COMP201
Computer Systems & Programming

Lecture #7 – Arrays and Pointers
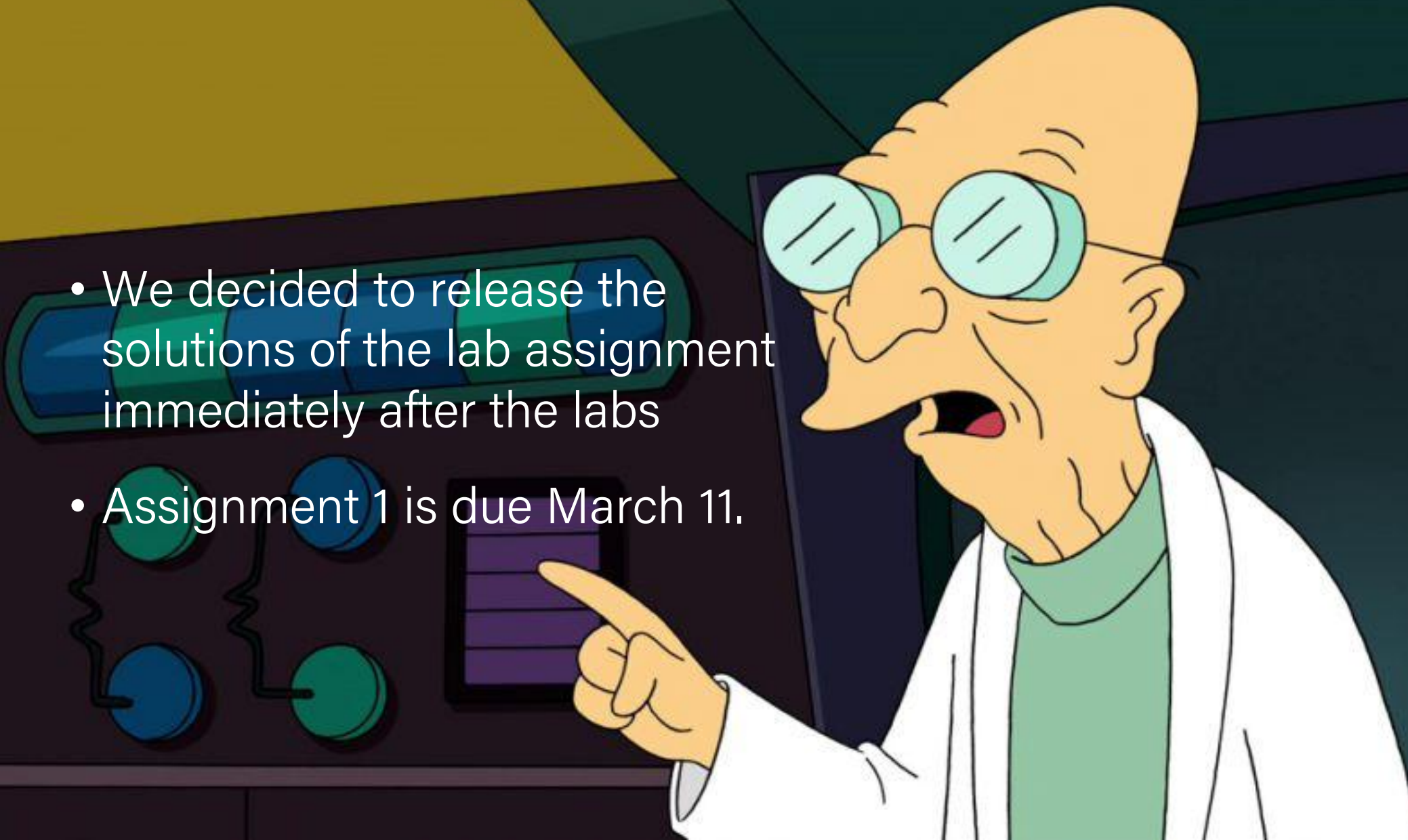
Aykut Erdem // Koç University // Spring 2021

KOÇ
UNIVERSITY

Good news, everyone!

- We decided to release the solutions of the lab assignment immediately after the labs

- Assignment 1 is due March 11.

COMP201 Topic 4: How can we effectively manage all types of memory in our programs?

Plan for Today

- Pointers and Parameters

- Double Pointers

- Arrays in Memory

- Arrays of Pointers

- Pointer Arithmetic

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli and Lisa Yan's Stanford CS107 class

Lecture Plan

- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Recap: 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.

Recap: 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'
	…

Recap: 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'
	…

Recap: 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
```

Recap: Pointers

- 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.

Recap: Pointers

- **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!
}
```

Recap: 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

Address	Value
	…
x 0x1f0	2
	…

main()

Recap: 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

Address	Value
	…
x 0x1f0	2
	…

main()

Recap: 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!
    ...
}
```
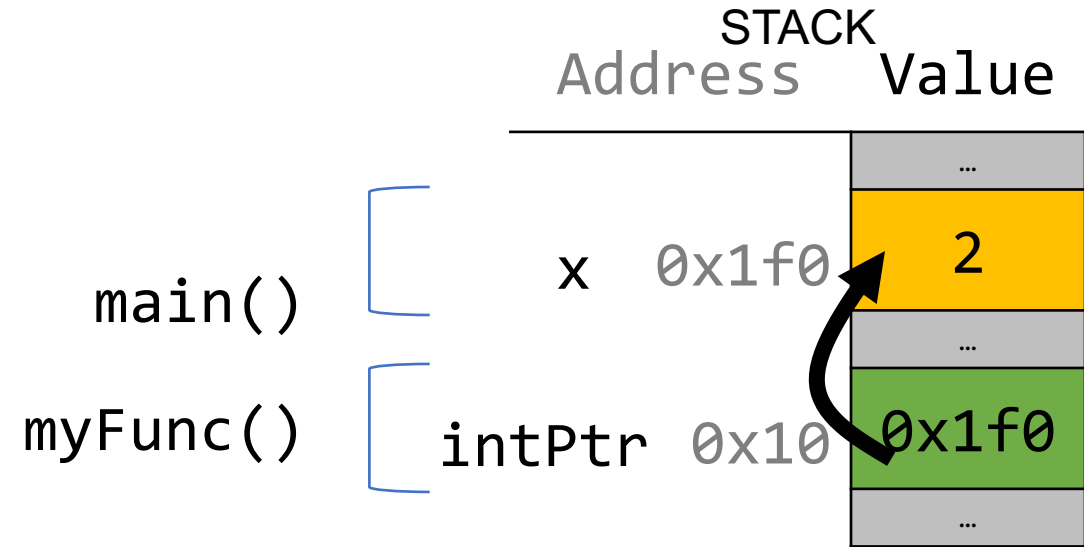
STACK

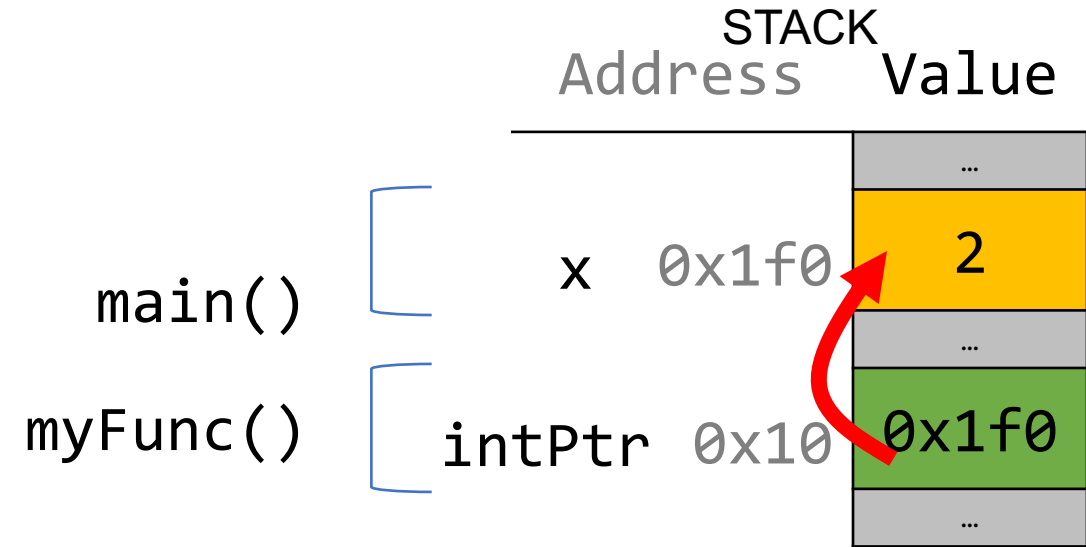| Address	Value
	…

main()

x 0x1f0   2

myFunc()

intPtr 0x10   0x1f0

Recap: 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

Address	Value
	…

main()

x 0x1f0   2

myFunc()

intPtr 0x10   0x1f0

… …

Recap: 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

Address	Value
	…
	3
x 0x1f0	
	…
intPtr 0x10	0x1f0
	…

main()

myFunc()

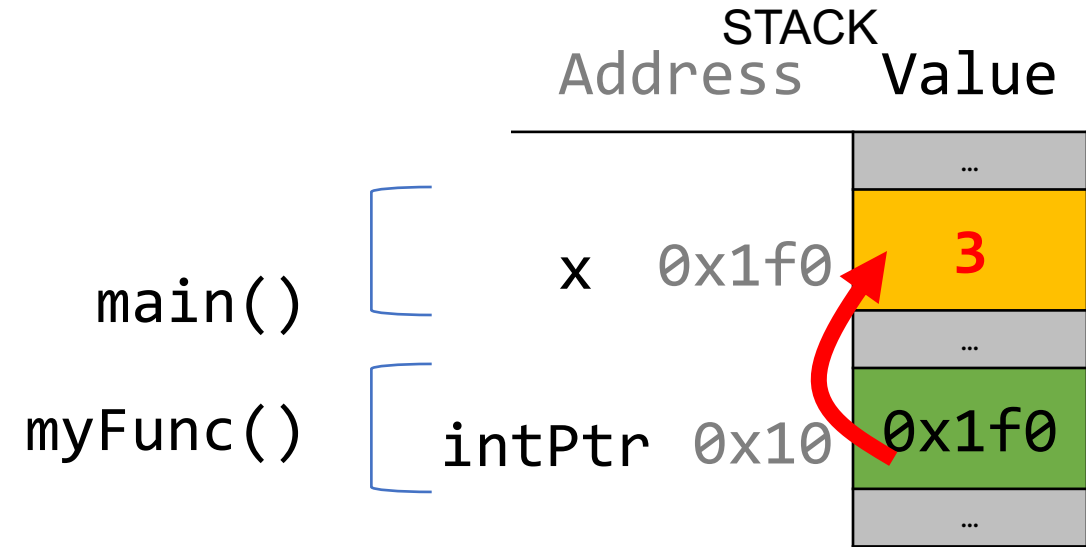# Recap: Pointers

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

Address	Value
	…
x 0x1f0	3
	…

main()

Recap: 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

Address	Value
	…
x 0x1f0	3
	…

main()

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.

```c
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.

```c
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.

C Parameters

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

```c
void myFunction(int x) {
    …
}


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

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.

```c
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.

