

# COMP201

## Computer Systems & Programming

### Lecture #7 – Arrays and Pointers



KOÇ  
UNIVERSITY

Aykut Erdem // Koç University // Fall 2024

# Recap: Common `string.h` Functions

Function	Description
<code>strlen(<i>str</i>)</code>	returns the # of chars in a C string (before null-terminating character).
<code>strcmp(<i>str1</i>, <i>str2</i>)</code> , <code>strncmp(<i>str1</i>, <i>str2</i>, <i>n</i>)</code>	compares two strings; returns 0 if identical, <0 if <b><i>str1</i></b> comes before <b><i>str2</i></b> in alphabet, >0 if <b><i>str1</i></b> comes after <b><i>str2</i></b> in alphabet. <b><i>strncmp</i></b> stops comparing after at most <b><i>n</i></b> characters.
<code>strchr(<i>str</i>, <i>ch</i>)</code> <code>strrchr(<i>str</i>, <i>ch</i>)</code>	character search: returns a pointer to the first occurrence of <b><i>ch</i></b> in <b><i>str</i></b> , or <b><i>NULL</i></b> if <b><i>ch</i></b> was not found in <b><i>str</i></b> . <code>strrchr</code> find the last occurrence.
<code>strstr(<i>haystack</i>, <i>needle</i>)</code>	string search: returns a pointer to the start of the first occurrence of <b><i>needle</i></b> in <b><i>haystack</i></b> , or <b><i>NULL</i></b> if <b><i>needle</i></b> was not found in <b><i>haystack</i></b> .
<code>strcpy(<i>dst</i>, <i>src</i>)</code> , <code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	copies characters in <b><i>src</i></b> to <b><i>dst</i></b> , including null-terminating character. Assumes enough space in <b><i>dst</i></b> . Strings must not overlap. <b><i>strncpy</i></b> stops after at most <b><i>n</i></b> chars, and <u>does not</u> add null-terminating char.
<code>strcat(<i>dst</i>, <i>src</i>)</code> , <code>strncat(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	concatenate <b><i>src</i></b> onto the end of <b><i>dst</i></b> . <b><i>strncat</i></b> stops concatenating after at most <b><i>n</i></b> characters. <u>Always</u> adds a null-terminating character.
<code>strspn(<i>str</i>, <i>accept</i>)</code> , <code>strcspn(<i>str</i>, <i>reject</i>)</code>	<b><i>strspn</i></b> returns the length of the initial part of <b><i>str</i></b> which contains <u>only</u> characters in <b><i>accept</i></b> . <b><i>strcspn</i></b> returns the length of the initial part of <b><i>str</i></b> which does <u>not</u> contain any characters in <b><i>reject</i></b> .



# 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: 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: 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");
```

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

str {

# Recap: 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 *`

# Recap: 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.

# Plan for Today

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory

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



# Lecture Plan

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory

# char \*

There is an important difference between the following two definitions:

```
char aString[] = "Hello, world!";    // an array  
char *pString = "Hello, world!";    // a pointer
```

- `aString` is an array, just big enough to hold the sequence of characters and also the NULL terminating symbol at the end.
- `pString` is a pointer, initialized to point to a string constant. Note the the pointer may be modified to point to a different location.

# 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) {  
    ...  
}
```

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

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

# Memory Locations

**Q:** Is there a way to check in code whether a string's characters are modifiable?

**A:** No. This is something you can only tell by looking at the code itself and how the string was created.

**Q:** So then if I am writing a string function that modifies a string, how can I tell if the string passed in is modifiable?

**A:** You can't! This is something you instead state as an assumption in your function documentation. If someone calls your function with a read-only string, it will crash, but that's not your function's fault :-)

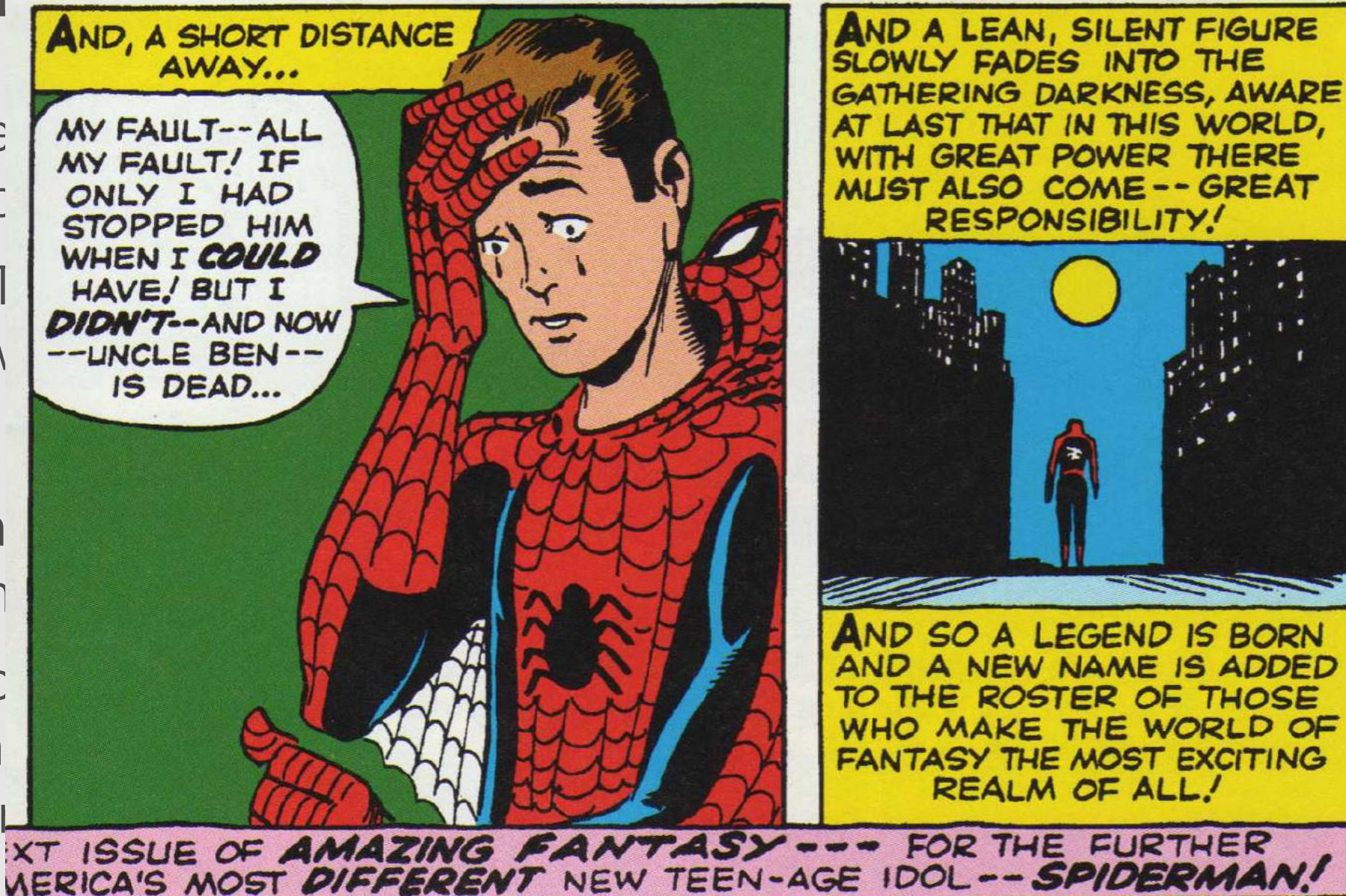
# Memory Locations

Q: Is the  
modifiak

A: No. 1  
and how

Q: So th  
I tell if th

A: You c  
your fun  
only stri



re

itself

low can

on in  
a read-

**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.

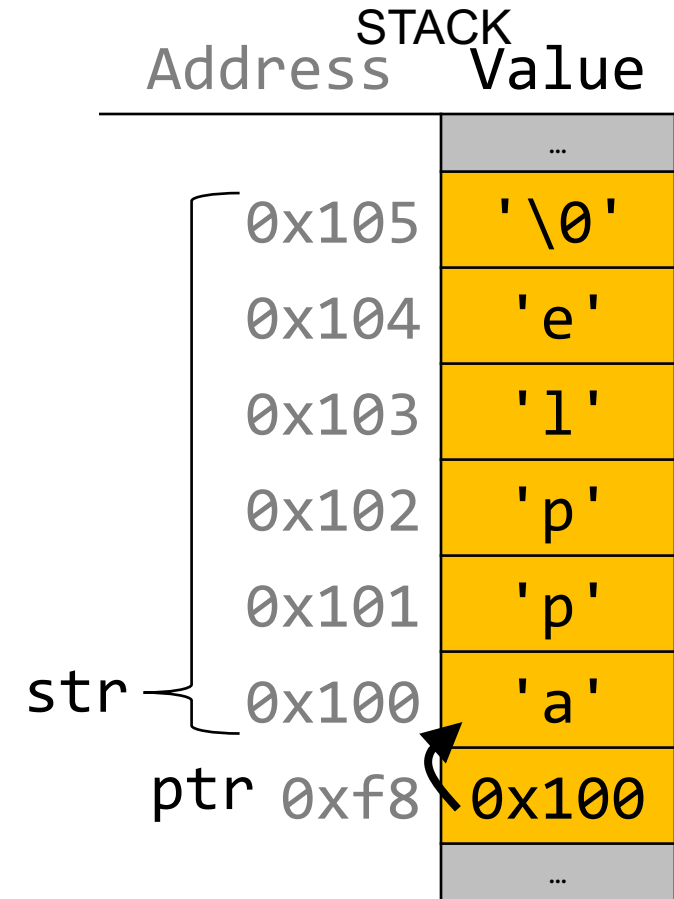
```
char *str = "apple";           // e.g. 0xffff0  
char *str2 = "apple 2";       // e.g. 0xfe0  
str = str2;                   // ok! Both store address 0xfe0
```

# 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;  
    ...  
}
```

main()



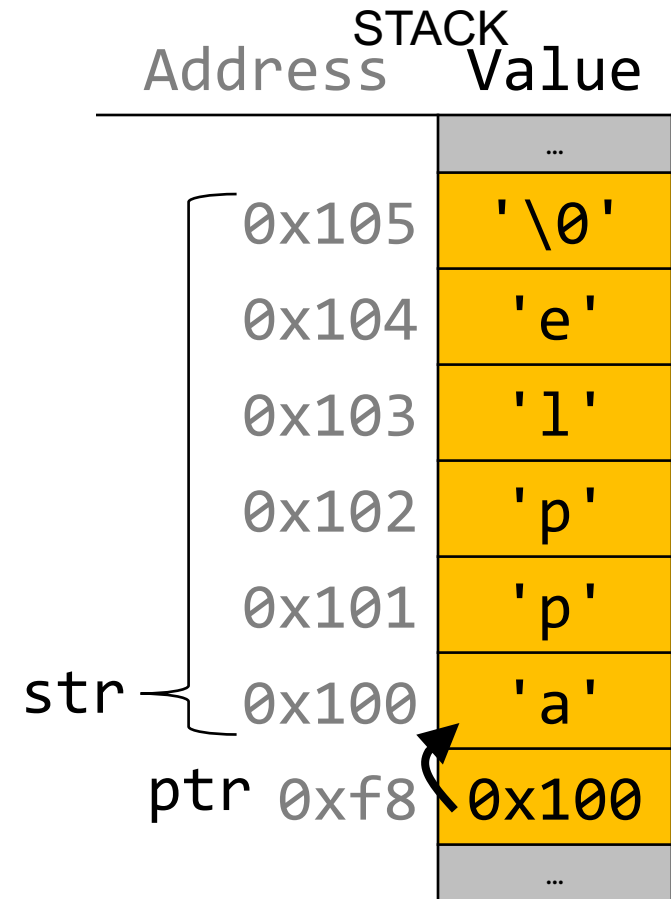


# Arrays and Pointers

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;  
    ...  
}
```

main()



