CS107, Lecture 5
More C Strings

Reading: K&R (1.6, 5.5, Appendix B3) or Essential C section 3

Plan For Today

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

Plan For Today

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

C Strings

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

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`','`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'!'`	`'\0'`

String operations use the null-terminating character to find the end of the string.

E.g. `strlen` calculates string length by counting up the characters it sees *before* reaching a null-terminating character.

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***.

C Strings As Parameters

Regardless of how you created the string, when you pass a string as a parameter it is always passed as a **char ***. **char *** still lets you use bracket notation to access individual characters (*How? We'll see later today!*).

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

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

Buffer Overflows

- It is your responsibility to ensure that memory operations you perform don't improperly read or write memory.
  - E.g. don't copy a string into a space that is too small!
  - E.g. don't ask for the string length of an uninitialized string!
- The **Valgrind** tool may be able to help track down memory-related issues.
  - See cs107.stanford.edu/resources/valgrind
  - We'll talk about Valgrind more when we talk about dynamically-allocated memory.

Demo: Memory Errors

Plan For Today

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

Arrays of Strings

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

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

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

```
char *stringArray[] = {
     "my string 1",
     "my string 2",
     "my string 3"
};
```

Arrays of Strings

You can access each string using bracket syntax:

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

When an array of strings is passed as a parameter, it is passed as a *pointer to the first element of the string array*. This is what **argv** is in **main**! This means you write the parameter type as:

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

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

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

Practice: Password Verification

Plan For Today

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

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.

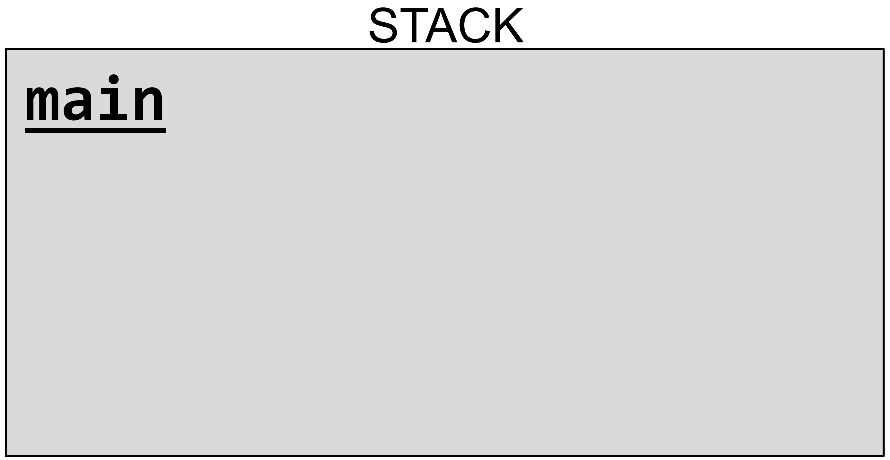
Pointers

```c
int x = 2;

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

// Dereference the pointer to get the data it points to.
// (* means "dereference")
printf("%d", *xPtr);    // prints 2
```

Pointers

A pointer is just a variable that stores a memory address!

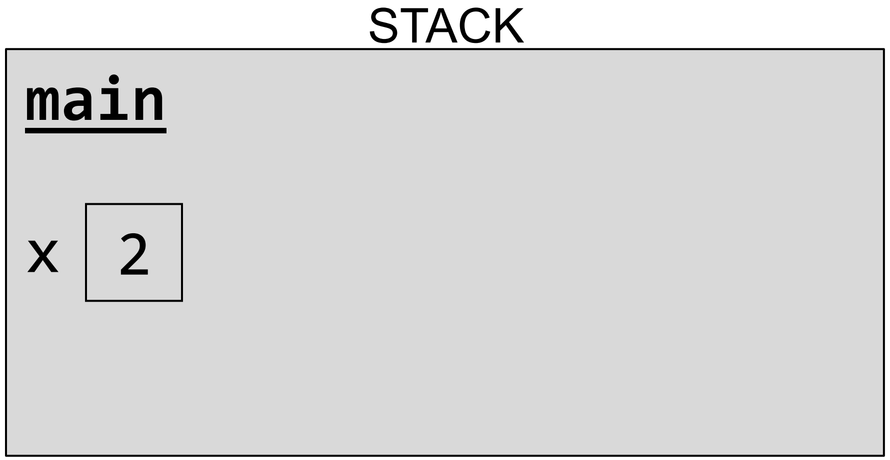
```
void myFunc(int *intPtr) {
	*intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

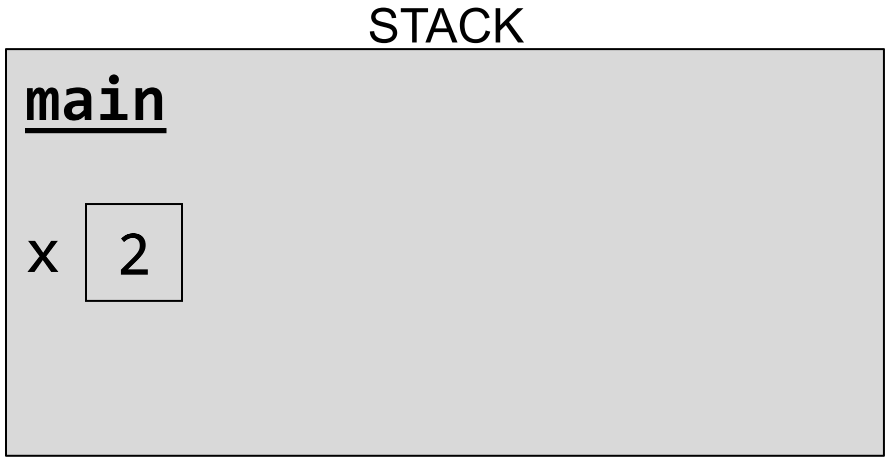
```c
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

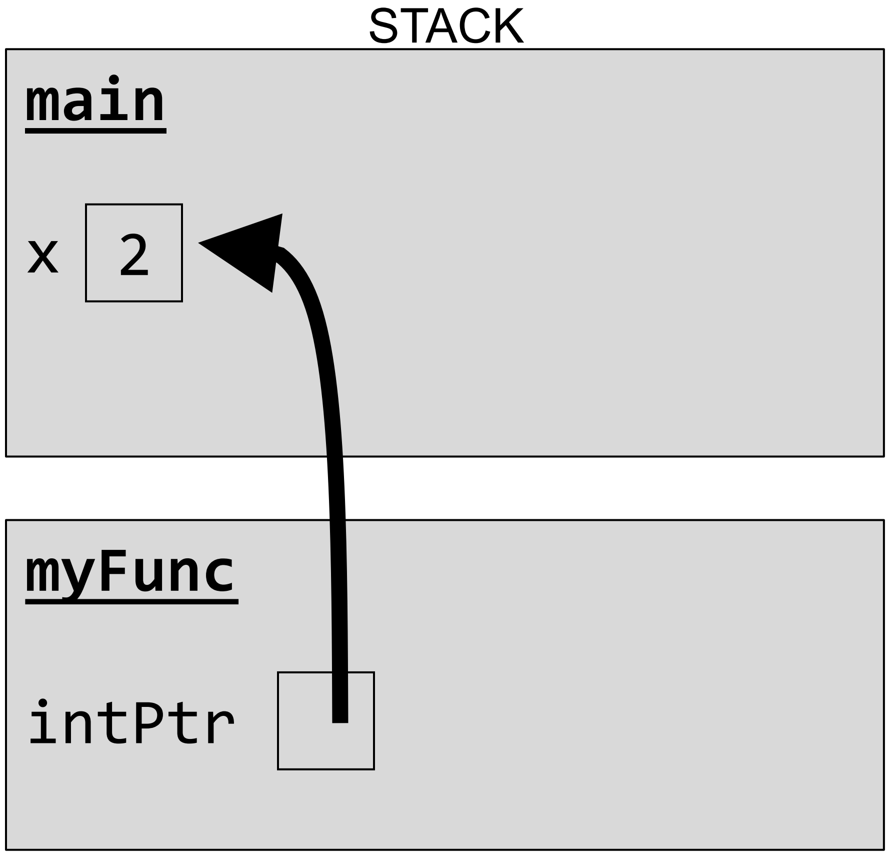
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

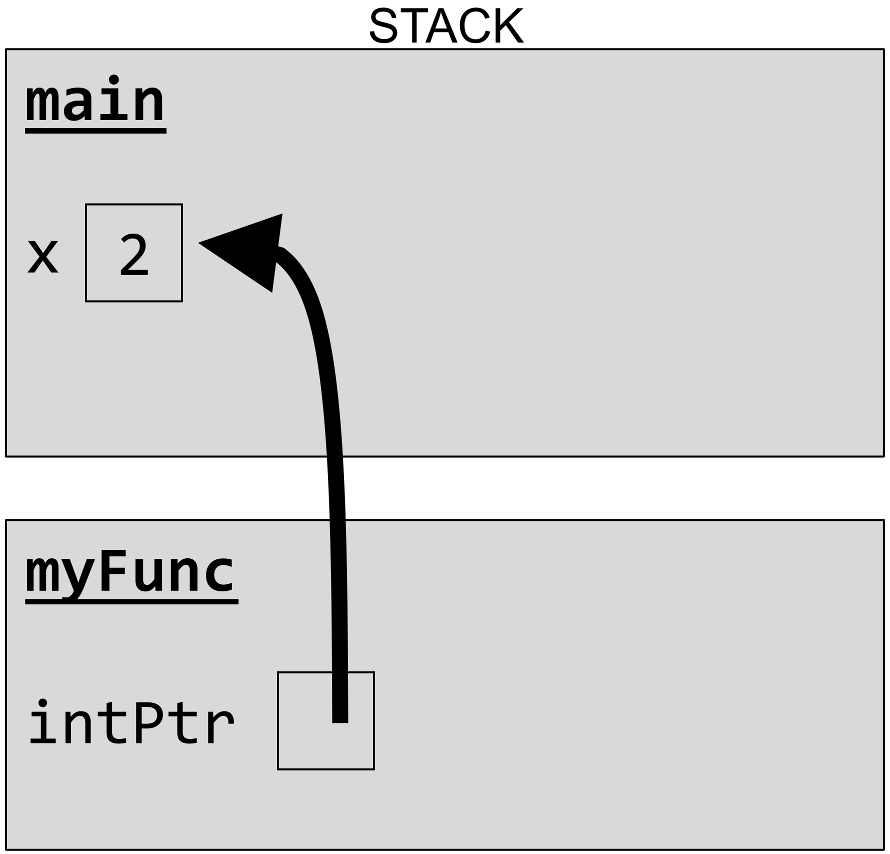
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

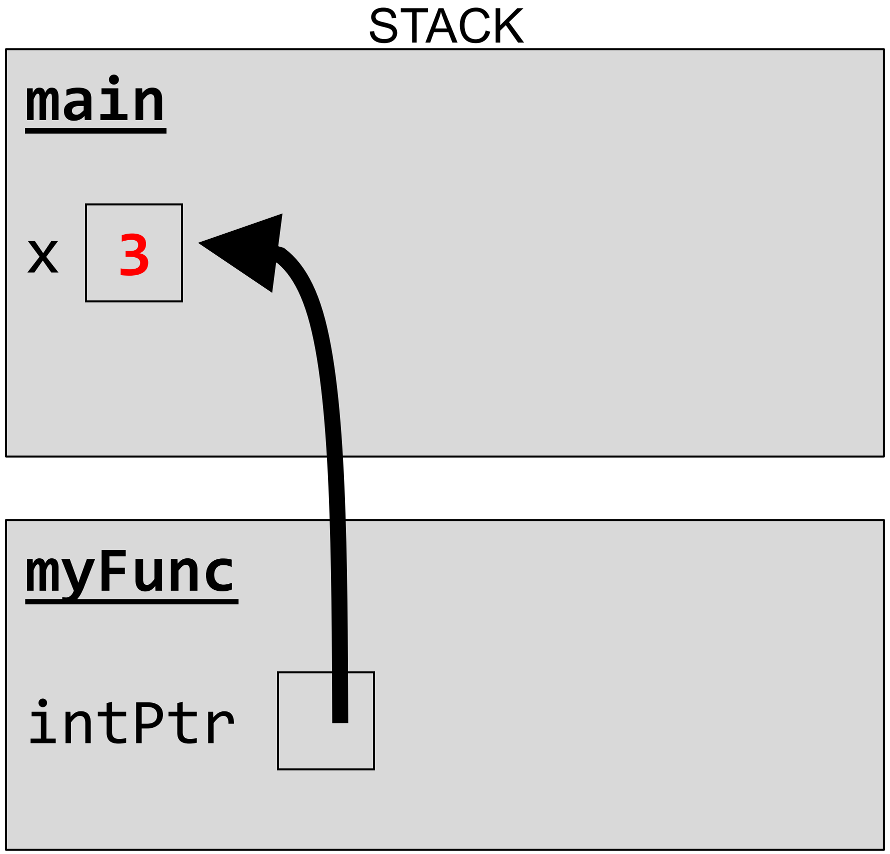
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

