COMP201

Computer Systems & Programming

Lecture #7 – Arrays and Pointers

Aykut Erdem // Koç University // Spring 2024

KOÇ UNIVERSITY

Recap: Common `string.h` Functions

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

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

```c
int x = 2;


// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;


// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xPtr);    // prints 2
```

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.

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 — 0x100

Recap: `char *`

When we declare a char pointer equal to a string literal, the characters are not stored on the stack. Instead, they are stored in a special area of memory called the "data segment". We *cannot modify memory in this segment*.

```
char *str = "hi";
```

The pointer variable (e.g. **str**) refers to the *address of the first character of the string in the data segment*.

This applies only to creating *new* strings with `char *`. This does **<u>not</u>** apply for making a `char *` that points to an existing stack string.

Address	Value
	…
STACK str 0xff0	0x10
	…
	…
0x12	'\0'
0x11	'i'
0x10	'h'
	…

STACK

DATA SEGMENT

7
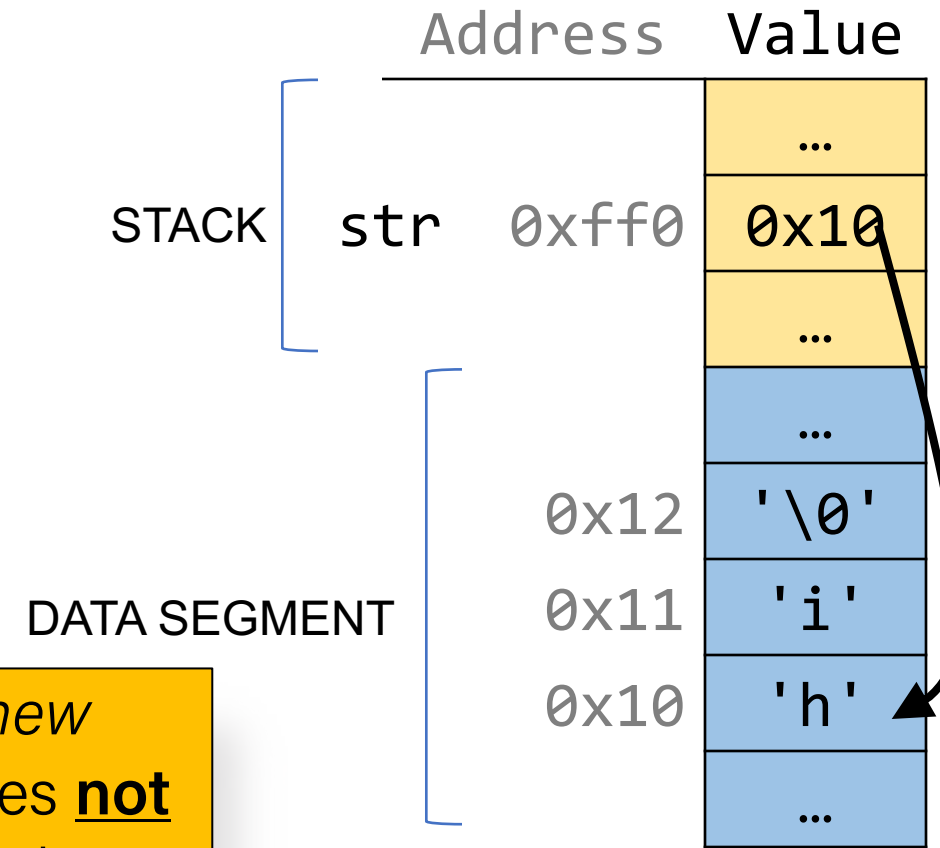
Recap: `char *`

There is another convenient way to create a string if we do not need to modify it later. We can create a `char *` and set it directly equal to a string literal.

```
char *myString = "Hello, world!";
char *empty = "";


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

Recap: `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.

Recap: Strings In Memory

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

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

> 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 *.

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

Recap: 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?

Recap: 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?

Recap: Memory Locations

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

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

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

13

Recap: 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?

Recap: Memory Locations

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

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


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

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

Q: Is the ... re modifiab...

A: No. ... itself and how...

Q: So th... ow can I tell if th...

A: You c... on in your fun... a read-only stri...

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

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

char *

A **char** * variable refers to a single character. We can reassign an existing **char** * pointer to be equal to another **char** * pointer.
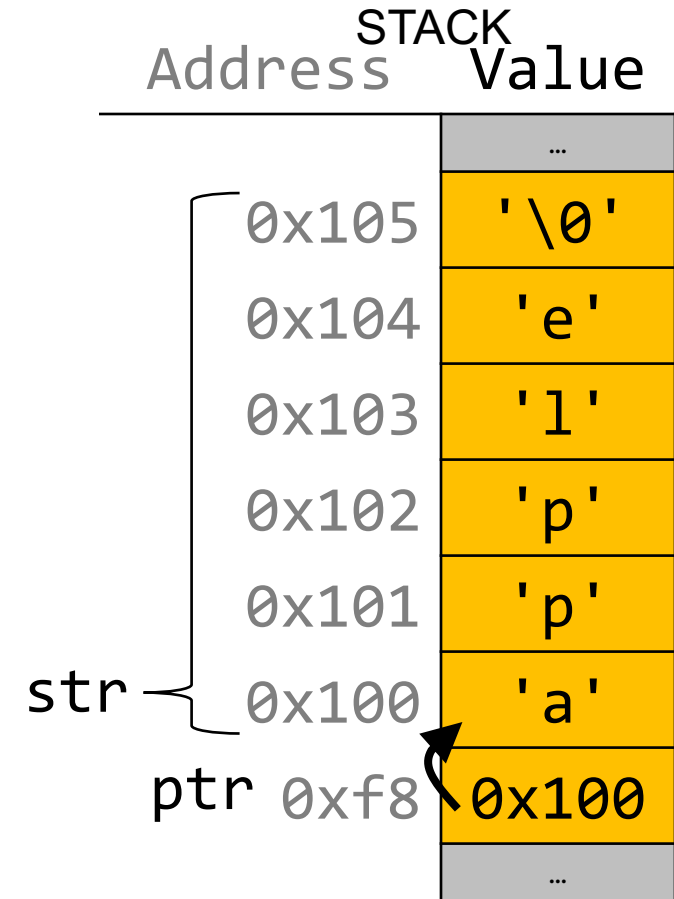
```
char *str = "apple";        // e.g. 0xfff0
char *str2 = "apple 2";     // e.g. 0xfe0
str = str2;    // ok!  Both store address 0xfe0
```

Arrays and Pointers

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

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


STACK

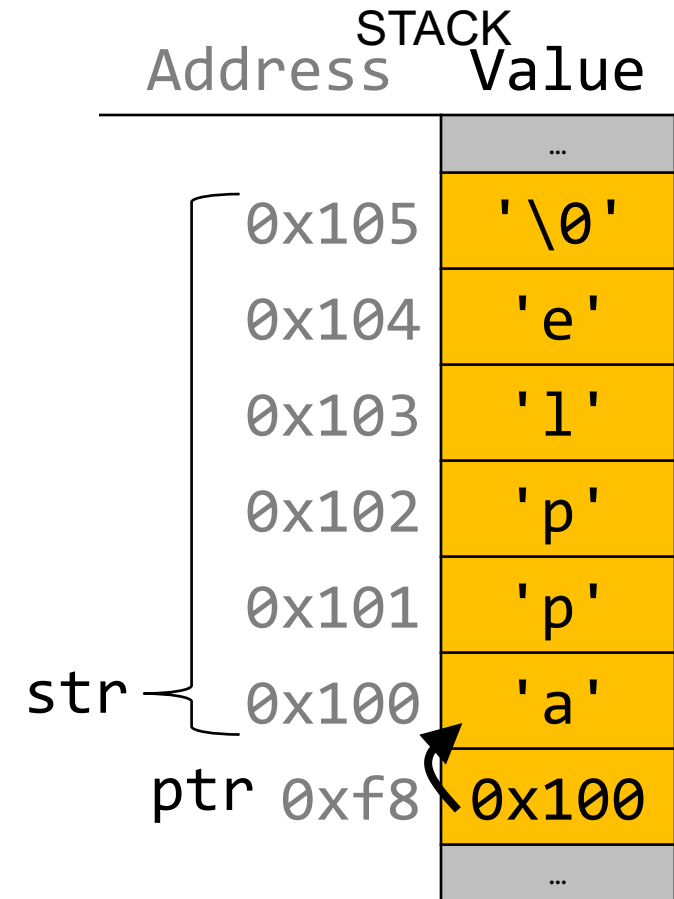| Address	Value
	…
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
0xf8	0x100
	…

main()

str

ptr

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;
    ...
}
```

