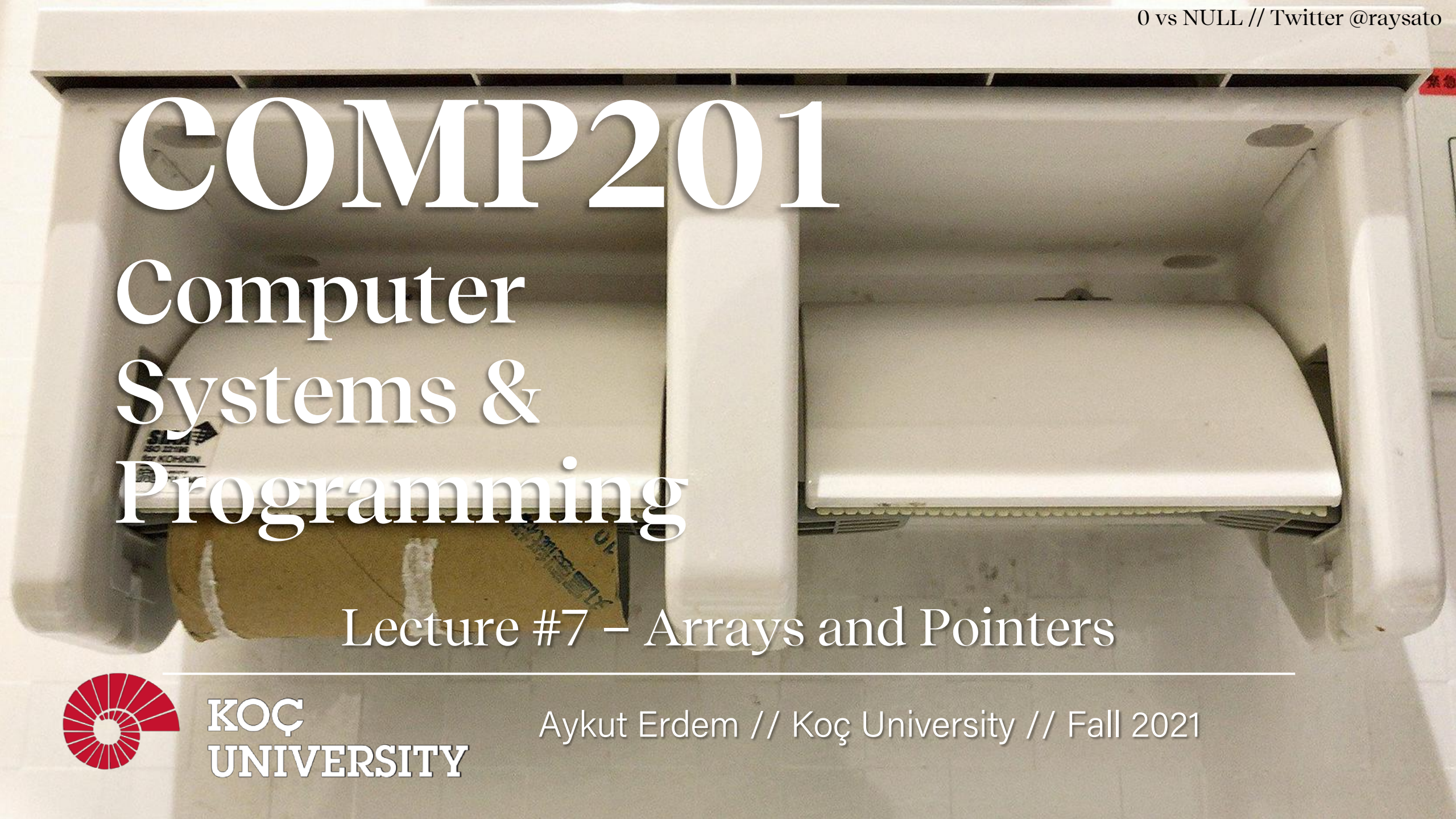
0 vs NULL // Twitter @raysato



COMP201

Computer Systems & Programming

Lecture #7 – Arrays and Pointers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Good news, everyone!

Image: Professor Farnsworth (Futurama)

- Assg1 is due Oct 21
- Assg2 will be out on Oct 21 (due Nov 4)

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

```
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
```

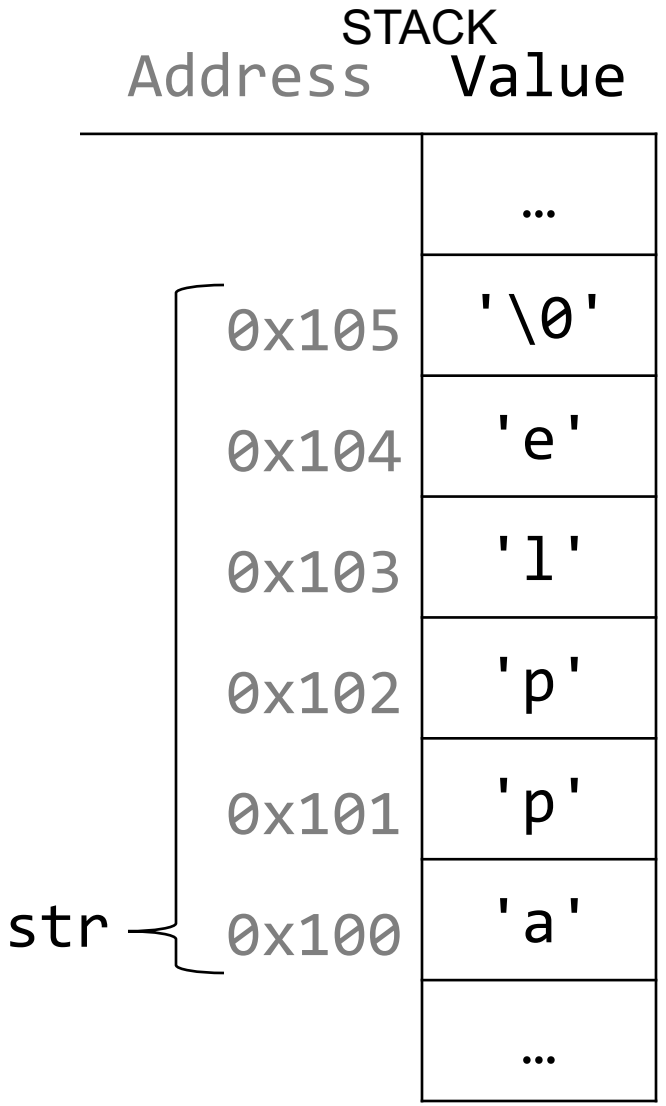
Recap: Strings In Memory

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");
```

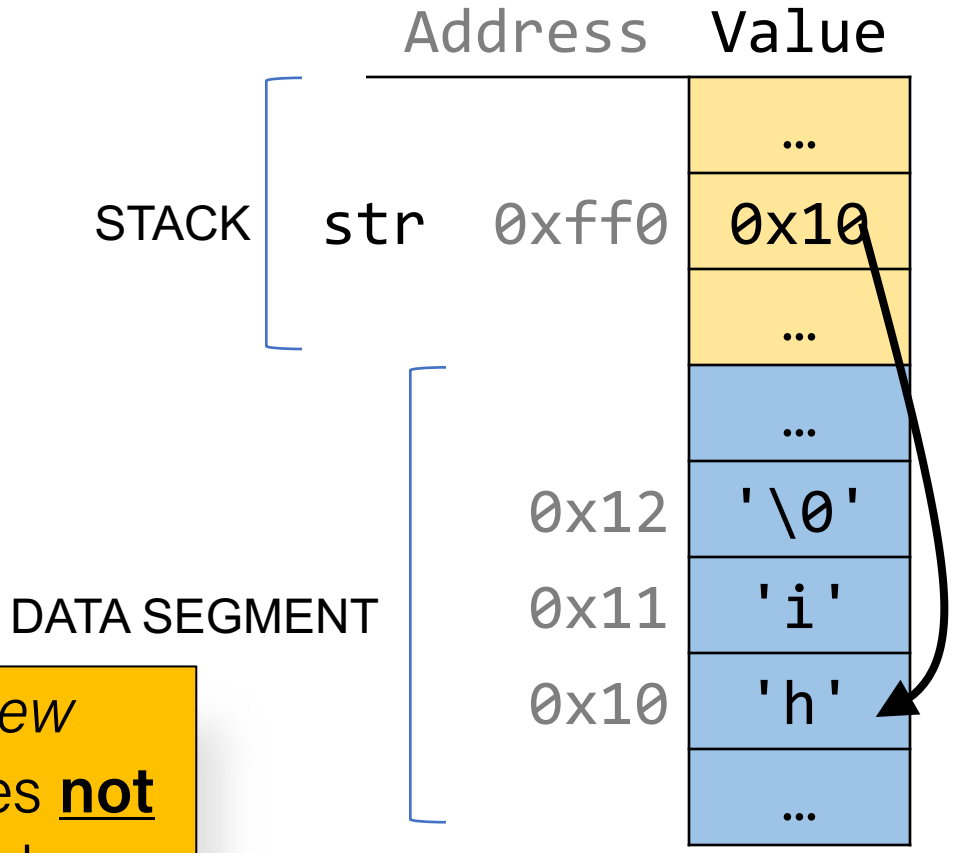


Recap: Recap: `char *`

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 **not** apply for making a `char *` that points to an existing stack string.

Plan for Today

- Strings in Memory (cont'd.)
- 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

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

Memory Locations

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

```
char myStr[6];
```

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

Memory Locations

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

```
char *myStr = "Hi";
```

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

Memory Locations

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

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

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

Memory Locations

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

```
char *otherStr = "Hi";
char *myStr = otherStr;
```

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

Memory Locations

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

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

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

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

Memory Locations

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

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

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

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

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

char *

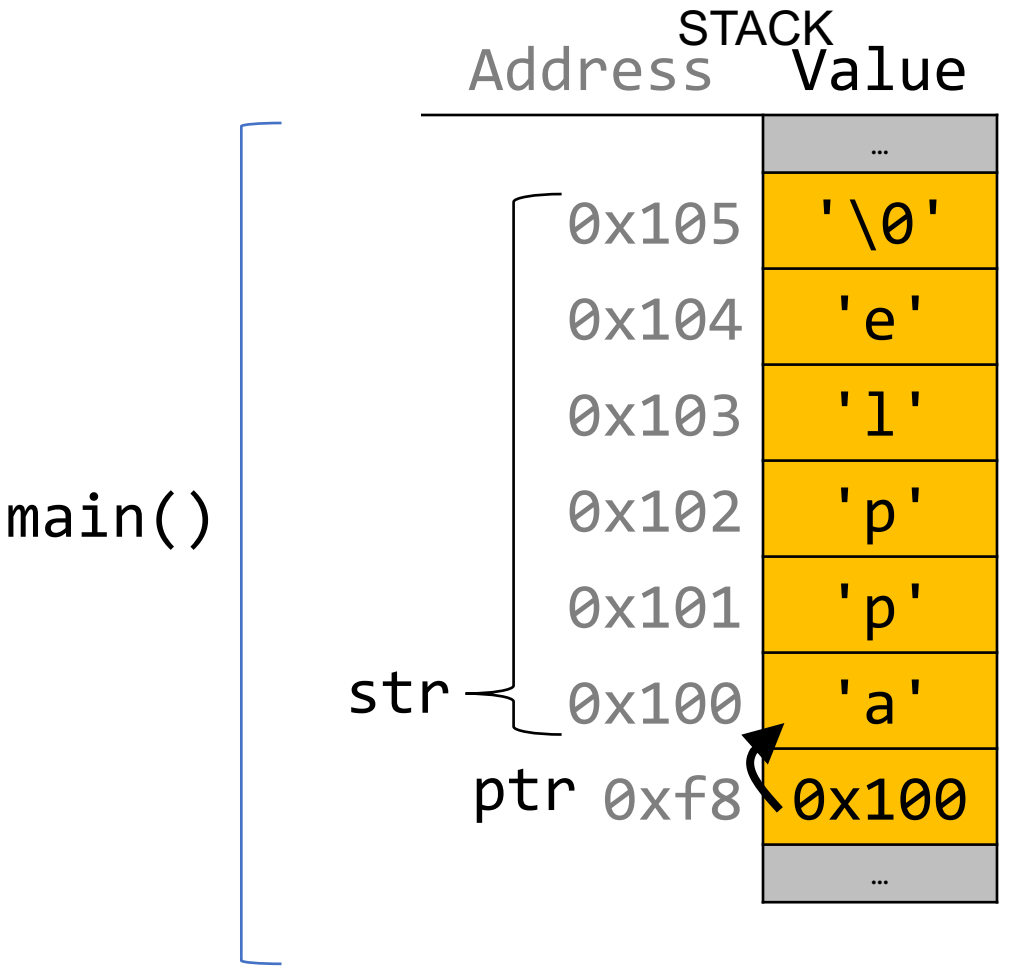
A **char *** variable refers to a single character. We can reassign an existing **char *** pointer to be equal to another **char *** pointer.

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

Arrays and Pointers

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

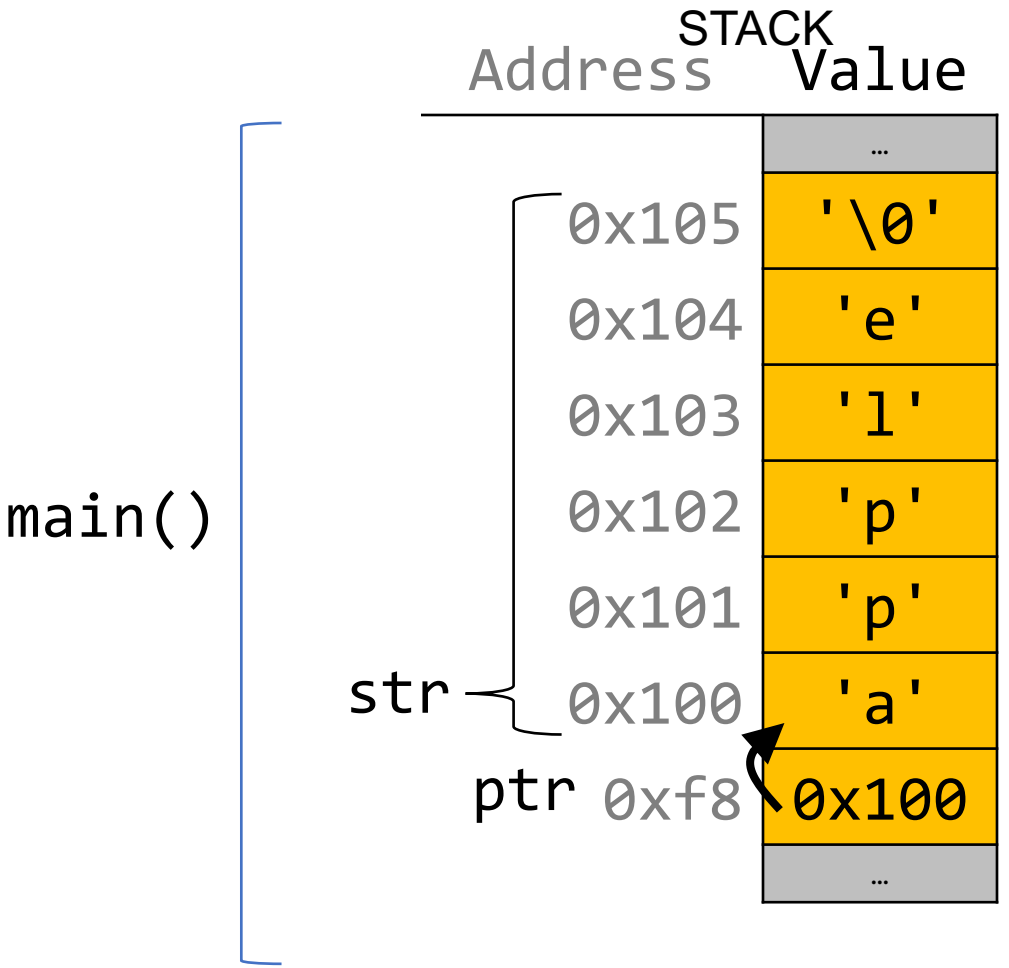
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;
      ...
}
```



