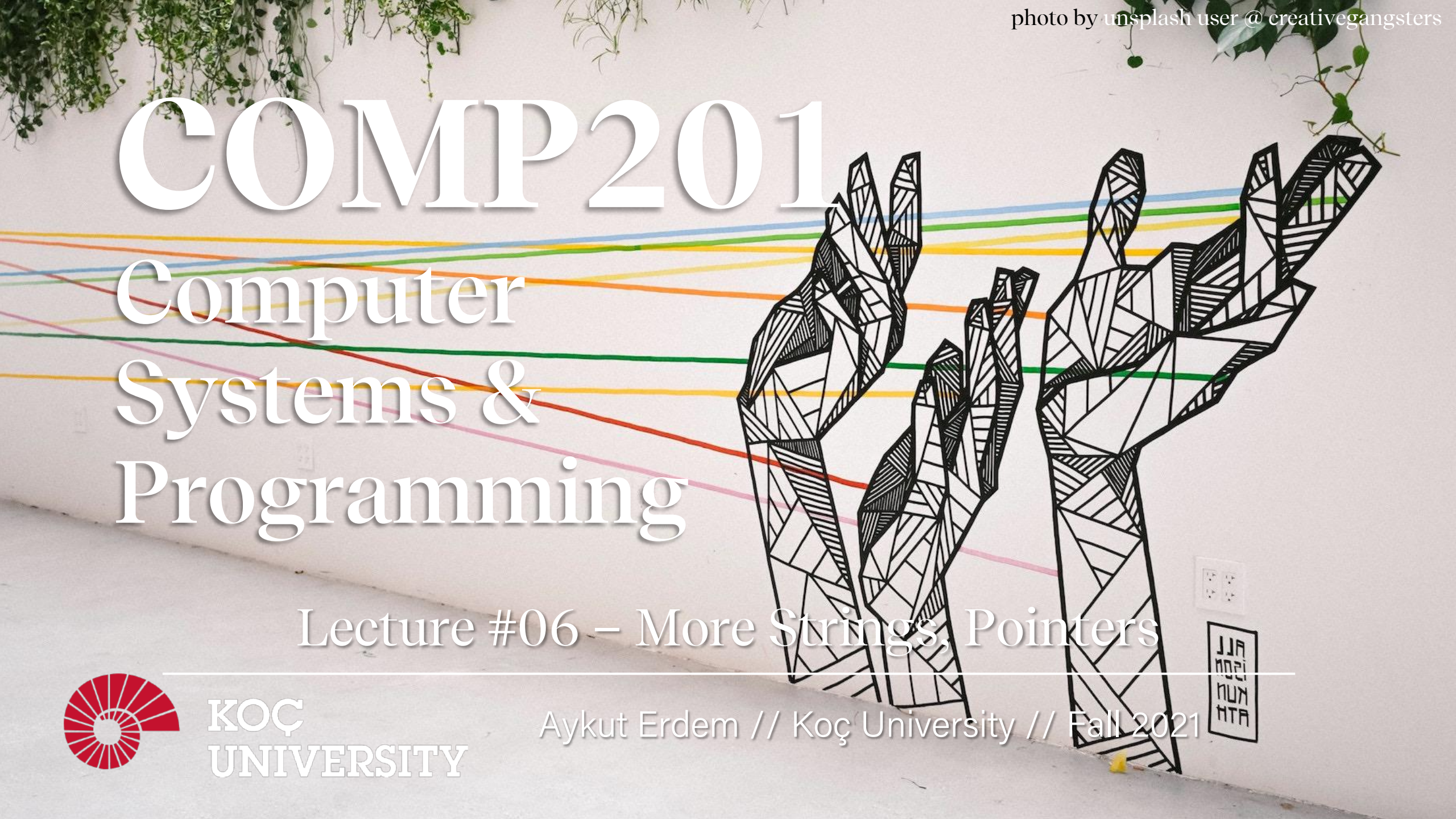


COMP201

Computer Systems & Programming

Lecture #06 – More Strings, Pointers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- Practice: Diamonds

Plan for Today

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Disclaimer: Slides for this lecture were borrowed from

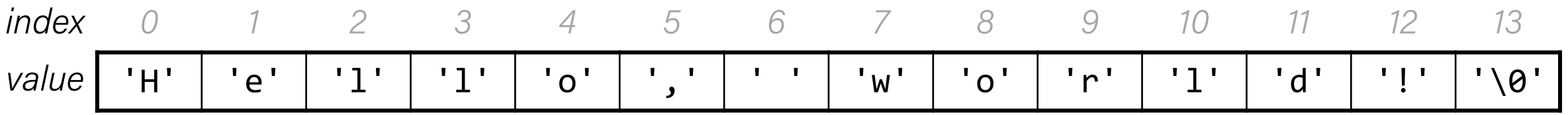
—Nick Troccoli and Lisa Yan's Stanford CS107 class

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

C Strings

C strings are arrays of characters ending with a **null-terminating character** '\0'.



String operations such as `strlen` use the null-terminating character to find the end of the string.

Side note: use `strlen` to get the length of a string. Don't use `sizeof`!

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`***.

Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```
char daisy[6];
strcpy(daisy, "Daisy");
char *letterA = strchr(daisy, 'a');
printf("%s\n", daisy);              // Daisy
printf("%s\n", letterA);            // aisy
```

If there are multiple occurrences of the letter, **`strchr`** returns a pointer to the *first* one. Use **`strrchr`** to obtain a pointer to the *last* occurrence.

Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```
char daisy[10];
strcpy(daisy, "Daisy Dog");
char *substr = strstr(daisy, "Dog");
printf("%s\n", daisy);              // Daisy Dog
printf("%s\n", substr);             // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char daisy[10];
strcpy(daisy, "Daisy Dog");
int spanLength = strspn(daisy, "aDeoi");        // 3
```

“How many places can we go in the first string before I encounter a character not in the second string?”

String Spans

`strcspn` (c = "complement") returns the *length* of the initial part of the first string which contains only characters not in the second string.

```
char daisy[10];
strcpy(daisy, "Daisy Dog");
int spanLength = strcspn(daisy, "driso");      // 2
```

"How many places can we go in the first string before I encounter a character in the second string?"

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char *.** We can still operate on the string the same way as with a char[]. (*We'll see why today!)*.

```
int doSomething(char *str) {
     char secondChar = str[1];
     ...
}

// can also write this, but it is really a pointer
int doSomething(char str[]) { ...
```

Arrays of Strings

We can make an array of strings to group multiple strings together:

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

We can also use the following shorthand to initialize a string array:

```
char *stringArray[] = {
     "Hello",
     "Hi",
     "Hey there"
};
```

Arrays of Strings

We can access each string using bracket syntax:

```
printf("%s\n", stringArray[0]);  // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the first element of the array*. This is what **argv** is in **main**! This means we write the parameter type as:

```
void myFunction(char **stringArray) {

// equivalent to this, but it is really a double pointer
void myFunction(char *stringArray[]) {
```

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Practice: Password Verification

Write a function **`verifyPassword`** that accepts a candidate password and certain password criteria and returns whether the password is valid.

```
bool verifyPassword(char *password, char *validChars,
char *badSubstrings[], int numBadSubstrings);
```

password is valid if it contains only letters in **validChars**, and does not contain any substrings in **badSubstrings**.

Practice: Password Verification

```
bool verifyPassword(char *password, char *validChars, char
*badSubstrings[], int numBadSubstrings);
```

Example:

```
char *invalidSubstrings[] = { "1234" };

bool valid1 = verifyPassword("1572", "0123456789",
          invalidSubstrings, 1);       // true
bool valid2 = verifyPassword("141234", "0123456789",
          invalidSubstrings, 1);       // false
```

Practice: Password Verification

`verify_password.c`

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

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 refer to any size memory location!
- Pointers are also essential for allocating memory on the heap, which we will cover later.
- Pointers also let us refer to memory generically, which we will cover later.

Memory

- Memory is a big array of bytes.
- Each byte has a unique numeric index that is commonly written in hexadecimal.
- A pointer stores one of these memory addresses.

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

Memory

- Memory is a big array of bytes.
- Each byte has a unique numeric index that is commonly written in hexadecimal.
- A pointer stores one of these memory addresses.

Address	Value
	…
261	'\0'
260	'e'
259	'l'
258	'p'
257	'p'
256	'a'
	…

Looking Closely at C

- All parameters in C are “pass by value.” For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.
- This means whenever we pass something as a parameter, we pass a copy.
- If we want to modify a parameter value in the function we call and have the changes persist afterwards, we can pass the location of the value instead of the value itself. This way we make a copy of the *address* instead of a copy of the *value*.

Pointers

```
int x = 2;


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


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

Pointers

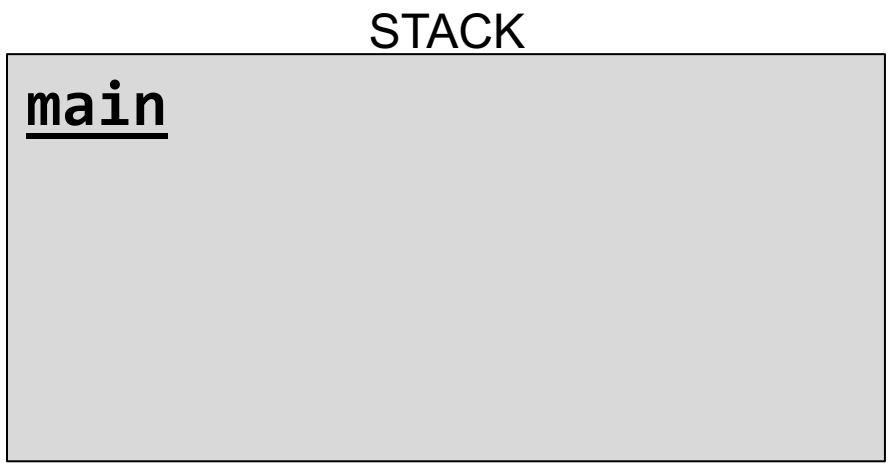
A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

A pointer is a variable that stores a memory address.

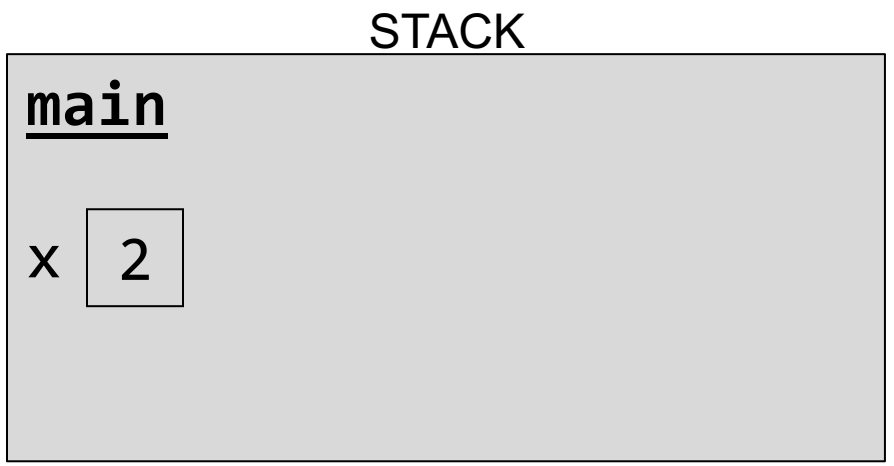
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

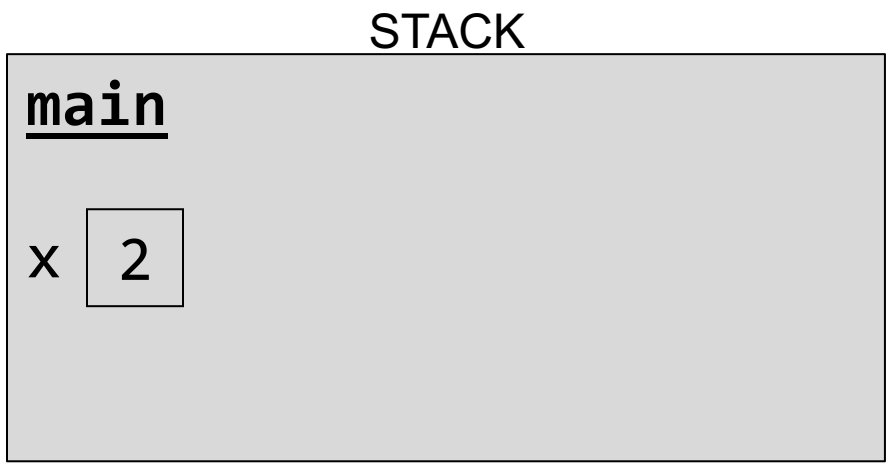
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