STACK

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

main()

str

ptr

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

Pointer Arithmetic

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

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

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

TEXT SEGMENT

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

26

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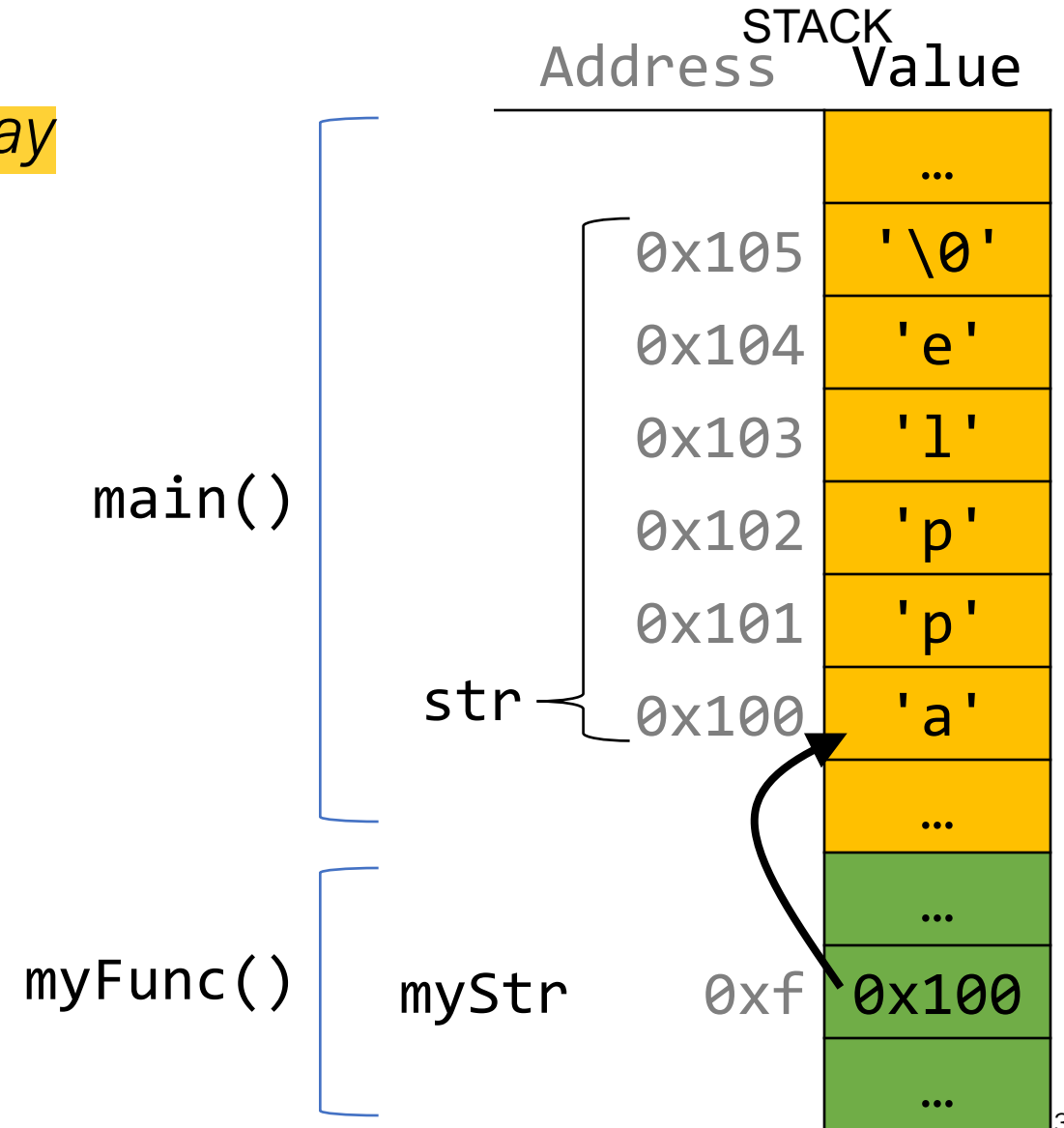
```
void myFunc(char *myStr) {
    ...
}

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

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 <mark>*copy of the address of the first array element*</mark> 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);
    ...
}
```

STACK

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

main()

str

myFunc()

myStr

30

Strings as Parameters

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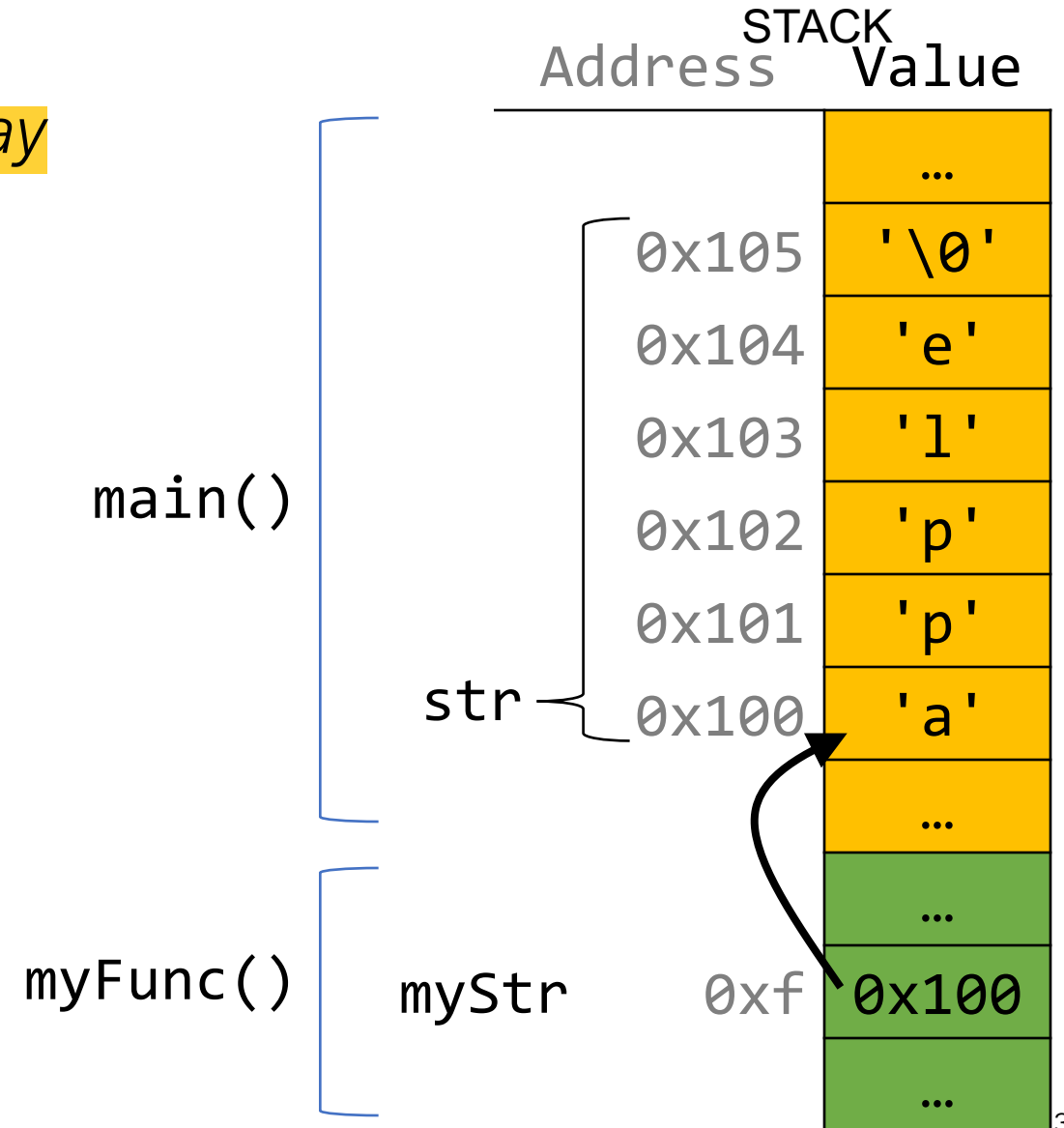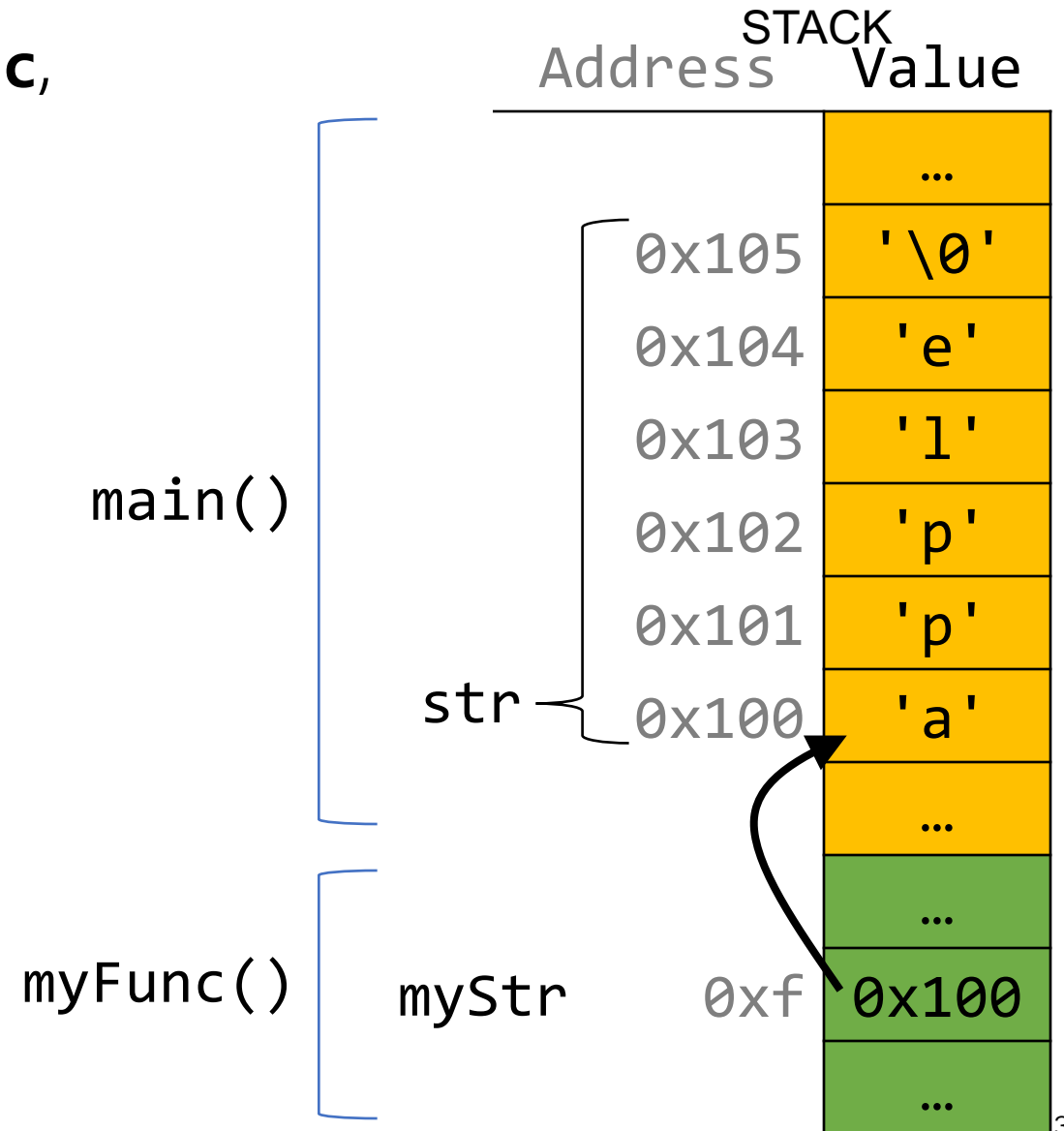
```
void myFunc(char *myStr) {
    ...
}

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

STACK

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

main()

str

myFunc()   myStr

31

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	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	…
	…
0xf	0x100
	…

main()

str

myFunc()   myStr

32

Strings as Parameters

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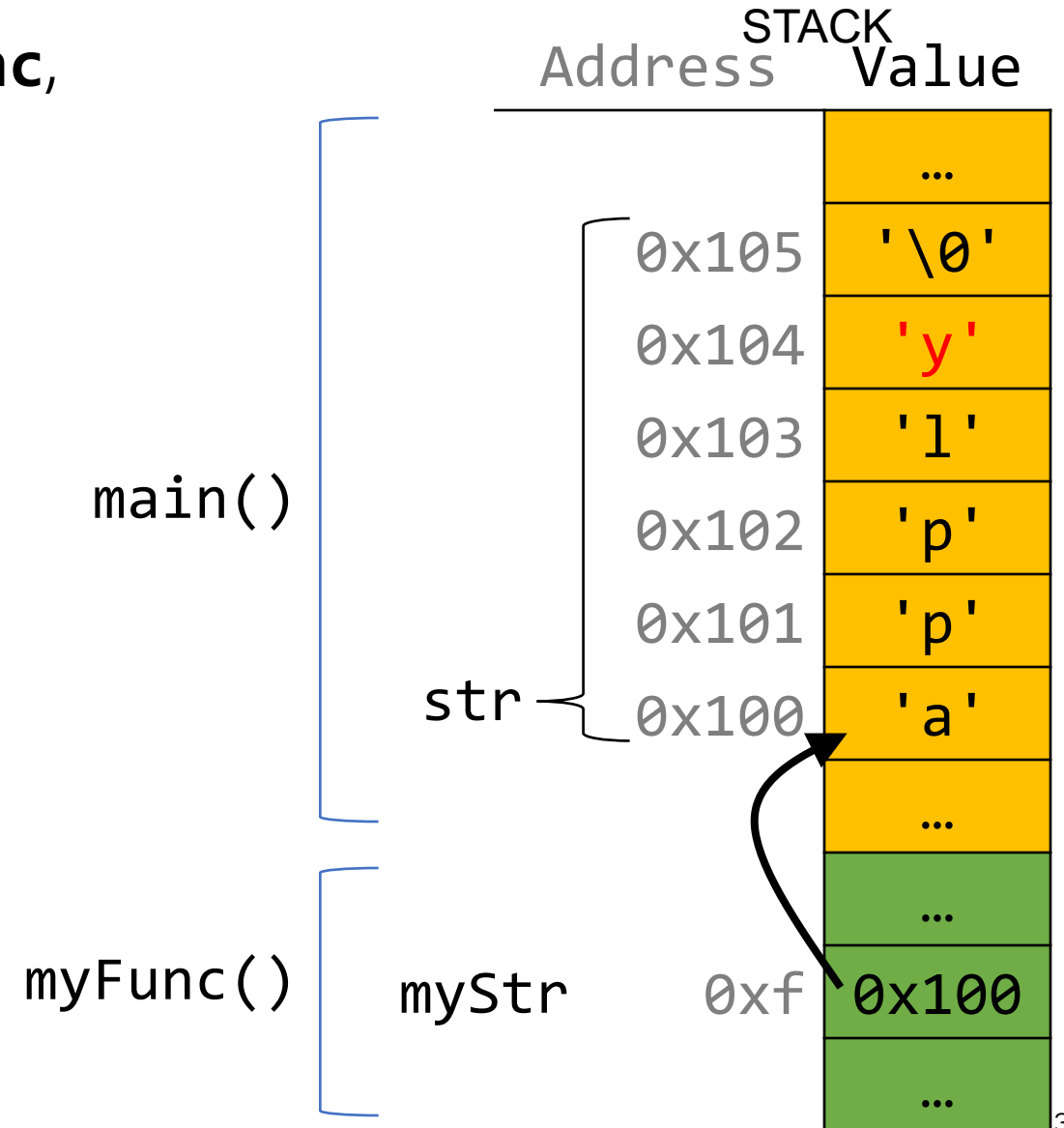
```c
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

