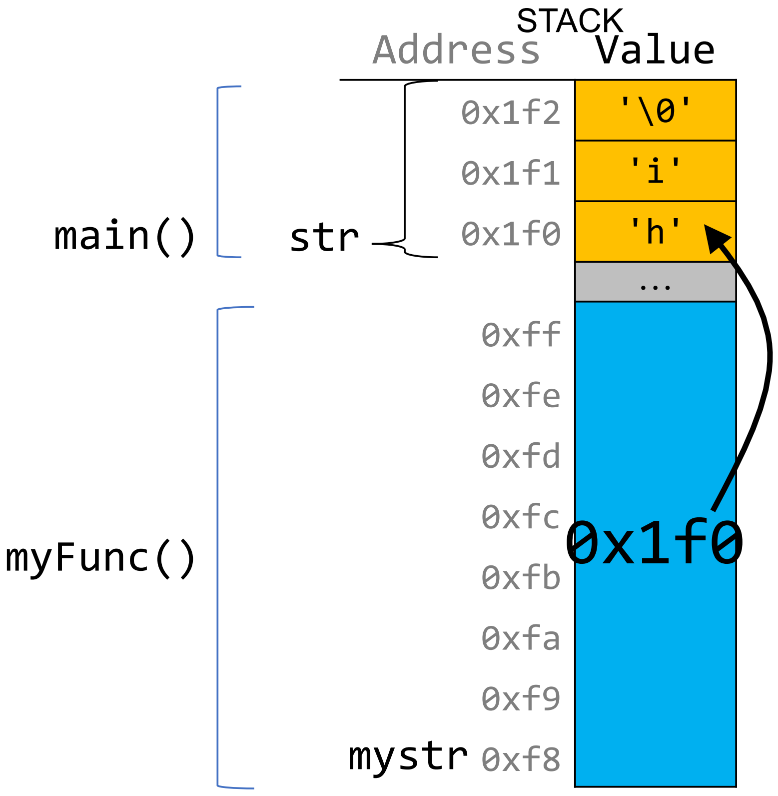
```
void myFunc(int val) {
     val = 3;
}

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



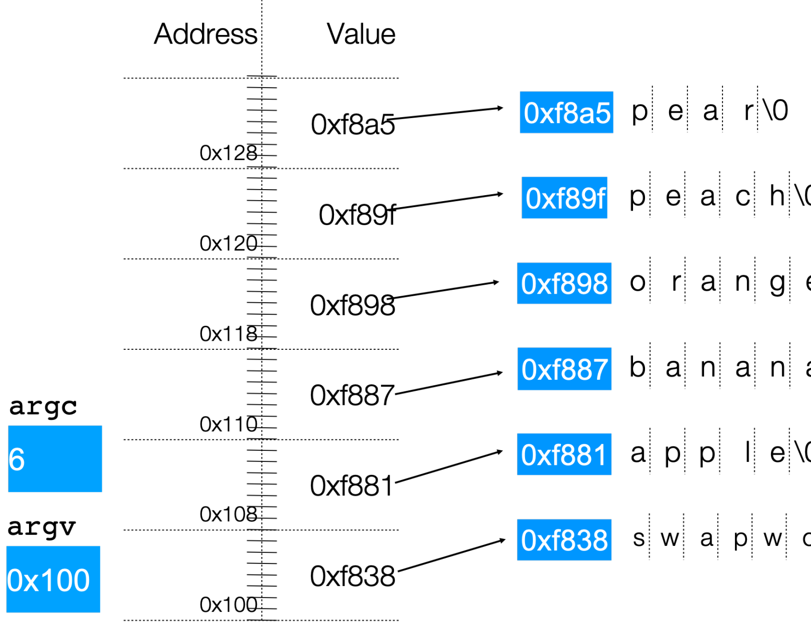
How to draw memory diagrams?



Slide 34



Slide 62



Slide 68

Choose whatever style is convenient for you, keeping in mind that (1) memory is contiguous, and (2) C types are different sizes.

C Parameters

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

```
void myFunction(int x) {
     …
}

int main(int argc, char *argv[]) {
     int num = 4;
     myFunction(num);          // passes copy of 4
}
```

C Parameters

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

```
void myFunction(int *x) {
    …
}

int main(int argc, char *argv[]) {
    int num = 4;
    myFunction(&num);      // passes copy of e.g. 0xffed63
}
```

C Parameters

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

```
void myFunction(char ch) {
     …
}

int main(int argc, char *argv[]) {
     char *myStr = "Hello!";
     myFunction(myStr[1]);          // passes copy of 'e'
}
```

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.

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.

```
void myFunction(char ch) {
    printf("%c", ch);
}

int main(int argc, char *argv[]) {
    char *myStr = "Hello!";
    myFunction(myStr[1]);       // prints 'e'
}
```

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.

```
int myFunction(int num1, int num2) {
    return num1 + num2;
}

int main(int argc, char *argv[]) {
    int x = 5;
    int y = 6;
    int sum = myFunction(x, y);        // returns 11
}
```

C Parameters

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

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

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

```
void capitalize(char *ch) {
    // modifies what is at the address stored in ch
}

int main(int argc, char *argv[]) {
    char letter = 'h';
    /* We don’t want to capitalize any instance of 'h'.
     * We want to capitalize *this* instance of 'h'! */
    capitalize(&letter);
    printf("%c", letter);  // want to print 'H';
}
```

Pointers

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

```
void doubleNum(int *x) {
    // modifies what is at the address stored in x
}

int main(int argc, char *argv[]) {
    int num = 2;
    /* We don’t want to double any instance of 2.
     * We want to double *this* instance of 2! */
    doubleNum(&num);
    printf("%d", num); // want to print 4;
}
```

Pointers

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```
void capitalize(char *ch) {
    // *ch gets the character stored at address ch.
    char newChar = toupper(*ch);

    // *ch = goes to address ch and puts newChar there.
    *ch = newChar;
}
```

Pointers

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```
void capitalize(char *ch) {
     /* go to address ch and put the capitalized version
      * of what is at address ch there. */
      *ch = toupper(*ch);
}
```

Pointers

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```
void capitalize(char *ch) {
    // this capitalizes the address ch! ☹
    char newChar = toupper(ch);

    // this stores newChar in ch as an address! ☹
    ch = newChar;
}
```

Lecture Plan

- Pointers and Parameters
- **Strings in Memory**
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

char *

- A char * is technically a pointer to a **<u>single character</u>**.
- We commonly use char * as string by having the character it points to be followed by more characters and ultimately a null terminator.
- A char * could also just point to a single character (not a string).

Strings In Memory

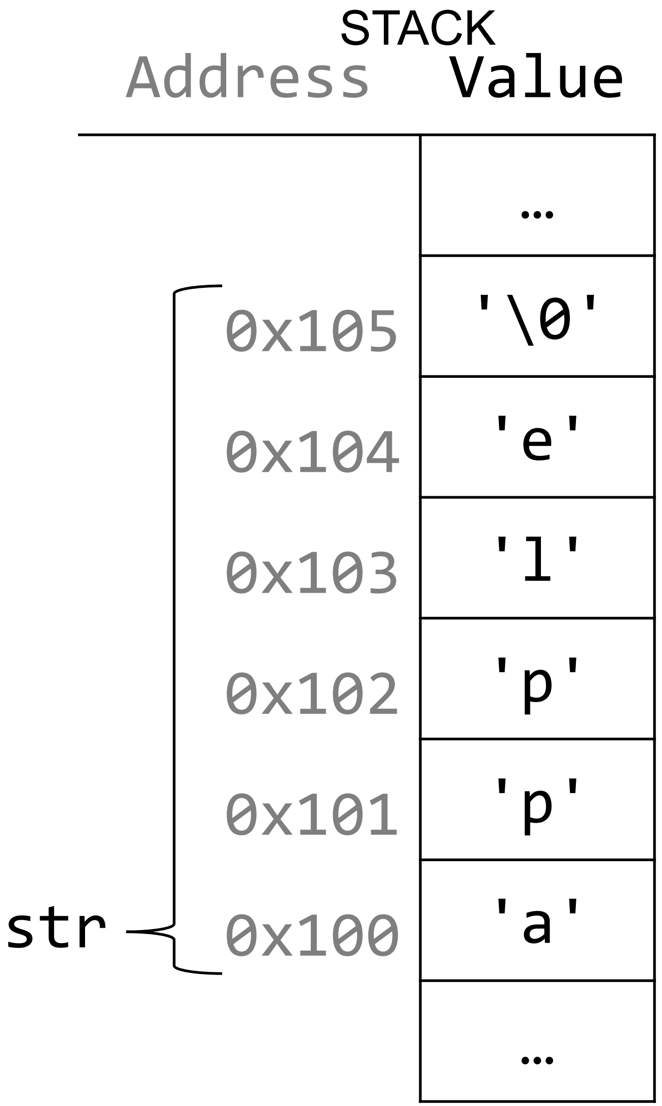
1. If we create a string as a **char[]**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **char[]** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **char[]** as a parameter, 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.

Character Arrays

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");
```



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!
```

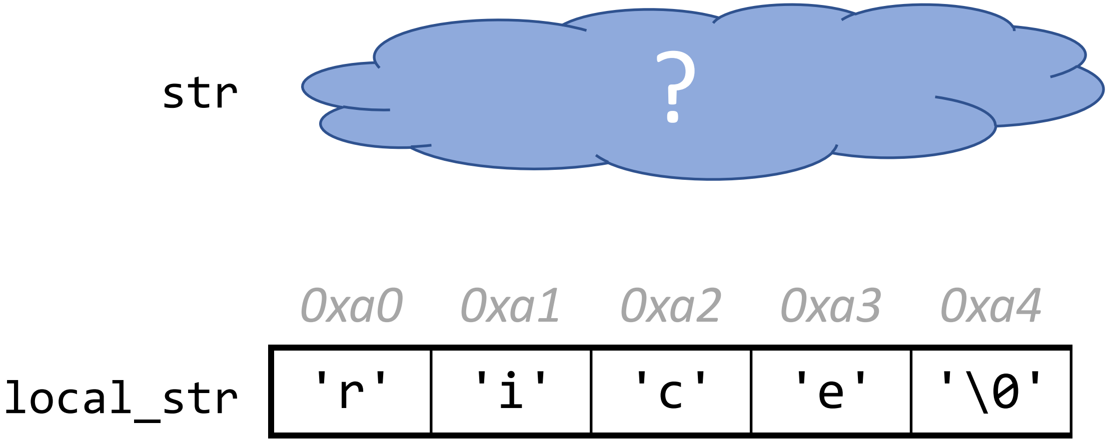
An array's size cannot be changed once we create it; we must create another new array instead.

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 ***.

String Parameters

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;
}
```

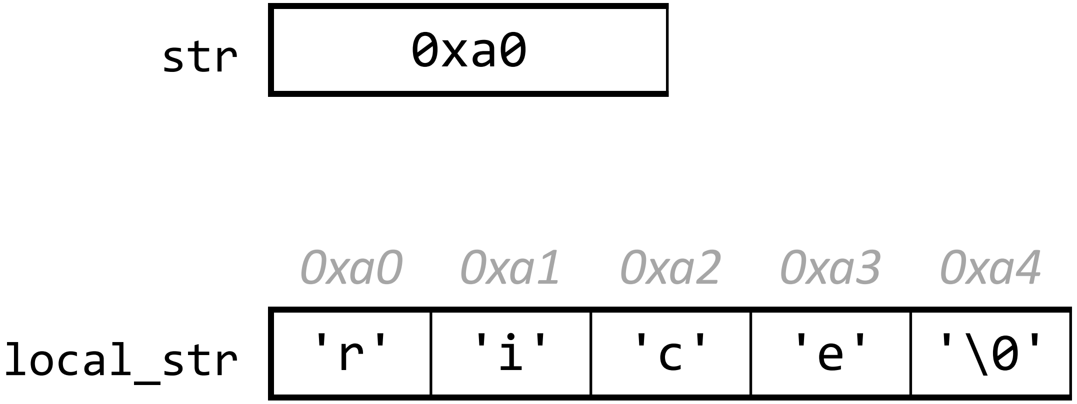


A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`

String Parameters

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

```
void fun_times(char *str) {
      ...
}

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

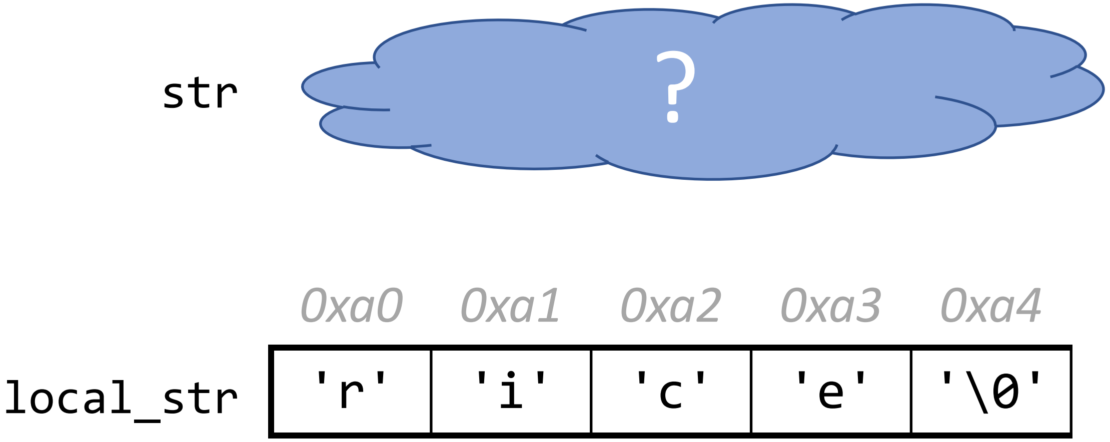


A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`

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;
}
```



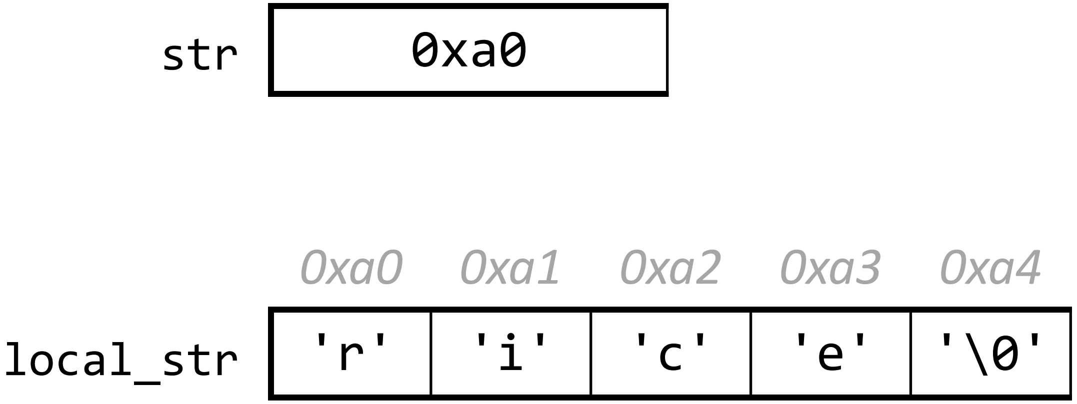
A. A copy of the array `local_str`

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;
}
```

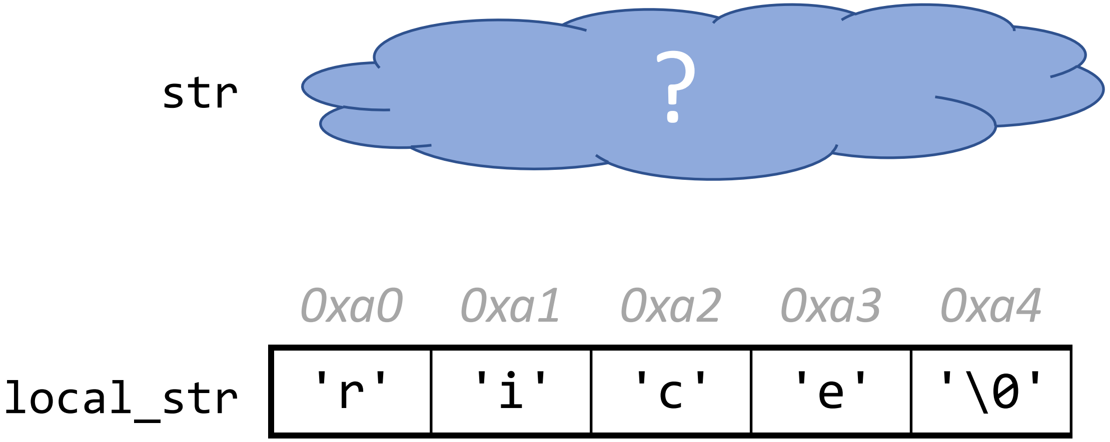


A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`

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;
}
```



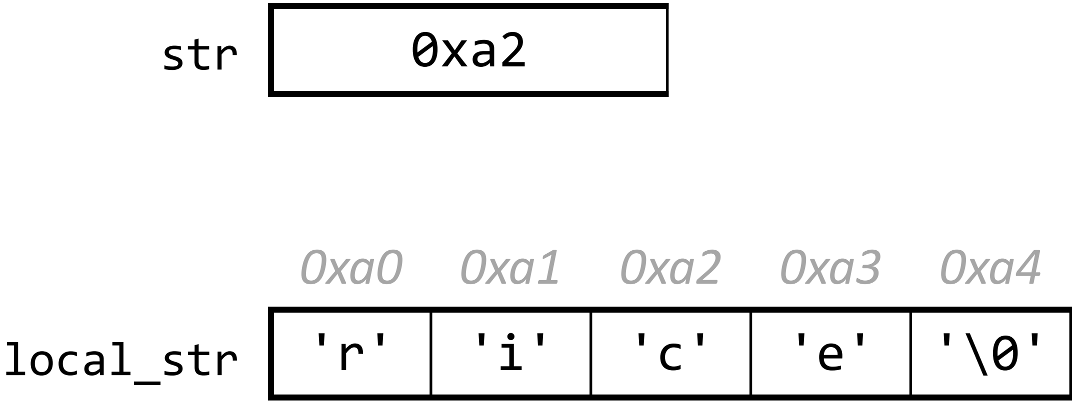
A. A copy of part of the array `local_str`

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?

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

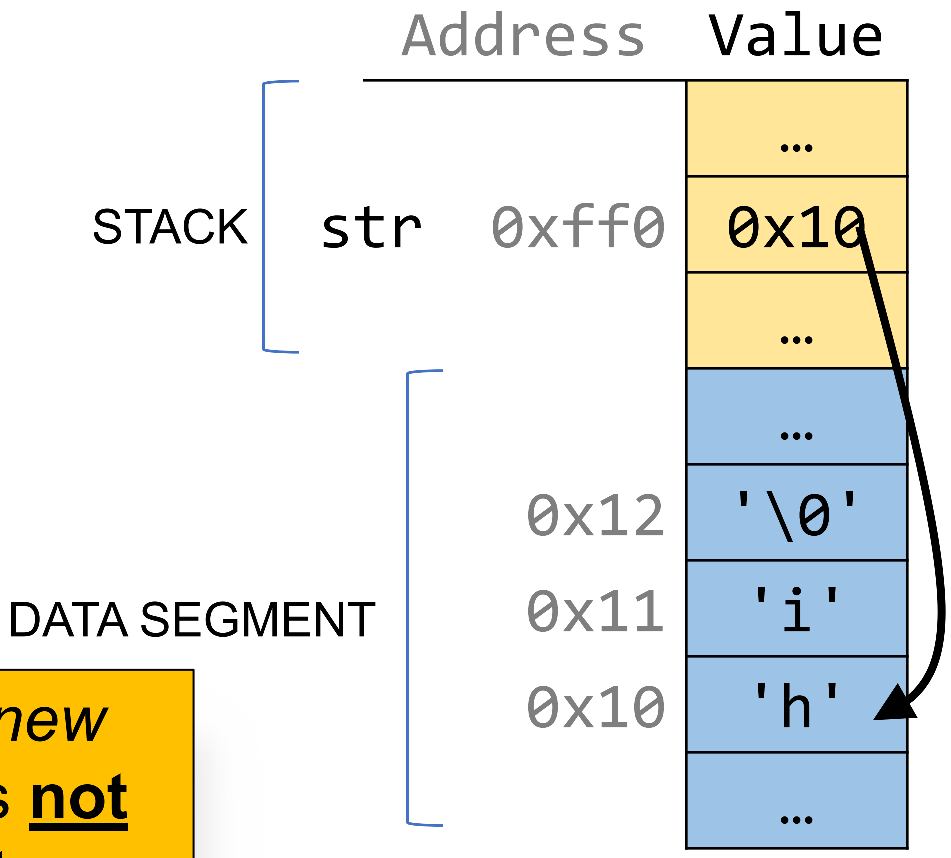
char *

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.



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?

Memory Locations

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**?

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

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

Memory Locations

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;
}
```