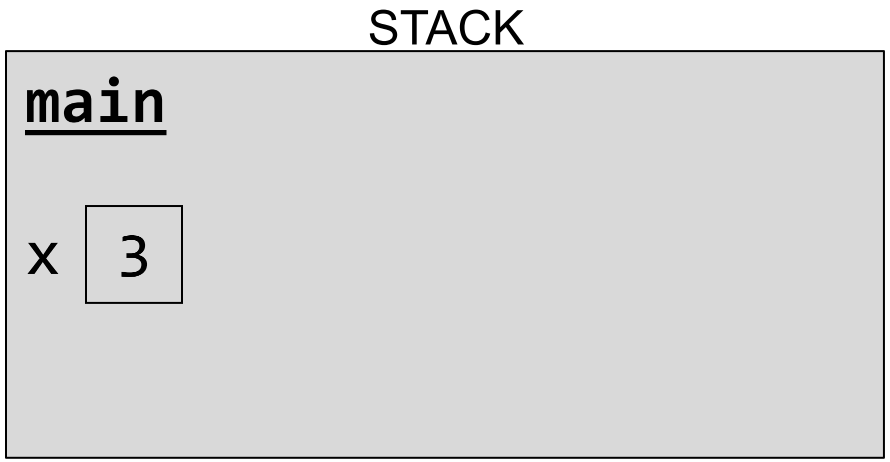
```
void myFunc(int *intPtr) {
	*intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

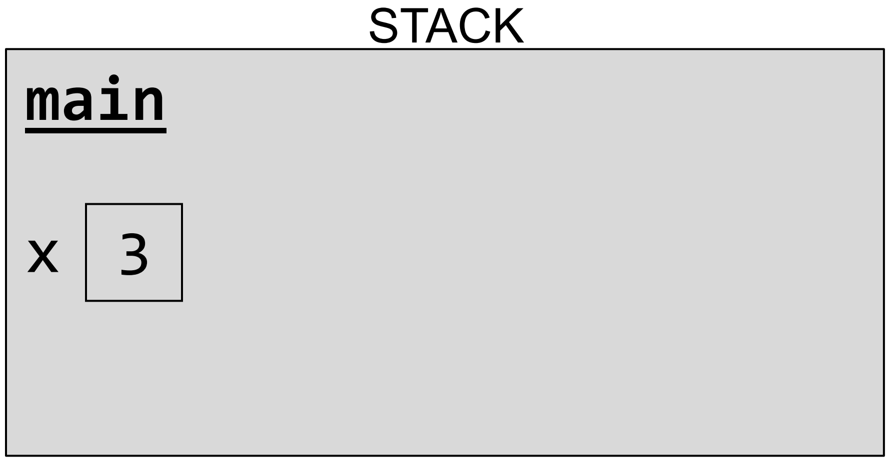
```c
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

A pointer is just 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

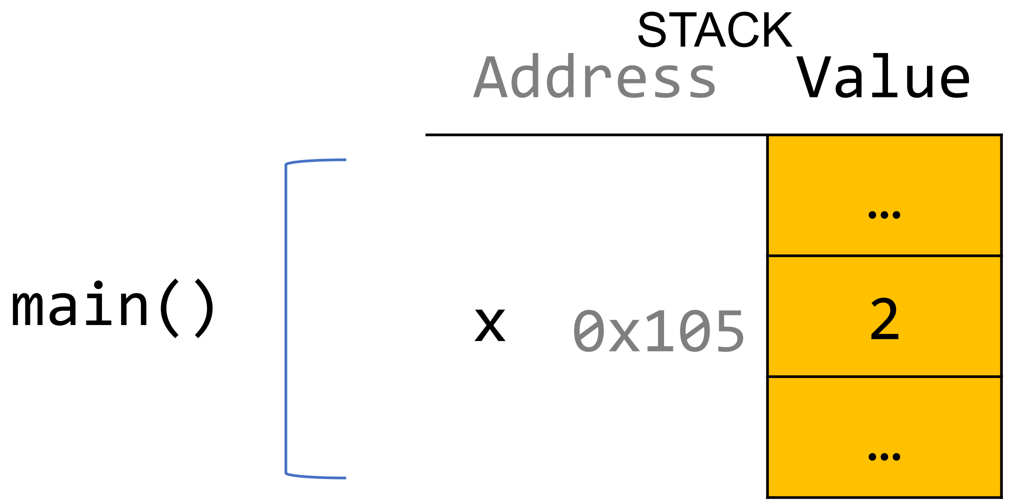
A pointer is just a variable that stores a memory address!

```
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 just a variable that stores a memory address!

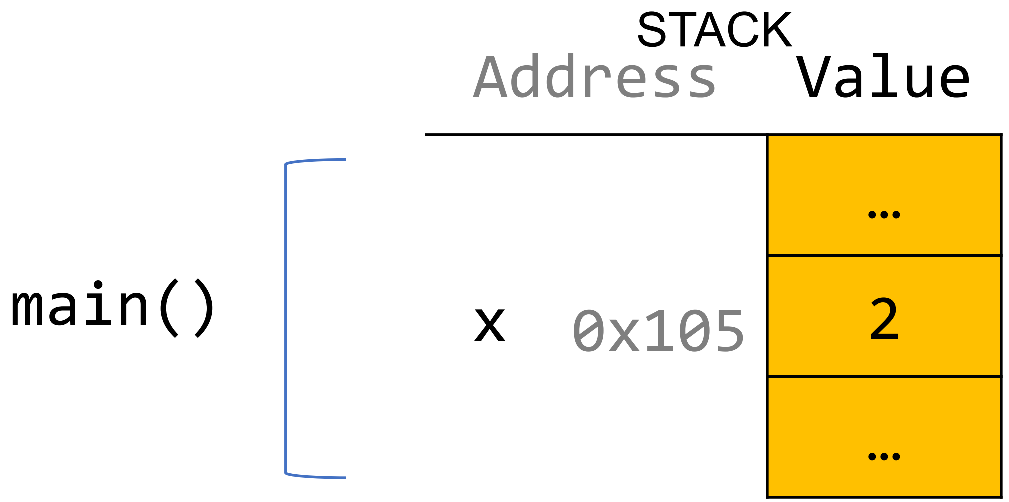
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

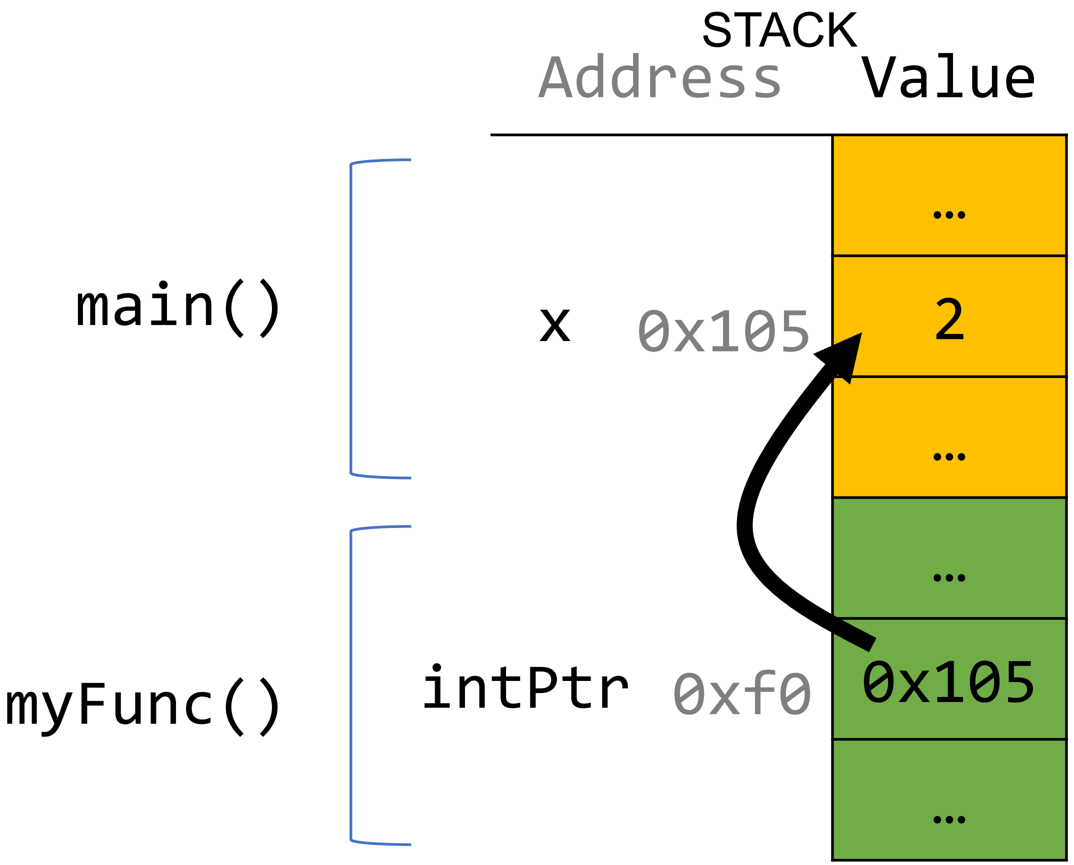
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

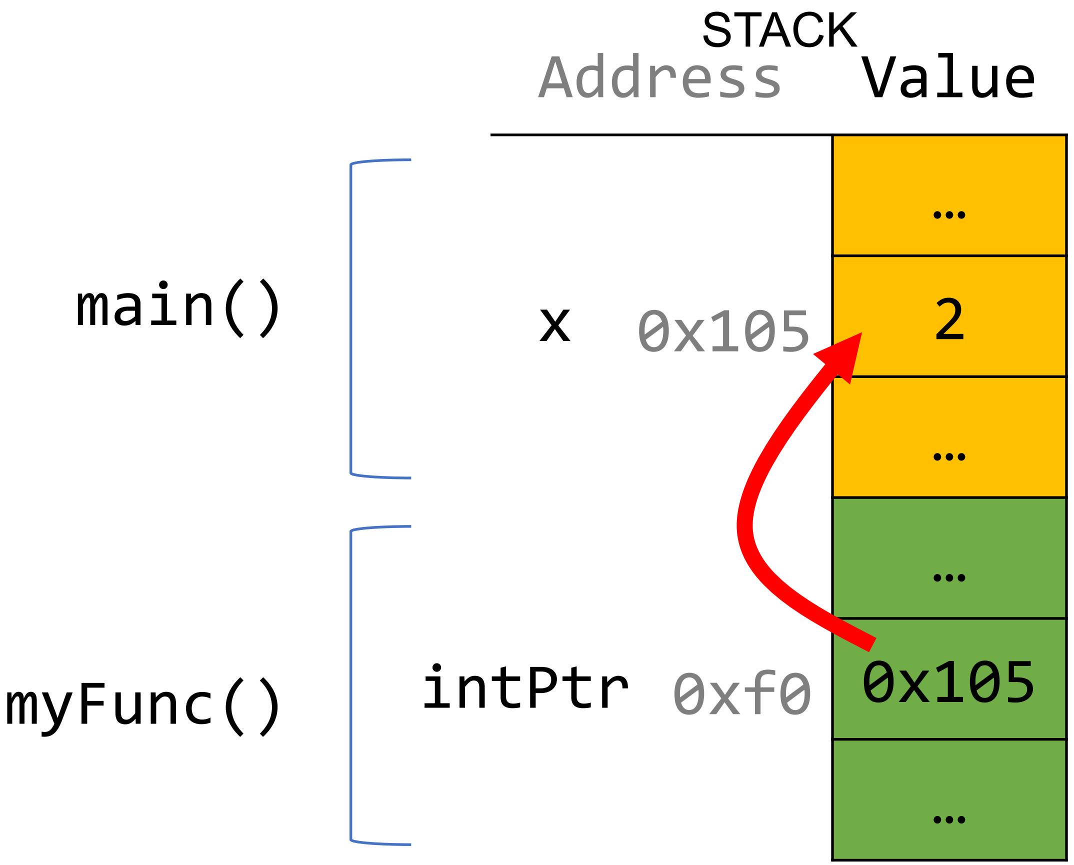
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

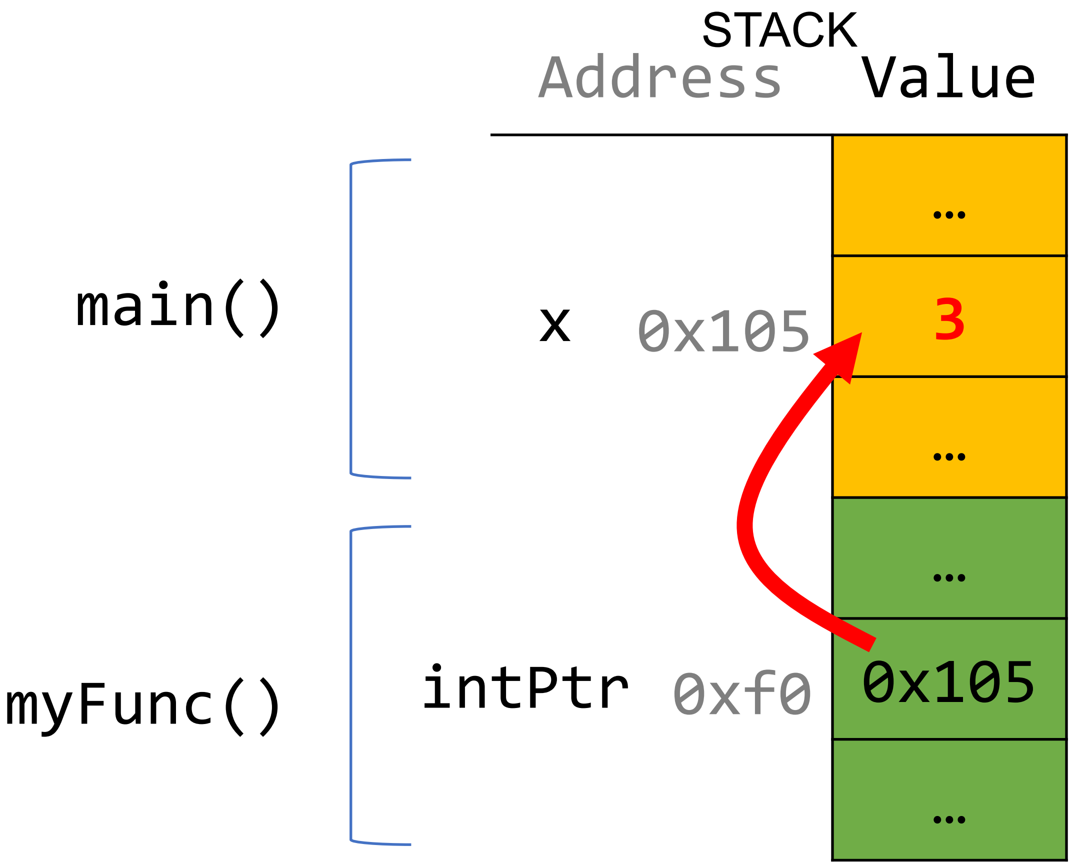
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

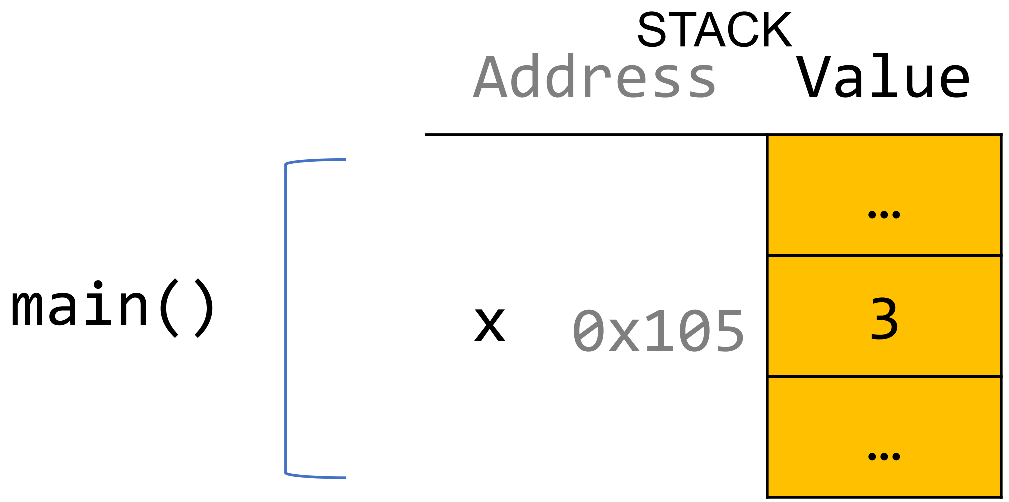
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

A pointer is just a variable that stores a memory address!

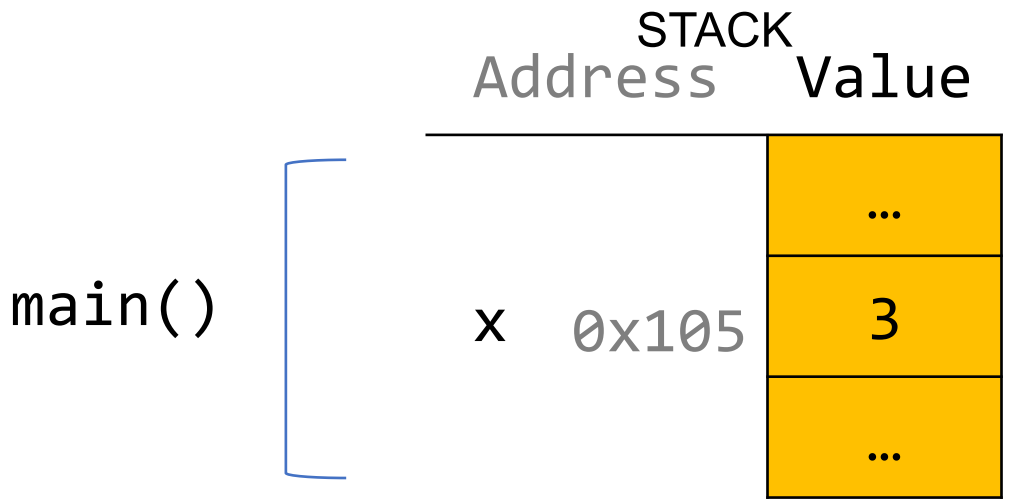
```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