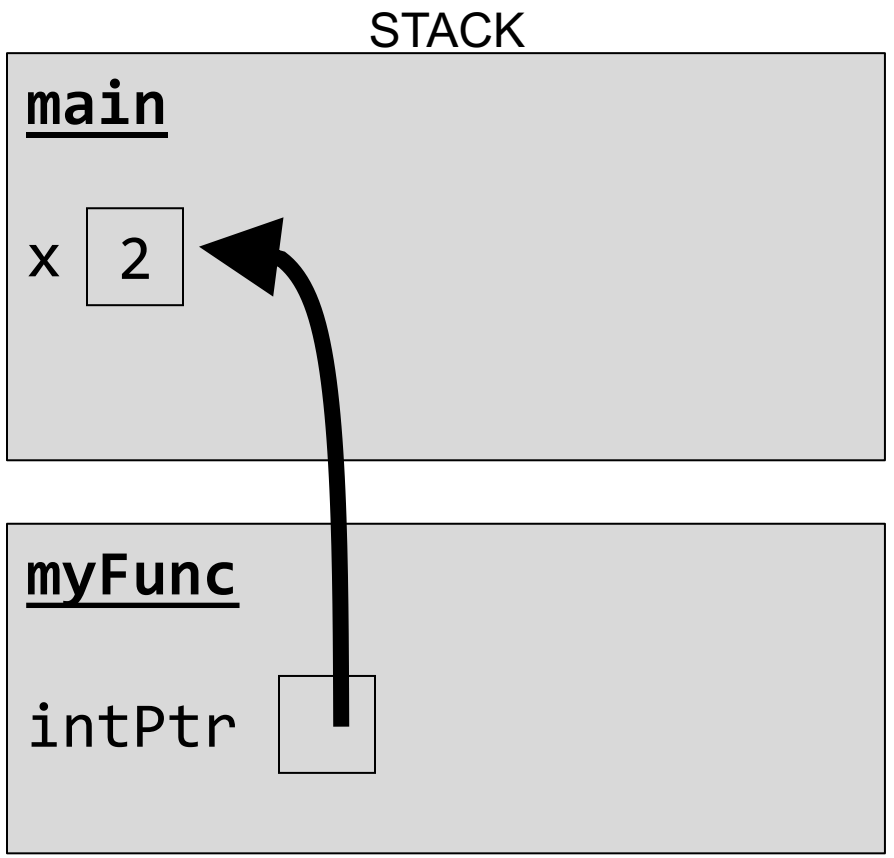
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

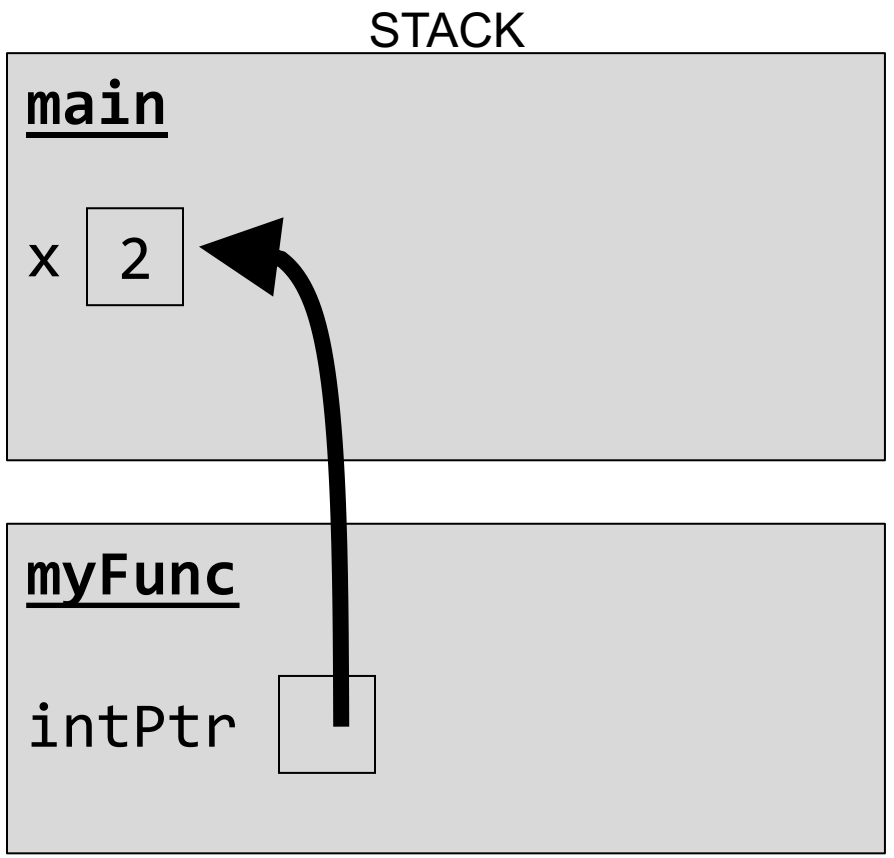
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

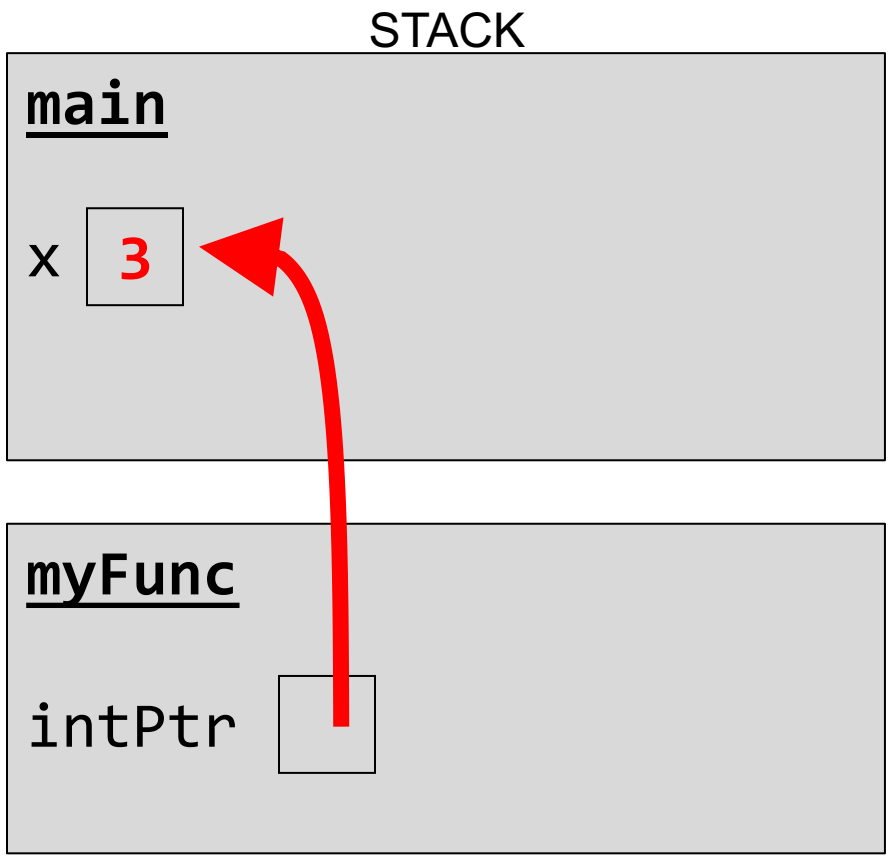
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

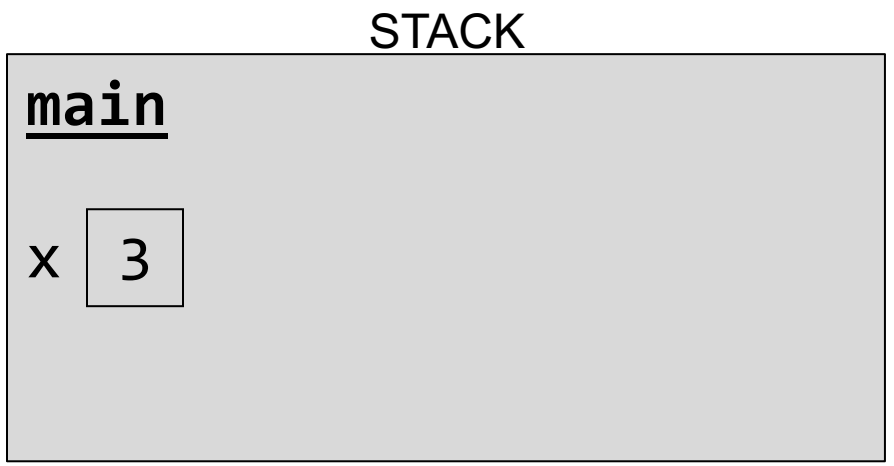
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

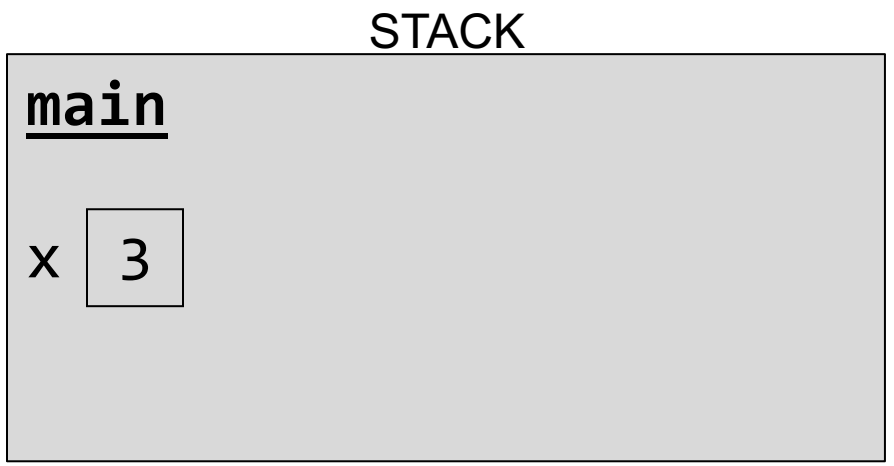
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

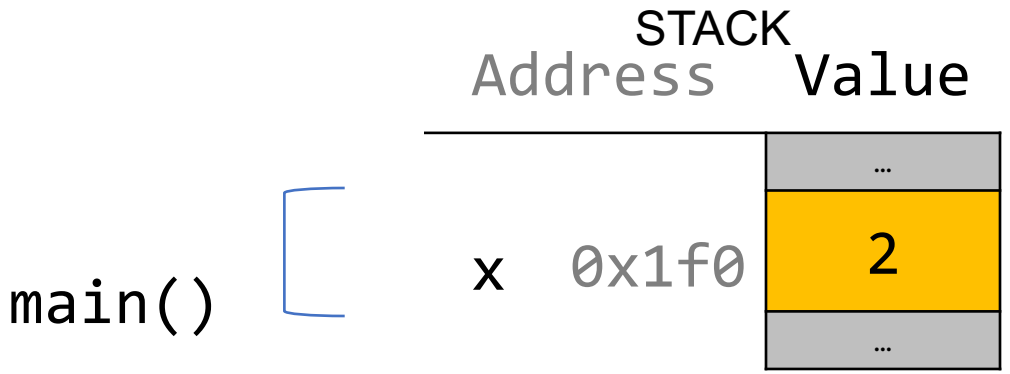
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