**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'
```

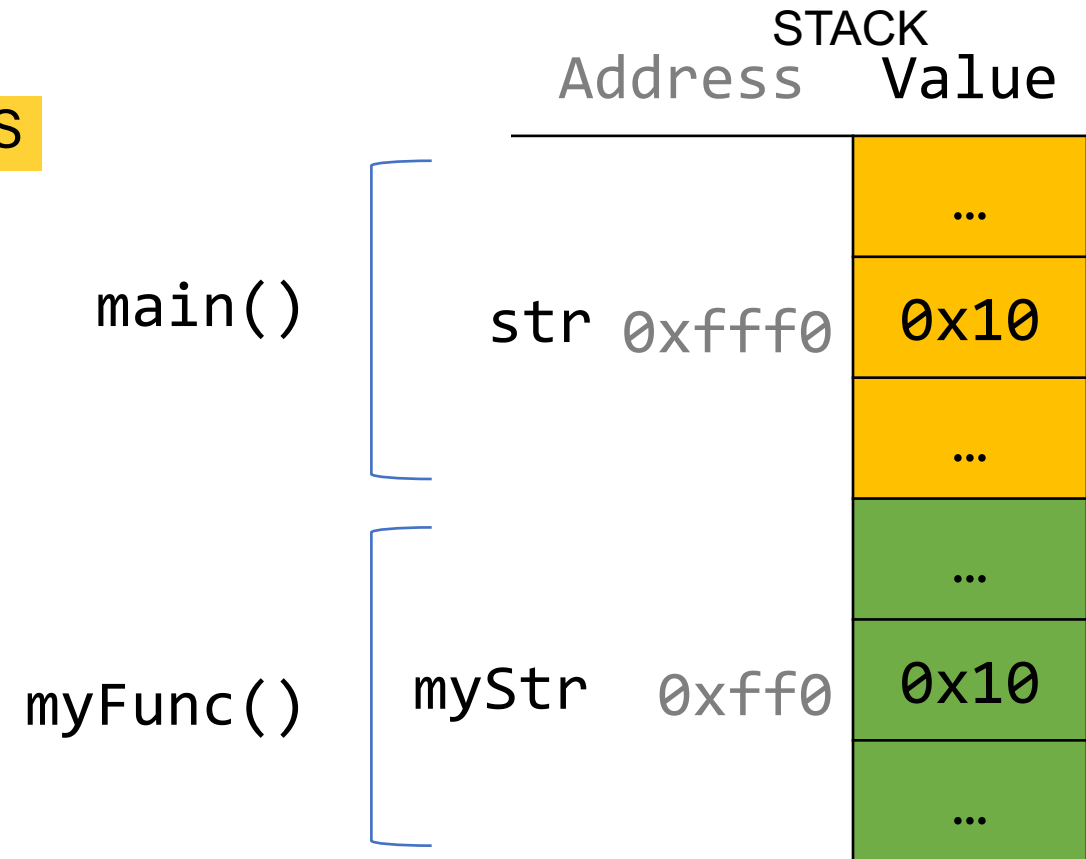
TEXT SEGMENT	
Address	Value
	...
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	...

**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);  
    ...  
}
```

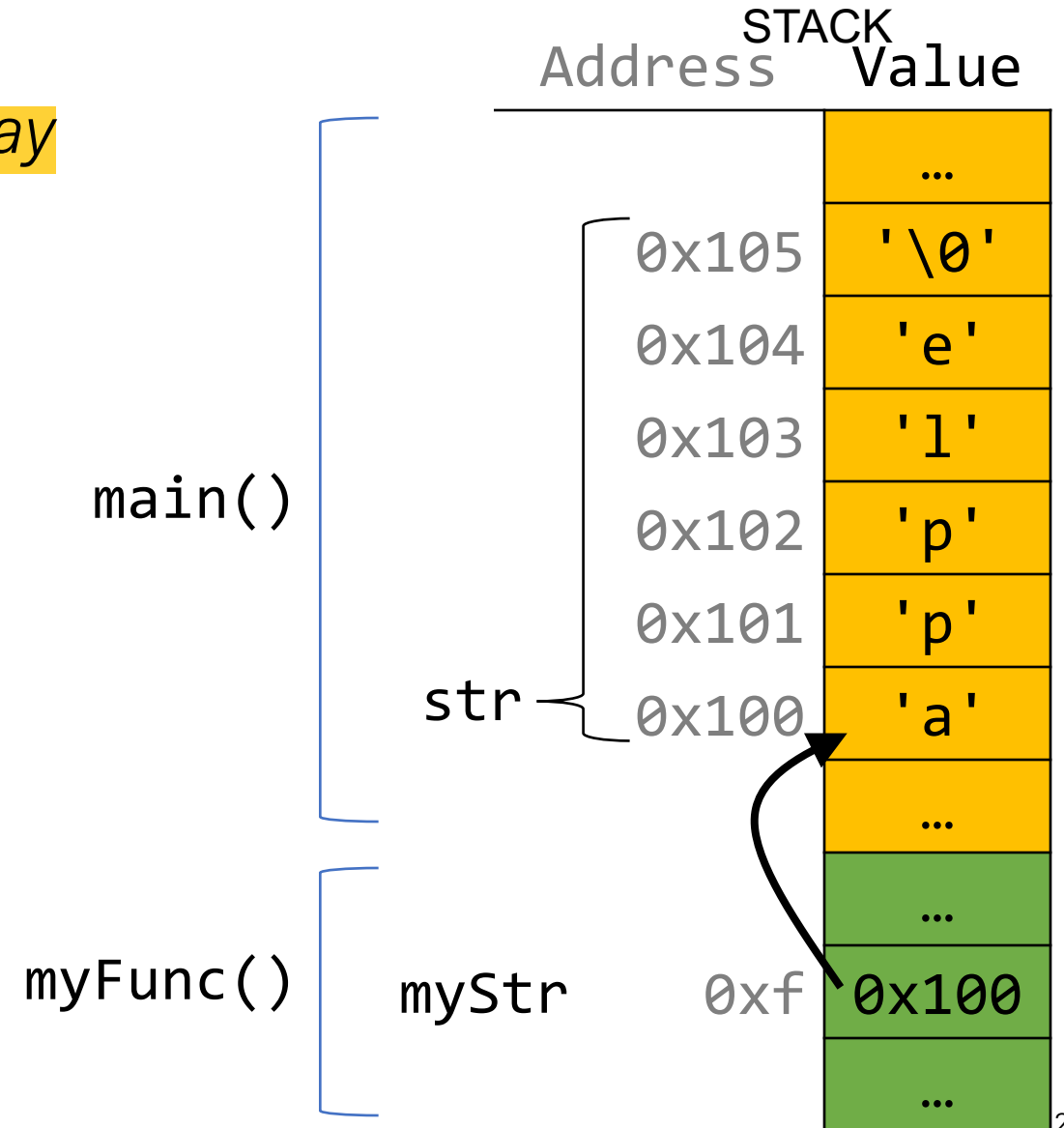




# Strings as Parameters

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.