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

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 *

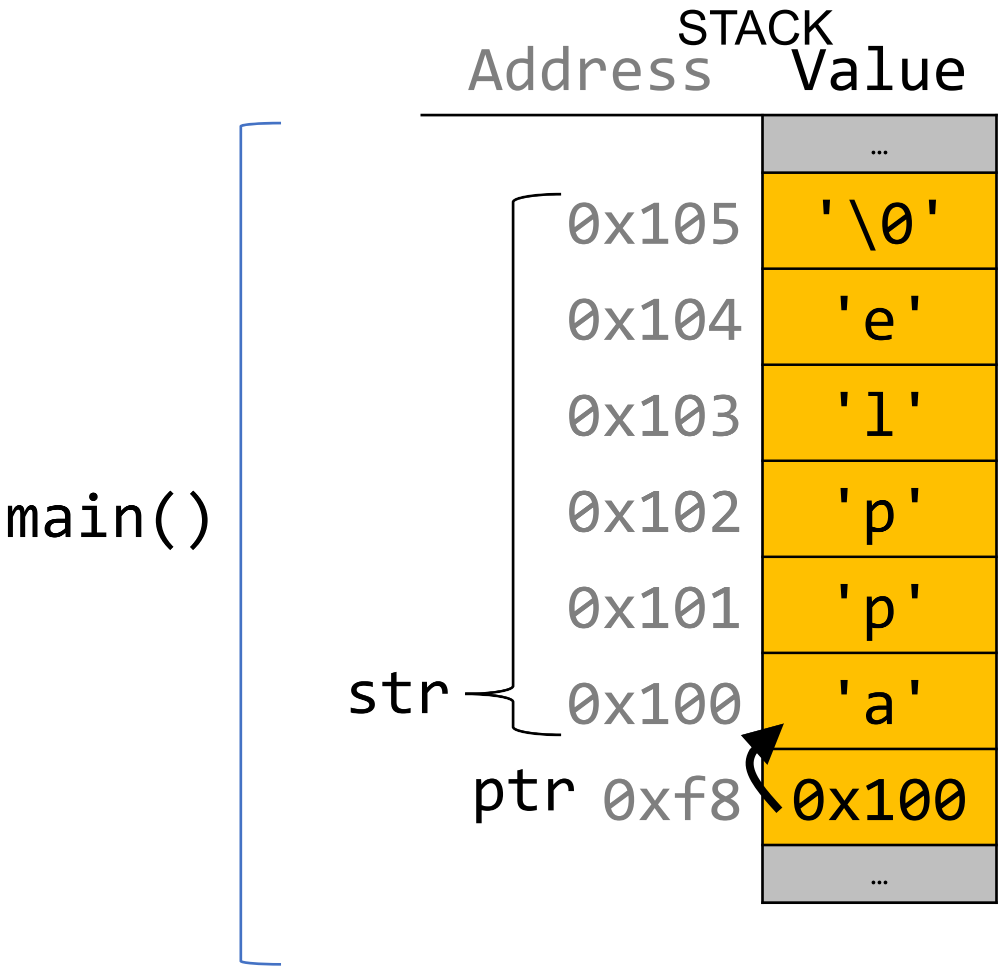
A **char *** variable refers to a single character. We can reassign an existing **char *** pointer to be equal to another **char *** pointer.

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

Arrays and Pointers

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

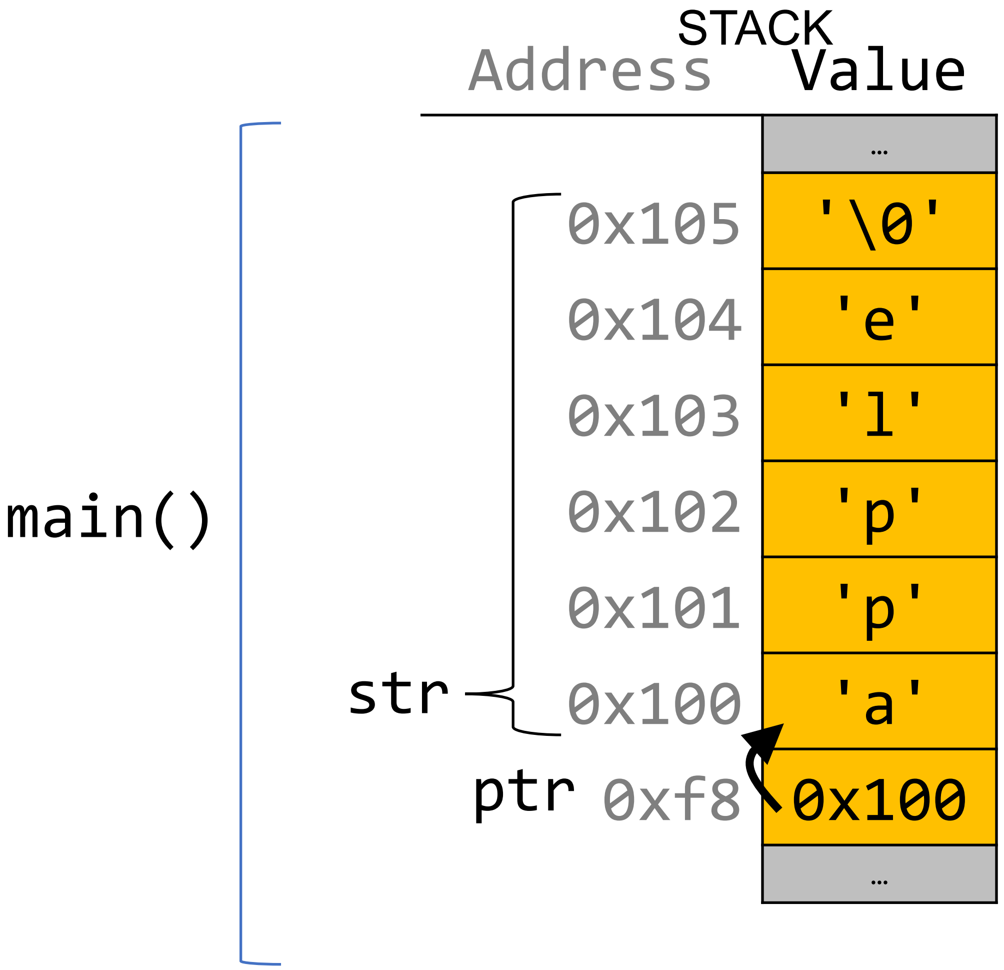
```
int main(int argc, char *argv[]) {
	char str[6];
	strcpy(str, "apple");
	char *ptr = str;
	...
}
```



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;

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

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



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

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'
	…

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

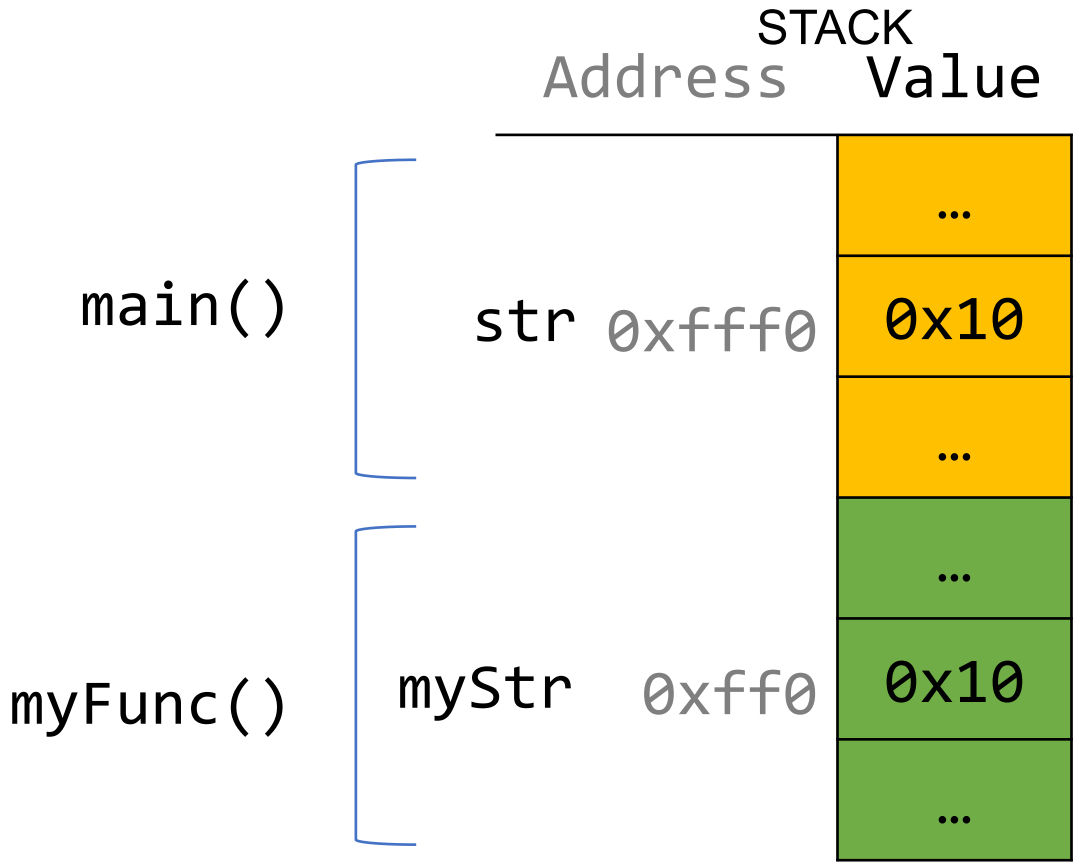
Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

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

Strings as Parameters

When we pass a **char *** string as a parameter, C makes a *copy* of the address stored in the **char *** and passes it to the function. This means they both refer to the same memory location.

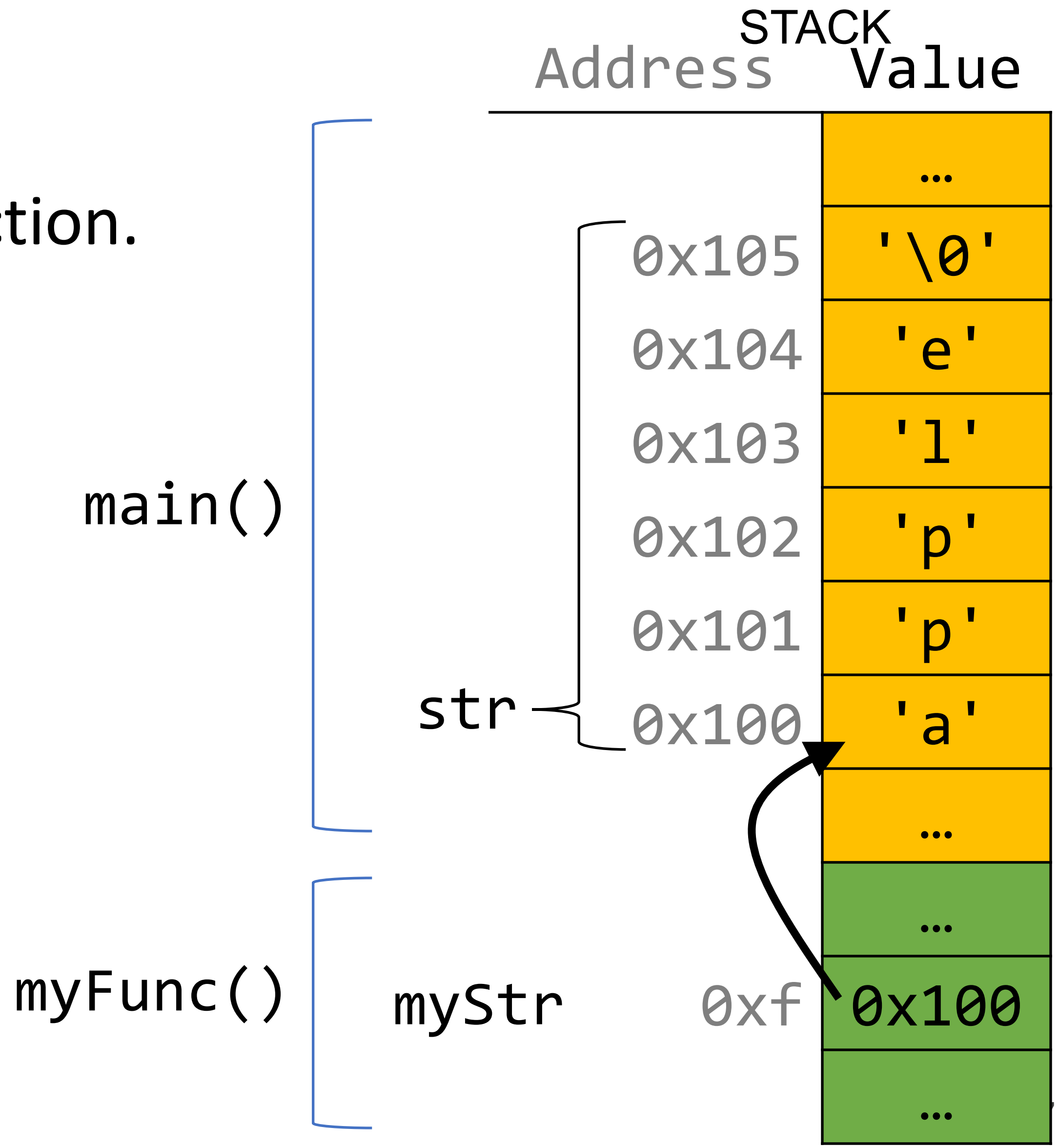
```
void myFunc(char *myStr) {
     ...
}

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



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.

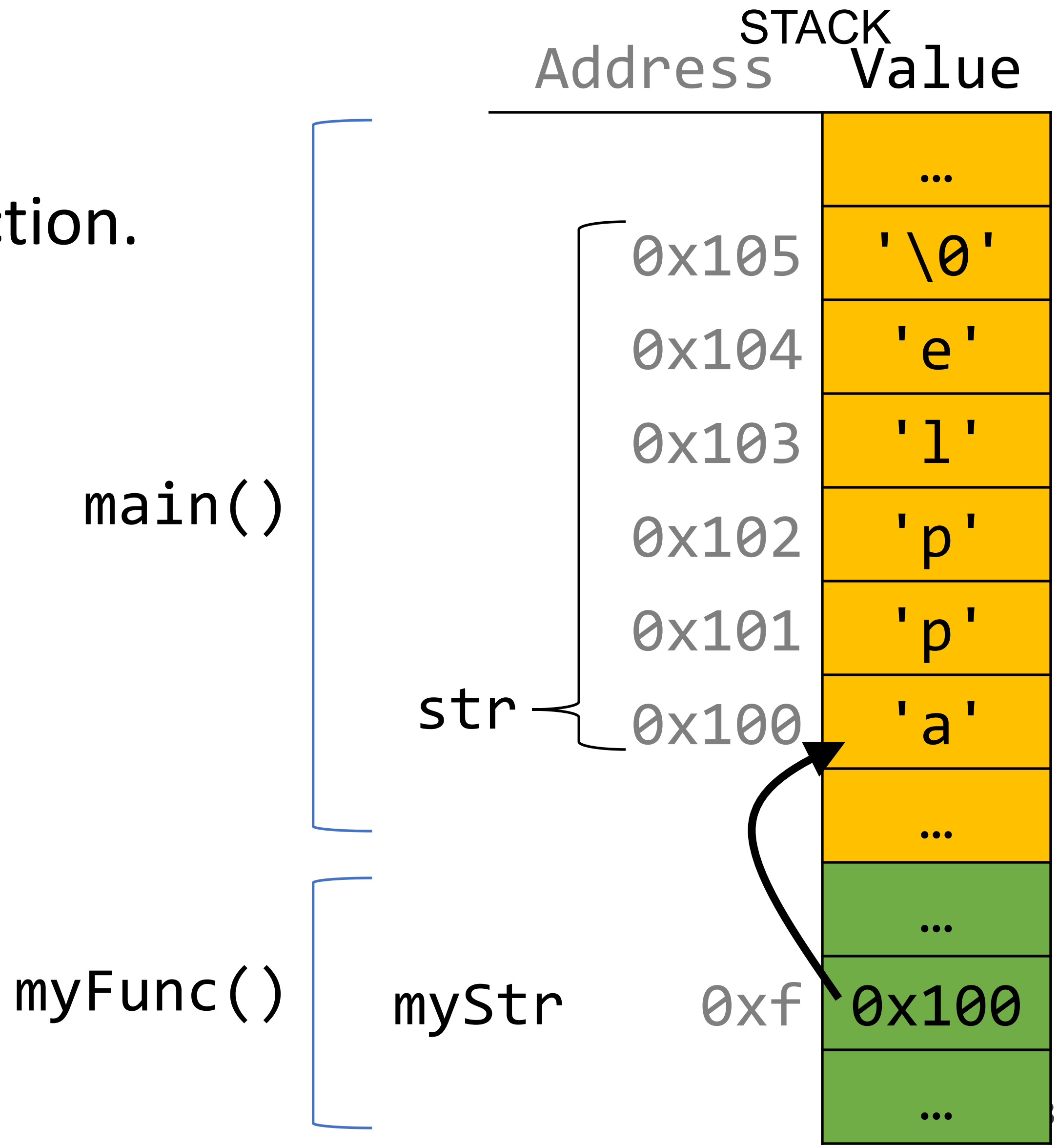
```
void myFunc(char *myStr) {
    ...
}

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



Strings as Parameters

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.

