

CS107, Lecture 6

More Pointers and Arrays

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

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

Lecture Plan

- Pointers and Parameters 4
- Double Pointers 42
- Arrays in Memory 57
- Arrays of Pointers 66
- Pointer Arithmetic 70
- Other topics: **const**, **struct** and ternary 80
- Live Session 99

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

Lecture Plan

- **Pointers and Parameters** 4
- Double Pointers 42
- Arrays in Memory 57
- Arrays of Pointers 66
- Pointer Arithmetic 70
- Other topics: **const**, **struct** and ternary 80
- Live Session 99

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

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'
	...

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'
	...

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
```

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()



STACK	
Address	Value
x	0x1f0
	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()



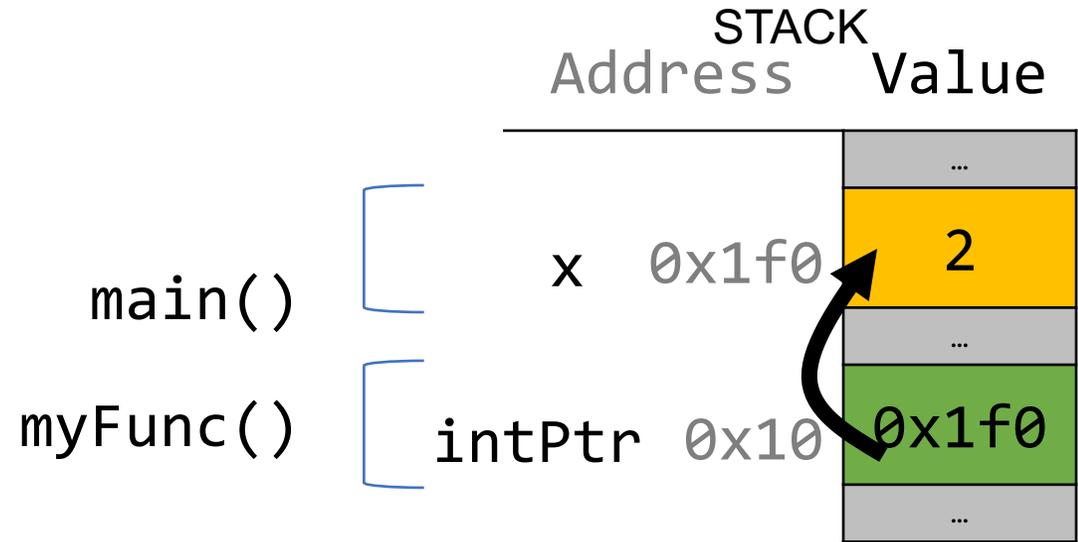
STACK		Value
Address		
		...
x	0x1f0	2
		...

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!  
    ...  
}
```

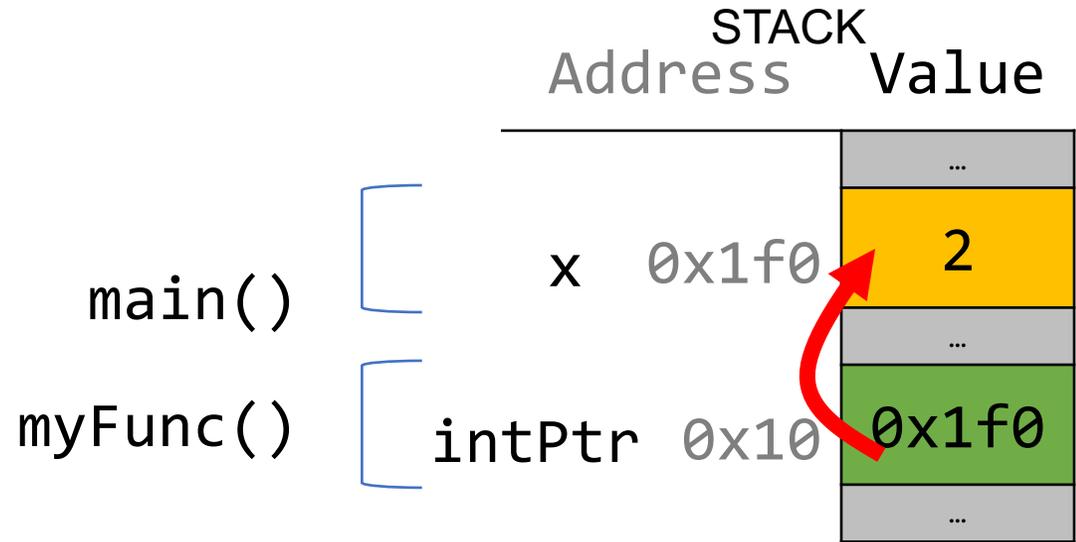


Pointers

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!  
    ...  
}
```

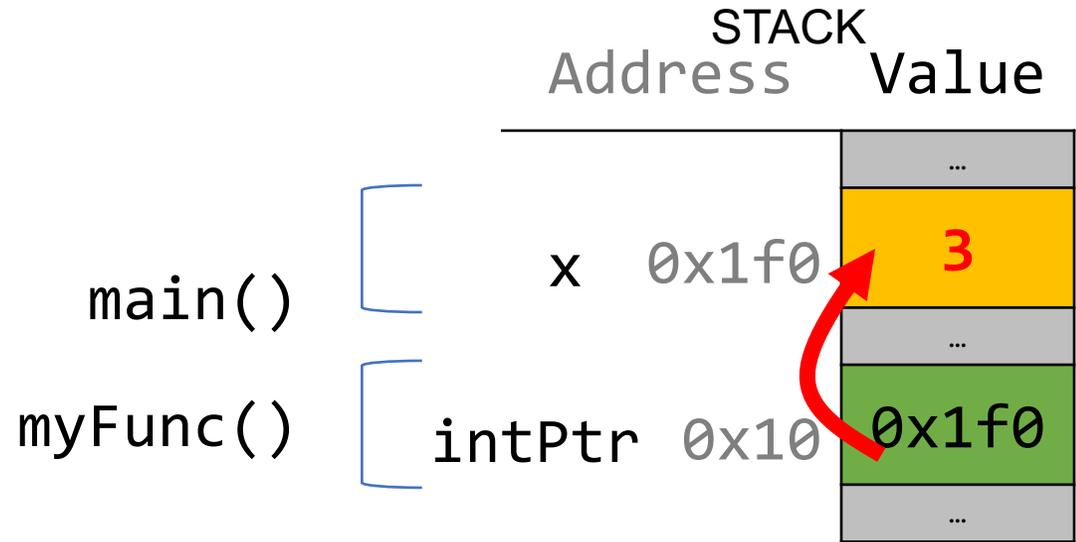


Pointers

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

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()



STACK	
Address	Value
	...
x 0x1f0	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!  
    ...  
}
```

main()



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

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 x + y;  
}  
  
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;  
}
```

char *

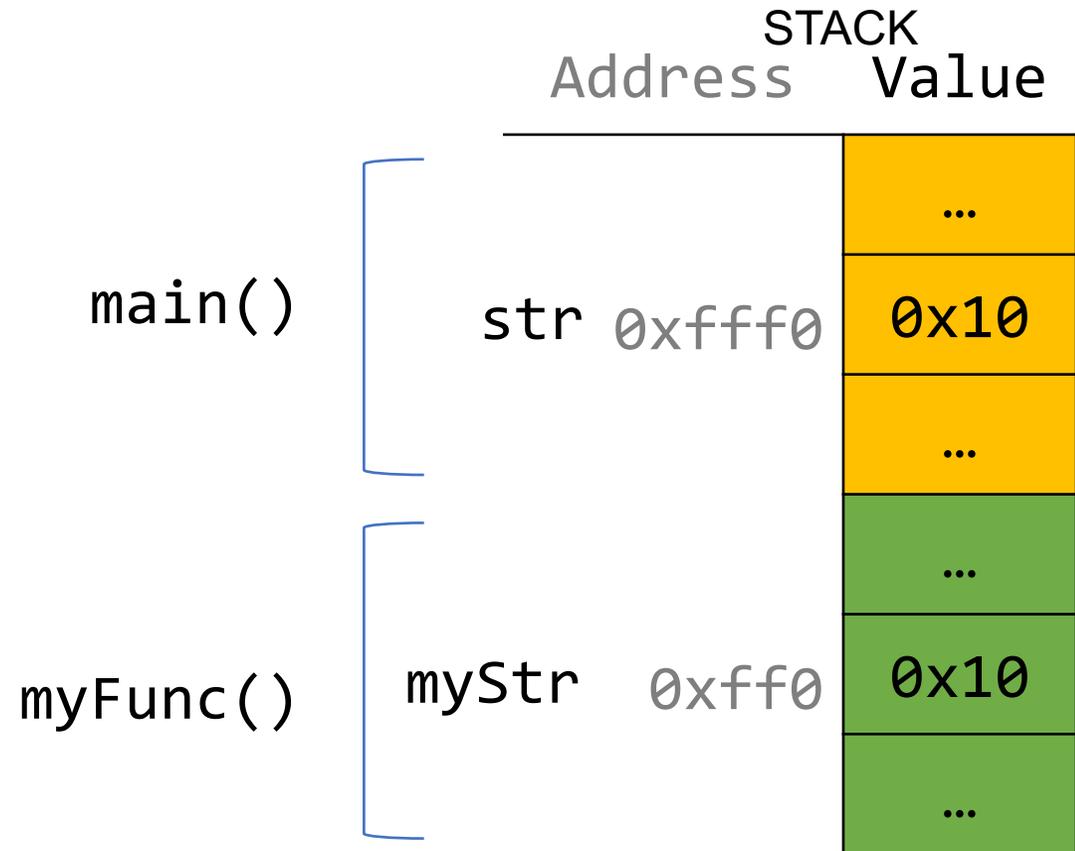
- A char * is technically a pointer to a **single character**.
- We commonly use char * as string by having the character it points to be followed by more characters and ultimately a null terminator.
- A char * could also just point to a single character (not a string).

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.

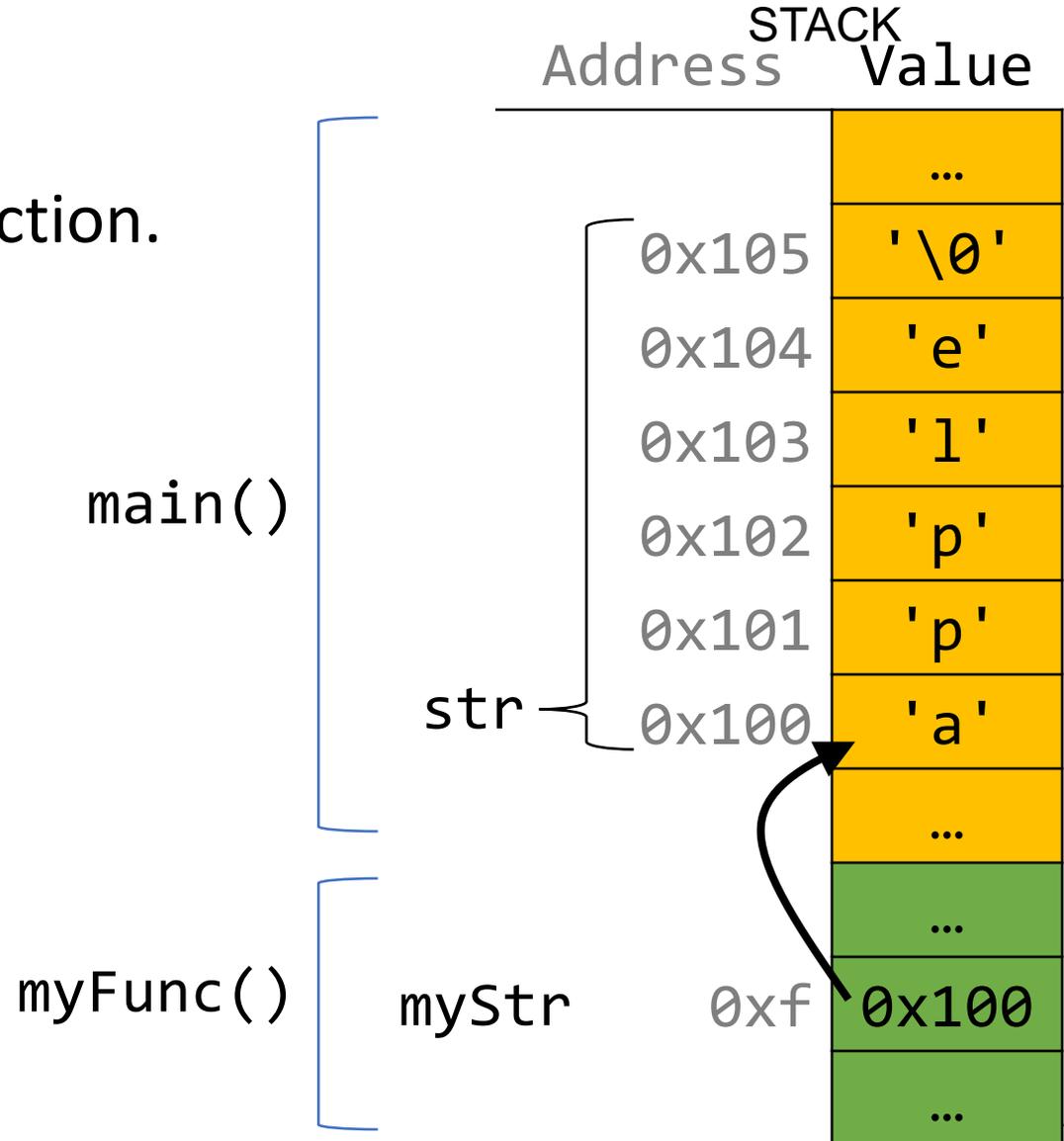
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char *str = "apple";  
    myFunc(str);  
    ...  
}
```



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.