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



Pointers

A pointer is just 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

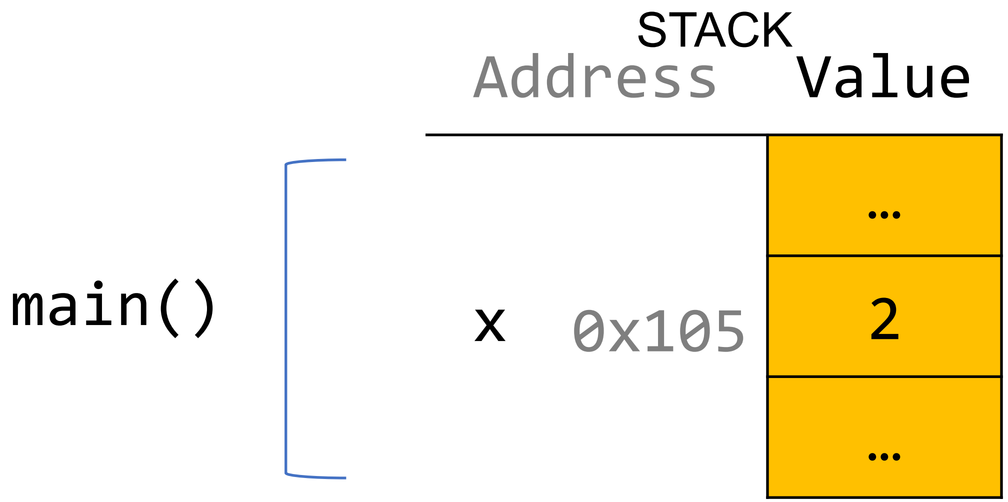
Without pointers, we would make copies.


```
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.

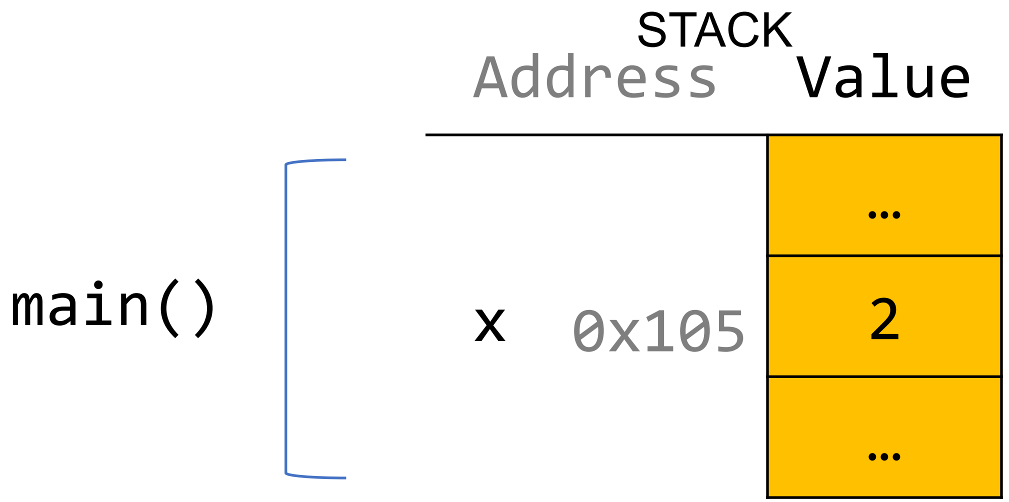
```
void myFunc(int val) {
     val = 3;
}

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



Pointers

Without pointers, we would make copies.