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

STACK

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

main()

str

myFunc()   myStr

33

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);
```

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

For each of the following instantiations:

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

char* vs char[] exercises

Suppose we use a variable `str` as follows:

For each of the following instantiations:

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

- 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);
```

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

For each of the following instantiations:

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

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

char* vs char[] exercises

Suppose we use a variable `str` as follows:

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

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

For each of the following instantiations:

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);
```

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

For each of the following instantiations:

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;
   ```

char* vs char[] exercises

Suppose we use a variable `str` as follows:

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

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

For each of the following instantiations:

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);
```

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

For each of the following instantiations:

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;
   ```

char* vs char[] exercises

Suppose we use a variable `str` as follows:

For each of the following instantiations:

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

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

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

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

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

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

<u>COMP201 Topic 4</u>: 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.

```c
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.

```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.

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.

```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;
}
```
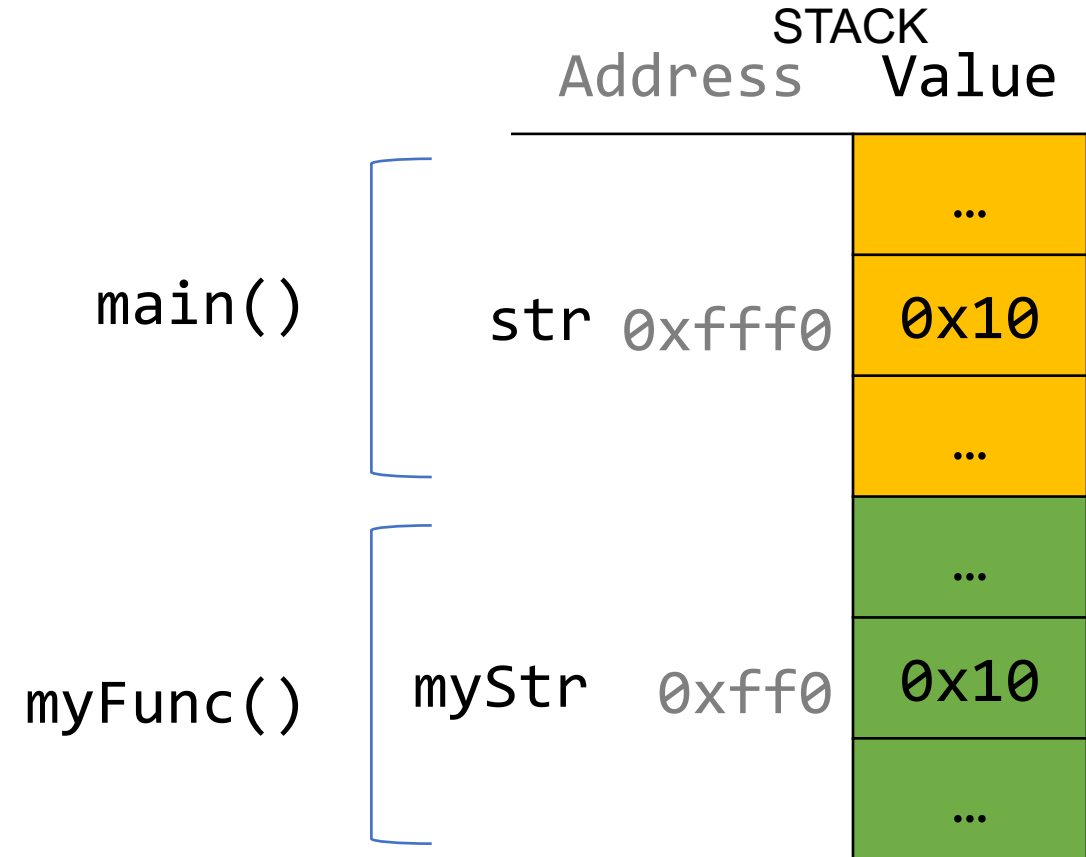
char *

- 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).