Arrays and Pointers

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

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

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

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



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

Pointer Arithmetic

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

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

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

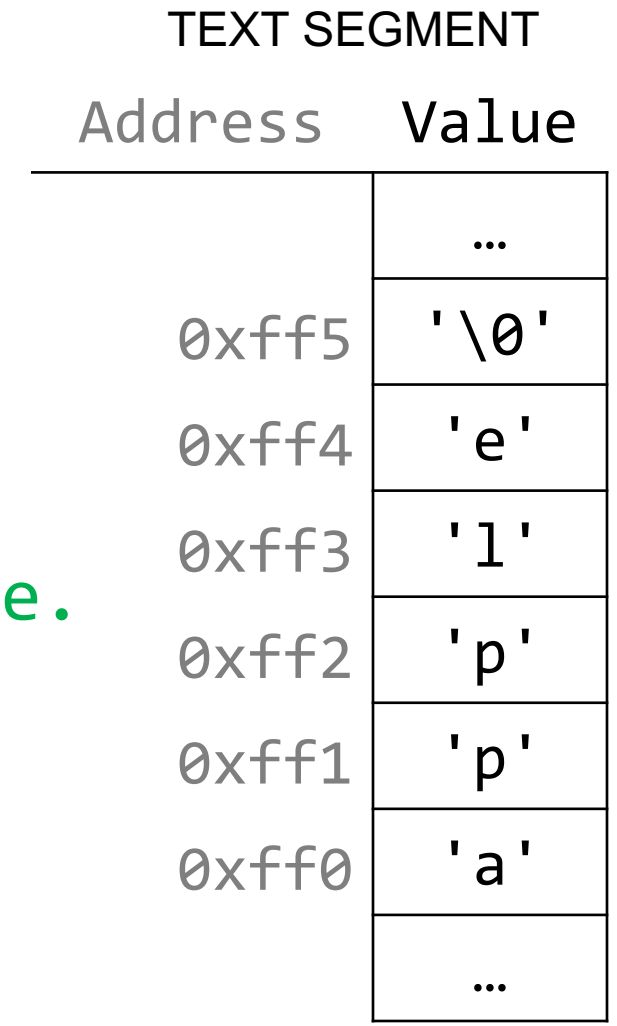
TEXT SEGMENT

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

char *

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

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

// both of these add three places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff3.
char thirdLetter = str[3];        // 'l'
char thirdLetter = *(str + 3);    // 'l'
```

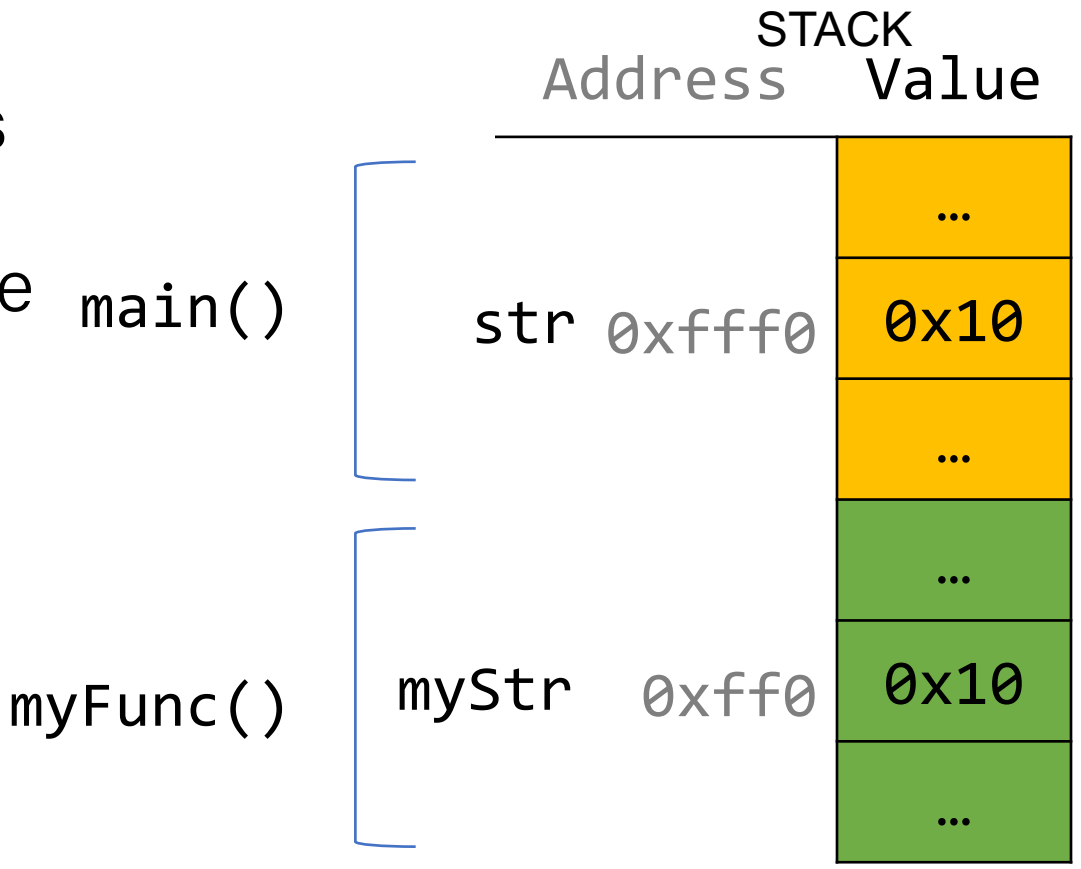


String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.

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.

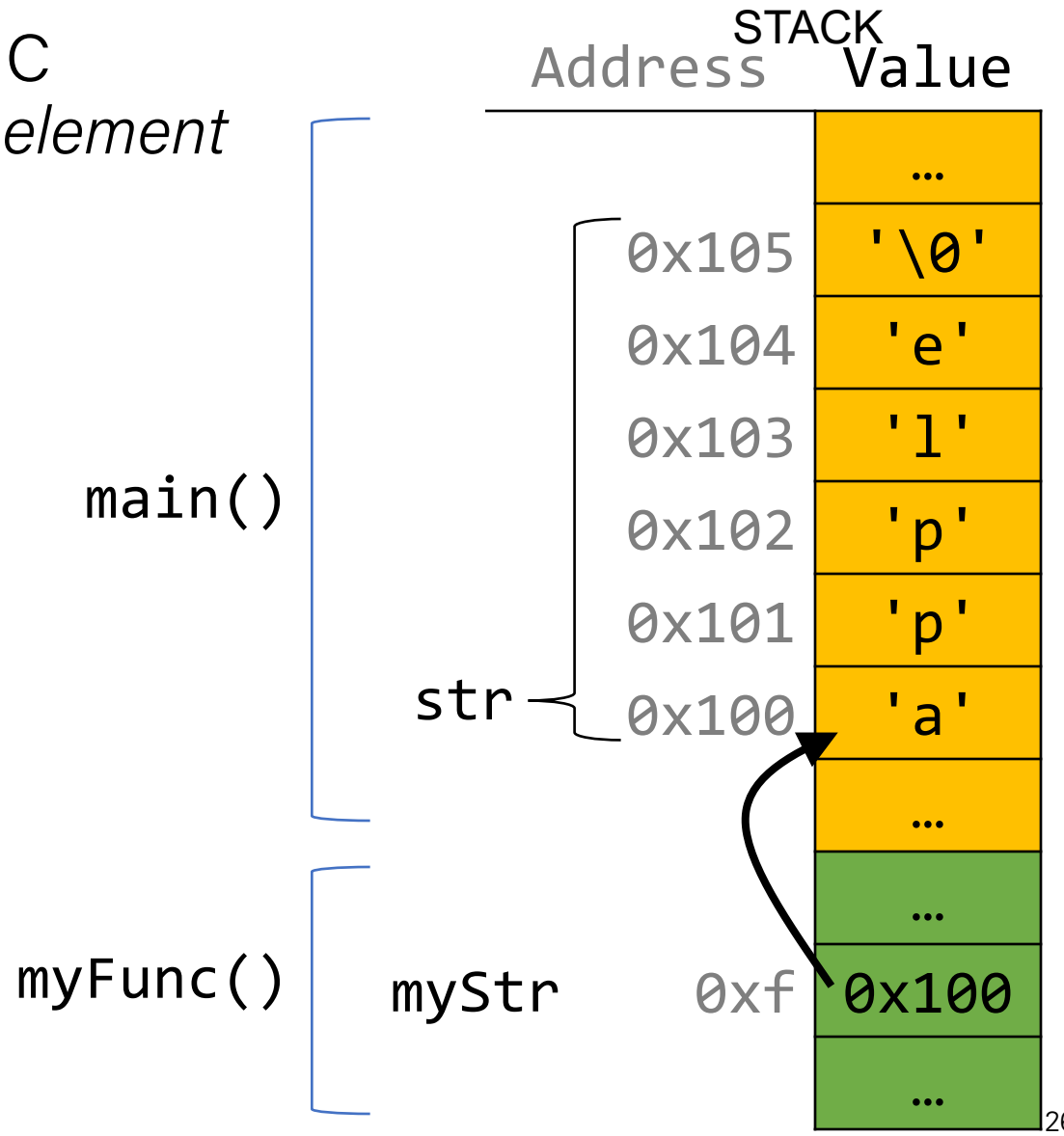
```
void myFunc(char *myStr) {
      ...
}

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.

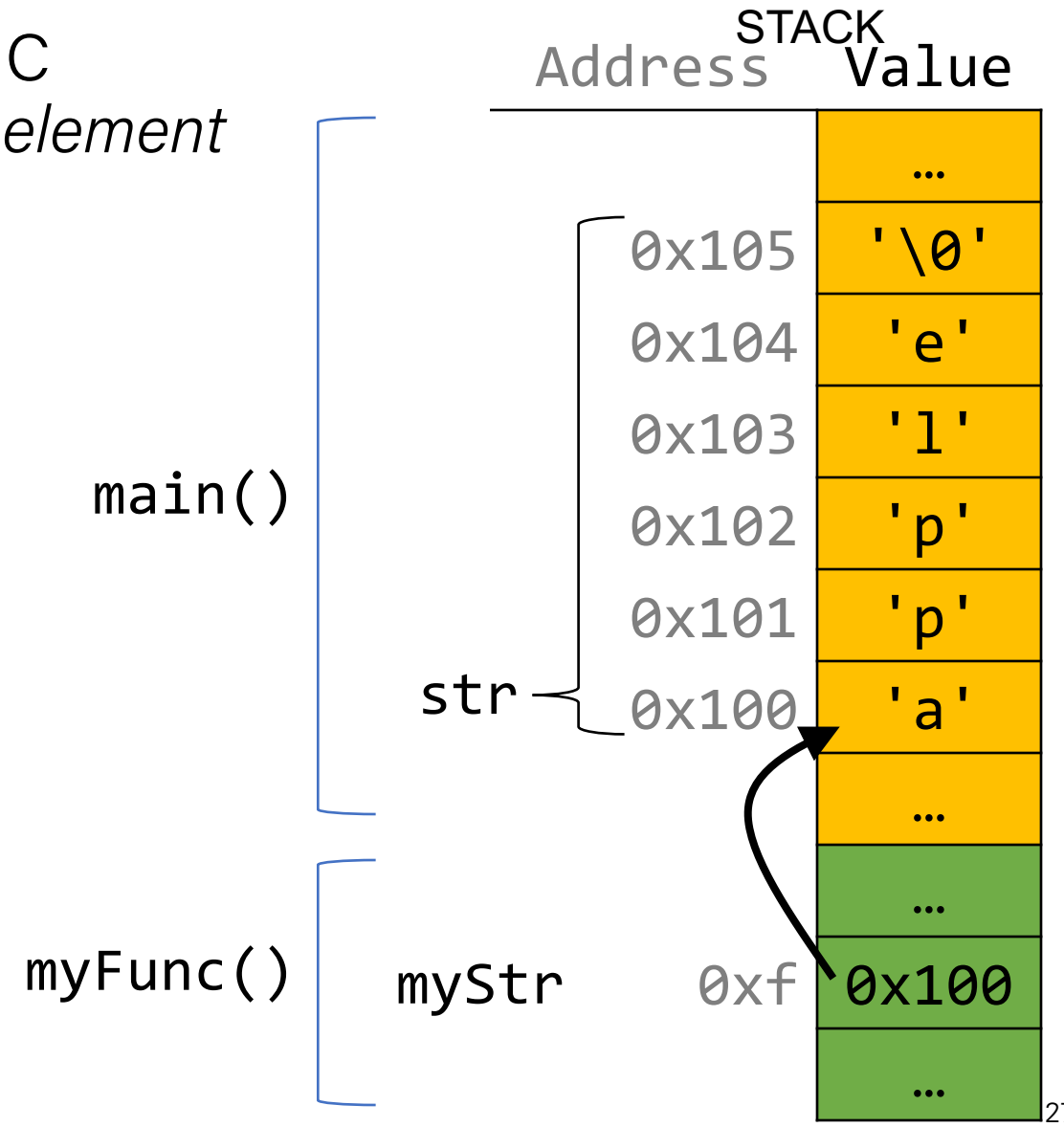
```
void myFunc(char *myStr) {
      ...
}

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.

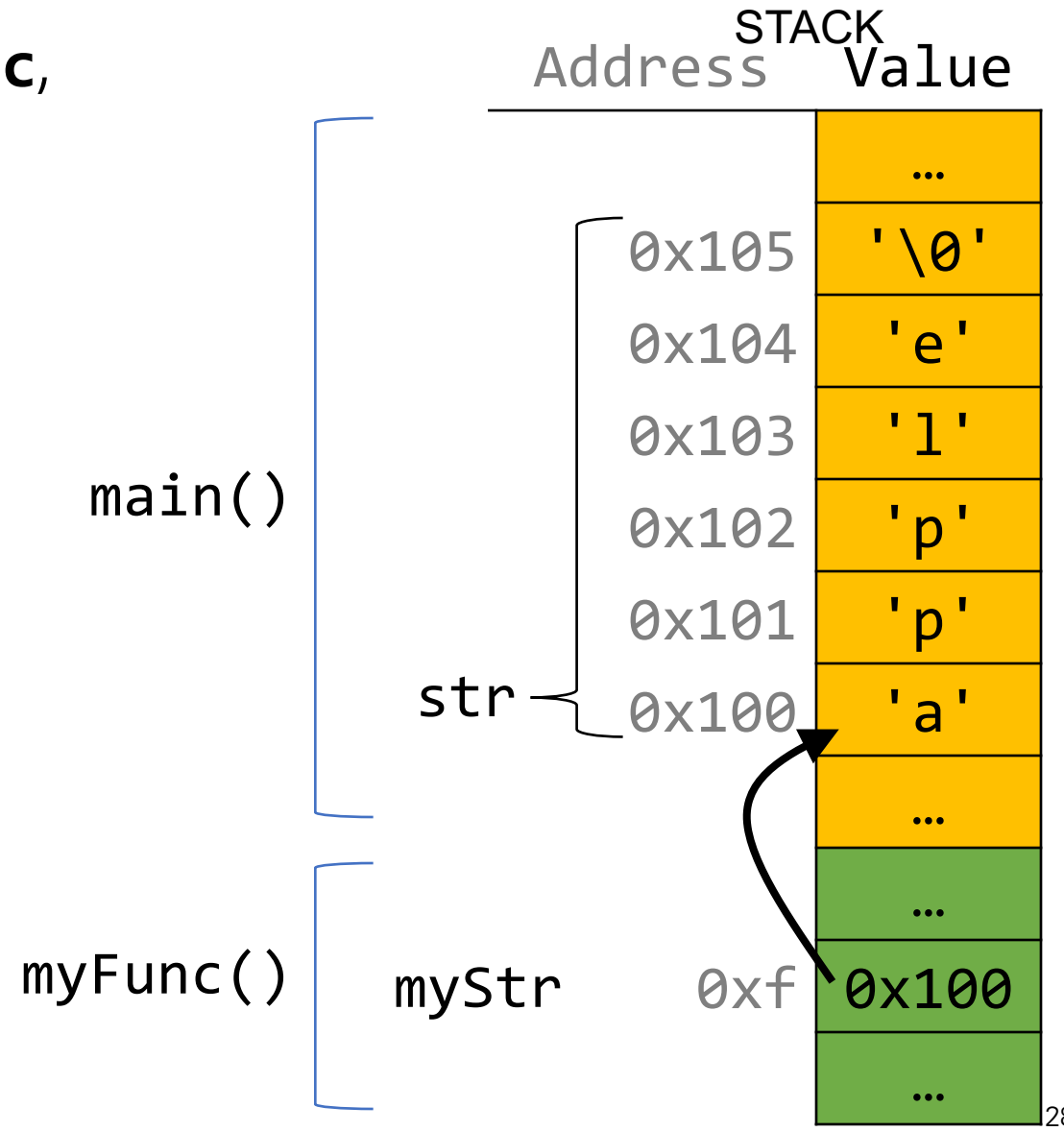
```
void myFunc(char *myStr) {
      ...
}

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**!