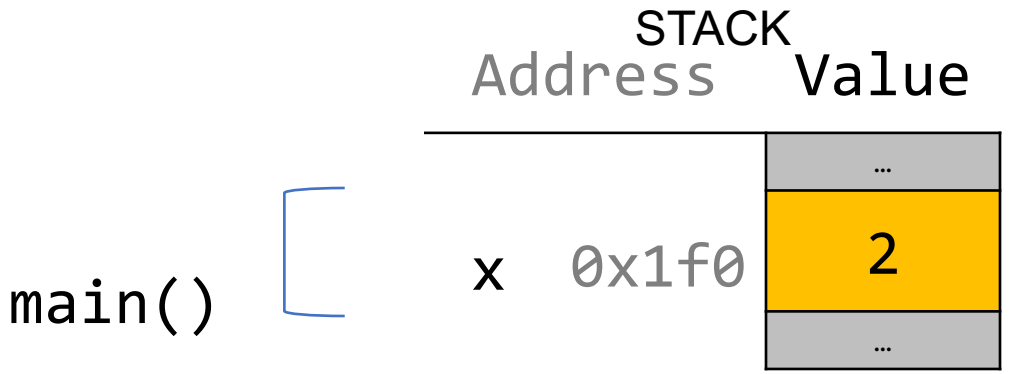
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

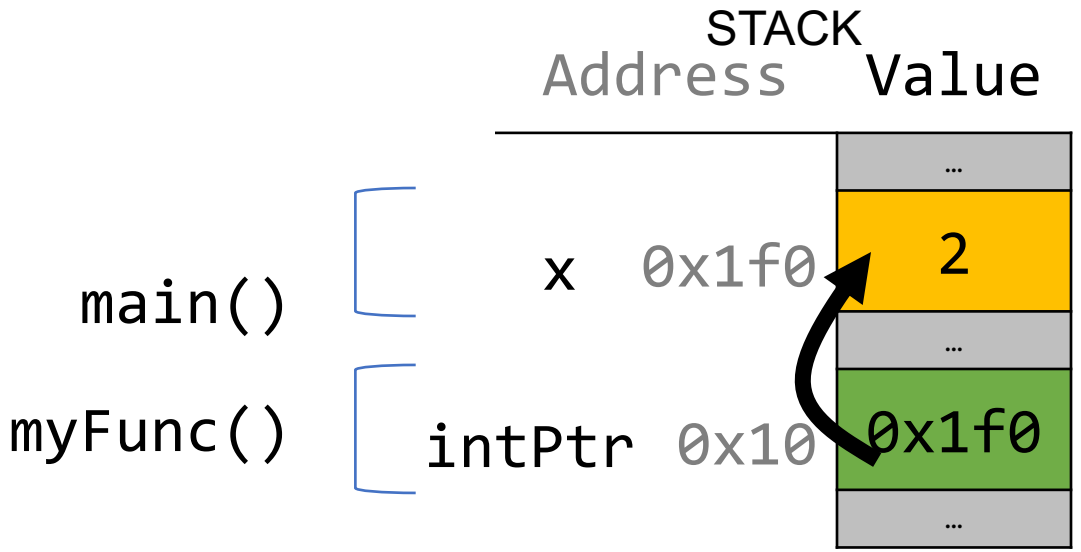
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

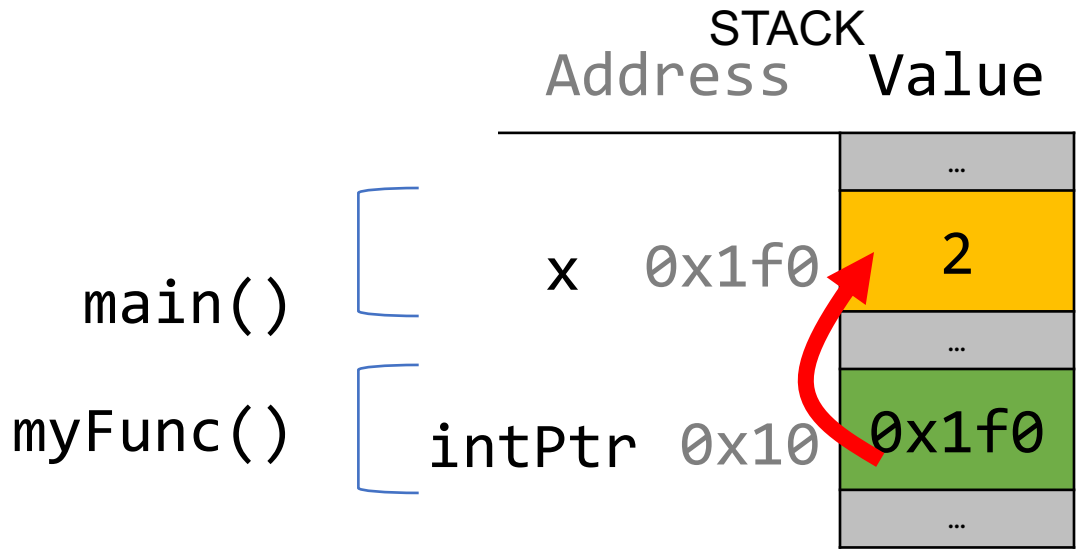
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

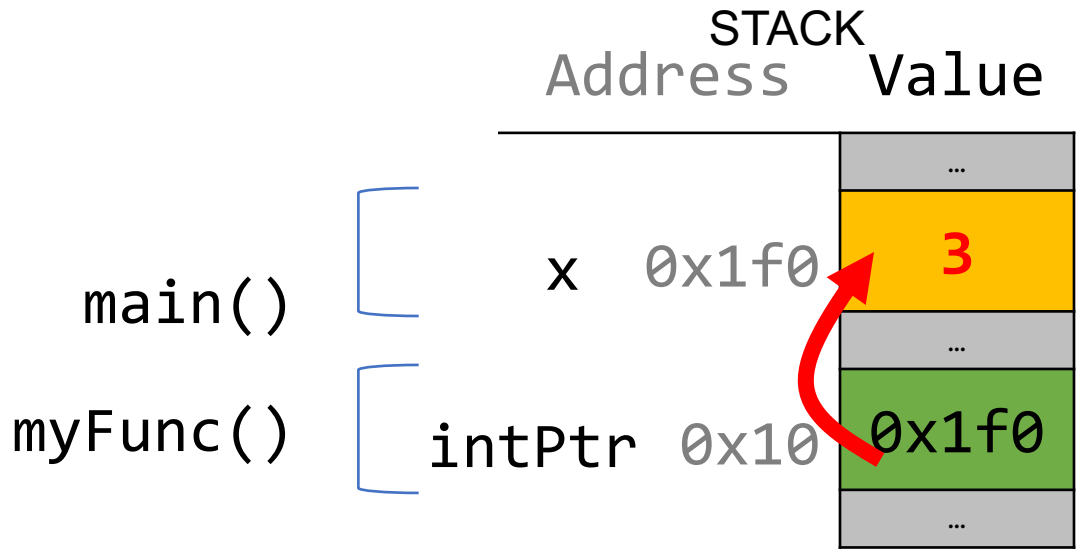
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(&x);
      printf("%d", x);     // 3!
      ...
}
```



Pointers

A pointer is a variable that stores a memory address.

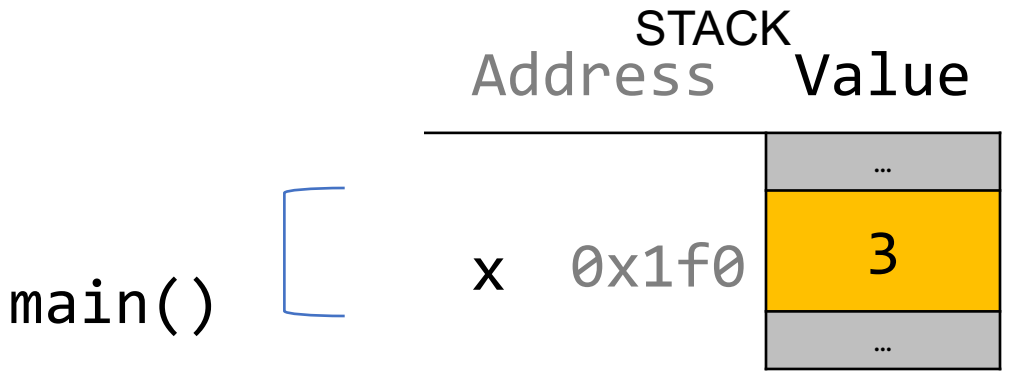
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

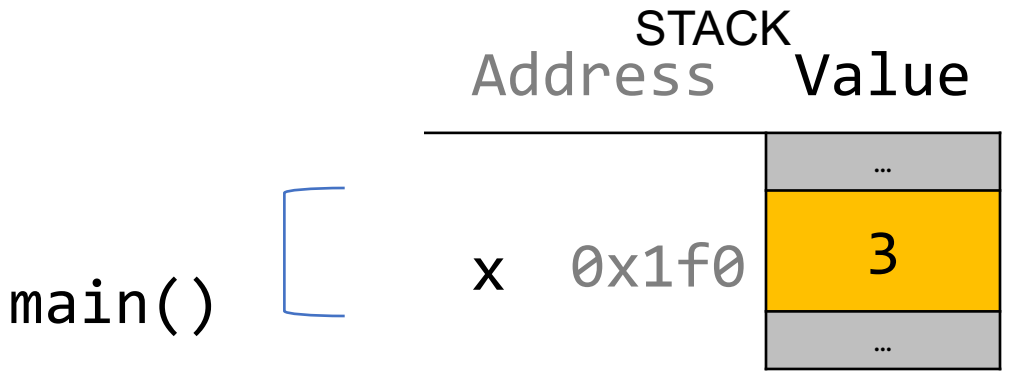
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```



Pointers

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```



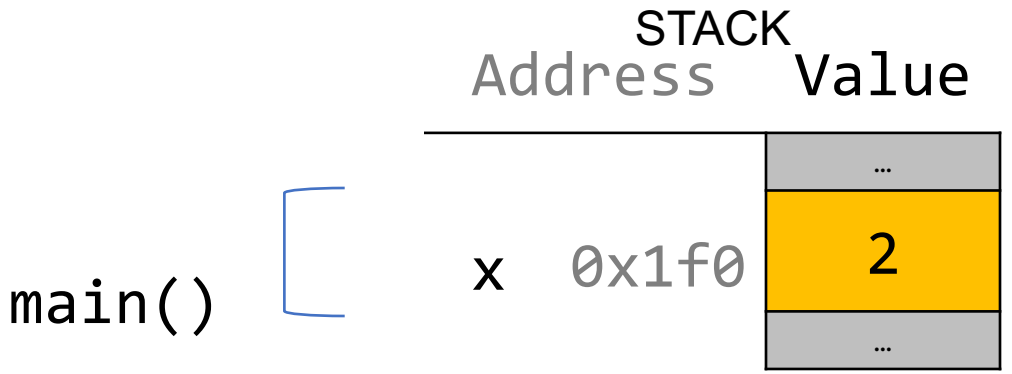
Pointers Summary

- If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**. This makes a copy of the data.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify. This makes a copy of the data's location.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

Pointers

Without pointers, we would make copies.

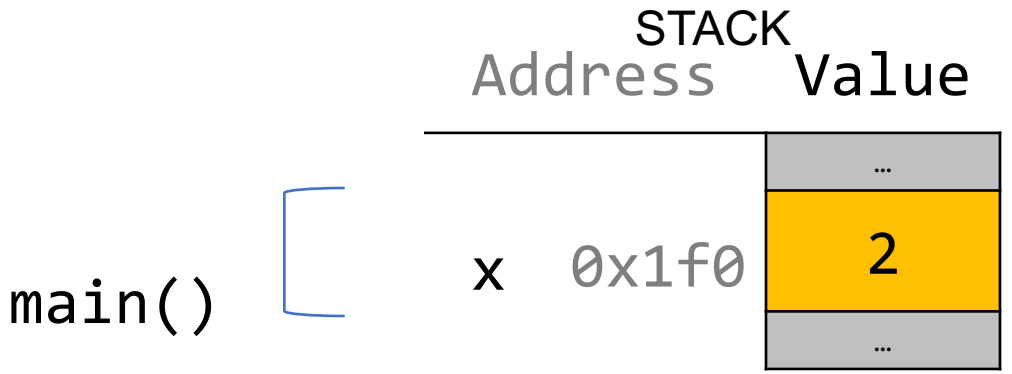
```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```



Pointers

Without pointers, we would make copies.