```c
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;
}
```
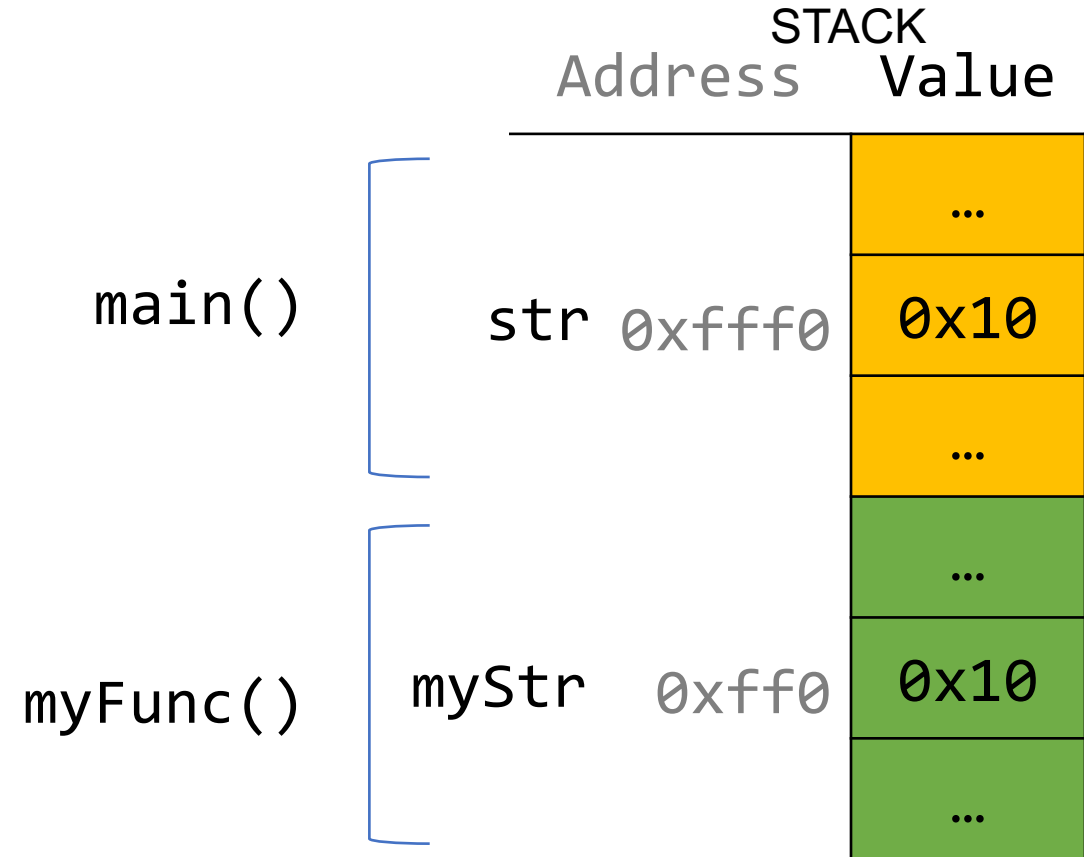
char *

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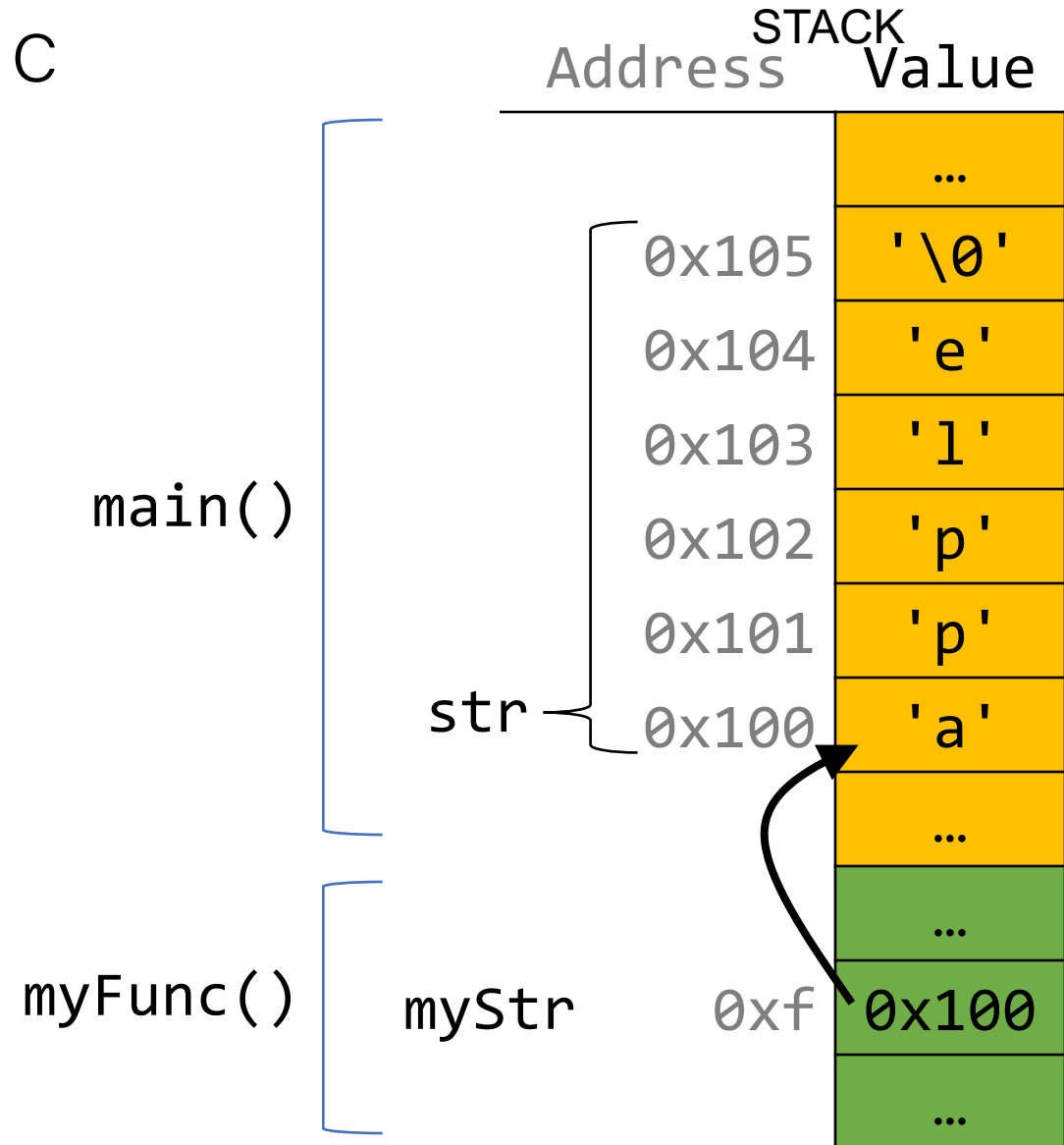
```
void myFunc(char *myStr) {
    ...
}

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

STACK

	Address	Value

main()

str	0xfff0	...
		0x10
		...

myFunc()

myStr	0xff0	...
		0x10
		...

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.

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

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

STACK

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

main()