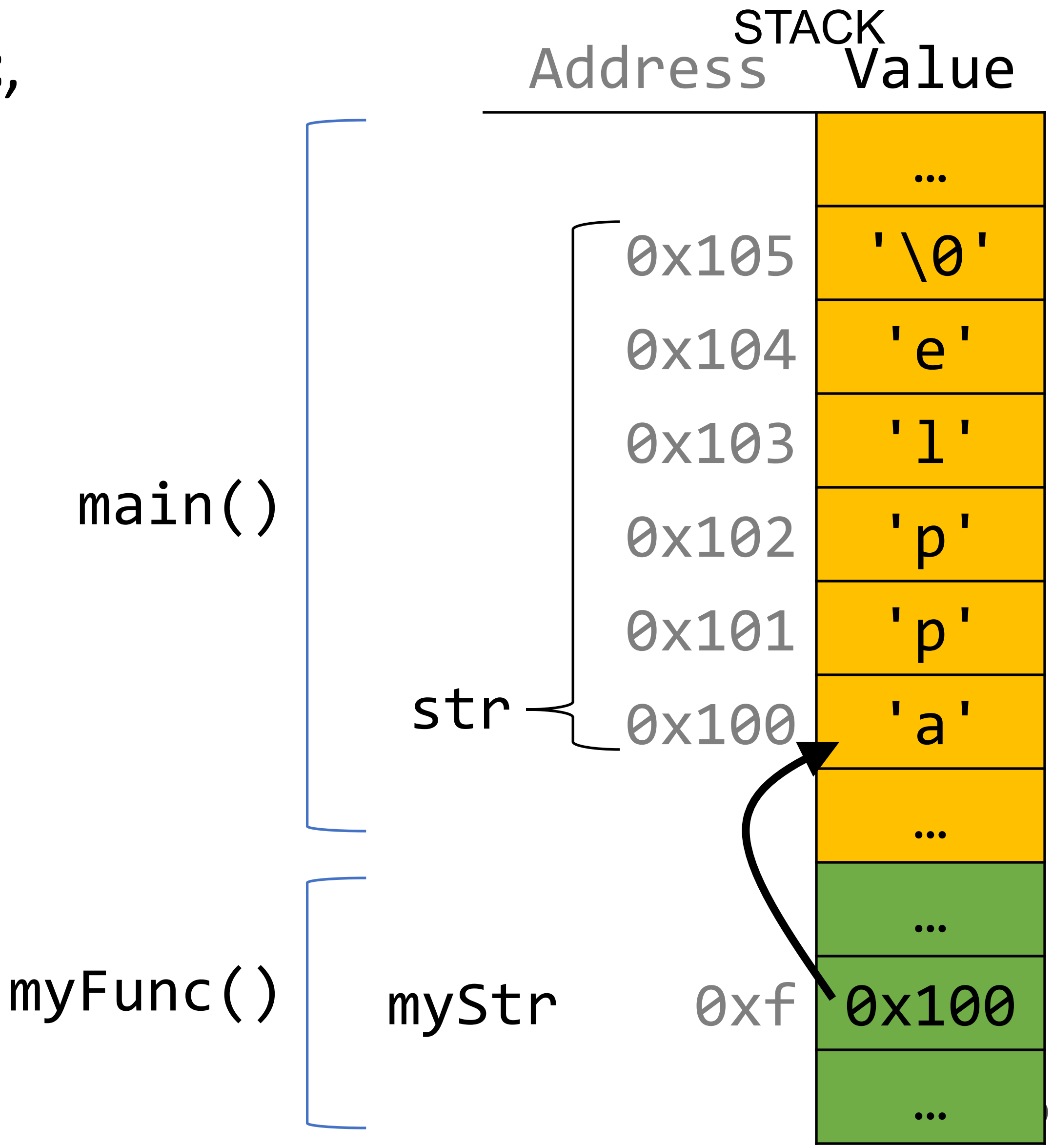
```
void myFunc(char *myStr) {
    ...
}

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



Strings as Parameters

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

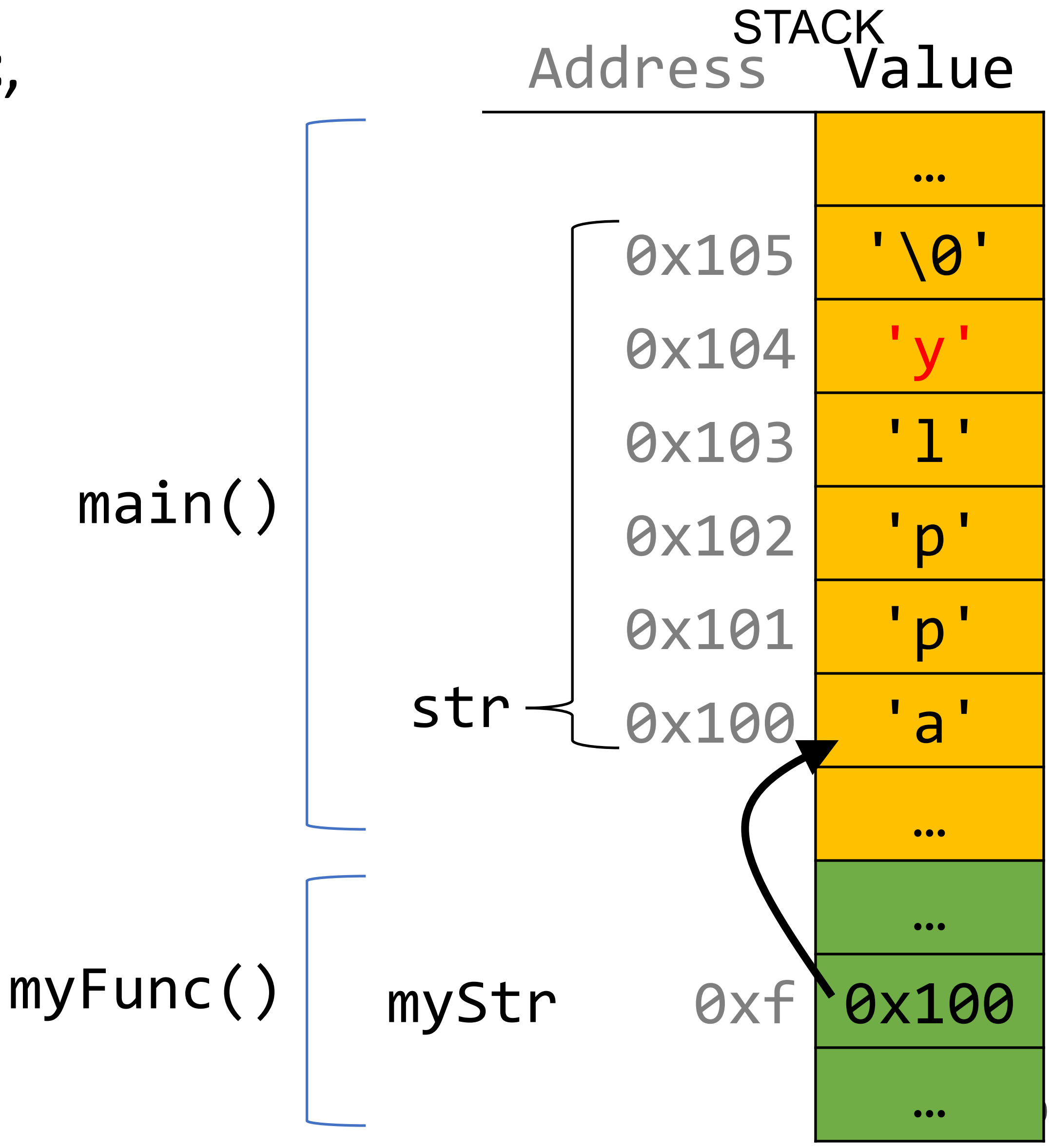
```
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

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



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
      ...
}
```



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.

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.
- &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

CS107 Topic 3: How can we effectively manage all types of memory in our programs?

Lecture Plan

- Pointers and Parameters
- Strings in Memory
- **Double Pointers**
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/lecture-code/lect6 .
```

Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```
void printSquare(__?__) {
    int square = __?__ * __?__;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(__?__);     // should print 9
}
```

Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```
void printSquare(int x) {
     int square = x * x;
     printf("%d", square);
}

int main(int argc, char *argv[]) {
     int num = 3;
     printSquare(num);  // should print 9
}
```

We are performing a calculation with some input and do not care about any changes to the input, so we pass the data type itself.

Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```
void printSquare(int x) {
    x = x * x;
    printf("%d", x);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // should print 9
}
```

We are performing a calculation with some input and do not care about any changes to the input, so we pass the data type itself.

Exercise 2

We want to write a function that flips the case of a letter. What should go in each of the blanks?

```
void flipCase(__?__) {
      if (isupper(__?__)) {
            __?__ = __?__;
      } else if (islower(__?__)) {
            __?__ = __?__;
      }
}

int main(int argc, char *argv[]) {
      char ch = 'g';
      flipCase(__?__);
      printf("%c", ch);     // want this to print ‘G’
}
```

Exercise 2

We want to write a function that flips the case of a letter. What should go in each of the blanks?

```
void flipCase(char *letter) {
	if (isupper(*letter)) {
		*letter = tolower(*letter);
	} else if (islower(*letter)) {
		*letter = toupper(*letter);
	}
}

int main(int argc, char *argv[]) {
	char ch = 'g';
	flipCase(&ch);
	printf("%c", ch);     // want this to print ‘G’
```

> We are modifying a specific instance of the letter, so we pass the *location* of the letter we would like to modify.

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**.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

Pointers Summary

Tip: setting a function parameter equal to a new value usually doesn't do what you want. Remember that this is setting the function's *own copy* of the parameter equal to some new value.

```
void doubleNum(int x) {
    x = x * x;     // modifies doubleNum's own copy!
}

void advanceStr(char *str) {
    str += 2;      // modifies advanceStr's own copy!
}
```

Exercise 3

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(__?__) {
     ...
}

int main(int argc, char *argv[]) {
     char *str = "    hello";
     skipSpaces(__?__);
     printf("%s", str);      // should print "hello"
}
```

Exercise 3

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(char **strPtr) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(&str);
    printf("%s", str);      // should print "hello"
}
```

We are modifying a specific instance of the string pointer, so we pass the *location* of the string pointer we would like to modify.

Exercise 3

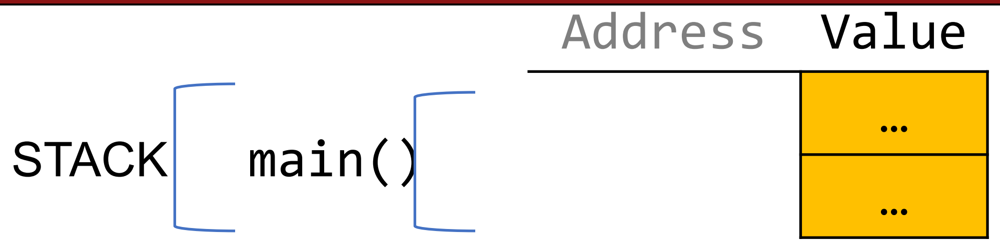
Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(char *strPtr) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(str);
    printf("%s", str);    // should print "hello"
}
```

This advances skipSpace's own copy of the string pointer, not the instance in main.

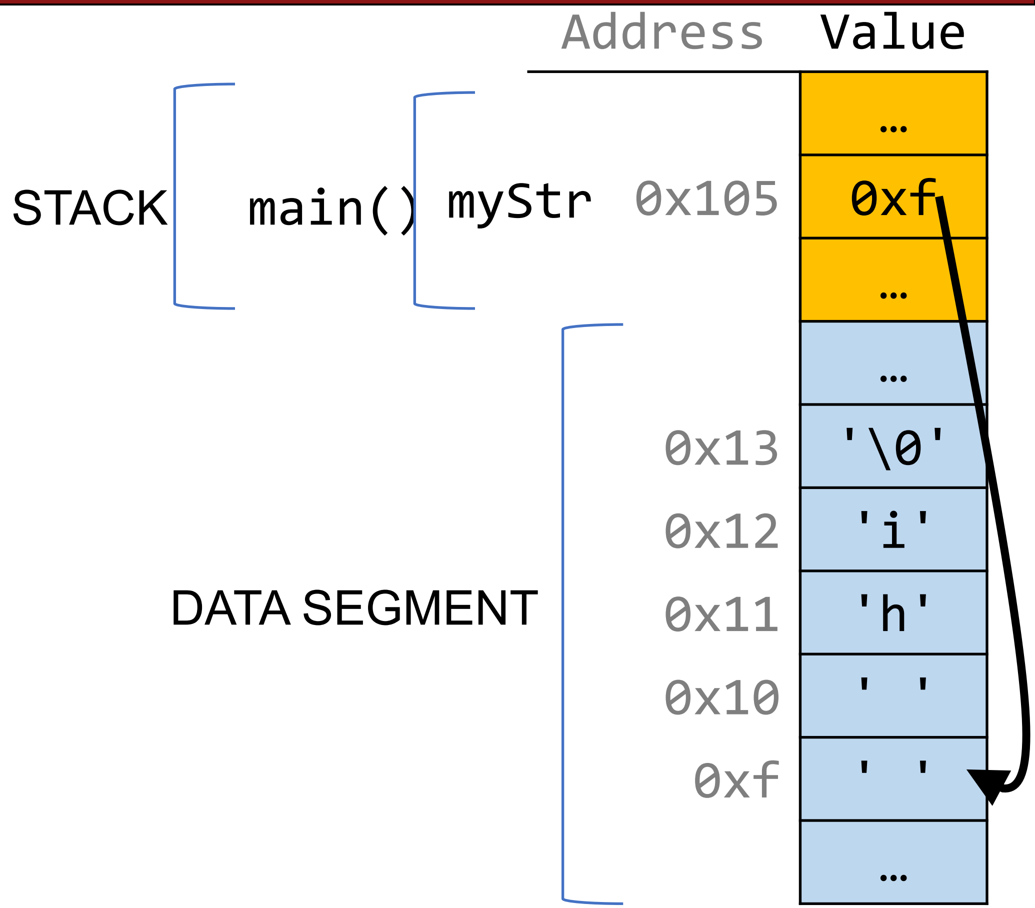
Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```



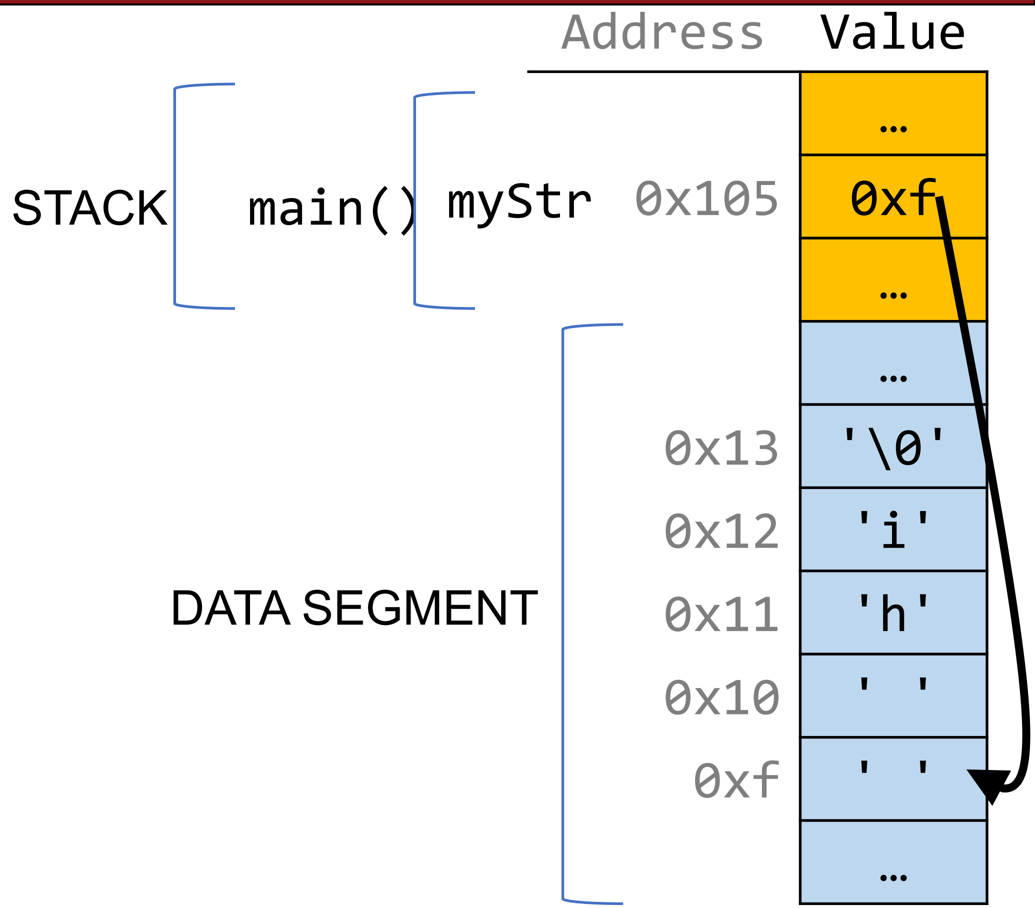
Pointers to Strings

```c
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);      // hi
    return 0;
}
```