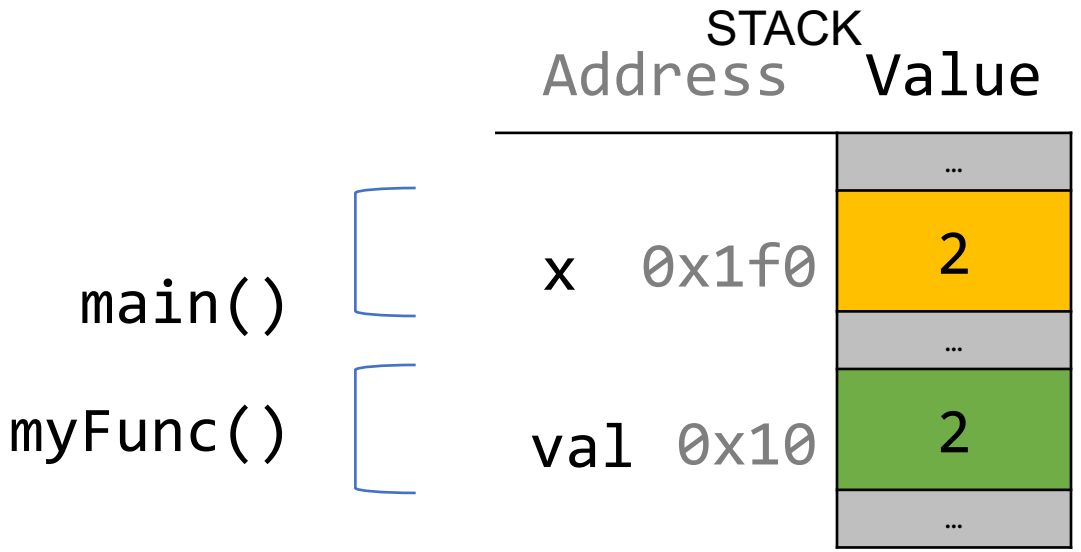
```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```



Pointers

Without pointers, we would make copies.

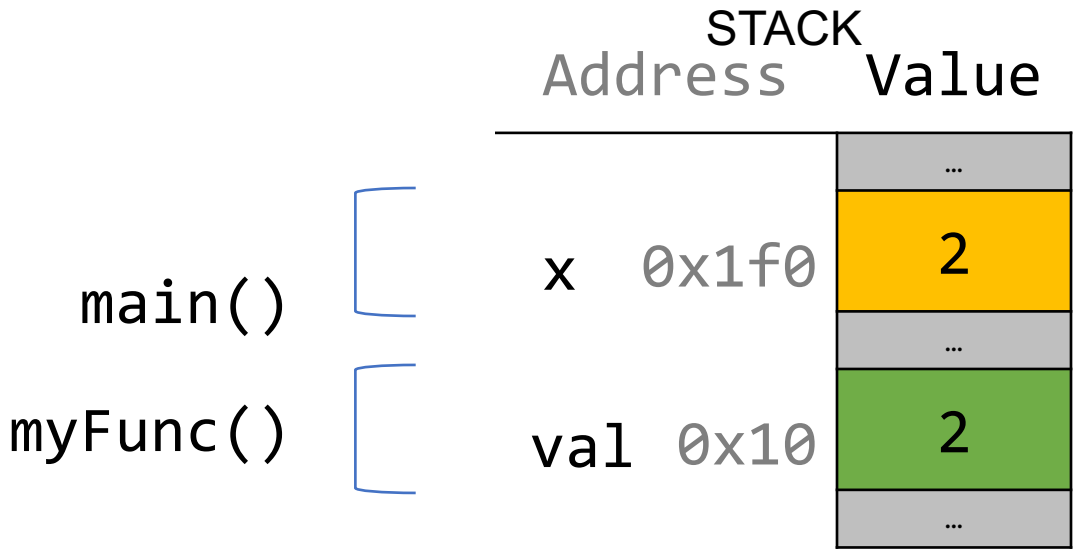
```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```



Pointers

Without pointers, we would make copies.

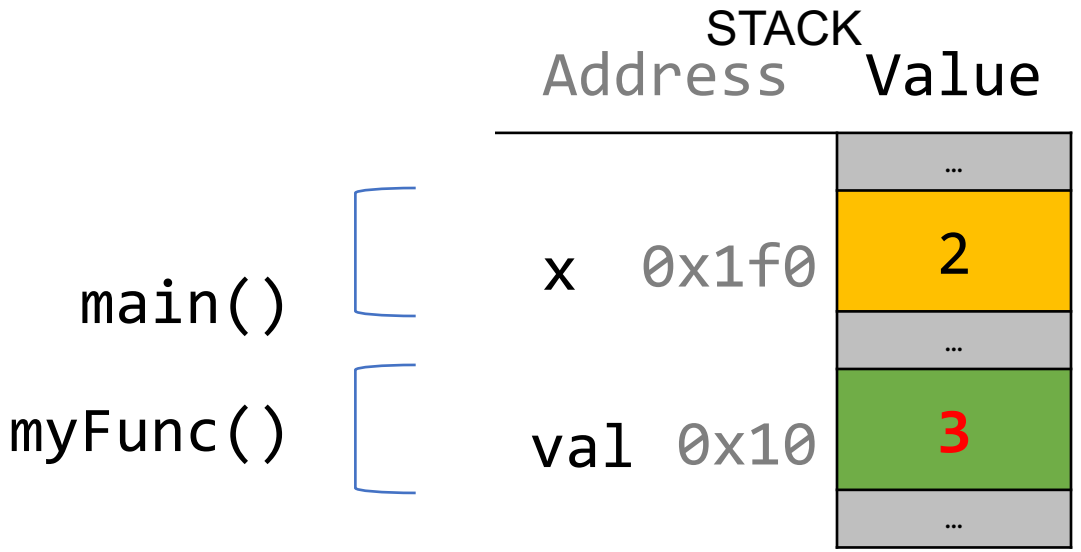
```
void myFunc(int val) {
      val = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(x);
      printf("%d", x);     // 2!
      ...
}
```



Pointers

Without pointers, we would make copies.

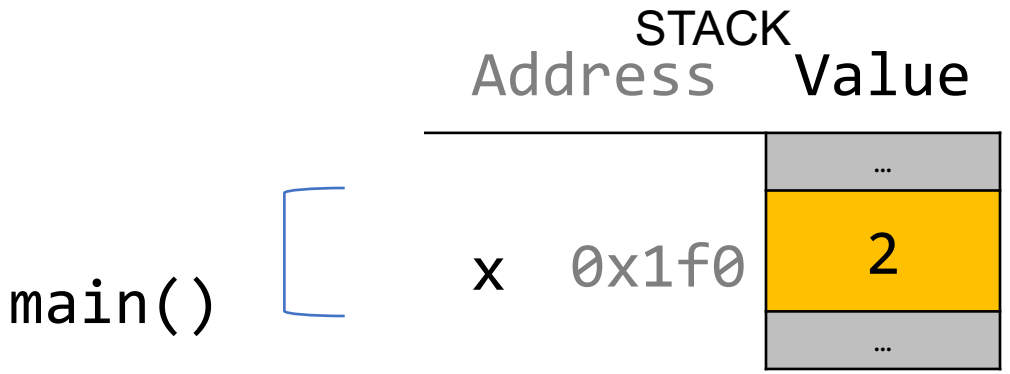
```
void myFunc(int val) {
      val = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(x);
      printf("%d", x);     // 2!
      ...
}
```



Pointers

Without pointers, we would make copies.

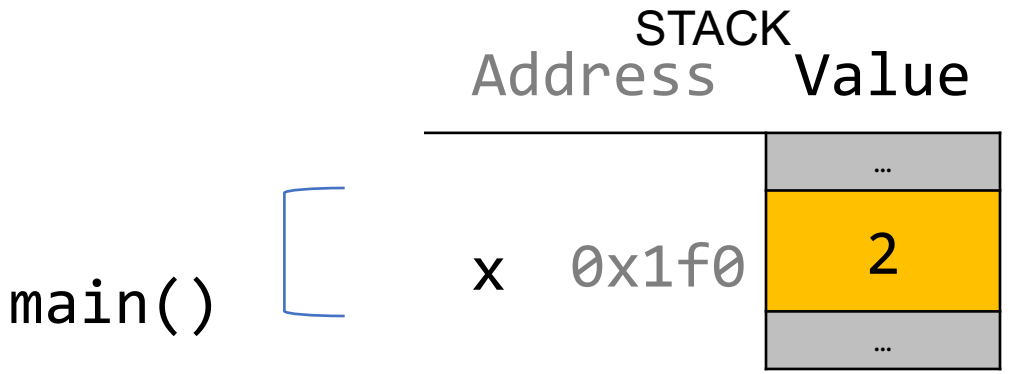
```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```



Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);    // 2!
     ...
}
```



Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Practice: Printing the value of a pointer

`pointer.c`

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Strings In Memory

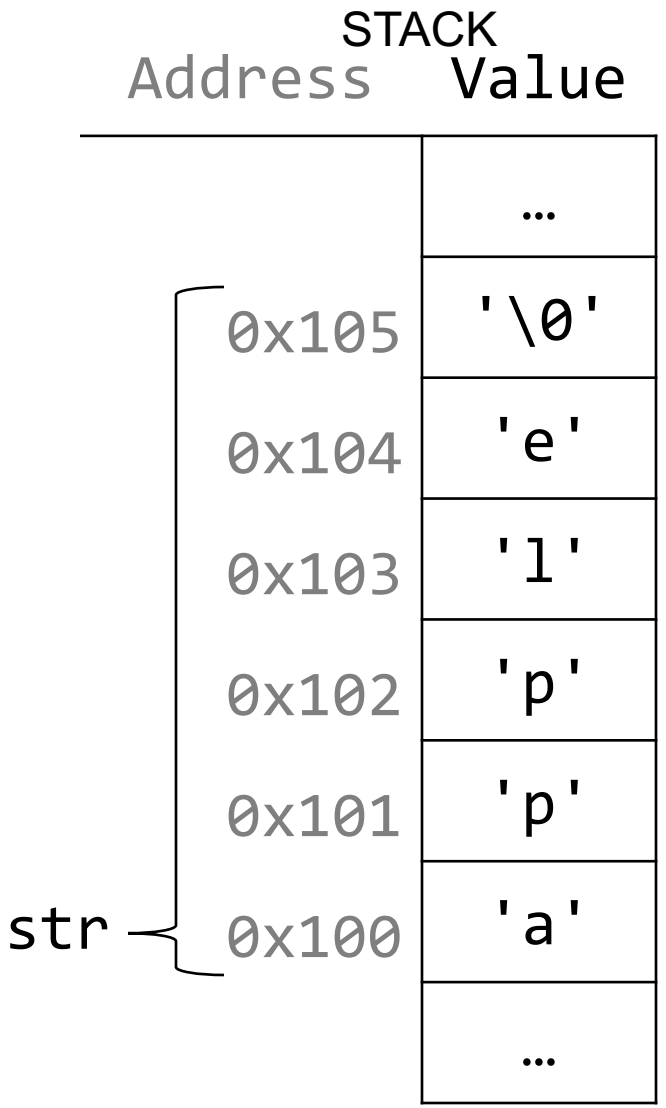
1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as 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.

String Behavior #1: If we create a string as a `char[]`, we can modify its characters because its memory lives in our stack space.