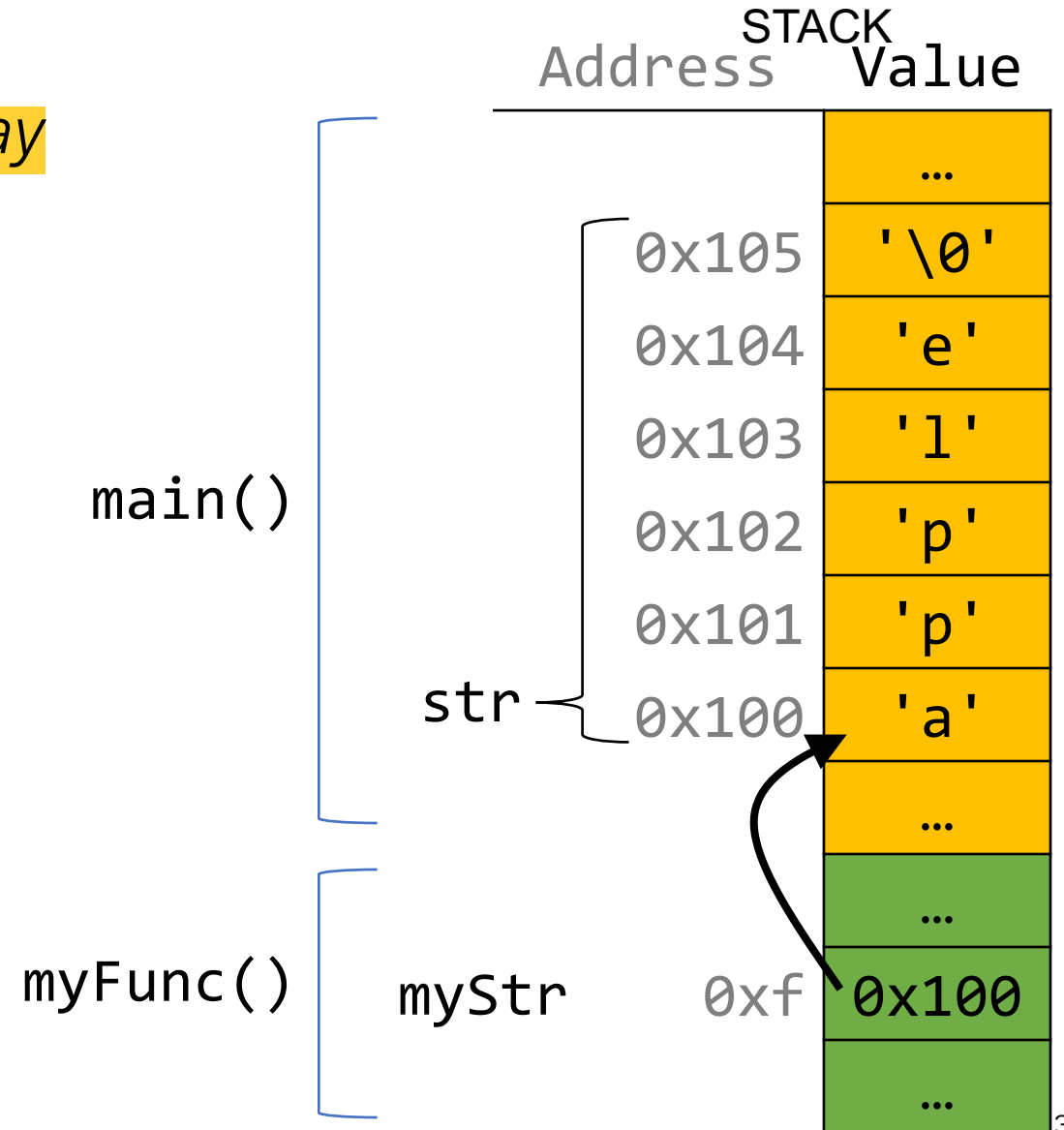
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    myFunc(str);  
    ...  
}
```



# 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.

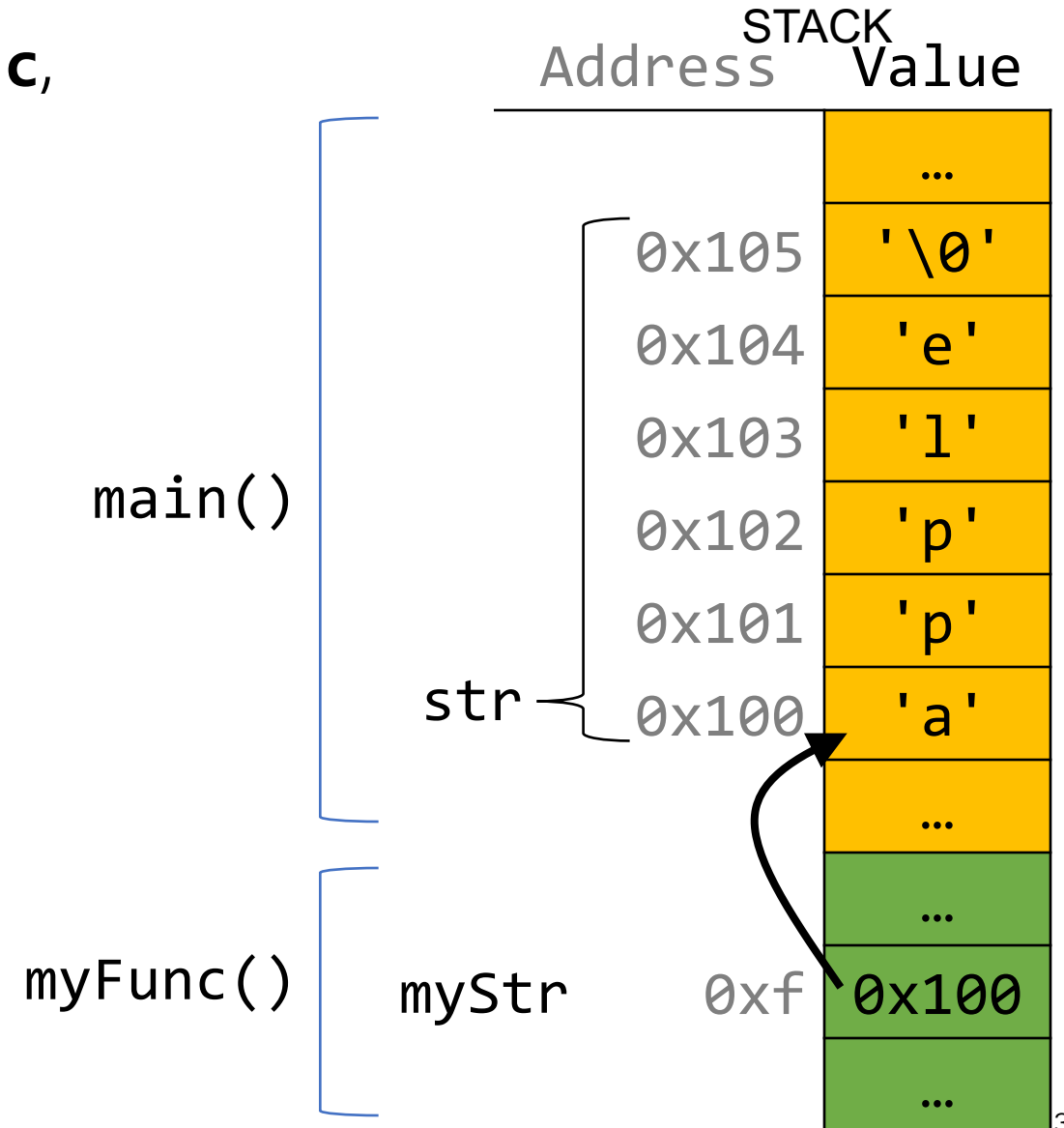
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    // equivalent  
    char *strAlt = str;  
    myFunc(strAlt);  
    ...  
}
```



# Strings as Parameters

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

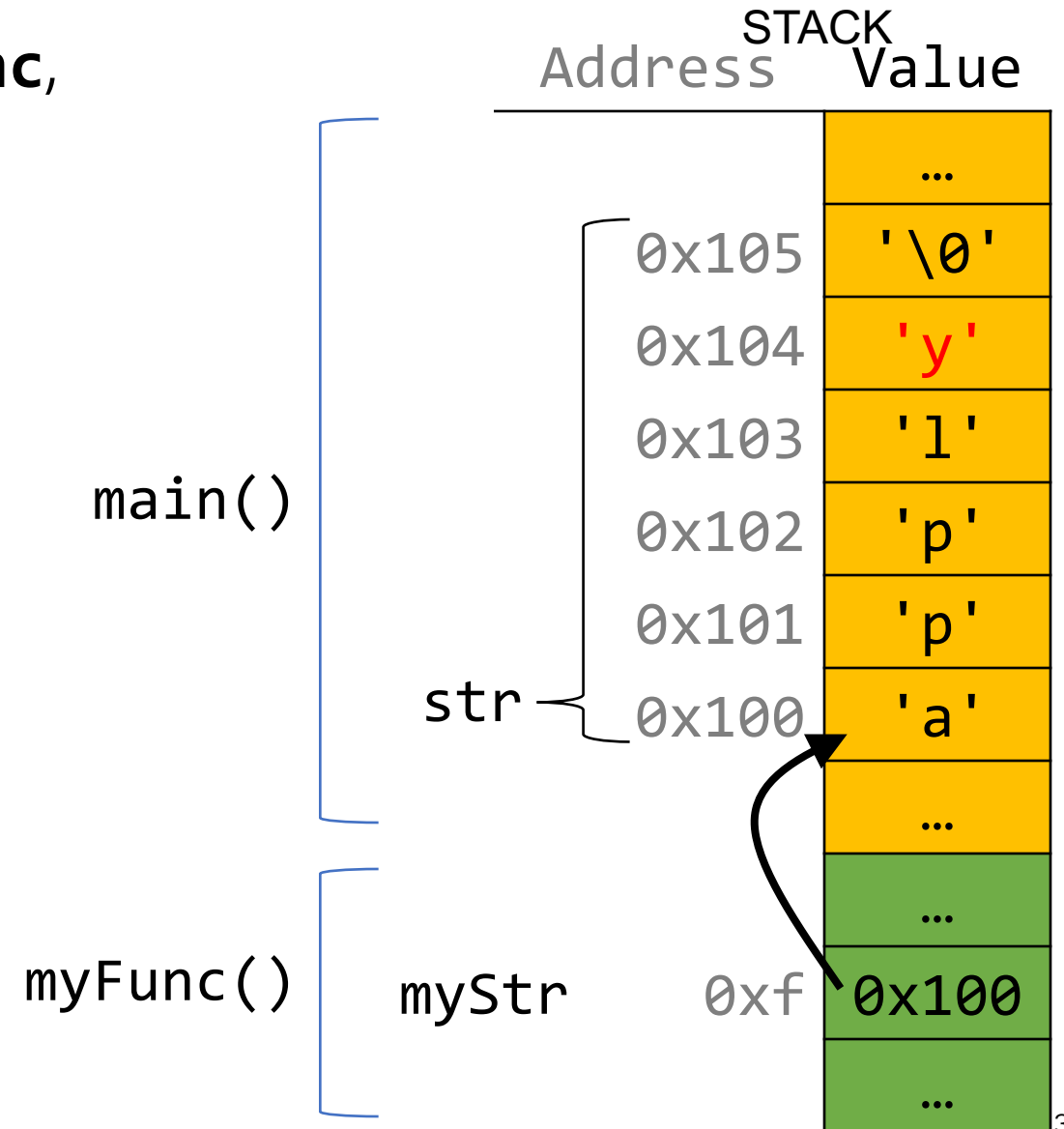
```
void myFunc(char *myStr) {  
    myStr[4] = 'y';  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    myFunc(str);  
    printf("%s", str);    // apply  
    ...  
}
```



# 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.

# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

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







**Will there be a compile error/segfault? If no errors, what is printed?**

① Start presenting to display the poll results on this slide.

# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`  
`strcpy(str, "Hello1");`  
Compile error (cannot reassign array)



# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`  
`strcpy(str, "Hello1");`  
Compile error (cannot reassign array)

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





**Will there be a compile error/segfault? If no errors, what is printed?**

① Start presenting to display the poll results on this slide.

# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`  
`strcpy(str, "Hello1");`  
Compile error (cannot reassign array)

2. `char *str = "Hello2";`  
Segmentation fault (string literal)



# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`  
`strcpy(str, "Hello1");`  
Compile error (cannot reassign array)

2. `char *str = "Hello2";`  
Segmentation fault (string literal)

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





**Will there be a compile error/segfault? If no errors, what is printed?**

① Start presenting to display the poll results on this slide.

# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`  
`strcpy(str, "Hello1");`  
Compile error (cannot reassign array)

2. `char *str = "Hello2";`  
Segmentation fault (string literal)

3. `char arr[7];`  
`strcpy(arr, "Hello3");`  
`char *str = arr;`  
Prints eulo3





# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`  
`strcpy(str, "Hello1");`  
Compile error (cannot reassign array)

2. `char *str = "Hello2";`  
Segmentation fault (string literal)

3. `char arr[7];`  
`strcpy(arr, "Hello3");`  
`char *str = arr;`  
Prints eulo3

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





**Will there be a compile error/segfault? If no errors, what is printed?**

① Start presenting to display the poll results on this slide.