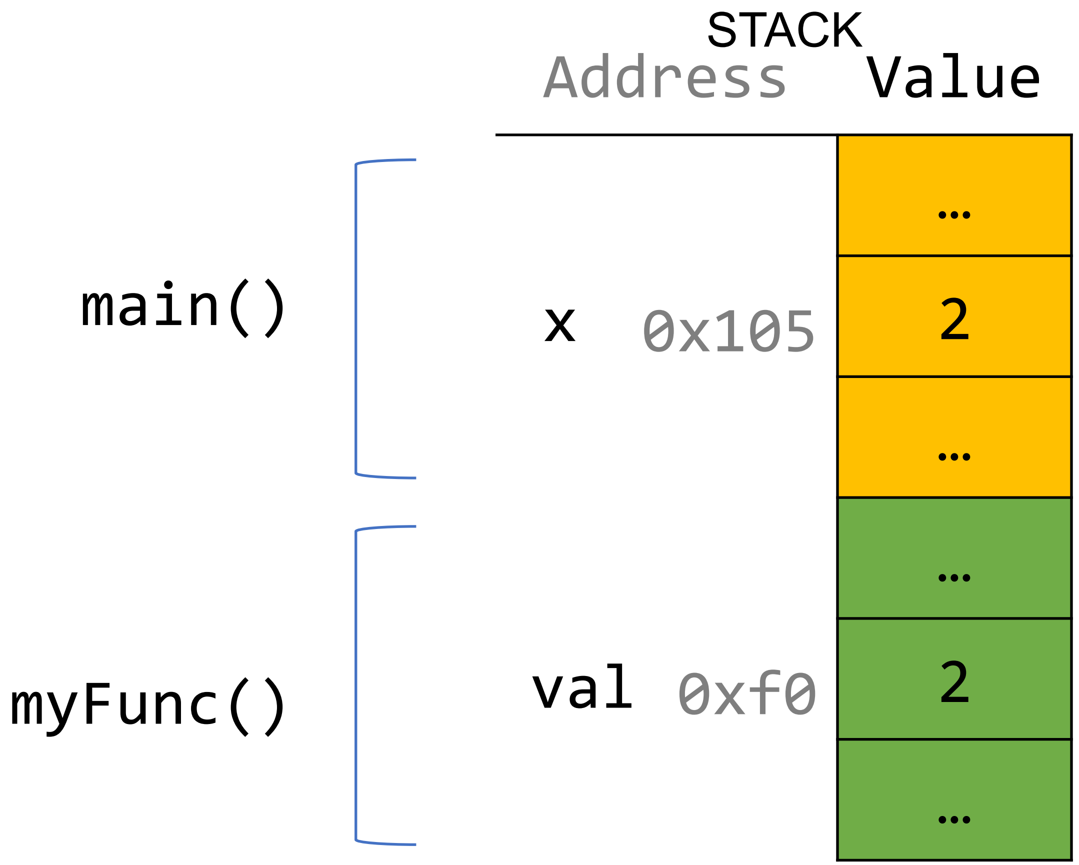
```
void myFunc(int val) {
     val = 3;
}

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



Pointers

Without pointers, we would make copies.

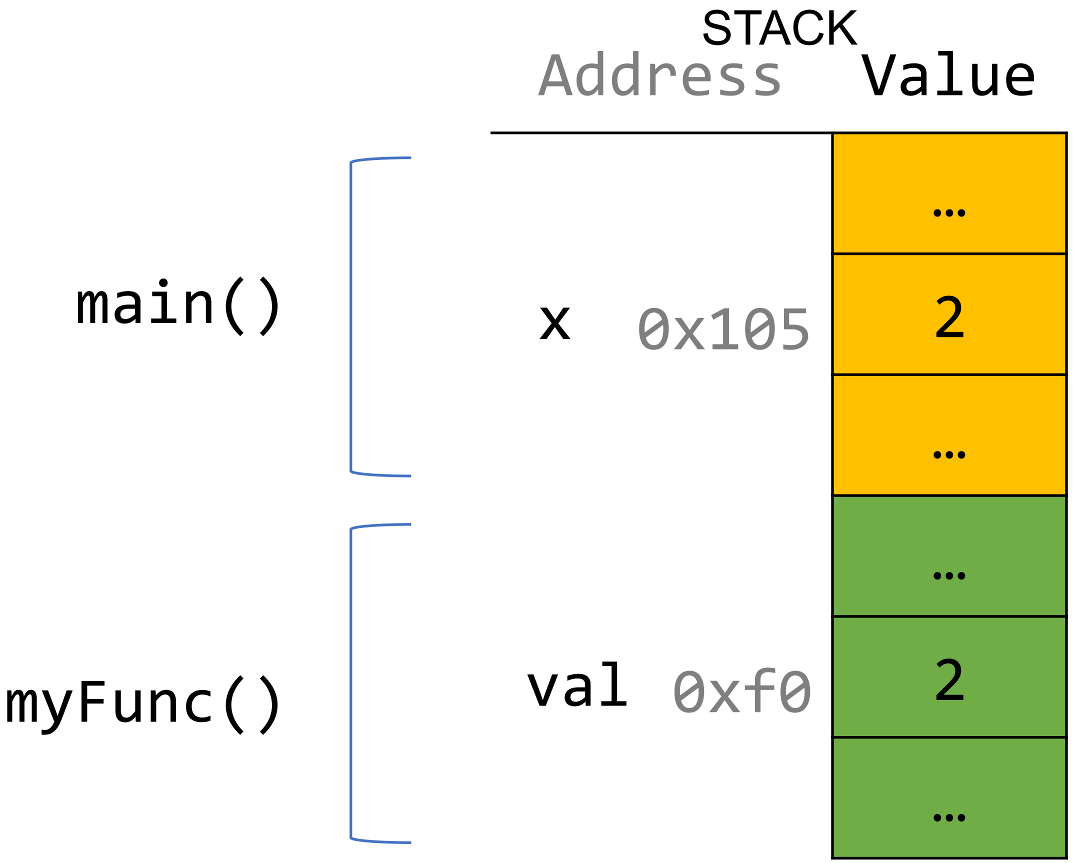
```
void myFunc(int val) {
    val = 3;
}

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



Pointers

Without pointers, we would make copies.

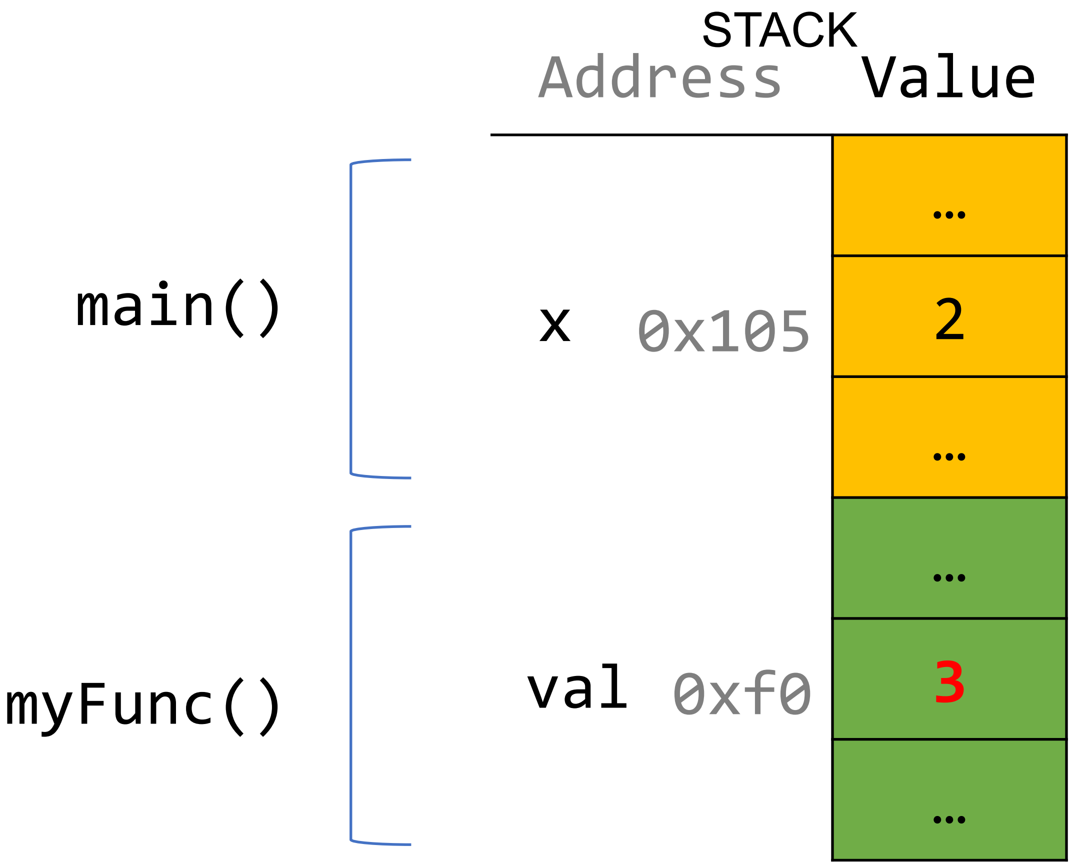
```
void myFunc(int val) {
	val = 3;
}

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



Pointers

Without pointers, we would make copies.

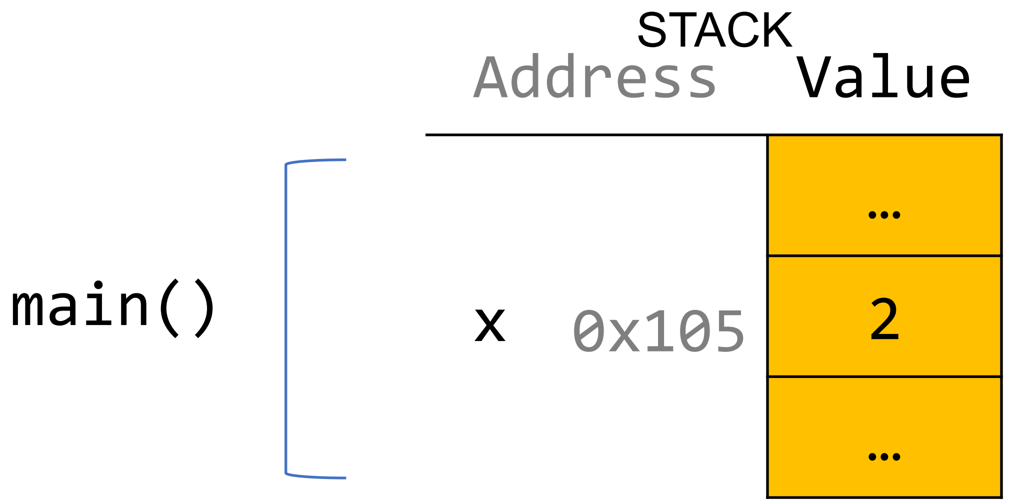
```
void myFunc(int val) {
	val = 3;
}

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



Pointers

Without pointers, we would make copies.

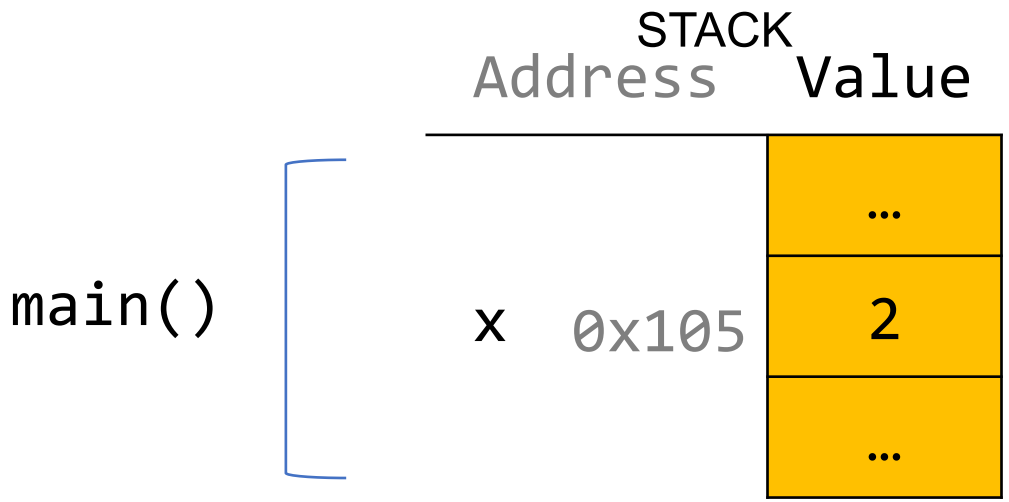
```
void myFunc(int val) {
      val = 3;
}

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



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!
     ...
}
```



Plan For Today

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

Announcements

- Assignment 1 due Monday 1/21 11:59PM PST
  - Grace period until Wed. 1/23 11:59PM PST
- Lab 2: C strings practice
- Assignment 2 released at Assignment 1 due date
  - Due Mon. 1/28 11:59PM PST, grace period until Wed. 1/30 11:59PM PST
  - Programs using C strings
  - Style guide published on course website