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");
```



String Behavior #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.

Character Arrays

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

```
char str[6];
strcpy(str, "apple");
char str2[8];
strcpy(str2, "apple 2");

str = str2;   // not allowed!
```

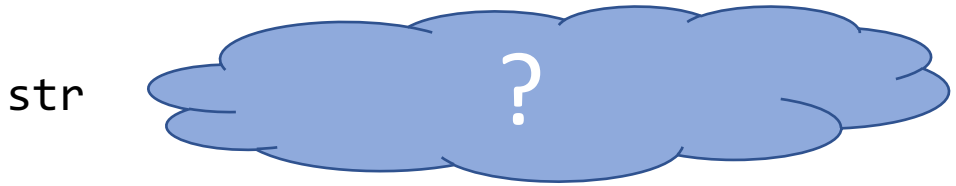
An array's size cannot be changed once we create it; we must create another new array instead.

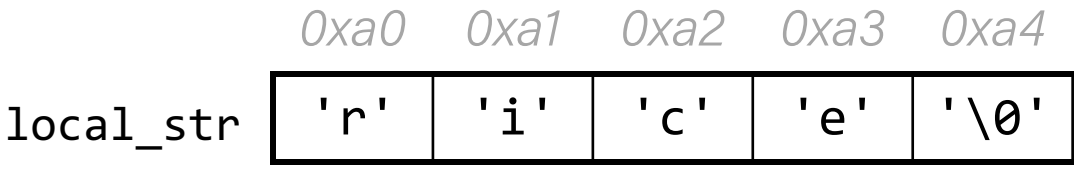
String Behavior #3: If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.

String Parameters

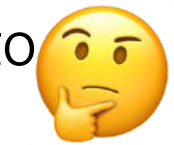
How do you think the parameter `str` is being represented?

```
void fun_times(char *str) {
      ...
}

int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      fun_times(local_str);
      return 0;
}
```





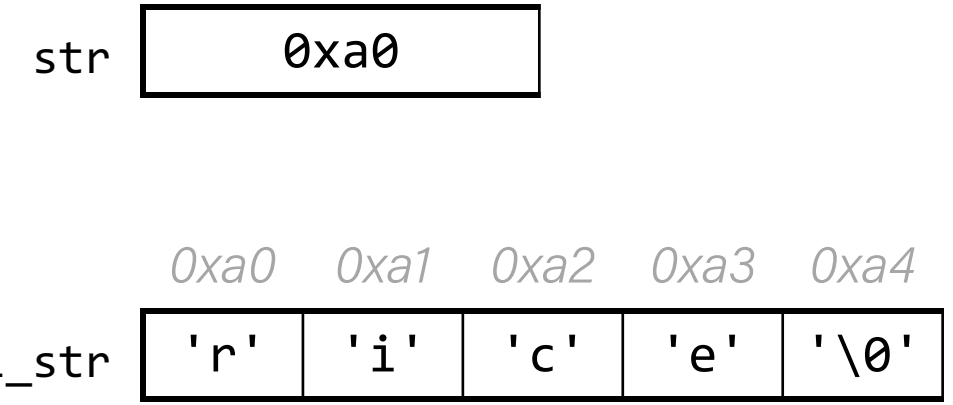
A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`



String Parameters

How do you think the parameter `str` is being represented?

```
void fun_times(char *str) {
      ...
}

int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      fun_times(local_str);
      return 0;
}
```



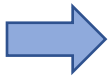
A. A copy of the array `local_str`

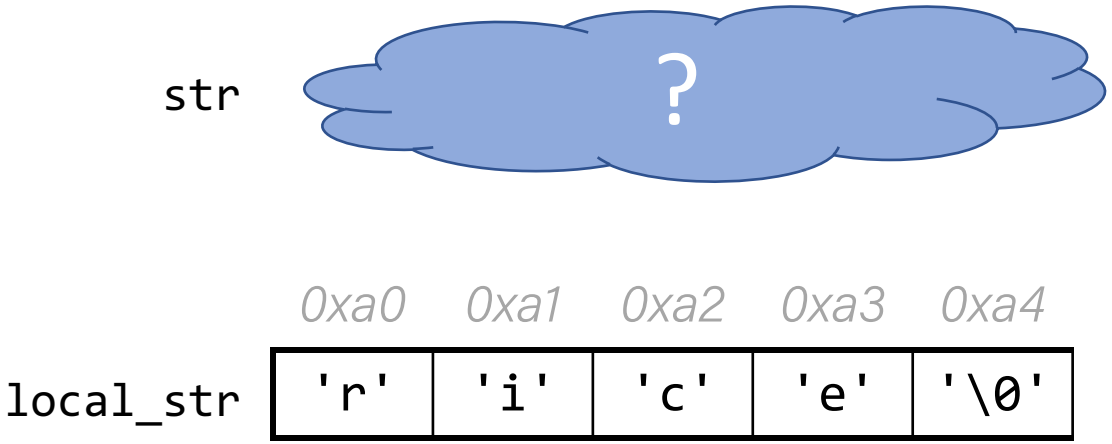
B. A pointer containing an address to the first element in `local_str`

char * Variables

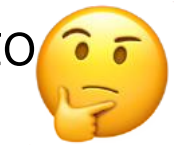
How do you think the local variable `str` is being represented?

```
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str;
    ...
    return 0;
}
```





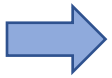
A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`

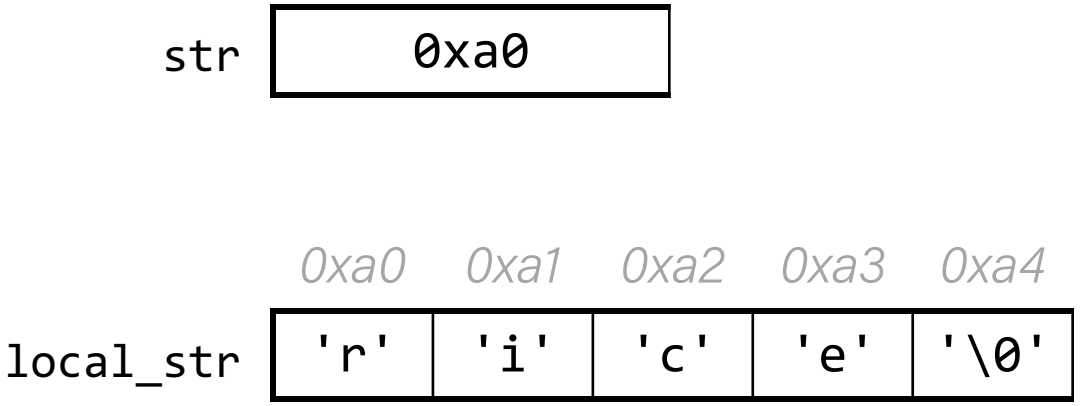


char * Variables

How do you think the local variable `str` is being represented?

```
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str;
    ...
    return 0;
}
```





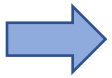
A. A copy of the array `local_str`

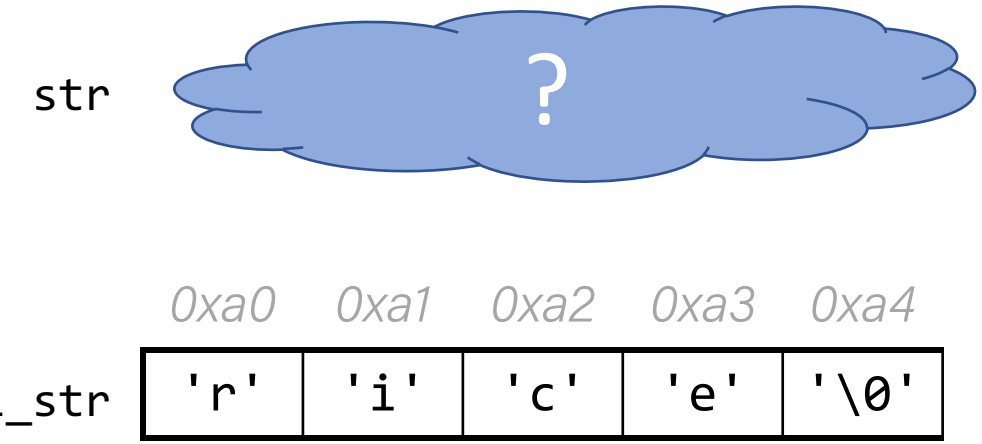
B. A pointer containing an address to the first element in `local_str`

char * Variables

How do you think the local variable `str` is being represented?

```
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```



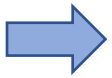


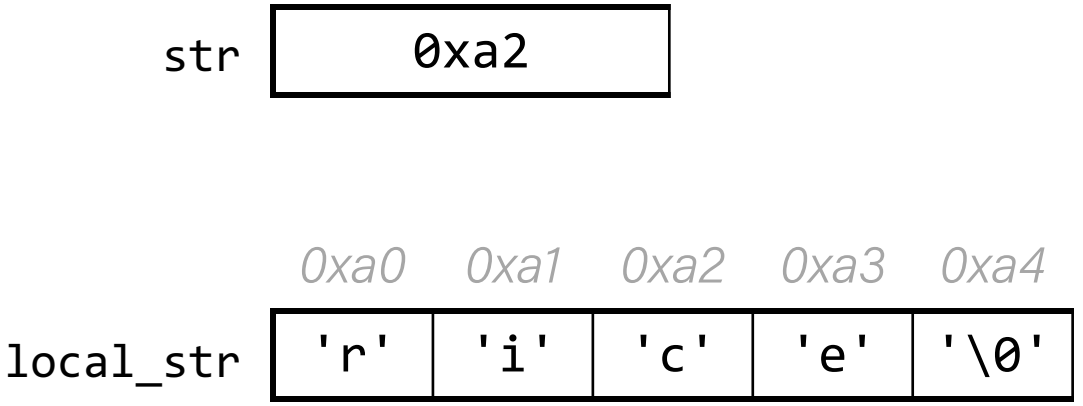
A. A copy of part of the array `local_str`
B. A pointer containing an address to the third element in `local_str`

char * Variables

How do you think the local variable `str` is being represented?

```
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```





A. A copy of part of the array `local_str`
B. A pointer containing an address to the third element in `local_str`

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 *`

String Behavior #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.

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.

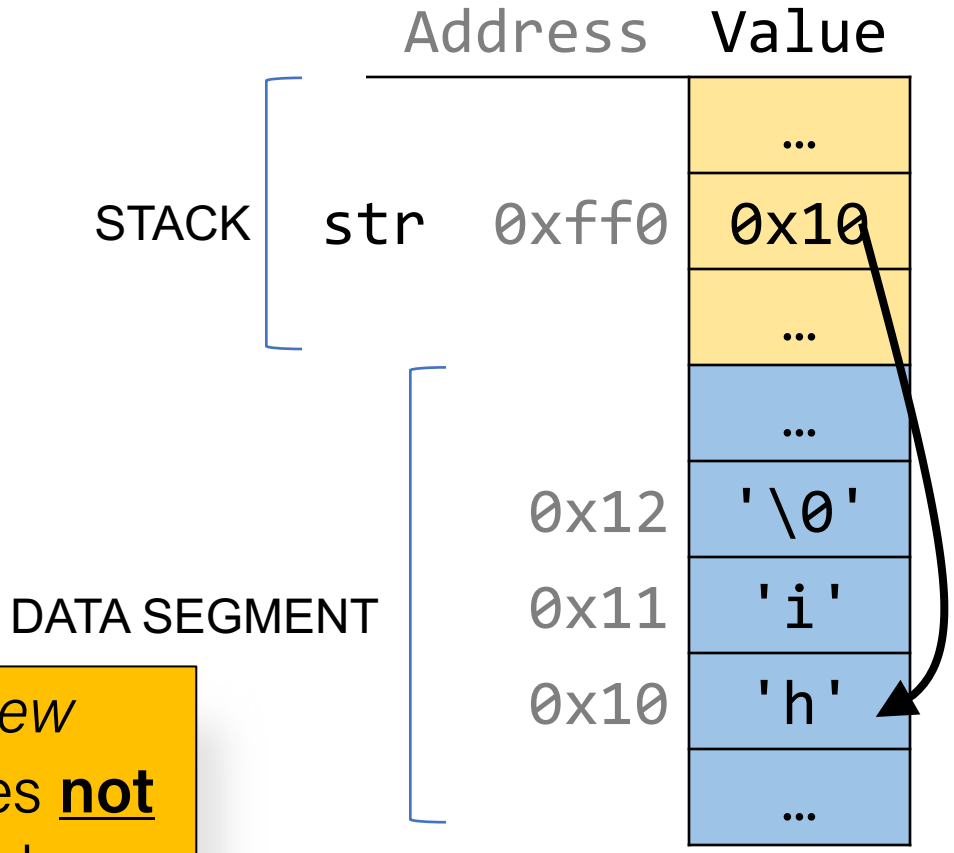
```
char *myString = "Hello, world!";
char *empty = "";

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

`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.