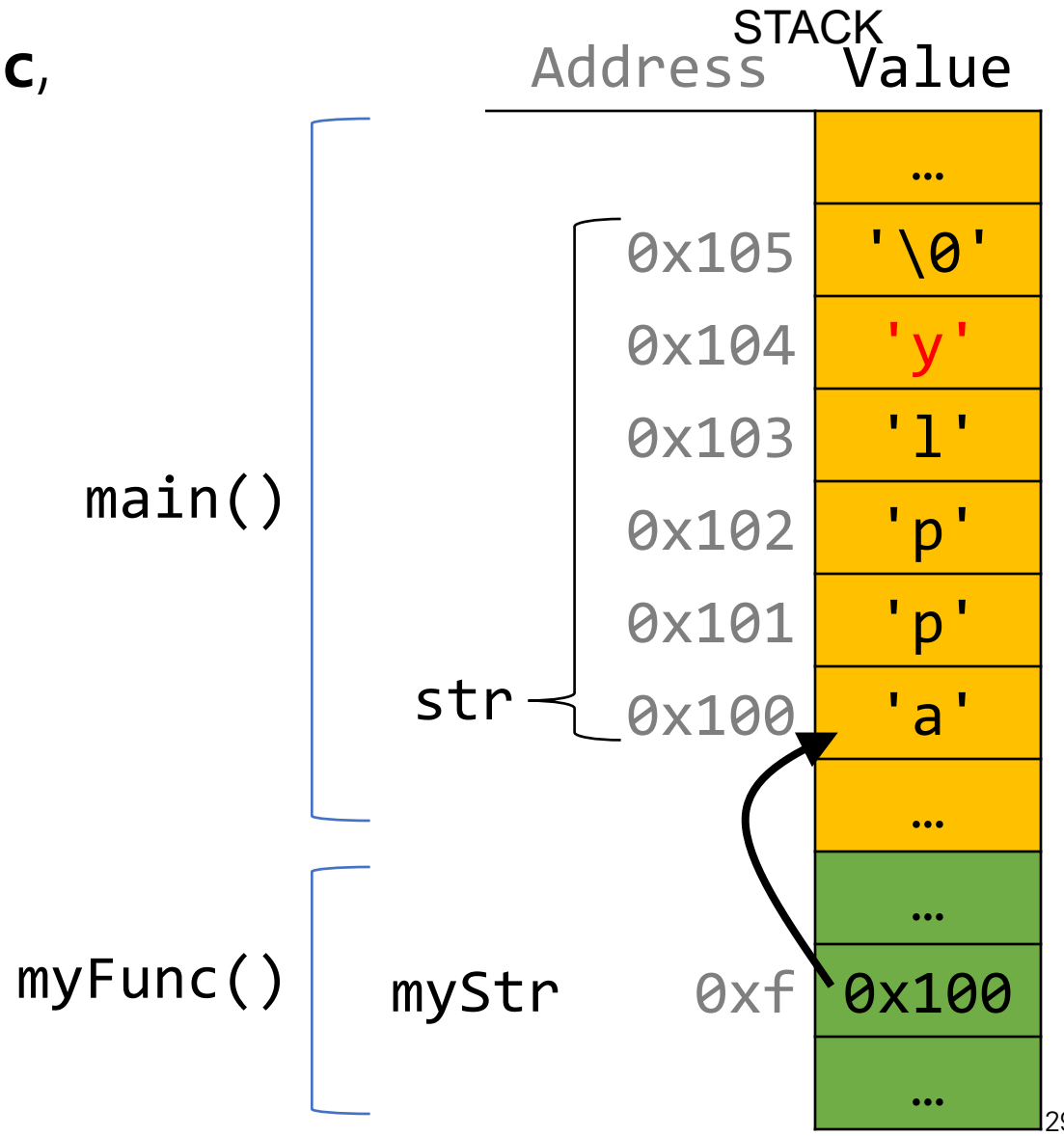
```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

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");
```

`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";
```

`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;
```

`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;
```

`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
- Arrays of Pointers
- Pointer Arithmetic

C Parameters

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

```
void myFunction(int x) {
    …
}

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

C Parameters

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

```
void myFunction(int *x) {
    …
}

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

C Parameters

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

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

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

C Parameters

If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

C Parameters

If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

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

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

C Parameters

If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

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

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

C Parameters

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

Do I care about modifying *this* instance of my data? If so, I need to pass where that instance lives, as a parameter, so it can be modified.

Pointers

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

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

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

Pointers

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

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

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

Pointers

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

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

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

Pointers

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

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

Pointers

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

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

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

`char *`

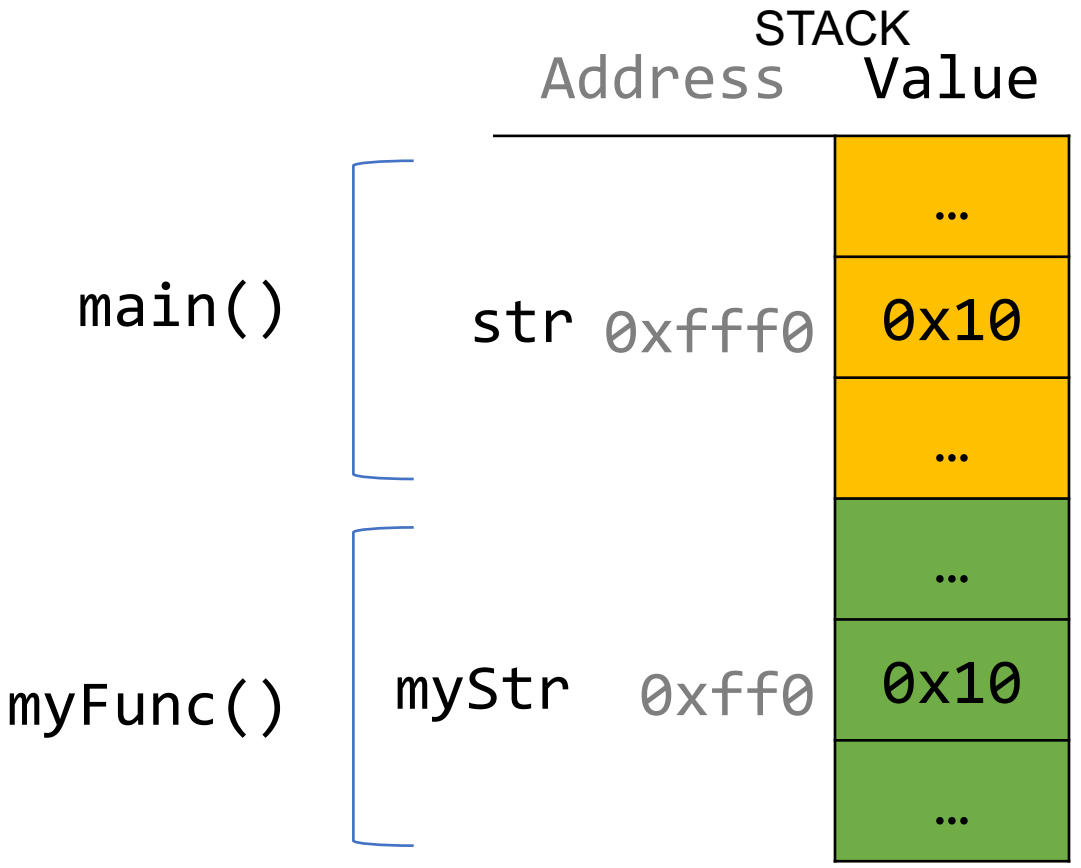
- A `char *` is technically a pointer to a **single character**.
- We commonly use `char *` as string by having the character it points to be followed by more characters and ultimately a null terminator.
- A `char *` could also just point to a single character (not a string).

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

Strings as Parameters

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

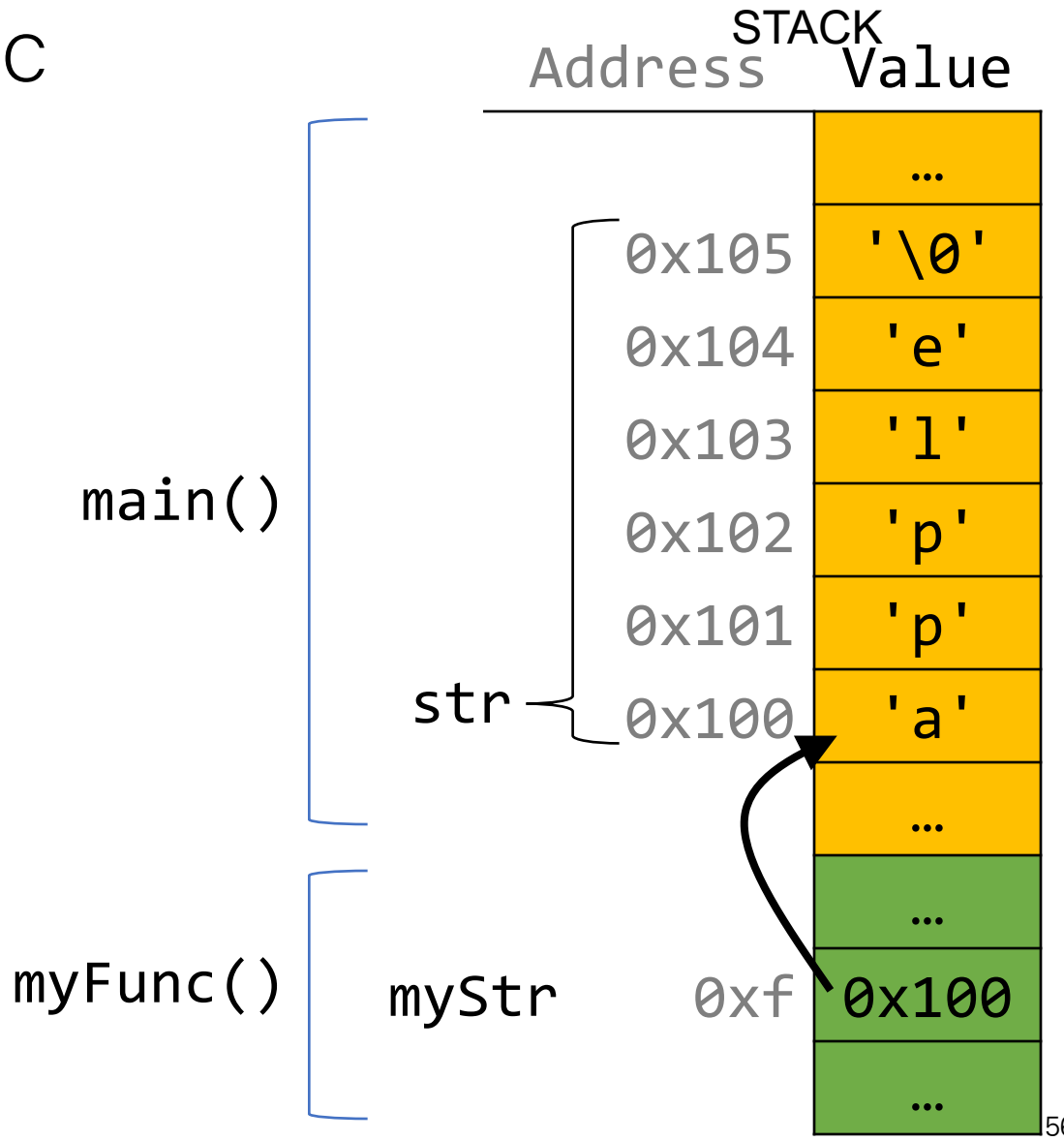
```
void myFunc(char *myStr) {
      ...
}

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



Strings as Parameters

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

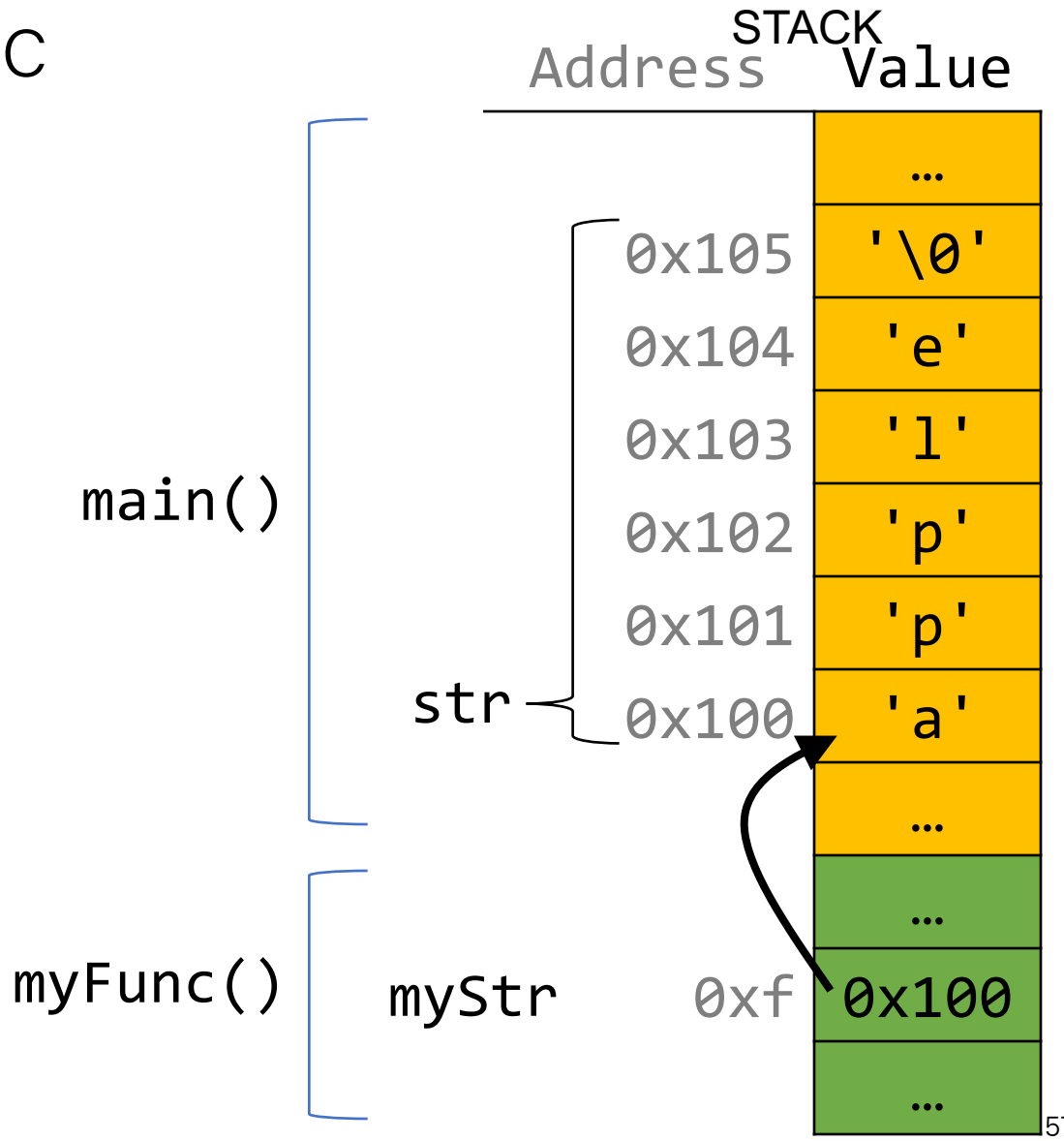
```
void myFunc(char *myStr) {
      ...
}

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



Strings as Parameters

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

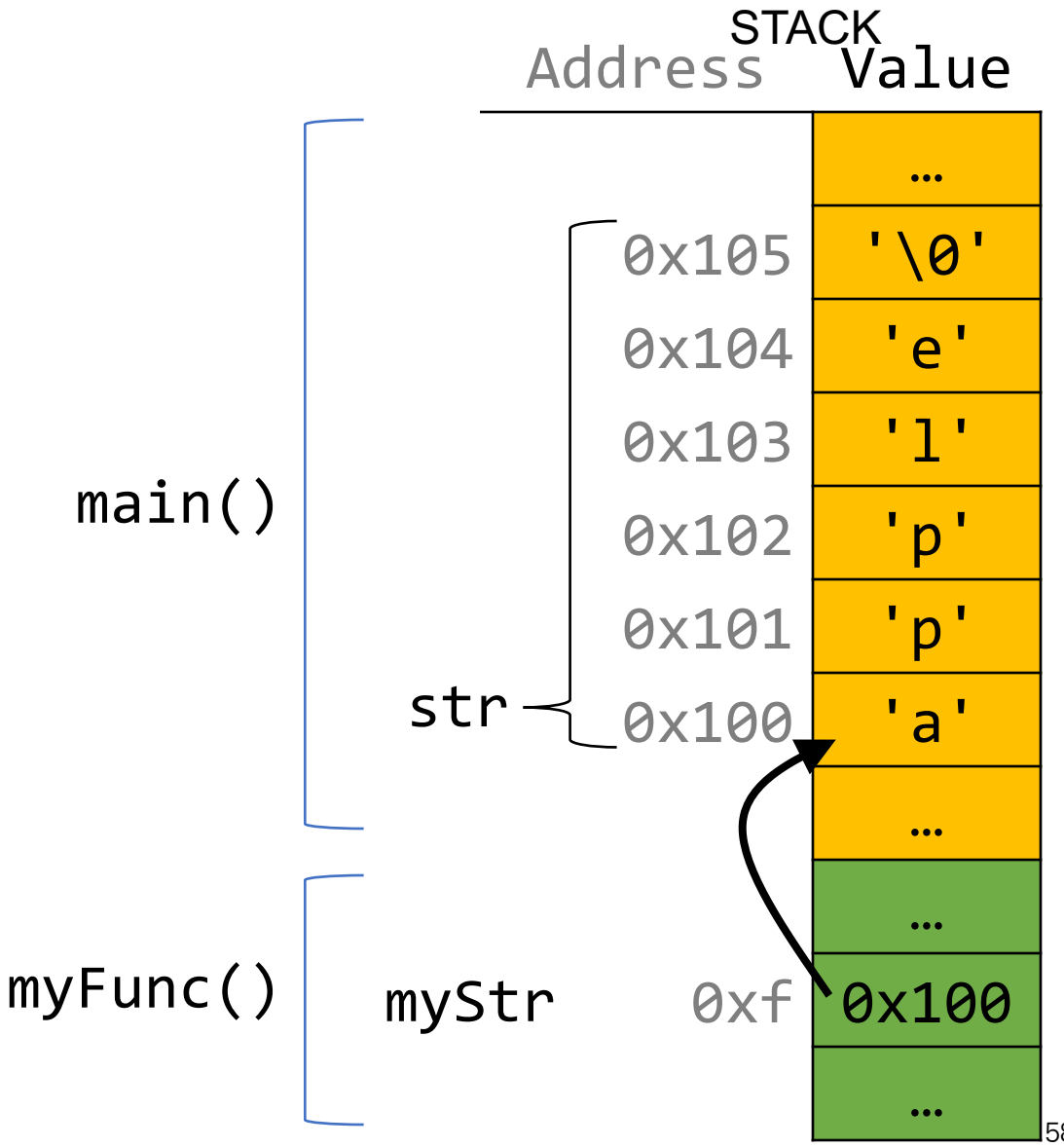
```
void myFunc(char *myStr) {
      ...
}