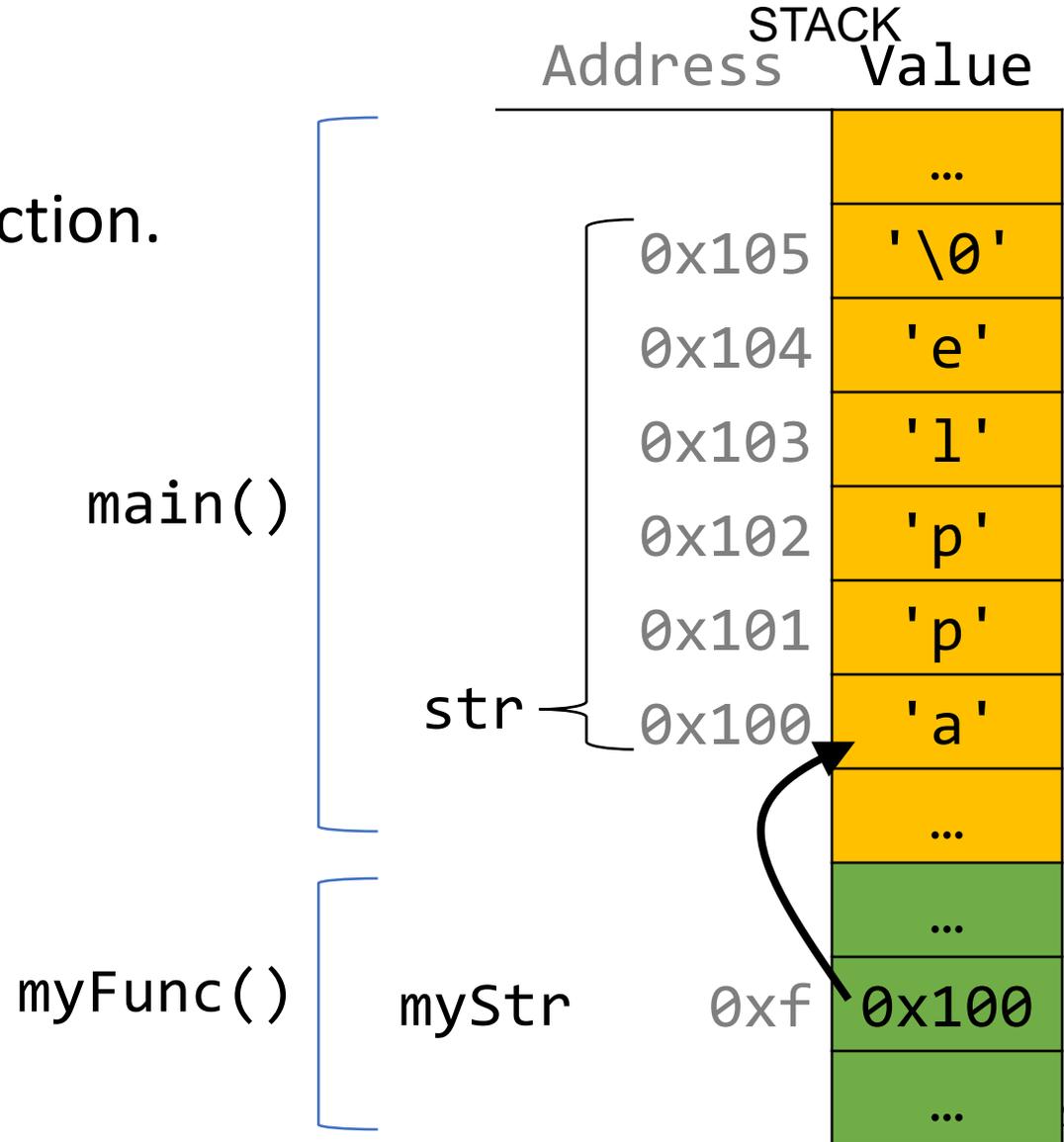
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    myFunc(str);  
    ...  
}
```



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.

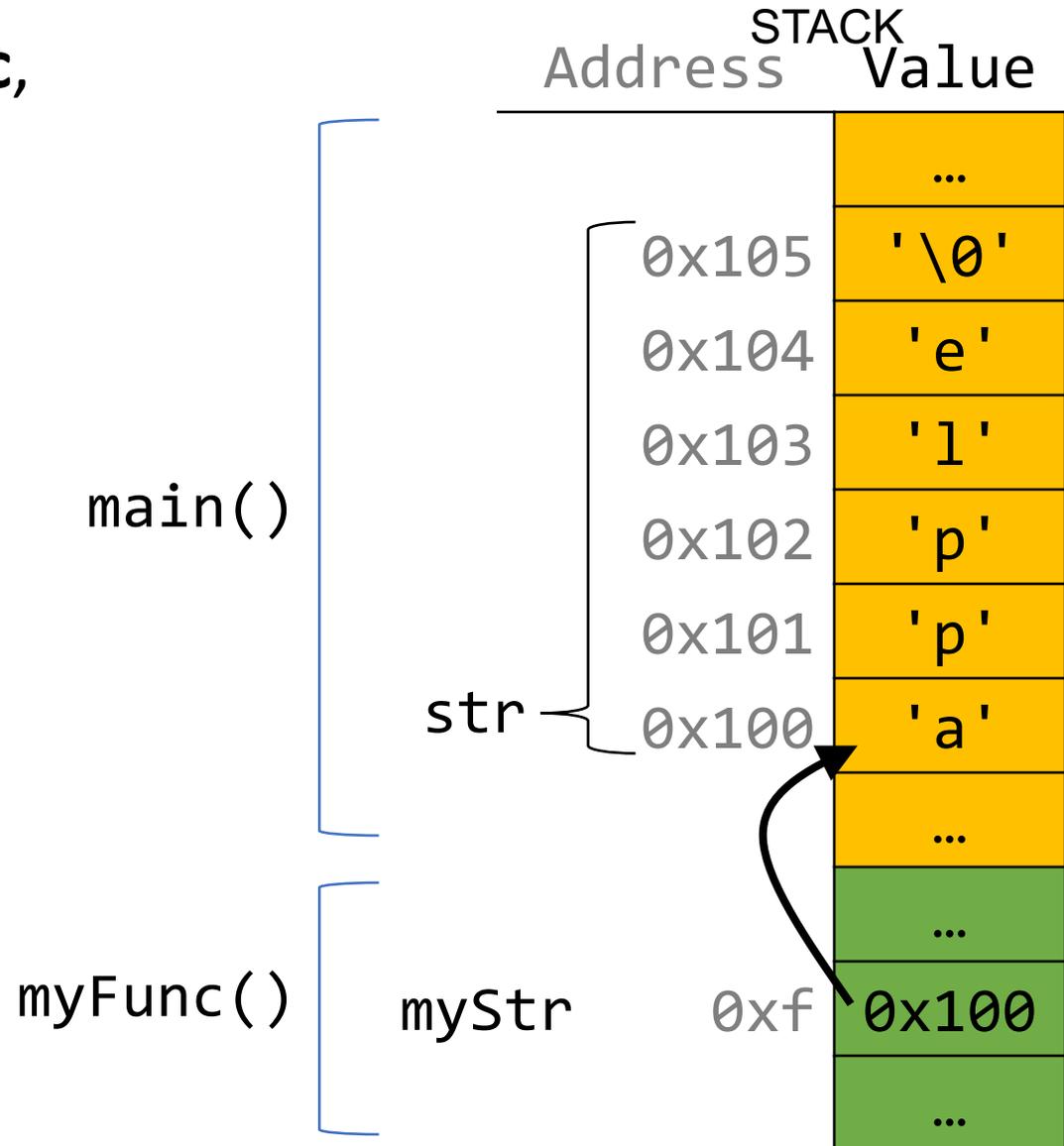
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    // equivalent  
    char *strAlt = str;  
    myFunc(strAlt);  
    ...  
}
```



Strings as Parameters

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

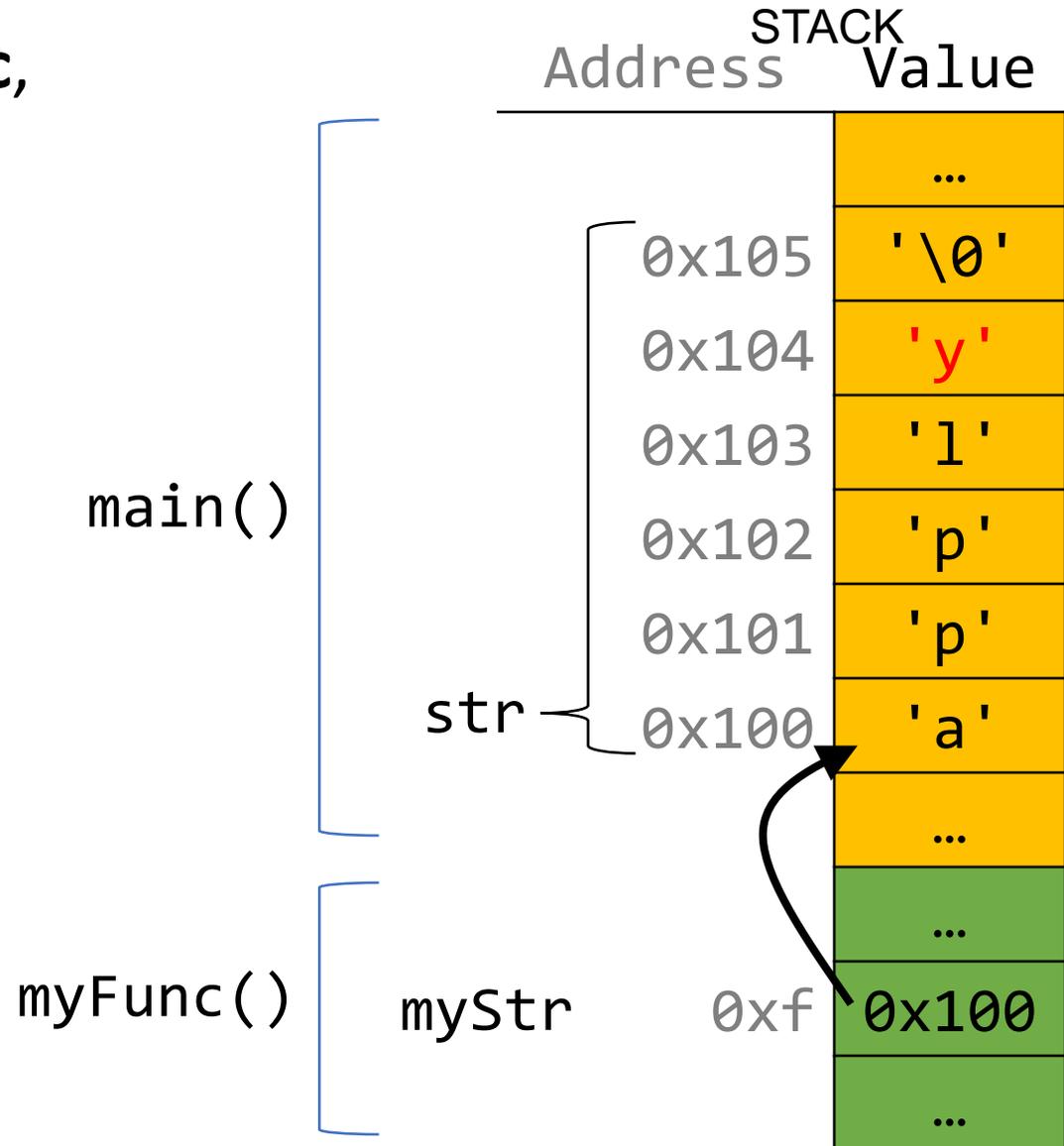
```
void myFunc(char *myStr) {  
    myStr[4] = 'y';  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    myFunc(str);  
    printf("%s", str); // apply  
    ...  
}
```



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  
    ...  
}
```



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);  
}
```

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.

```
int main(int argc, char *argv[]) {  
    int num = 3;  
    printSquare(num); // 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) {  
    x = x * x;  
    printf("%d", x);  
}
```

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.

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

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!  
}
```

Lecture Plan

- Pointers and Parameters 4
- **Double Pointers** 42
- Arrays in Memory 57
- Arrays of Pointers 66
- Pointer Arithmetic 70
- Other topics: **const**, **struct** and ternary 80
- Live Session 99

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

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) {  
    ...  
}
```

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

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

Demo: Skip Spaces



skip_spaces.c

Pointers to Strings

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

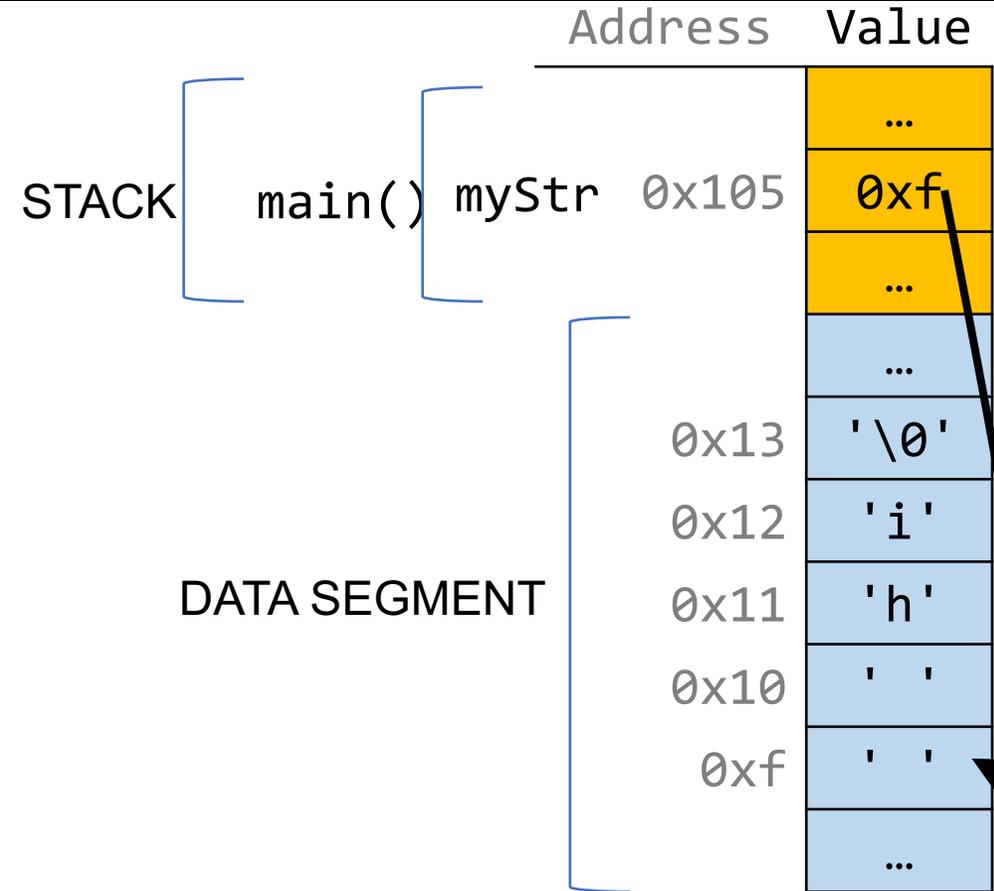
```
int main(int argc, char *argv[]) {  
    char *myStr = " hi";  
    skipSpaces(&myStr);  
    printf("%s\n", myStr);           // hi  
    return 0;  
}
```

STACK [main() [

Address	Value
	...
	...