str

myFunc() myStr

Strings as Parameters

When we pass a `char array` as a parameter, C makes a *copy of the address of the first array element,* and passes it (as a `char *`) to the function.
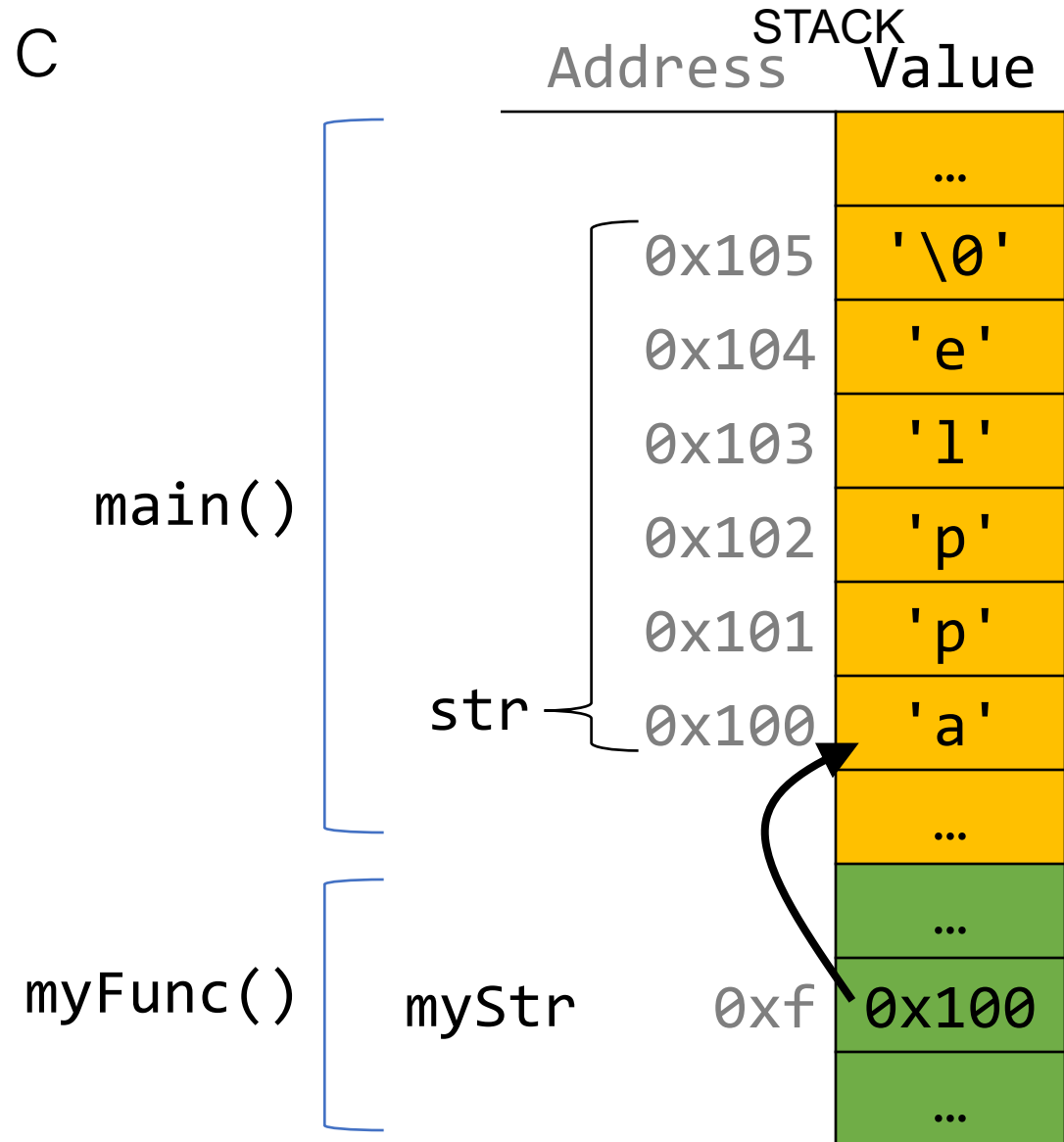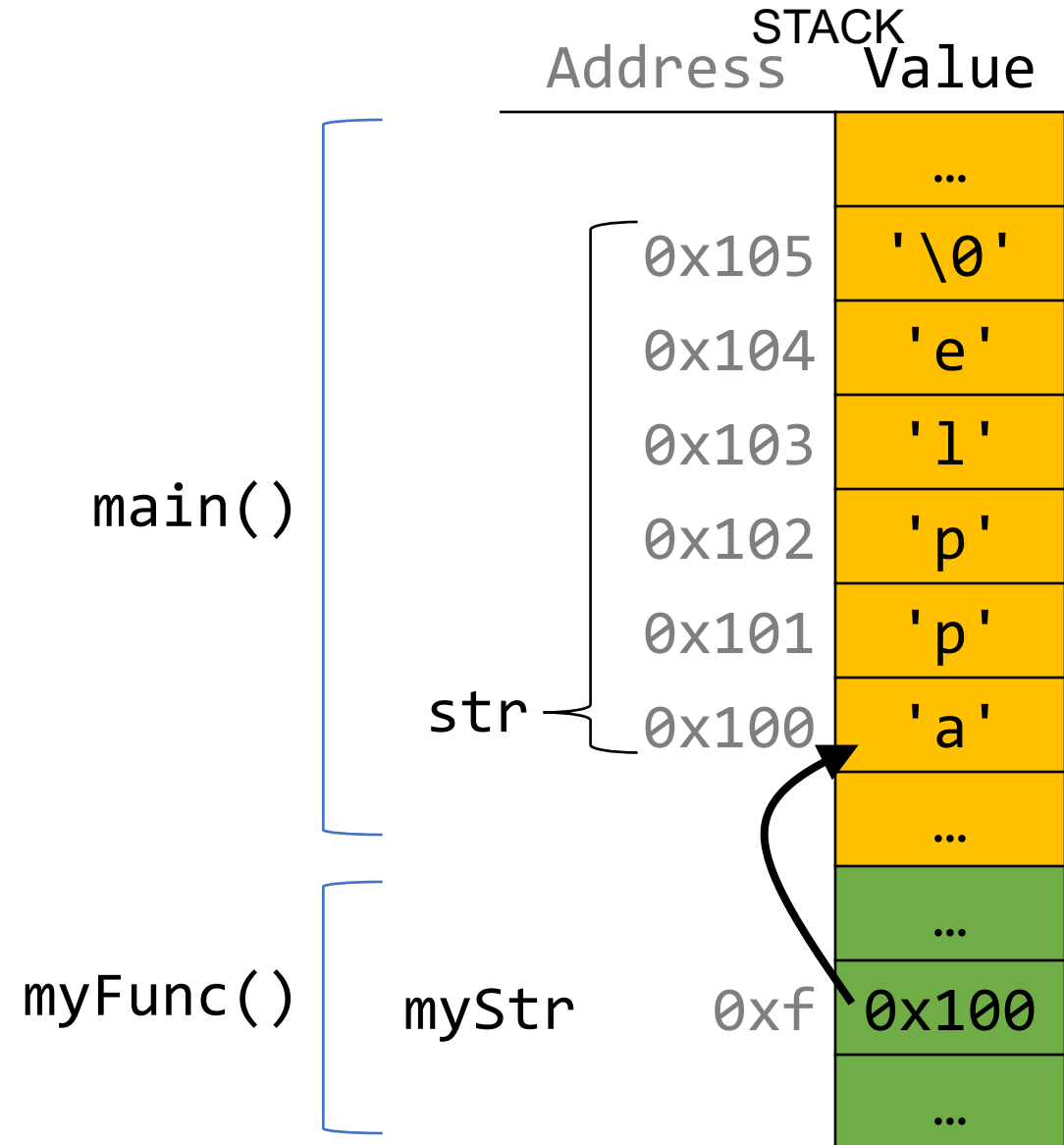
```c
void myFunc(char *myStr) {
    ...
}

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

STACK

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

main()

str

myFunc()   myStr

Strings as Parameters

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

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

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


STACK

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

main()

str

myFunc()   myStr   0xf   0x100

Strings as Parameters

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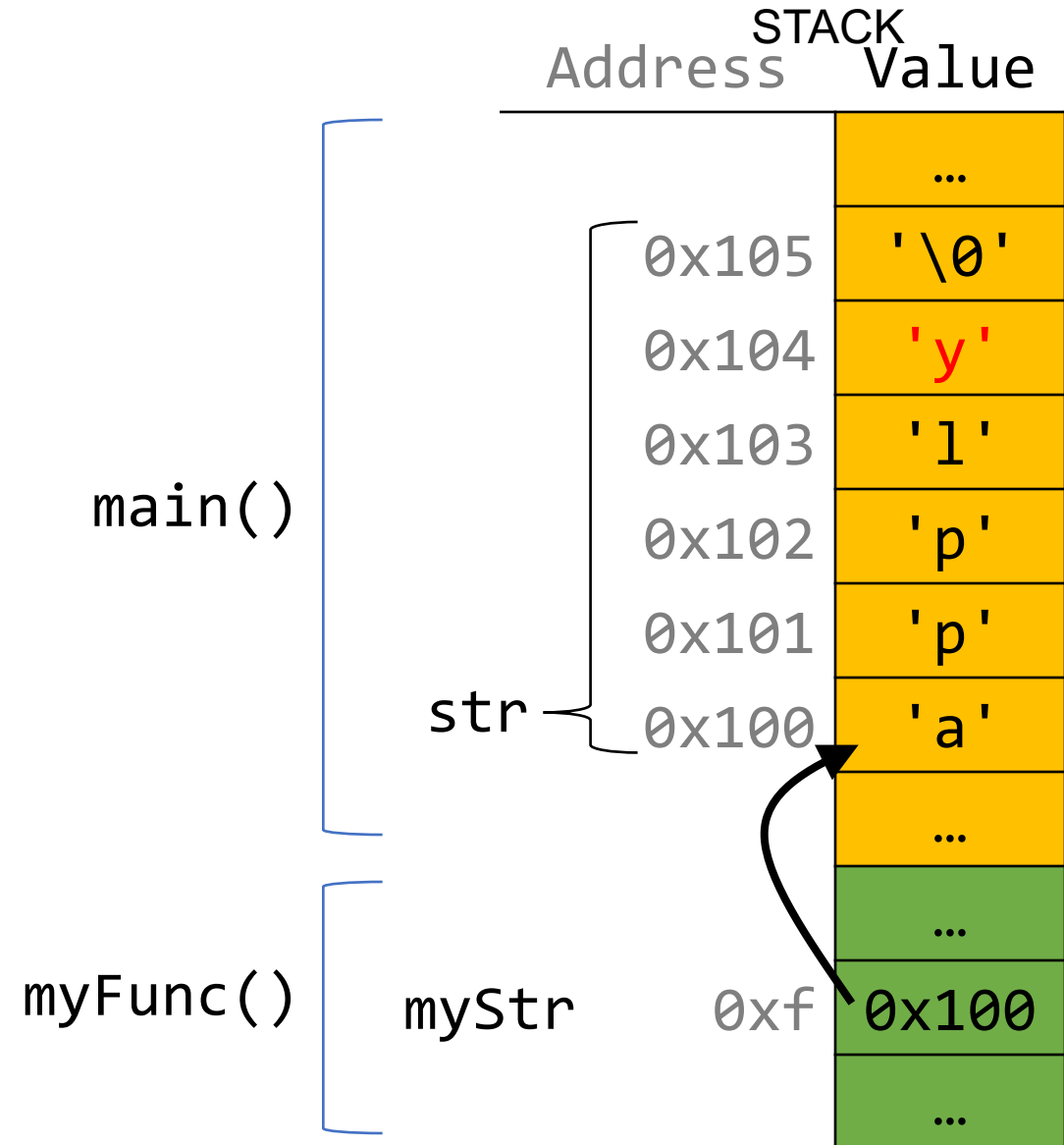
```
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

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

STACK

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

main()

str

myFunc()   myStr

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?

```c
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.

```c
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
- **Double Pointers**
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

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?

```c
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.
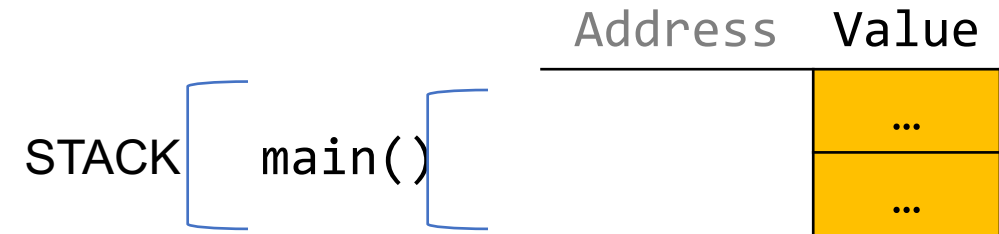
Demo: Skip Spaces

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;
}
```

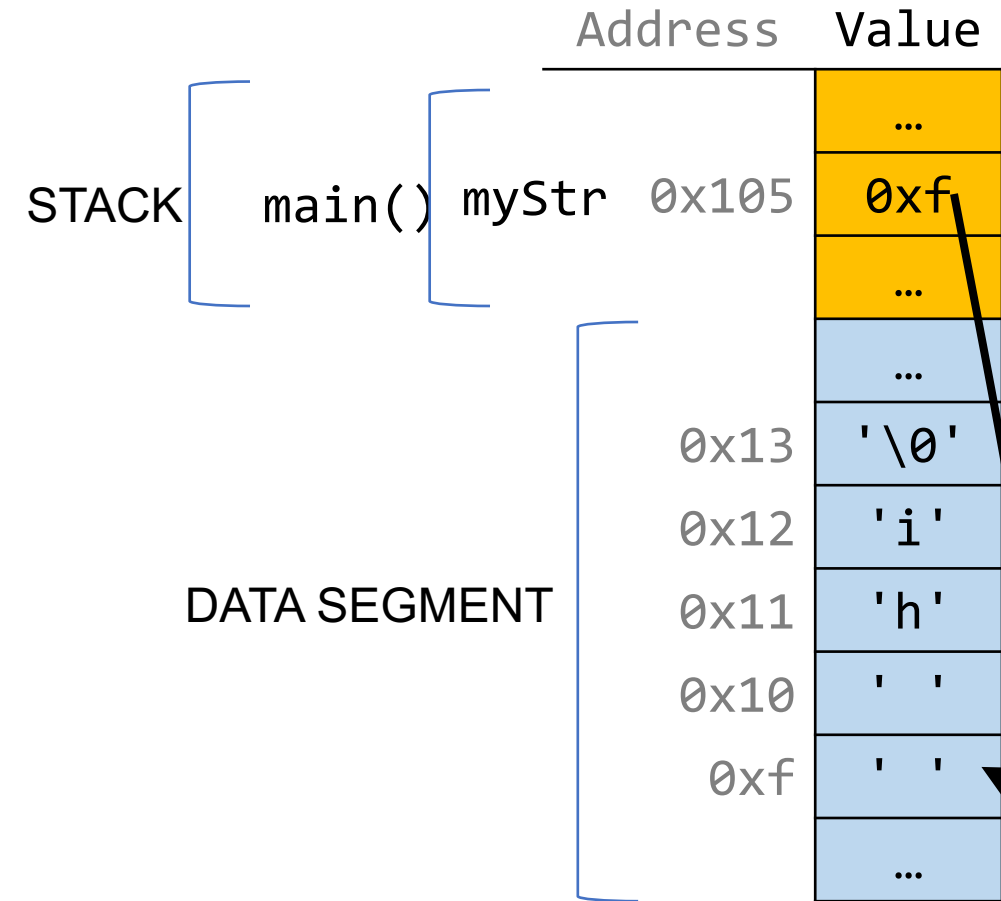Address   Value

STACK   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;
}
```

Address	Value
	…
0x105	0xf
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

STACK main() myStr
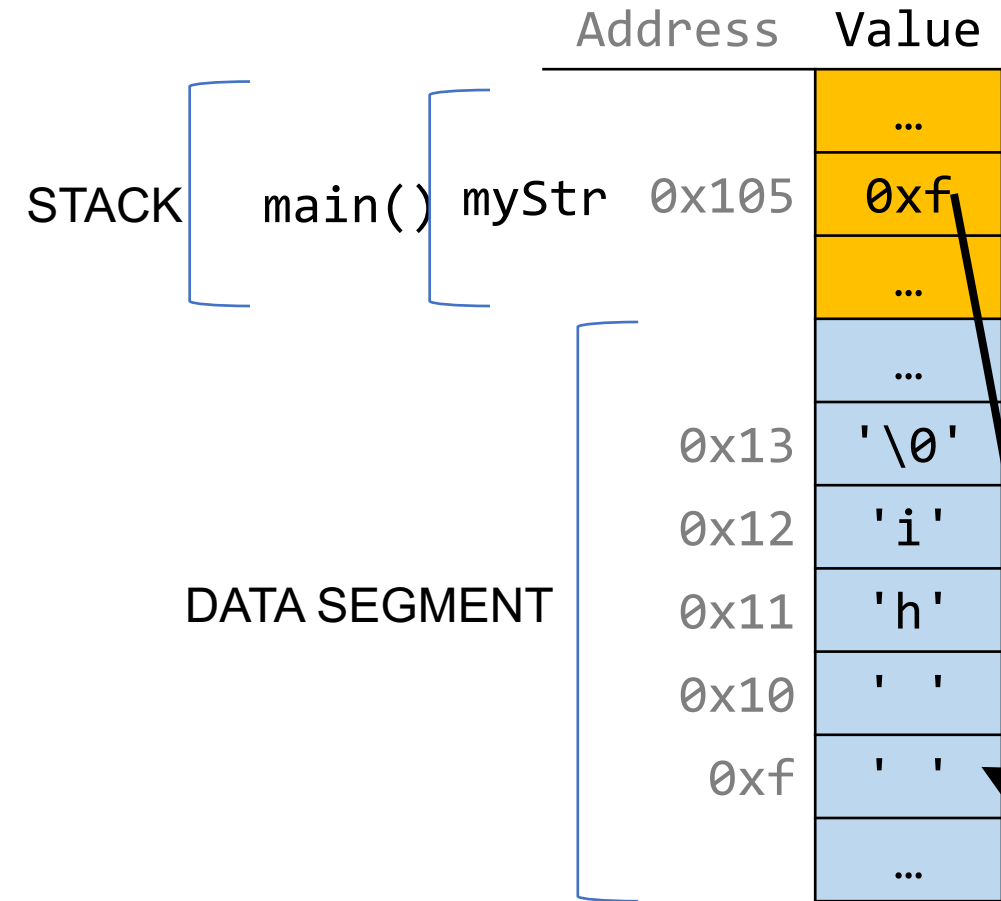
DATA SEGMENT

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;
}
```