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 *

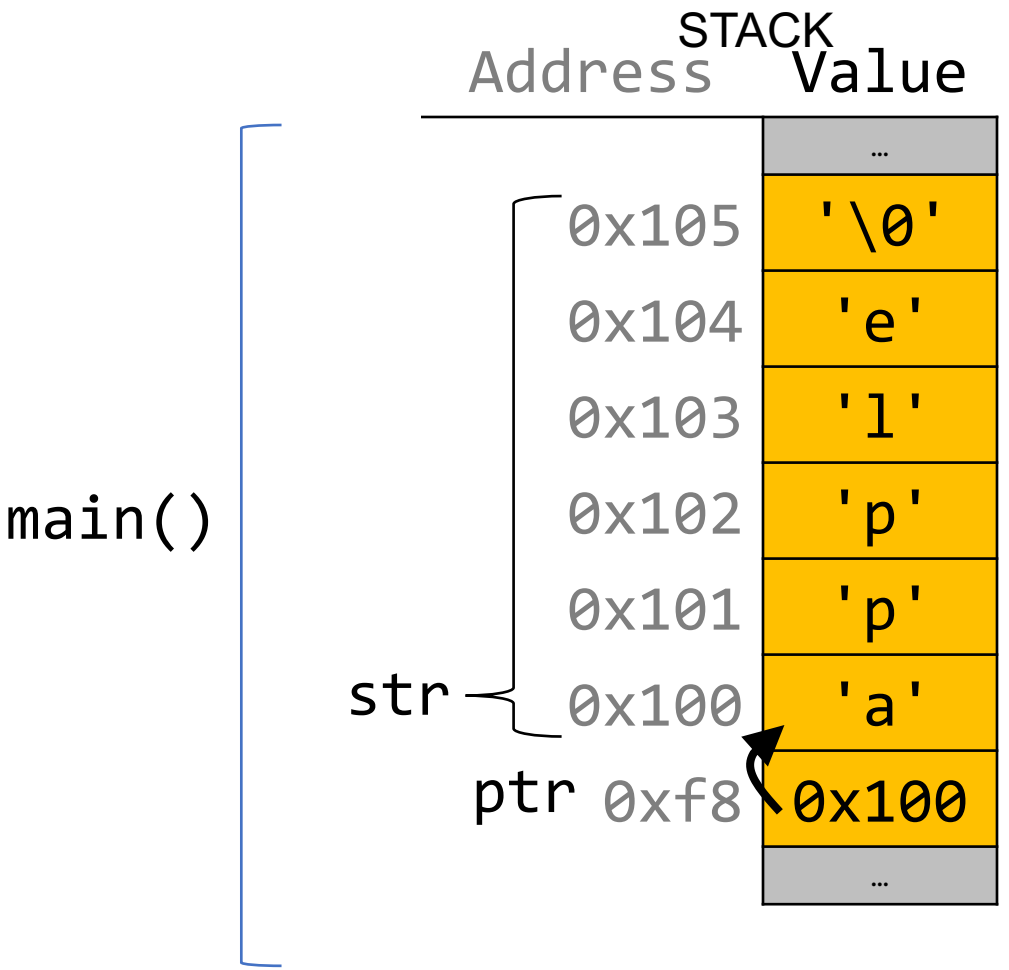
A **char *** variable refers to a single character. We can reassign an existing **char *** pointer to be equal to another **char *** pointer.

```
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.

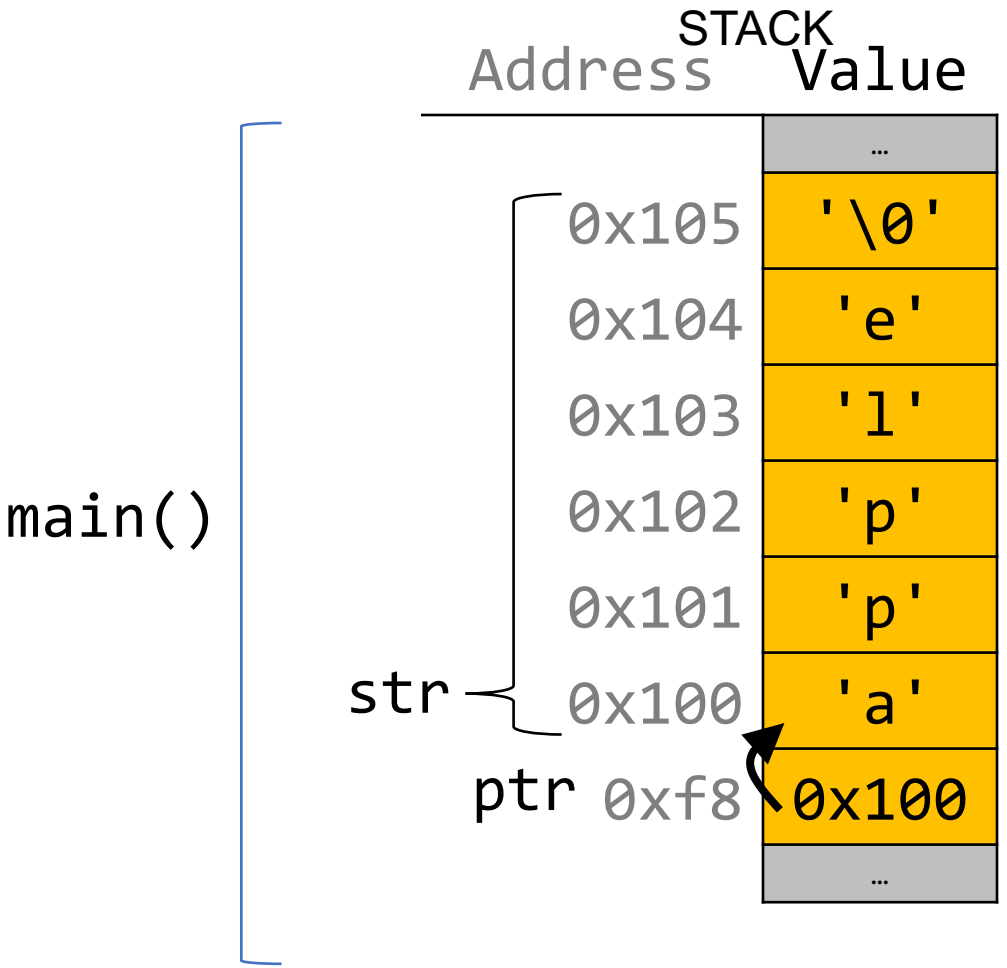
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;
      ...
}
```



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;
      ...
}
```

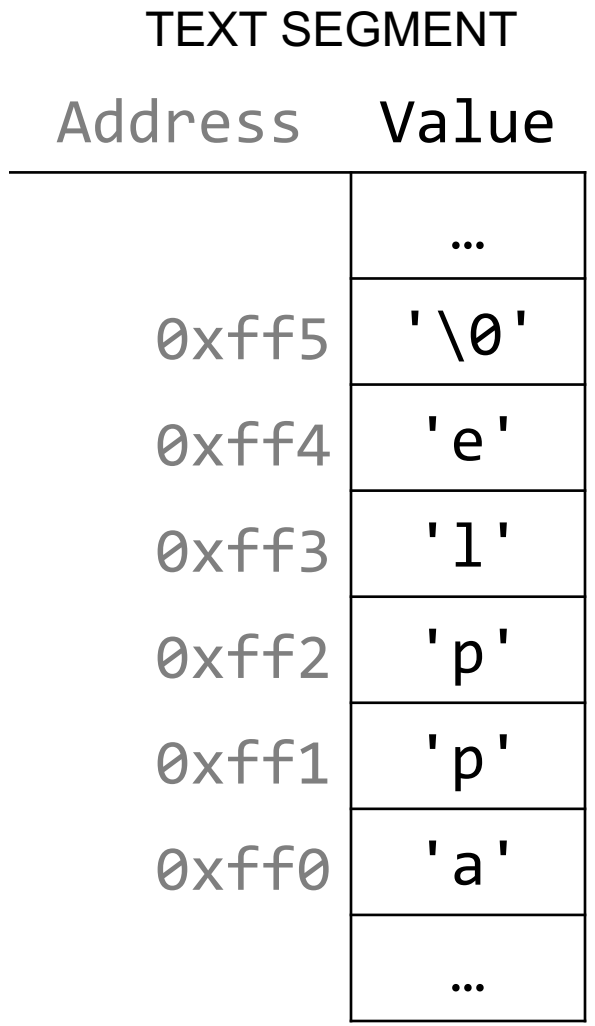


String Behavior #6: Adding an offset to a C string gives us a substring that many places past the first character.

Pointer Arithmetic

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

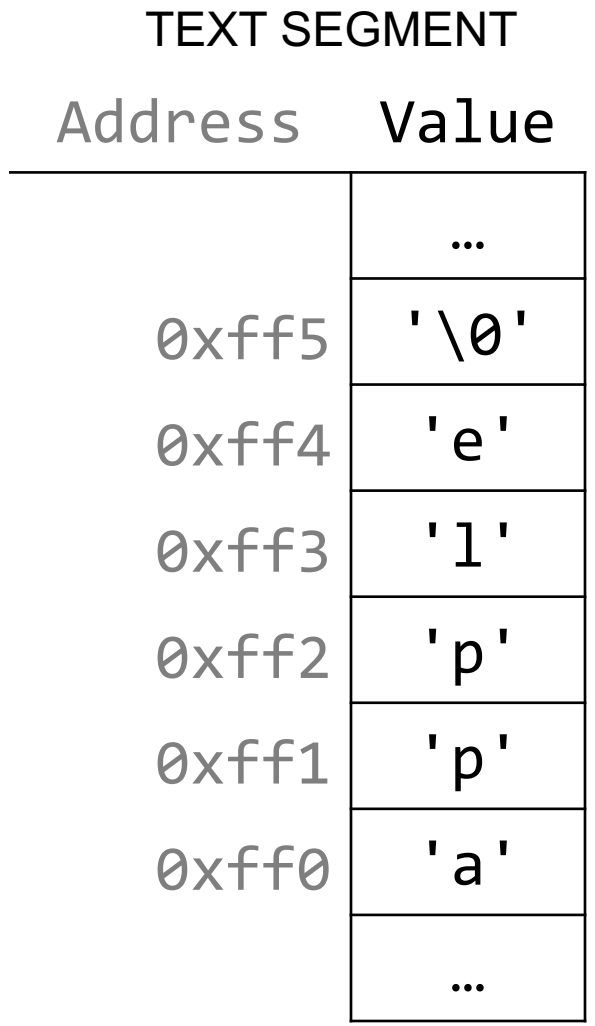
```
char *str = "apple";    // e.g. 0xff0
char *str2 = str + 1;   // e.g. 0xff1
char *str3 = str + 3;   // e.g. 0xff3