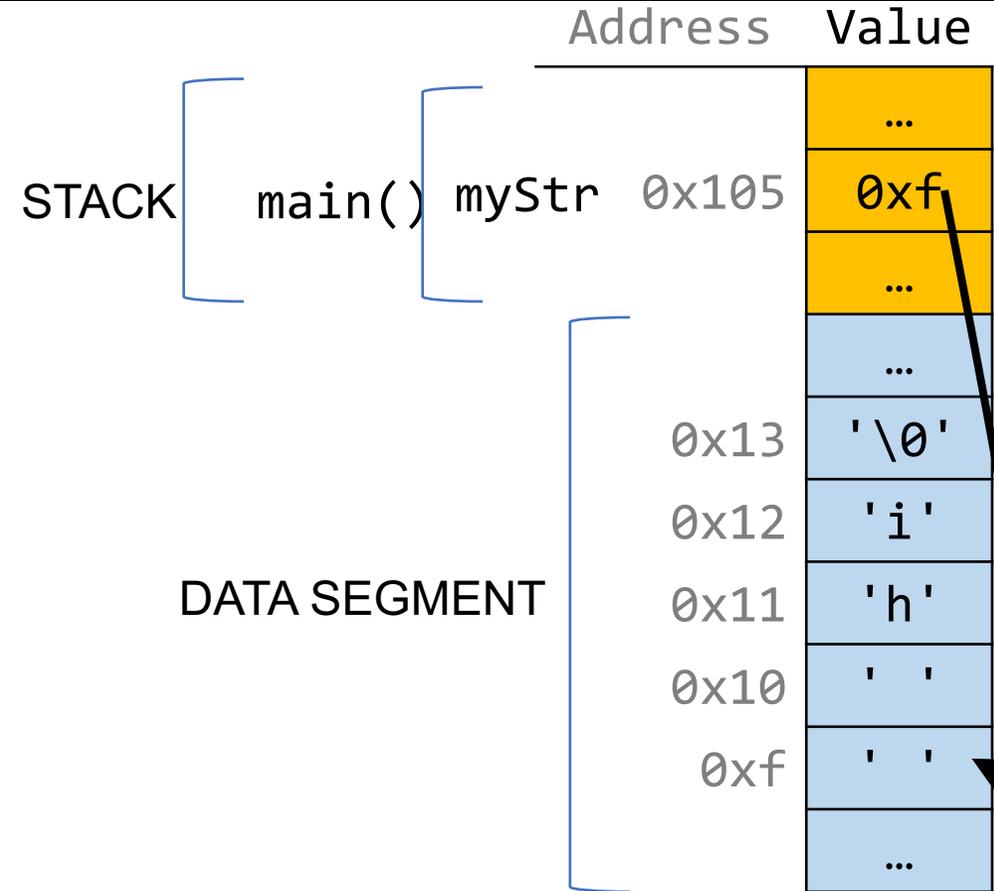
Pointers to Strings

```
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;  
}
```



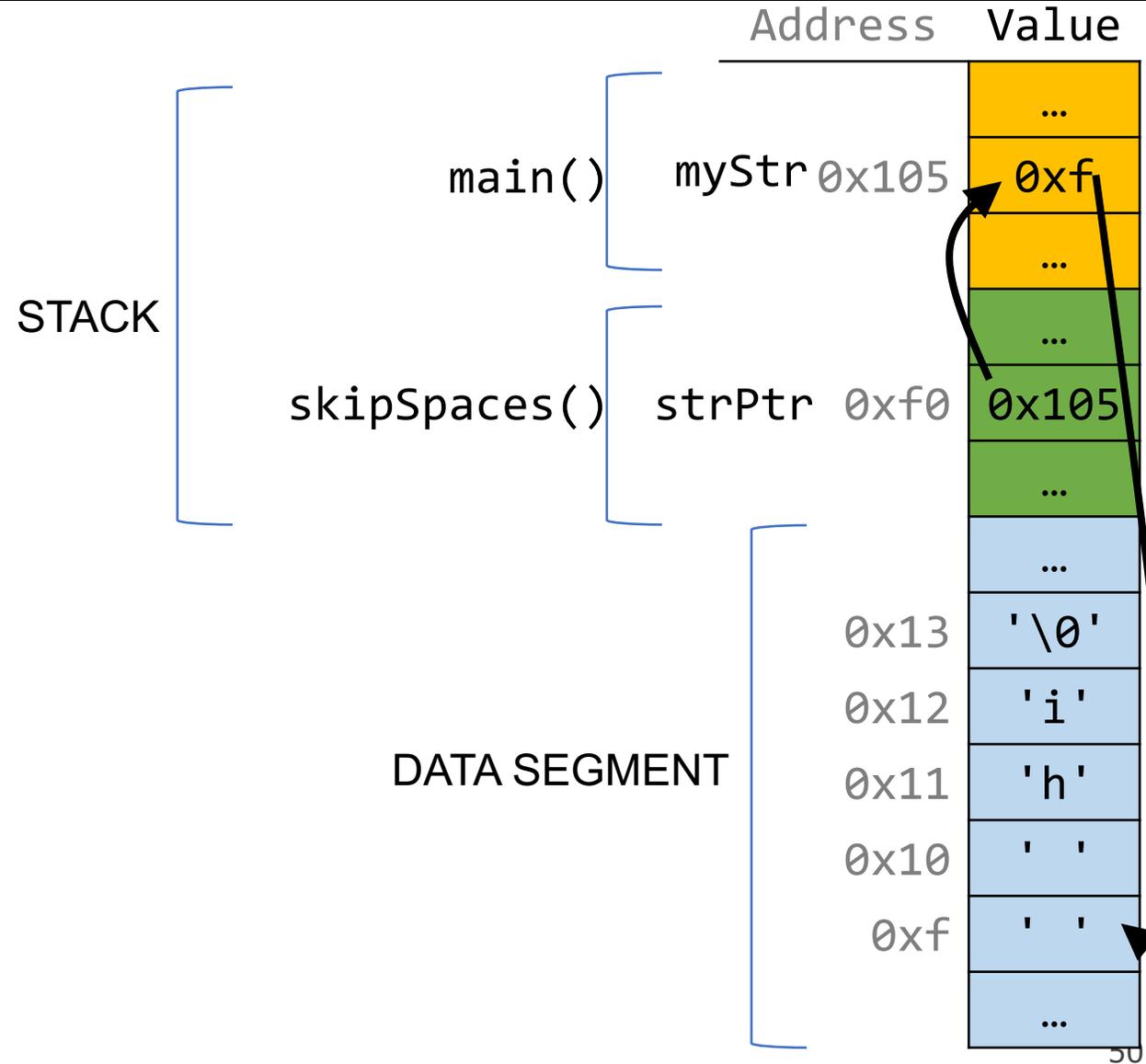
Pointers to Strings

```
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;  
}
```



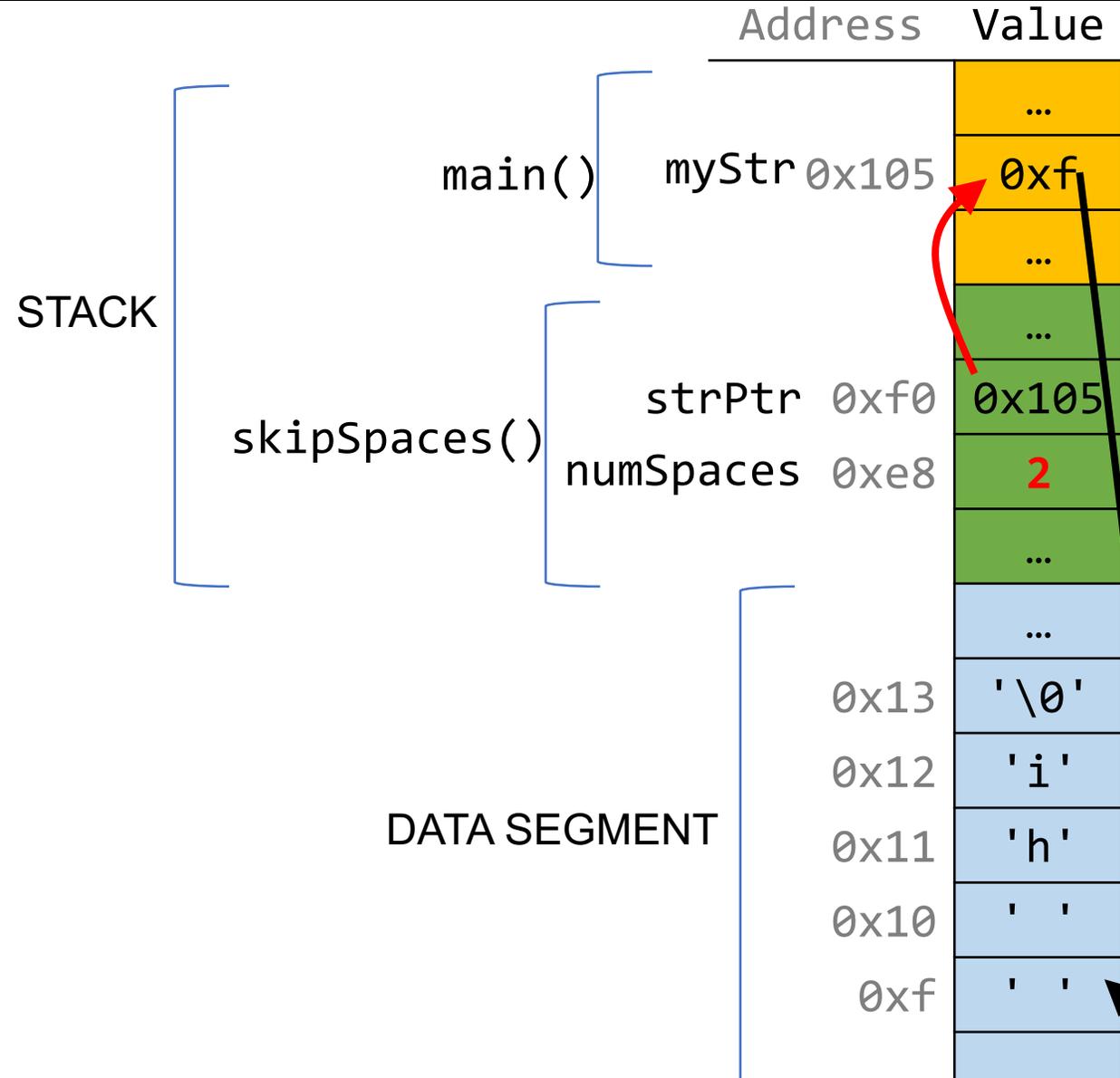
Pointers to Strings

```
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;  
}
```



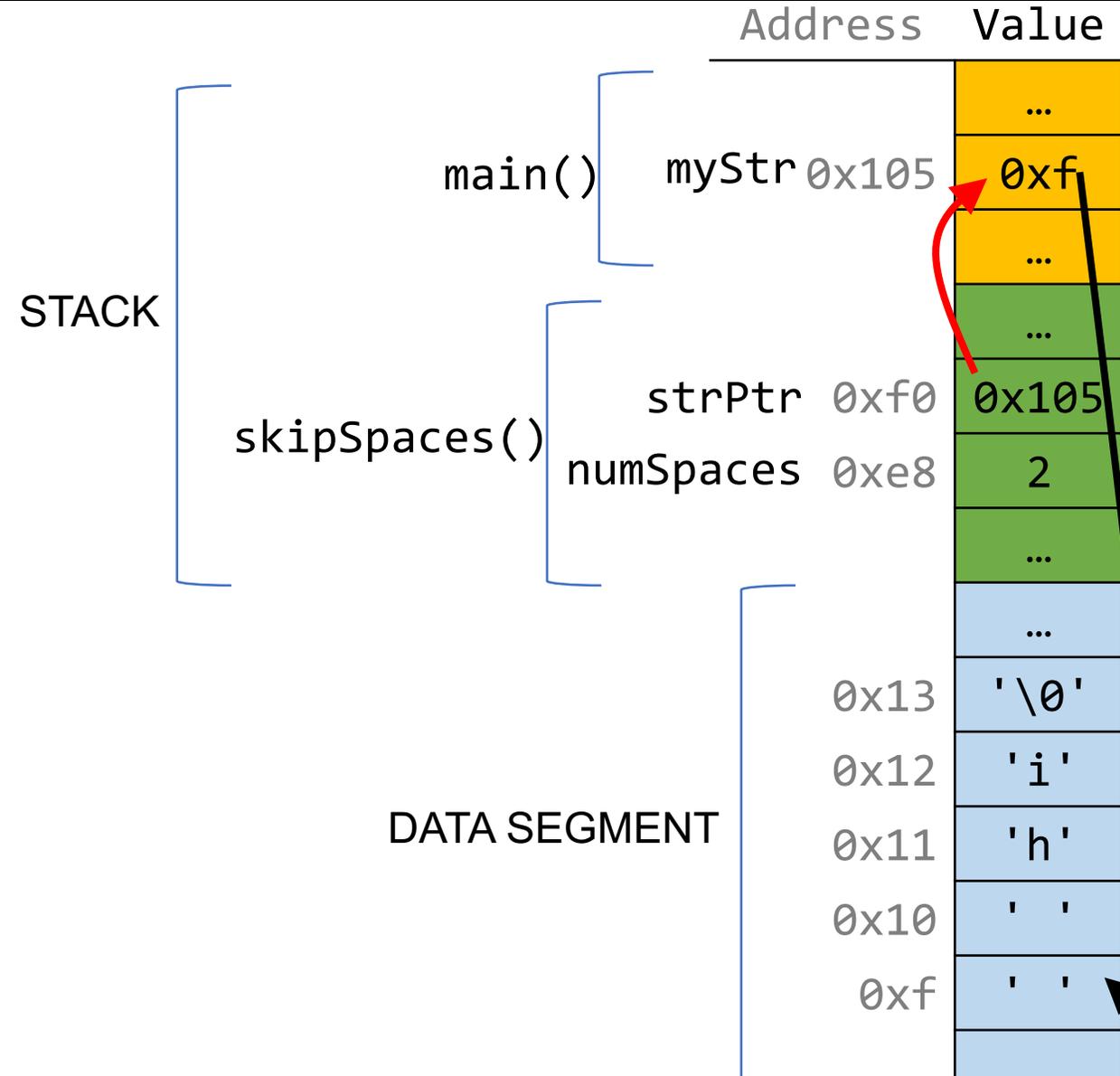
Pointers to Strings

```
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;  
}
```



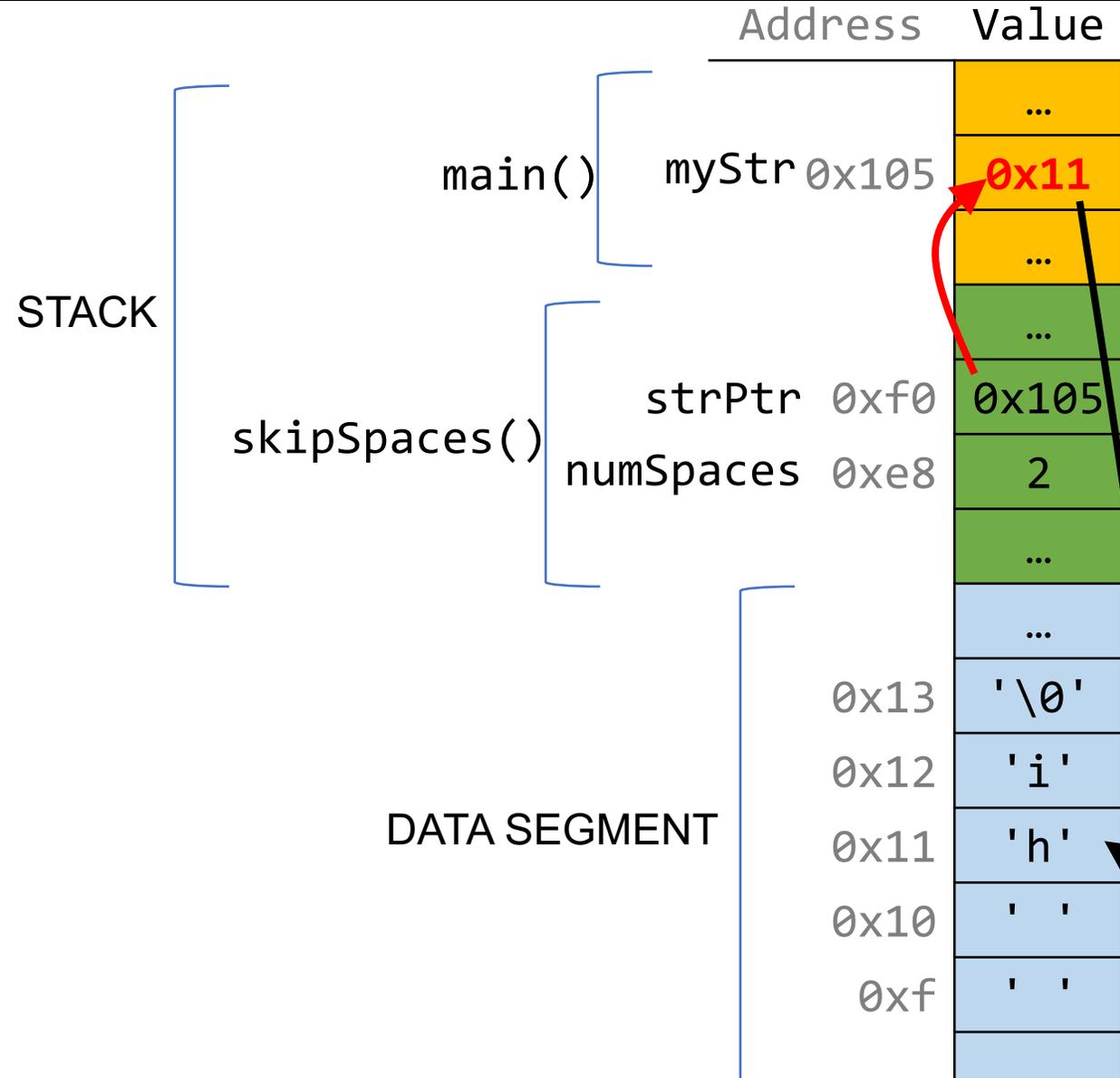
Pointers to Strings

```
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;  
}
```