Address	Value
	…
0x105	0xf
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

STACK   main() myStr

DATA SEGMENT

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;
}
```


Address   Value

main()   myStr 0x105   0xf

STACK

skipSpaces()   strPtr 0xf0   0x105

DATA SEGMENT

0x13   '\0'
0x12   'i'
0x11   'h'
0x10   ' '
0xf   ' '

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;
}
```

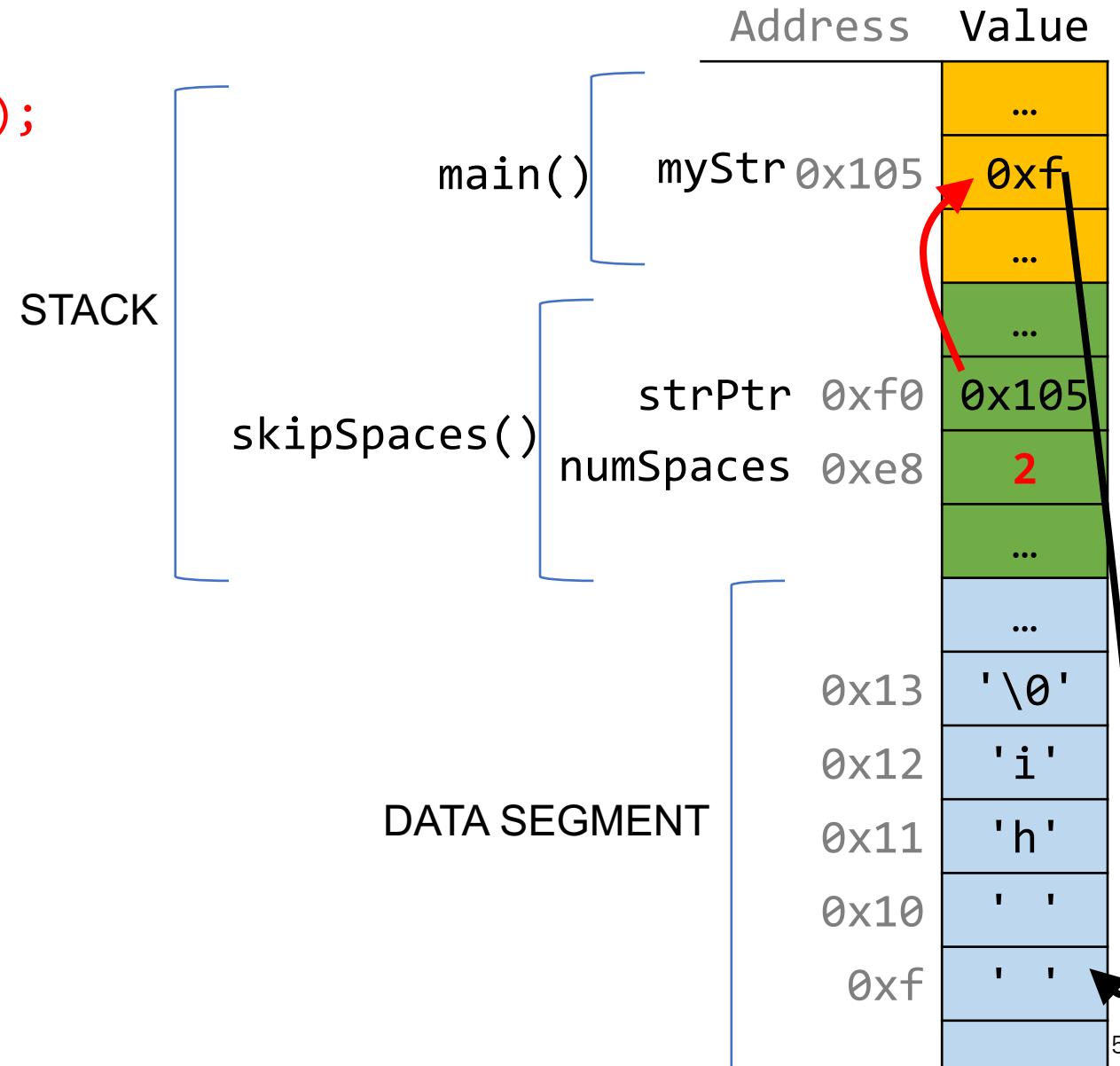| Address	Value
	…
myStr 0x105	0xf
	…
	…
strPtr 0xf0	0x105
numSpaces 0xe8	2
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '

main()

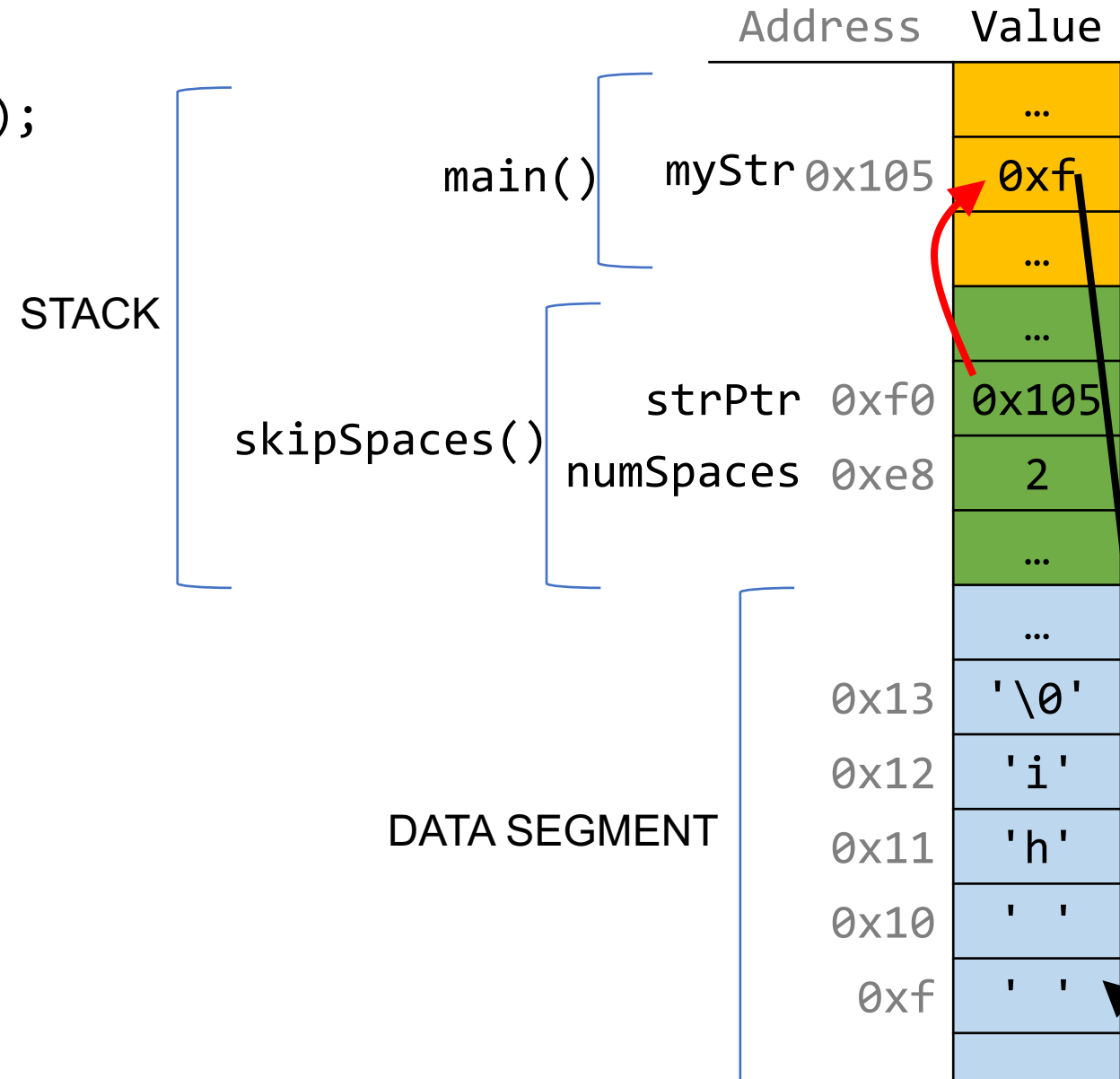skipSpaces()

STACK

DATA SEGMENT

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;
}
```

Address   Value

main()   myStr 0x105   0xf
          …

STACK

skipSpaces()   strPtr 0xf0   0x105
                numSpaces 0xe8   2
                …

DATA SEGMENT

…
0x13   '\0'
0x12   'i'
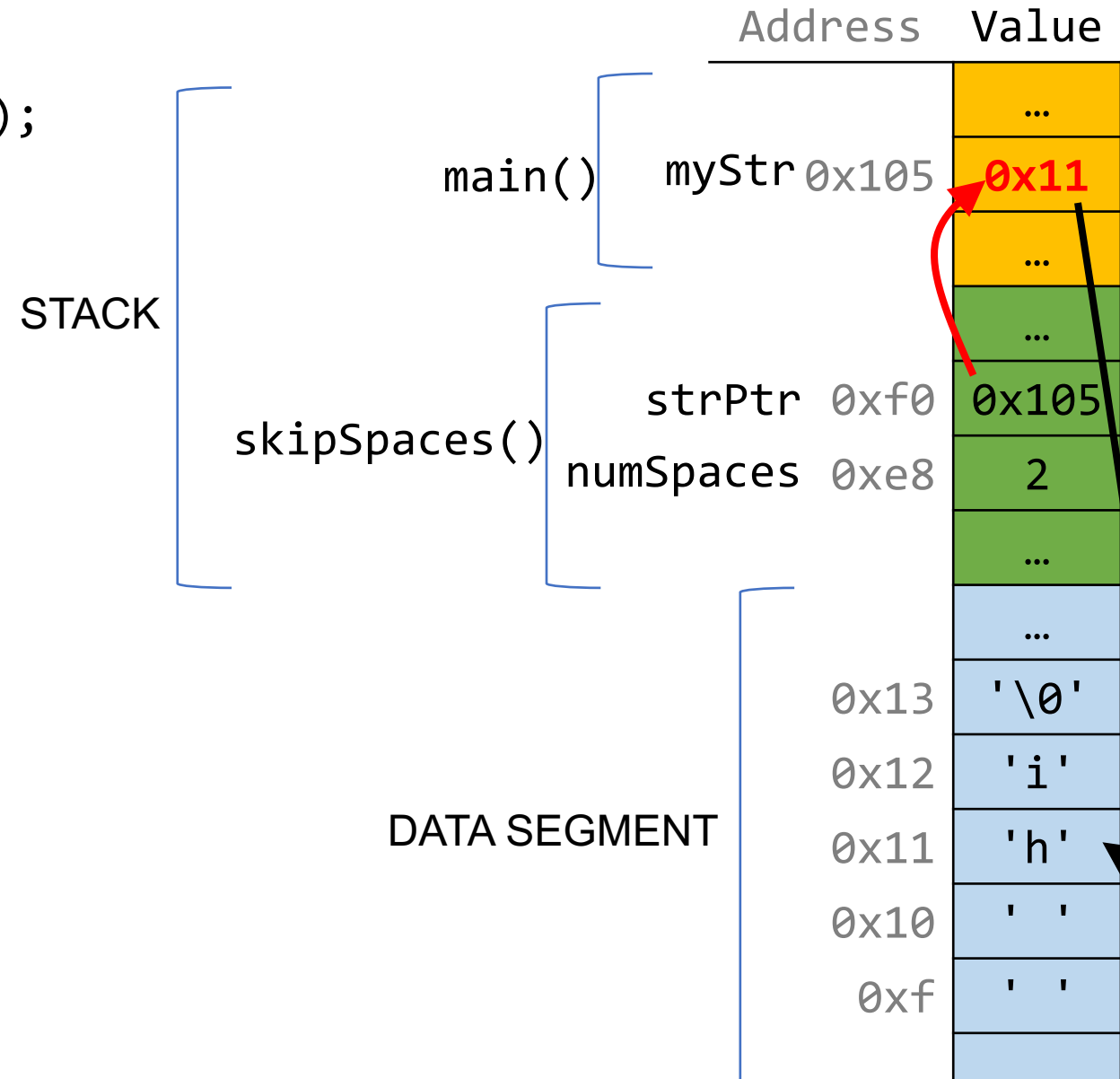0x11   'h'
0x10   ' '
0xf   ' '

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;
}
```