Plan For Today

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

Character Arrays

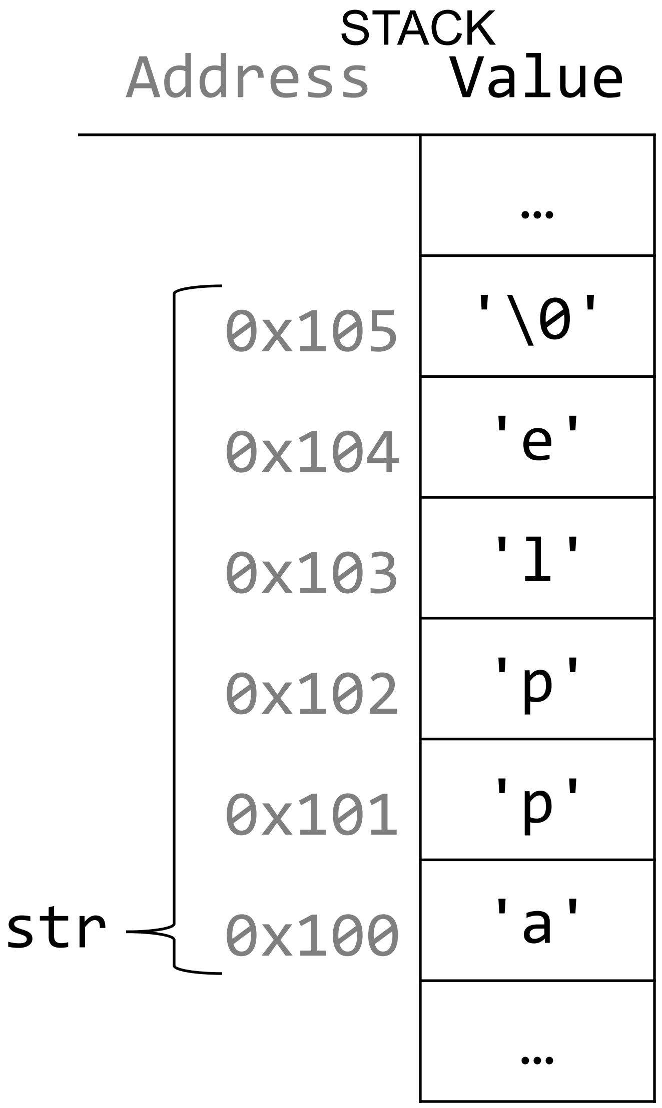
When you declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array.

```
char str[6] = "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
```

*(so then why do we need **strlen**? We'll see soon!)*



Character Arrays

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

```
char str[6] = "apple";
char str2[8] = "apple 2";
str = str2;   // not allowed!
```

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

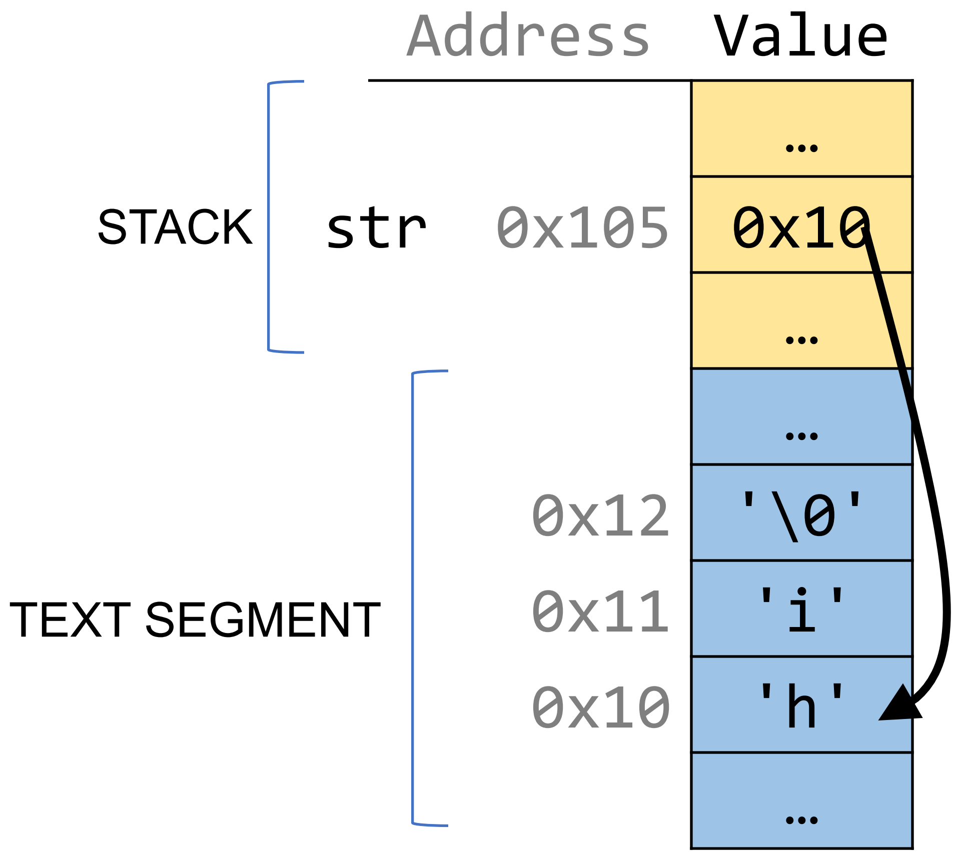
char *

When you declare a char pointer equal to a string literal, the string literal is *not* stored on the stack. Instead, it's stored in a special area of memory called the "Text segment". You *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 text segment*. Since this variable is just a pointer, **sizeof** returns 8, no matter the total size of the string!

```
int stringBytes = sizeof(str);    // 8
```



char *

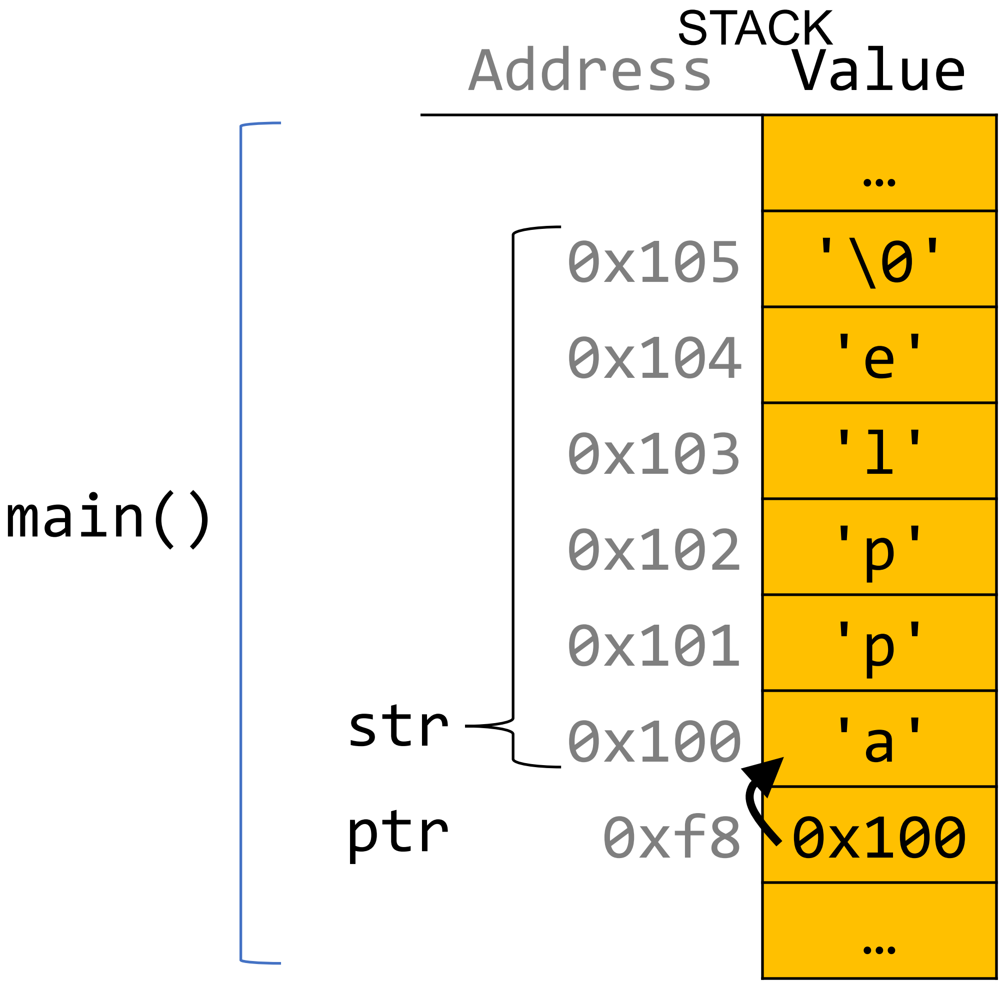
A **char *** variable refers to a single character. You can reassign an existing **char *** pointer to be equal to another **char *** pointer.

```
char *str = "apple";          // e.g. 0xff5
char *str2 = "apple 2";       // e.g. 0xfe2
str = str2;    // ok!  Both store address 0xfe2
```

Arrays and Pointers

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

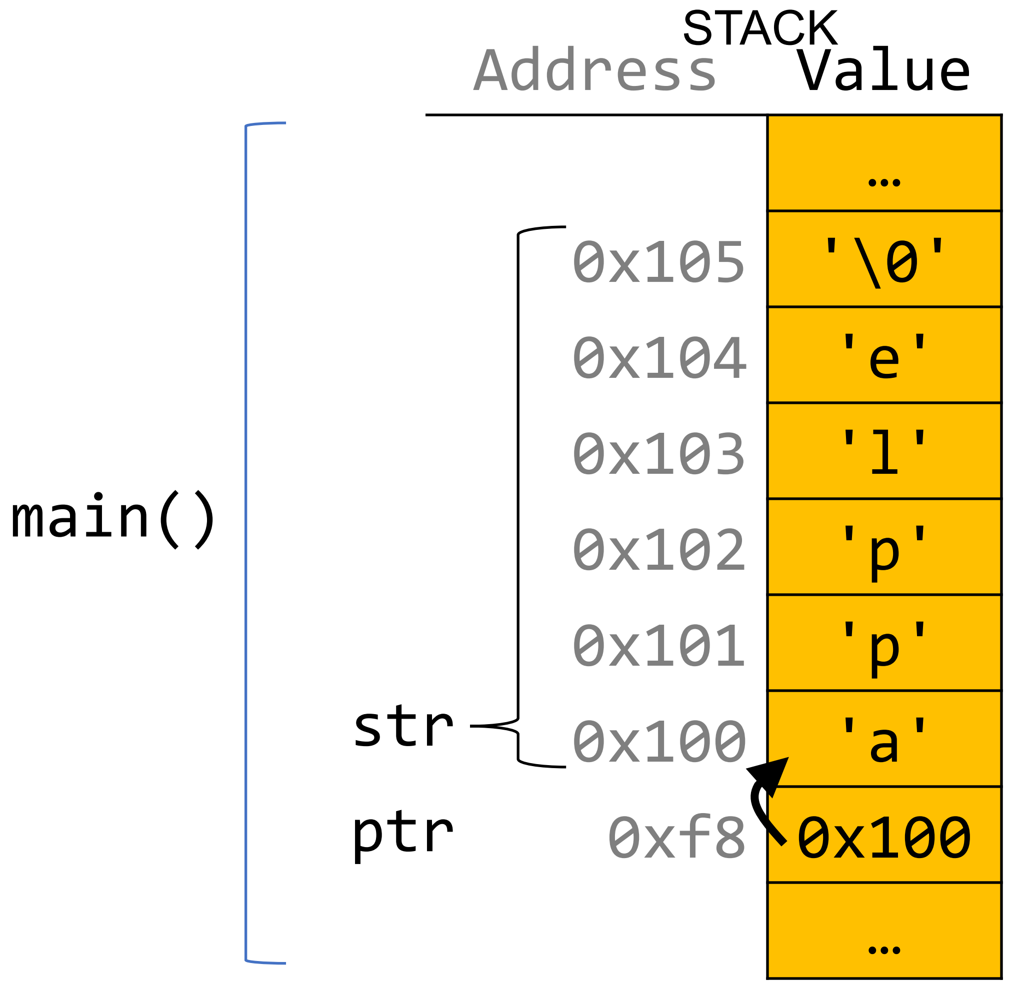
```
int main(int argc, char *argv[]) {
     char str[6] = "apple";
     char *ptr = str;
     ...
}
```