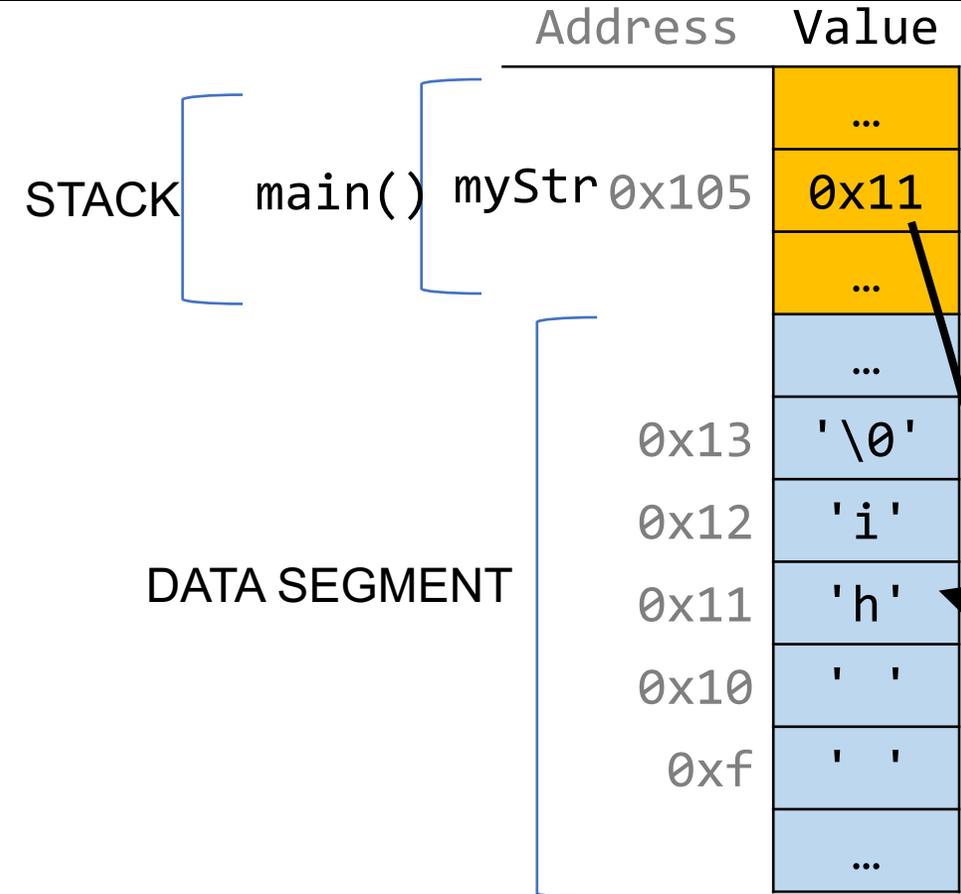
Pointers to Strings

```
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;  
}
```



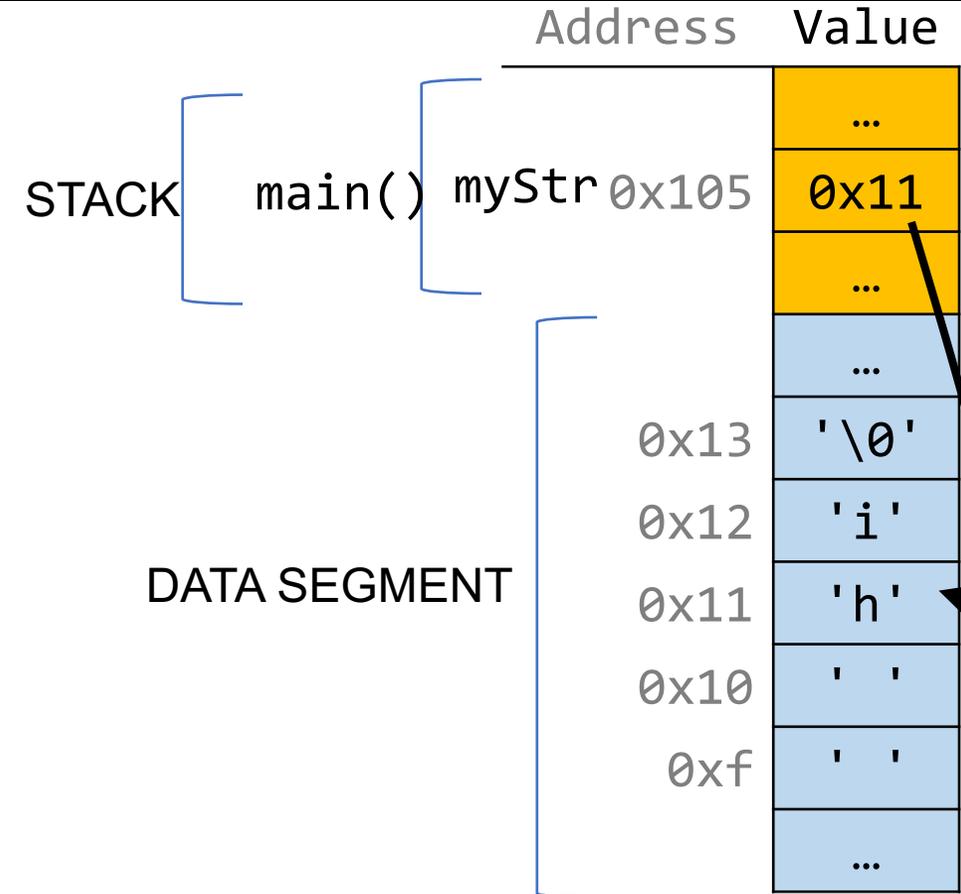
Pointers to Strings

```
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;  
}
```



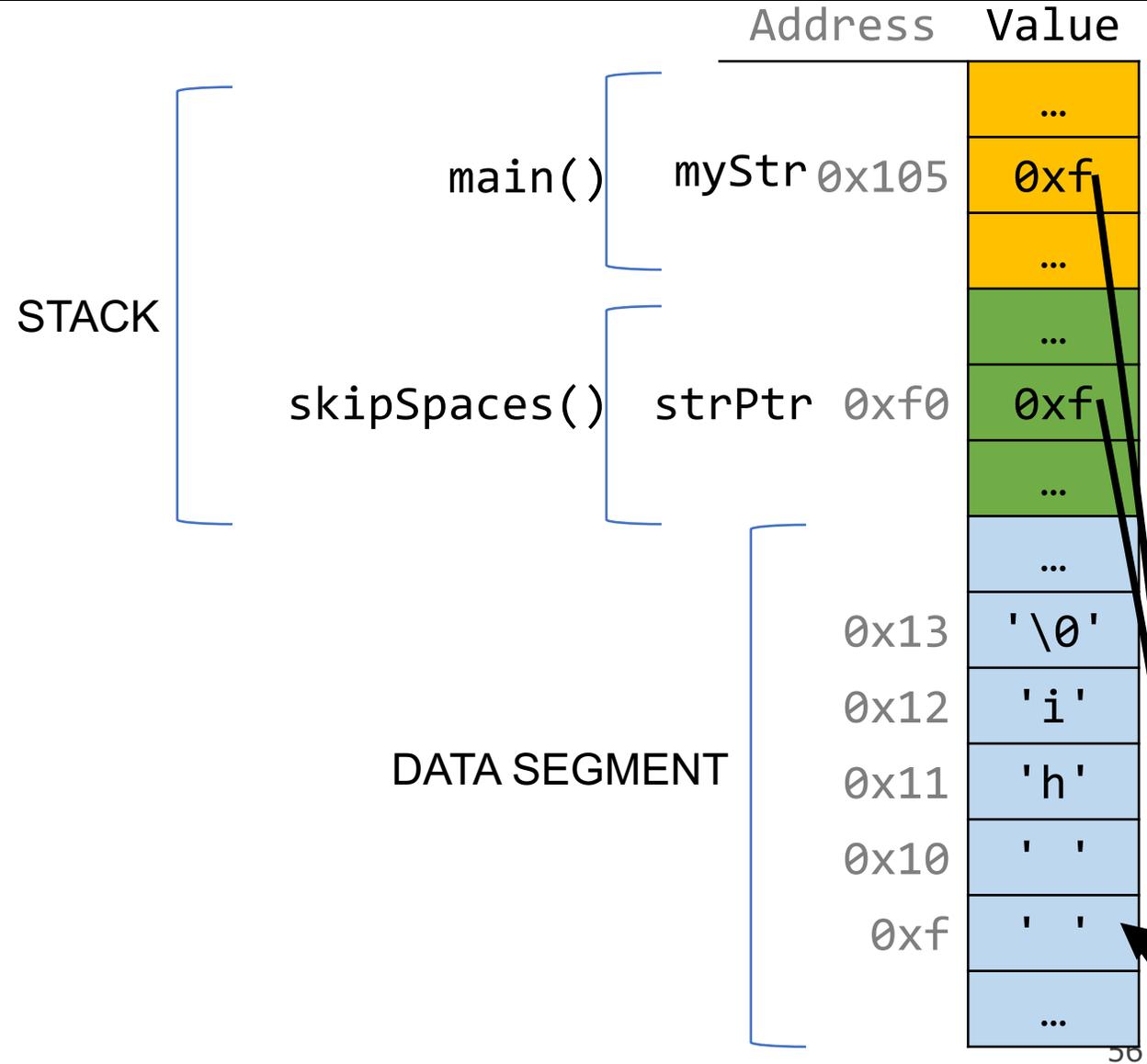
Pointers to Strings

```
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;  
}
```



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 4
- Double Pointers 42
- **Arrays in Memory** 57
- Arrays of Pointers 66
- Pointer Arithmetic 70
- Other topics: const, struct and ternary 80
- Live Session 99

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

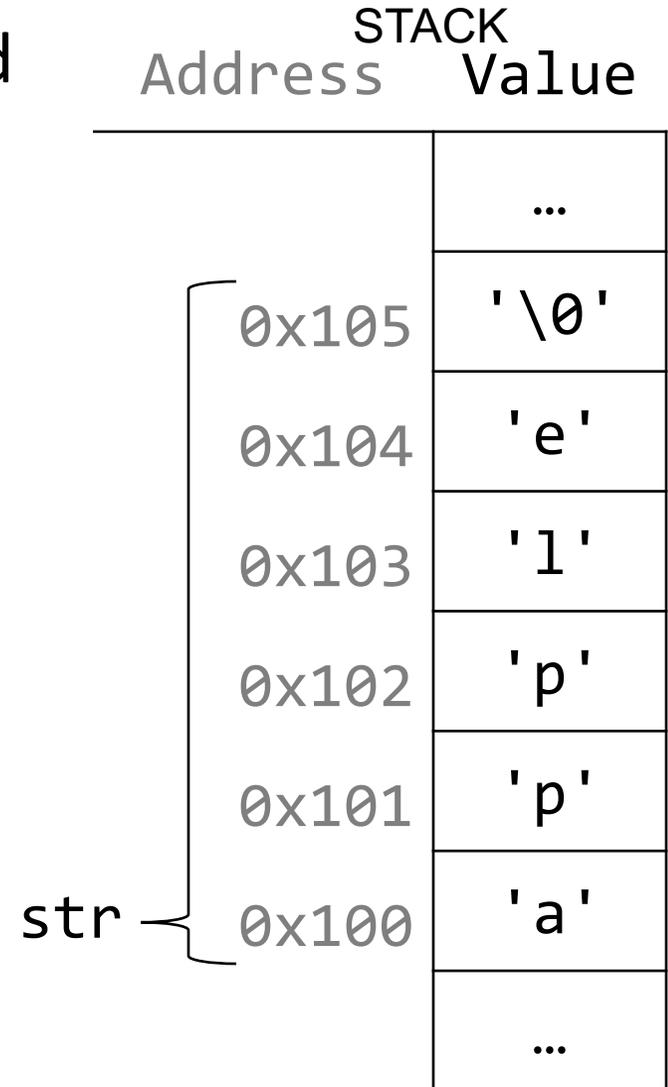
Arrays

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.

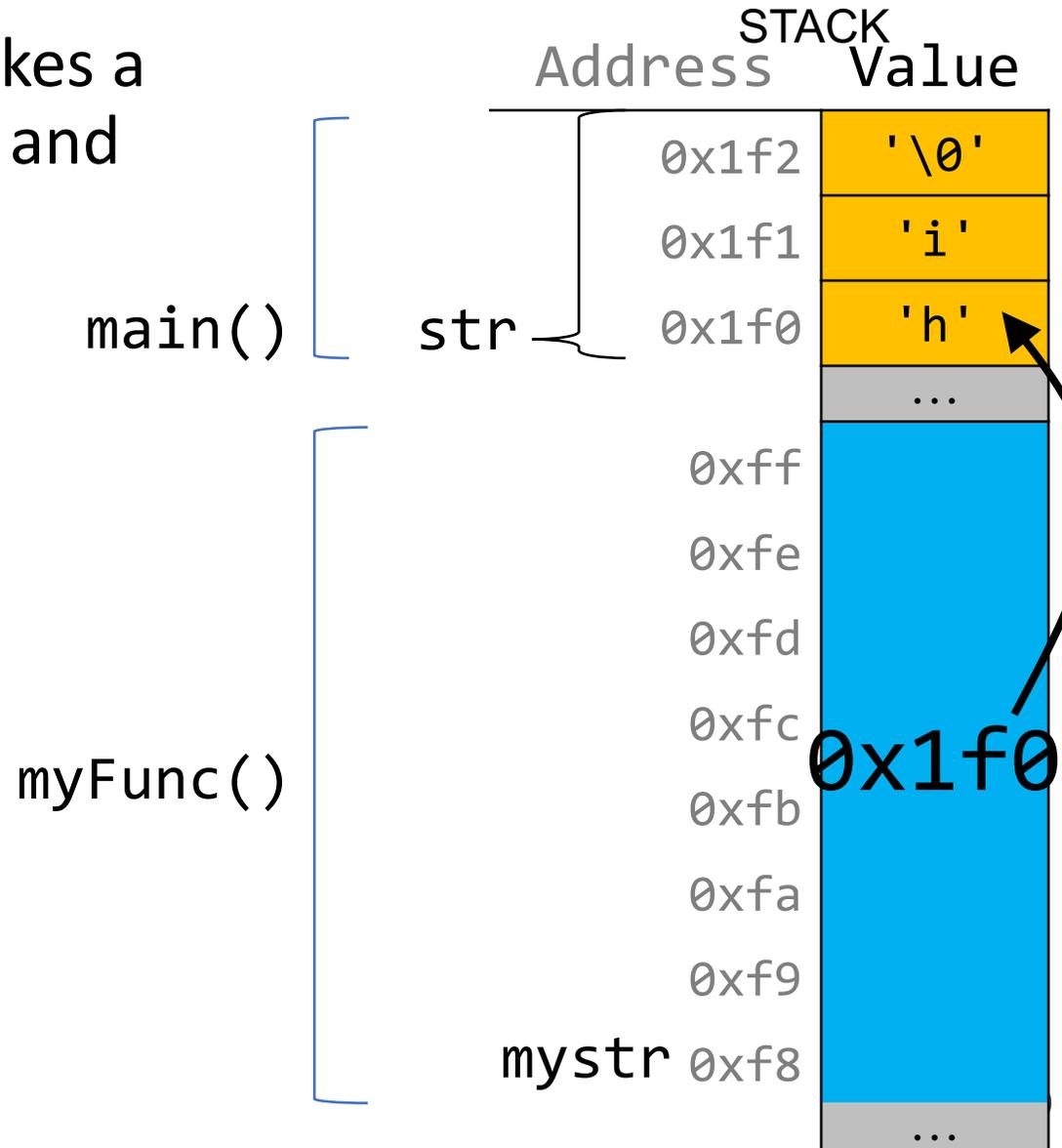
```
int nums[] = {1, 2, 3};  
int nums2[] = {4, 5, 6, 7};  
nums = nums2; // not allowed!
```