# char\* vs char[] exercises



Suppose we use a variable `str` as follows:

```
str = str + 1;  
str[1] = 'u';  
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`  
`strcpy(str, "Hello1");`  
Compile error (cannot reassign array)

2. `char *str = "Hello2";`  
Segmentation fault (string literal)

3. `char arr[7];`  
`strcpy(arr, "Hello3");`  
`char *str = arr;`  
Prints eulo3

4. `char *ptr = "Hello4";`  
`char *str = ptr;`  
Segmentation fault (string literal)



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

# Lecture Plan

- Strings in Memory (cont'd.)
- **Pointers and Parameters**
- Double Pointers
- Arrays in Memory

# 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 \*

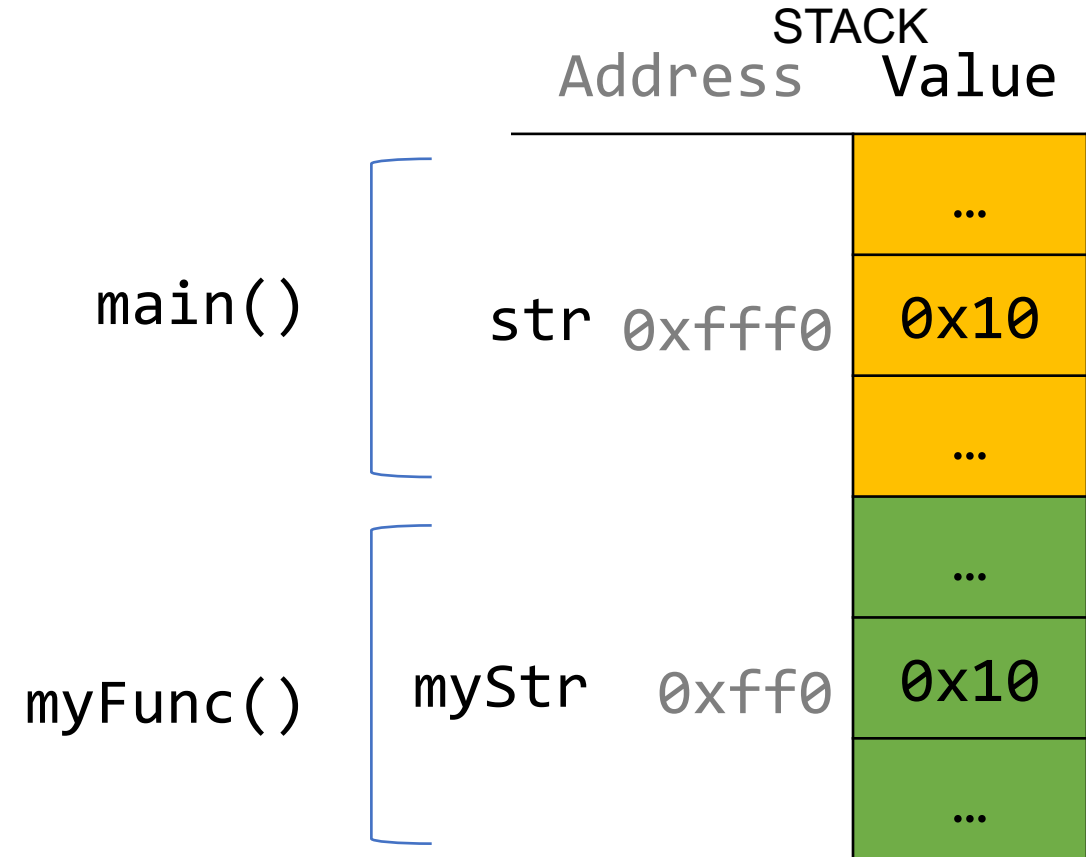
- A char \* is technically a pointer to a **single character**.
- We commonly use char \* as string by having the character it points to be followed by more characters and ultimately a null terminator.
- A char \* could also just point to a single character (not a string).

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

# Revisited: 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.

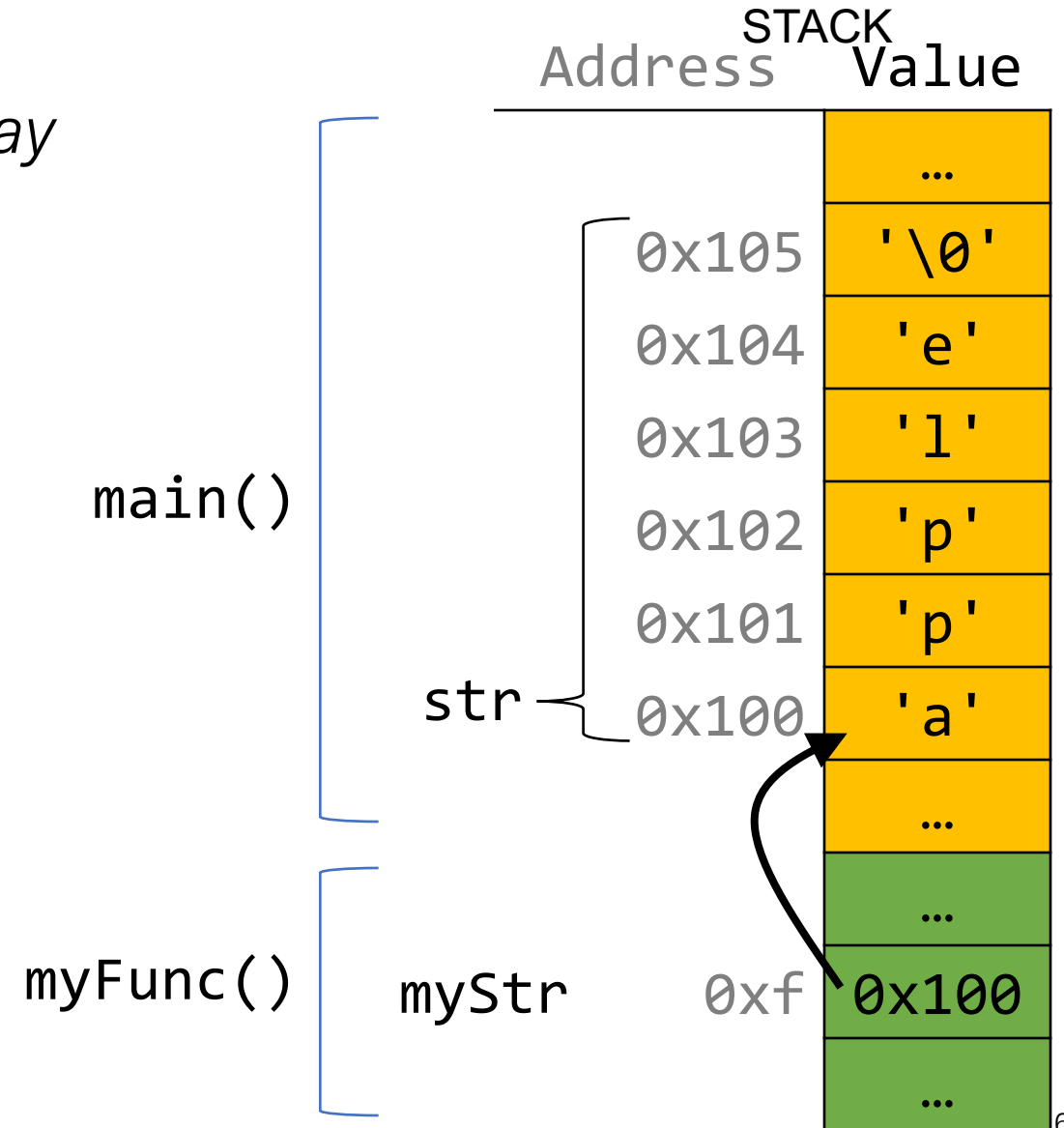
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char *str = "apple";  
    myFunc(str);  
    ...  
}
```