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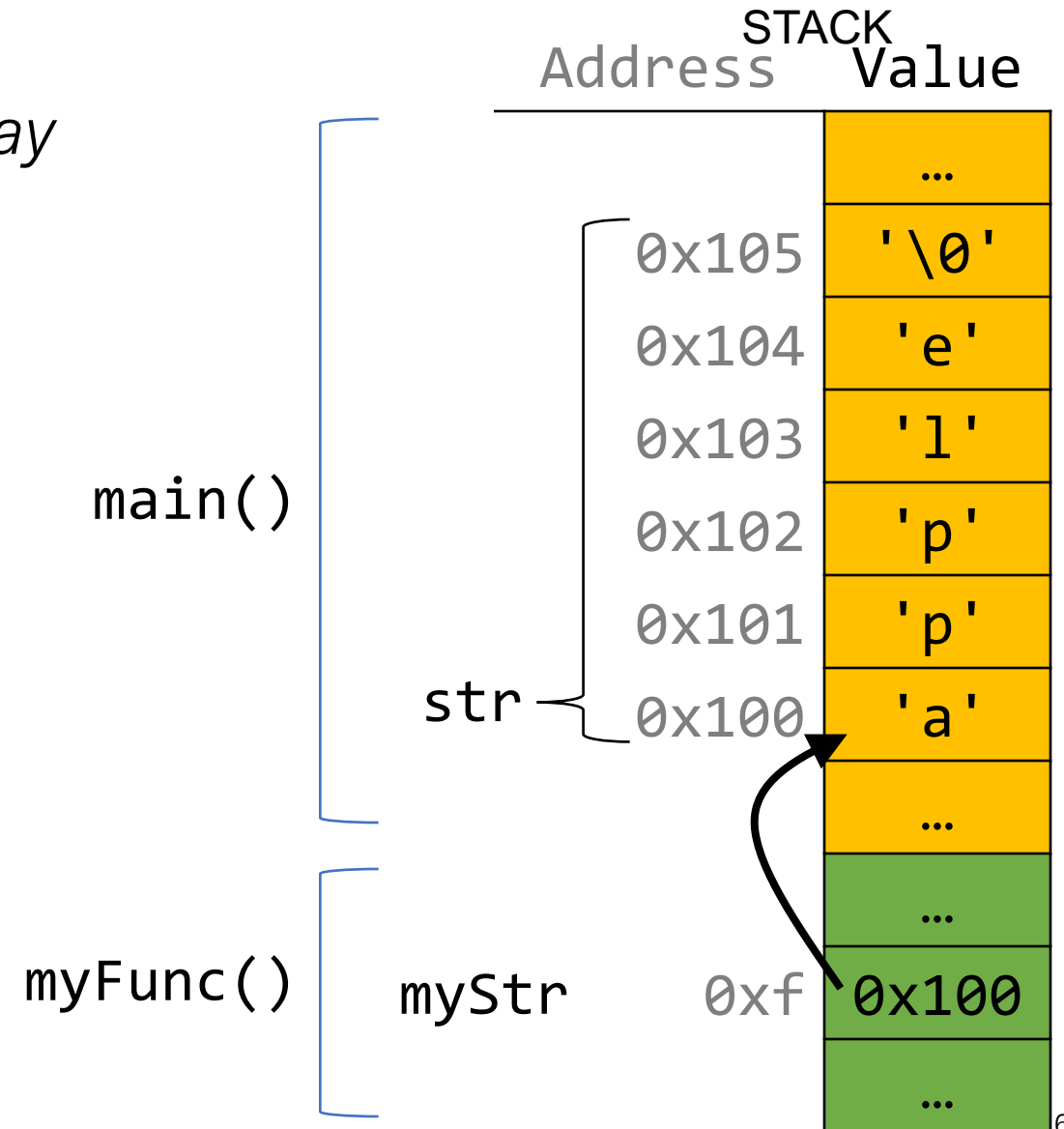
```
void myFunc(char *myStr) {
    ...
}

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

STACK

Address	Value

main()

... (orange)

str 0xfff0 | 0x10 (orange)

... (orange)

myFunc()

... (green)

myStr 0xff0 | 0x10 (green)

... (green)

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);
    ...
}
```

STACK

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

main()

str

myFunc()   myStr

60

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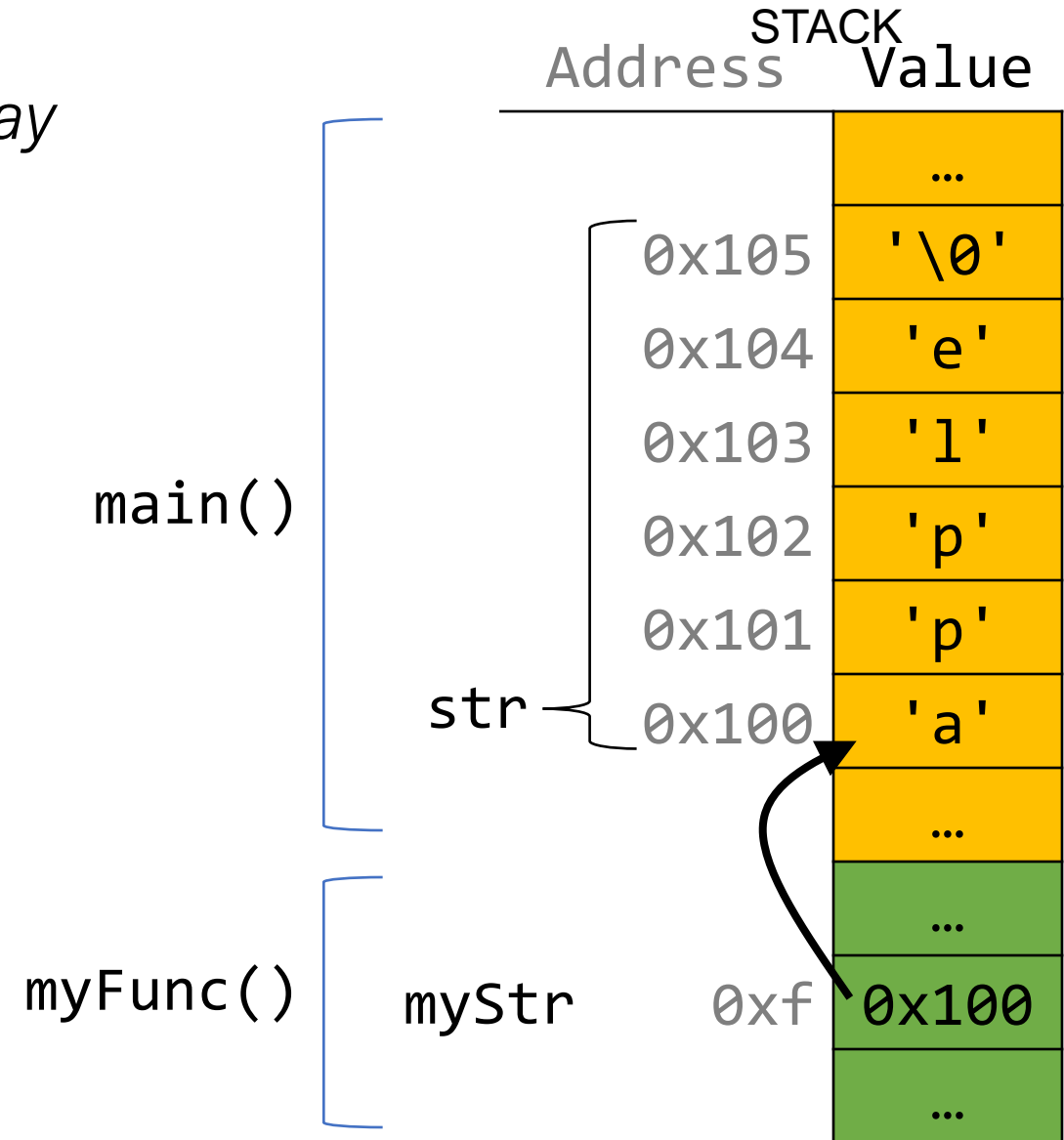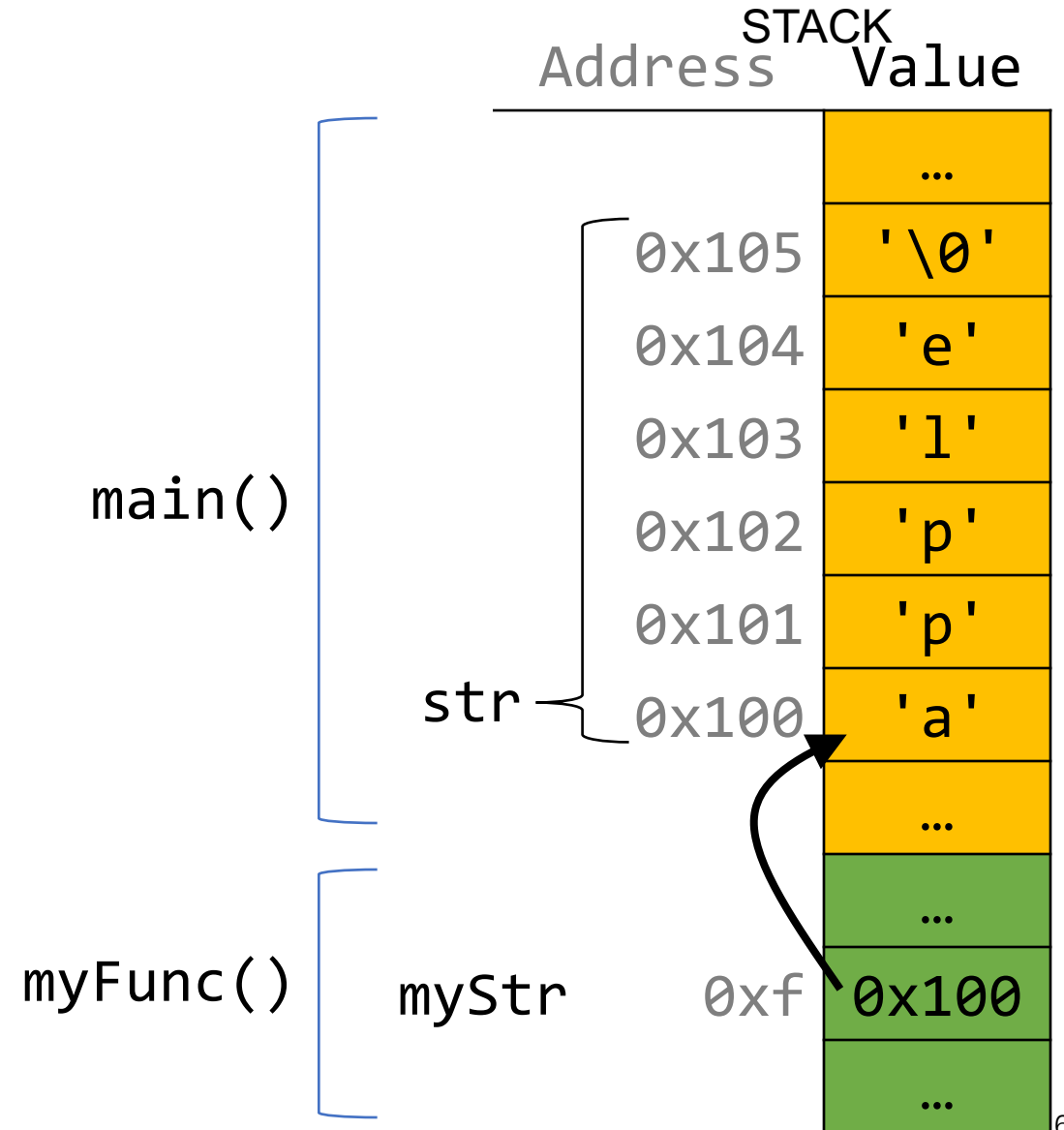
```c
void myFunc(char *myStr) {
    ...
}

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

61

Revisited: Strings as Parameters

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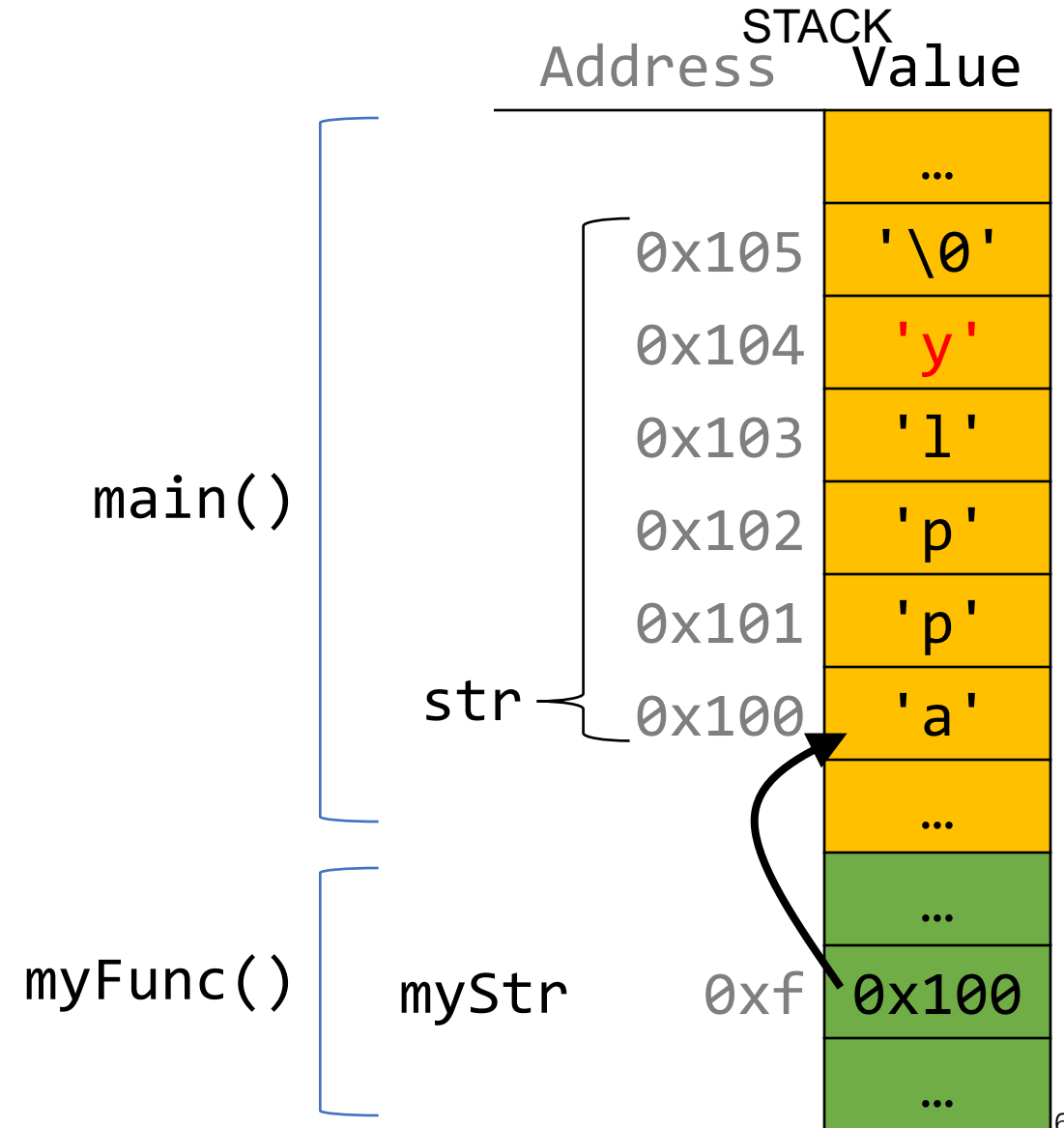
```c
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