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

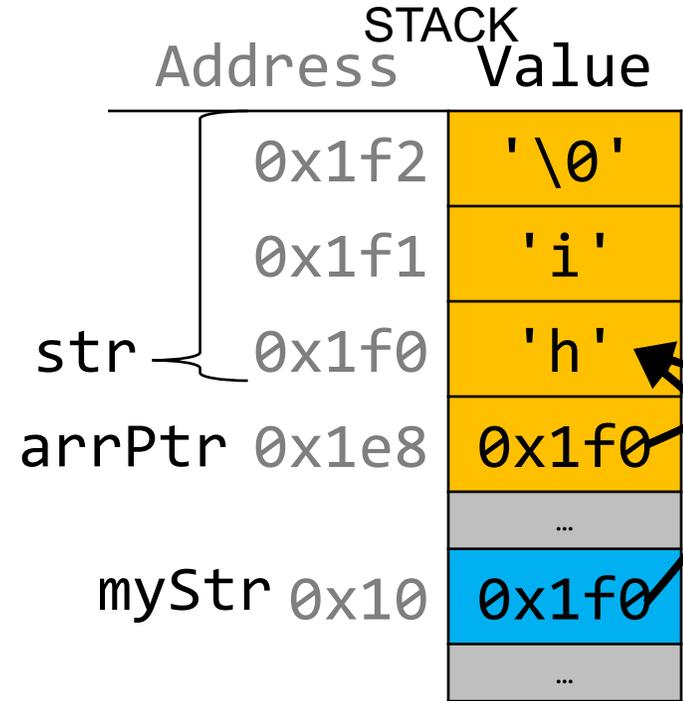
When you pass an **array** as a parameter, C makes a *copy of the address of the first array element* and passes it (a pointer) to the function.

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

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

main()

myFunc()

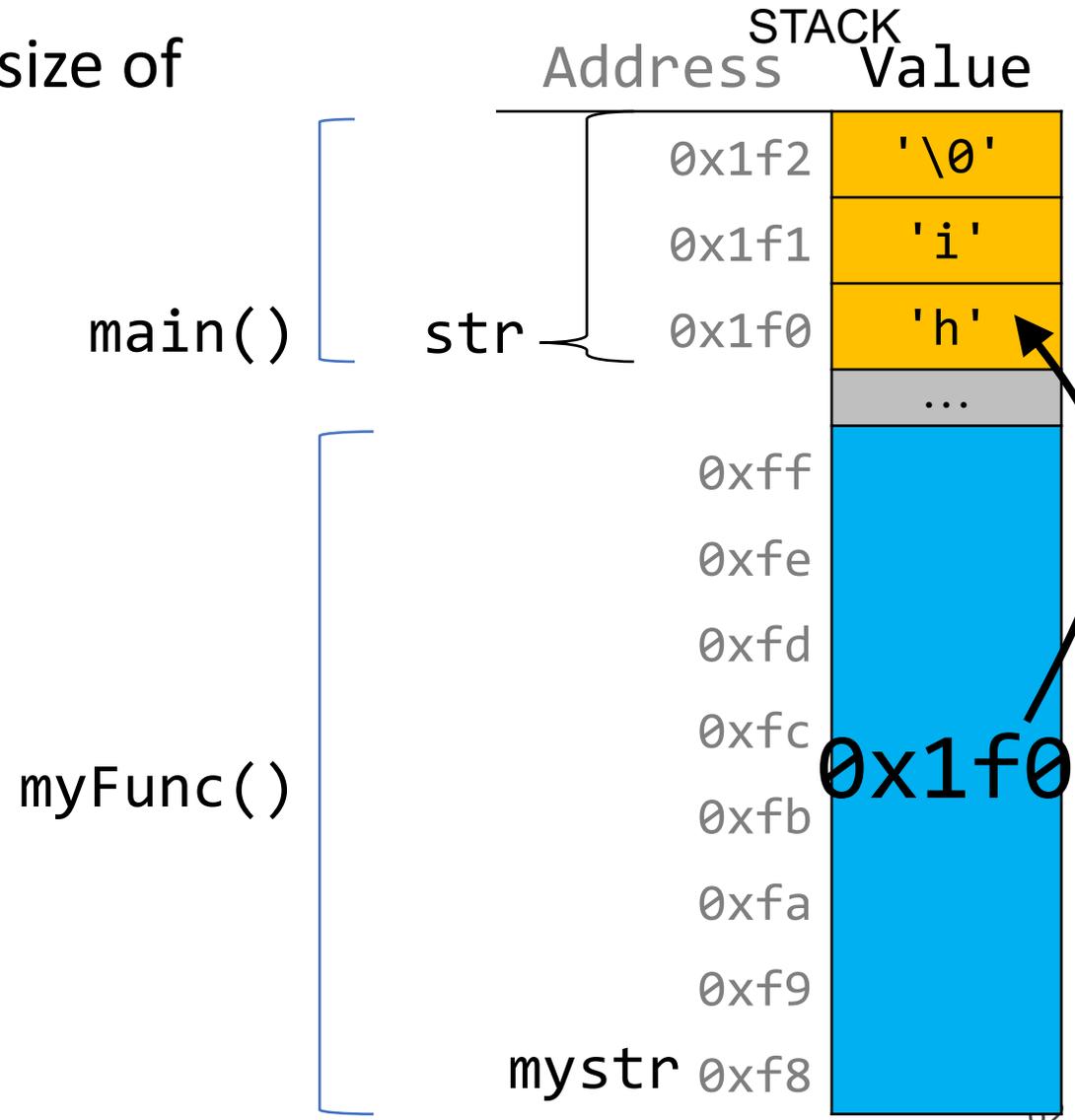


Arrays as Parameters

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);  
    ...  
}
```



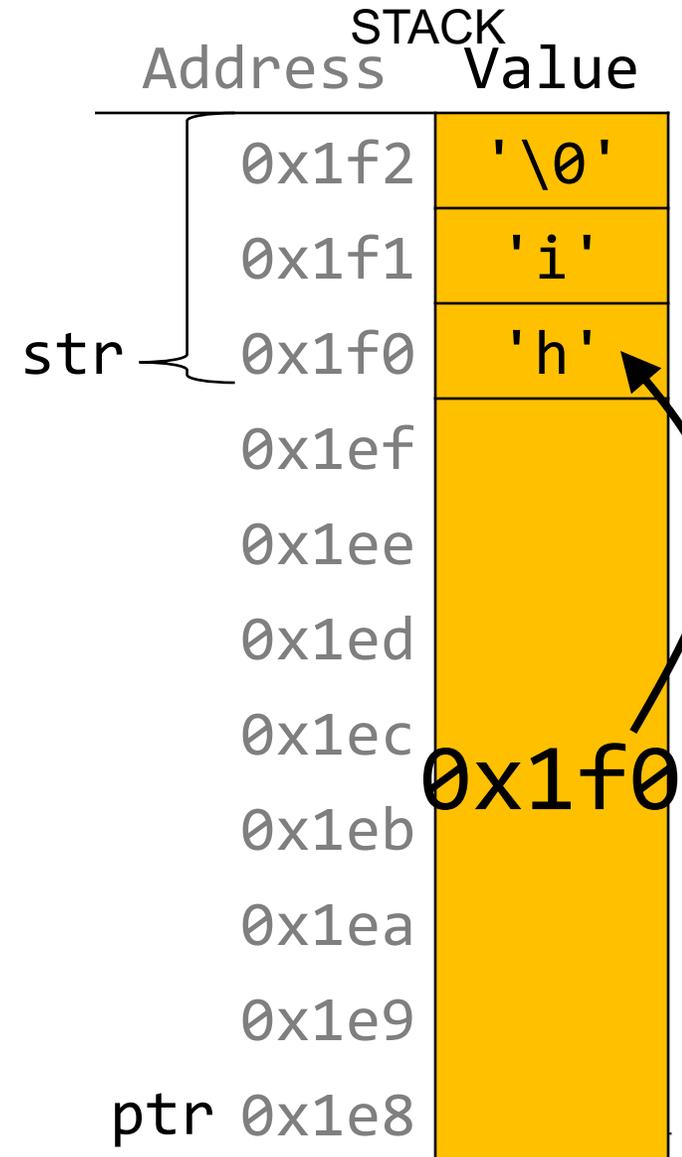
sizeof returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **sizeof** to get its full size.

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;  
    ...  
}
```

main()

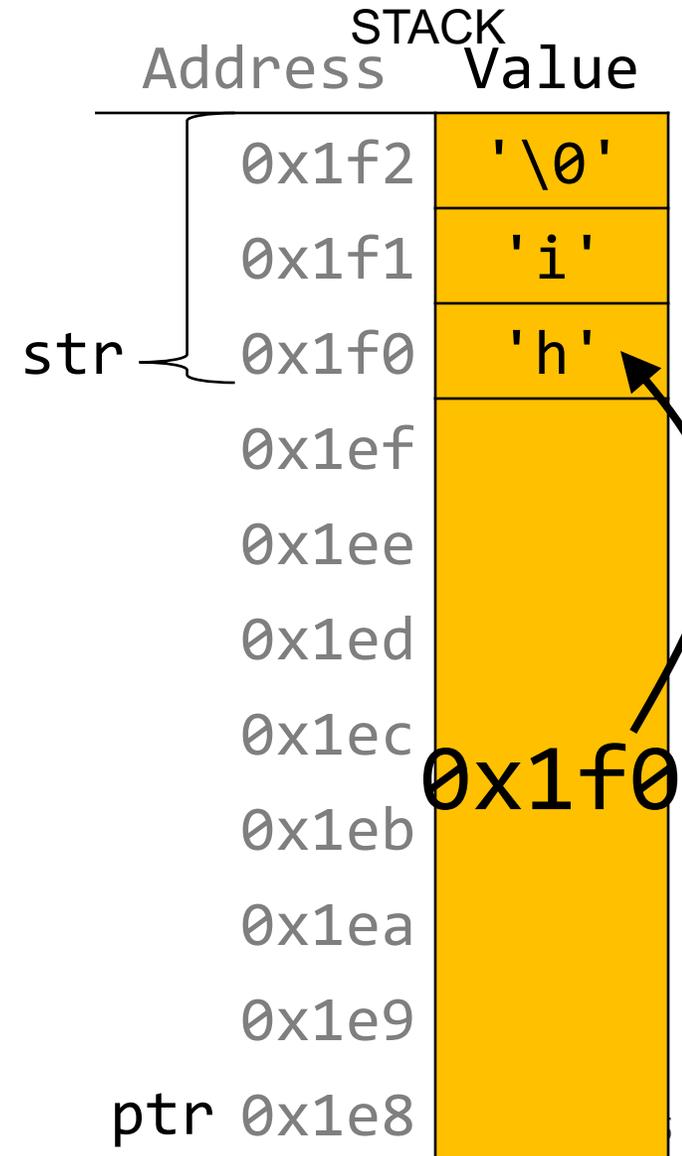


Arrays and Pointers

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;  
    ...  
}
```

main()



Lecture Plan

- Pointers and Parameters 4
- Double Pointers 42
- Arrays in Memory 57
- **Arrays of Pointers** 66
- Pointer Arithmetic 70
- Other topics: const, struct and ternary 80

```
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:

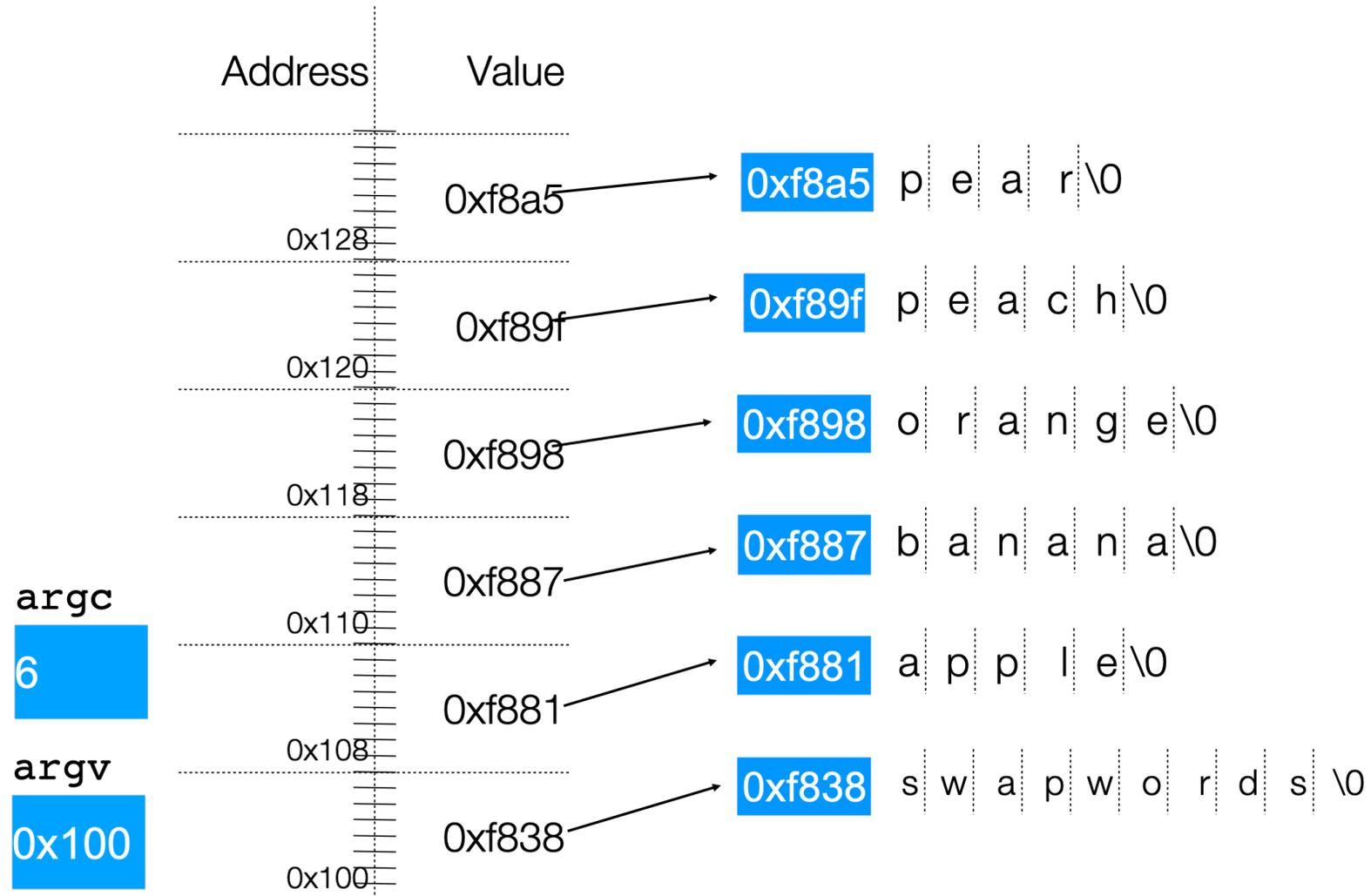
```
char *stringArray[5]; // space to store 5 char *s
```