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



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'
```

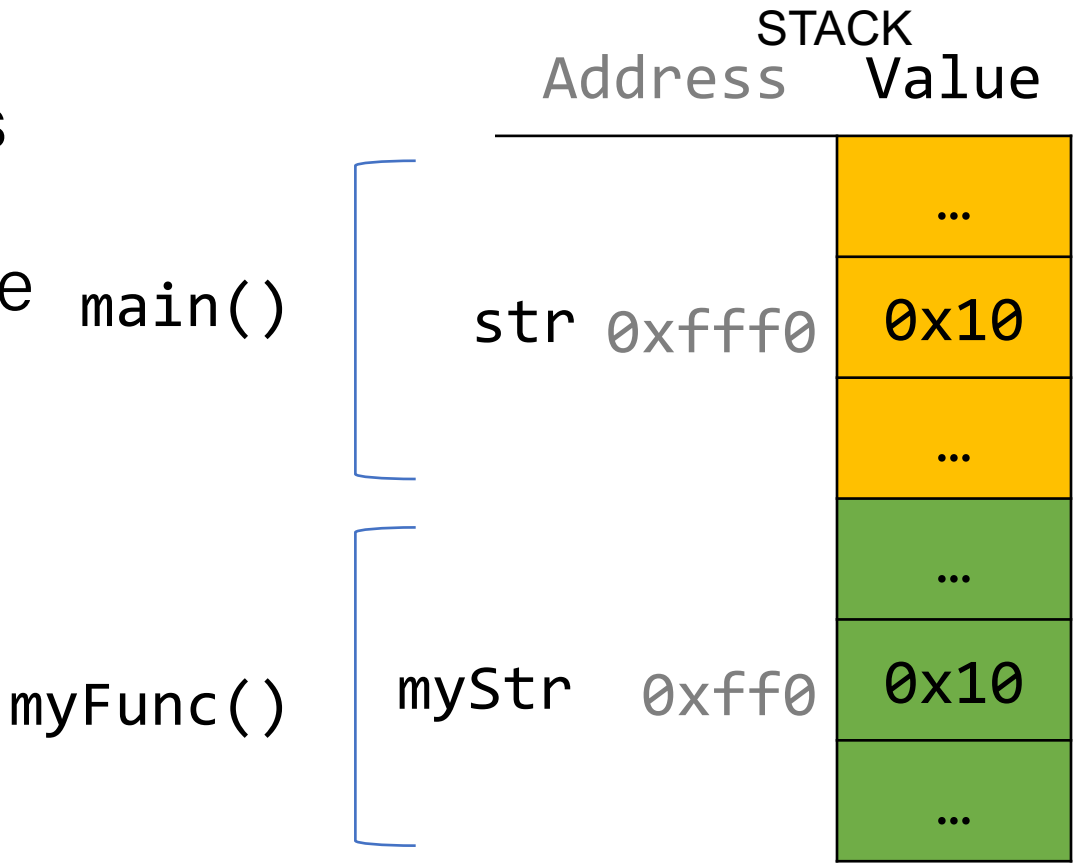


String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.

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.

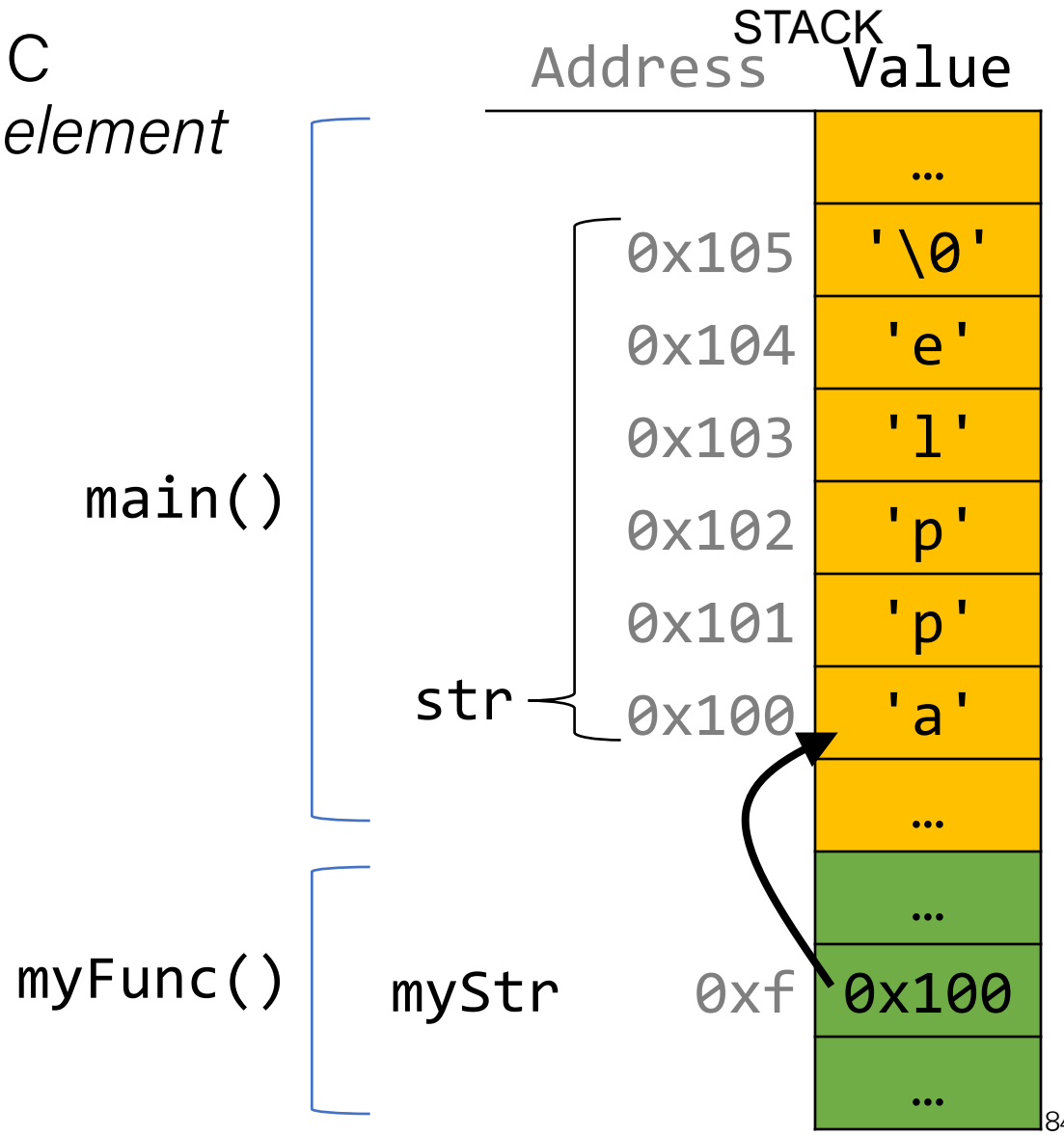
```
void myFunc(char *myStr) {
      ...
}

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.

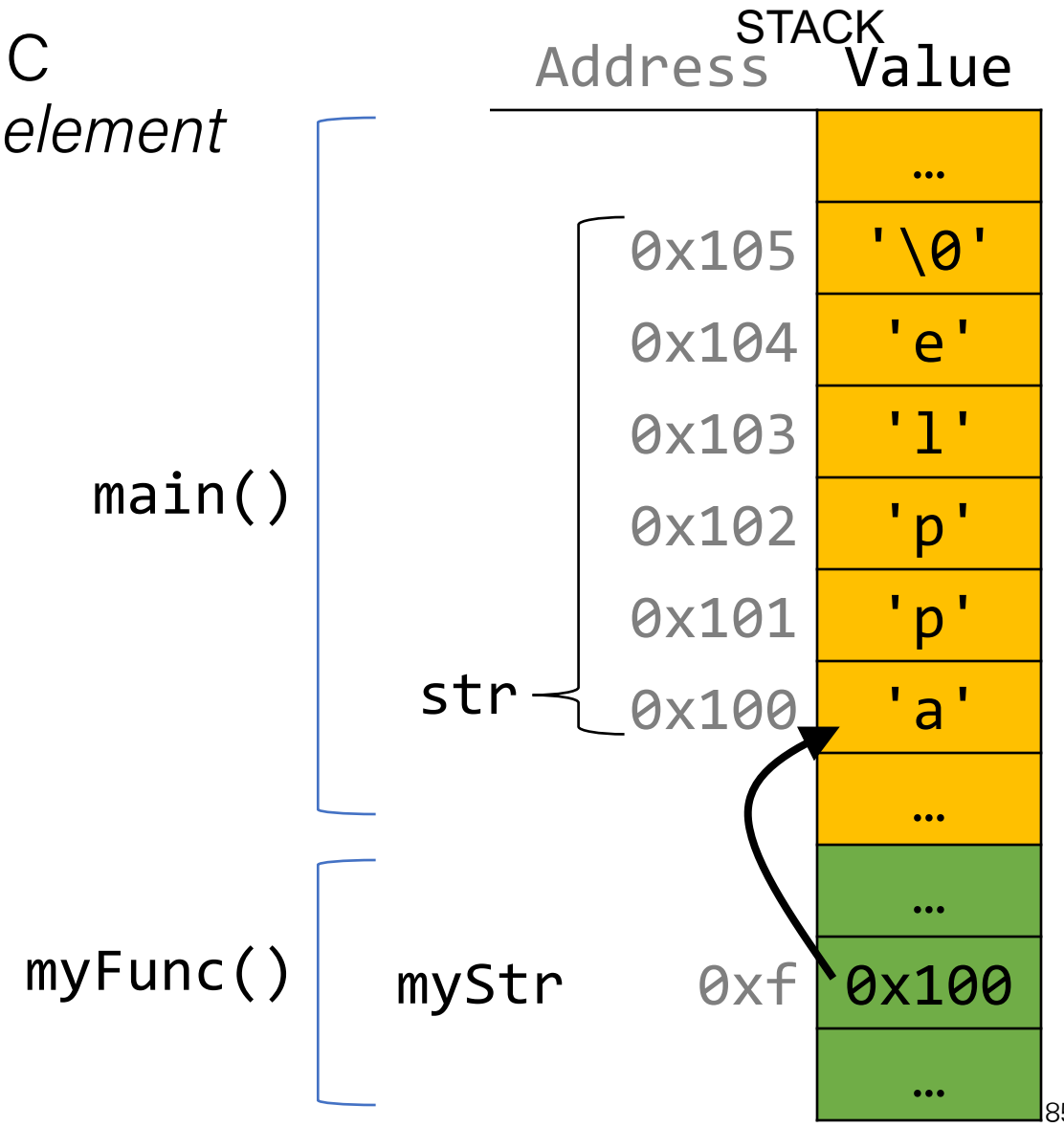
```
void myFunc(char *myStr) {
      ...
}

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.

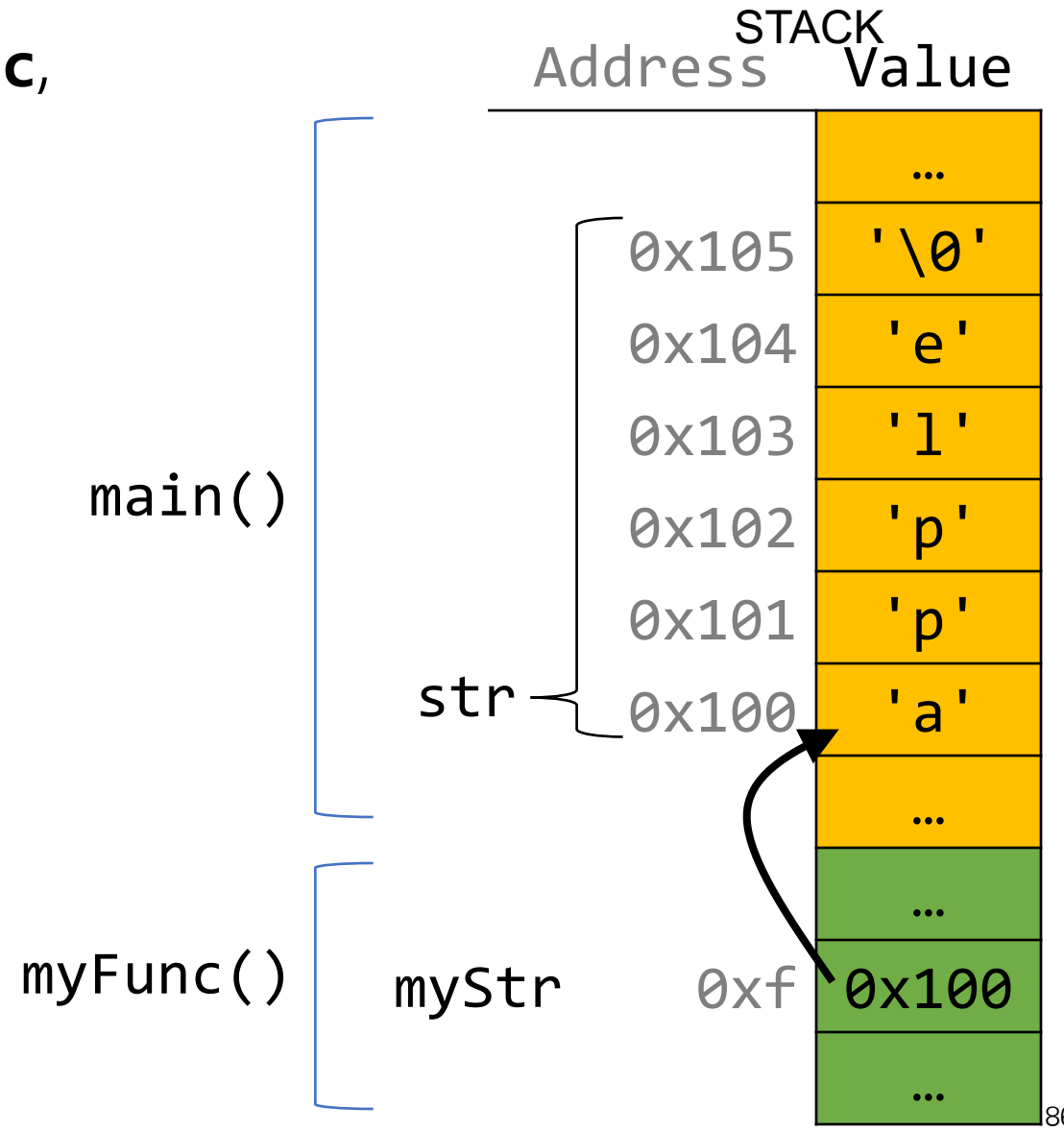
```
void myFunc(char *myStr) {
      ...
}

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**!