# Revisited: 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.

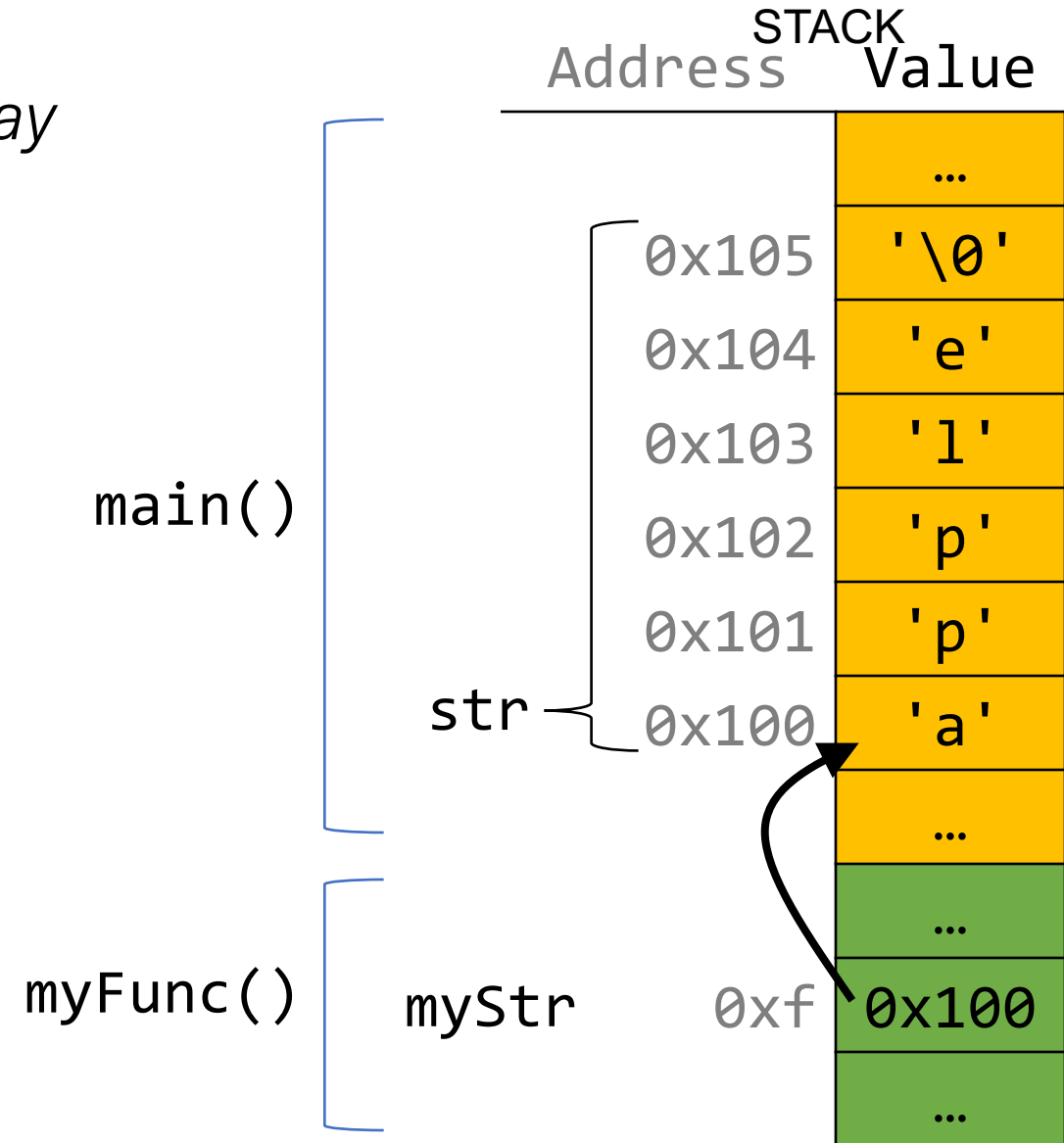
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    myFunc(str);  
    ...  
}
```



# Revisited: 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.

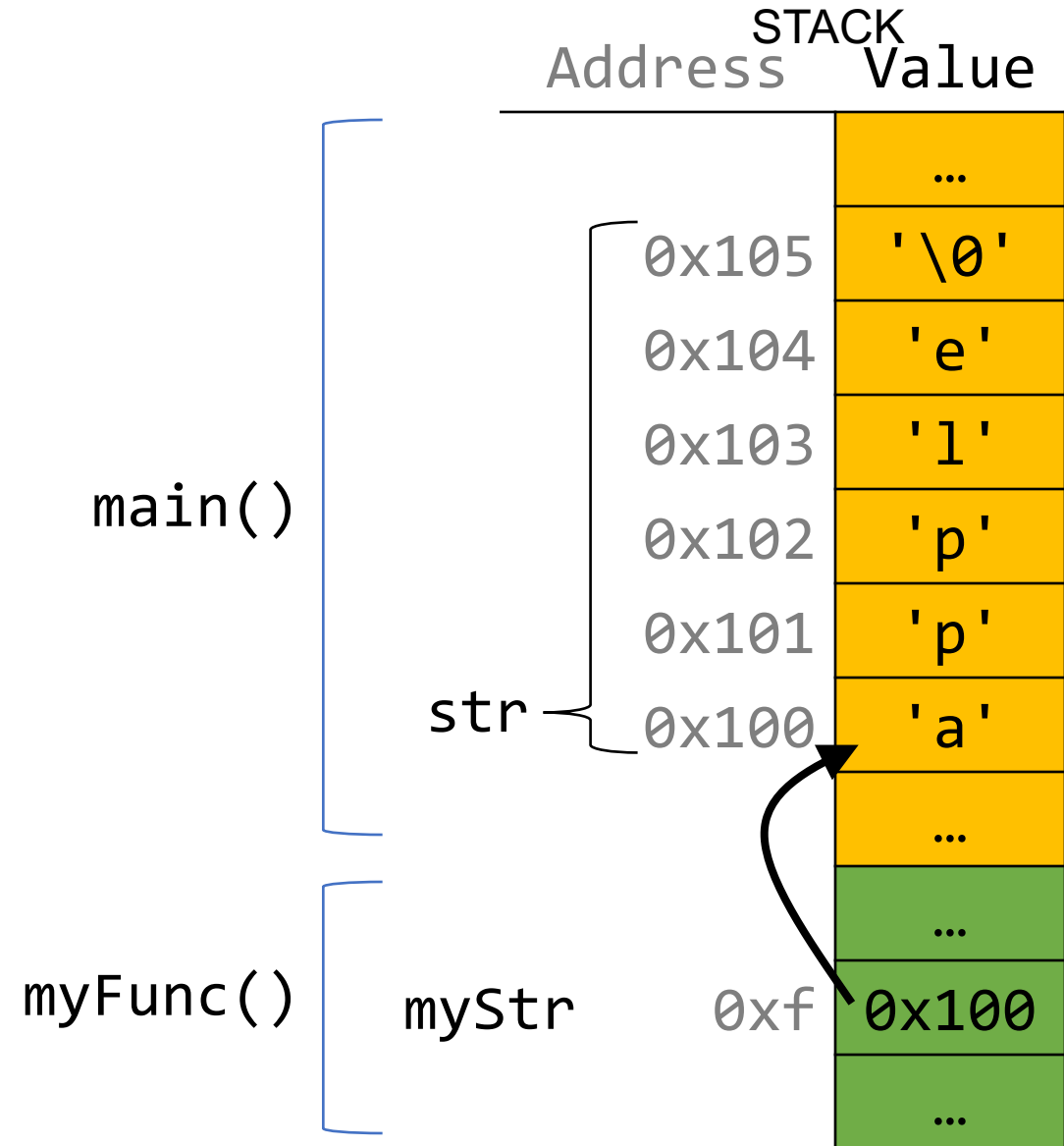
```
void myFunc(char *myStr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    // equivalent  
    char *strAlt = str;  
    myFunc(strAlt);  
    ...  
}
```



# Revisited: Strings as Parameters

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

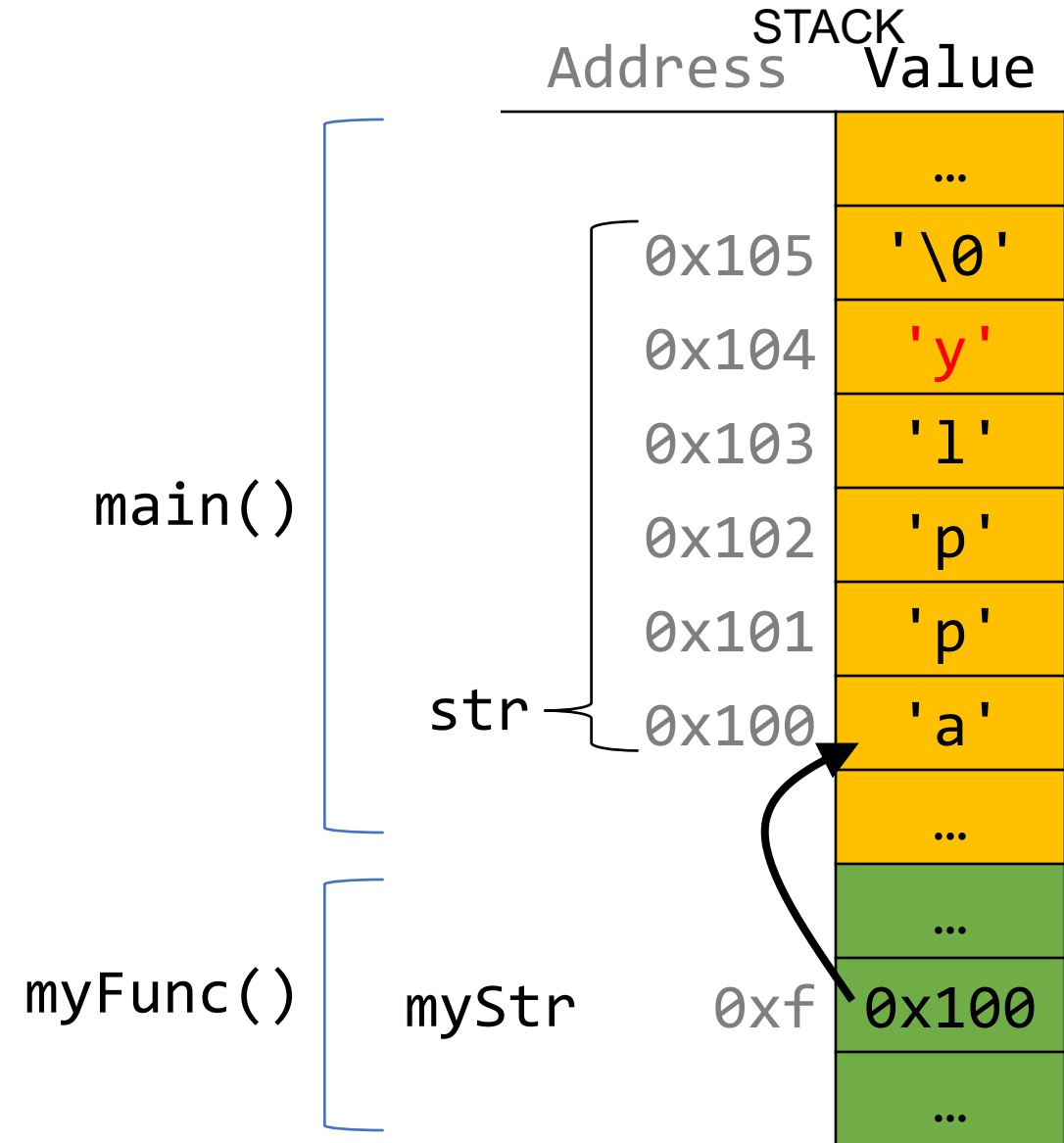
```
void myFunc(char *myStr) {  
    myStr[4] = 'y';  
}  
  
int main(int argc, char *argv[]) {  
    char str[6];  
    strcpy(str, "apple");  
    myFunc(str);  
    printf("%s", str);    // apply  
    ...  
}
```



# Revisited: 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);  
    }  
}
```

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

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

# 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

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers



# 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);  
}
```

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

// 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"  
}
```

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

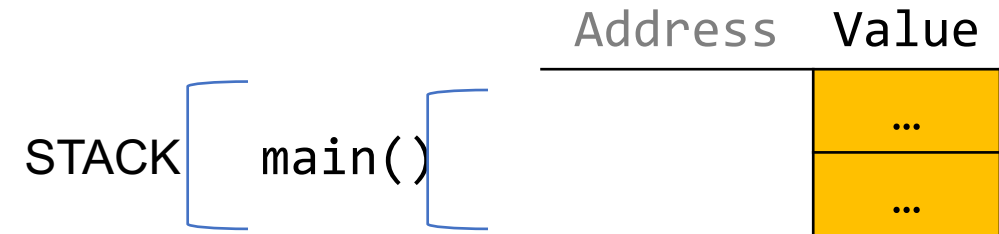
# Demo: Skip Spaces



skip\_spaces.c

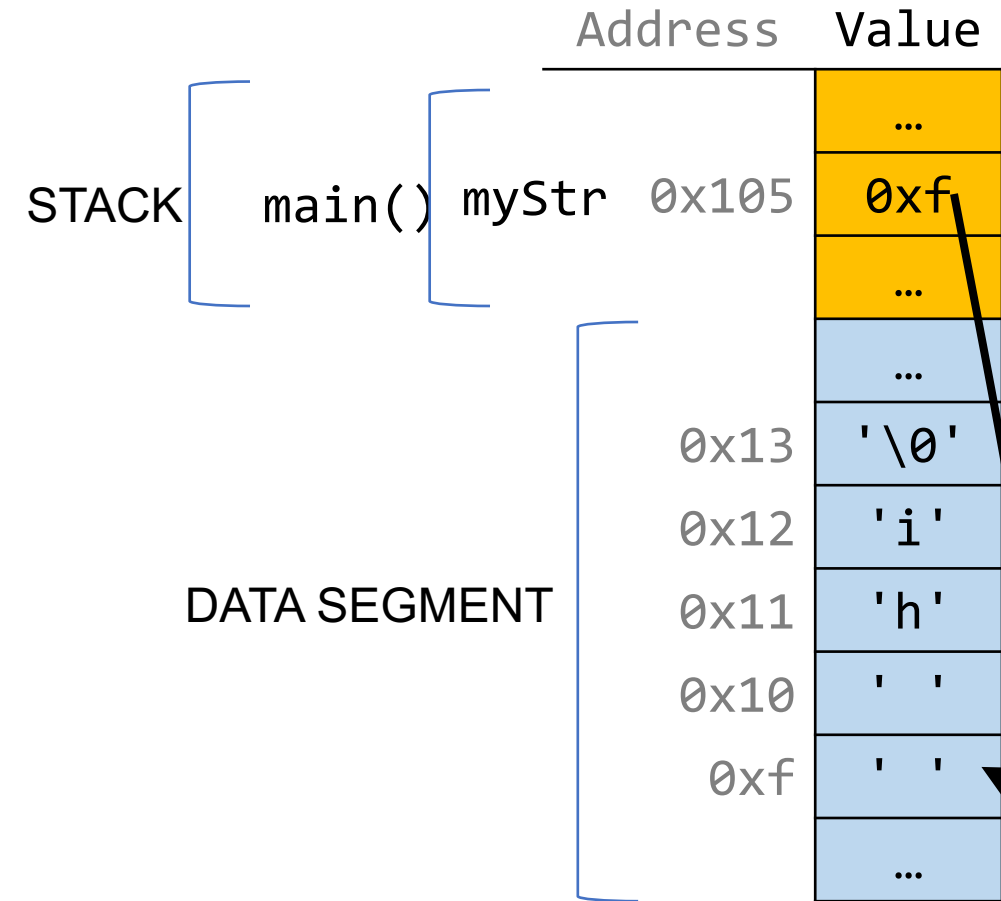
# Pointers to Strings

```
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;  
}
```