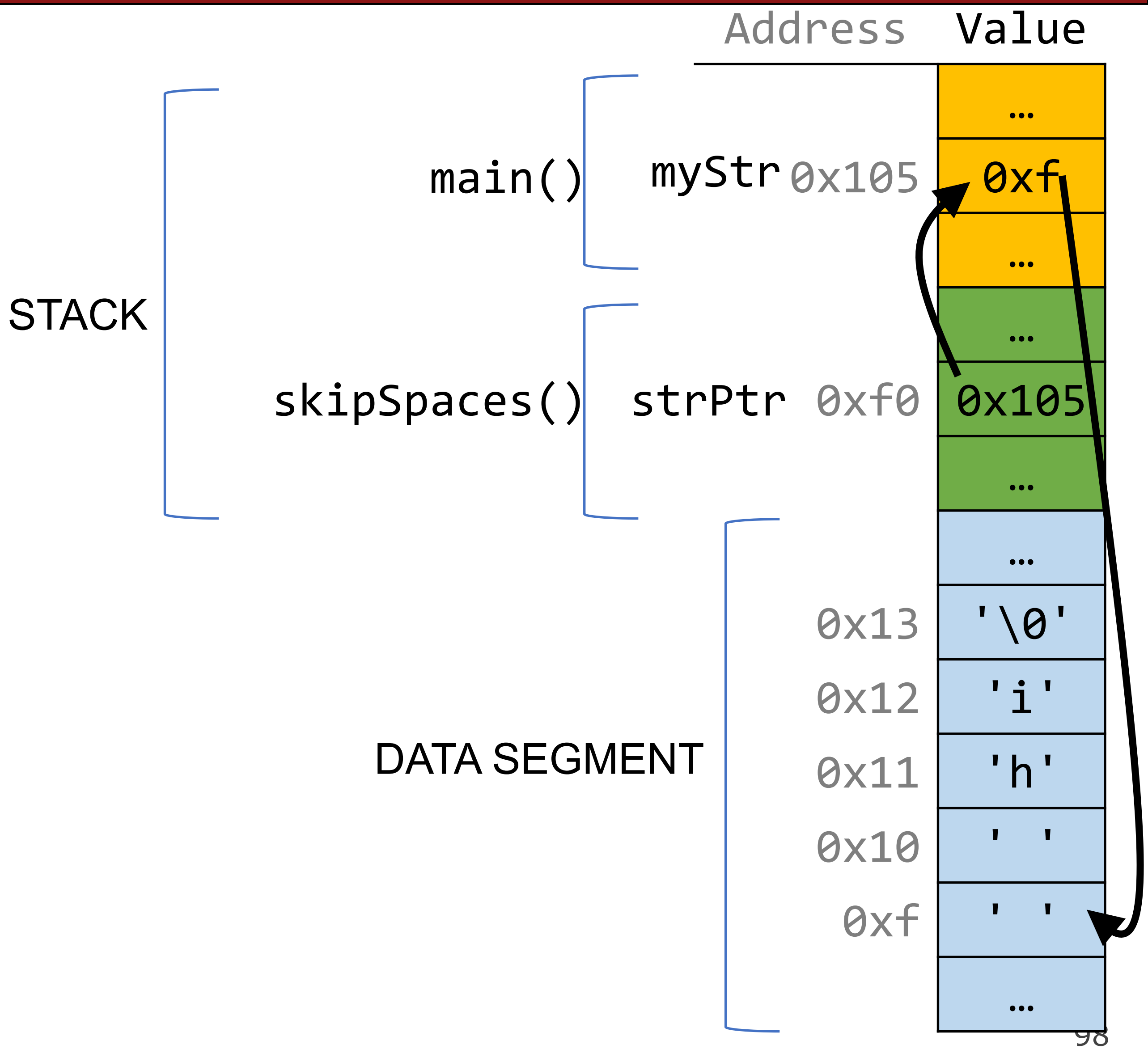
Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);      // hi
    return 0;
}
```



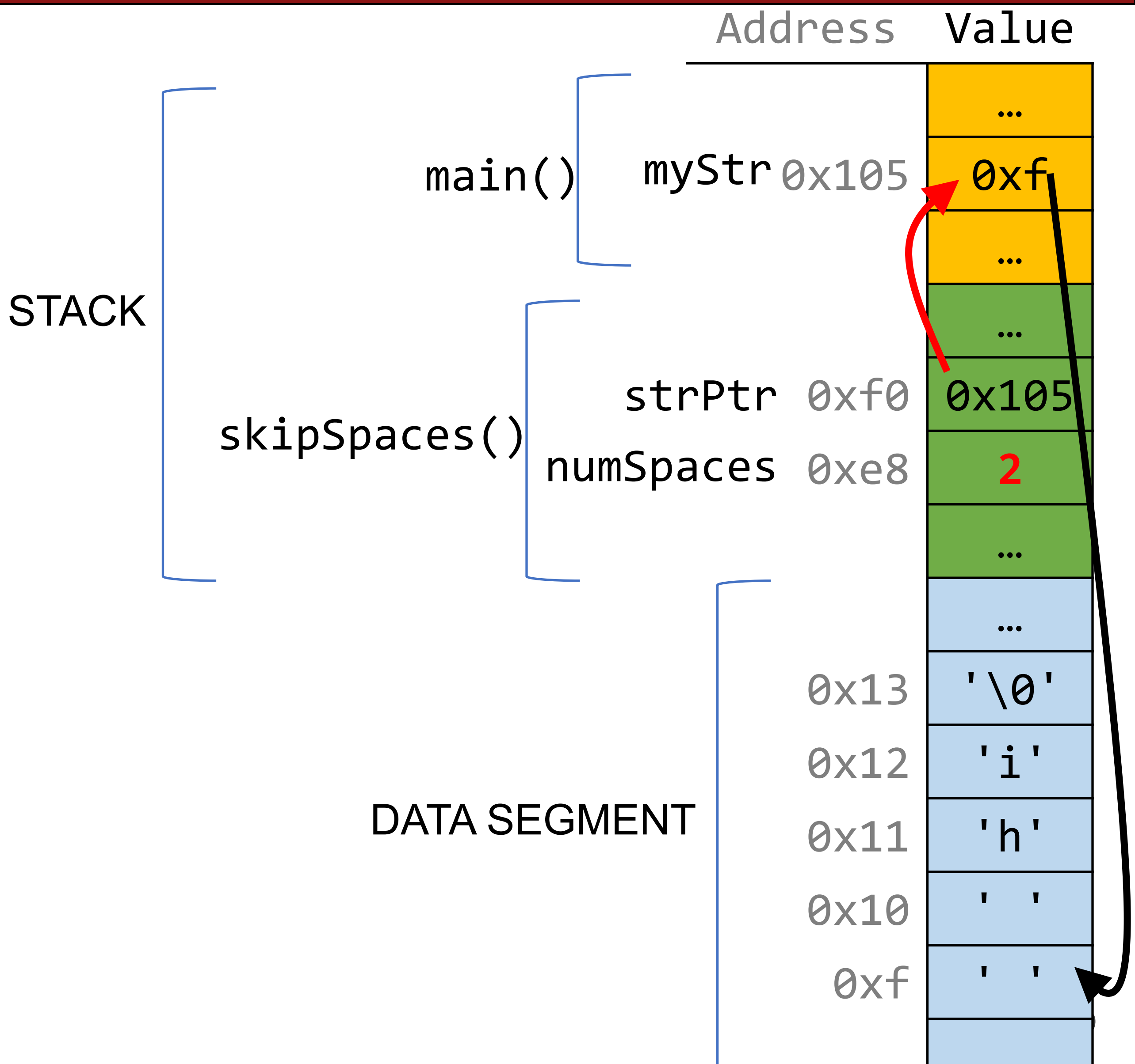
Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```



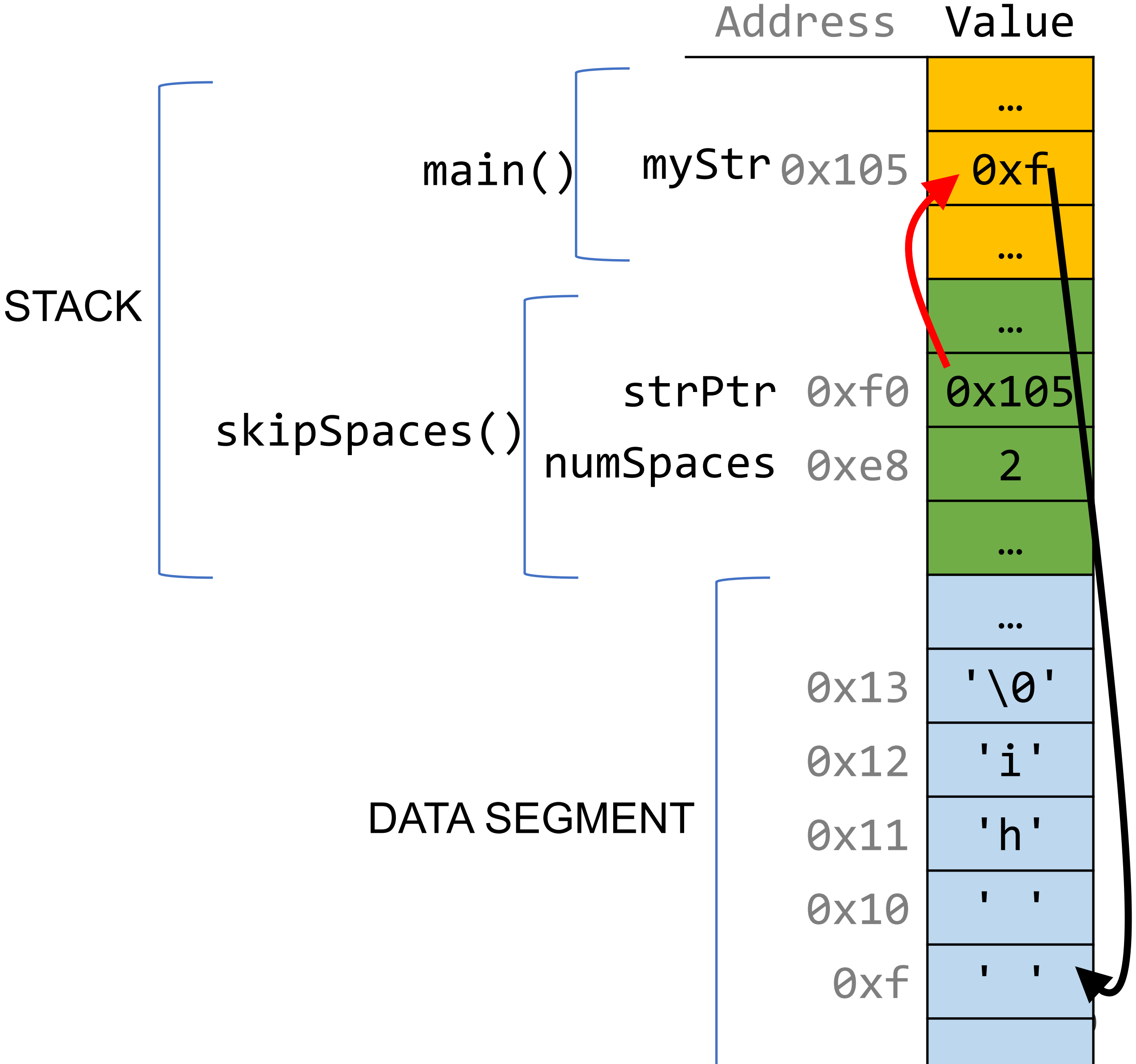
Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```



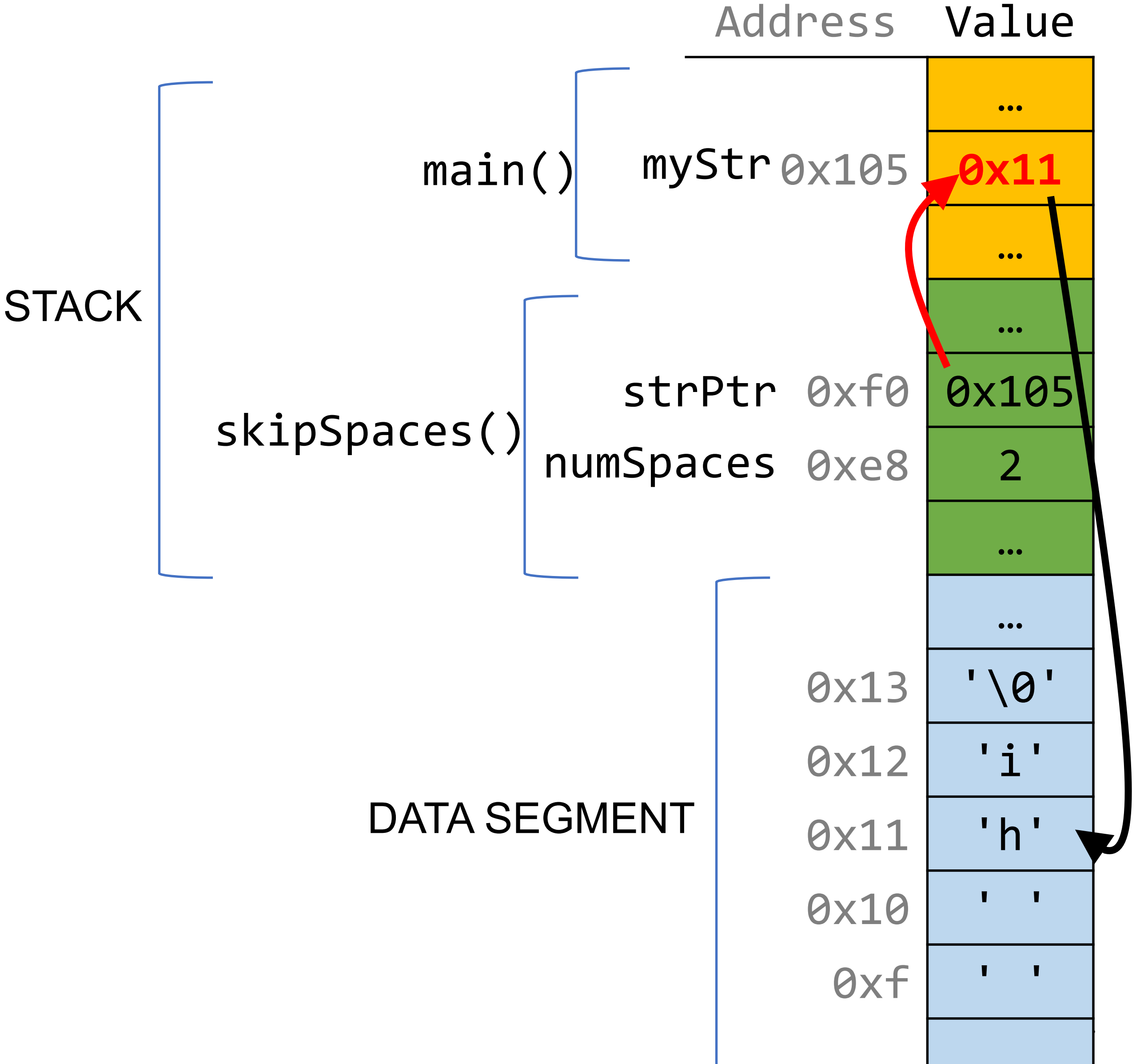
Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```



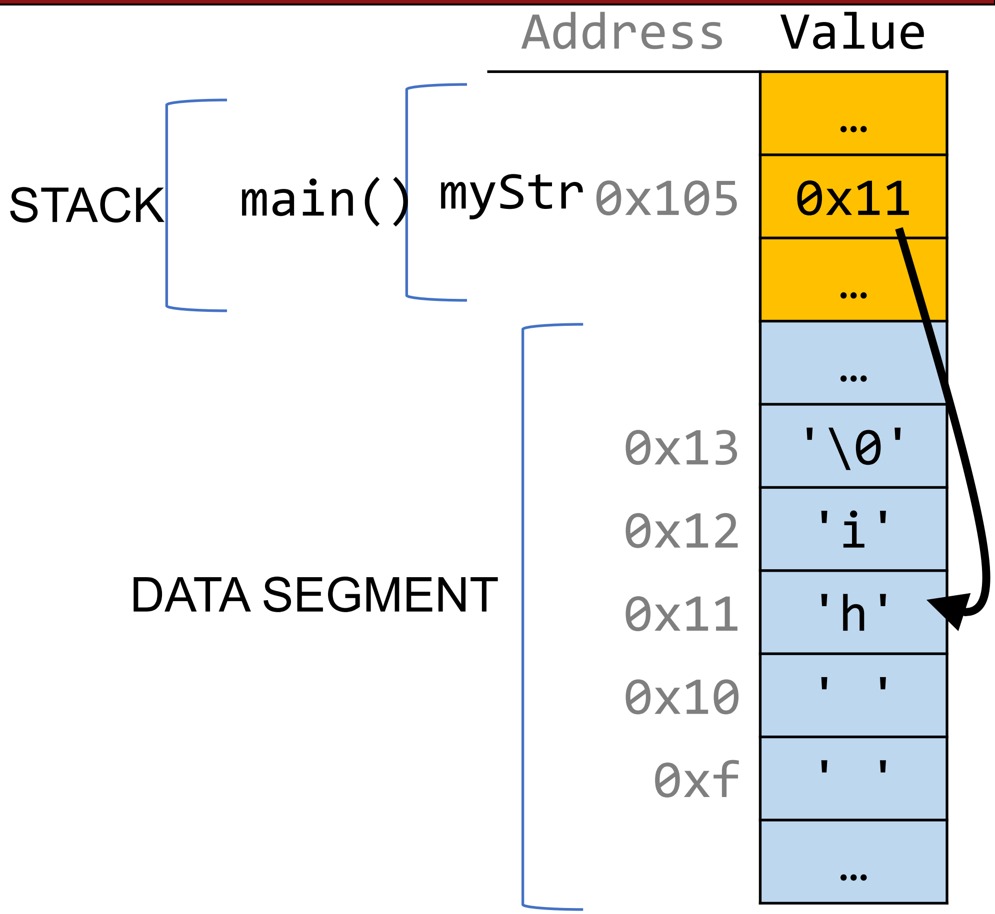
Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```



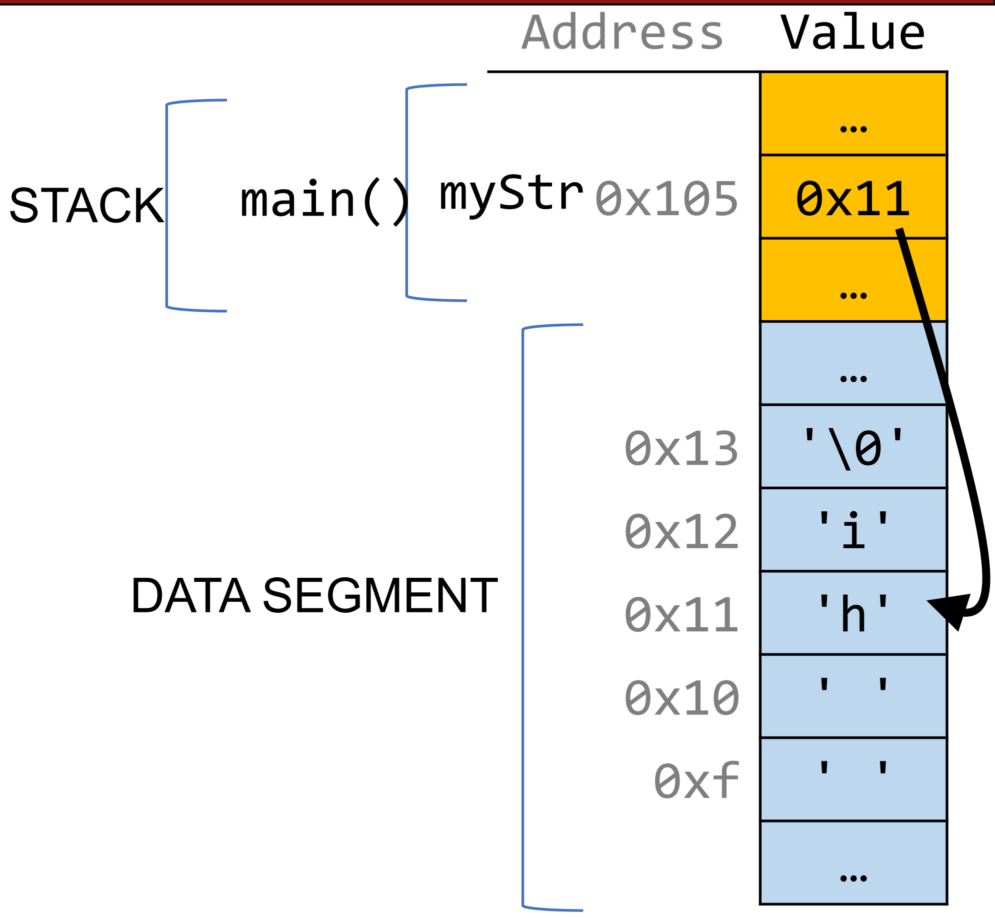
Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```



Pointers to Strings

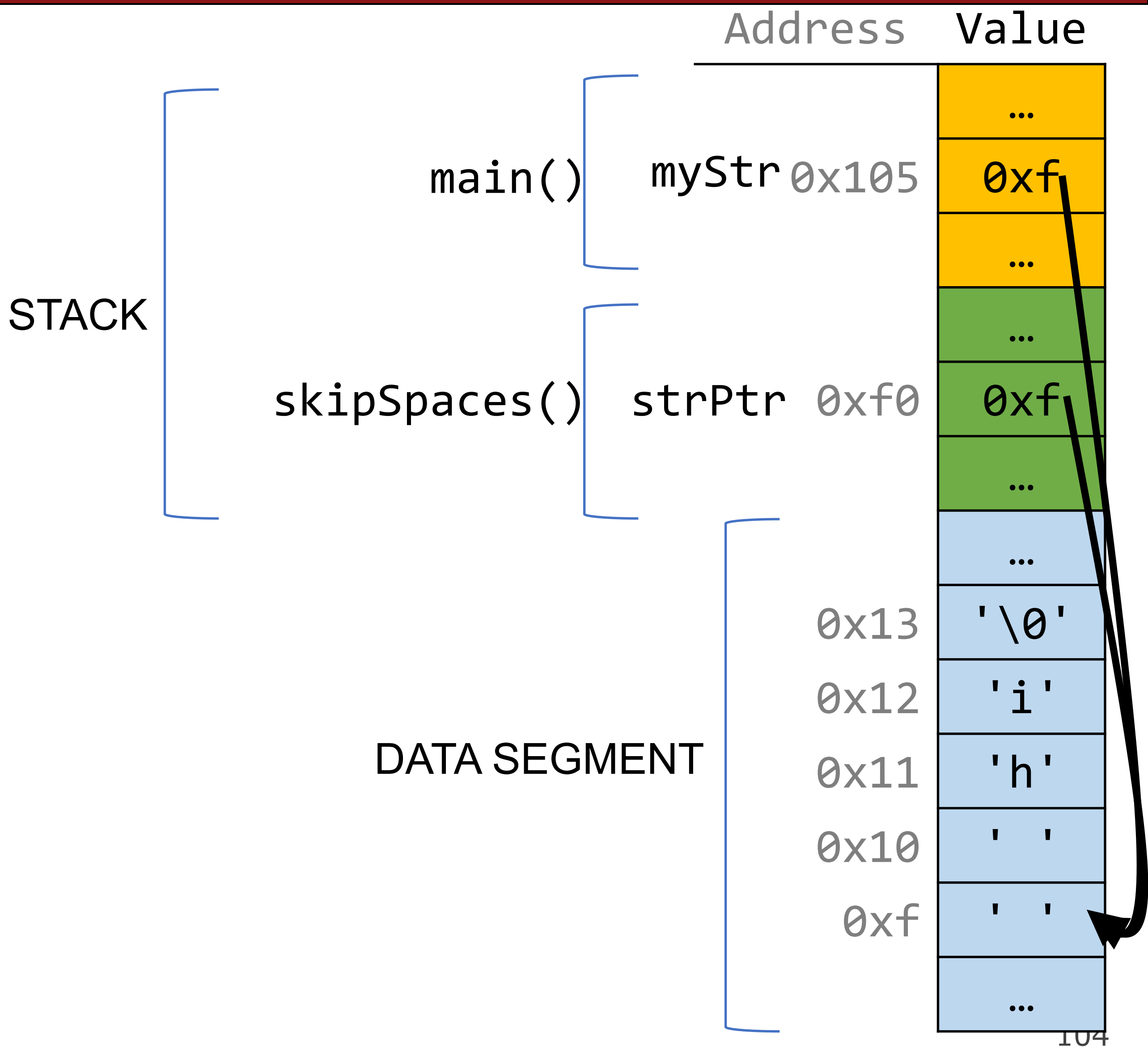
```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```



Weird thought – **0x11** *is a string*.