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[6] = "apple";
     char *ptr = str;

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

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



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. 0xff1
char *str2 = str + 1;    // e.g. 0xff2
char *str3 = str + 3;    // e.g. 0xff4

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

TEXT SEGMENT

Address	Value
	…
0xff6	'\0'
0xff5	'e'
0xff4	'l'
0xff3	'p'
0xff2	'p'
0xff1	'a'
	…

Pointer Arithmetic

Pointer arithmetic does *not* add bytes. Instead, it adds the *size of the type it points to*.

```
// nums points to an int array
int *nums = …             // e.g. 0xff1
int *nums2 = nums + 1;    // e.g. 0xff5
int *nums3 = nums + 3;    // e.g. 0xffd

printf("%d", *nums);      // 52
printf("%d", *nums2);     // 23
printf("%d", *nums3);     // 34
```

STACK

Address	Value
	…
0x1005	1
0x1001	16
0xffd	34
0xff9	12
0xff5	23
0xff1	52
	…

char *

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

```
char *str = "apple";    // e.g. 0xff1

// both of these add two places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff3.
char thirdLetter = str[2];        // 'p'
char thirdLetter = *(str + 2);    // 'p'
```

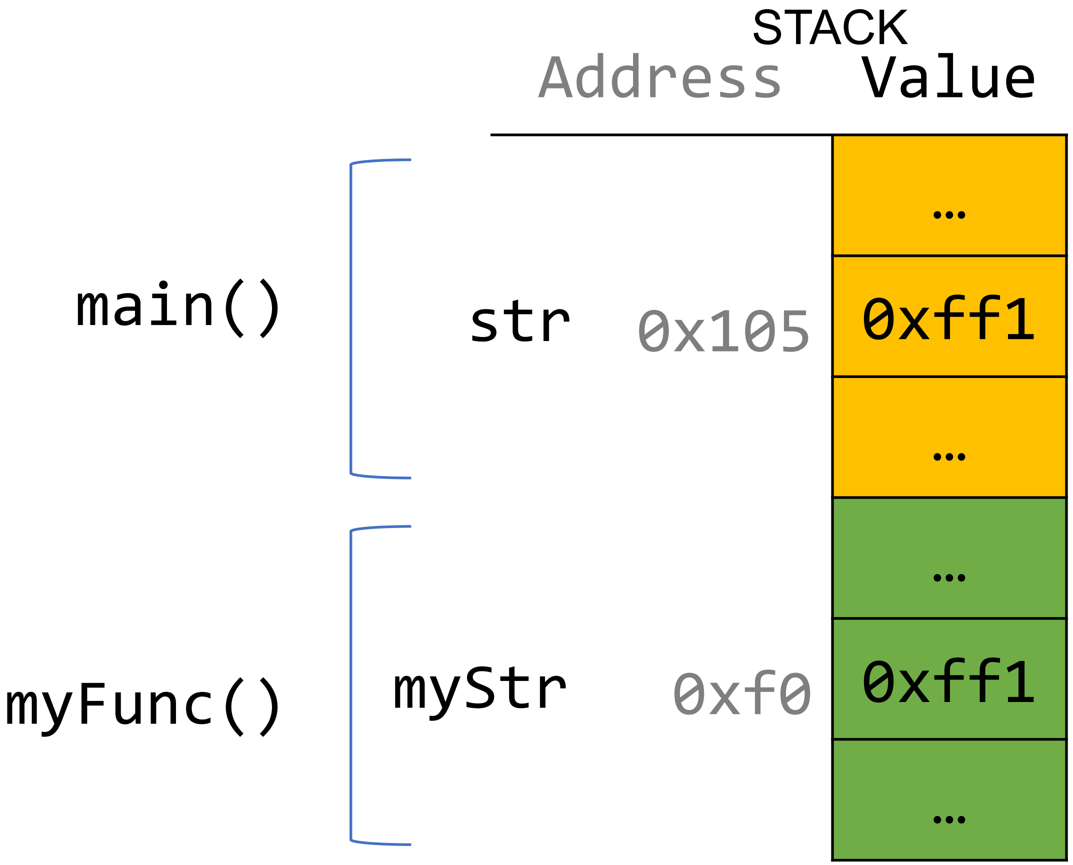
TEXT SEGMENT

Address	Value
	…
0xff6	'\0'
0xff5	'e'
0xff4	'l'
0xff3	'p'
0xff2	'p'
0xff1	'a'
	…

Strings as Parameters

When you 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.

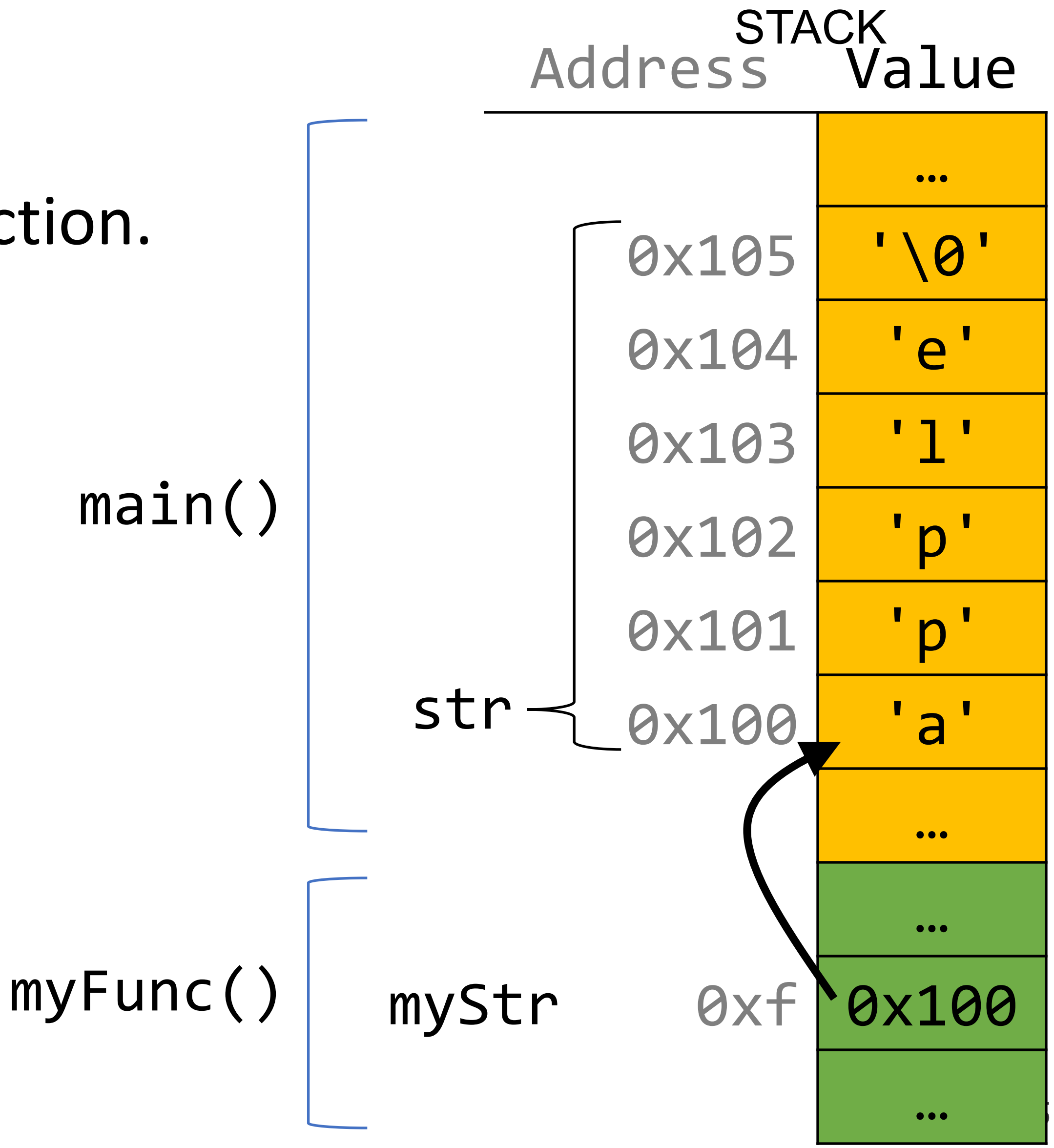
```
void myFunc(char *myStr) {
      …
}

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



Strings as Parameters

When you 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.

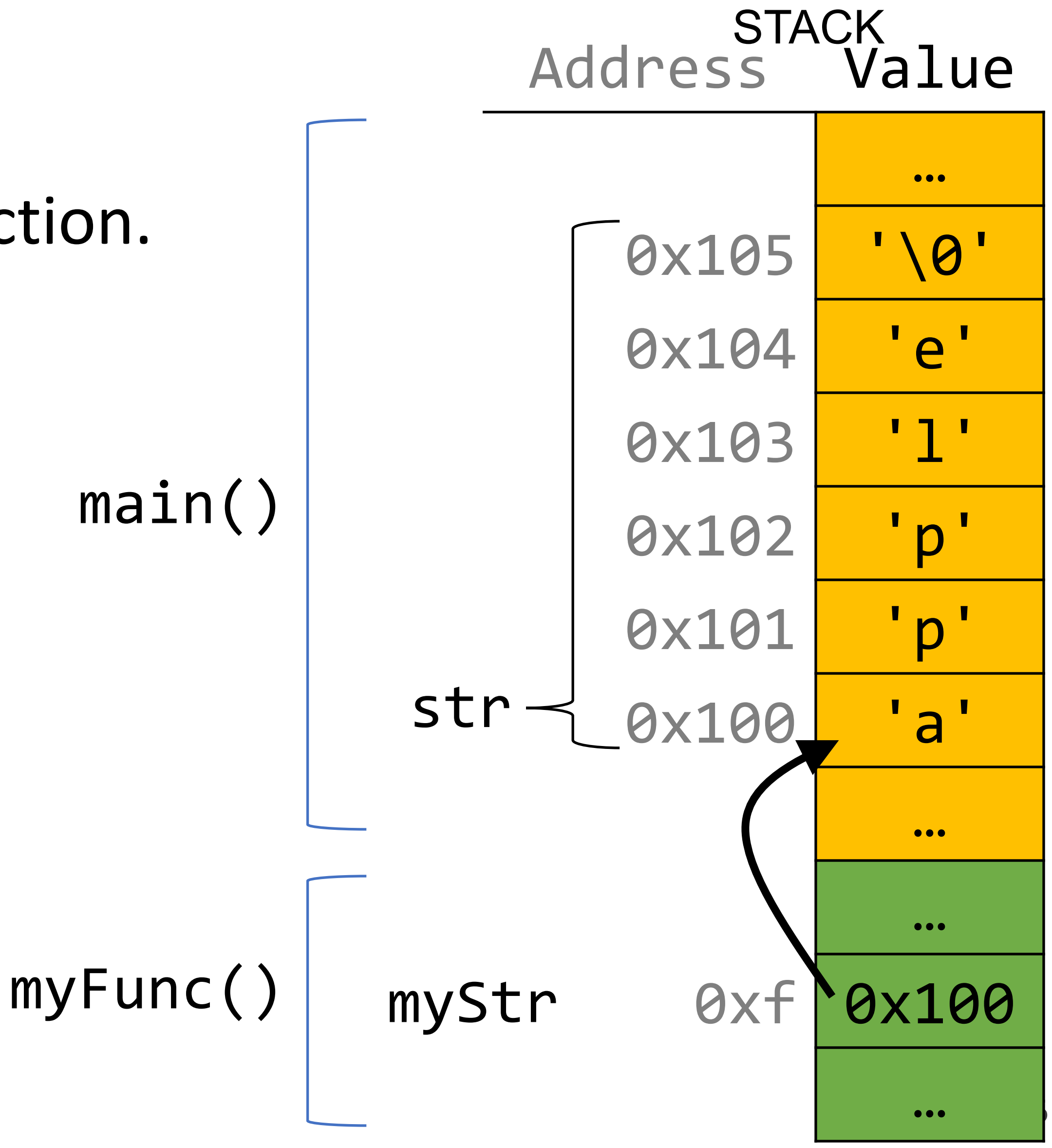
```
void myFunc(char *myStr) {
    …
}

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



Strings as Parameters

When you 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.