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



Strings as Parameters

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

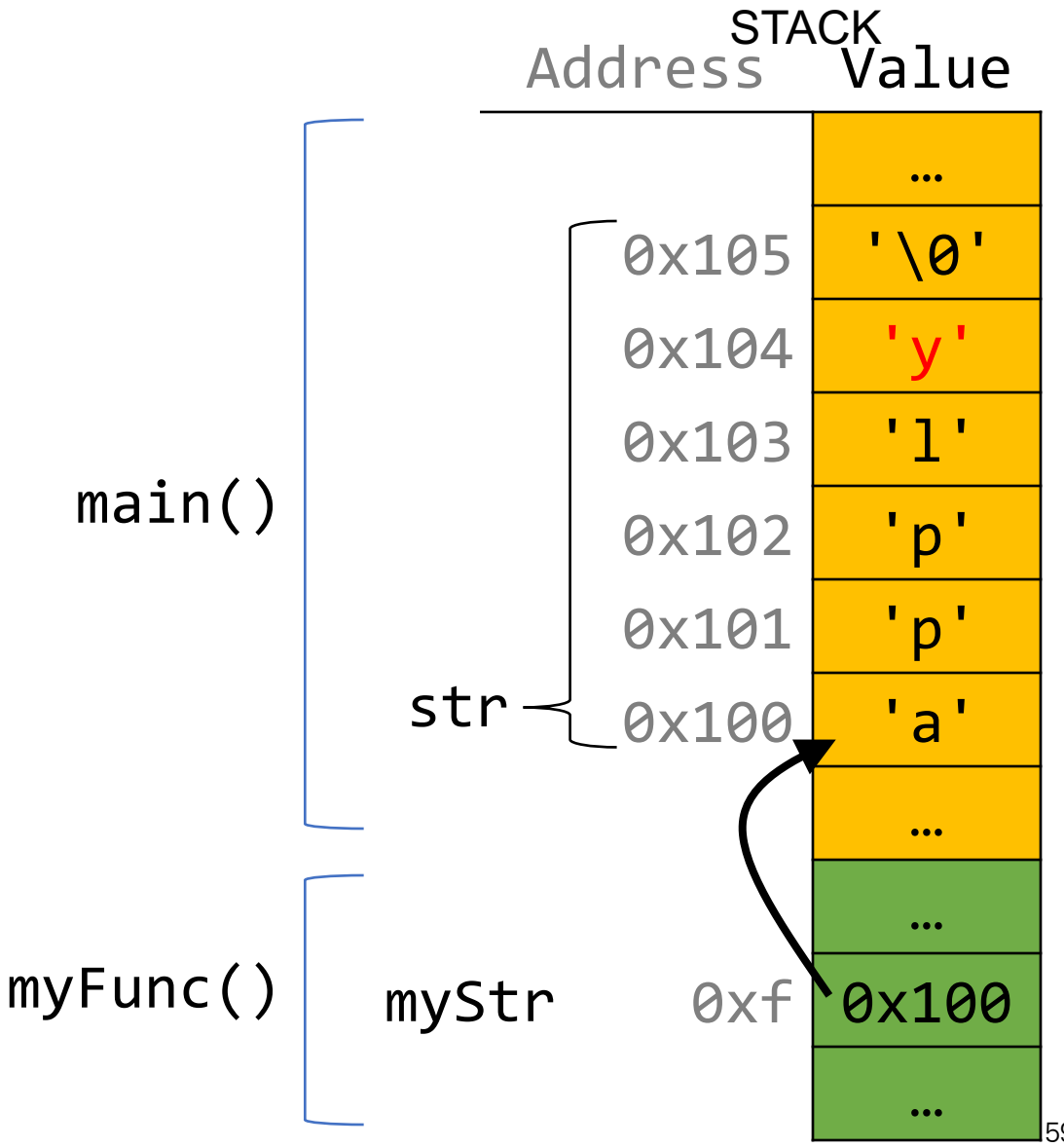
```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

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



Strings as Parameters

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

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

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



Exercise 1

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

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

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

Exercise 1

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

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

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?

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.

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

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

- Strings in Memory (cont'd.)
- 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?

```
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"
}
```

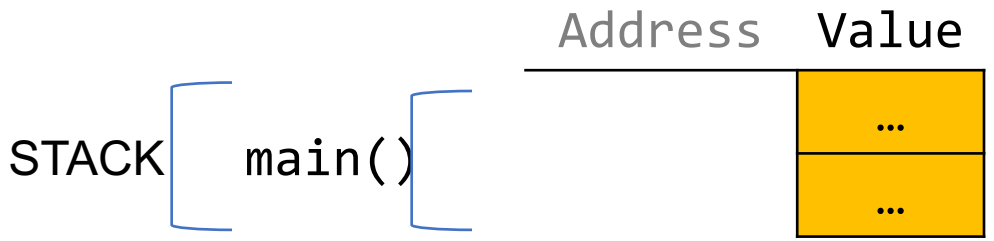
This advances `skipSpace`'s own copy of the string pointer, not the instance in main.

Demo: Skip Spaces

`skip_spaces.c`

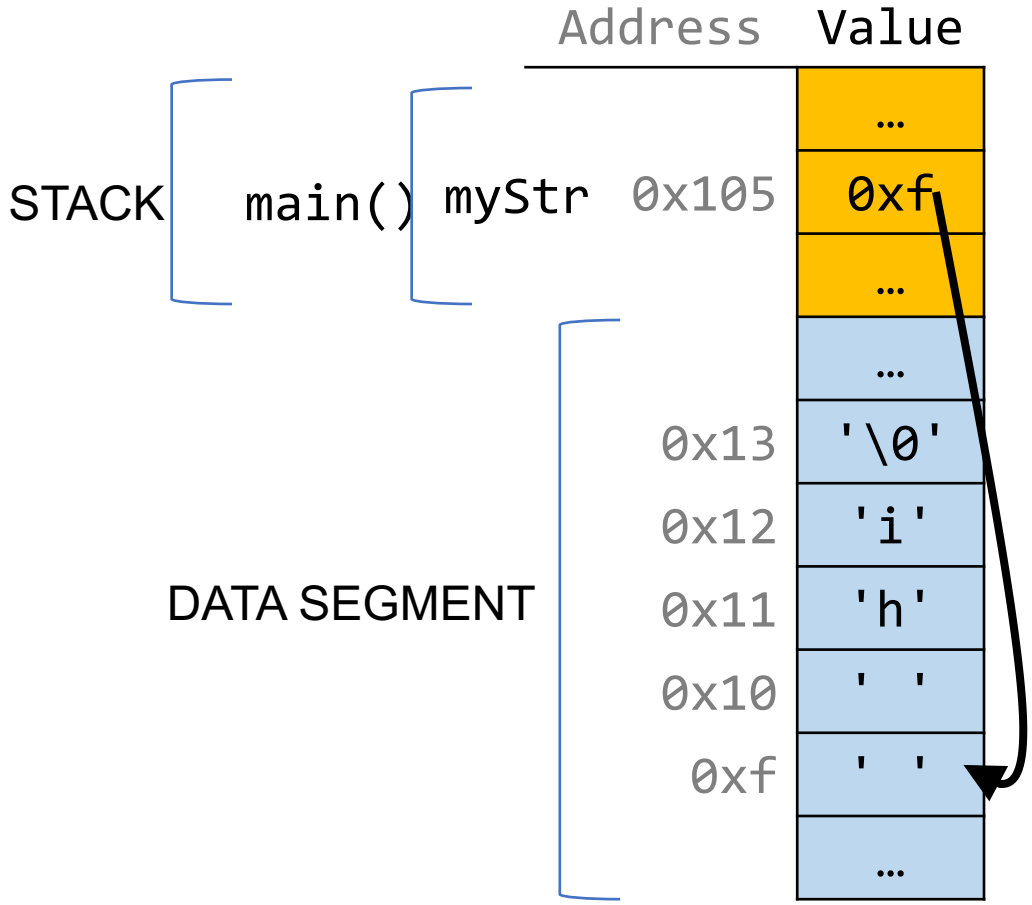
Pointers to Strings