Making Copies

```
void skipSpaces(char *strPtr) {
    int numSpaces = strspn(strPtr, " ");
    strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(myStr);
    printf("%s\n", myStr);        //   hi
    return 0;
}
```



Lecture Plan

- Pointers and Parameters
- Strings in Memory
- Double Pointers
- **Arrays in Memory**
- Arrays of Pointers
- Pointer Arithmetic
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/lecture-code/lect06 .
```

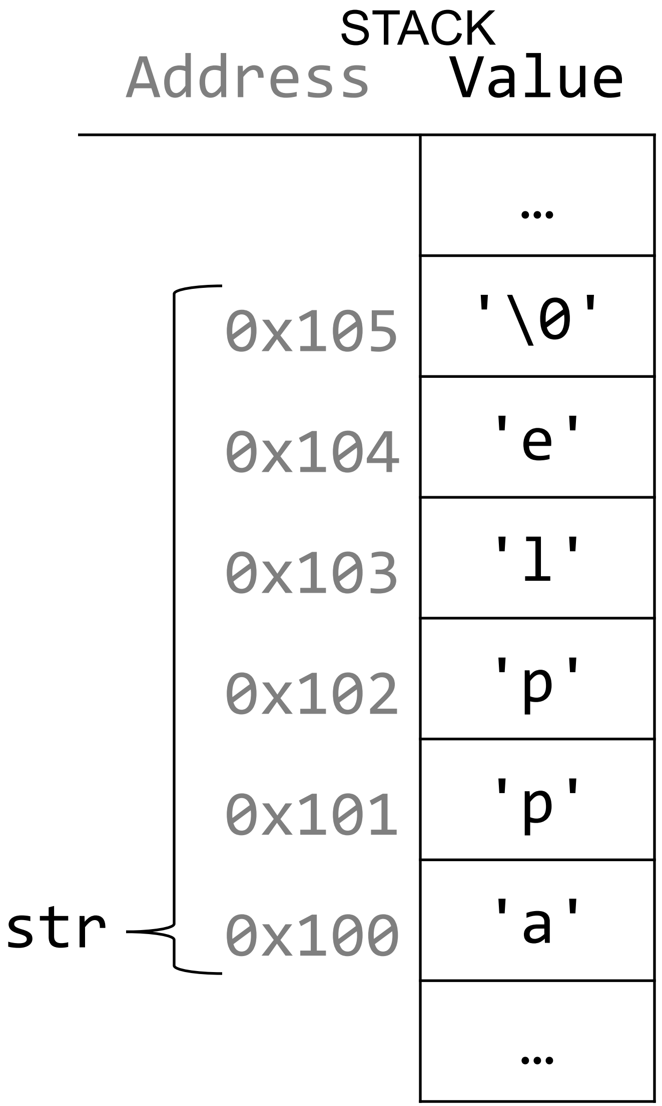
Arrays

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

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

The array variable (e.g. **str**) is not a pointer; it refers to the entire array contents. In fact, **sizeof** returns the size of the entire array!

```
int arrayBytes = sizeof(str);    // 6
```



Arrays

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

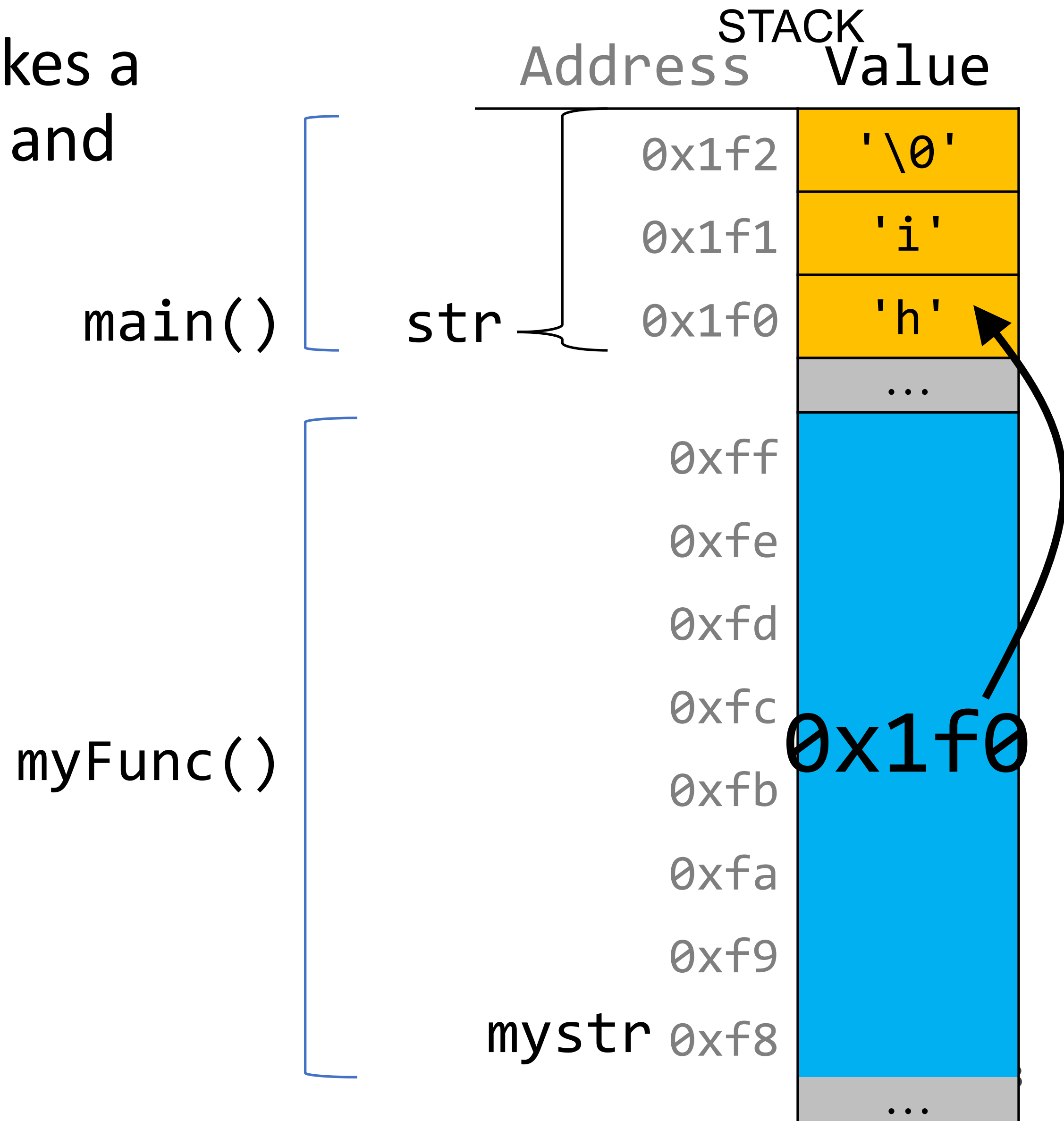
```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

An array's size cannot be changed once you create it; you must create another new array instead.

Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element*, and passes it (a pointer) to the function.

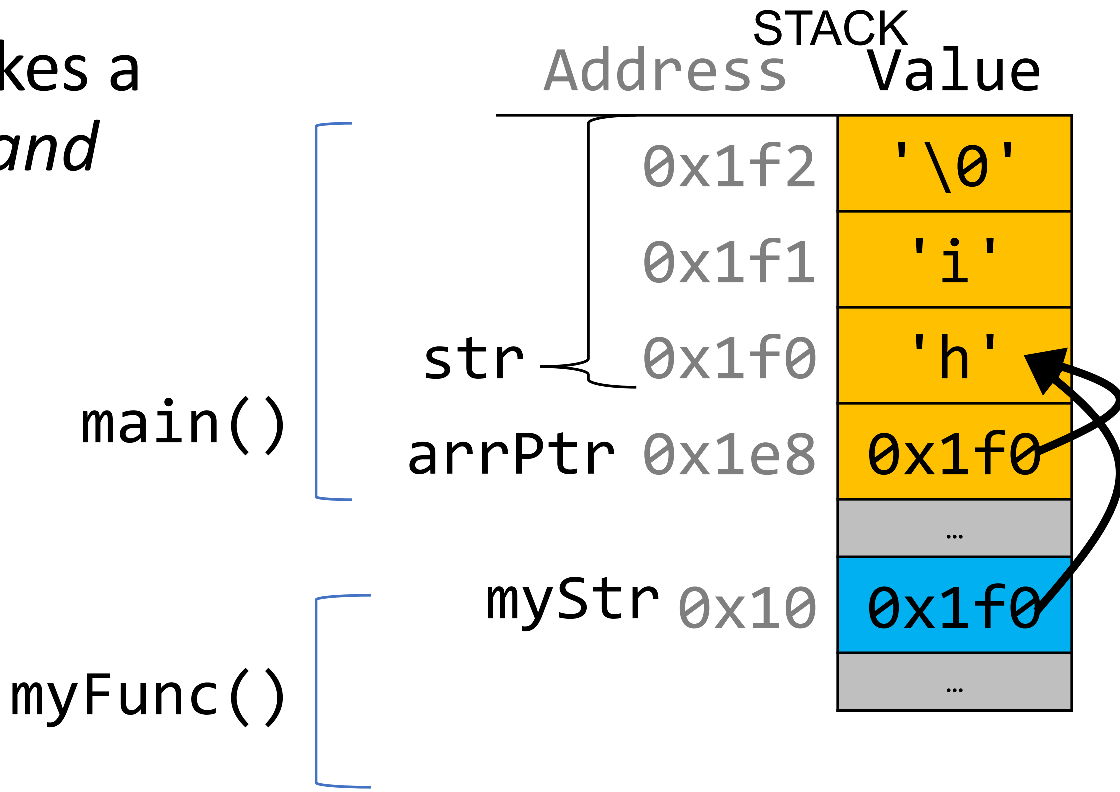
```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      myFunc(str);
      ...
}
```



Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element and* passes it (a pointer) to the function.

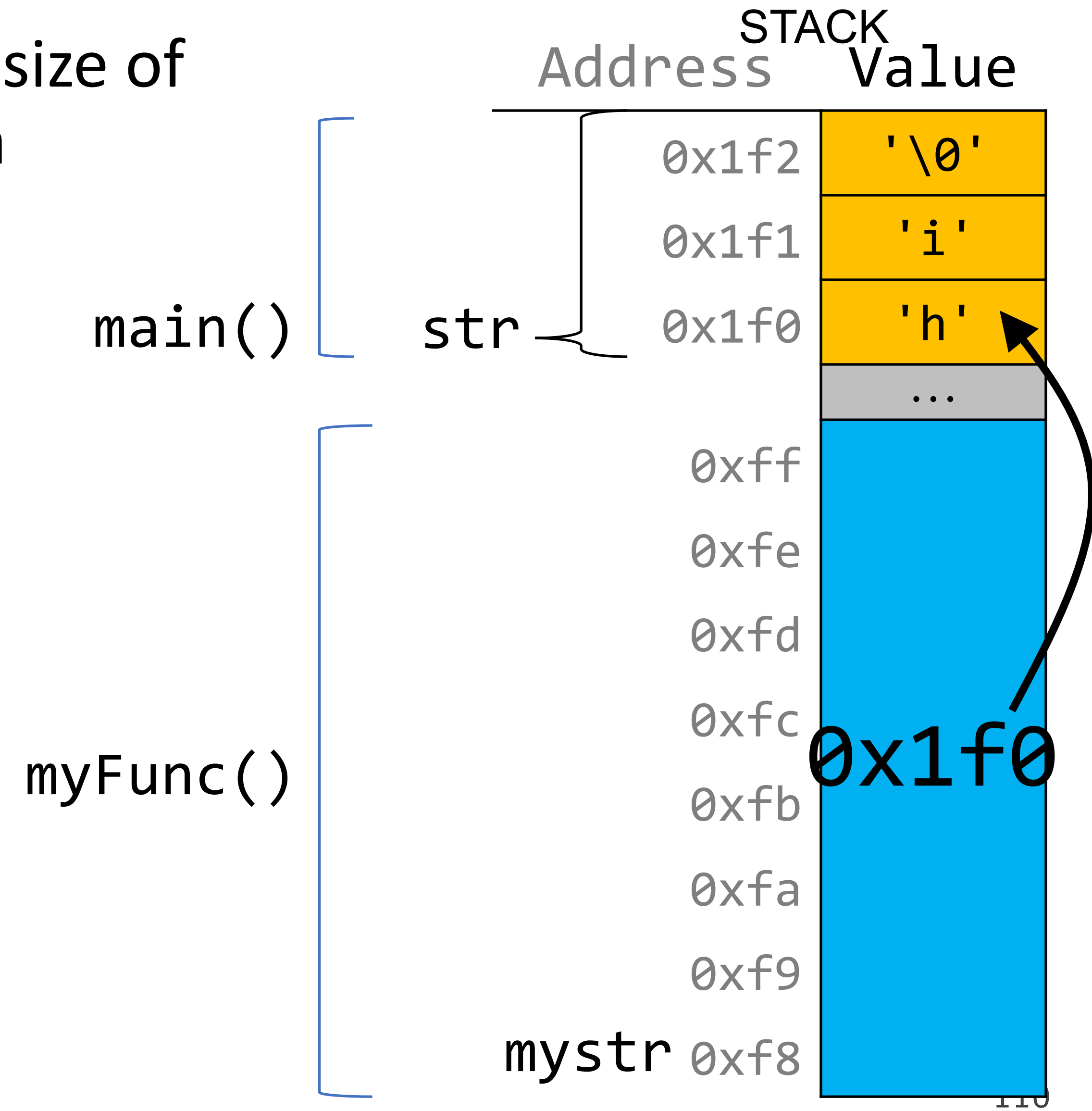
```
void myFunc(char *myStr) {
     …
}

int main(int argc, char *argv[]) {
     char str[3];
     strcpy(str, "hi");
     // equivalent
     char *arrPtr = str;
     myFunc(arrPtr);
     ...
```