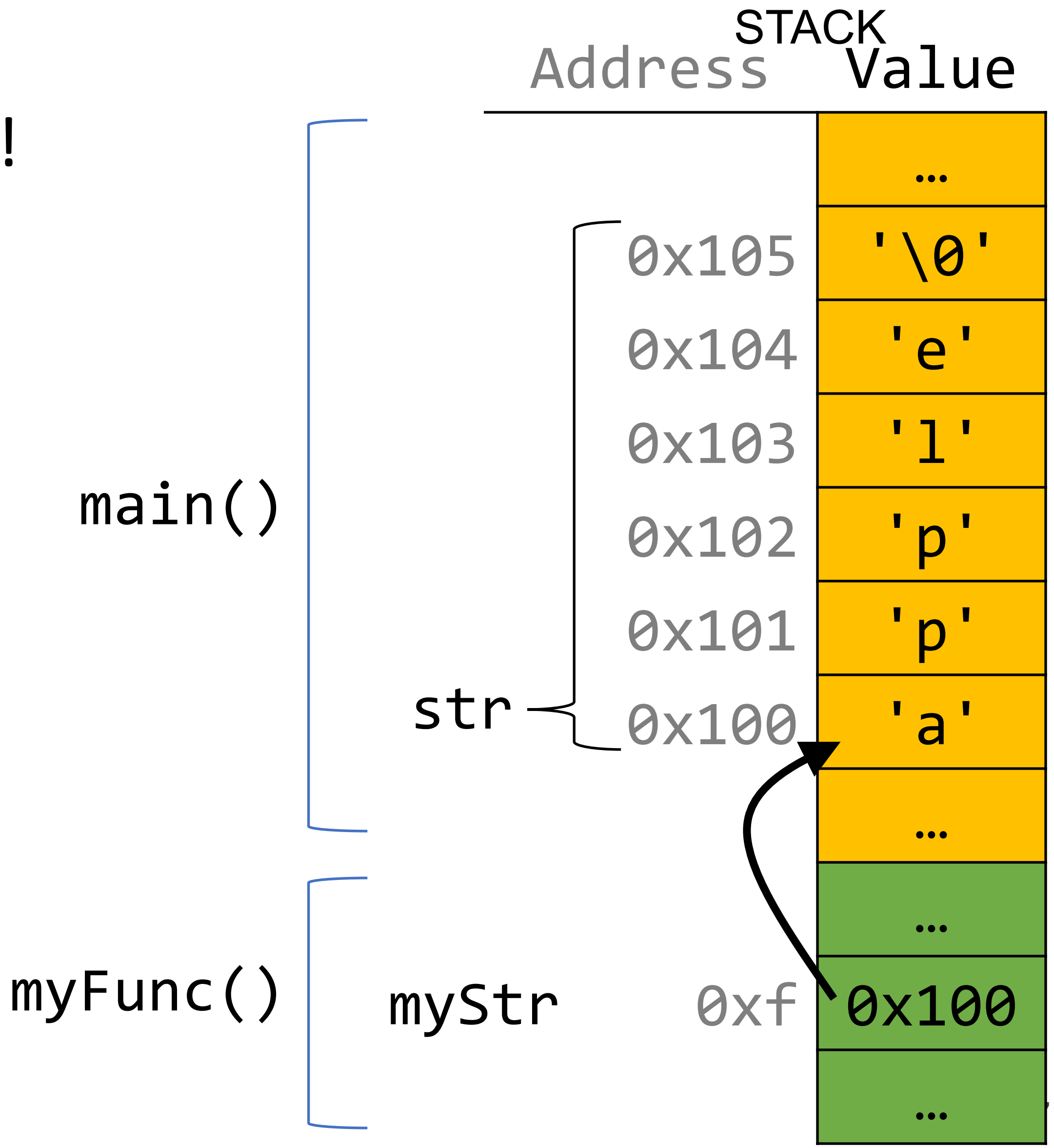
```
void myFunc(char *myStr) {
     …
}

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



Strings as Parameters

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

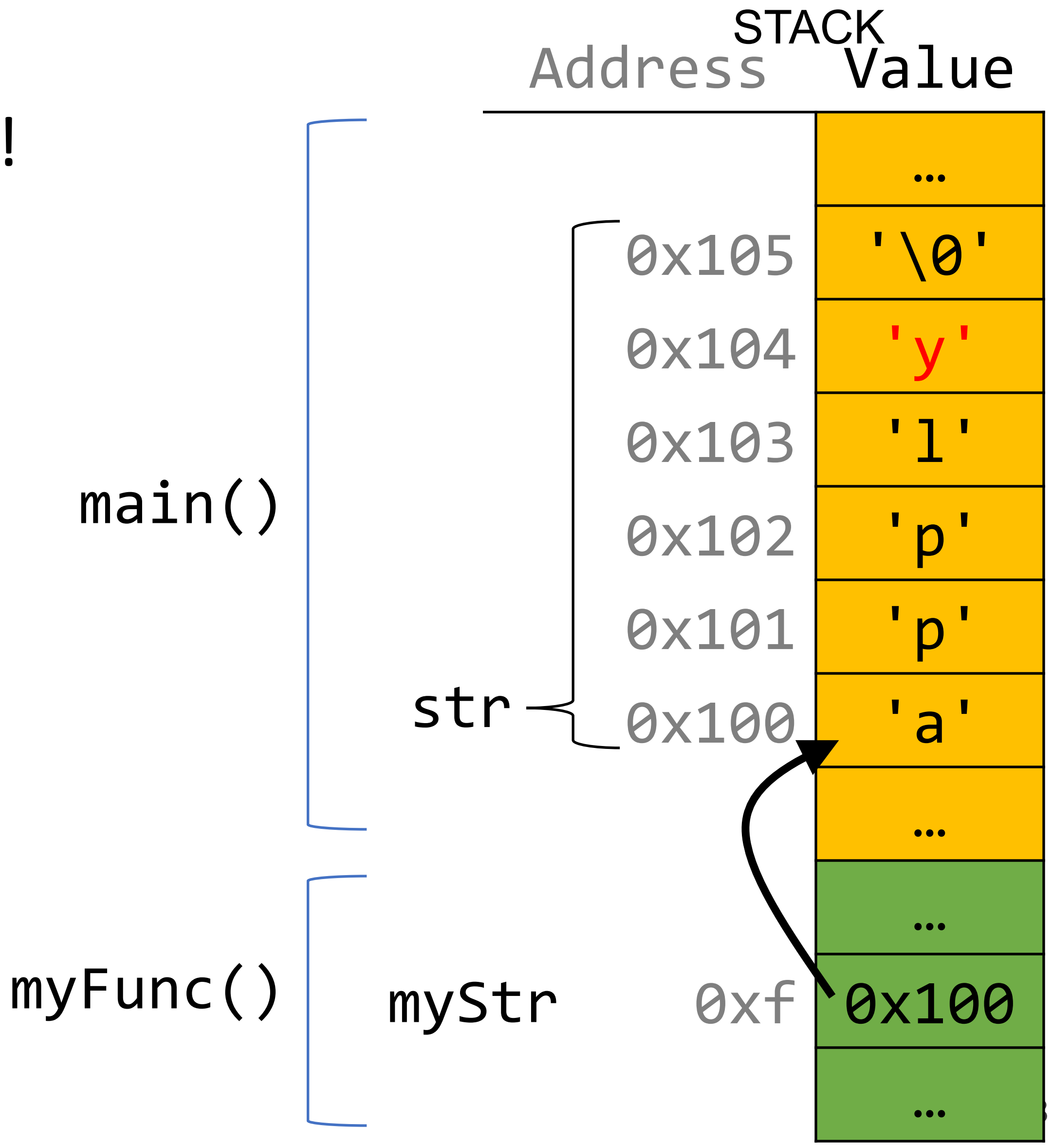
```
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

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



Strings as Parameters

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

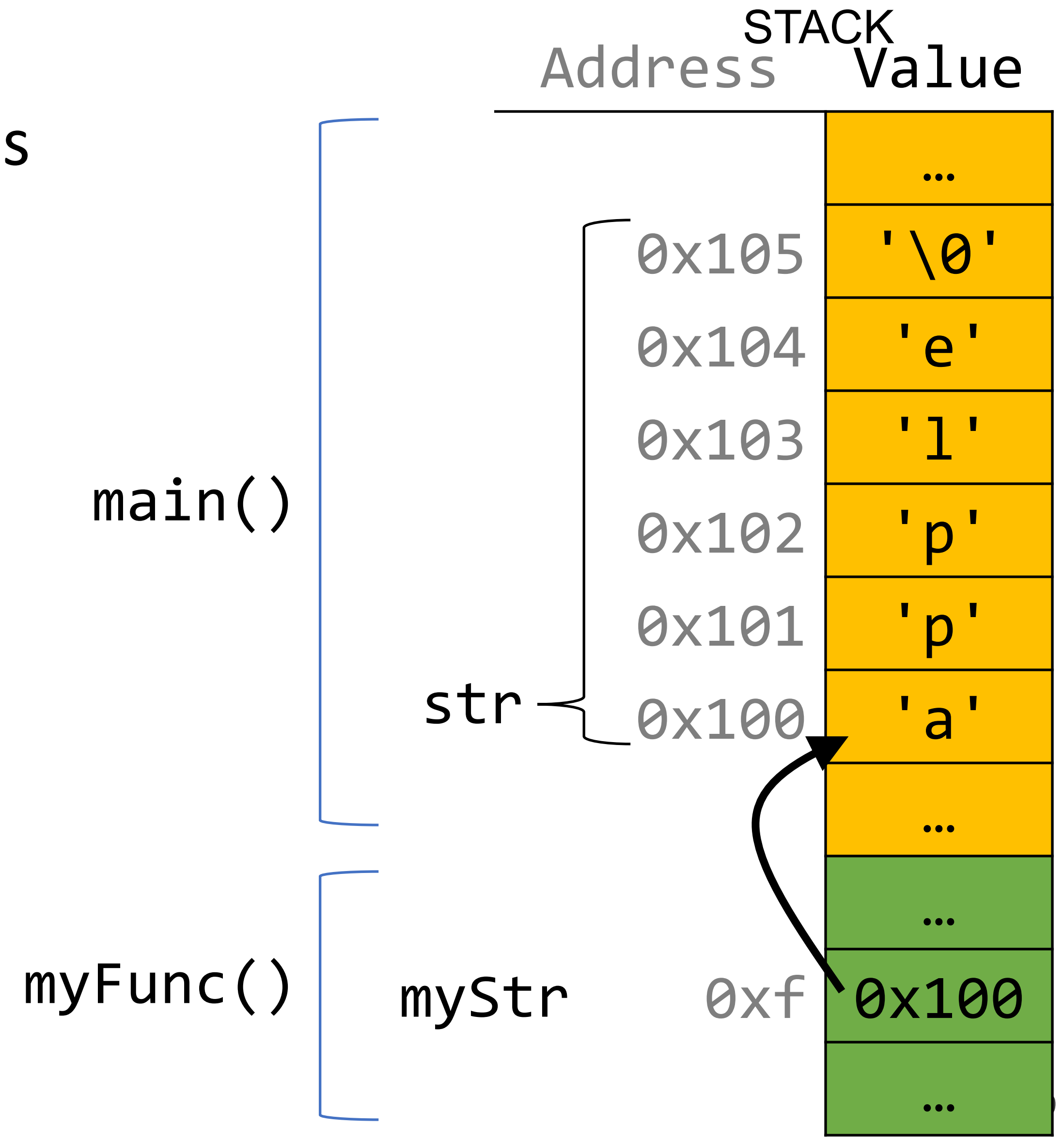
```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

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



Strings as Parameters

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

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

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



Strings and Memory

These memory behaviors explain why strings behave the way they do:

1. We can modify a string created as a **char[]** because its memory lives in our stack space.
2. We cannot modify a string created as a **char*** because its memory does not live in our stack space; it lives in the text segment.
3. We can set a **char*** equal to another value, because it is just a pointer.
4. 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.
5. If we change characters in a string passed to a function, these changes will persist outside of the function.
6. When we pass a char array as a parameter, we can no longer use **sizeof** to get its full size.

Demo: Strings and Memory

Plan For Today

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

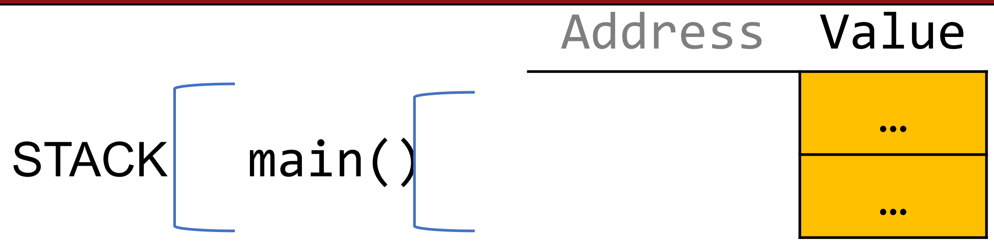
Pointers to Strings

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to.

- Ex. Write a function **skipCSPrefix** that takes in a **char *** representing a class name, and modifies it to advance past the "CS" prefix, if any, in the string.