Address   Value

STACK

main()   myStr 0x105   0x11

…

skipSpaces()   strPtr 0xf0   0x105
                numSpaces 0xe8   2

DATA SEGMENT

0x13   '\0'
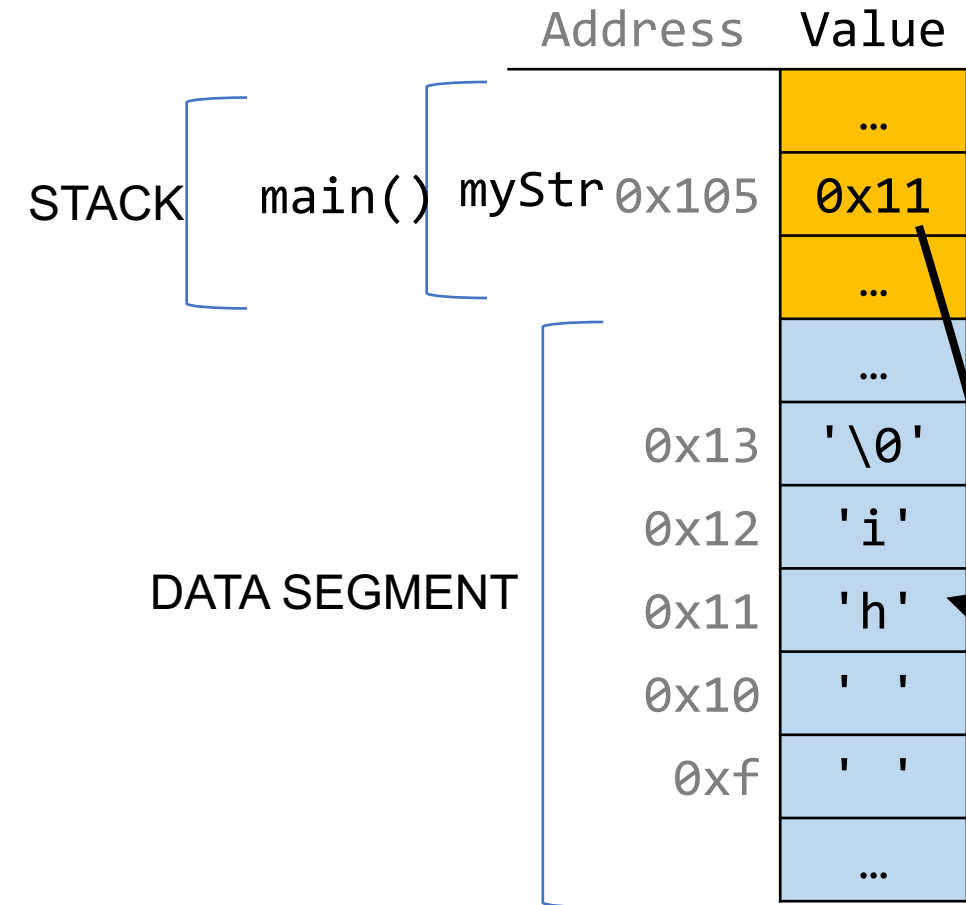0x12   'i'
0x11   'h'
0x10   ' '
0xf   ' '

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;
}
```

Address	Value
	…
0x105	0x11
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

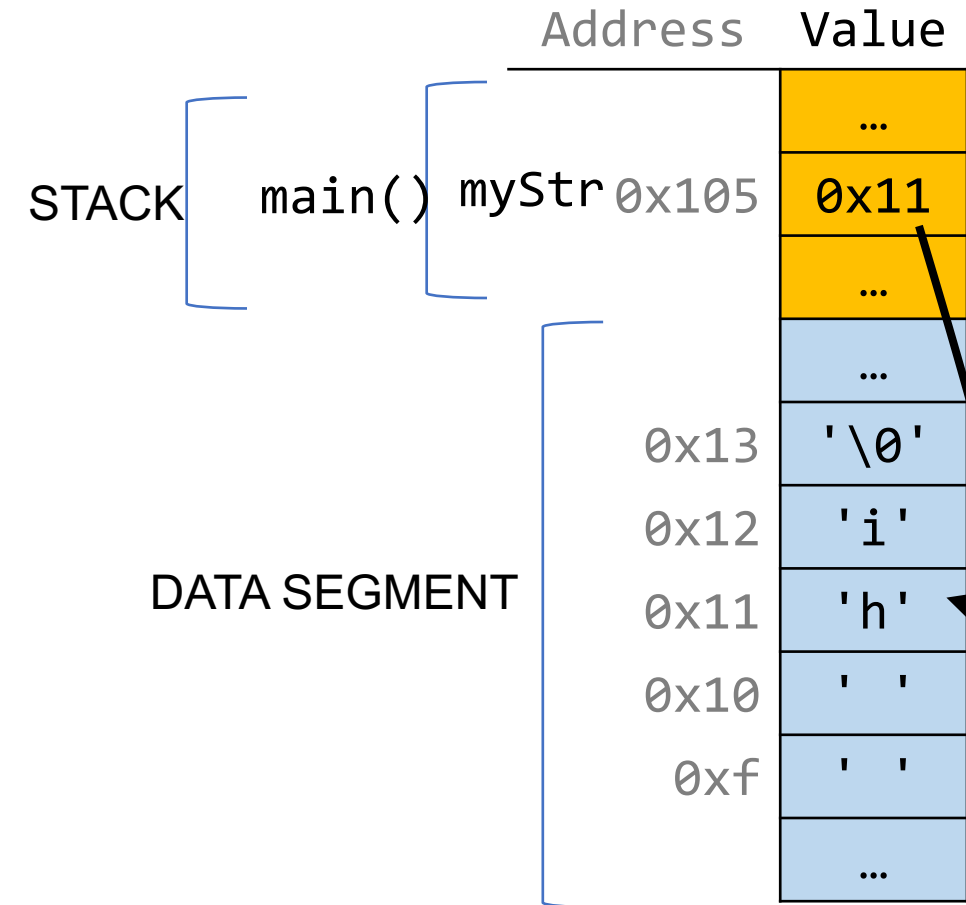STACK   main() myStr

DATA SEGMENT

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;
}
```

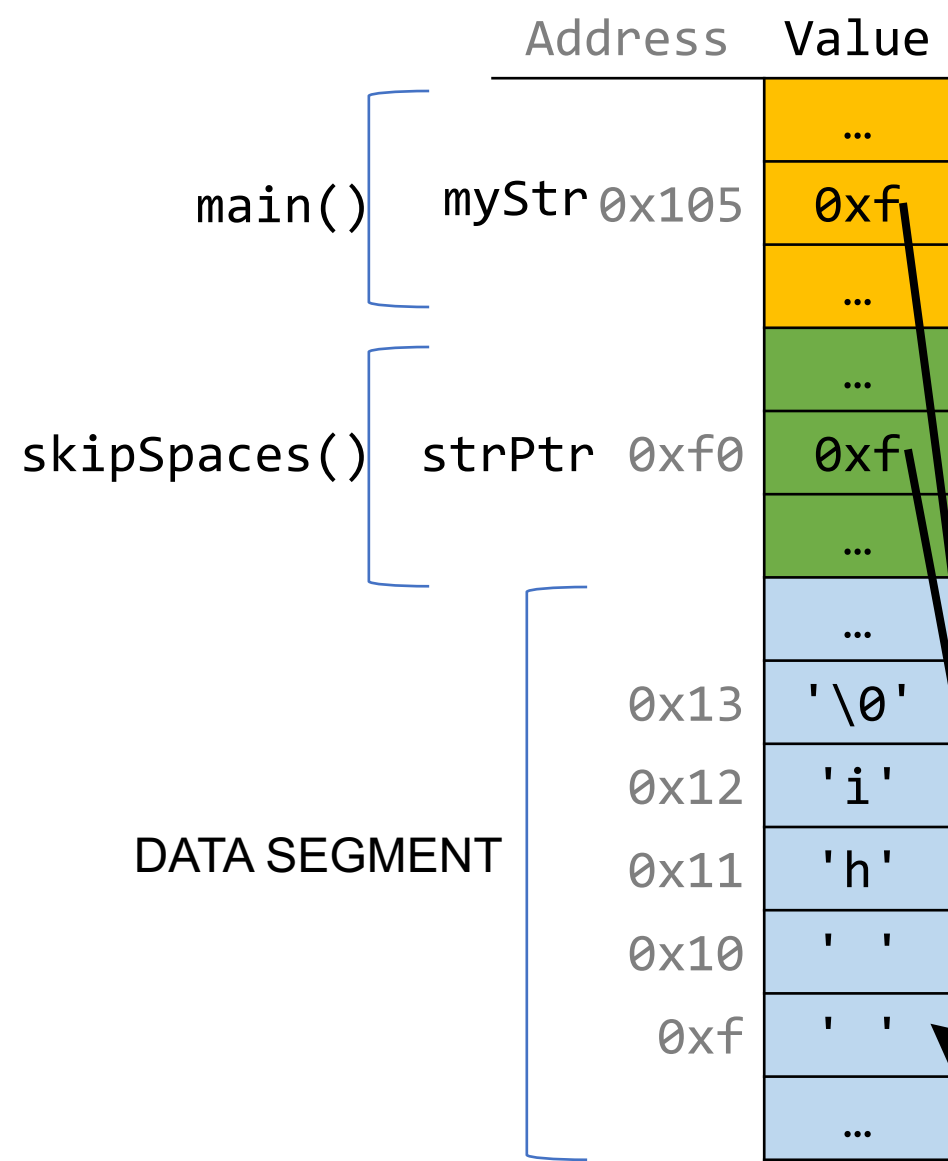| Address	Value
	…
0x105	0x11
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

STACK   main() myStr

DATA SEGMENT

Making Copies

```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);
    return 0;
}
```

myFunc myFunc
//   hi

Address	Value
	…
0x105	0xf
	…
	…
0xf0	0xf
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

main() myStr

STACK
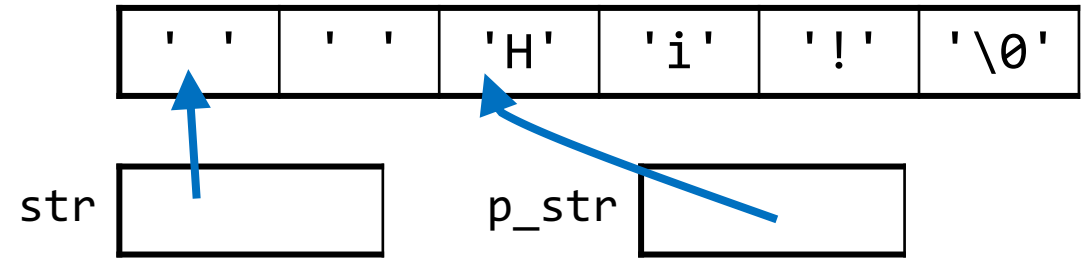
skipSpaces() strPtr

DATA SEGMENT

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`)?