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

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

Arrays Of Pointers

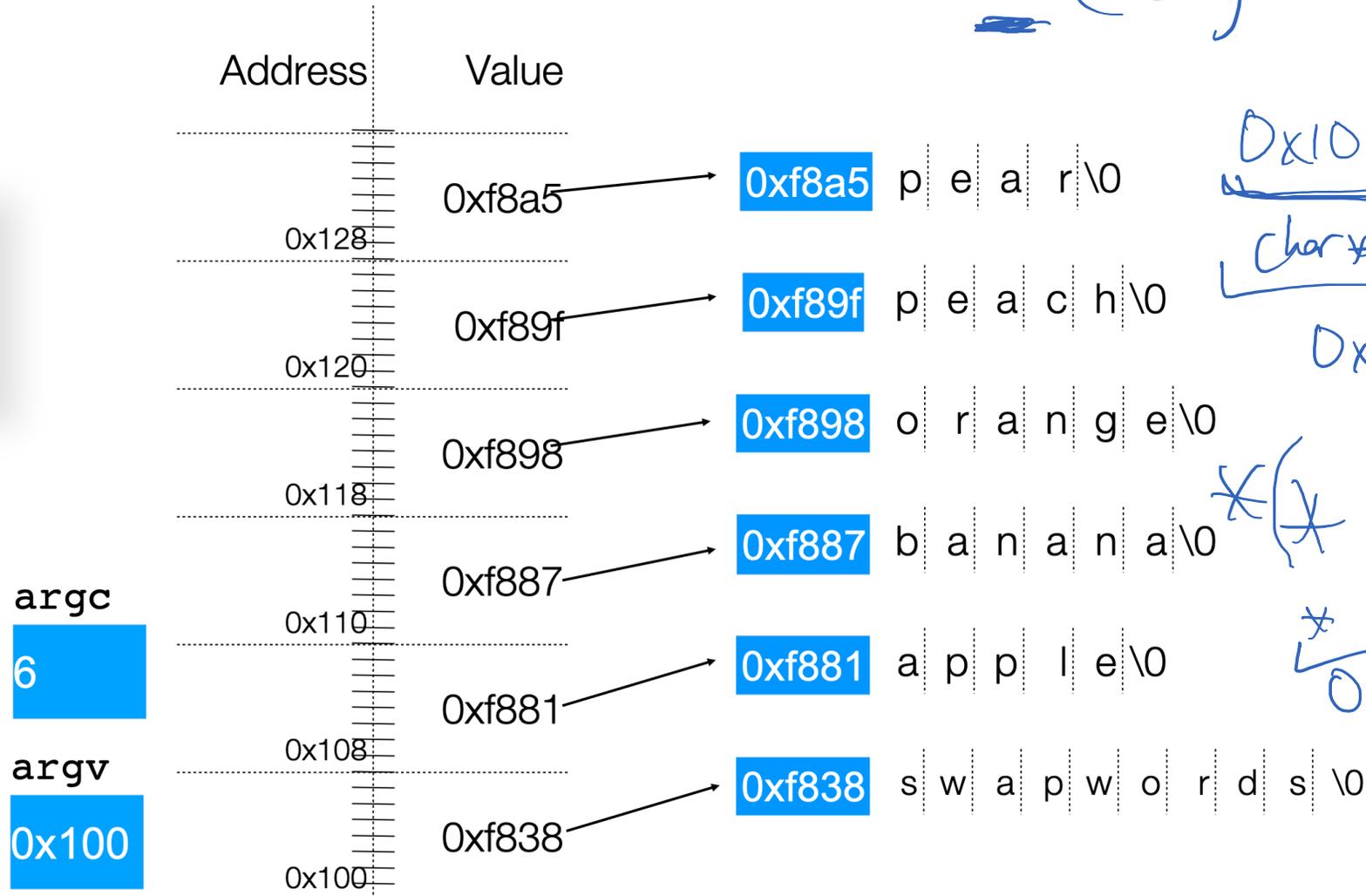
```
./swapwords apple banana orange peach pear
```



Arrays Of Pointers

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

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



x (argv + 2)

$0x100 + 2$
char

$0x110$

$*(x (argv) + 2)$

$0xf838 + 2$
char

0xf83a

Lecture Plan

- Pointers and Parameters 4
- Double Pointers 42
- Arrays in Memory 57
- Arrays of Pointers 66
- **Pointer Arithmetic** 70
- Other topics: const, struct and ternary 80
- Live Session 99

```
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).

```
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
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...

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
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...

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
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...

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

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 4
- Double Pointers 42
- Arrays in Memory 57
- Arrays of Pointers 66
- Pointer Arithmetic 70
- **Other topics: const, struct and ternary** 80
- Live Session 99

```
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 = ...
```

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

Next Time: dynamically allocated memory

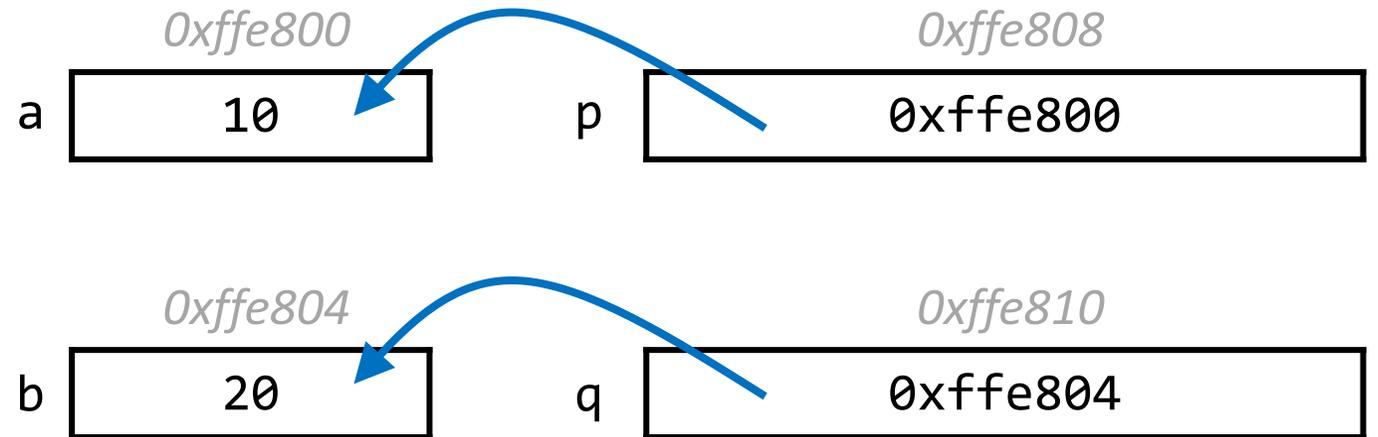
Live Session Slides

Slide 99

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 }
```

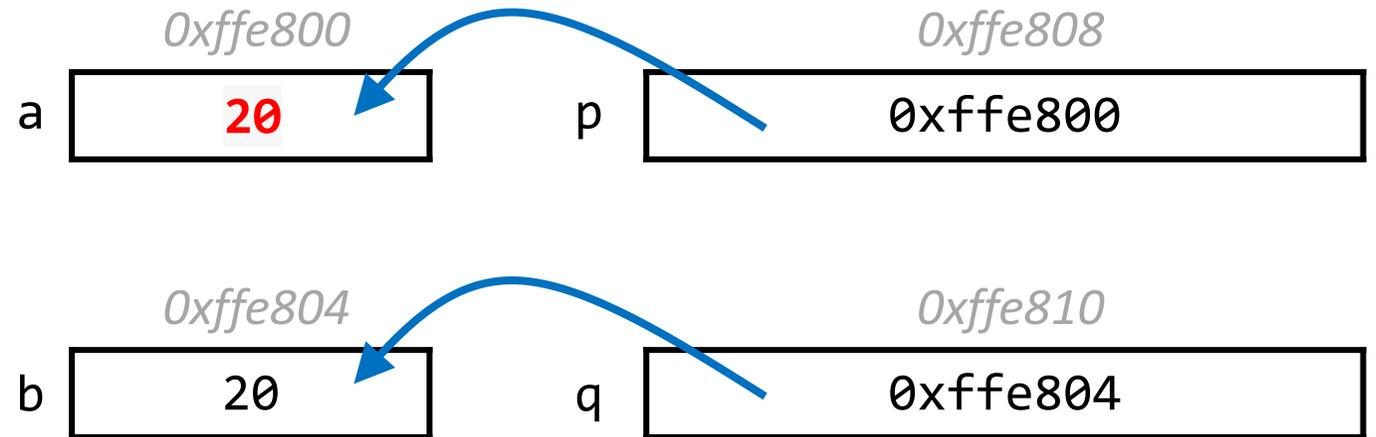
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



Pen and paper: A * Wars Story

```
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 }
```

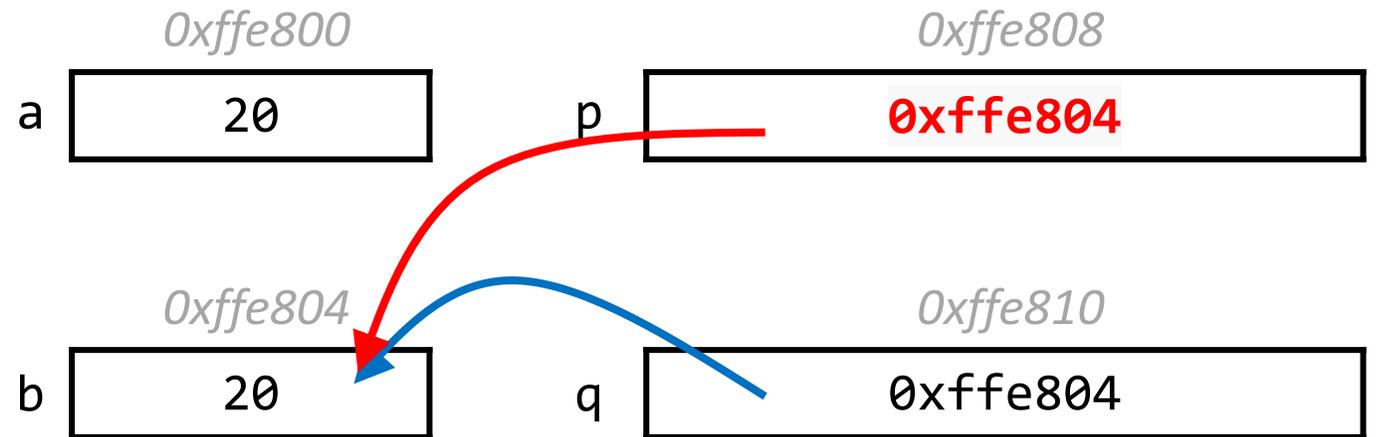
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



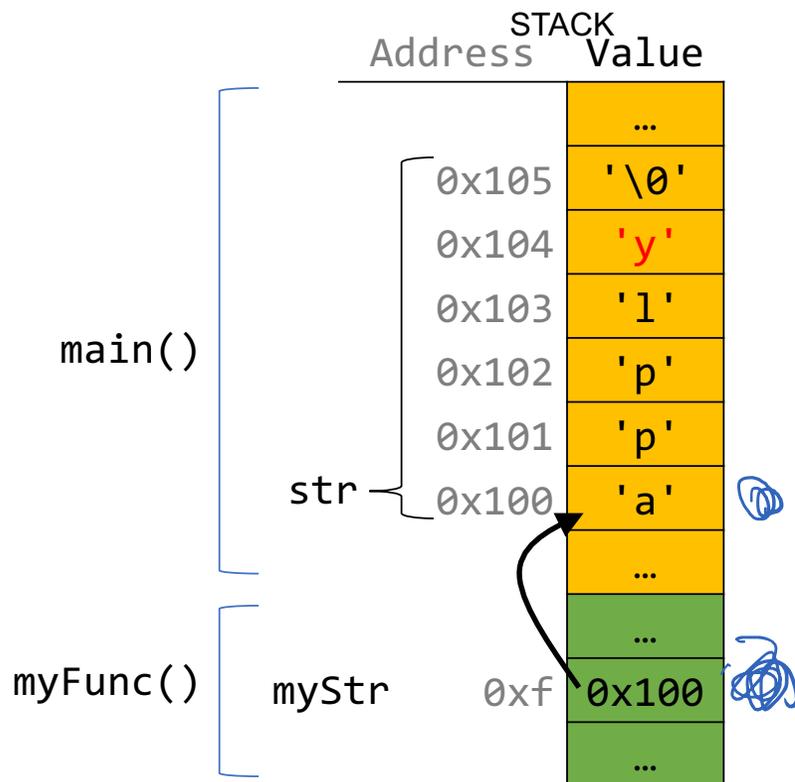
Pen and paper: A * Wars Story