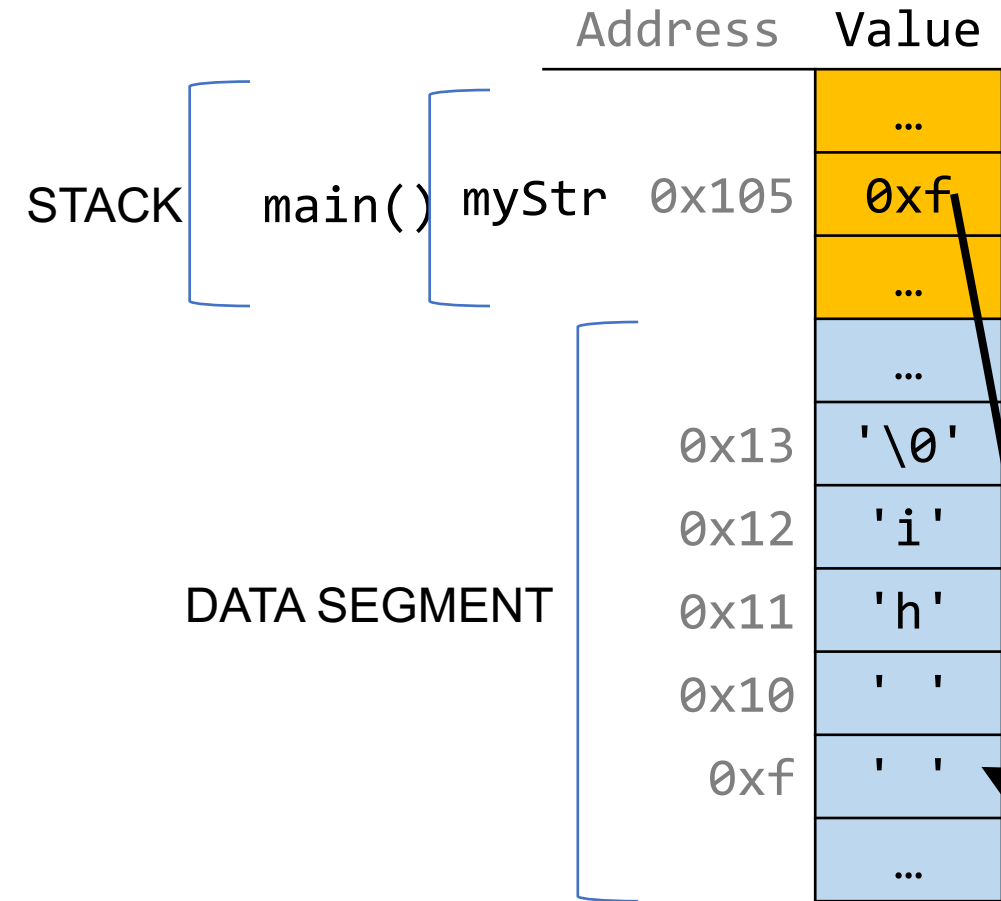
# Pointers to Strings

```
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;  
}
```

STACK

main()

myStr 0x105

skipSpaces()

strPtr 0xf0

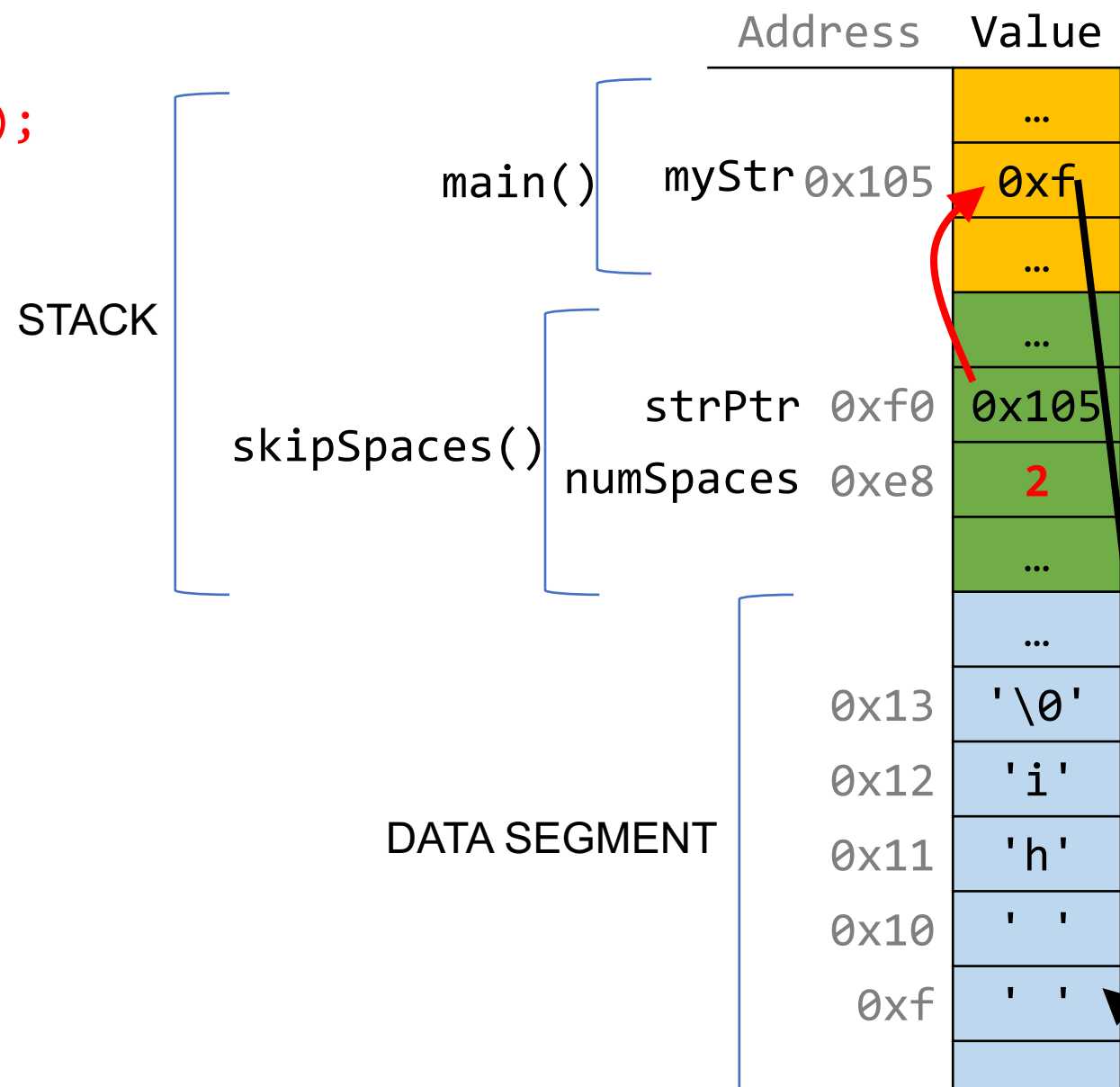
DATA SEGMENT

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



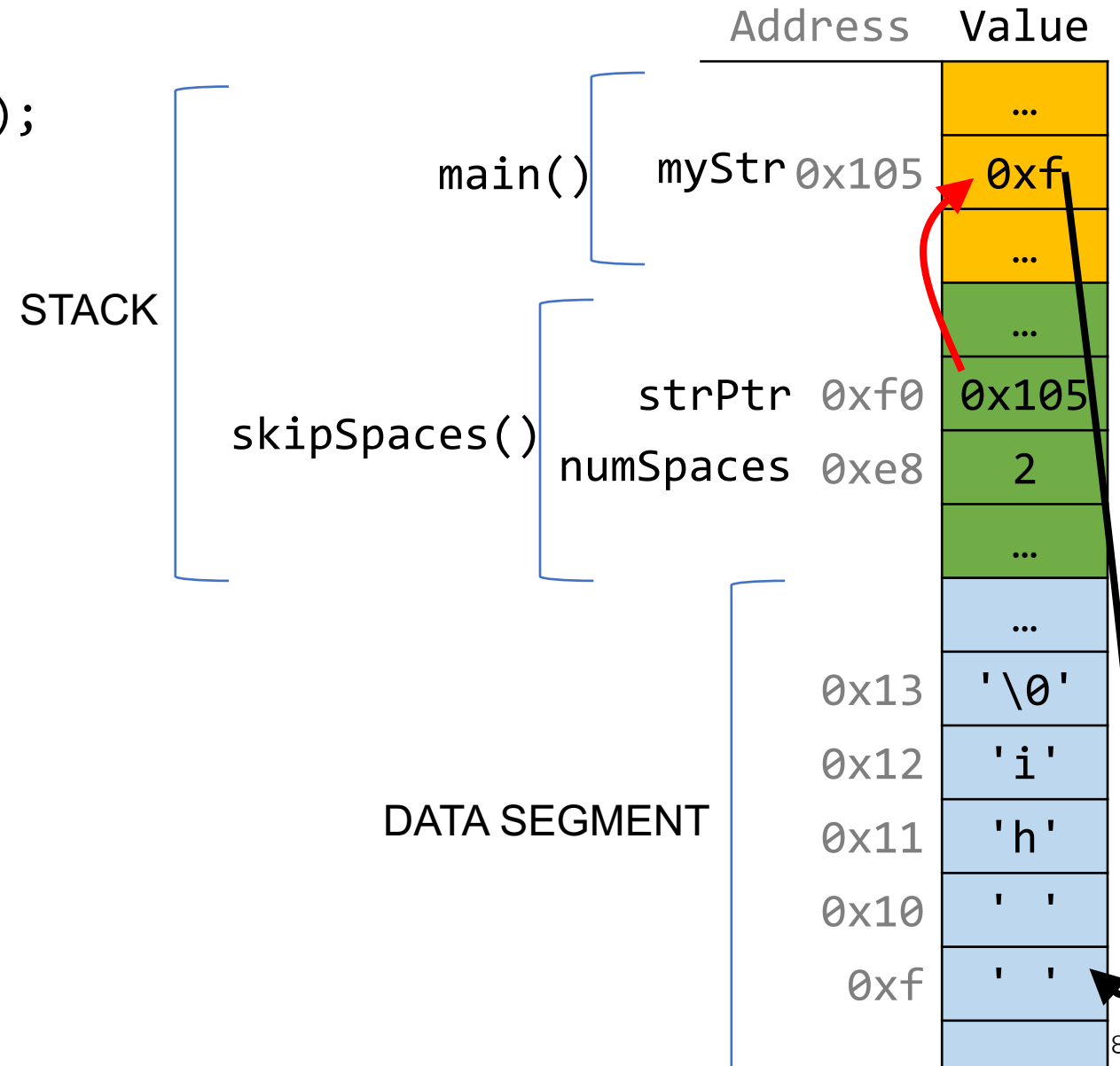
# Pointers to Strings

```
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;  
}
```