```
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;
}
```



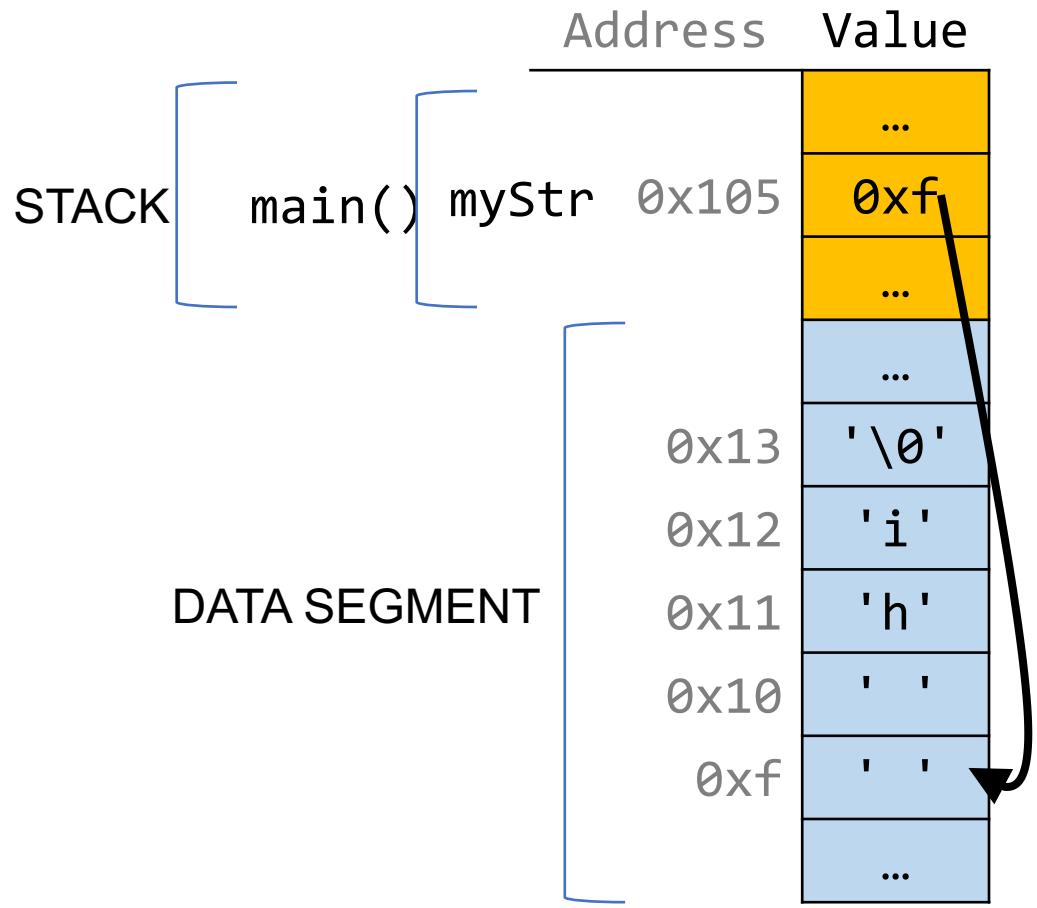
Pointers to Strings

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

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



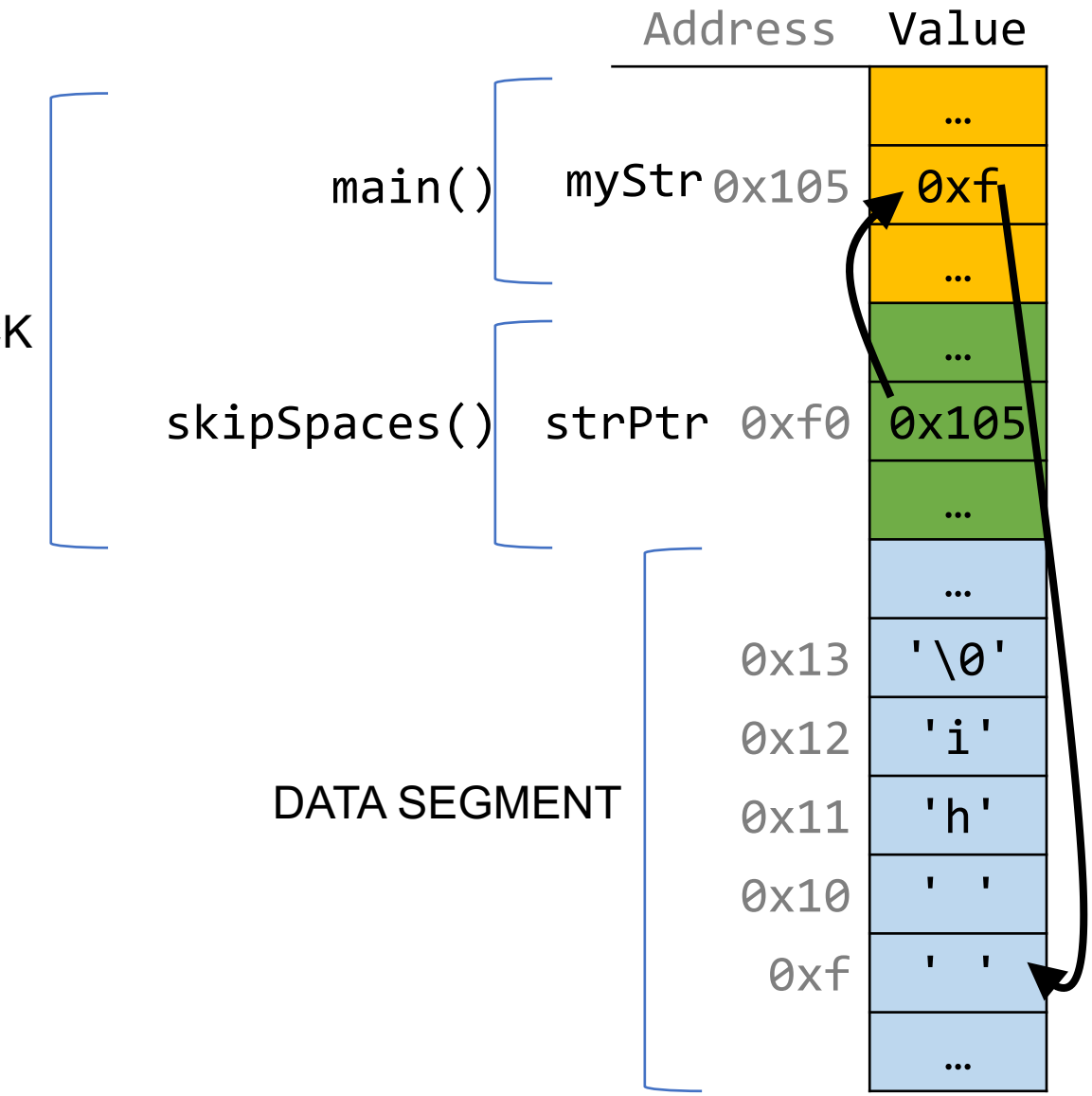
Pointers to Strings

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

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



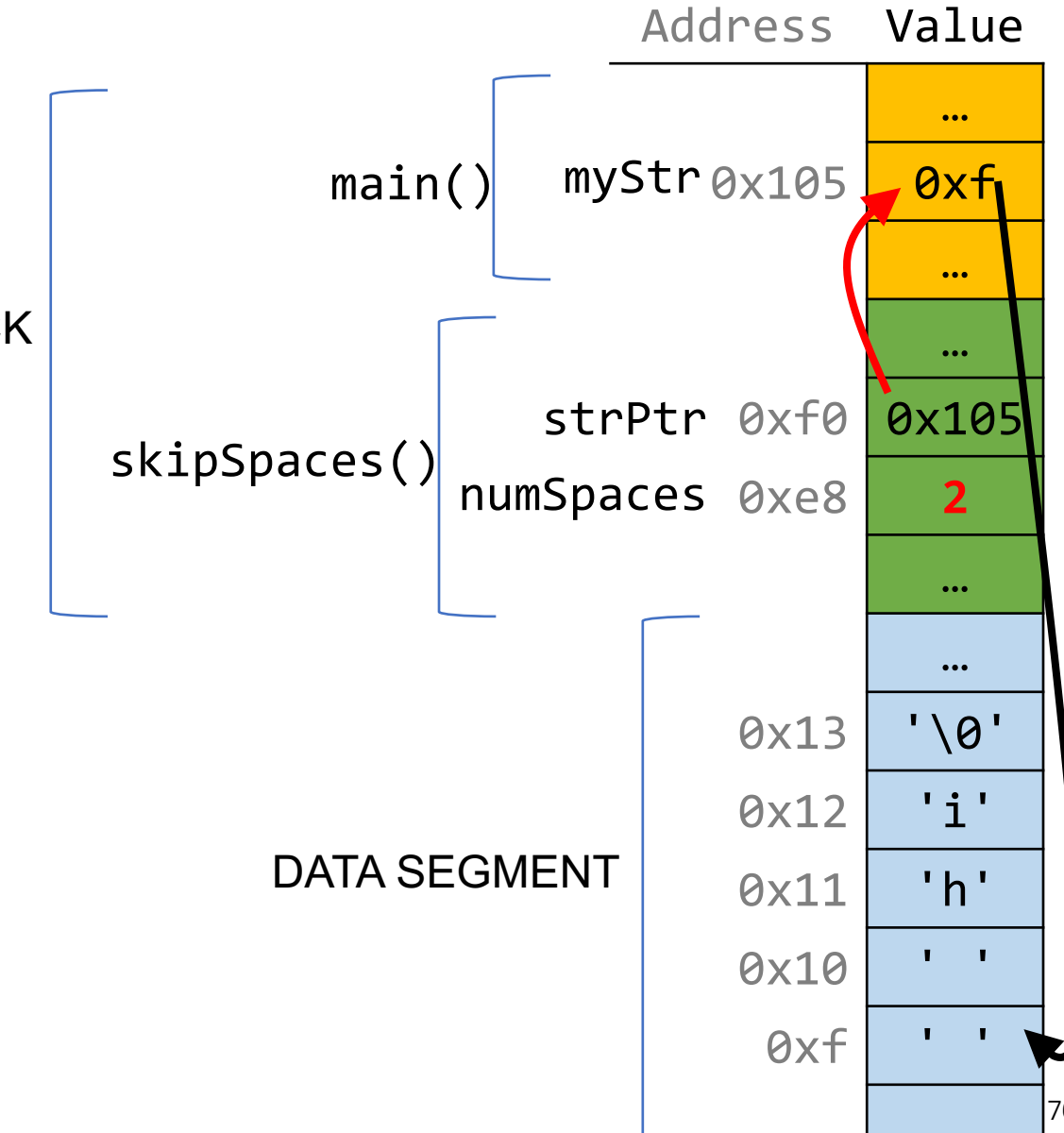
Pointers to Strings

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

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



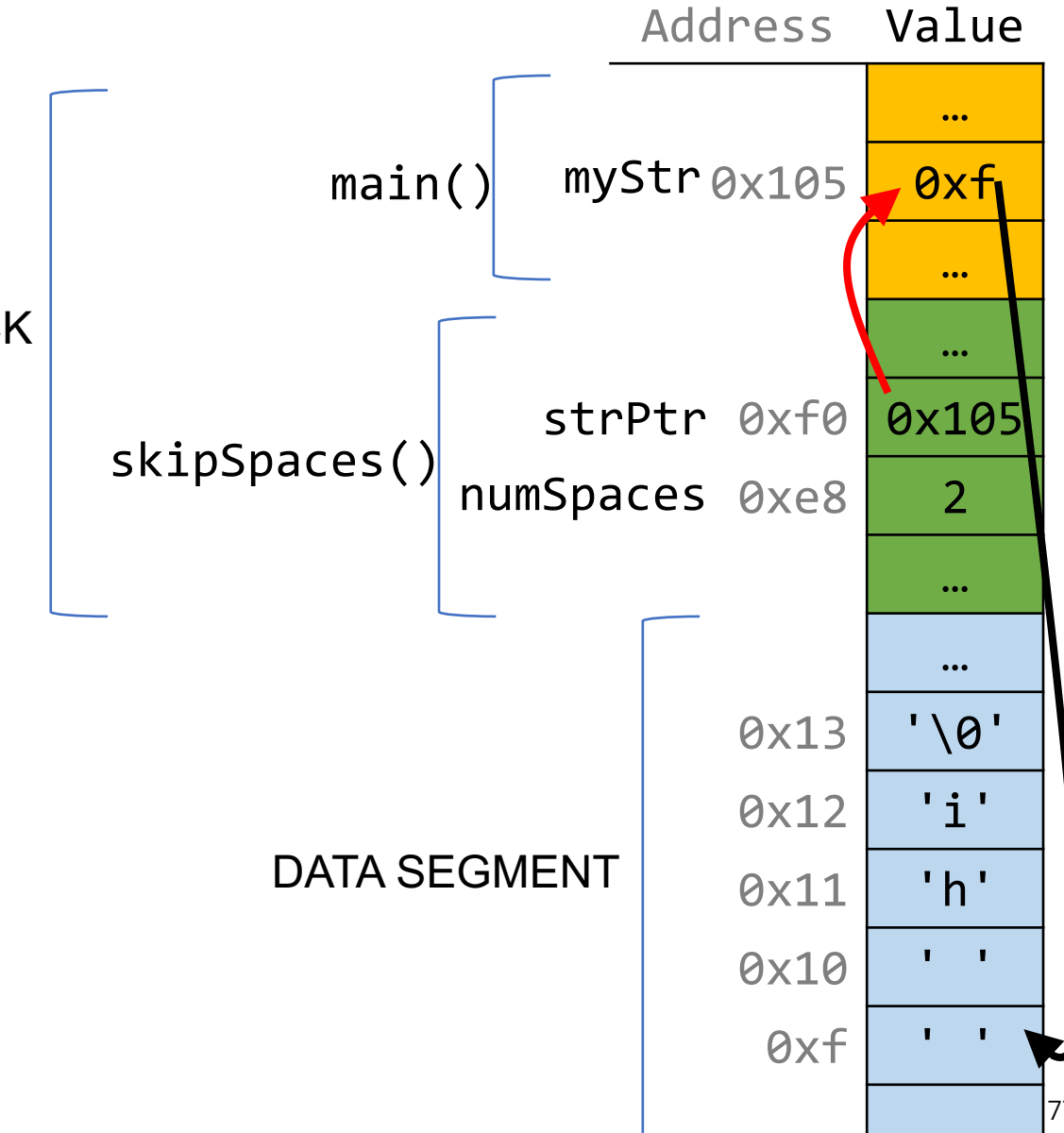
Pointers to Strings

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

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



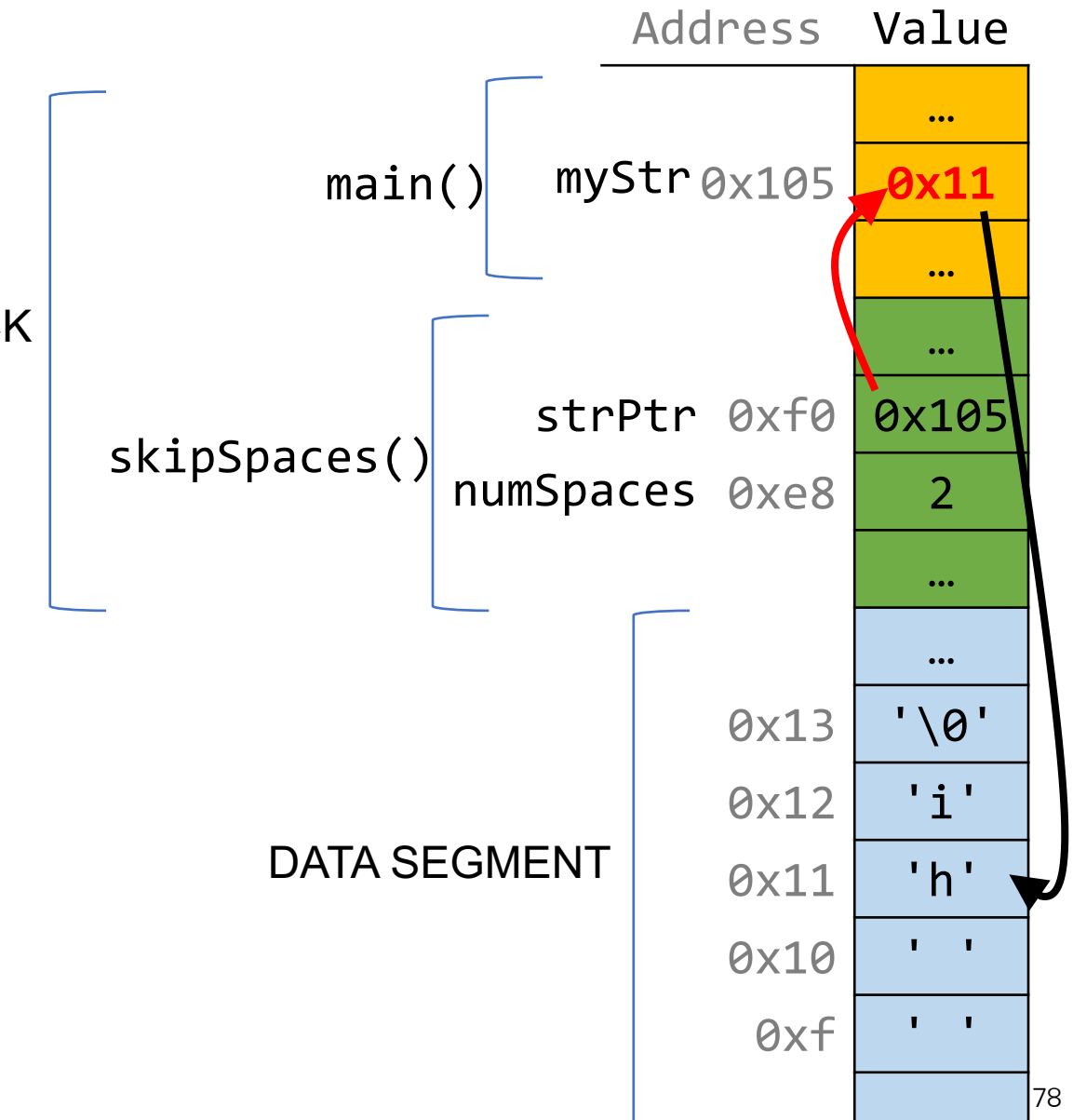
Pointers to Strings

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

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



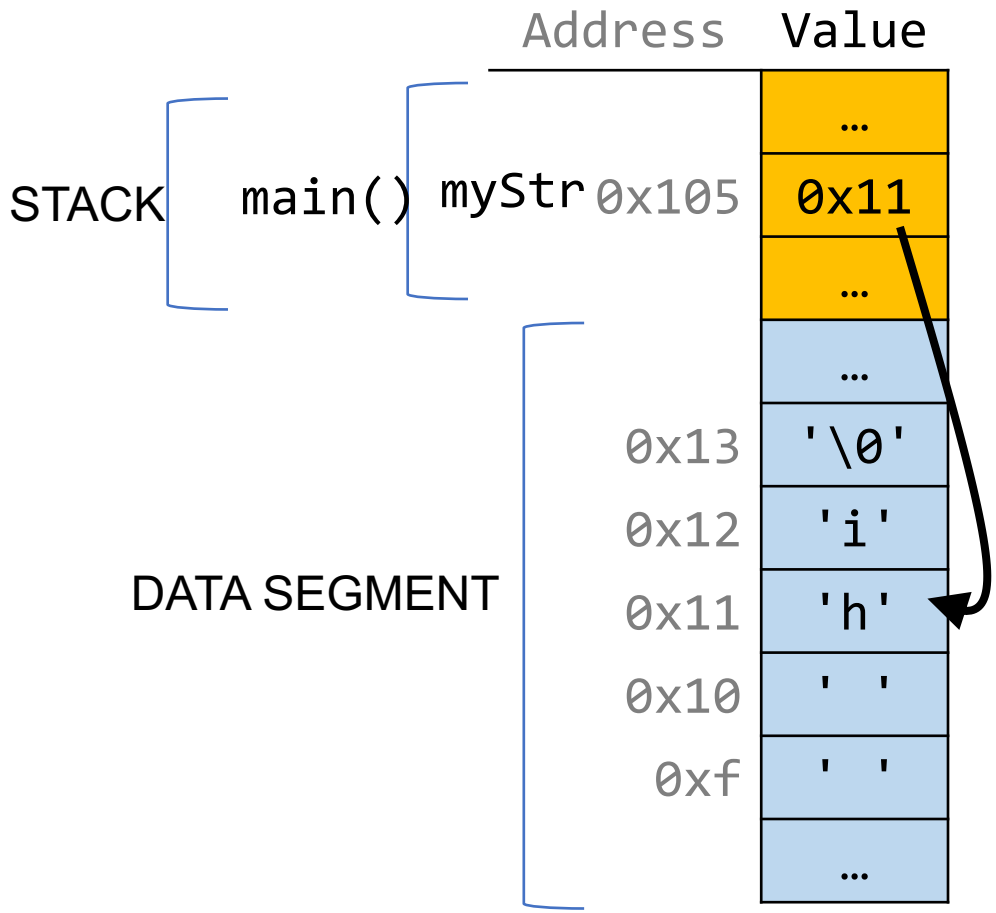
Pointers to Strings

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

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



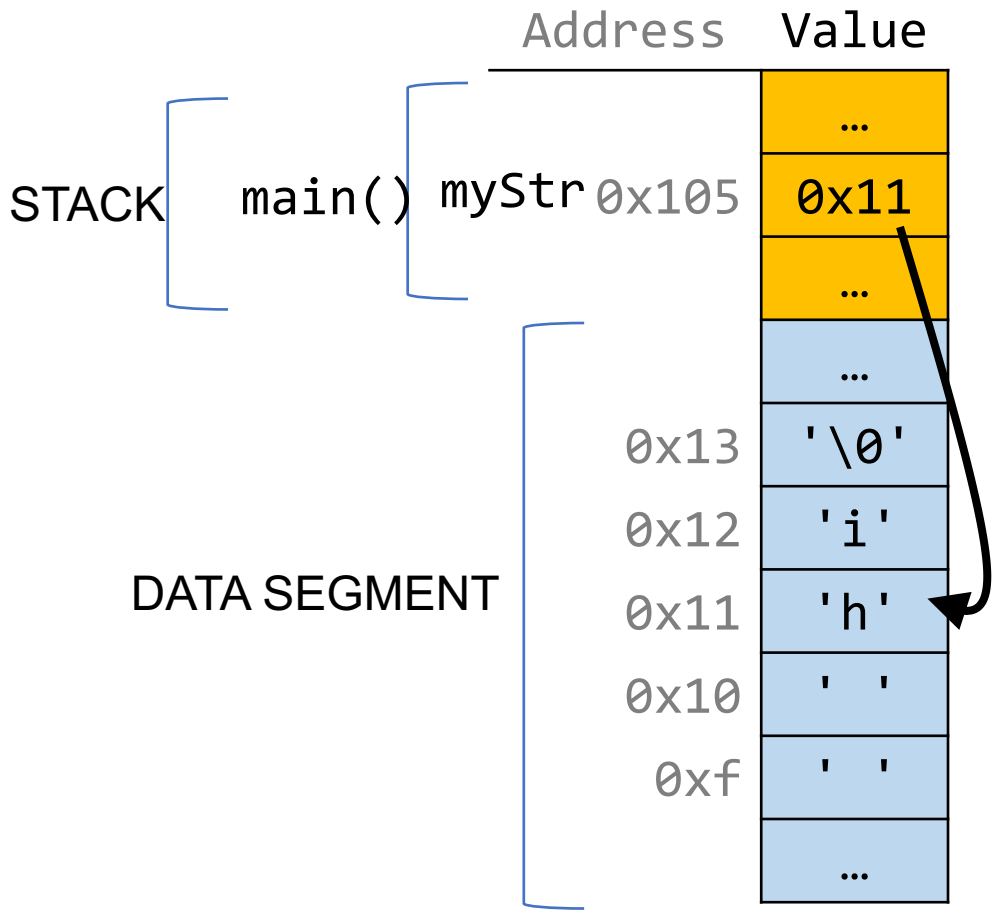
Pointers to Strings

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

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



Pointers to Strings

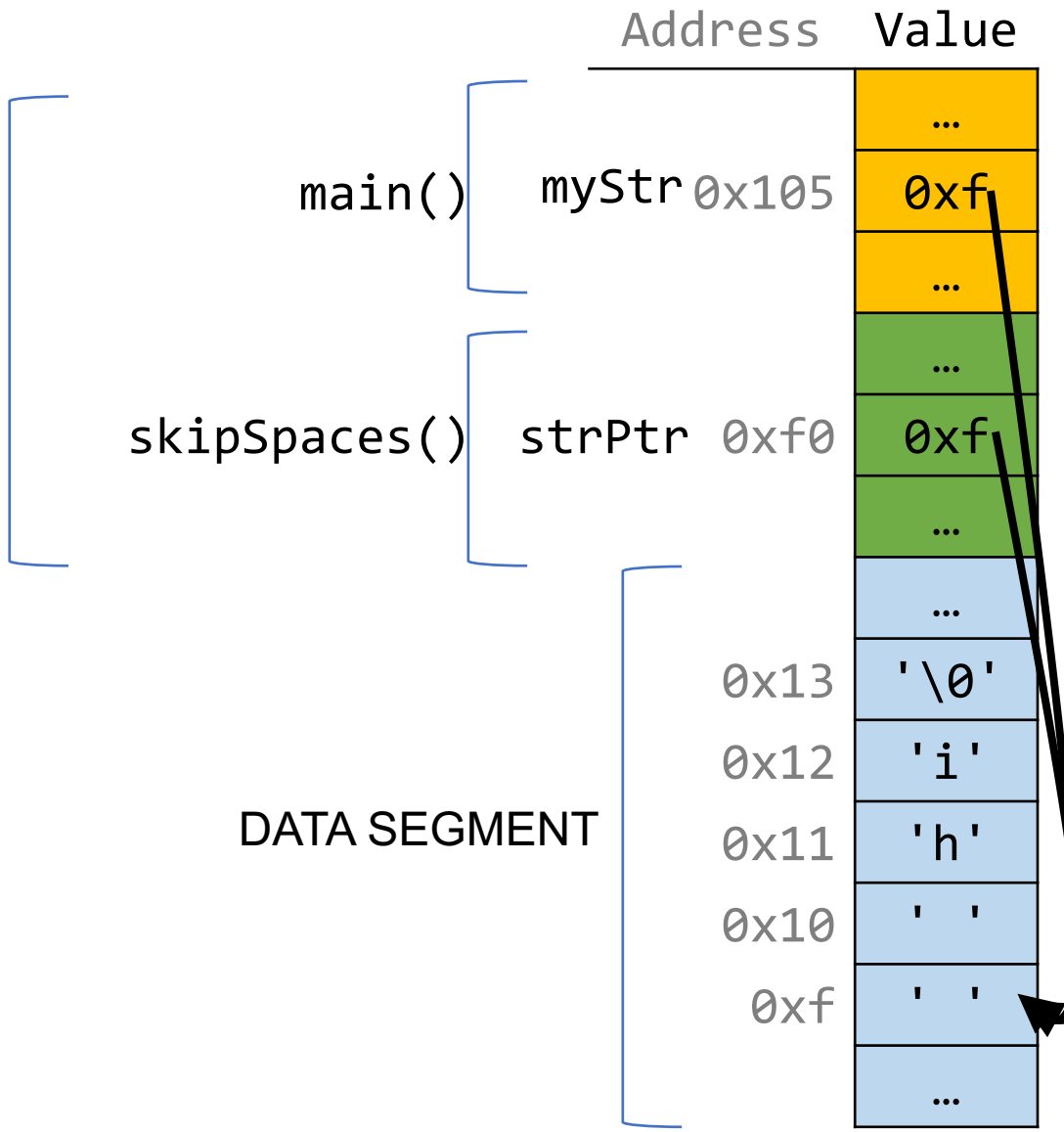
```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