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'
	…
	…
0xf	0x100
	…

main()

str

myFunc()   myStr

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
    ...
}
```

STACK

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

main()

str

myFunc() myStr

63

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);
}


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

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

Exercise 2

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

```c
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

- 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?

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

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

Exercise 3

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

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

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

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

Exercise 3

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

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

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

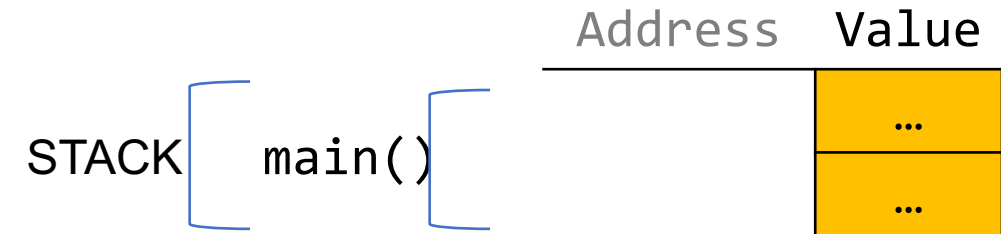This advances `skipSpace`'s own copy of the string pointer, not the instance in main.

Demo: Skip Spaces

`skip_spaces.c`

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
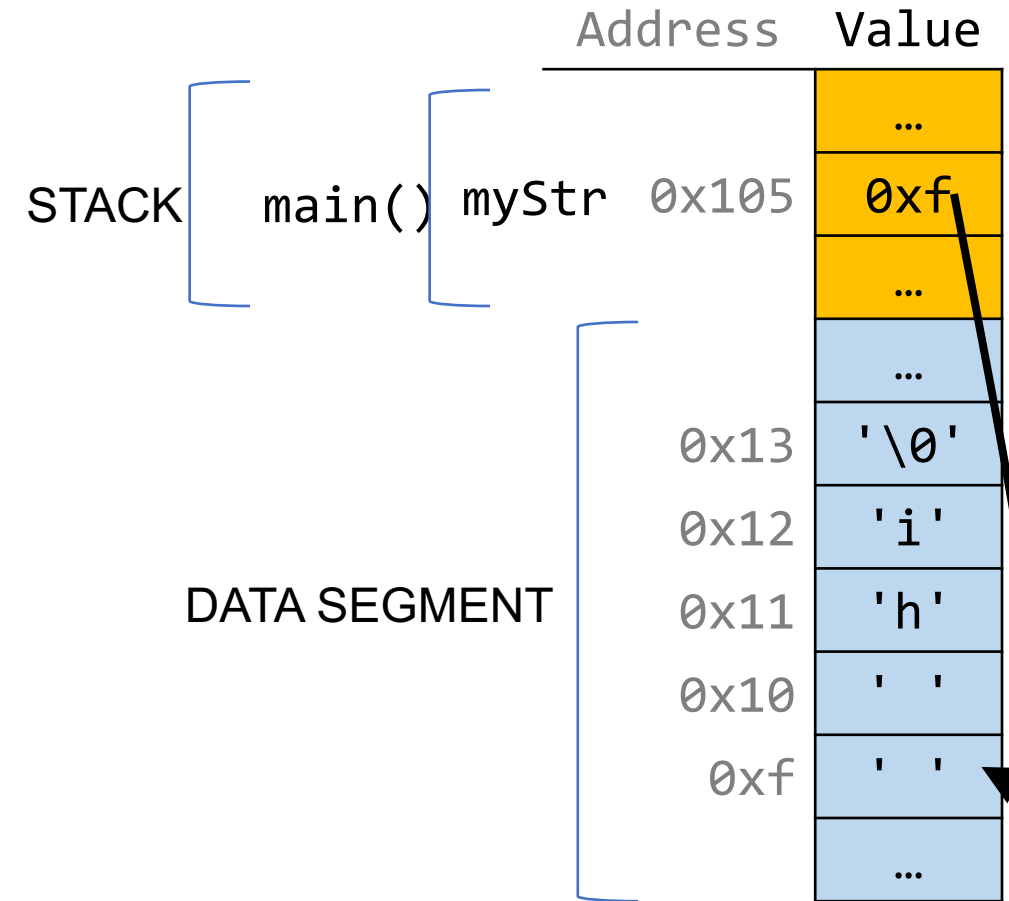
STACK   main()

…

…

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	' '
	…

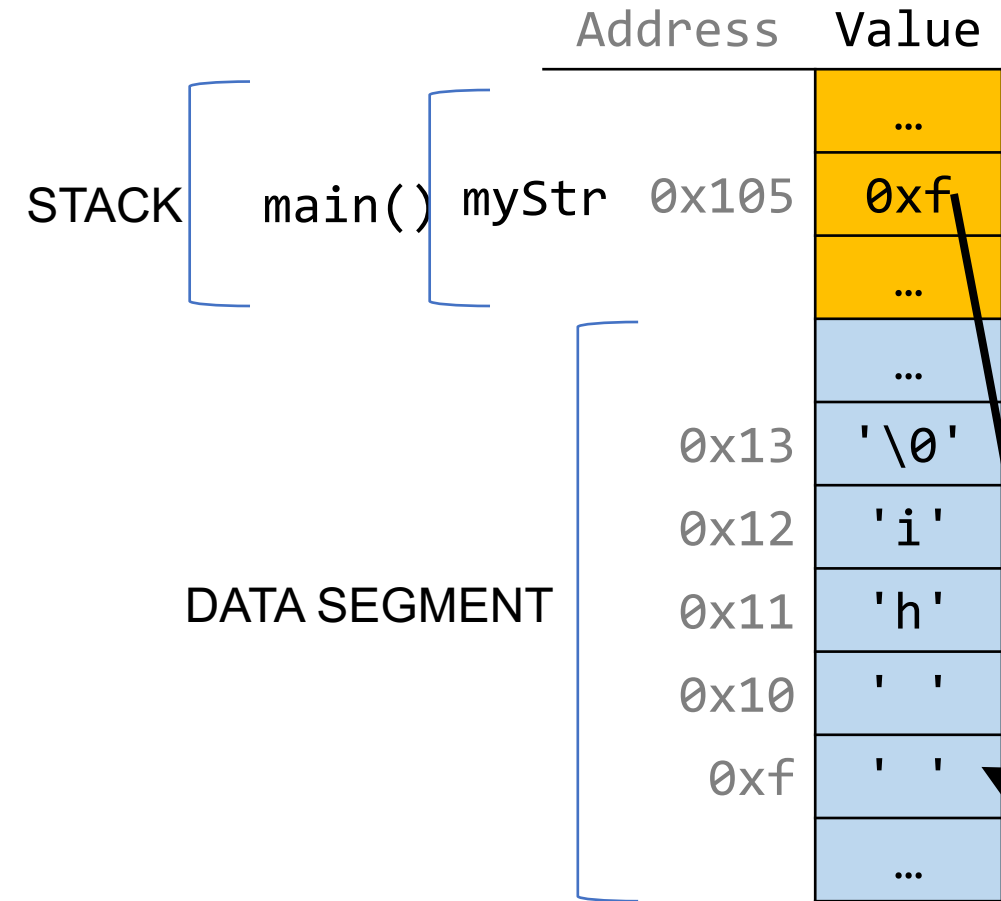STACK   main() myStr

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
	…
STACK main() myStr 0x105	0xf
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