Arrays as Parameters

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

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

int main(int argc, char *argv[]) {
    char str[3];
    strcpy(str, "hi");
    int size = sizeof(str);   // 3
    myFunc(str);
    ...
}
```

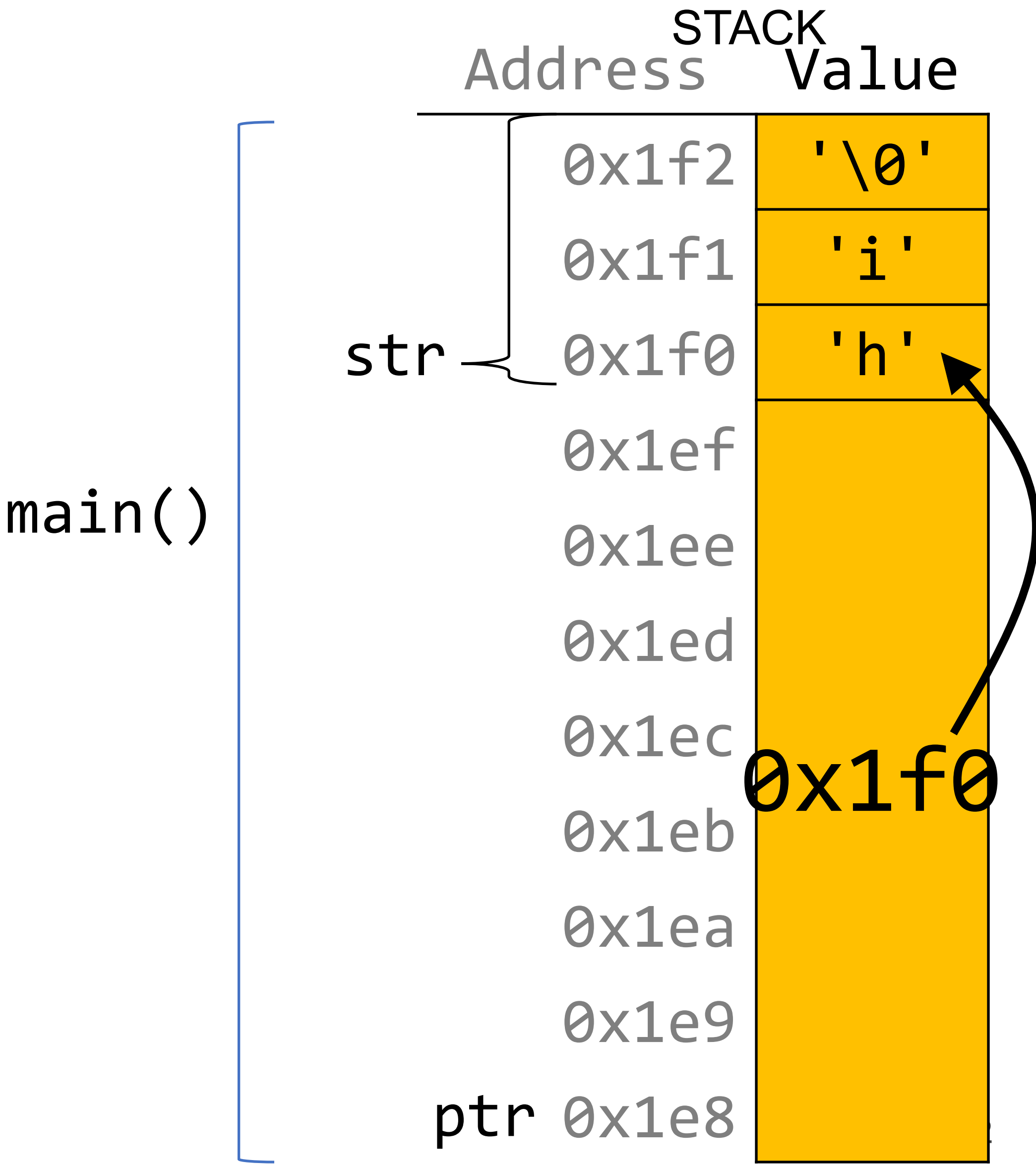


`sizeof` returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **`sizeof`** to get its full size.

Arrays and Pointers

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

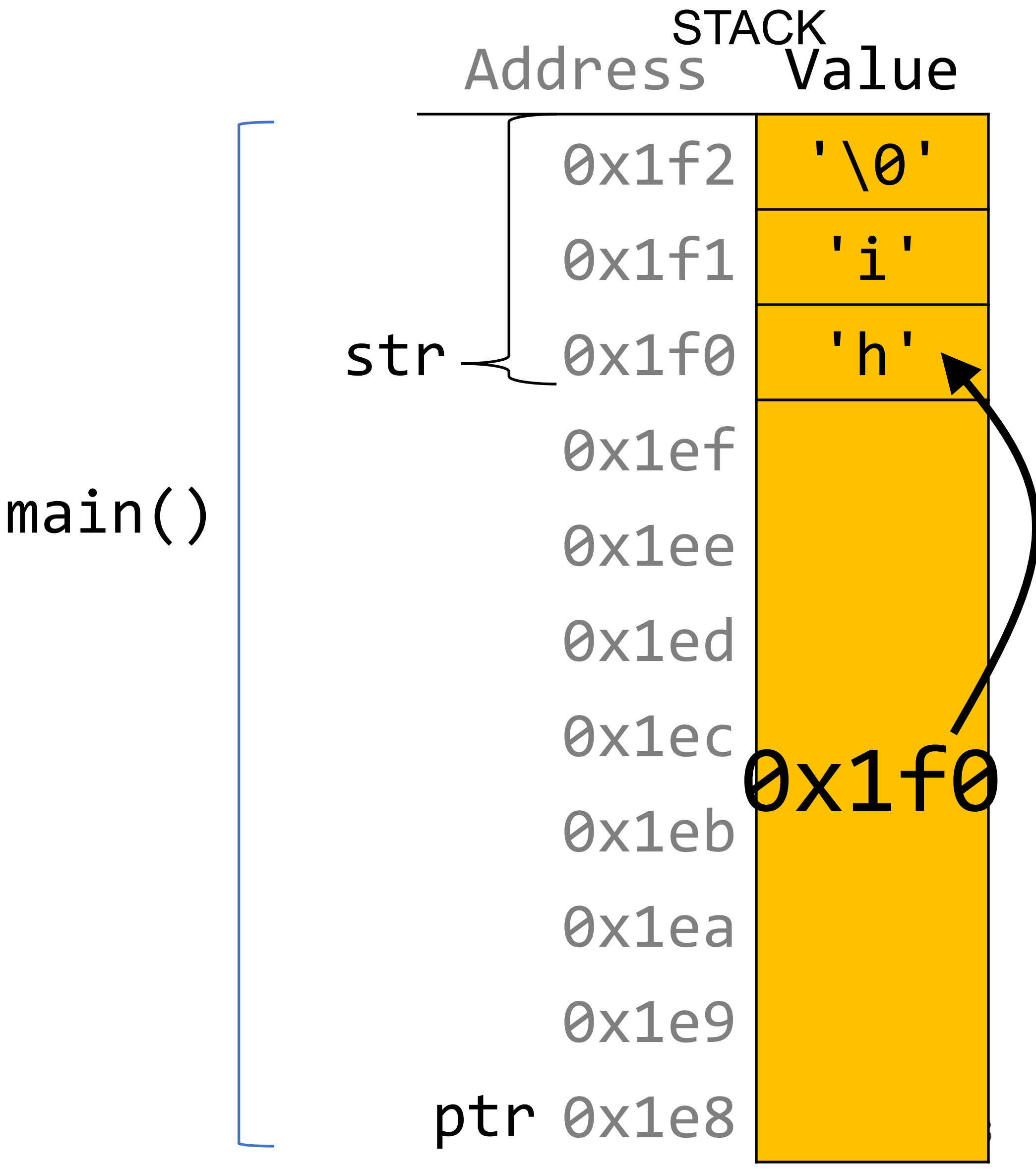
```
int main(int argc, char *argv[]) {
    char str[3];
    strcpy(str, "hi");
    char *ptr = str;
    ...
}
```



Arrays and Pointers

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

```
int main(int argc, char *argv[]) {
    char str[3];
    strcpy(str, "hi");
    char *ptr = str;

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

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



Lecture Plan

- Pointers and Parameters
- Strings in Memory
- Double Pointers
- Arrays in Memory
- **Arrays of Pointers**
- Pointer Arithmetic
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/lecture-code/lect06 .
```

Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:

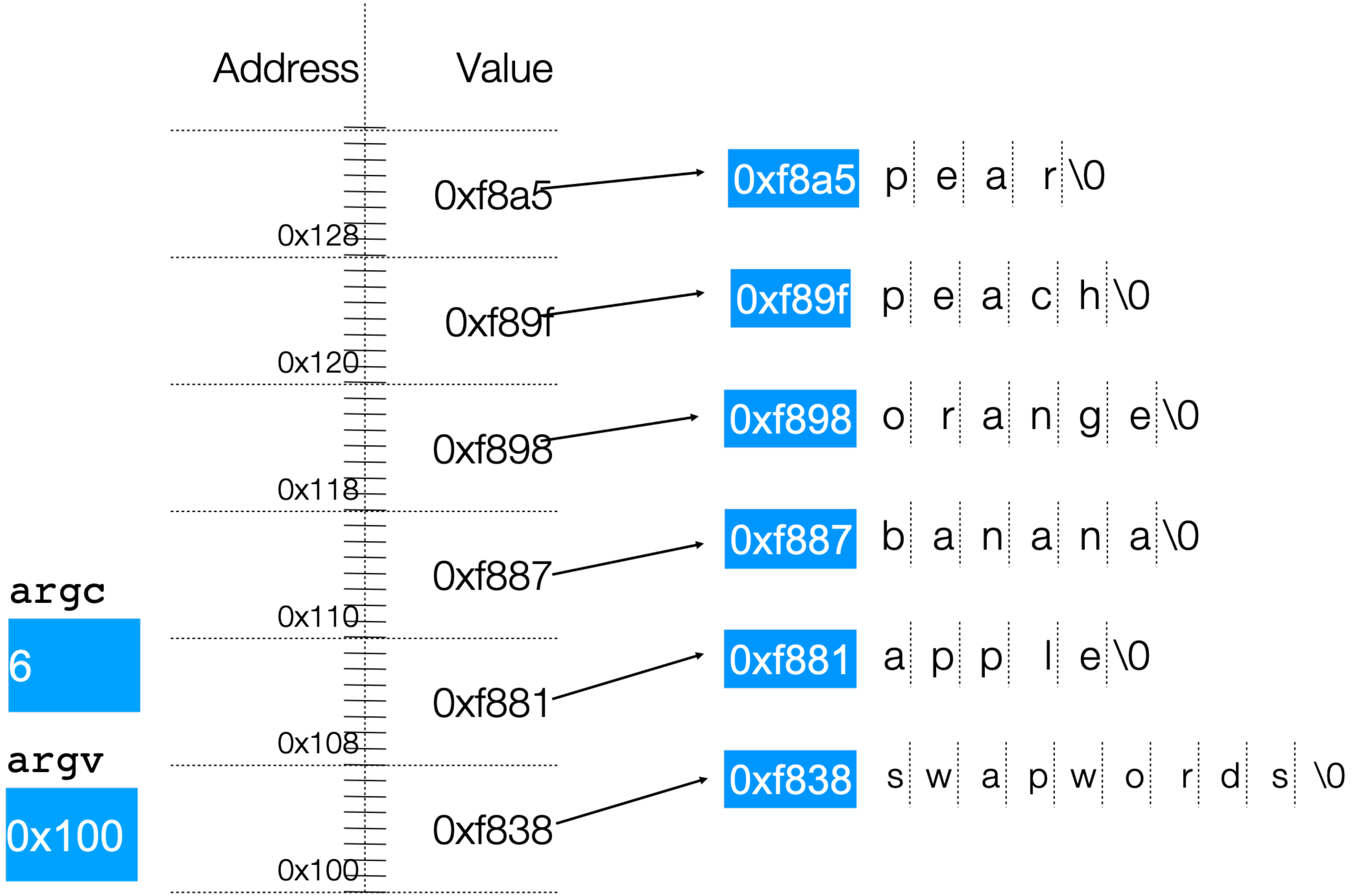
```
char *stringArray[5];   // space to store 5 char *s
```

This stores 5 **char *s**, *not* all of the characters for 5 strings!

```
char *str0 = stringArray[0];     // first char *
```

Arrays Of Pointers

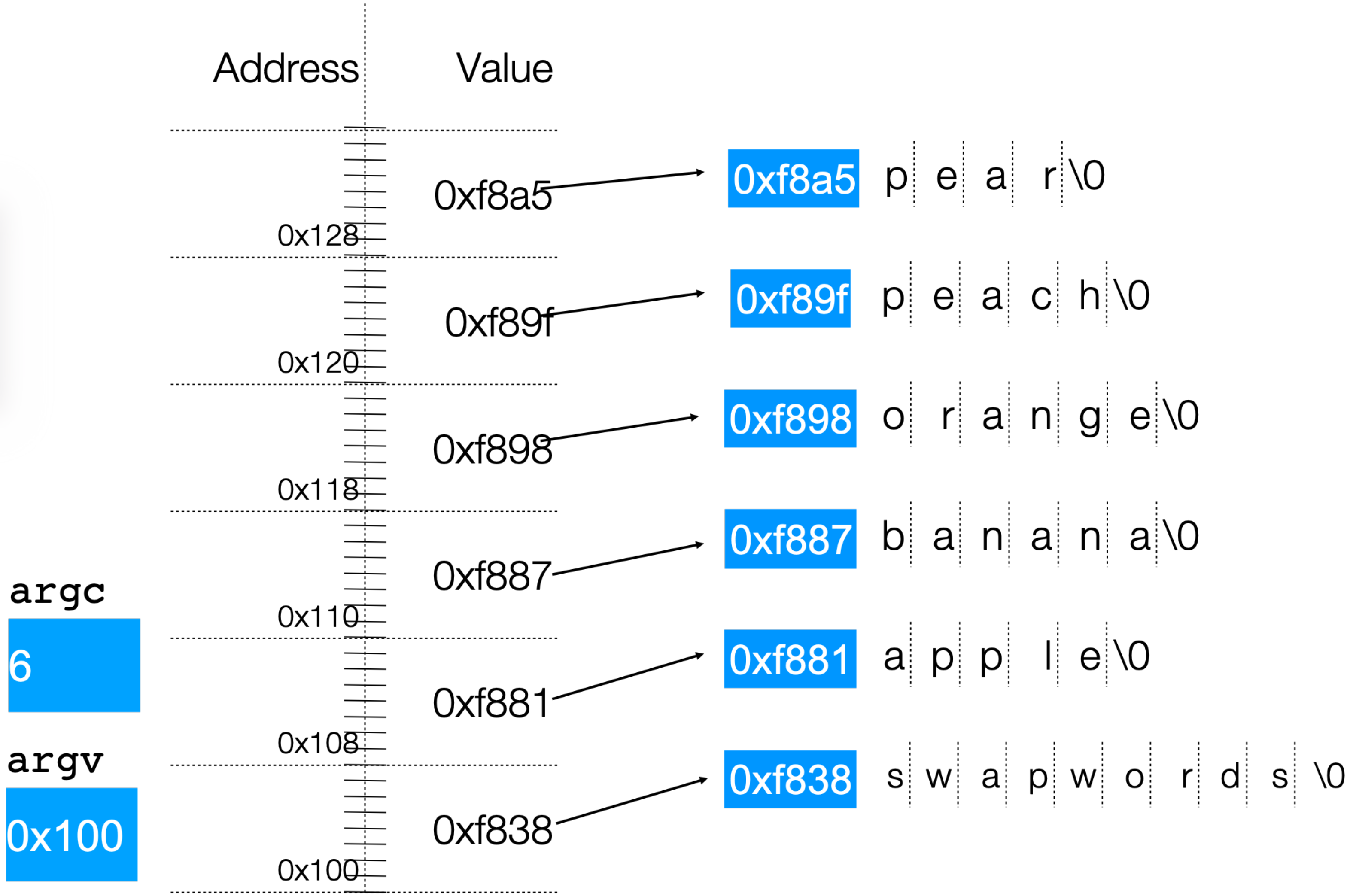
```
./swapwords apple banana orange peach pear
```



Arrays Of Pointers

```
./swapwords apple banana orange peach pear
```

What is the value of argv[2] in this diagram?



Lecture Plan

- Pointers and Parameters
- Strings in Memory
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- **Pointer Arithmetic**
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/lecture-code/lect06 .
```

Pointer Arithmetic

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

```c
char *str = "apple";    // e.g. 0xff0
char *str1 = str + 1;   // e.g. 0xff1
char *str3 = str + 3;   // e.g. 0xff3

printf("%s", str);      // apple
printf("%s", str1);     // pple
printf("%s", str3);     // le
```

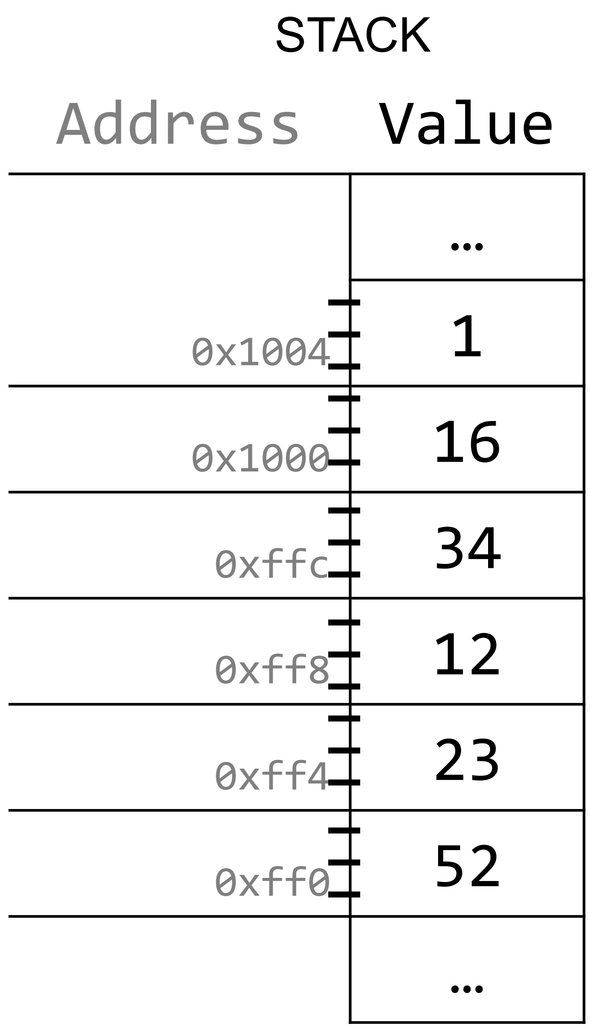
DATA SEGMENT

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

Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

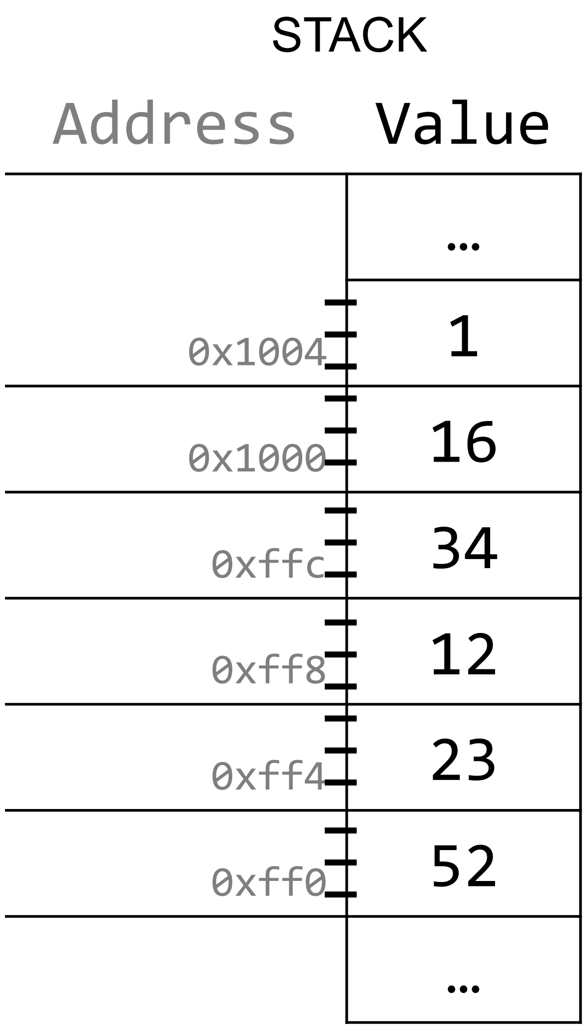
```
// nums points to an int array
int *nums = …               // e.g. 0xff0
int *nums1 = nums + 1;  // e.g. 0xff4
int *nums3 = nums + 3;  // e.g. 0xffc

printf("%d", *nums);    // 52
printf("%d", *nums1);   // 23
printf("%d", *nums3);   // 34
```



Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
// nums points to an int array
int *nums = …              // e.g. 0xff0
int *nums3 = nums + 3;     // e.g. 0xffc
int *nums2 = nums3 - 1;    // e.g. 0xff8

printf("%d", *nums);       // 52
printf("%d", *nums2);      // 12
printf("%d", *nums3);      // 34
```



Pointer Arithmetic

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

```
char *str = "apple";    // e.g. 0xff0

// both of these add two places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff2.
char thirdLetter = str[2];         // 'p'
char thirdLetter = *(str + 2);     // 'p'
```

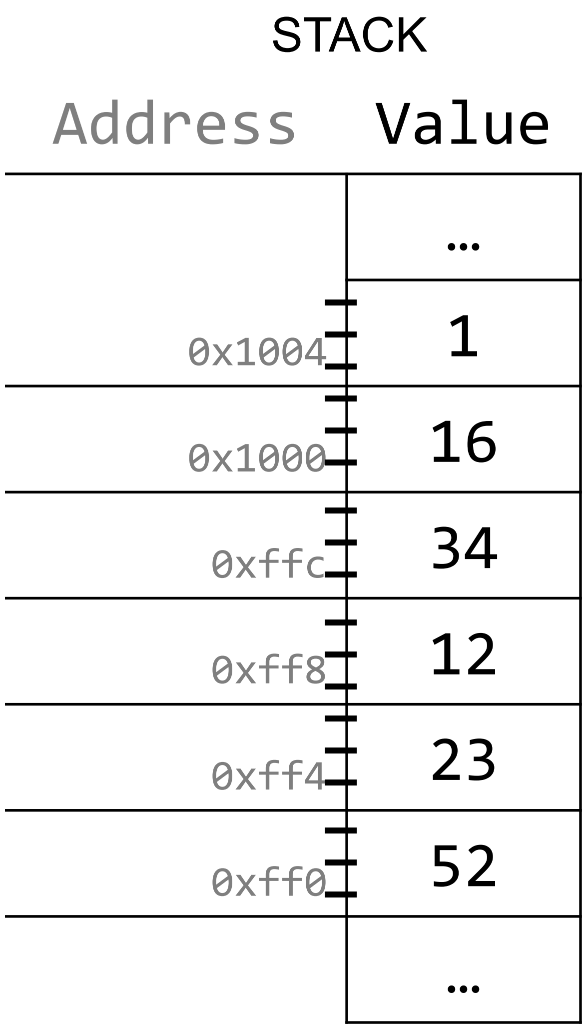
DATA SEGMENT

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

Pointer Arithmetic

Pointer arithmetic with two pointers does *not* give the byte difference. Instead, it gives the number of *places* they differ by.