```
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 }
```

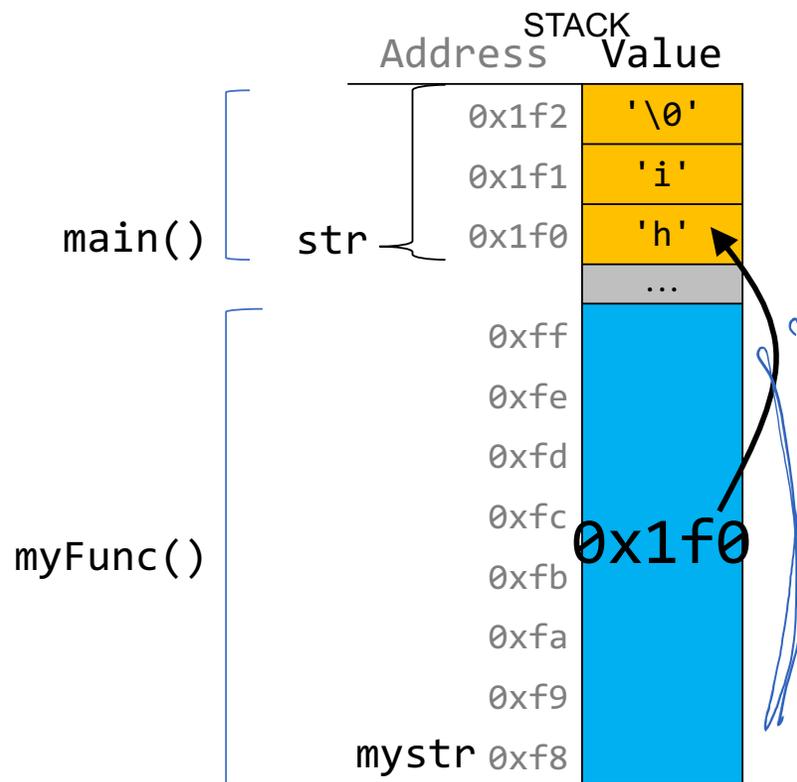
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



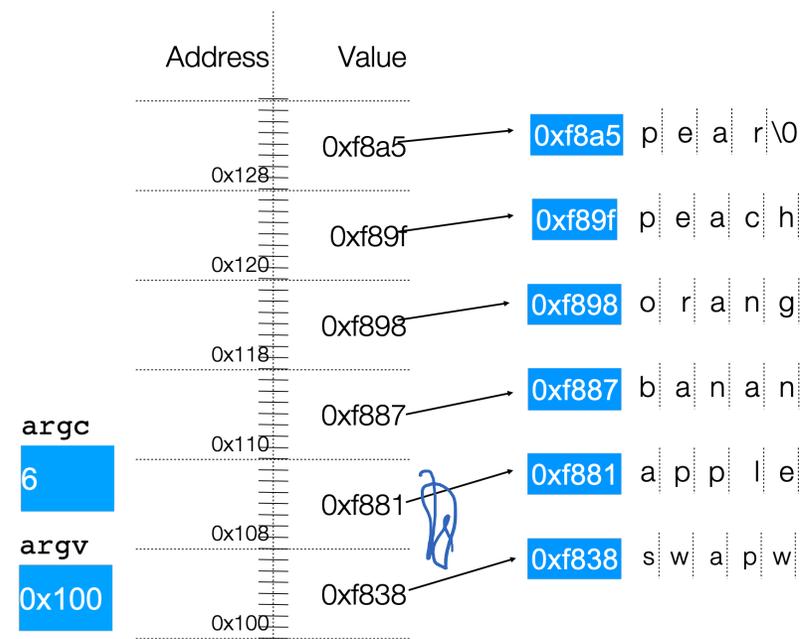
How to draw memory diagrams?



Slide 34



Slide 62

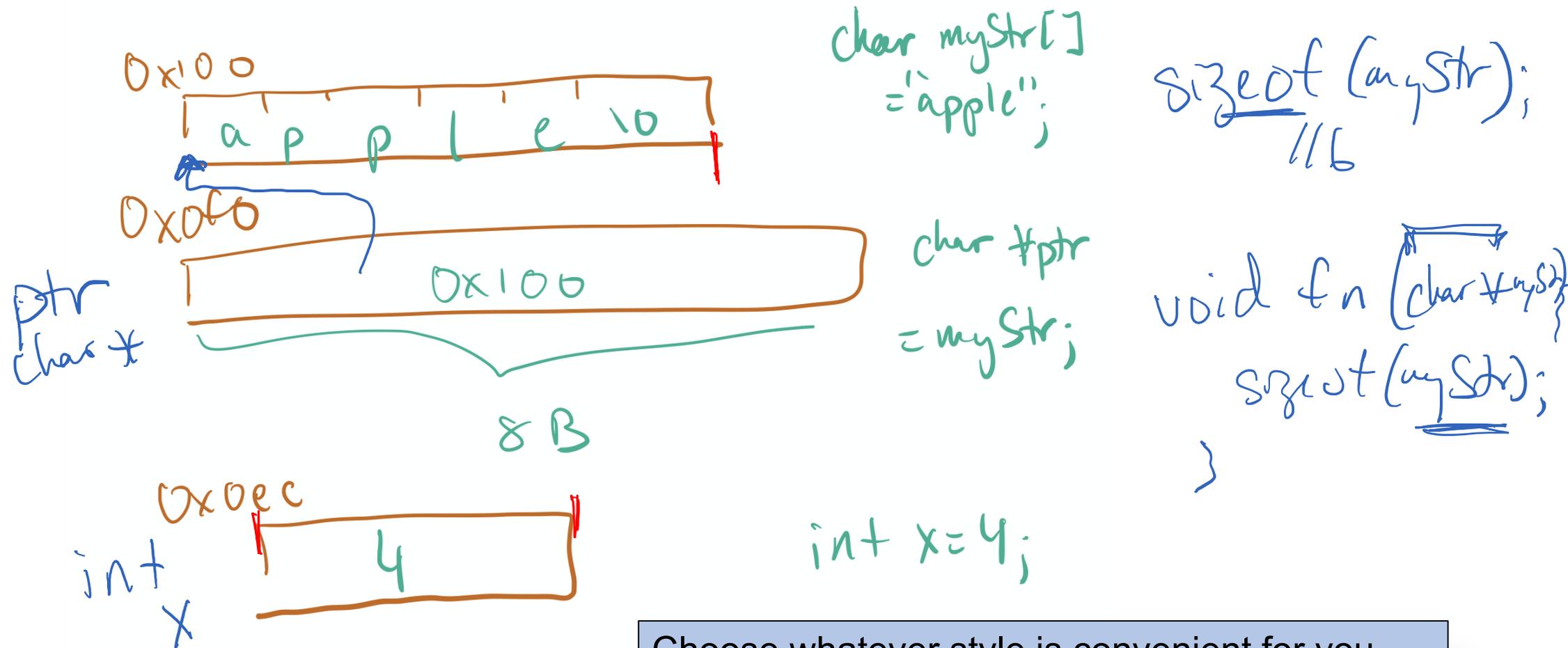


Slide 68

Discussion:
<https://edstem.org/us/courses/3085/discussion/226089>

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

How to draw memory diagrams?



Discussion:

<https://edstem.org/us/courses/3085/discussion/226089>

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

* Wars: Episode I (of 2)

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

```
char ch = 'r';
```

ch stores a char

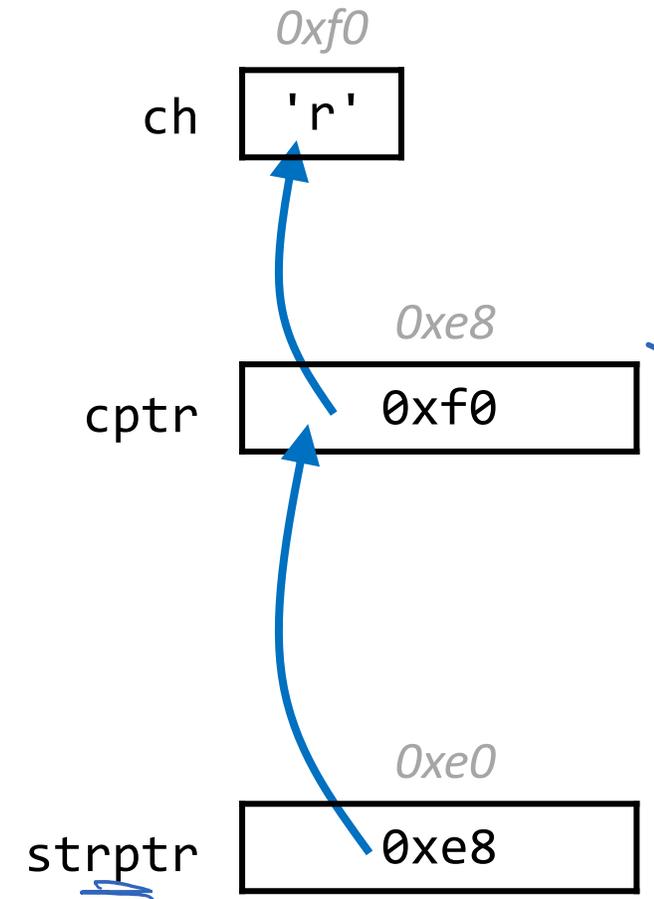
```
char _*ptr = &ch;
```

cptr stores an address of a char
(**points to** a char)

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

(Handwritten annotations: a bracket under the first asterisk, an arrow pointing to the second asterisk, and two arrows pointing to the 'c' and 'p' in '&cptr')

strptr stores an address of a char *
(**points to** a char *)



* Wars: Episode II (of 2)

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

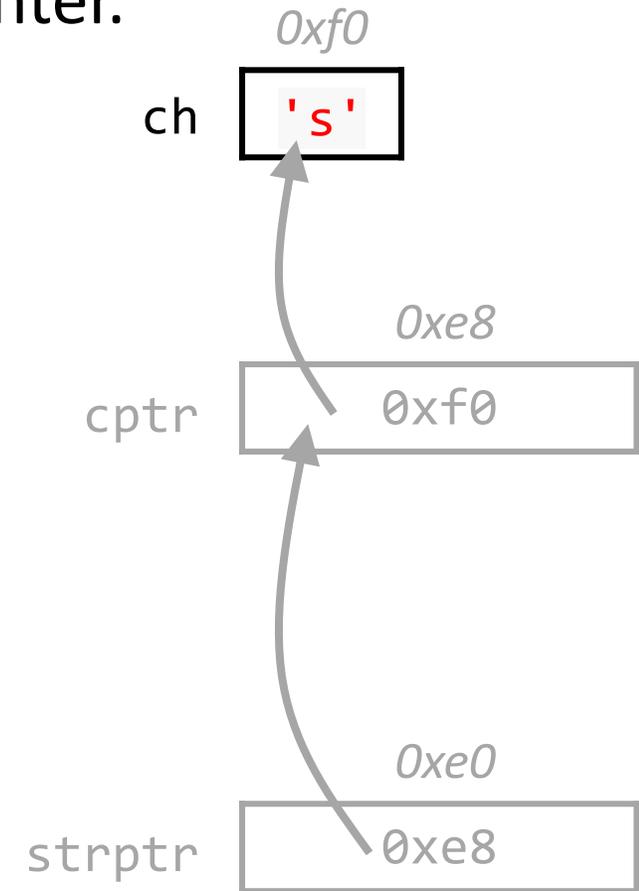
```
char ch = 'r';
```

```
ch = ch + 1;
```

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

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

Increment value stored in ch



* Wars: Episode II (of 2)

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

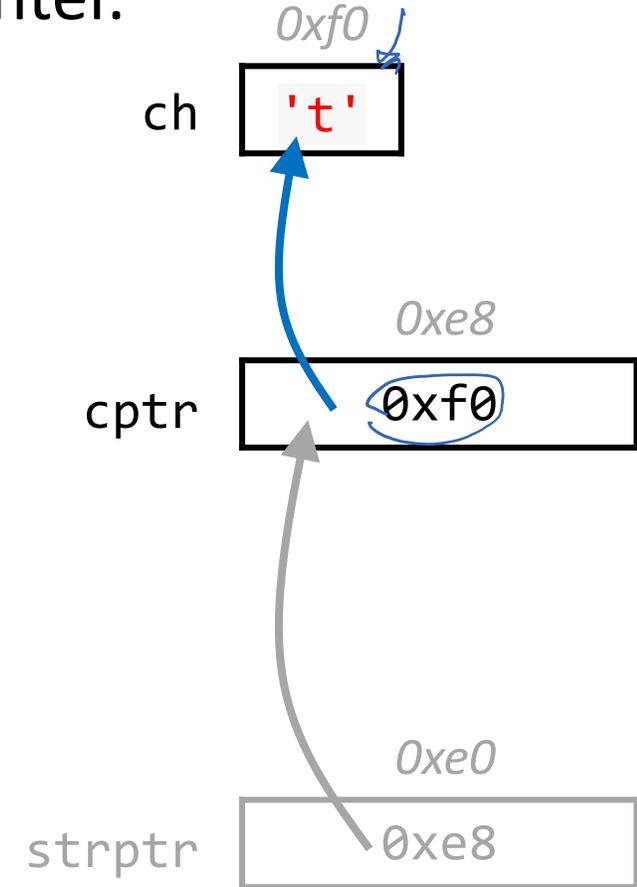
```
char ch = 'r';  
ch = ch + 1; ←
```

Increment value stored in ch

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

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;
```

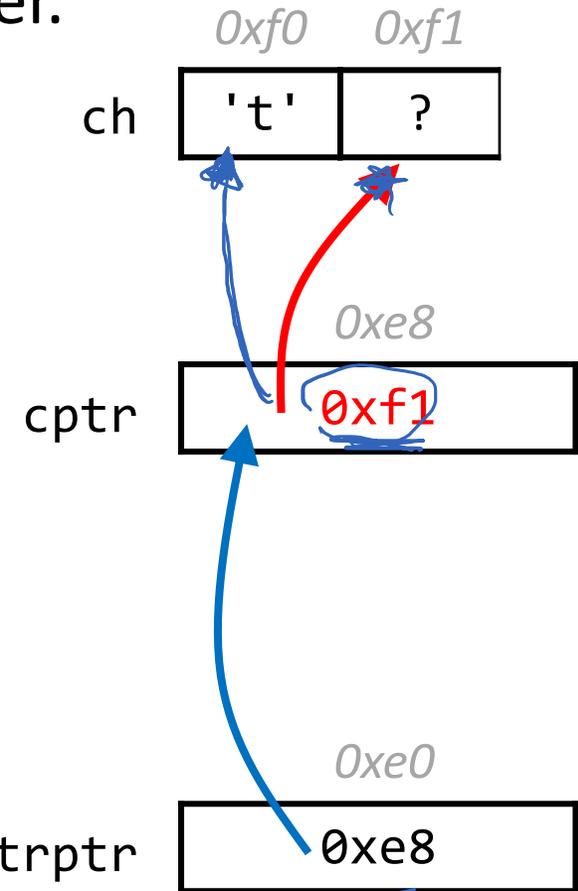
Increment value stored at
memory address in cptr
(increment char **pointed to**)

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

↑
*(0xe8)

0xf0 char*

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



Pointer arithmetic

Array indexing is “syntactic sugar” for pointer arithmetic:



⚠ Pointer arithmetic **does not work in bytes**; it works on the type it points to. On `int*` addresses scale by `sizeof(int)`, on `char*` scale by `sizeof(char)`.

- This means too-large/negative subscripts will compile 😊

`arr[99]` `arr[-1]`
`* (arr + 99)` `* (arr - 1)`

- You can use either syntax on either pointer or array.

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};
```

1. `int *ptr0 = arr;`
2. `int *elt0 = *arr;`
3. `int elt = *(arr + 3);`
4. `int **ptr1 = &ptr;`

```
// initializes stack array  
// with 4 ints
```

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 address
name> (except sizeof)

```
int arr[] = {3, 4, -1, 2};
```

1. `int *ptr0 = arr;`
2. `int *elt0 = *arr;`
3. `int elt = *(arr + 3);`
4. `int **ptr1 = &ptr;`

```
// initializes stack array  
// with 4 ints
```

Address arr

Value at address arr

The value at address <3 ints
after address arr>

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};
```

1. `int *ptr0 = arr;`
2. `int *elt0 = *arr;`
3. `int elt = *(arr + 3);`
4. `int **ptr1 = &ptr;`

```
// initializes stack array  
// with 4 ints
```

Type check with a diagram!

Is there a difference?

```
size_t get_total_strlen(char *strs[], size_t num) {  
    ...  
}
```

Intent: strs is an array of strings

```
void *skip_spaces(char **p_str) {  
    ...  
}
```

Intent: p_str is a pointer to a string

No difference to the compiler—it's char**!
But it clarifies the *intent* of a function/a parameter *for the programmer*.

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!**

```
1 size_t get_total_strlen(char *strs[], size_t num) {  
2     size_t tot = 0;  
3     for (int i = 0; i < num; i++) {  
4         // fill this in ←  
5     }  
6     return tot;  
7 }
```

1. _____
2. _____

buffer
split

str[0] : Luke
str[1] : I am
str[2] : your dad
16



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**!