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
	…
	…
skipSpaces() strPtr 0xf0	0x105
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

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;
}
```
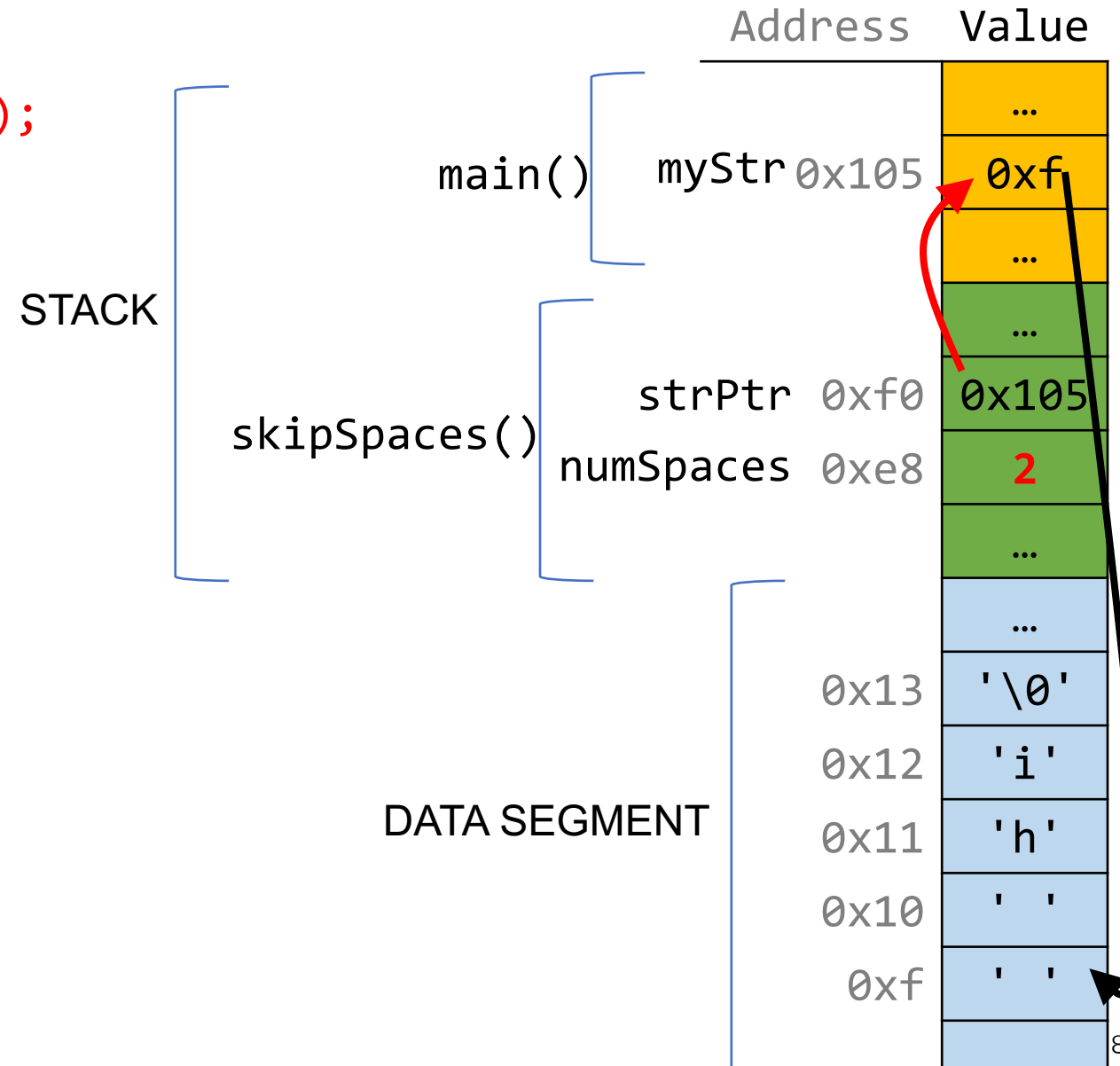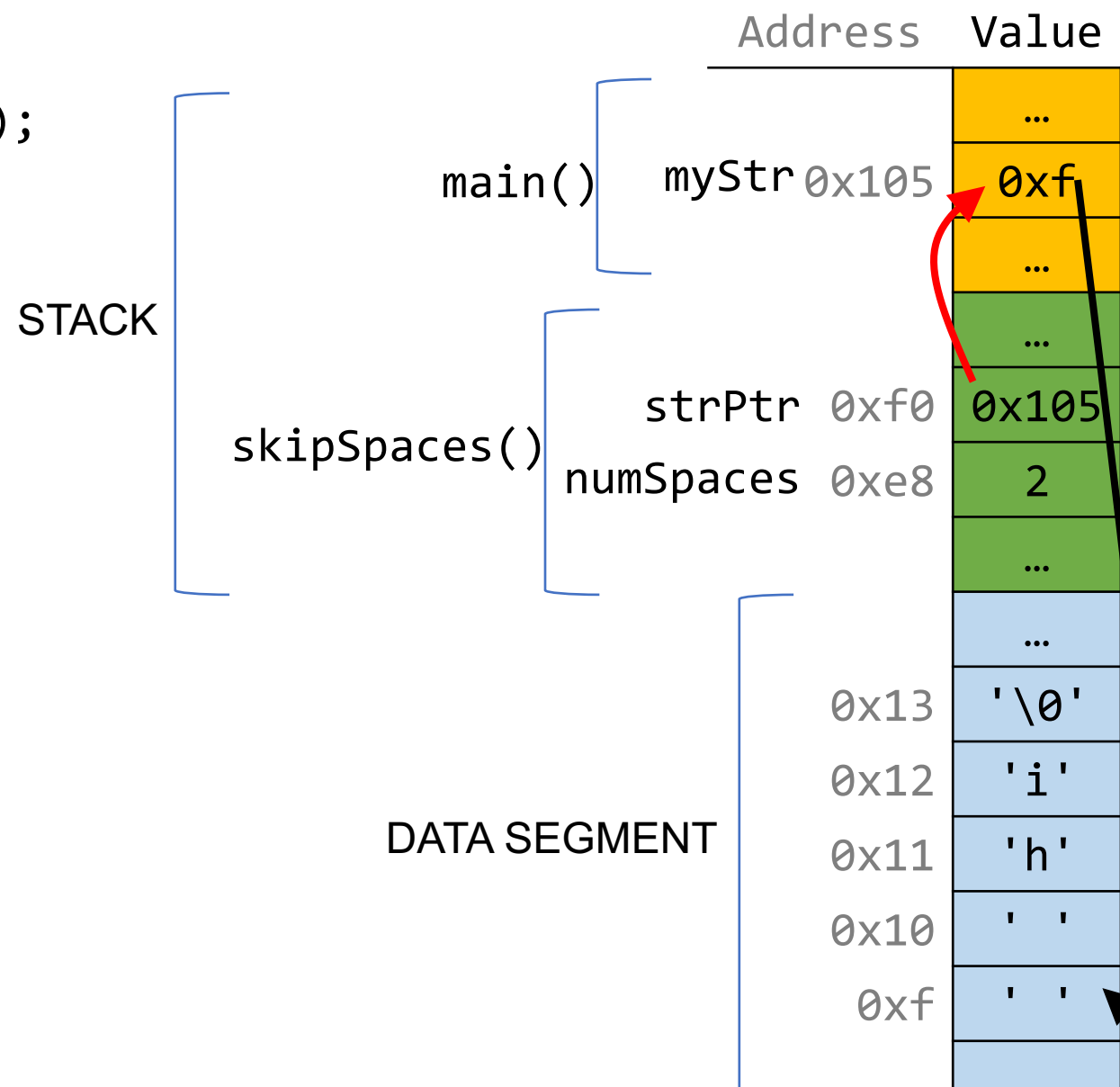
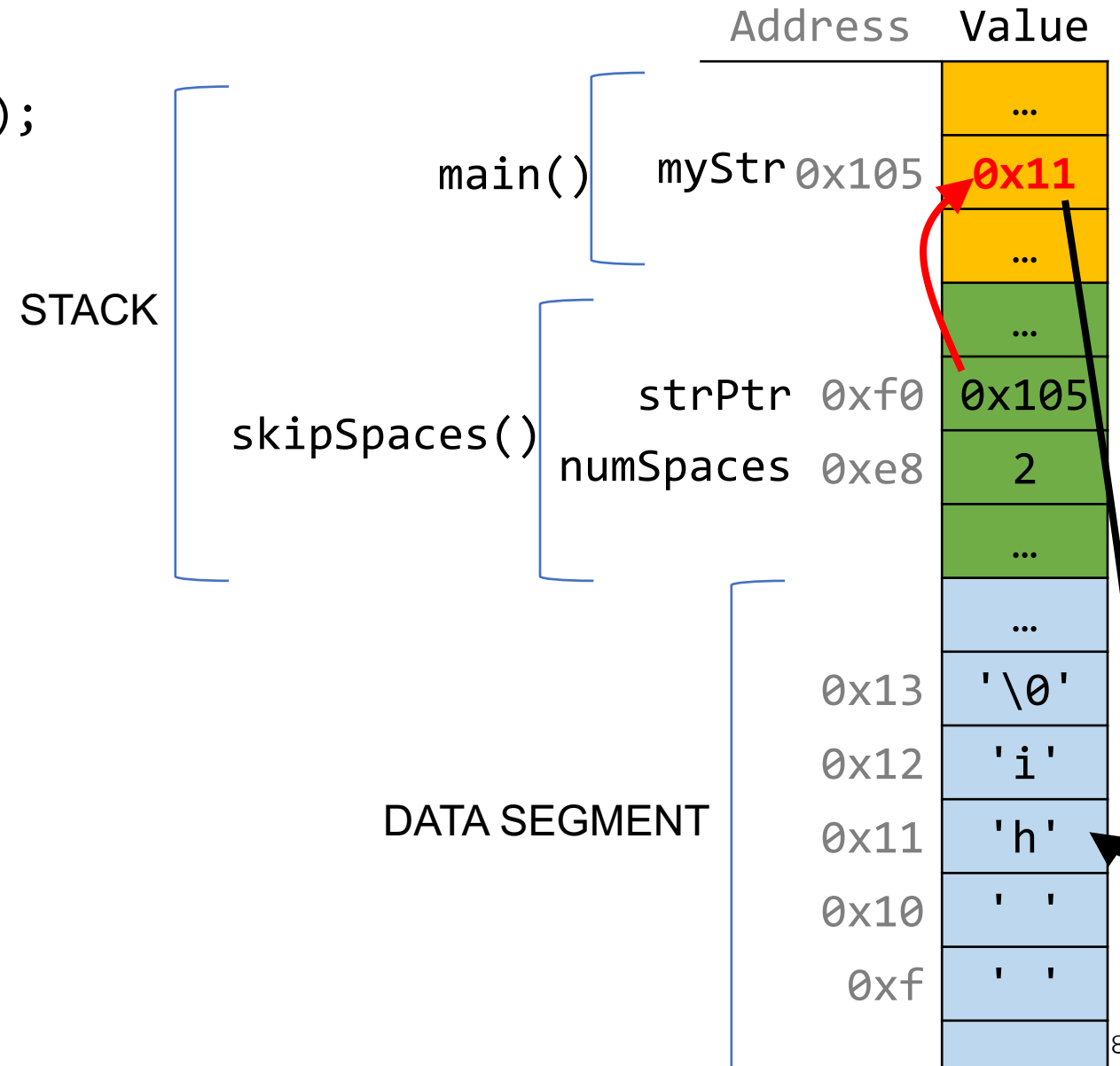|	Address	Value
		…
main() myStr	0x105	0xf
		…
		…
strPtr	0xf0	0x105
skipSpaces() numSpaces	0xe8	**2**
		…
		…
	0x13	'\0'
	0x12	'i'
	0x11	'h'
	0x10	' '
	0xf	' '

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;
}
```