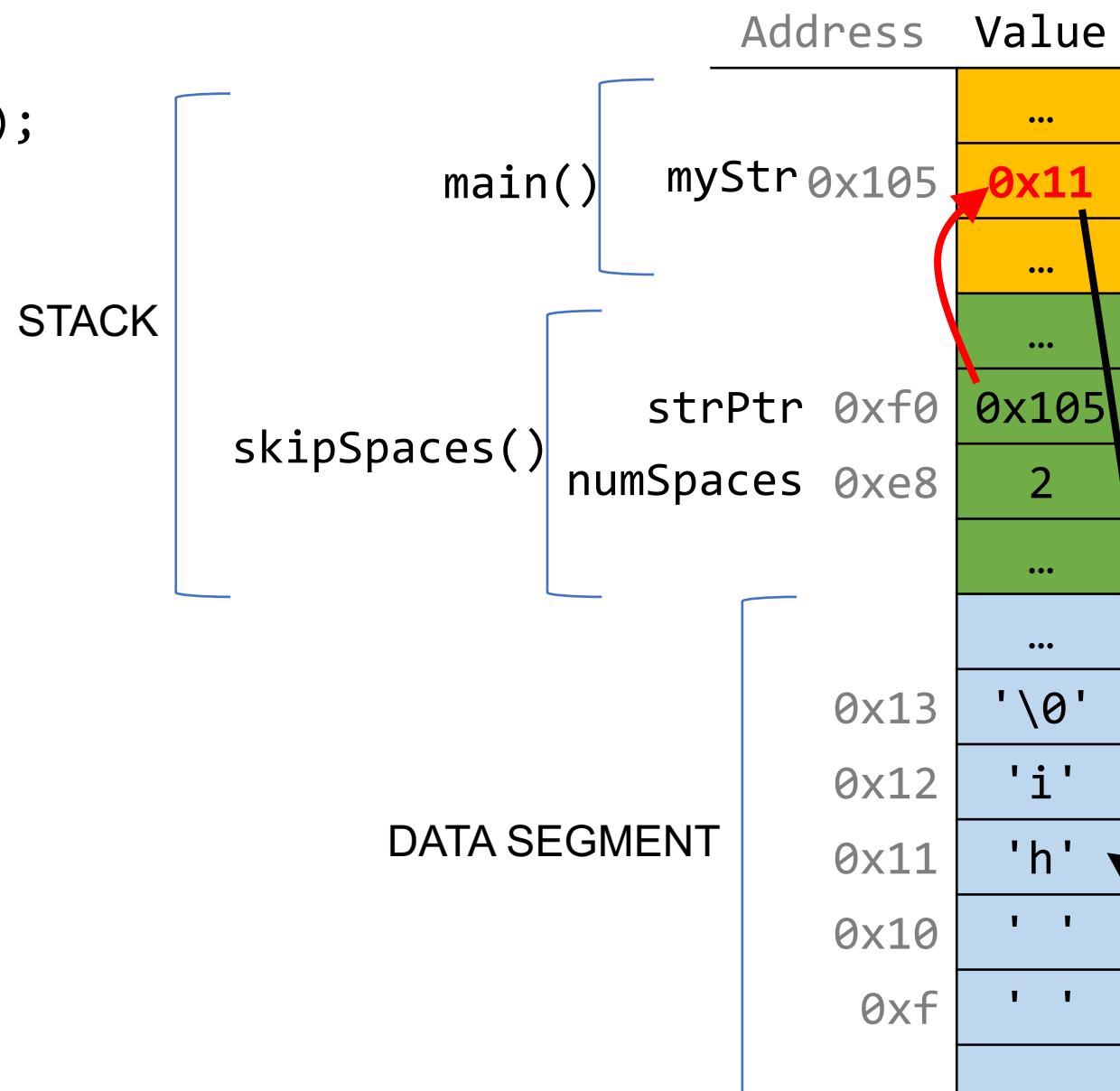
# Pointers to Strings

```
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;  
}
```



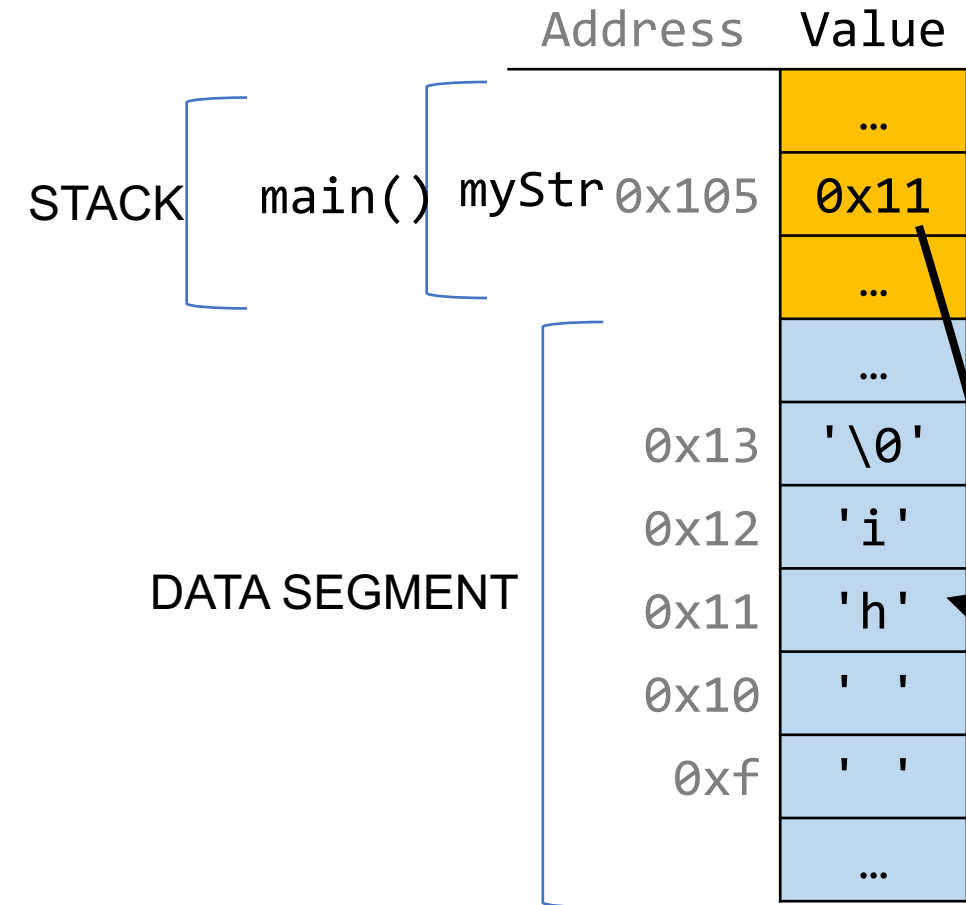
# Pointers to Strings

```
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;  
}
```



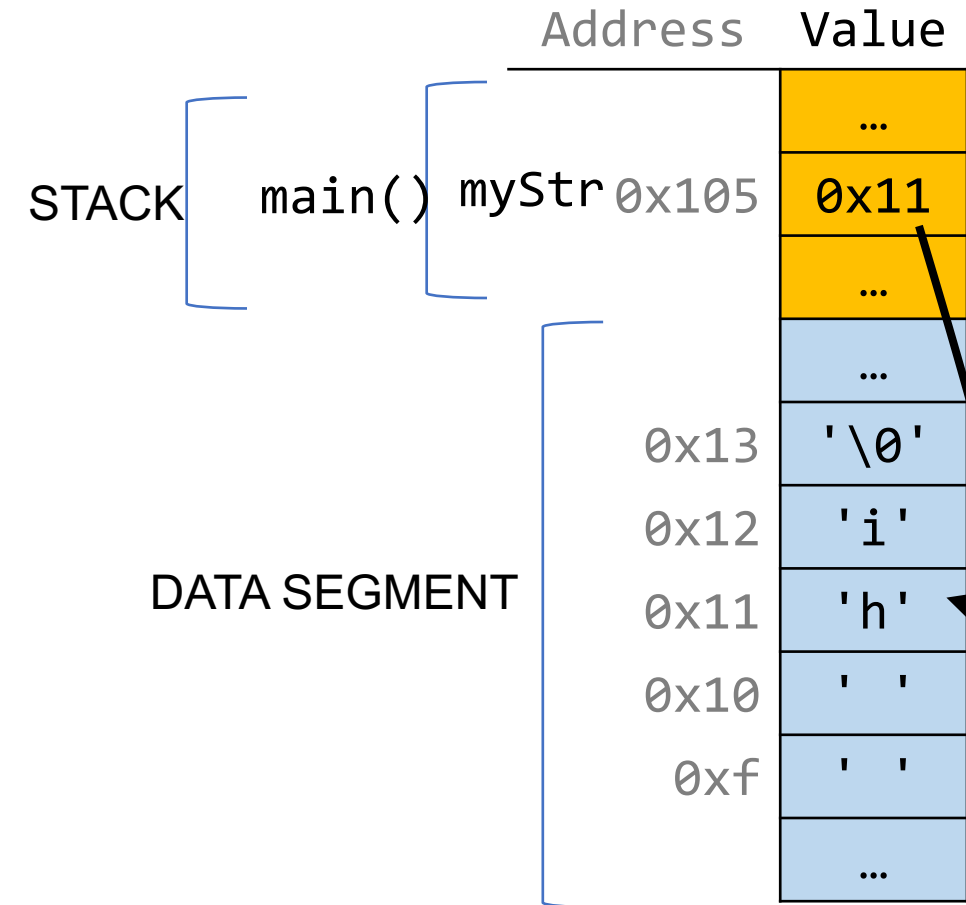
# Pointers to Strings

```
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;  
}
```