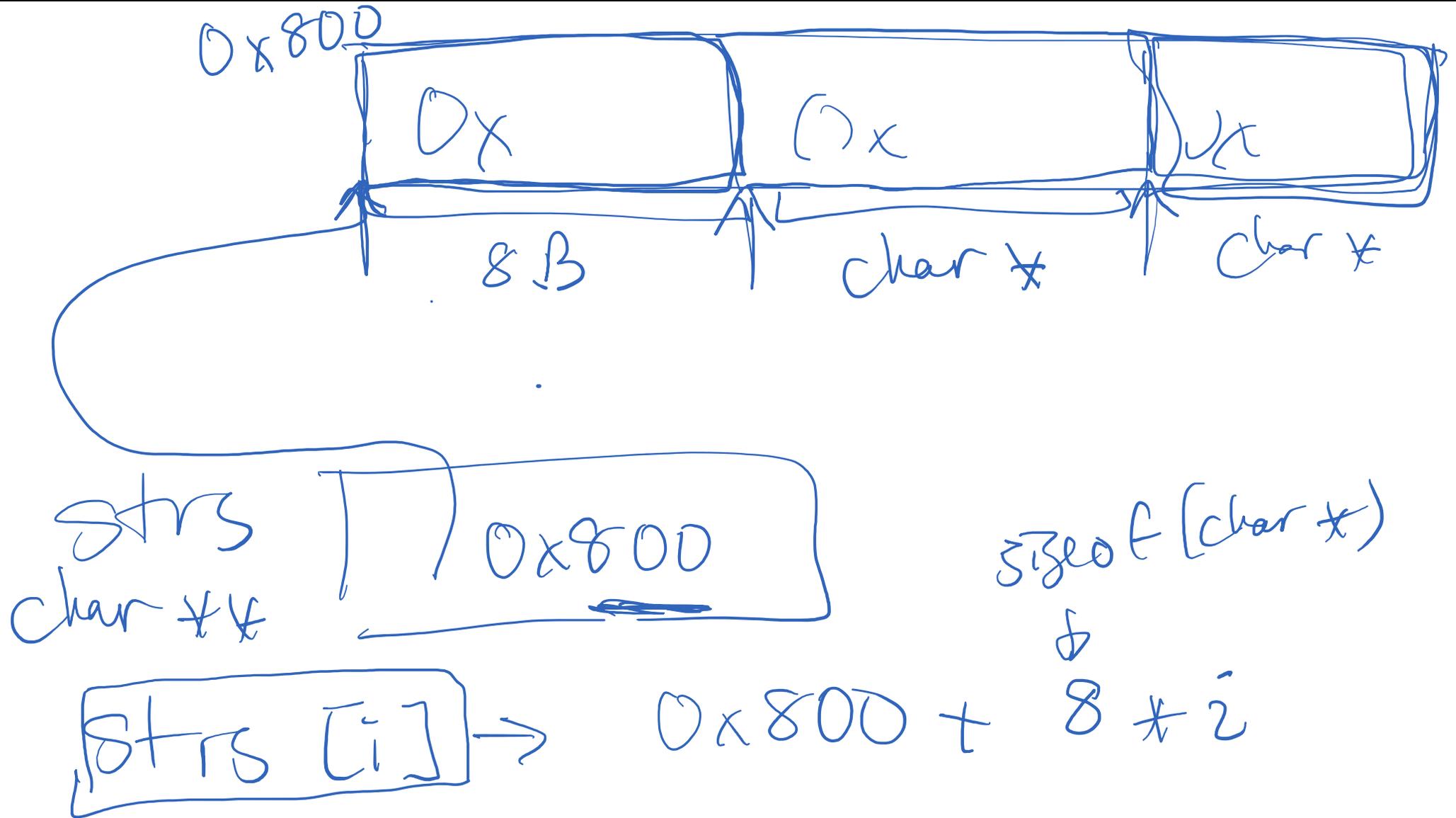
```
1 size_t get_total_strlen(char *strs[], size_t num) {
2     size_t tot = 0;
3     for (int i = 0; i < num; i++) {
4         // TODO: fill this in two ways
5     }
6     return tot;
7 }
```

Equivalent:

1. tot += strlen(strs[i]);

2. tot += strlen(*(strs + i));

Exercise: Implementation



char * v

strs

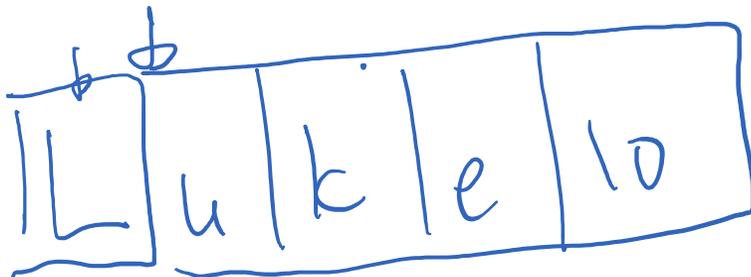


0xe7d0

char *



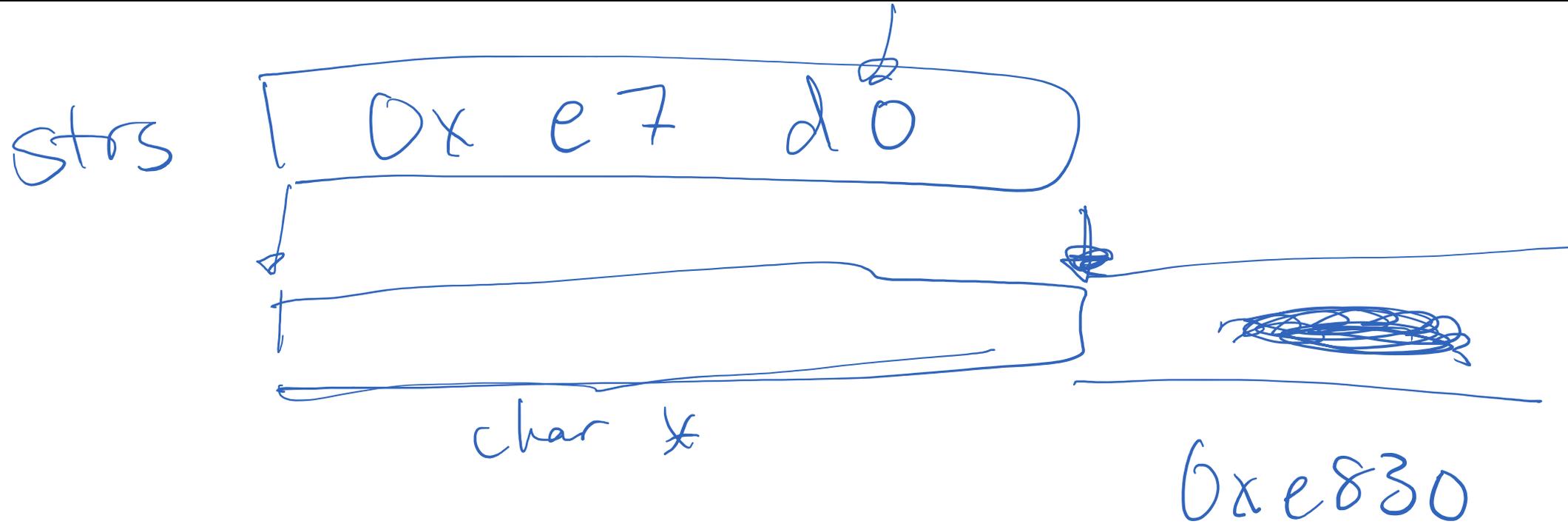
0xe820



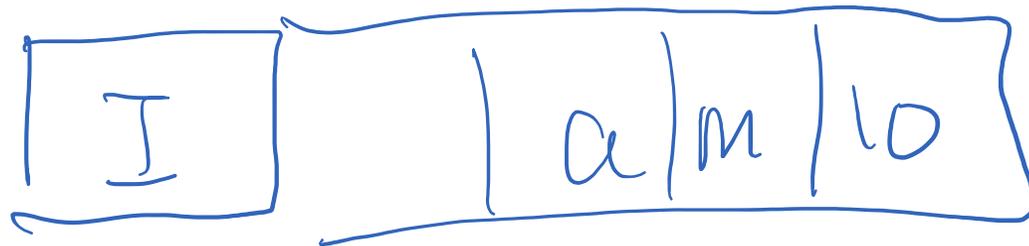
*strs + 1

0xe820

0xe821



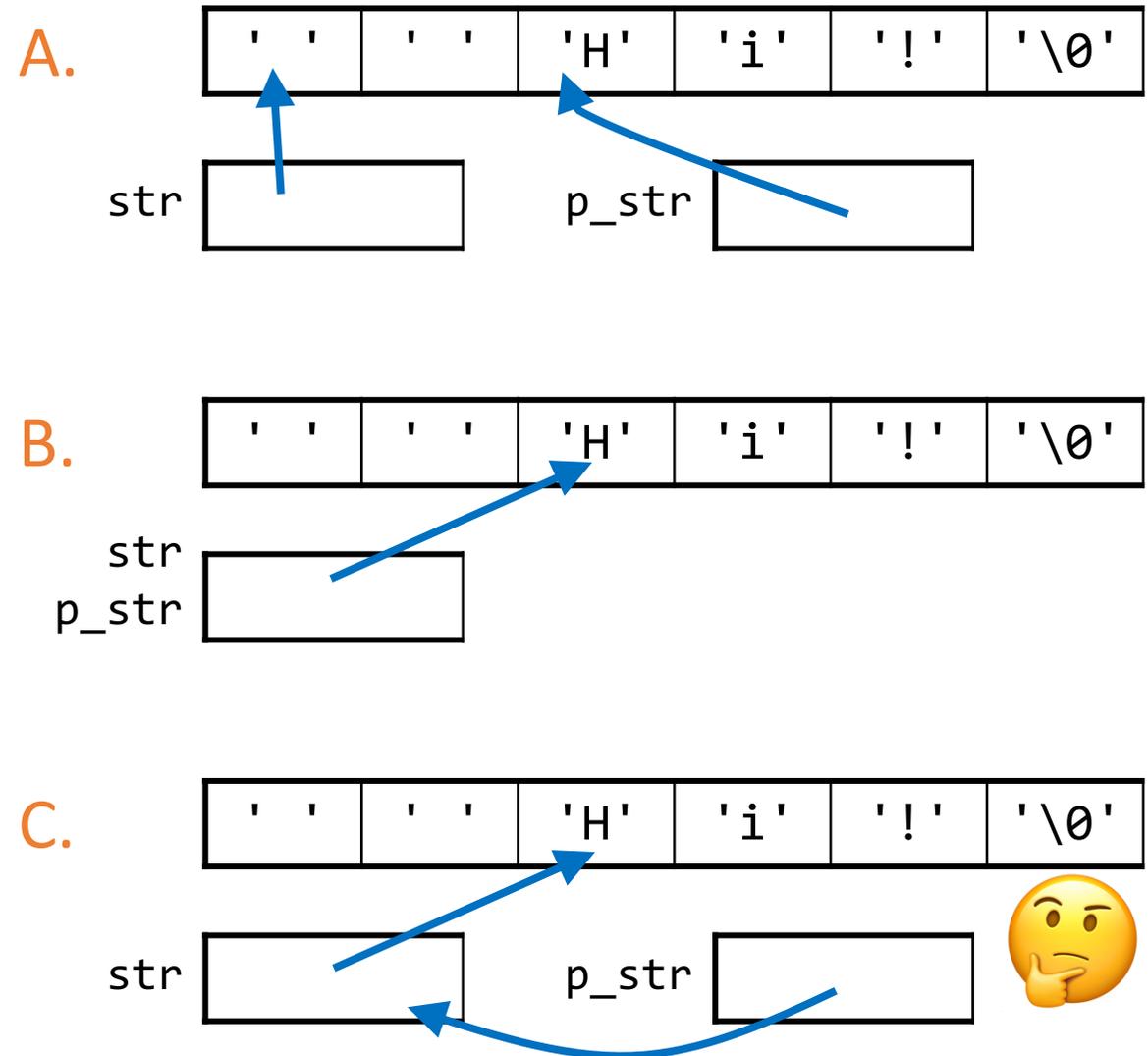
0xe830



Skip spaces

```
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 }
```

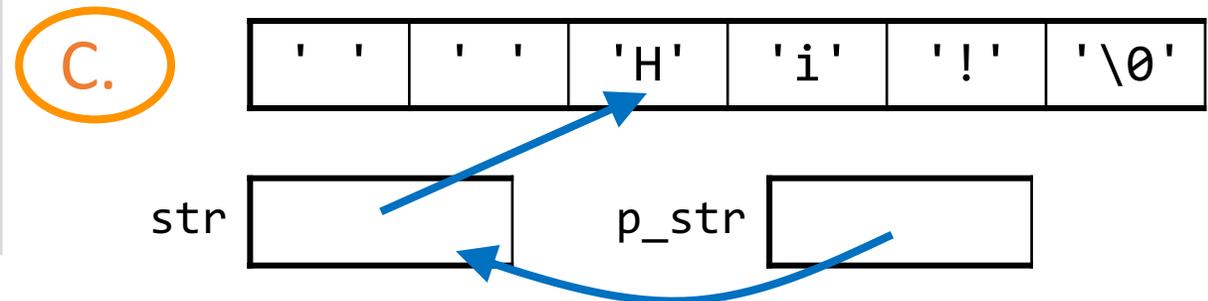
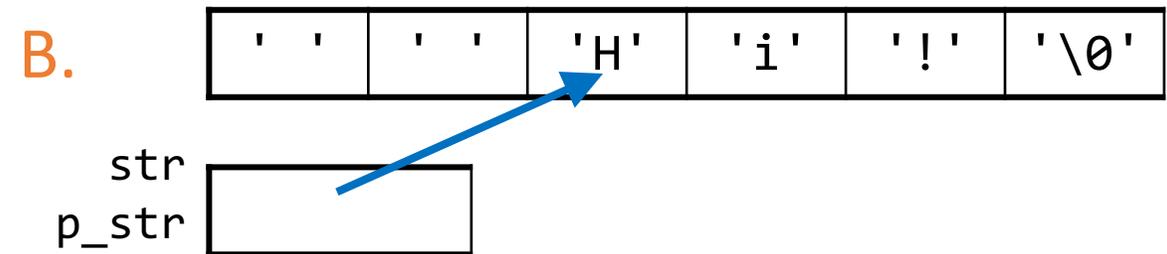
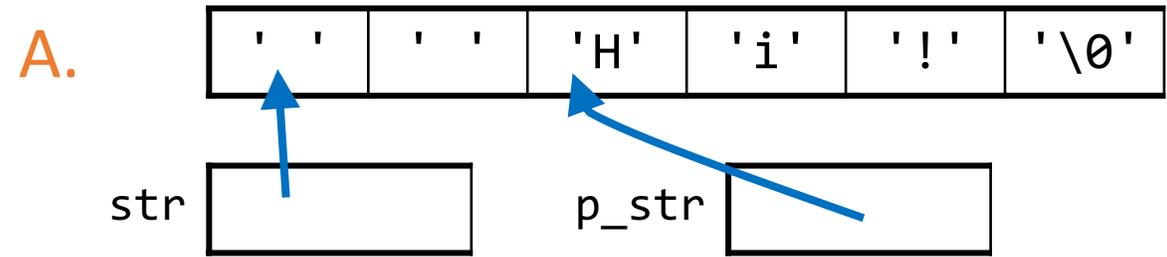
What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to main)?



Skip spaces

```
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 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

Discussion:

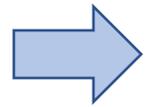
<https://edstem.org/us/courses/3085/discussion/226556>

const (from the quiz)

```
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 (from the quiz)

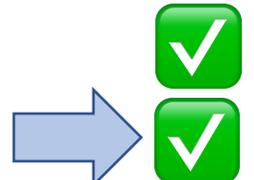


```
1 char buf[6];  
2 strcpy(buf, "Hello");  
3 const char *str = buf;  
4 str[0] = 'M';  
5 str = "Mello";  
6 buf[0] = 'M';
```

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 (from the quiz)



```
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 (from the quiz)

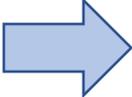


```
1 char buf[6];  
2 strcpy(buf, "Hello");  
3 const char *str = buf;  
4 str[0] = 'M';  
5 str = "Mello";  
6 buf[0] = 'M';
```

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.

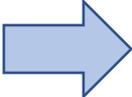
const (from the quiz)

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

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 (from the quiz)

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

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 (from the quiz)

✓ 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.

char *ptr = 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.

char *str = "Mello";
char buf[] = "Mello";