A.

B.
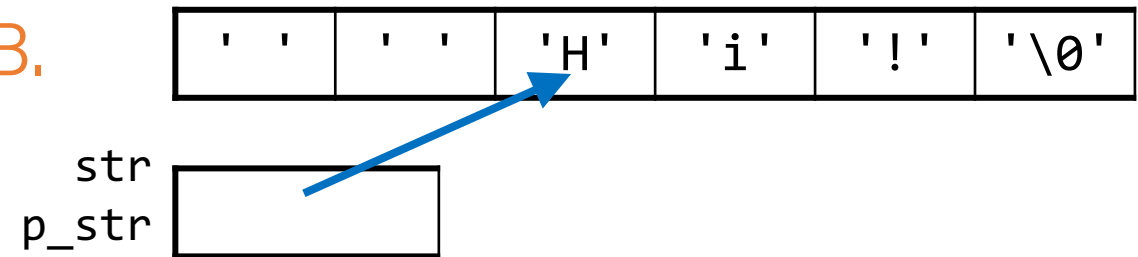
C. 🤔

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`)?
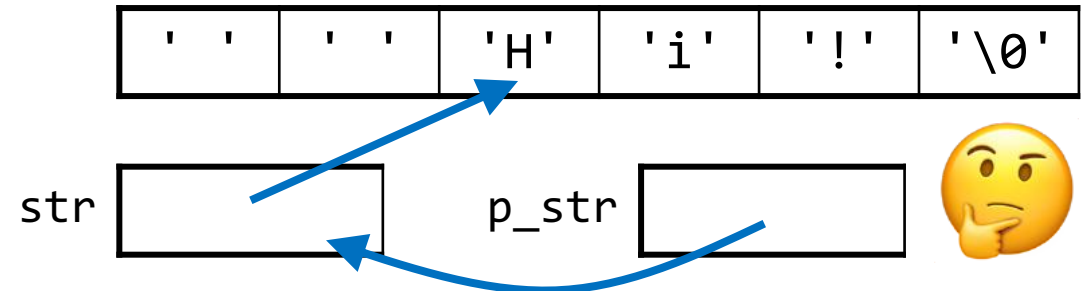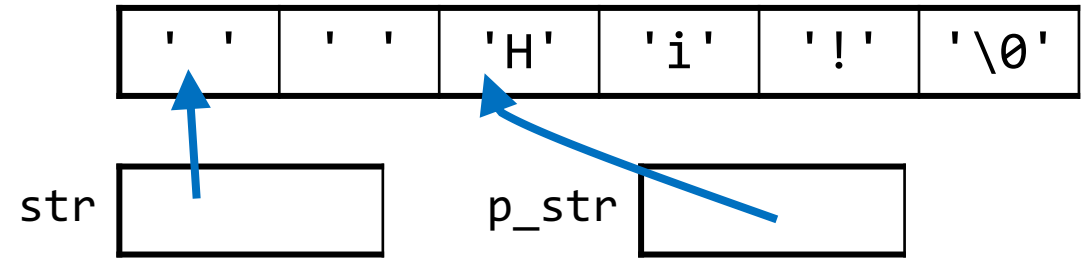
A.

| ' ' | ' ' | 'H' | 'i' | '!' | '\0' |

str

p_str

B.

| ' ' | ' ' | 'H' | 'i' | '!' | '\0' |

str
p_str

C.

| ' ' | ' ' | 'H' | 'i' | '!' | '\0' |

str

p_str
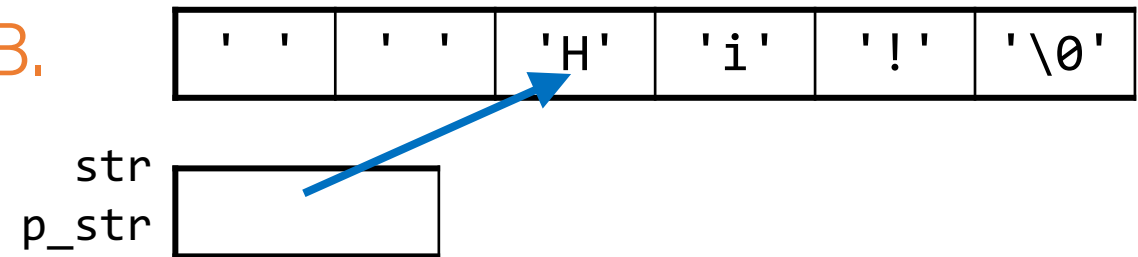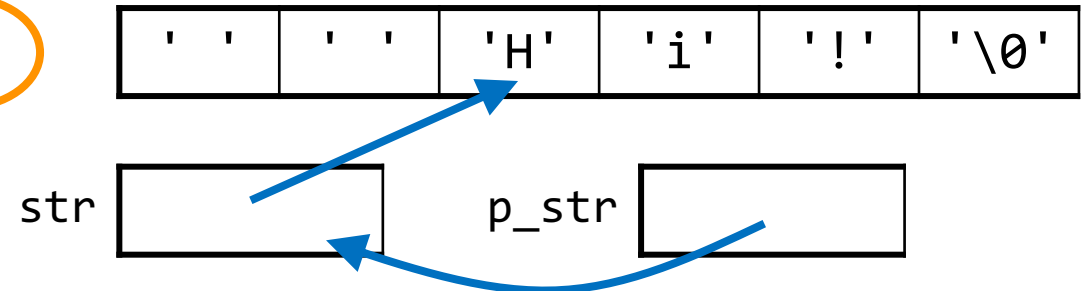
Lecture Plan

- Pointers and Parameters

- Double Pointers

- **Arrays in Memory**

- Arrays of Pointers

- Pointer Arithmetic

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
```

STACK

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

str
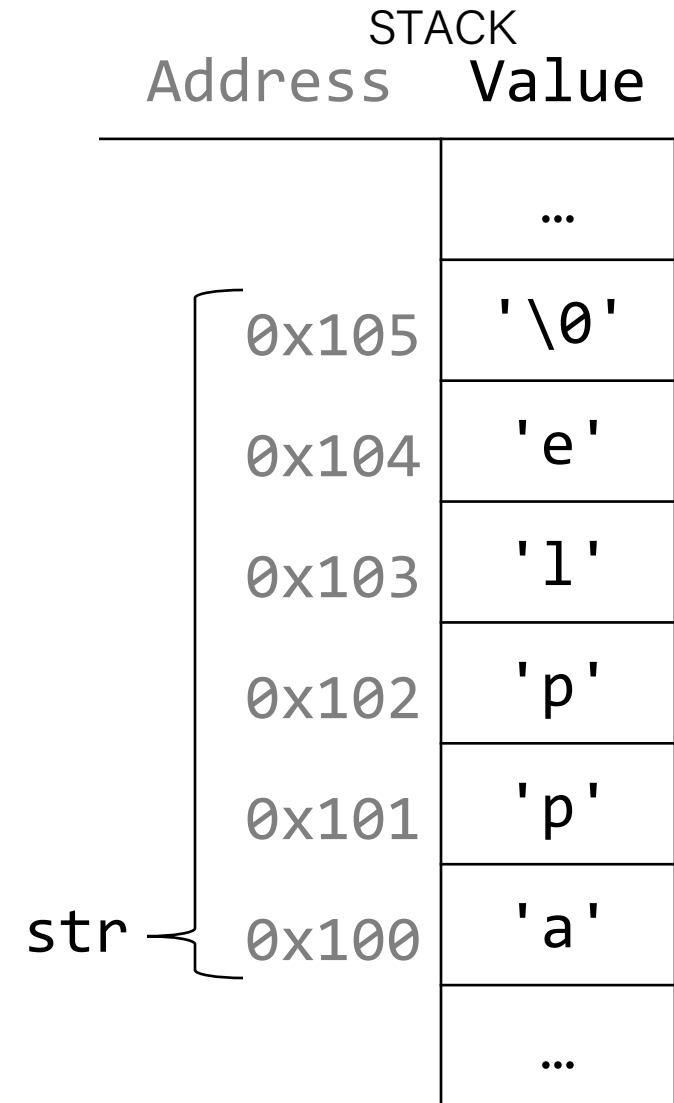
Arrays

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);
    ...
}
```
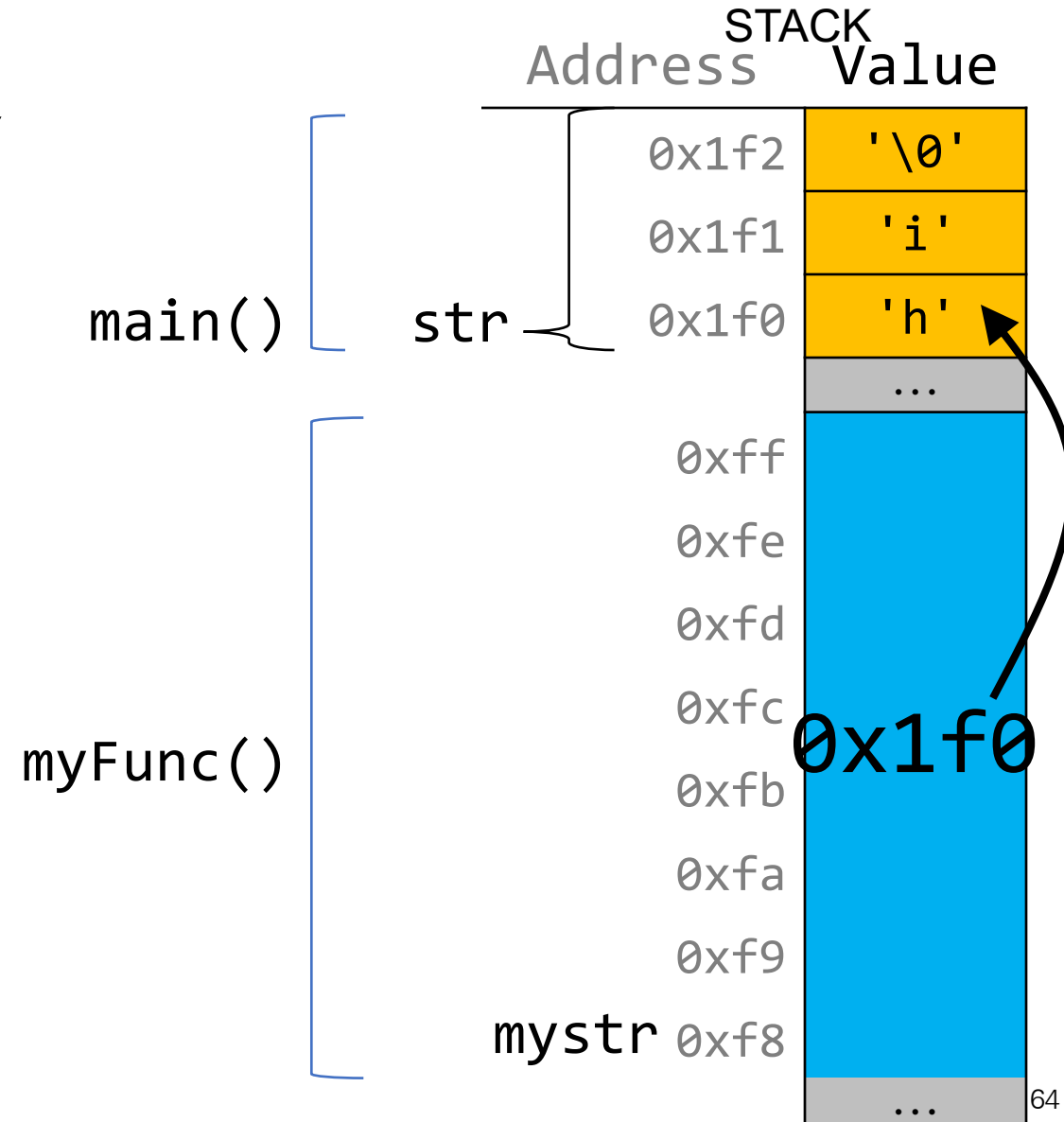
STACK

Address	Value
0x1f2	'\0'
0x1f1	'i'
0x1f0	'h'
...	

main()   str

0xff	
0xfe	
0xfd	
0xfc	
0xfb	0x1f0
0xfa	
0xf9	
0xf8	

myFunc()

mystr 0xf8

...