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



Making Copies

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

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

myFunc myFunc
// hi

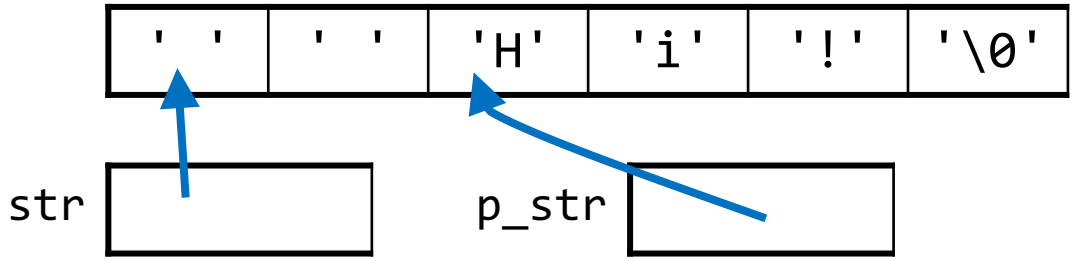


Skip spaces

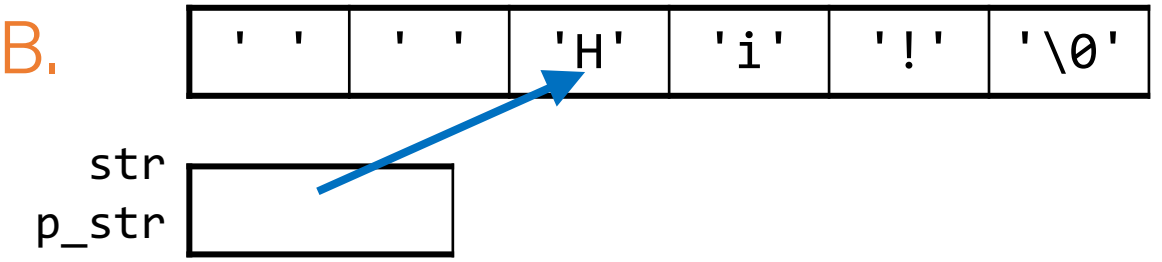
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

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

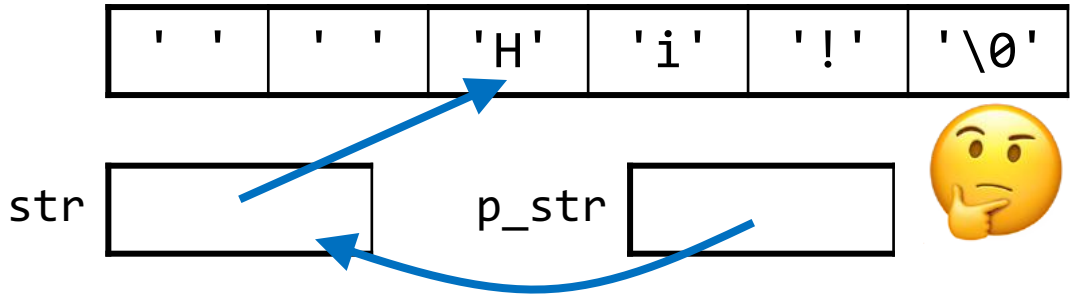
A.



B.



C.

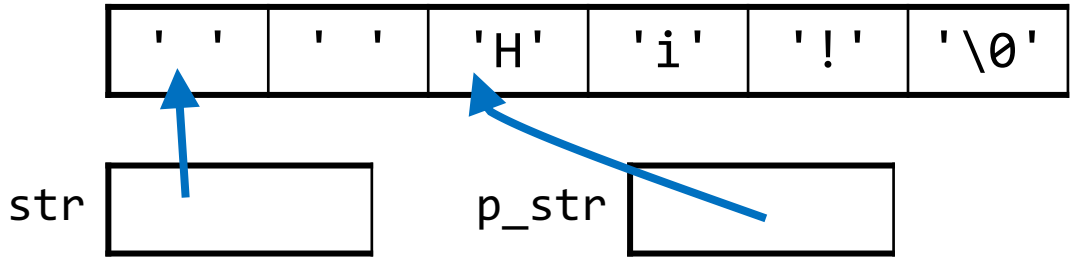


Skip spaces

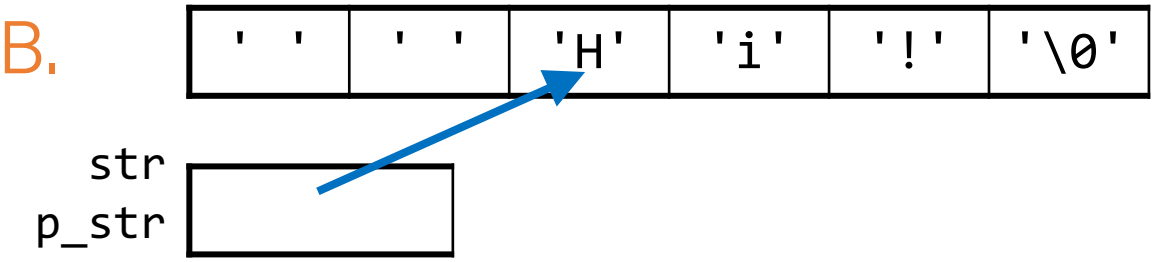
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

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

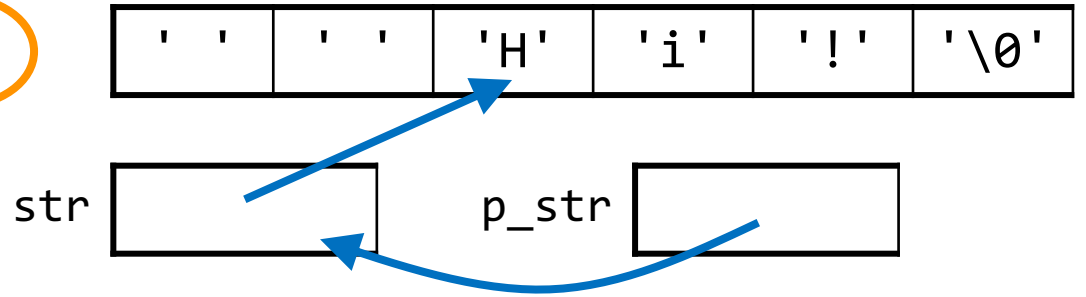
A.



B.



C.



Lecture Plan

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

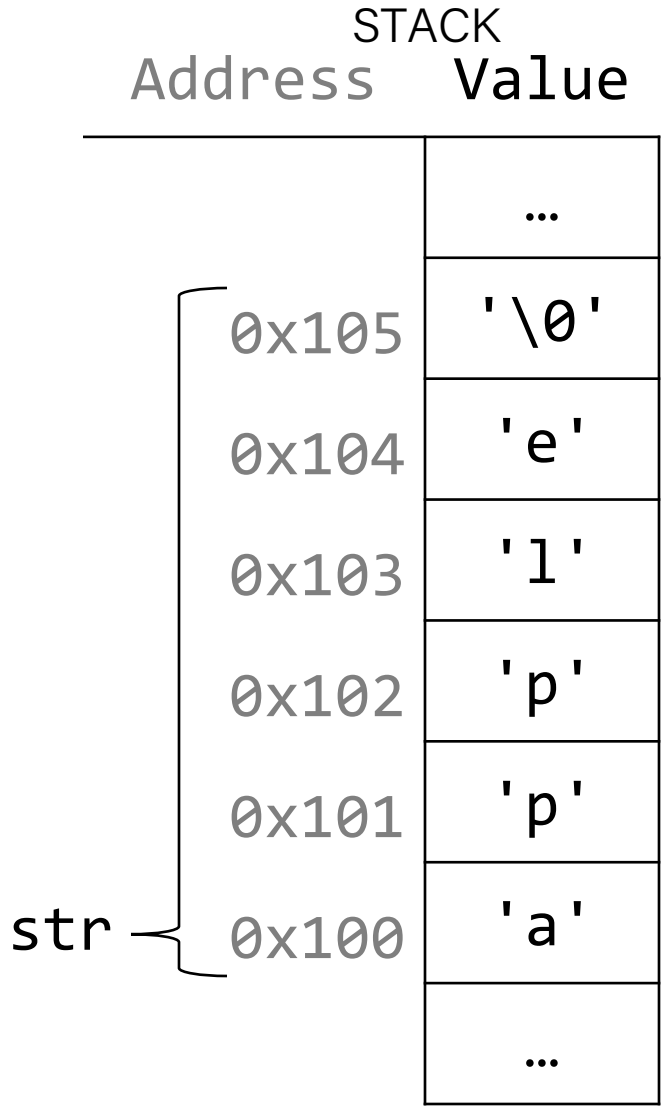
Arrays

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
```



Arrays

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

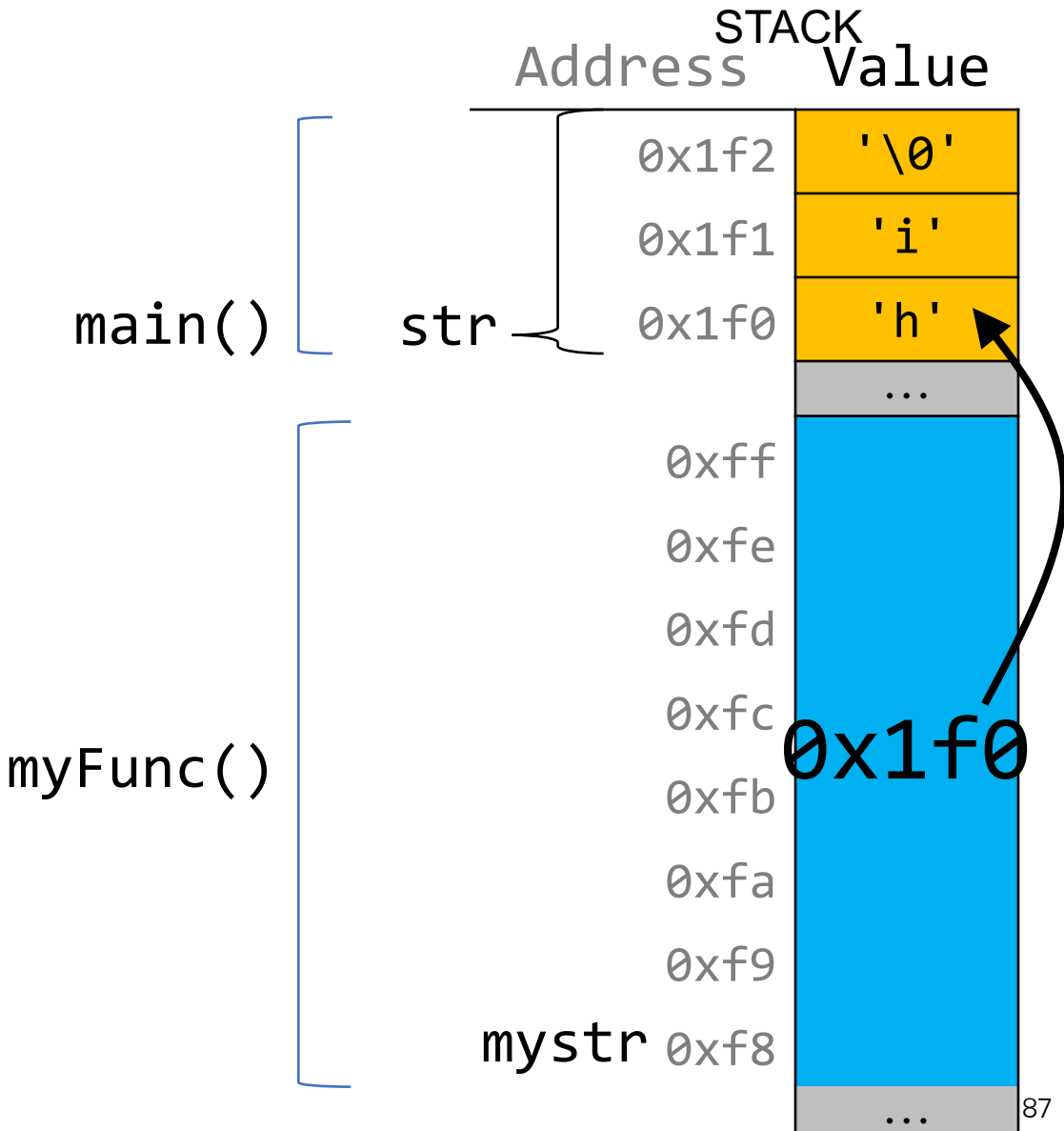
```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