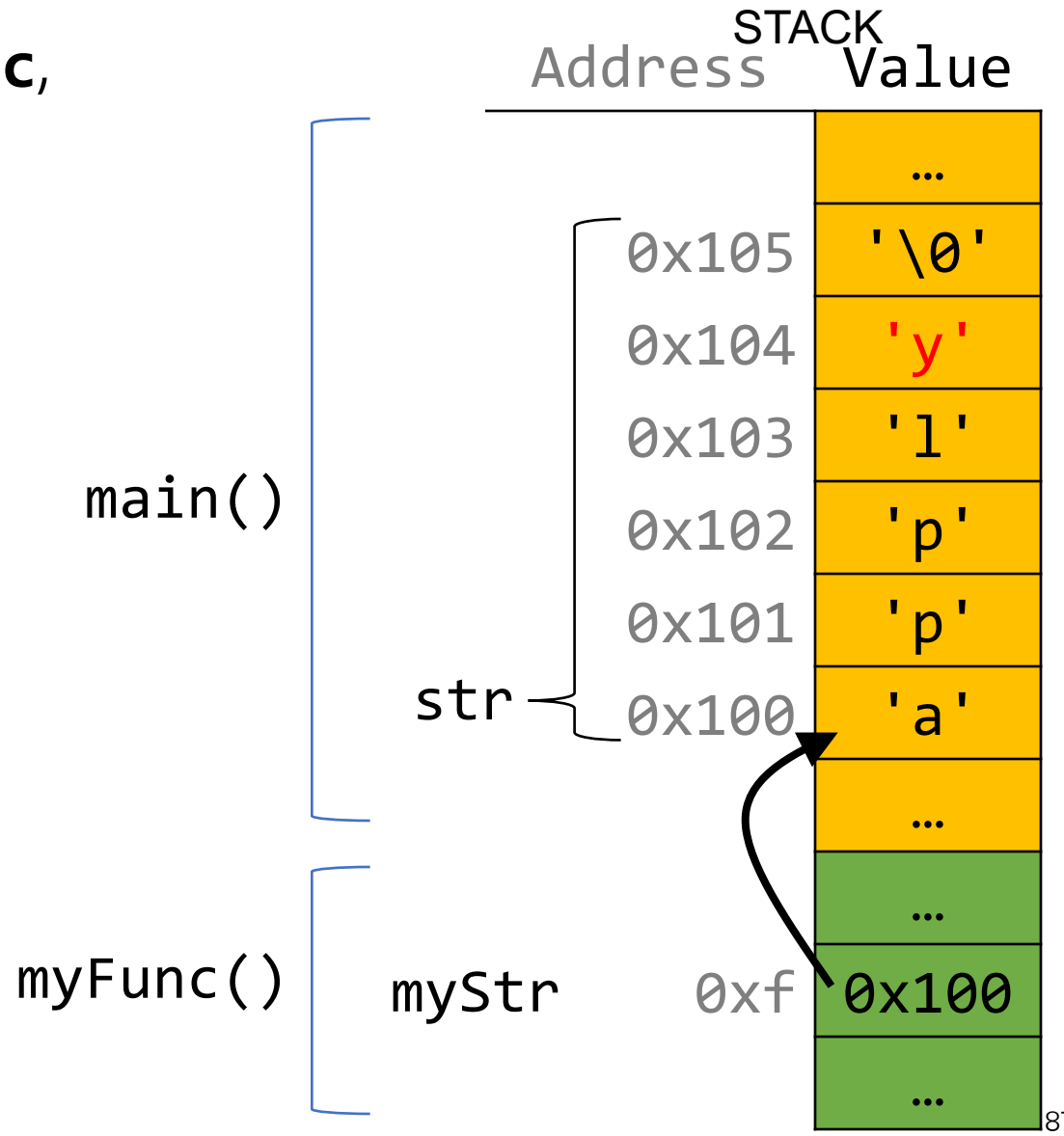
```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

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)

Extra Slides

1. Pointer arithmetic

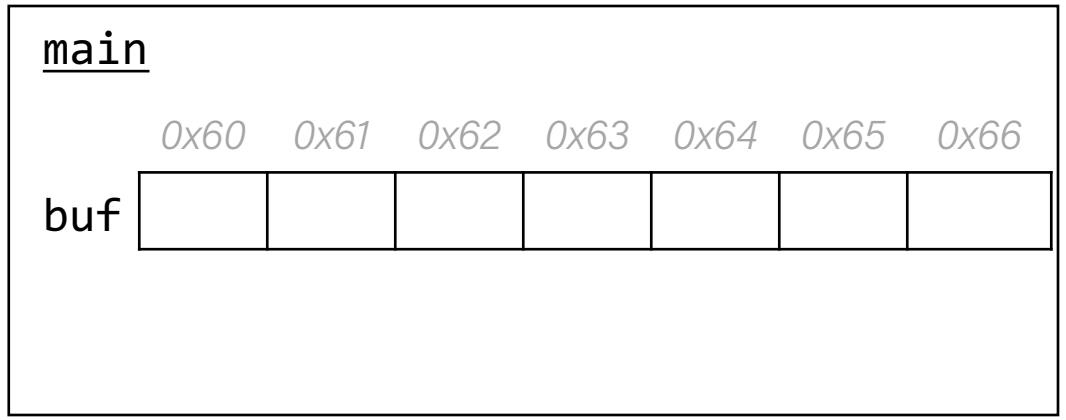
```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

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`

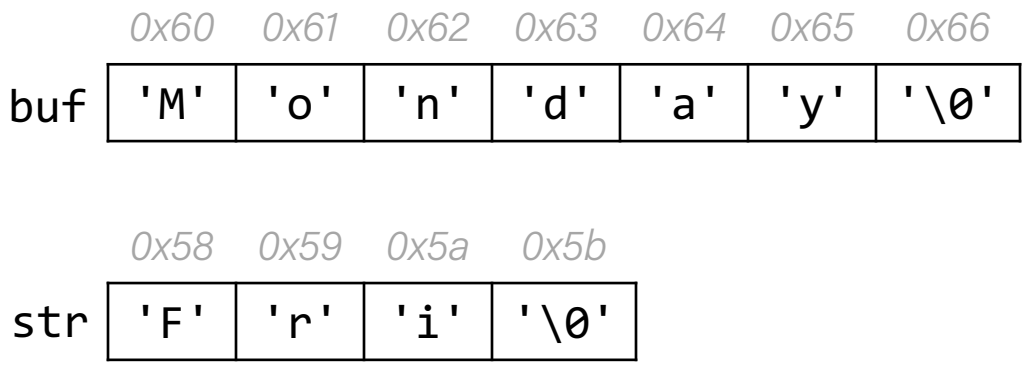
STRCPY(3) Linux Programmer's Manual STRCPY(3)

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:

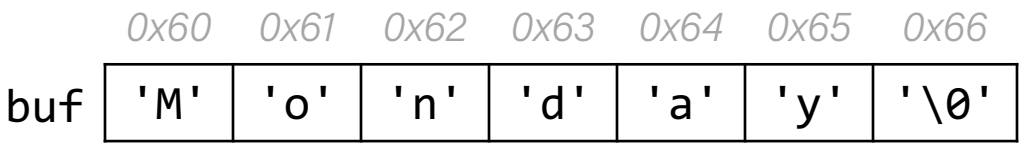


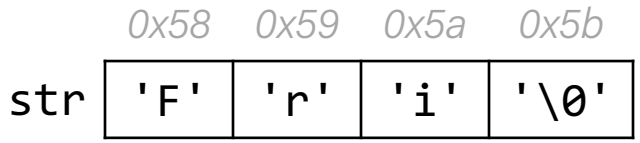

```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

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

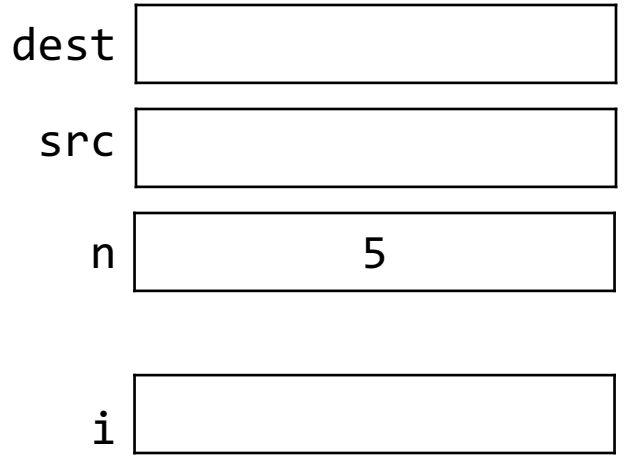
2. Code study: `strncpy`

```
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:
```






```
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. char* vs char[] exercises

Suppose we use a variable `str` as follows:

```
// initialize as below
A  str = str + 1;
B  str[1] = 'u';
C  printf("%s", str)
```

For each of the following initializations:

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

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

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

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

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

3. char* vs char[] exercises

Suppose we use a variable `str` as follows:

```
   // initialize as below
A  str = str + 1;
B  str[1] = 'u';
C  printf("%s", str)
```

For each of the following initializations:

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

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

Line A: Compile error (cannot reassign array)

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

Line B: Segmentation fault (string literal)

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

Prints `eulo3`

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

Line B: Segmentation fault (string literal)

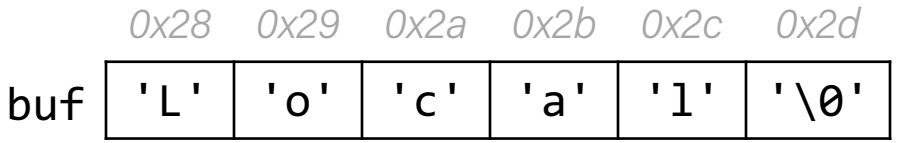
4. Bonus: Tricky addresses

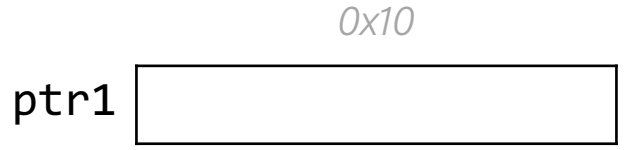
```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);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

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

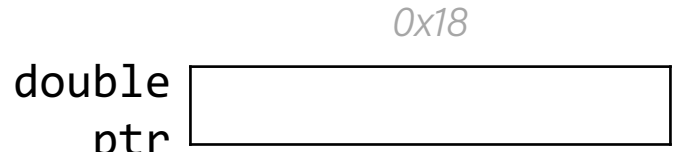
What is stored in each variable? (We cover double pointers later in the course)

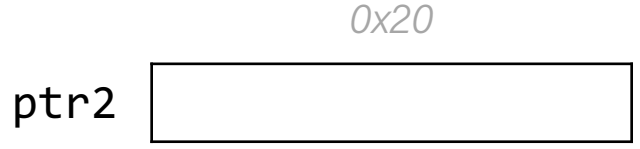
4. Bonus: Tricky addresses

```
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

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Next time: Arrays and Pointers