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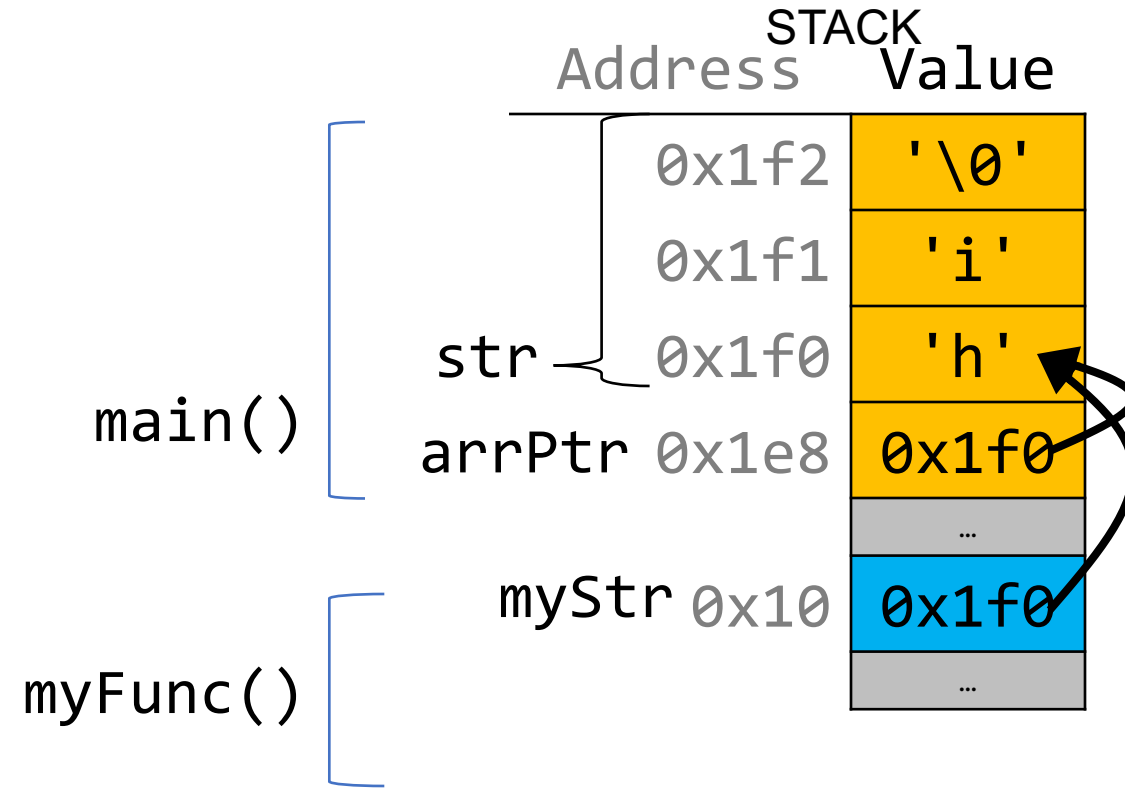
```
void myFunc(char *myStr) {
    …
}

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

STACK

Address   Value

	Address	Value
	0x1f2	'\0'
	0x1f1	'i'
str	0x1f0	'h'
arrPtr	0x1e8	0x1f0
		…
myStr	0x10	0x1f0
		…

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);
    ...
}
```

STACK

Address	Value
0x1f2	'\0'
0x1f1	'i'
0x1f0	'h'
	...
0xff	
0xfe	
0xfd	
0xfc	0x1f0
0xfb	
0xfa	
0xf9	
0xf8	

main()   str

myFunc()

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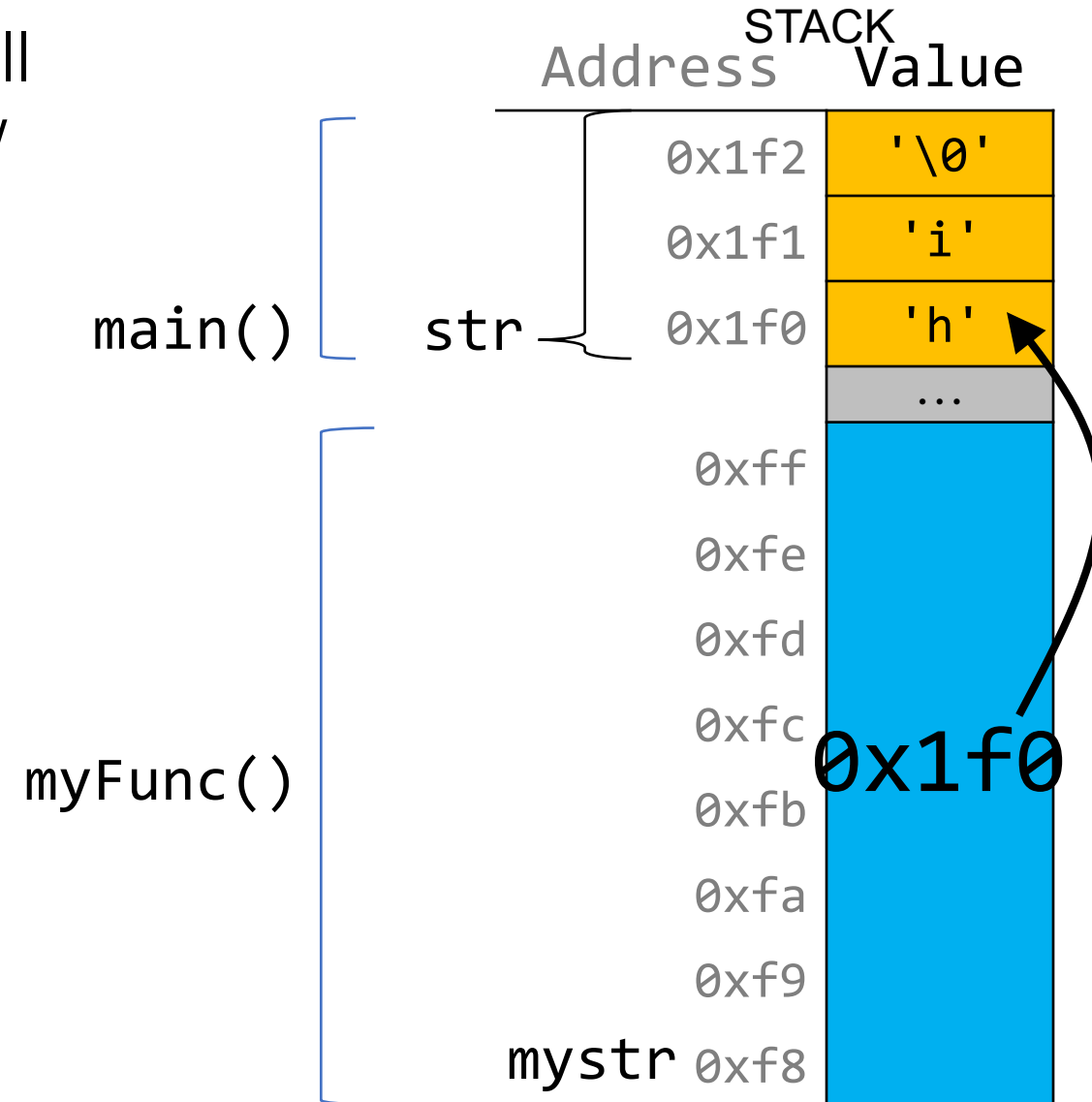
sizeof returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **sizeof** to get its full size.

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()

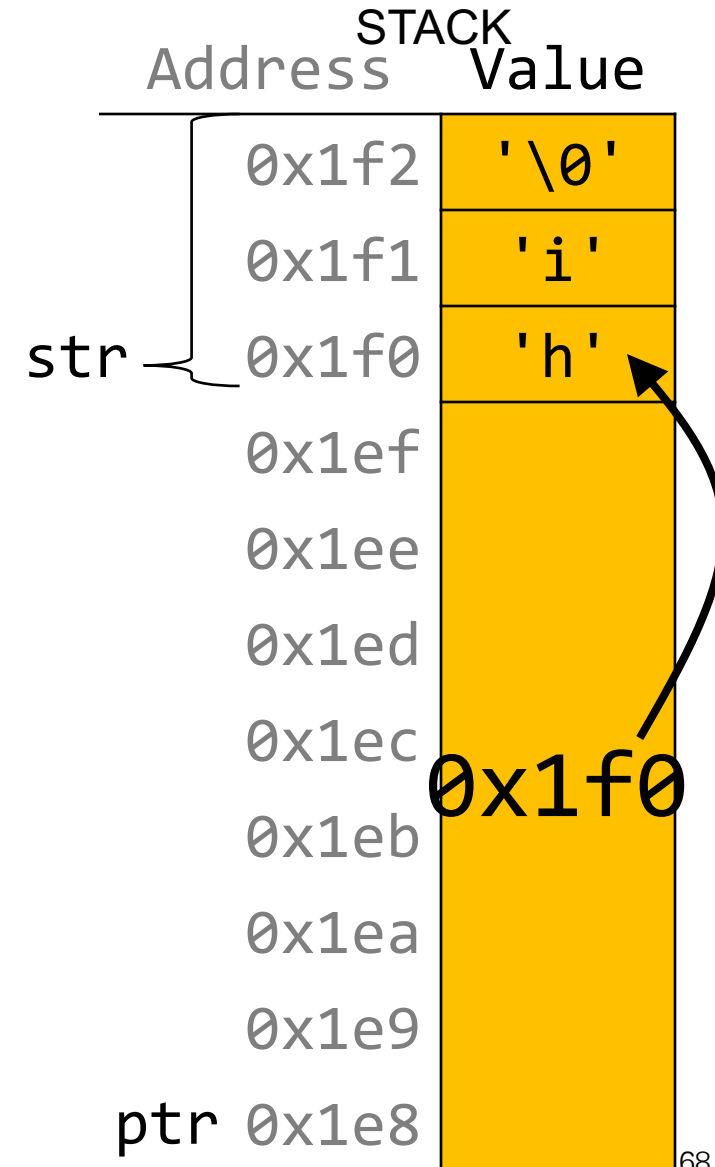| Address	Value
0x1f2	'\0'
0x1f1	'i'
str 0x1f0	'h'
0x1ef	
0x1ee	
0x1ed	
0x1ec	0x1f0
0x1eb	
0x1ea	
0x1e9	
ptr 0x1e8	