STACK

main()

skipSpaces()

DATA SEGMENT

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

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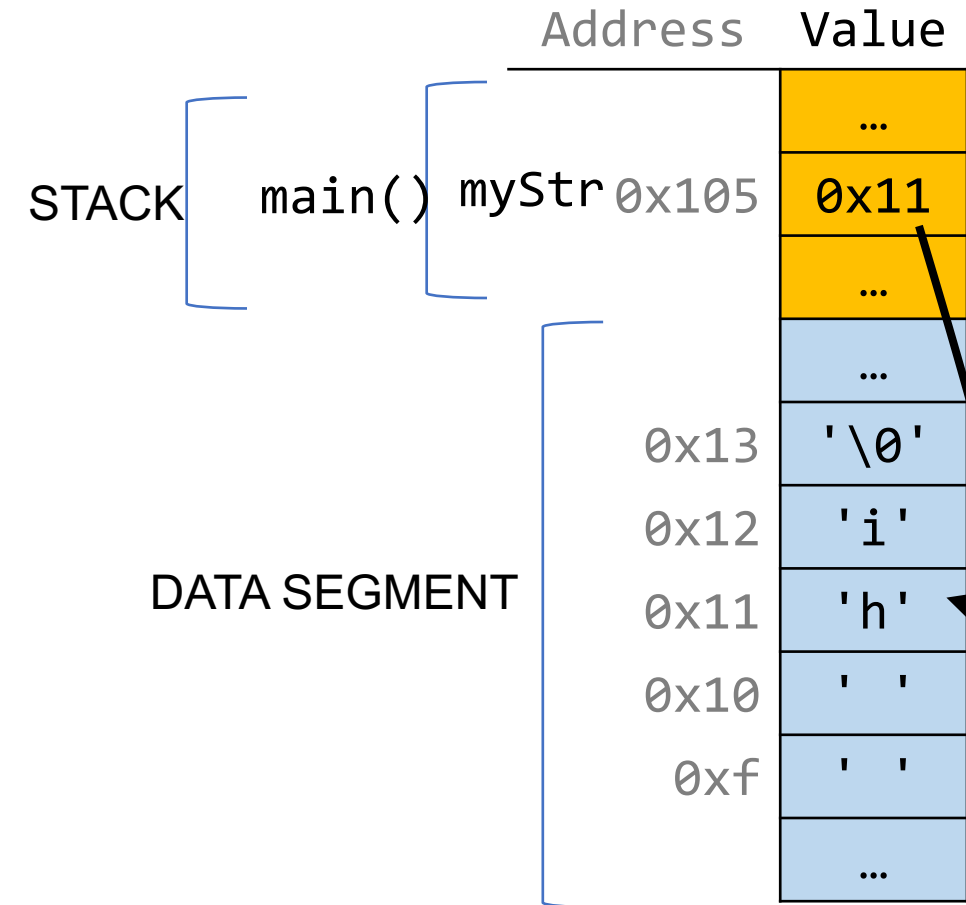
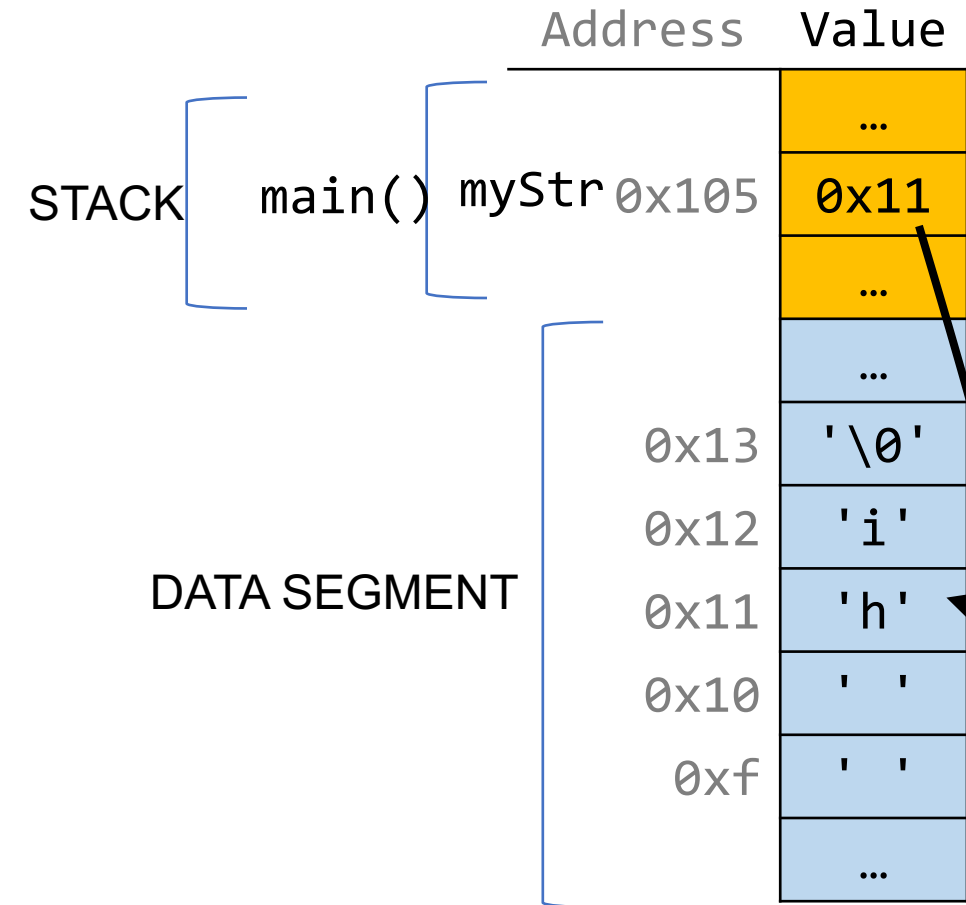| Address	Value
	…
myStr 0x105	0x11
	…
	…
strPtr 0xf0	0x105
numSpaces 0xe8	2
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '

STACK

main()

skipSpaces()

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

STACK   main() myStr 0x105   0x11
                              ...