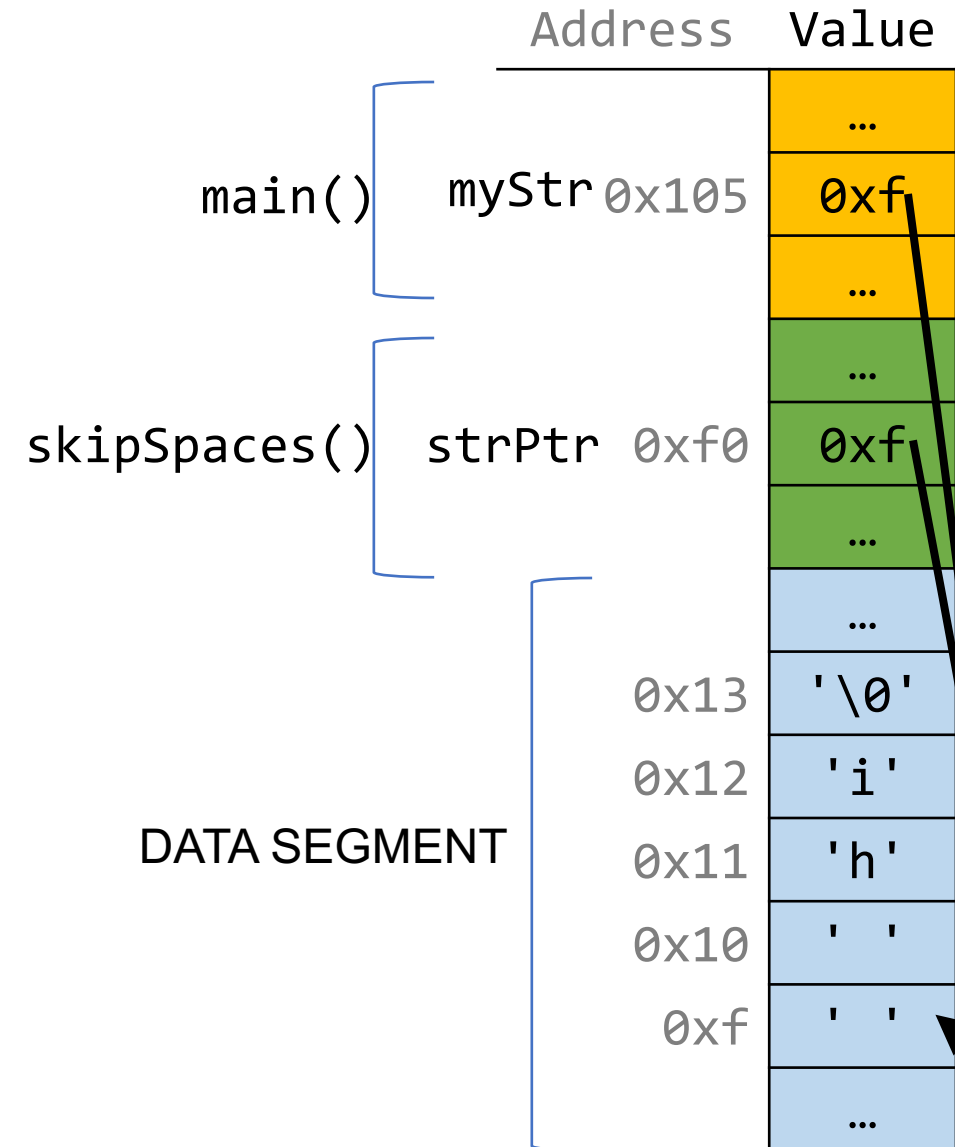


# Making Copies

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

myFunc myFunc  
// hi

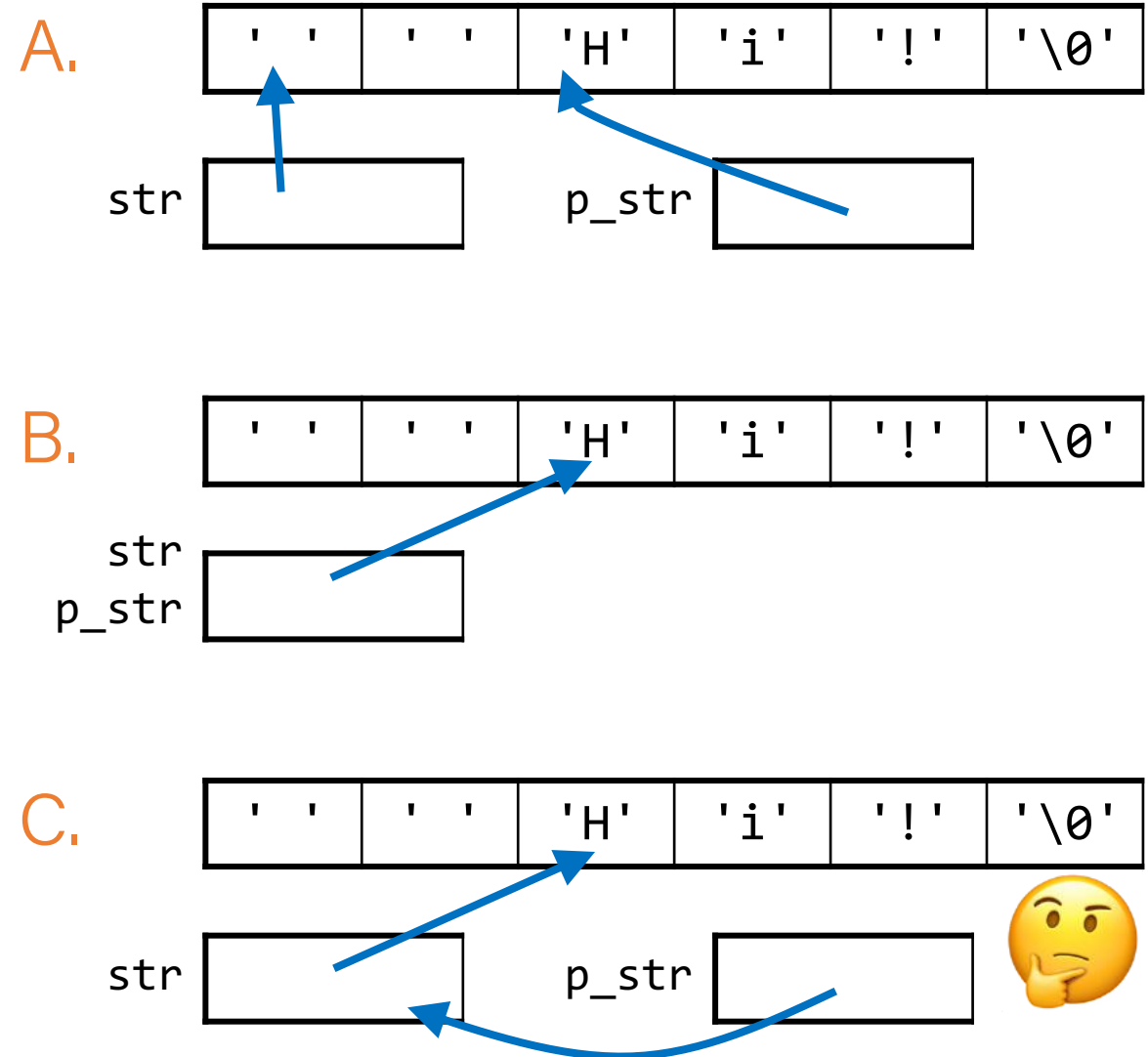
STACK



# 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`)?

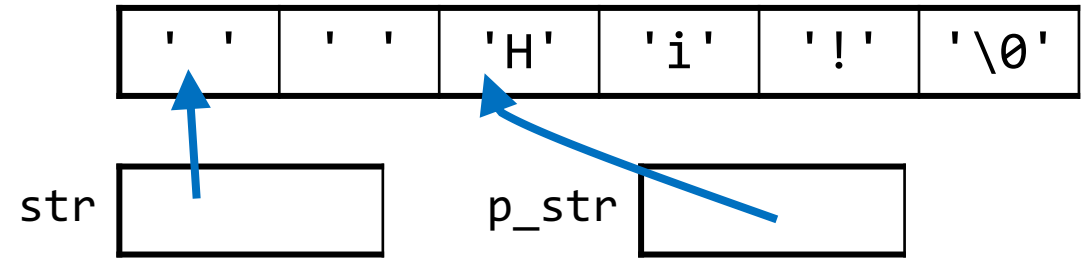


# Skip spaces

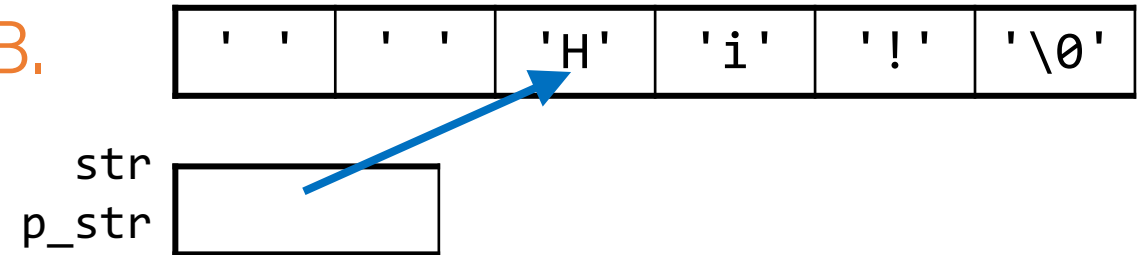
```
1 void skip_spaces(char **p_str) {  
2     int num = strspn(*p_str, " ");  
3     *p_str = *p_str + num;  
4 }  
5 int main(int argc, char *argv[]){  
6     char *str = "  Hi!";  
7     skip_spaces(&str);  
8     printf("%s", str); // "Hi!"  
9     return 0;  
10 }
```

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

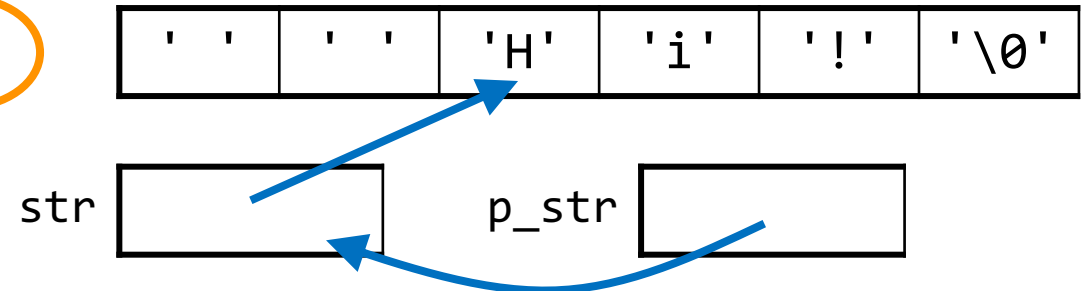
A.



B.



C.





# Recap

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers

**Next Time:** *arrays in memory, pointer arithmetic, dynamically allocated memory*