```
// nums points to an int array
int *nums = …                 // e.g. 0xff0
int *nums3 = nums + 3;        // e.g. 0xffc
int diff = nums3 - nums;      // 3
```



Pointer Arithmetic

How does the code know how many bytes it should look at once it visits an address?

```
int x = 2;
int *xPtr = &x;            // e.g. 0xff0

// How does it know to print out just the 4 bytes at xPtr?
printf("%d", *xPtr);    // 2
```

Pointer Arithmetic

How does the code know how many bytes it should add when performing pointer arithmetic?

```
int nums[] = {1, 2, 3};

// How does it know to add 4 bytes here?
int *intPtr = nums + 1;

char str[6];
strcpy(str, "CS107");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Pointer Arithmetic

- At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.
- For this reason, when the program runs, it knows the correct number of bytes to address or add/subtract for each data type.

Lecture Plan

- Pointers and Parameters
- Strings in Memory
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- **Other topics: const, struct and ternary**

```
cp -r /afs/ir/class/cs107/lecture-code/lect06 .
```

Const

- Use **const** to declare global constants in your program. This indicates the variable cannot change after being created.

```
const double PI = 3.1415;
const int DAYS_IN_WEEK = 7;

int main(int argc, char *argv[]) {
    …
    if (x == DAYS_IN_WEEK) {
        …
    }
    …
}
```

Const

- Use **const** with pointers to indicate that the data that is pointed to cannot change.

```
char str[6];
strcpy(str, "Hello");
const char *s = str;

// Cannot use s to change characters it points to
s[0] = 'h';
```

Const

Sometimes we use **const** with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
	int count = 0;
	for (int i = 0; i < strlen(str); i++) {
		if (isupper(str[i])) {
			count++;
		}
	}
	return count;
}
```

Const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify.

```
// This function promises to not change str’s characters
int countUppercase(const char *str) {
     // compiler warning and error
    char *strToModify = str;
    strToModify[0] = …
}
```

Const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify. **Think of const as part of the variable type**.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
    const char *strToModify = str;
    strToModify[0] = …
}
```

Const

const can be confusing to interpret in some variable types.

```
// cannot modify this char
const char c = 'h';

// cannot modify chars pointed to by str
const char *str = …

// cannot modify chars pointed to by *strPtr
const char **strPtr = …
```

const vs #define

```
#define THIRD_BIT 1 << 3

// cannot modify this char
const char c = 'h';

// cannot modify chars
pointed to by str
const char *str = …

// cannot modify chars
pointed to by *strPtr
const char **strPtr = …
```

(Slide 86)

#define is a hard-coded substitution that gcc will make when compiling your code.

Const signals that this variable (in this scope) should not be modified.

- In CS107, you often won't have to declare const variables, but you will be provided parameters or use functions that have it
- Const directly modifies the adjacent keyword

Structs

A *struct* is a way to define a new variable type that is a group of other variables.

```
struct date {             // declaring a struct type
    int month;
    int day;              // members of each date structure
};
…

struct date today;                            // construct structure instances
today.month = 1;
today.day = 28;

struct date new_years_eve = {12, 31};   // shorter initializer syntax
```

Structs

Wrap the struct definition in a **typedef** to avoid having to include the word **struct** every time you make a new variable of that type.

```
typedef struct date {
    int month;
    int day;
} date;
…

date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct.

```
void advance_day(date d) {
      d.day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(my_date);
      printf("%d", my_date.day); // 28
      return 0;
}
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct. **Use a pointer to modify a specific instance.**

```
void advance_day(date *d) {
     (*d).day++;
}

int main(int argc, char *argv[]) {
     date my_date = {1, 28};
     advance_day(&my_date);
     printf("%d", my_date.day); // 29
     return 0;
}
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
void advance_day(date *d) {
      d->day++;          // equivalent to (*d).day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(&my_date);
      printf("%d", my_date.day); // 29
      return 0;
}
```

Structs

C allows you to return structs from functions as well. It returns whatever is contained within the struct.

```
date create_new_years_date() {
     date d = {1, 1};
     return d;          // or return (date){1, 1};
}

int main(int argc, char *argv[]) {
     date my_date = create_new_years_date();
     printf("%d", my_date.day); // 1
     return 0;
}
```

Structs

sizeof gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```
typedef struct date {
      int month;
      int day;
  } date;

int main(int argc, char *argv[]) {
      int size = sizeof(date);    // 8
      return 0;
}
```

Arrays of Structs

You can create arrays of structs just like any other variable type.

```
typedef struct my_struct {
    int x;
    char c;
} my_struct;

…

my_struct array_of_structs[5];
```

Arrays of Structs

To initialize an entry of the array, you must use this special syntax to confirm the type to C.

```
typedef struct my_struct {
    int x;
    char c;
} my_struct;

…

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```

Arrays of Structs

You can also set each field individually.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…
my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
```

Ternary Operator

The ternary operator is a shorthand for using if/else to evaluate to a value.

```
condition ? expressionIfTrue : expressionIfFalse
```

```
int x;
if (argc > 1) {
    x = 50;
} else {
    x = 0;
}

// equivalent to
int x = argc > 1 ? 50 : 0;
```

Recap

- Pointers and Parameters
- Strings in Memory
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- Other topics: **const**, **struct** and ternary

Next time: dynamically allocated memory

Lecture 6 takeaway: pointers let us store the addresses of data and pass them as parameters. We can perform arithmetic with pointers to change where they point to. Arrays in C also "decay to pointers" as parameters and in arithmetic expressions.

Extra Practice

1. Pointer arithmetic

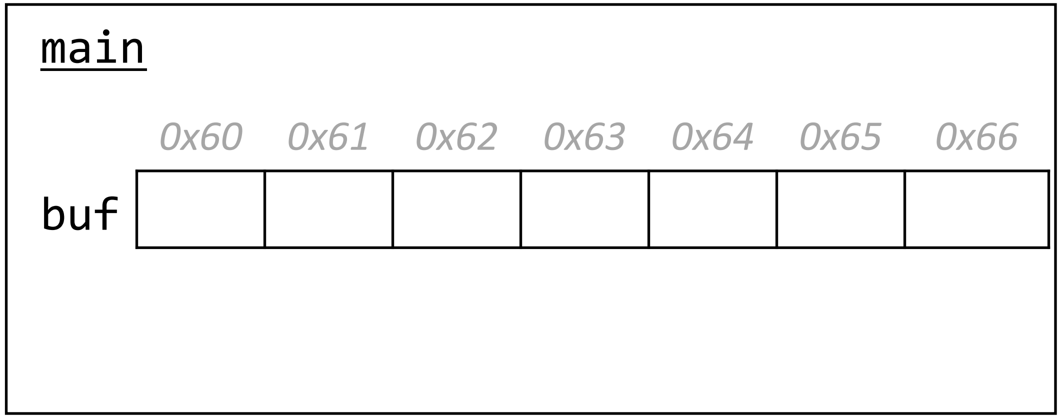
```c
void func(char *str) {
    str[0] = 'S';
    str++;
    *str = 'u';
    str = str + 3;
    str[-2] = 'm';
}

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

- 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. Pointer arithmetic

```c
void func(char *str) {
    str[0] = 'S';
    str++;
    *str = 'u';
    str = str + 3;
    str[-2] = 'm';
}

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



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

2. char* vs char[ ] exercises

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;
```

2. char* vs char[] exercises

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)

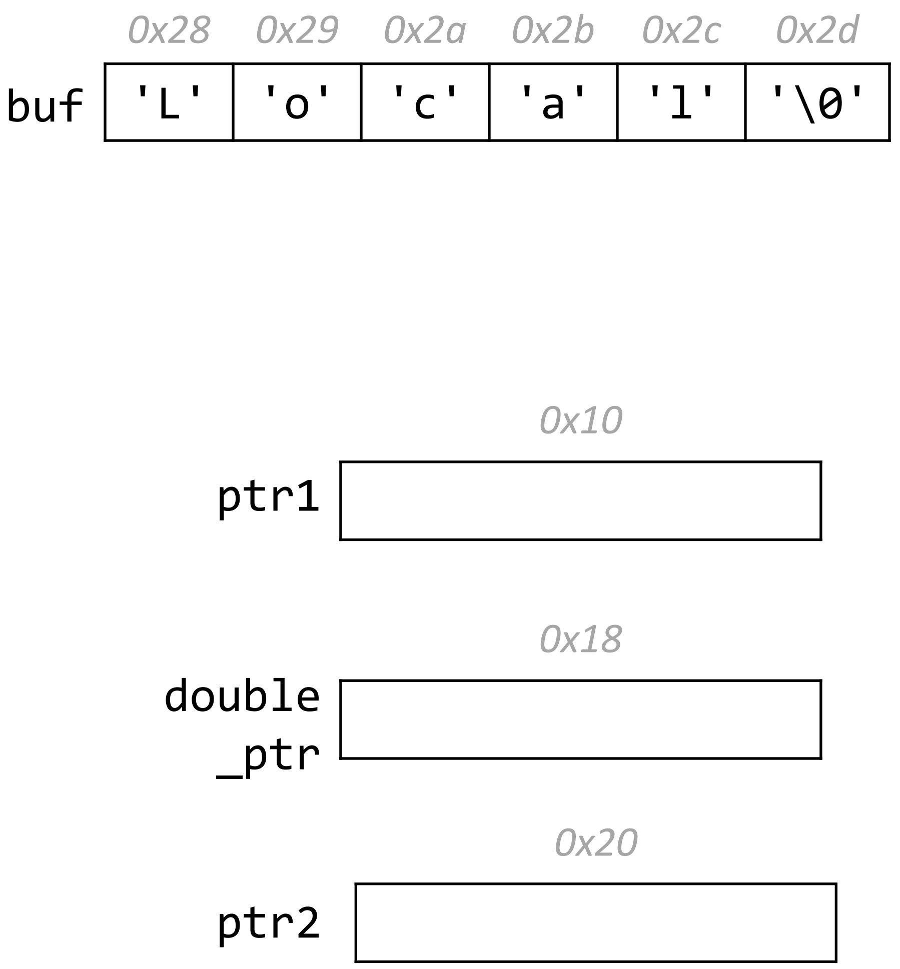
3. Bonus: Tricky addresses

```c
void tricky_addresses() {
  char buf[] = "Local";
  char *ptr1 = buf;
  char **double_ptr = &ptr1;
  printf("ptr1's value:     %p\n", ptr1);
  printf("ptr1's deref    : %c\n", *ptr1);
  printf("       address:   %p\n", &ptr1);
  printf("double_ptr value: %p\n", double_ptr);
  printf("buf's address:    %p\n", &buf);

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

What is stored in each variable?

3. Bonus: Tricky addresses

```
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 }
```



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

Translating C into English

***** 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"

& "address of"

<ptr name> address

<arr name> address (except sizeof)

```
int arr[] = {3, 4, -1, 2};   // initializes stack array
                             // with 4 ints
1. int *ptr0  = arr;
2. int *elt0  = *arr;
3. int elt    = *(arr + 3);
4. int **ptr1 = &ptr;
```

Type check with a diagram!

Translating C into English

***** 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"

& "address of"

<ptr name> address

<arr name> address (except sizeof)

```
   int arr[] = {3, 4, -1, 2};    // initializes stack array
                                 // with 4 ints
1. int *ptr0  = arr;
2. int *elt0  = *arr;
3. int elt    = *(arr + 3);
4. int **ptr1 = &ptr;
```

1. **Address arr**
2. **Value at address arr**
3. **The value at address <3 ints after address arr>**
4. **address of ptr**

Type check with a diagram!

Translating C into English

* 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"

& "address of"

<ptr name> address

<arr name> address (except sizeof)

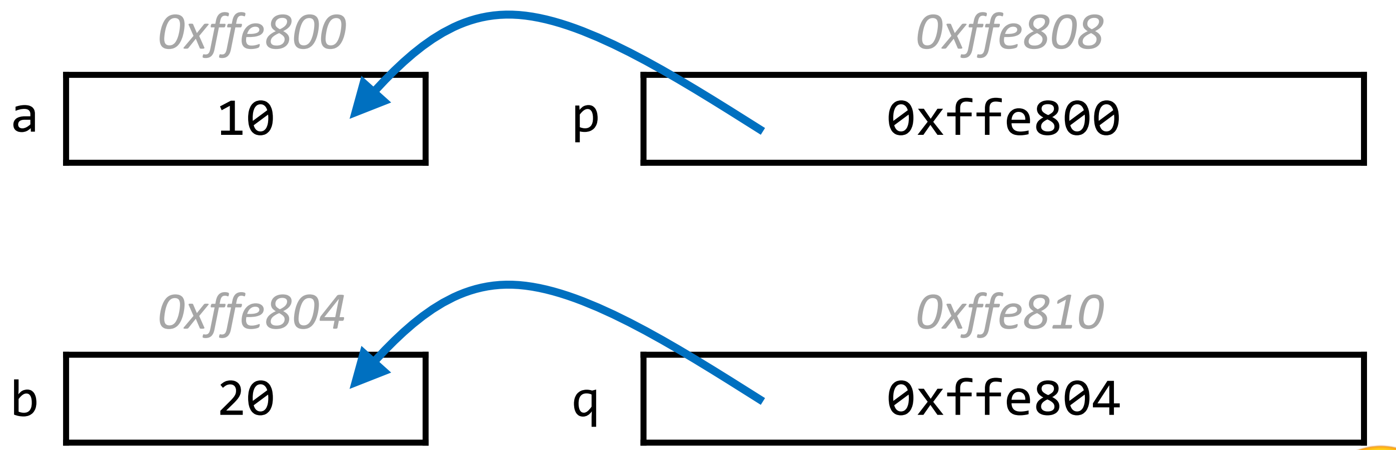
```
int arr[] = {3, 4, -1, 2};     // initializes stack array
                               // with 4 ints
1. int *ptr0  = arr;
2. int *elt0  = *arr;
3. int elt    = *(arr + 3);
4. int **ptr1 = &ptr;
```

Type check with a diagram!

Pen and paper: A * Wars Story

```
1 void binky() {
2       int a = 10;
3       int b = 20;
4       int *p = &a;
5       int *q = &b;
6
7       *p = *q;
8       p = q;
9 }
```

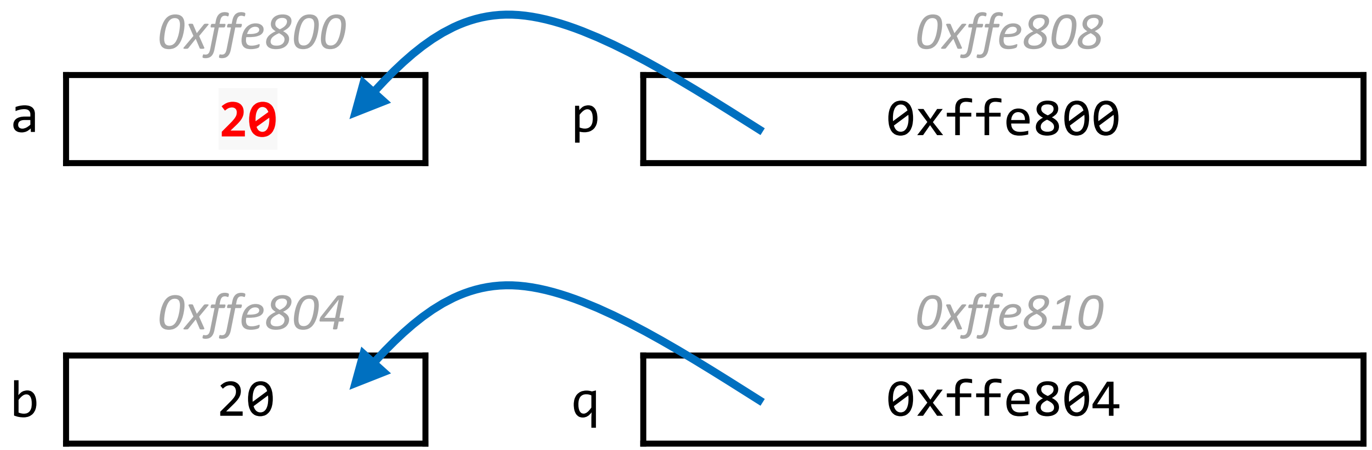
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



Pen and paper: A * Wars Story

```c
1 void binky() {
2       int a = 10;
3       int b = 20;
4       int *p = &a;
5       int *q = &b;
6
7       *p = *q;
8       p = q;
9 }
```

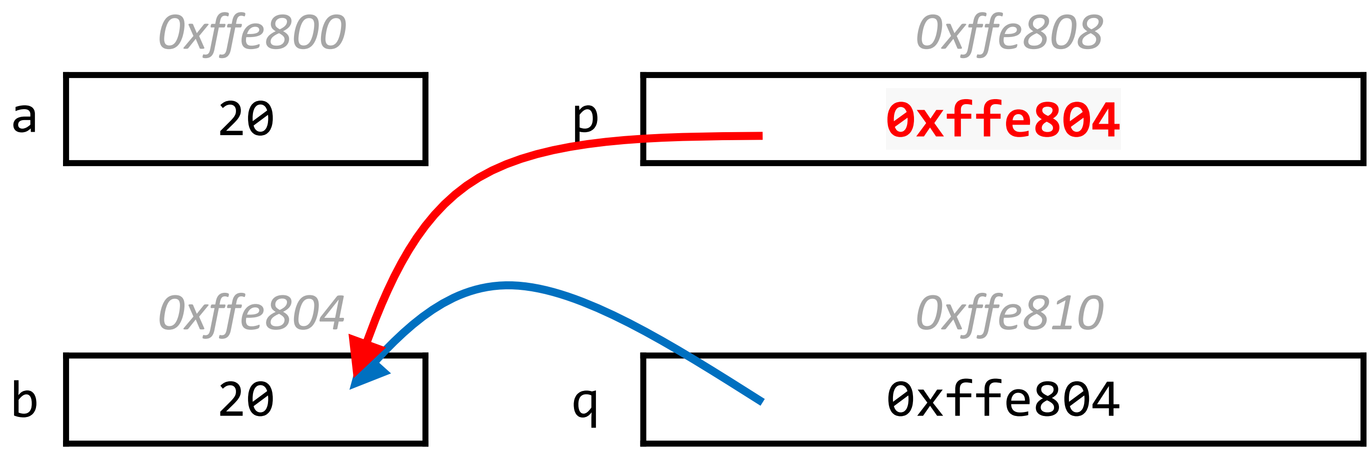
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



Pen and paper: A * Wars Story

```
1 void binky() {
2       int a = 10;
3       int b = 20;
4       int *p = &a;
5       int *q = &b;
6
7       *p = *q;
8       p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



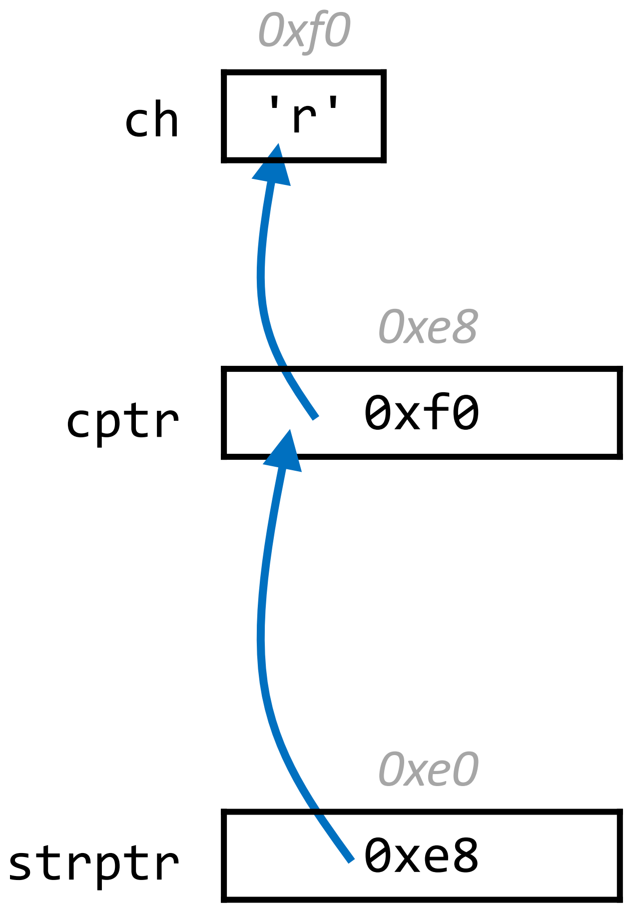
* Wars: Episode I (of 2)

In variable **declaration**, * creates a **pointer**.