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

Arrays as Parameters

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

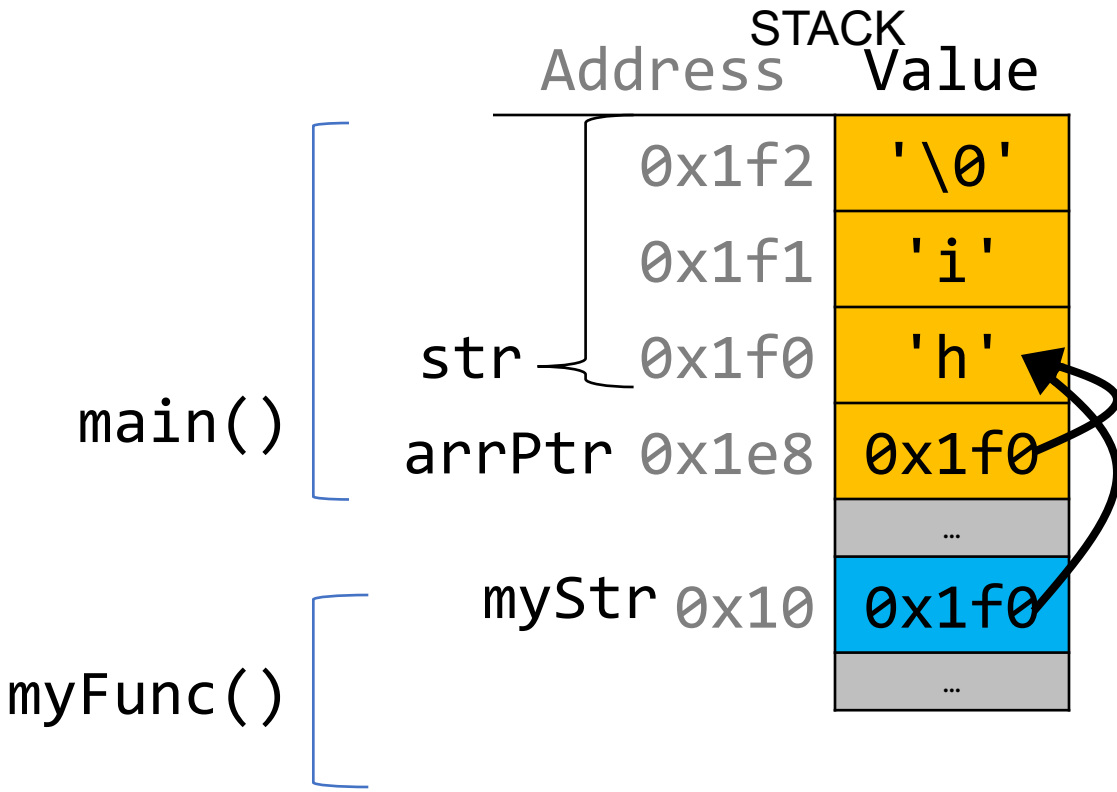
```
void myFunc(char *myStr) {
      ...
}

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



Arrays as Parameters

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

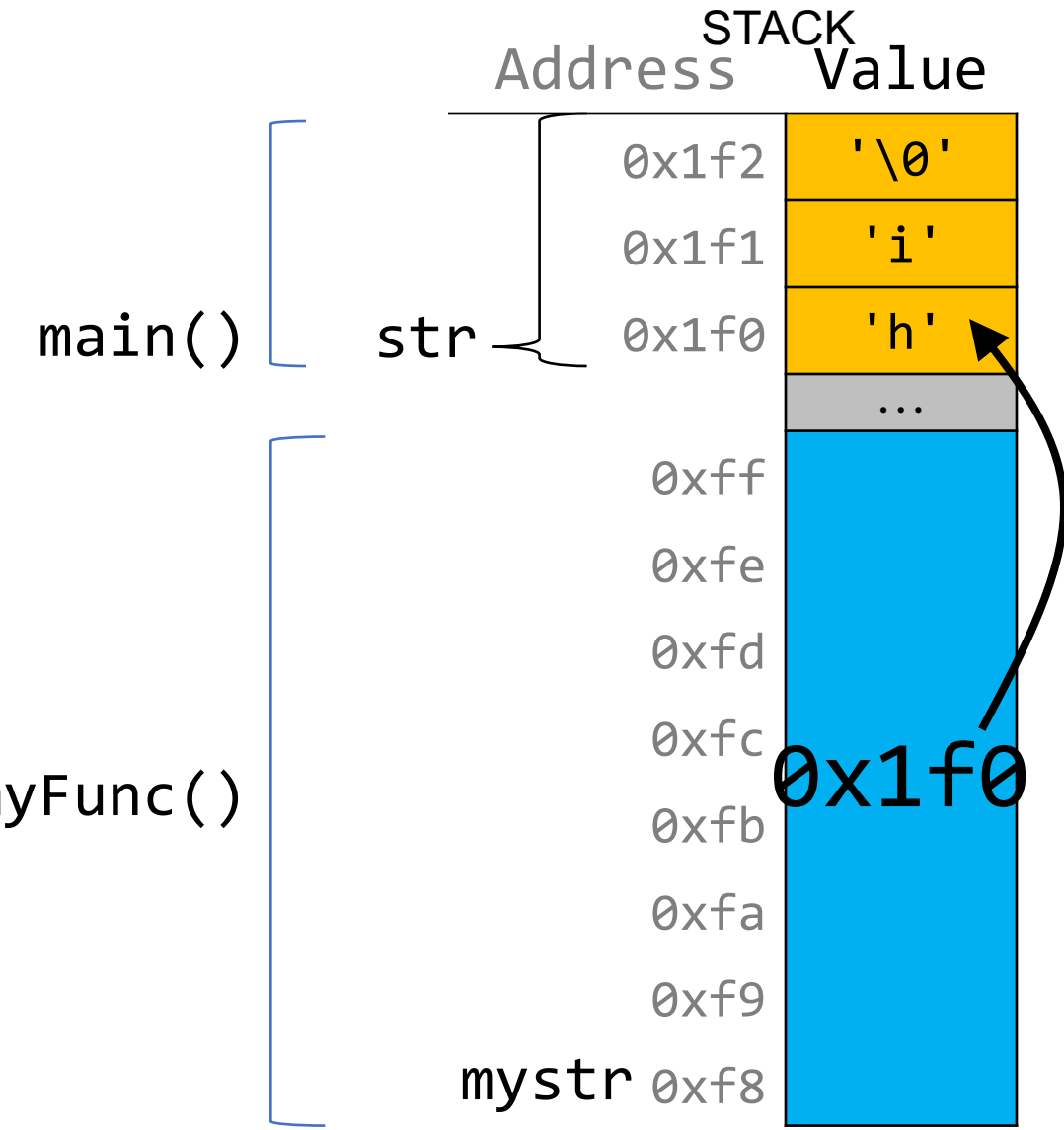
```
void myFunc(char *myStr) {
      …
}

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



Arrays as Parameters

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

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

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

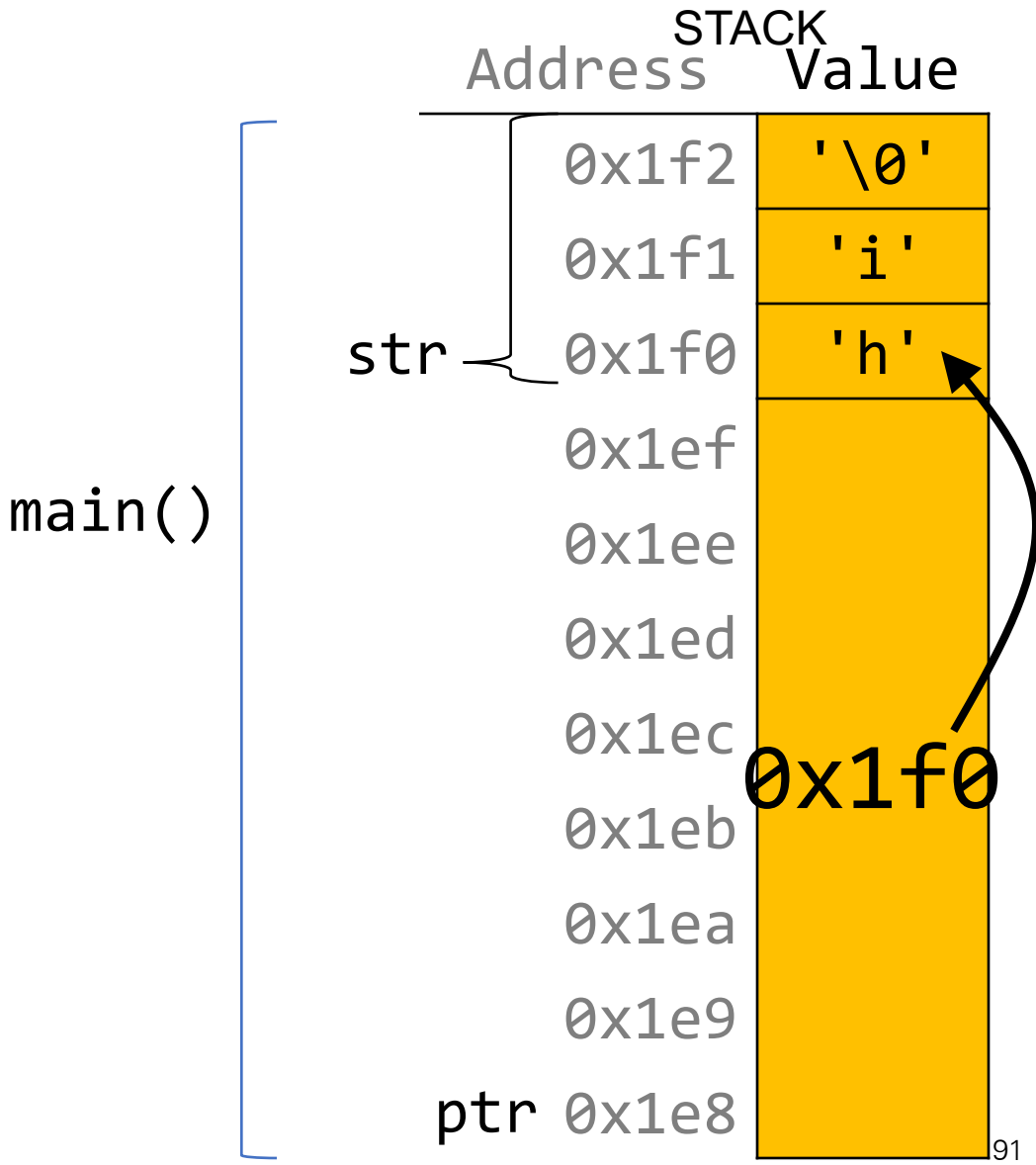


`sizeof` returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **`sizeof`** to get its full size.

Arrays and Pointers

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

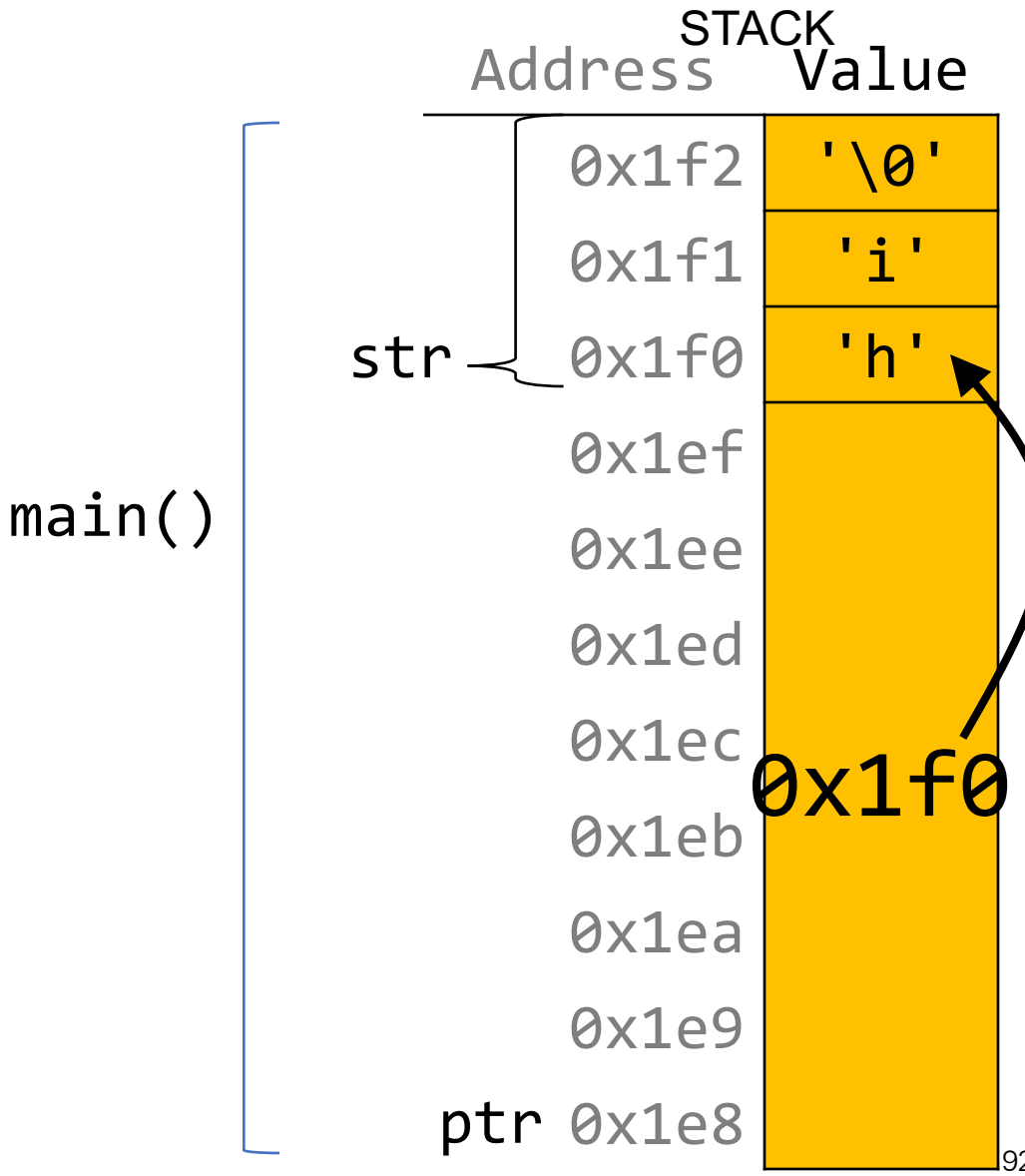
```
int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      char *ptr = str;
      ...
}
```



Arrays and Pointers

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

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

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

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



Lecture Plan

- Strings in Memory (cont'd.)
- 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:

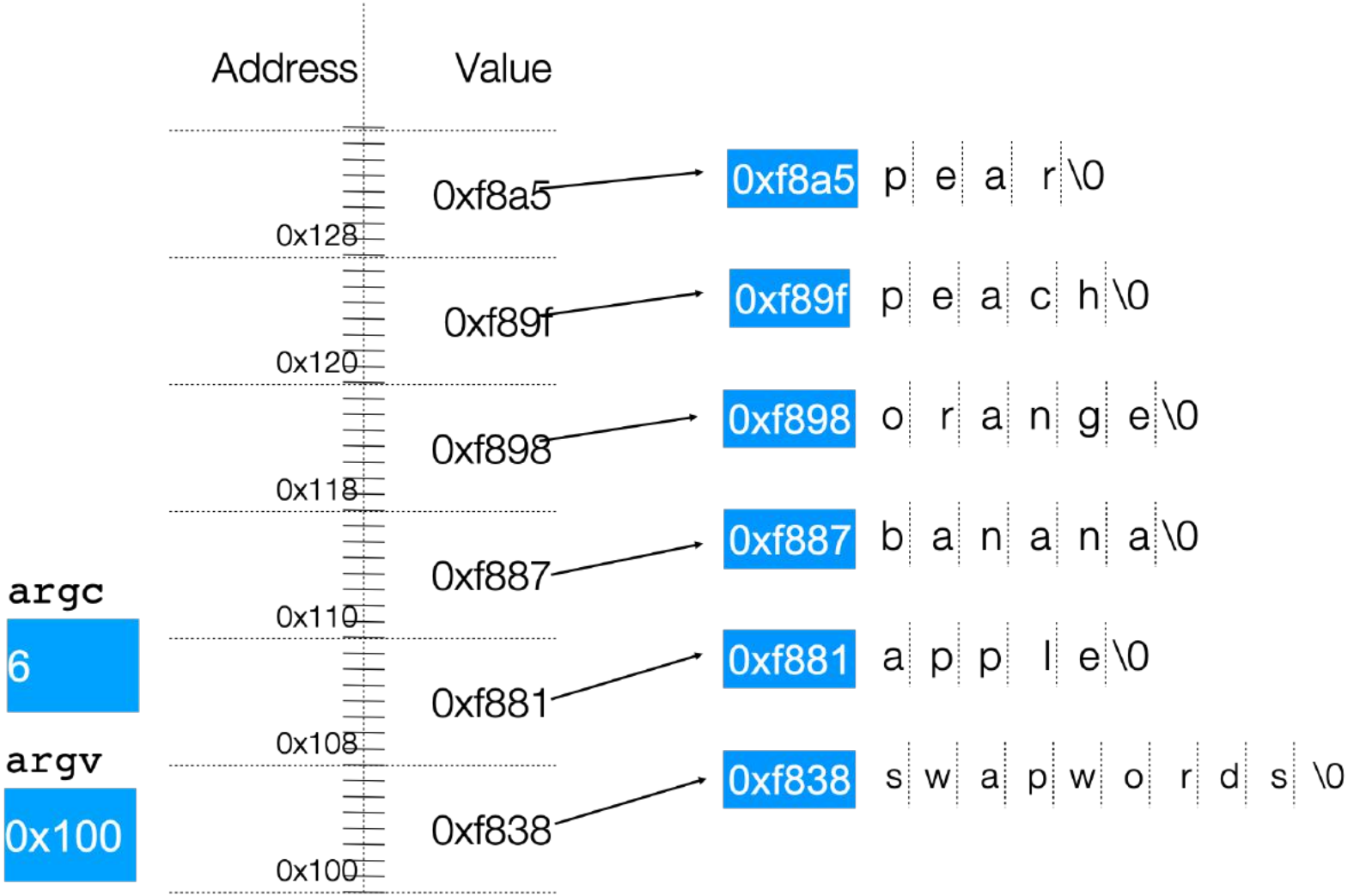
```
char *stringArray[5];  // space to store 5 char *s
```