68

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;
    ...
}
```
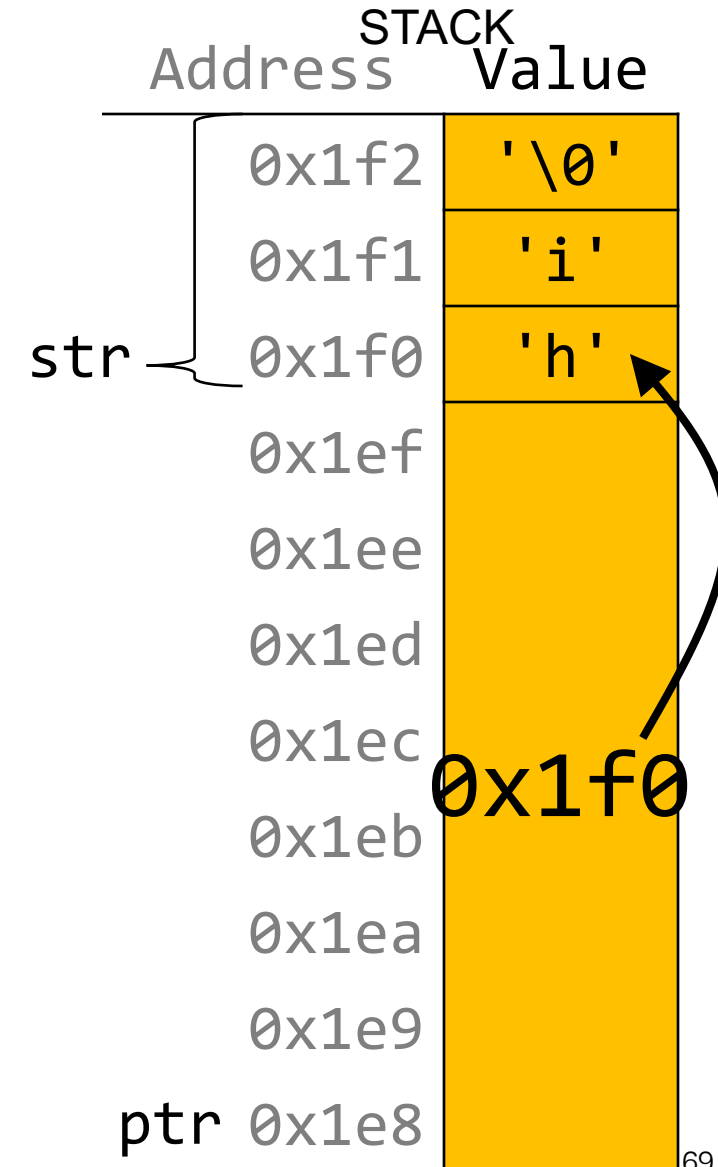
STACK

Address	Value
0x1f2	'\0'
0x1f1	'i'
str 0x1f0	'h'
0x1ef	
0x1ee	
0x1ed	
0x1ec	
0x1eb	0x1f0
0x1ea	
0x1e9	
ptr 0x1e8	

main()

Lecture Plan

• Pointers and Parameters

• Double Pointers

• Arrays in Memory

• Arrays of Pointers

• Pointer Arithmetic

Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:
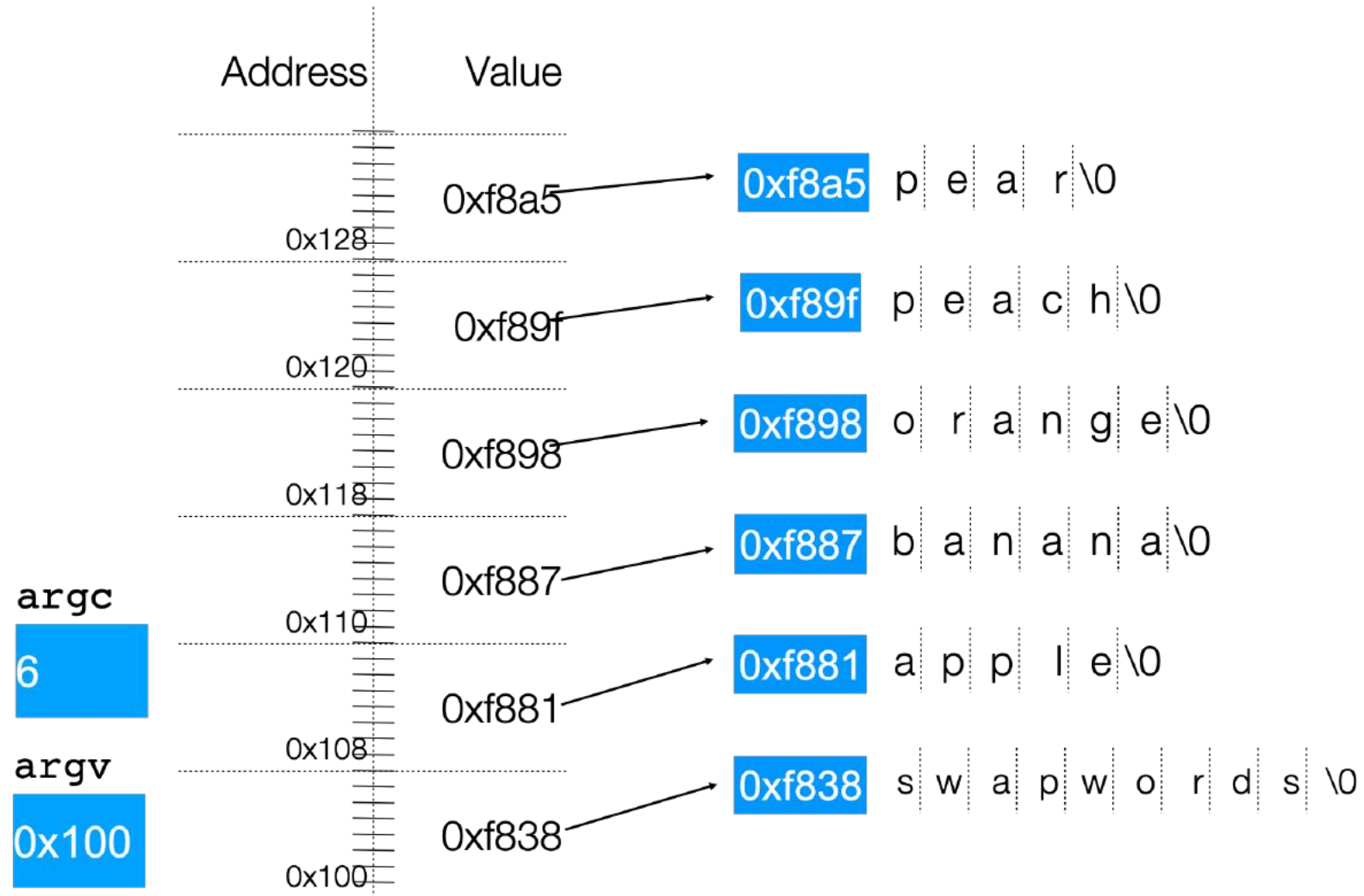
```
char *stringArray[5];   // space to store 5 char *s
```

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
```

Address | Value

0xf8a5 → 0xf8a5 | p | e | a | r | \0

0x128

0xf89f → 0xf89f | p | e | a | c | h | \0

0x120

0xf898 → 0xf898 | o | r | a | n | g | e | \0

0x118

0xf887 → 0xf887 | b | a | n | a | n | a | \0

argc
6

0x110

0xf881 → 0xf881 | a | p | p | l | e | \0

0x108

argv
0x100

0xf838 → 0xf838 | s | w | a | p | w | o | r | d | s | \0

0x100

Arrays Of Pointers

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

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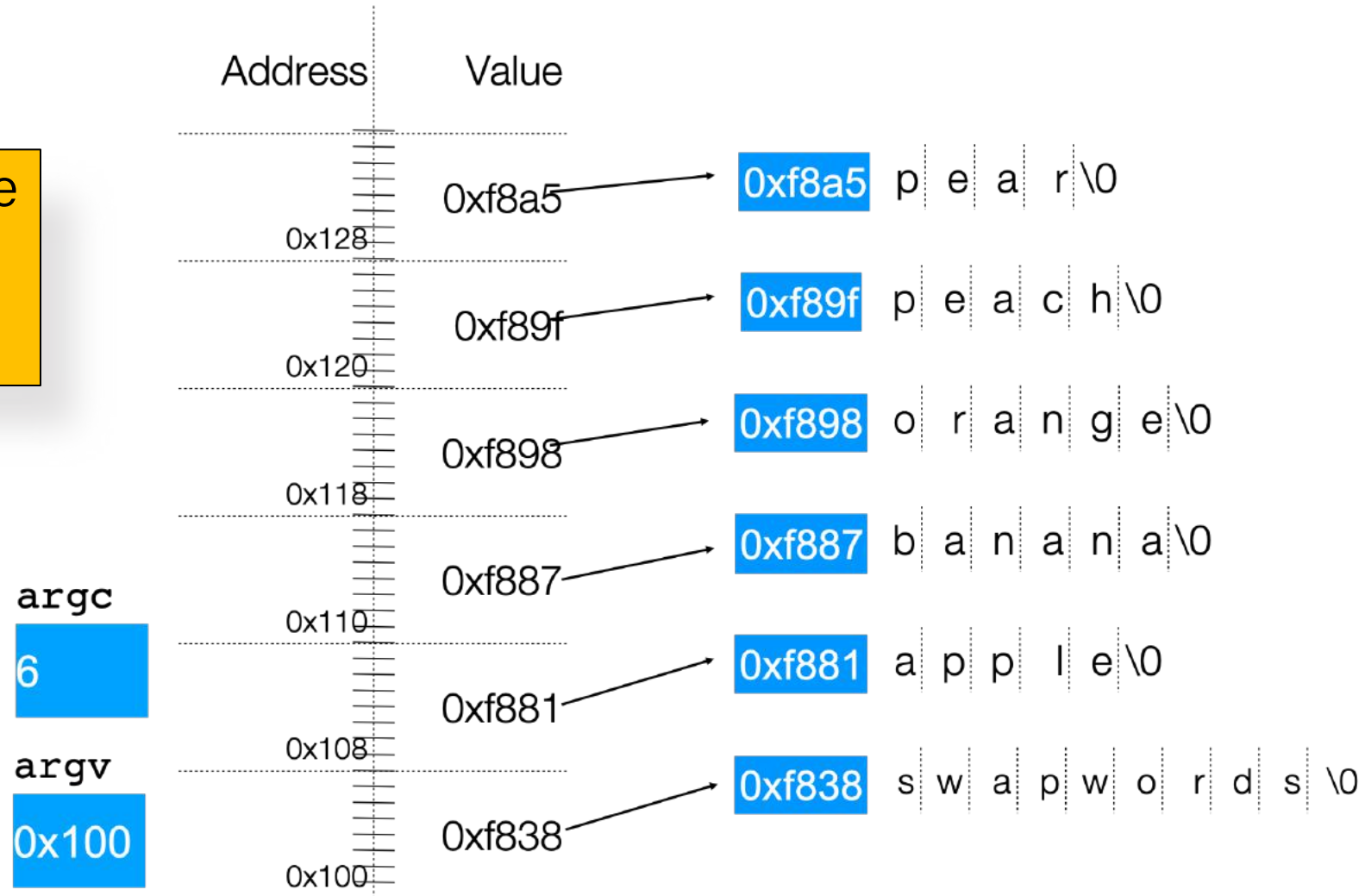
Address   Value

0xf8a5 → 0xf8a5 | p | e | a | r | \0

0x128

0xf89f → 0xf89f | p | e | a | c | h | \0

0x120

0xf898 → 0xf898 | o | r | a | n | g | e | \0

0x118

0xf887 → 0xf887 | b | a | n | a | n | a | \0

argc

0x110

6

0xf881 → 0xf881 | a | p | p | l | e | \0

0x108

argv

0xf838 → 0xf838 | s | w | a | p | w | o | r | d | s | \0

0x100

0x100

Lecture Plan

- Pointers and Parameters

- Double Pointers

- Arrays in Memory

- Arrays of Pointers

- **Pointer Arithmetic**

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

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, "COMP201");


// 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.

Pointer arithmetic

Array indexing is "syntactic sugar" for pointer arithmetic:

$$\texttt{ptr + i} \qquad \Leftrightarrow \qquad \texttt{\&ptr[i]}$$

$$\texttt{*(ptr + i)} \qquad \Leftrightarrow \qquad \texttt{ptr[i]}$$

⚠️ 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]`

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

Recap

- Pointers and Parameters

- Double Pointers

- Arrays in Memory

- Arrays of Pointers

- Pointer Arithmetic

Next Time: pointer arithmetic, dynamically allocated memory