`char ch = 'r';` — ch stores a char

`char *cptr = &ch;` — cptr stores an address of a char (**points to** a char)

`char **strptr = &cptr;` — strptr stores an address of a char * (**points to** a char *)



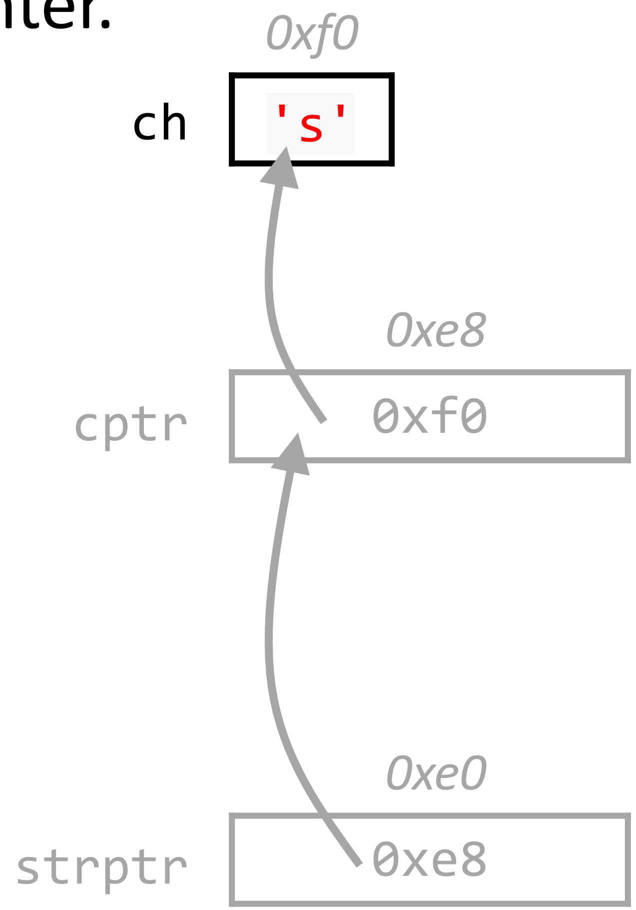
* Wars: Episode II (of 2)

In **<u>reading values from/storing values</u>**, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in ch

```
char *cptr = &ch;


char **strptr = &cptr;
```



* Wars: Episode II (of 2)

In **<u>reading values from/storing values</u>**, * **dereferences** a pointer.

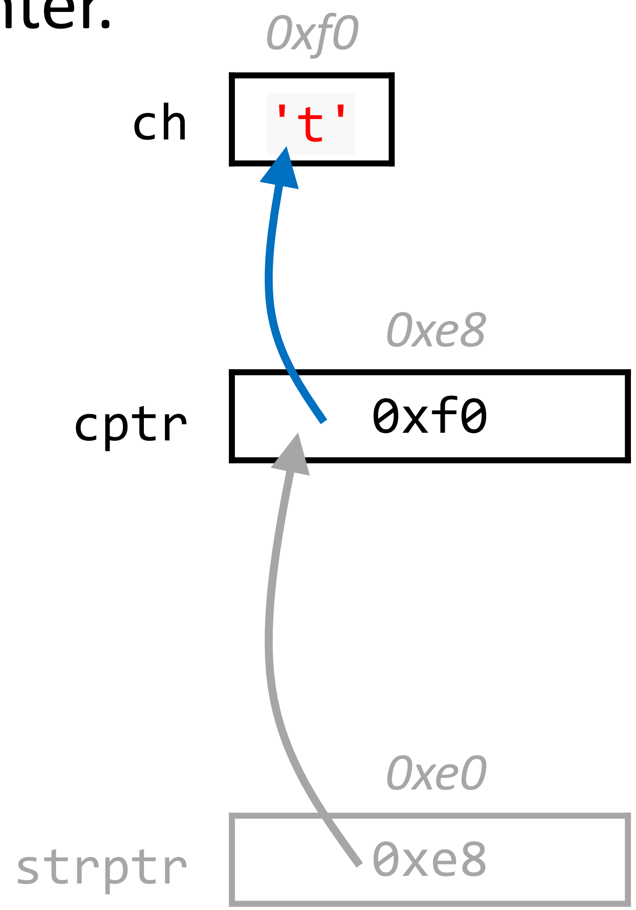
```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in ch

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in cptr (increment char **pointed to**)

```
char **strptr = &cptr;
```



* Wars: Episode II (of 2)

In **reading values from/storing values**, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

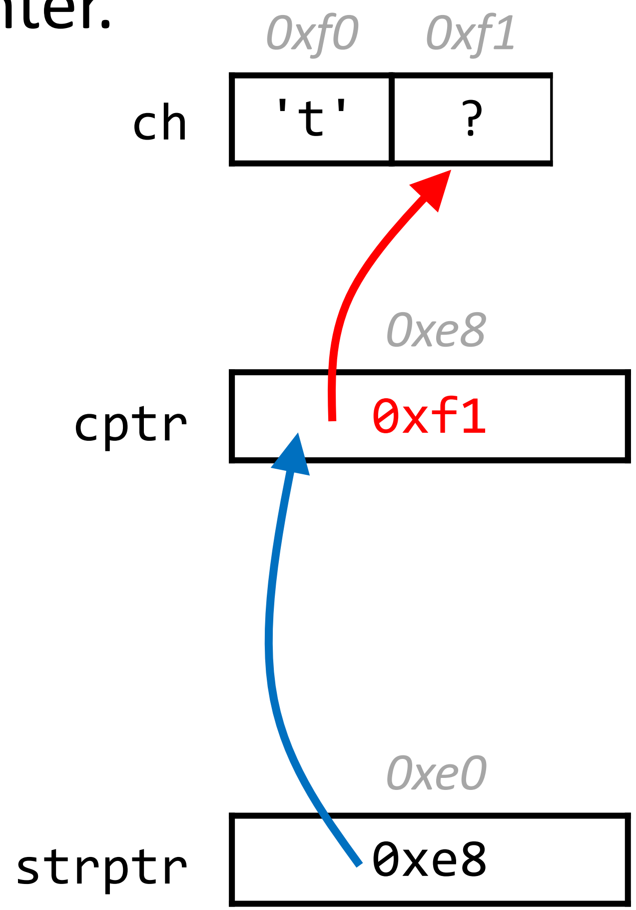
Increment value stored in ch

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in cptr (increment char **pointed to**)

```
char **strptr = &cptr;
*strptr = *strptr + 1;
```

Increment value stored at memory address in cptr (increment address **pointed to**)



Exercise: Implementation

The below function sums up the string lengths of the num strings in `strs`.

- Try both 1. <u>array [ ] syntax</u> and 2. <u>pointer arithmetic</u>!

```
size_t get_total_strlen(char *strs[], size_t num) {
    size_t total_length = 0;
    for (int i = 0; i < num; i++) {
        // fill this in
    }
    return total_length;
}
```

Exercise: Implementation

The below function sums up the string lengths of the num strings in `strs`.

- Try both 1. array [ ] syntax and 2. pointer arithmetic!

```c
size_t get_total_strlen(char *strs[], size_t num) {
    size_t total_length = 0;
    for (int i = 0; i < num; i++) {
        // TODO: fill this in two ways
    }
    return total_length;
}
```

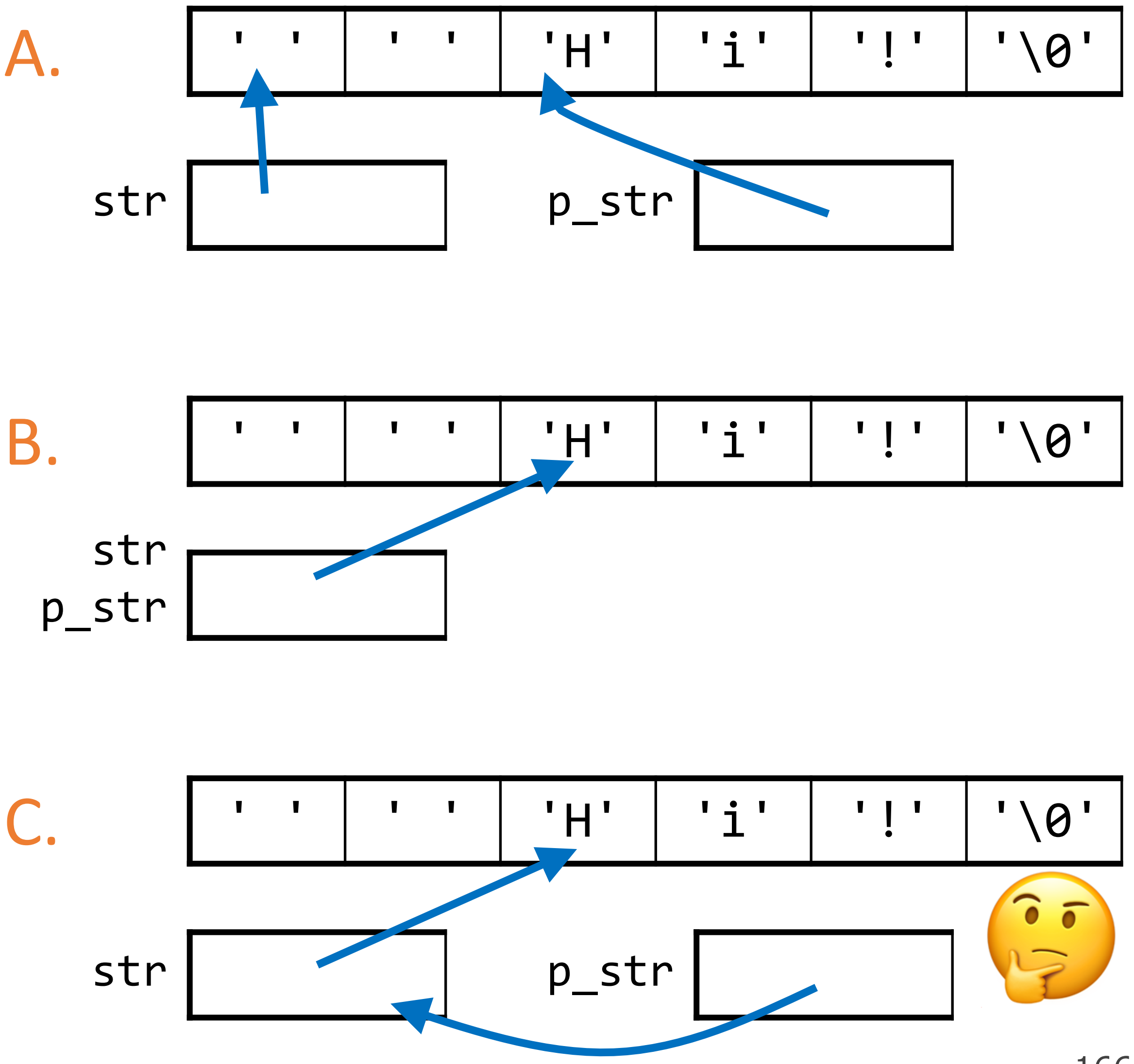
Equivalent:

1. `total_length += strlen(strs[i]);`
2. `total_length += strlen(*(strs + i));`

Skip spaces

```c
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

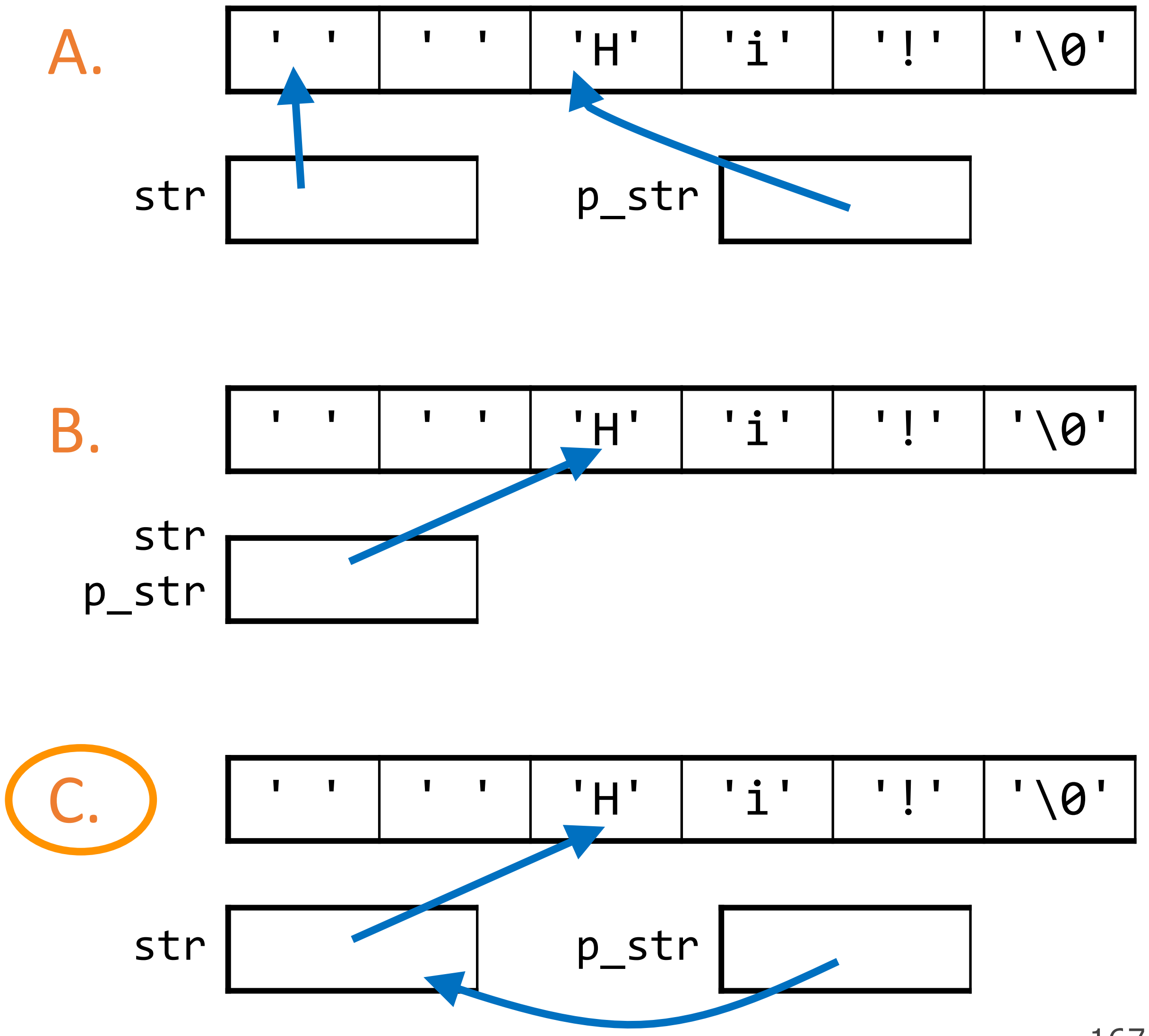
What diagram most accurately depicts program state at Line 4 (before skip_spaces returns to main)?



Skip spaces

```c
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

What diagram most accurately depicts program state at Line 4 (before skip_spaces returns to main)?



const

```
1  char buf[6];
2  strcpy(buf, "Hello");
3  const char *str = buf;
4  str[0] = 'M';
5  str = "Mello";
6  buf[0] = 'M';
```

Which lines (if any) above will cause an error due to violating const? Remember that const char * means that the characters at the location it stores cannot be changed.

const

```
1    char buf[6];
2    strcpy(buf, "Hello");
3    const char *str = buf;
4    str[0] = 'M';
5    str = "Mello";
6    buf[0] = 'M';
```

Line 1 makes a typical modifiable character array of 6 characters.

Which lines (if any) above will cause an error due to violating const? Remember that const char * means that the characters at the location it stores cannot be changed.

const

```
1 char buf[6];
2 strcpy(buf, "Hello");
3 const char *str = buf;
4 str[0] = 'M';
5 str = "Mello";
6 buf[0] = 'M';
```

Line 2 copies characters into this modifiable character array.

Which lines (if any) above will cause an error due to violating const? Remember that const char * means that the characters at the location it stores cannot be changed.

const

```
1    char buf[6];
2    strcpy(buf, "Hello");
3    const char *str = buf;
4    str[0] = 'M';
5    str = "Mello";
6    buf[0] = 'M';
```

Line 3 makes a const pointer that points to the first element of buf. We cannot use str to change the characters it points to because it is const.

Which lines (if any) above will cause an error due to violating const? Remember that const char * means that the characters at the location it stores cannot be changed.

const

```
1   char buf[6];
2   strcpy(buf, "Hello");
3   const char *str = buf;
4   str[0] = 'M';
5   str = "Mello";
6   buf[0] = 'M';
```

Line 4 is not allowed – it attempts to use a const pointer to characters to modify those characters.

Which lines (if any) above will cause an error due to violating const? Remember that const char * means that the characters at the location it stores cannot be changed.

const

```
✅ 1  char buf[6];
✅ 2  strcpy(buf, "Hello");
✅ 3  const char *str = buf;
❌ 4  str[0] = 'M';
✅ 5  str = "Mello";
   6  buf[0] = 'M';
```

Line 5 is ok – str's type means that while you cannot change the characters at which it points, you can change str itself to point somewhere else. **str** is not const – its characters are.

Which lines (if any) above will cause an error due to violating const? Remember that const char * means that the characters at the location it stores cannot be changed.

const

```
✅ 1  char buf[6];
✅ 2  strcpy(buf, "Hello");
✅ 3  const char *str = buf;
❌ 4  str[0] = 'M';
✅ 5  str = "Mello";
✅ 6  buf[0] = 'M';
```

Line 6 is ok – **buf** is a modifiable char array, and we can use it to change its characters. Declaring str as const doesn't mean that place in memory is not modifiable at all – it just means that you cannot modify it using **str**.

Which lines (if any) above will cause an error due to violating const? Remember that const char * means that the characters at the location it stores cannot be changed.