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

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

Arrays Of Pointers

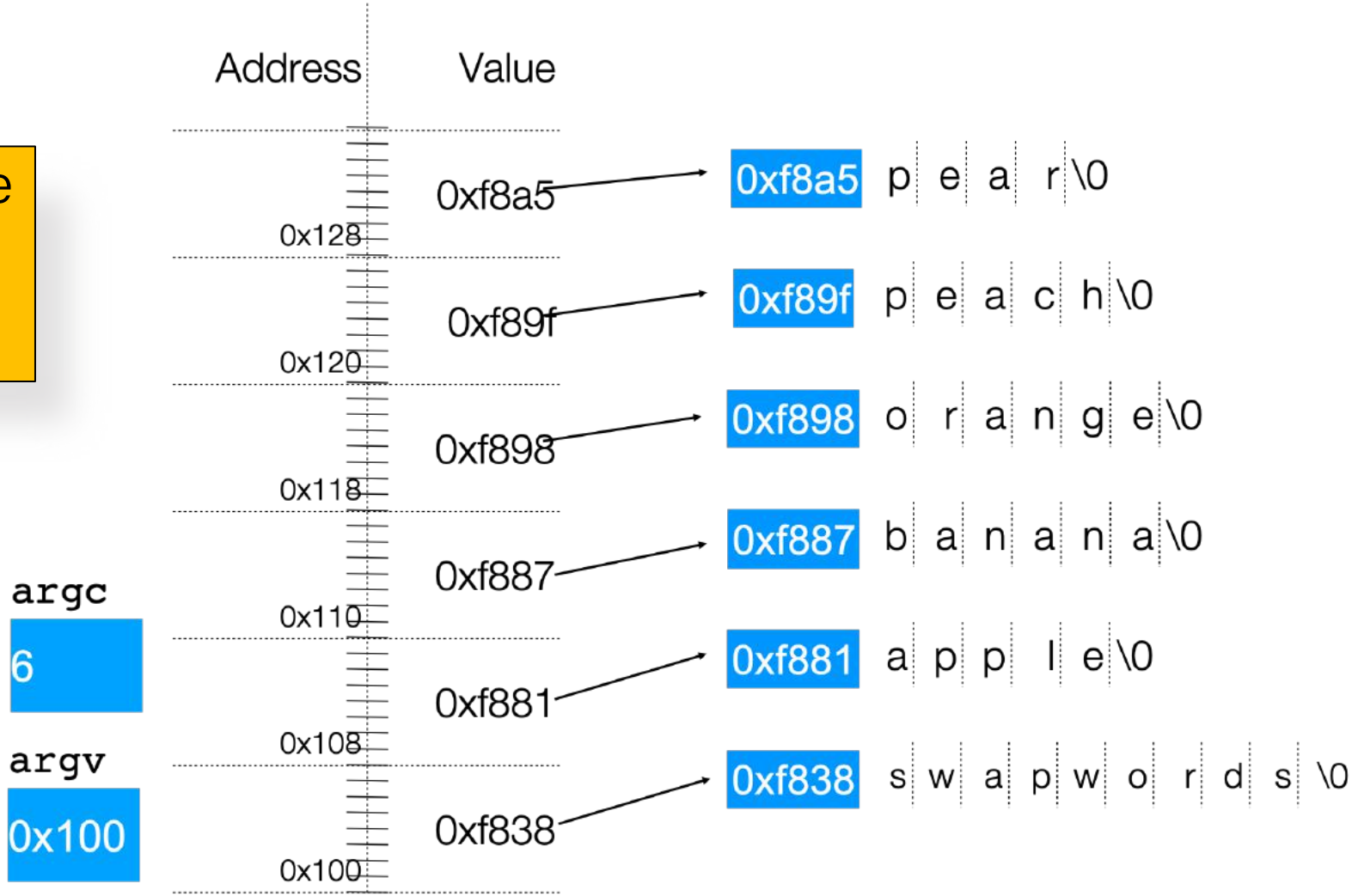
```
./swapwords apple banana orange peach pear
```



Arrays Of Pointers

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

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



Lecture Plan

- Strings in Memory (cont'd.)
- 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
```

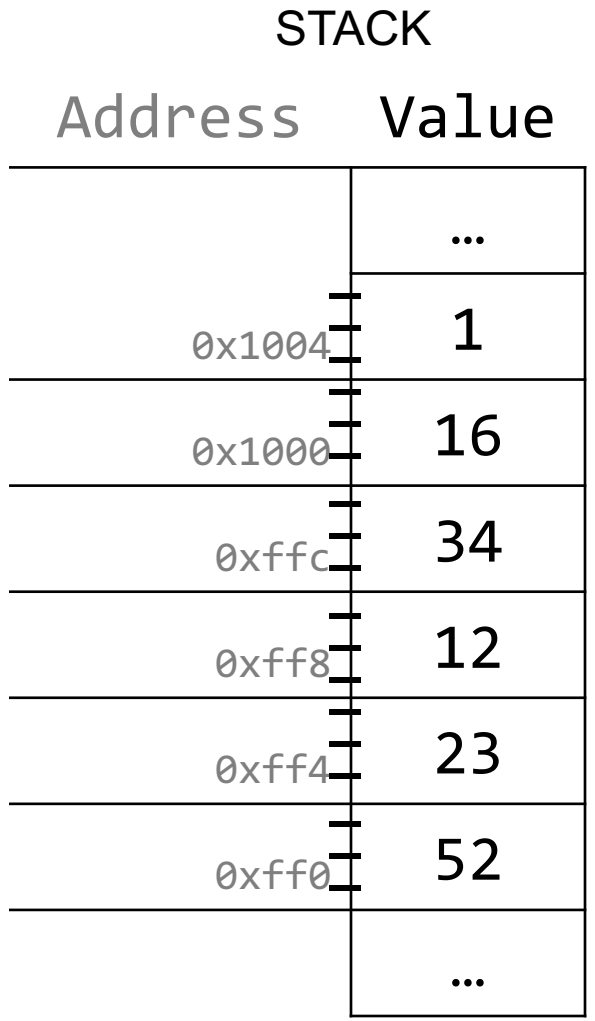
DATA SEGMENT

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
```



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'
```

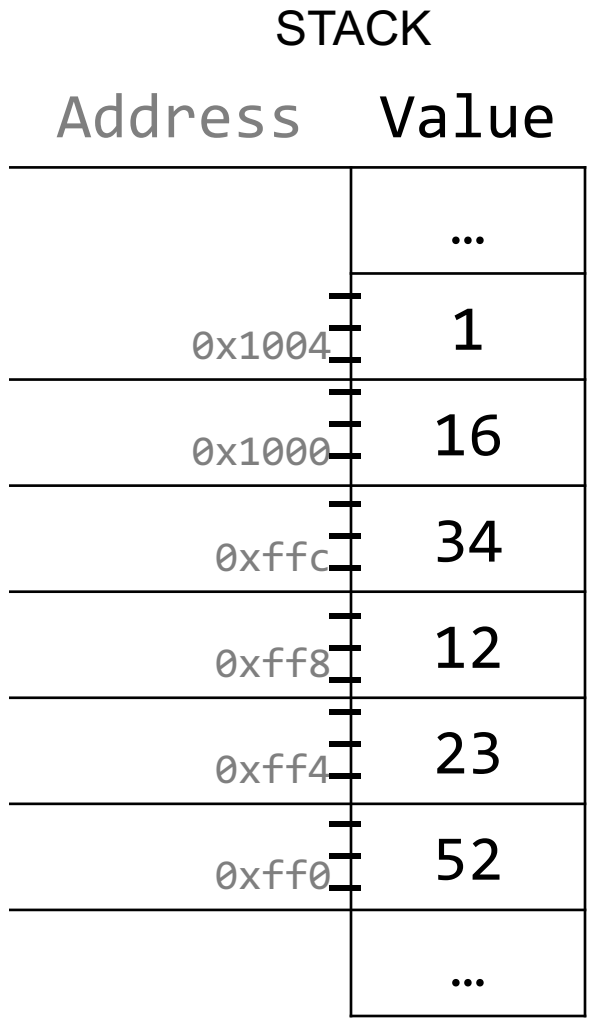
DATA SEGMENT

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
```



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:

```
ptr + i          ⇔        &ptr[i]
*(ptr + i)       ⇔        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.

Extra Slides

1. Pointer arithmetic

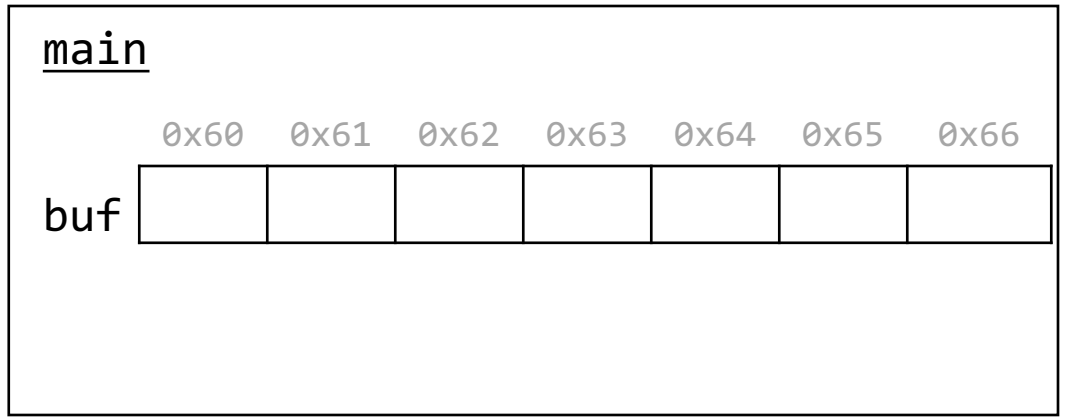
```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```

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

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

1. Pointer arithmetic

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```



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

2. Code study: `strncpy`

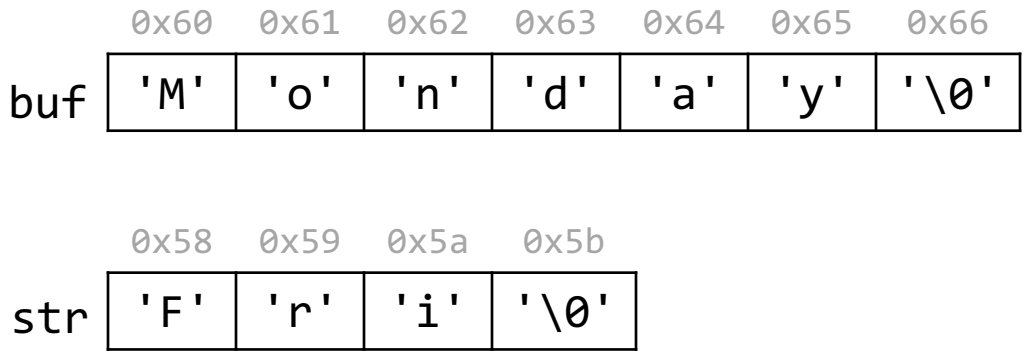
STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, **strncpy**() writes additional null bytes to dest to ensure that a total of n bytes are written.

A simple implementation of **strncpy**() might be:

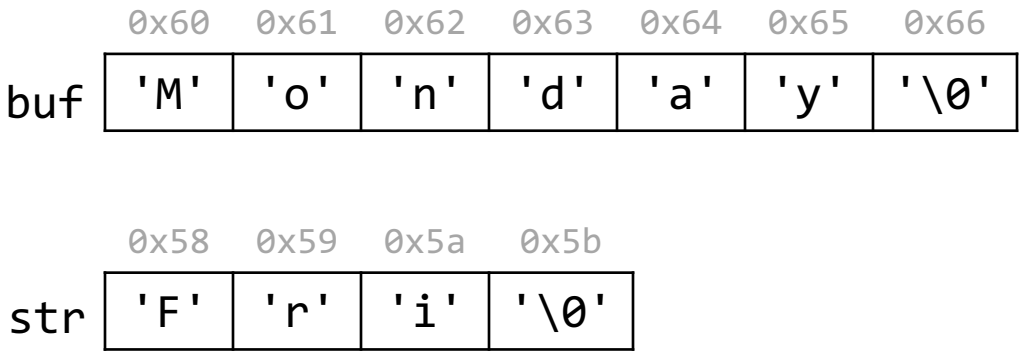



```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

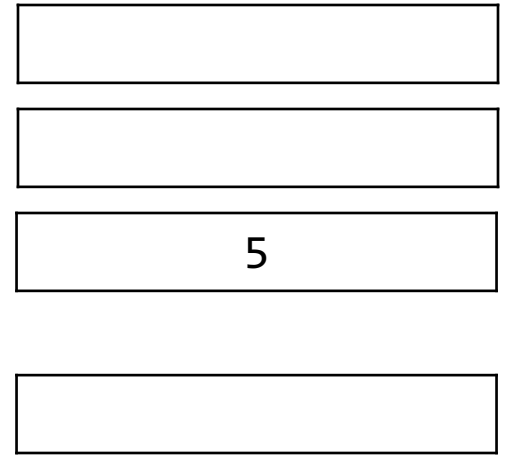
What happens if we call `strncpy(buf, str, 5);`?

2. Code study: `strncpy`

```
STRCPY(3)                  Linux Programmer's Manual                  STRCPY(3)

DESCRIPTION
       The strncpy() function is similar, except that at most n bytes of src  are
       copied.  Warning: If there is no null byte among the first n bytes of src,
       the string placed in dest will not be null-terminated.

       If the length of src is less than  n,  strncpy()  writes  additional  null
       bytes to dest to ensure that a total of n bytes are written.

       A simple implementation of strncpy() might be:
```




```c
1 char *strncpy(char *dest, const char *src, size_t n) {
2     size_t i;
3     for (i = 0; i < n && src[i] != '\0'; i++)
4         dest[i] = src[i];
5     for ( ; i < n; i++)
6         dest[i] = '\0';
7     return dest;
8 }
```



What happens if we call `strncpy(buf, str, 5);`?

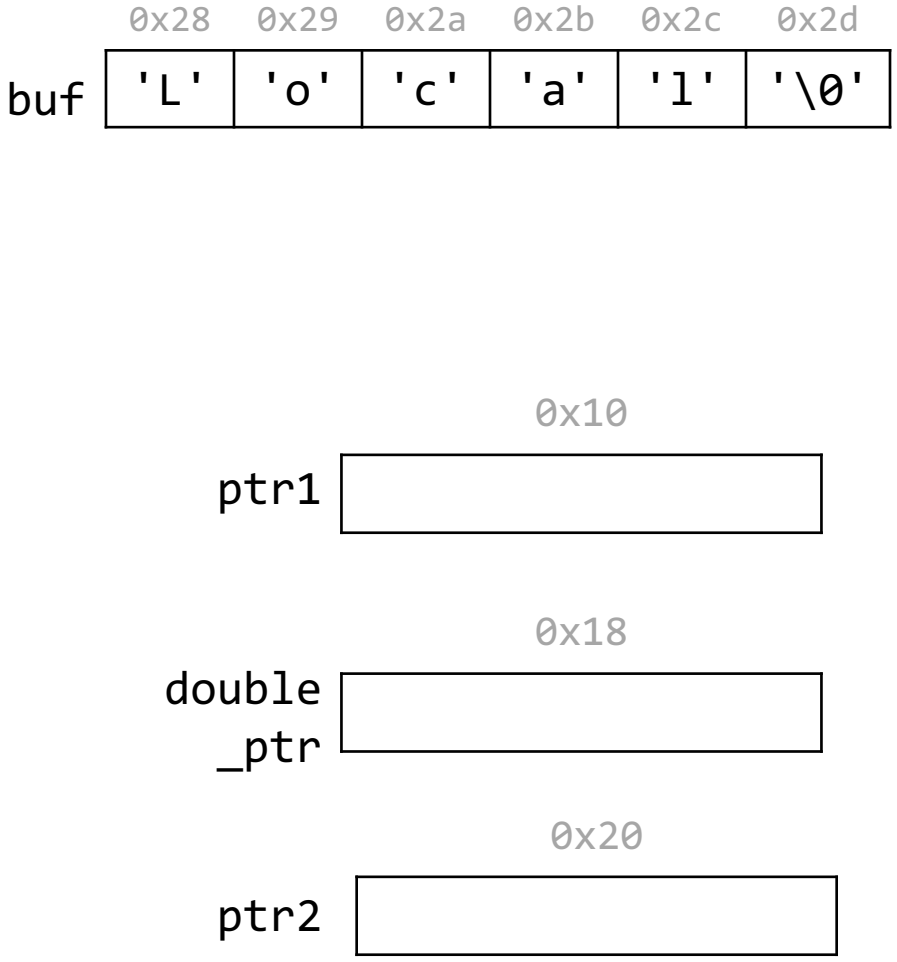
3. Bonus: Tricky addresses

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```

What is stored in each variable? (We cover double pointers later in the course)

3. Bonus: Tricky addresses

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```



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

Recap

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

Next Time: dynamically allocated memory