                              ...
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;
}
```
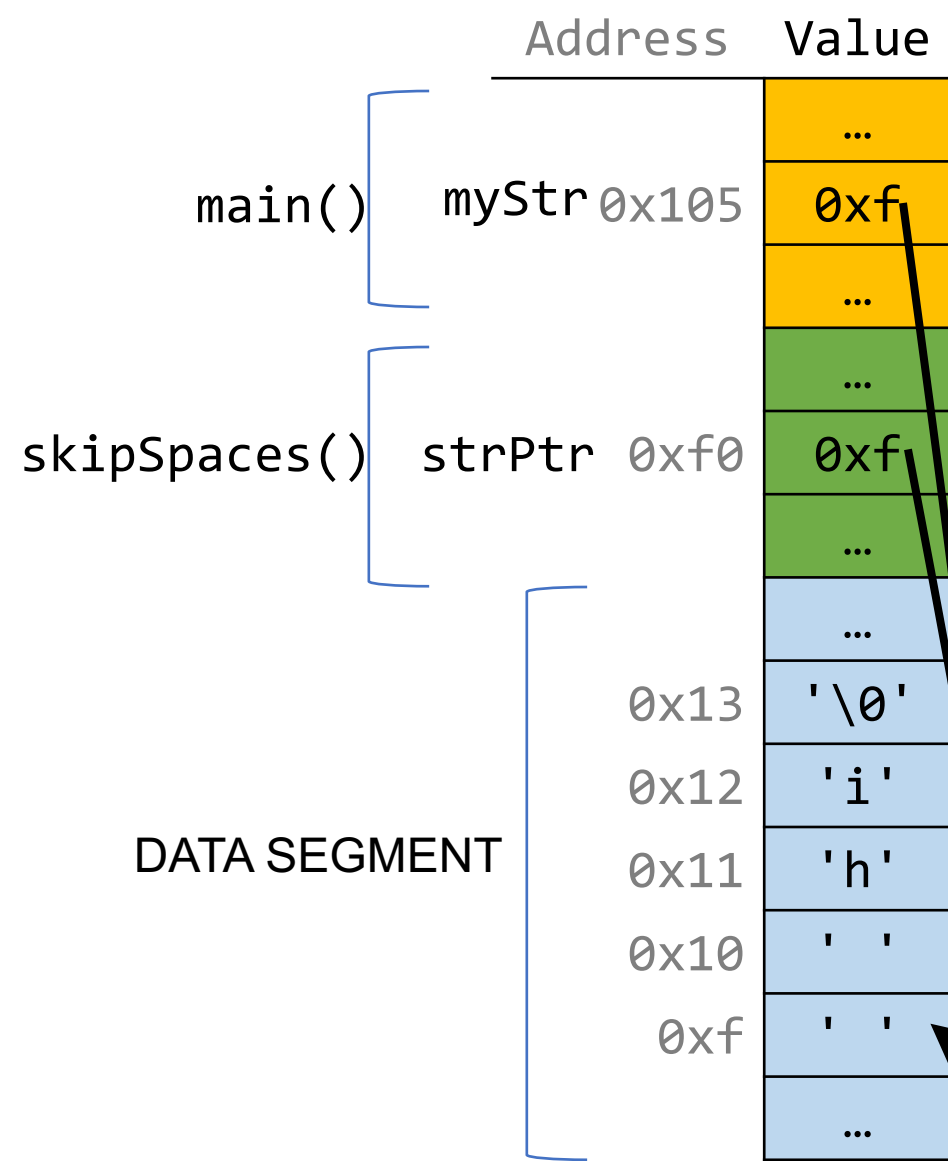
Address	Value
	…
0x105	0x11
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

STACK   main() myStr

DATA SEGMENT

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;
}
```
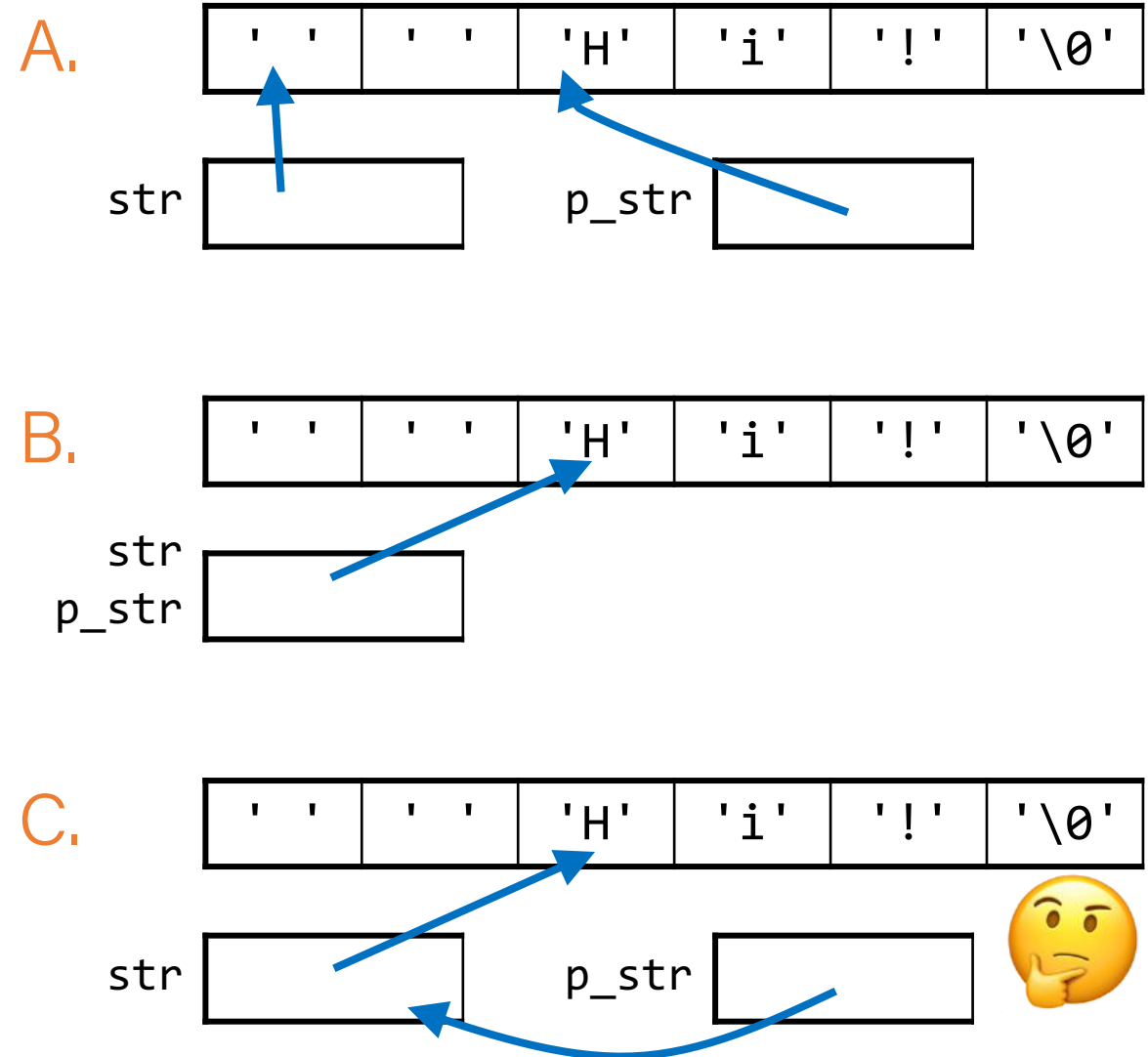
myFunc myFunc
//   hi

STACK

DATA SEGMENT

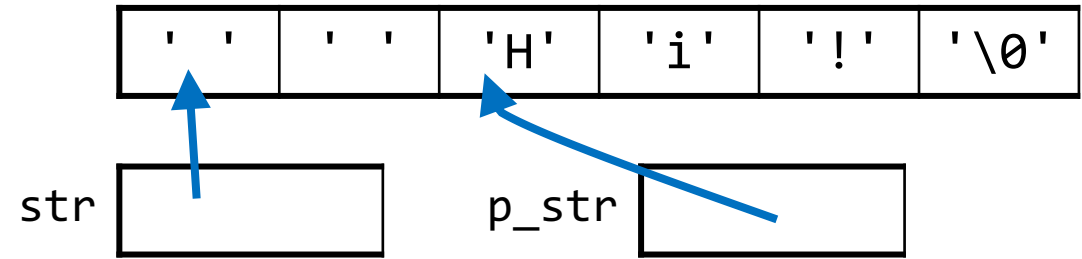| Address	Value
	…
main() myStr 0x105	0xf
	…
	…
skipSpaces() strPtr 0xf0	0xf
	…
	…
0x13	'\0'
0x12	'i'
0x11	'h'
0x10	' '
0xf	' '
	…

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
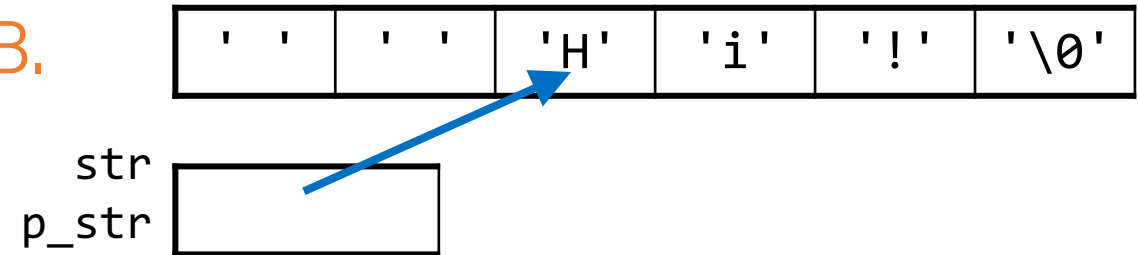
p_str

🤔

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`)?
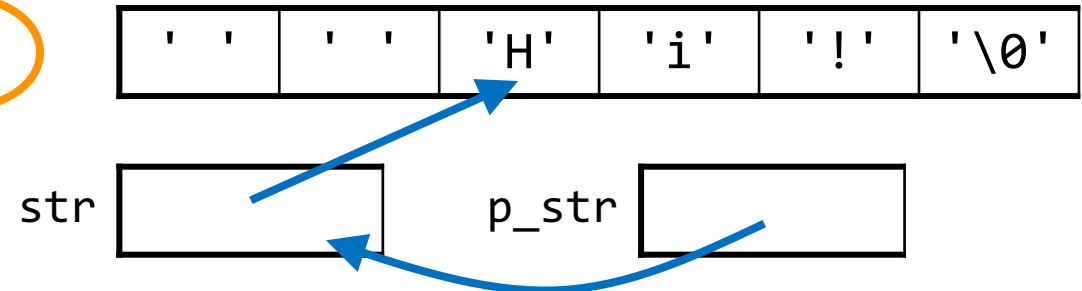
A.

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

str

p_str

B.

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

str
p_str

C.

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

str

p_str

Recap

- Strings in Memory (cont'd.)

- Pointers and Parameters

- Double Pointers

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