```
char *myStr = "CS41";
skipCSPrefix(&myStr);
printf("%s\n", myStr);  // 41
```

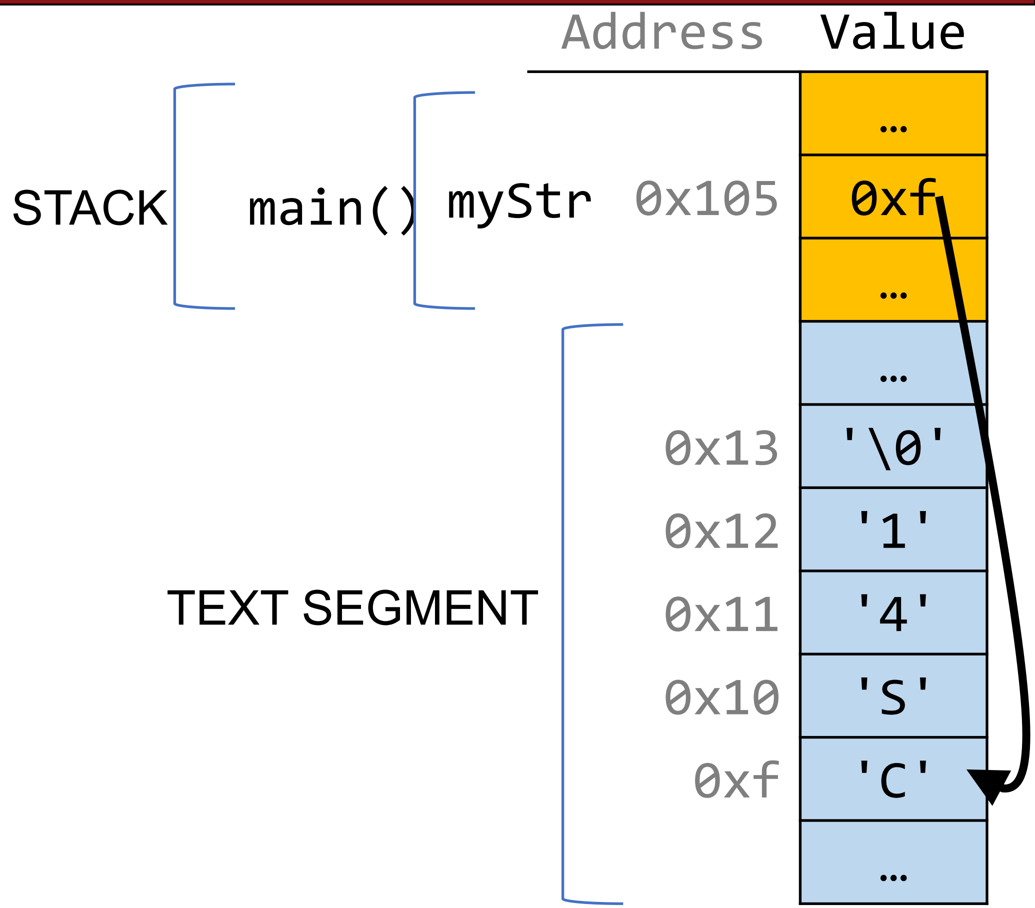
Pointers to Strings




```c
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);      // 41
    return 0;
}
```

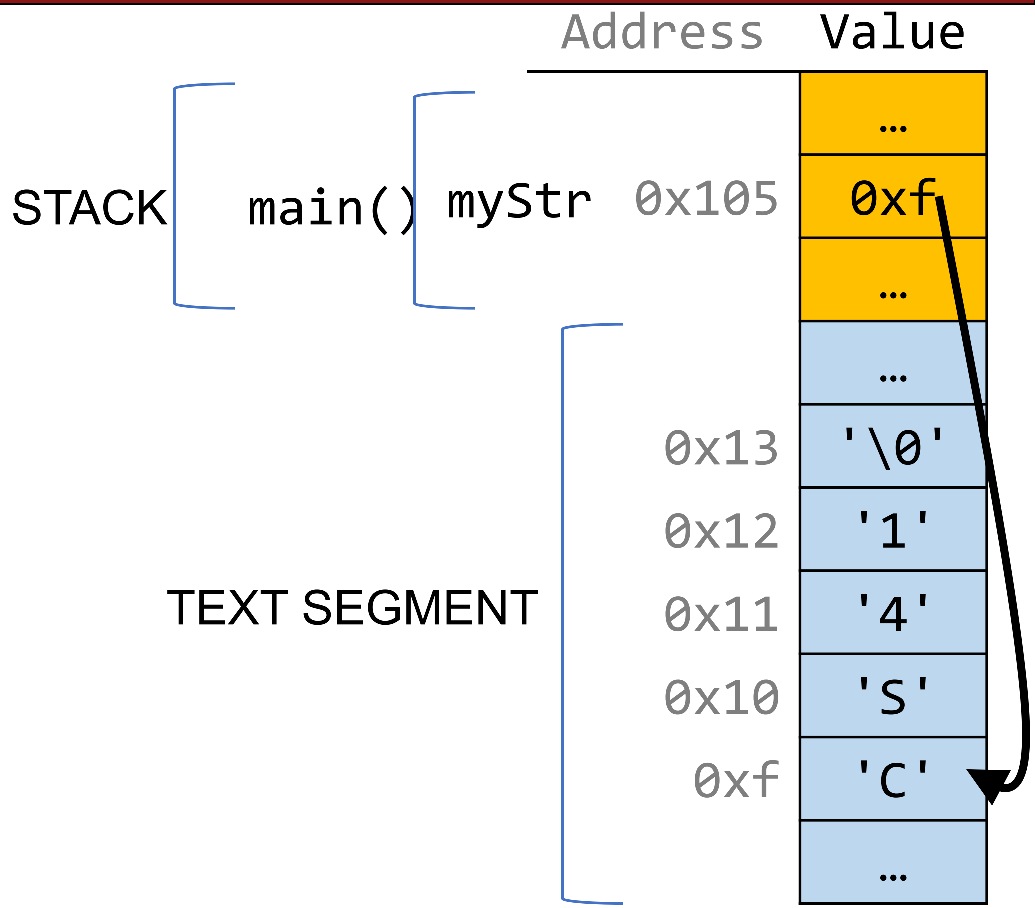
Pointers to Strings

```c
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);       // 41
    return 0;
}
```



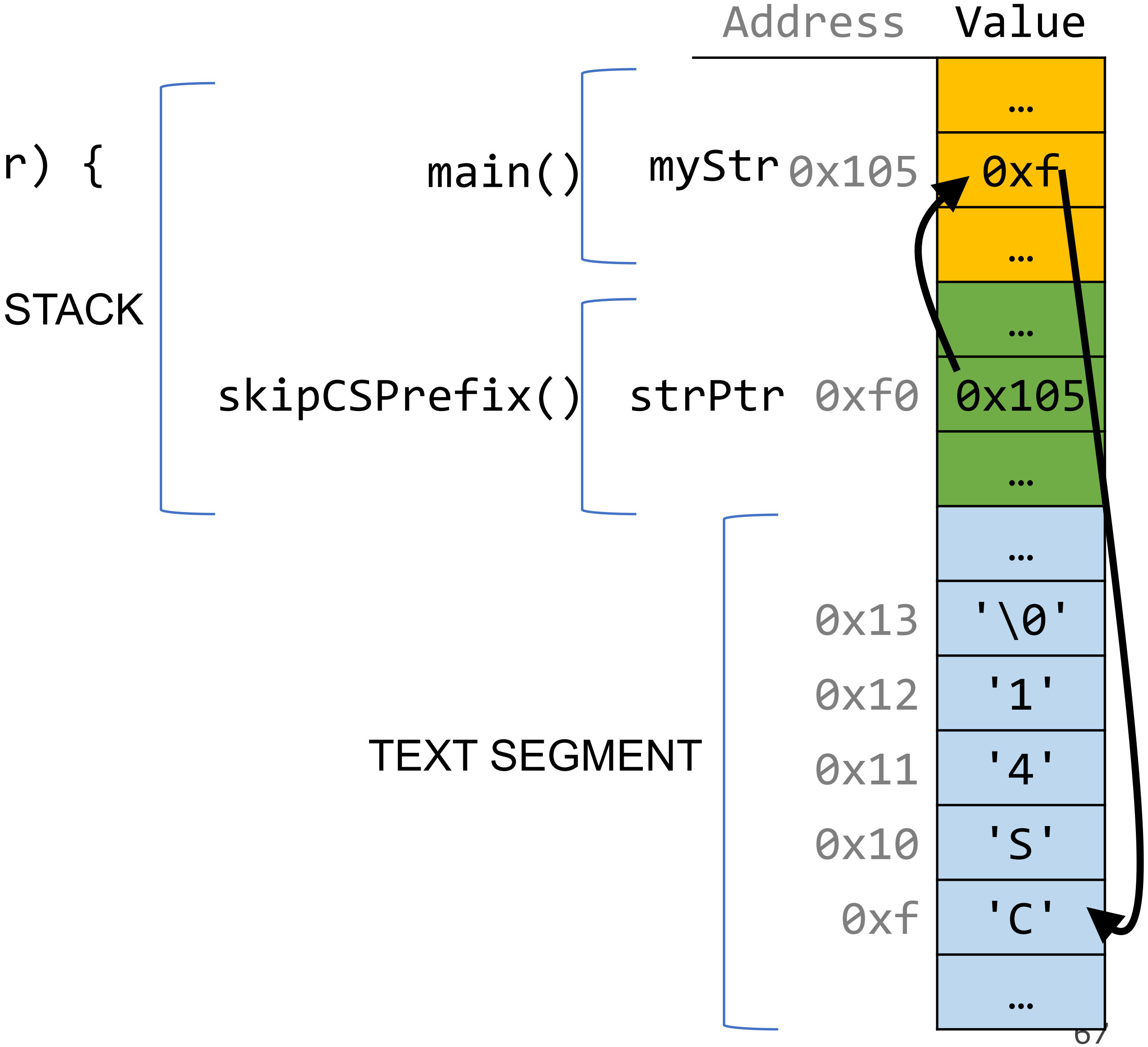
Pointers to Strings

```
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);        // 41
    return 0;
}
```



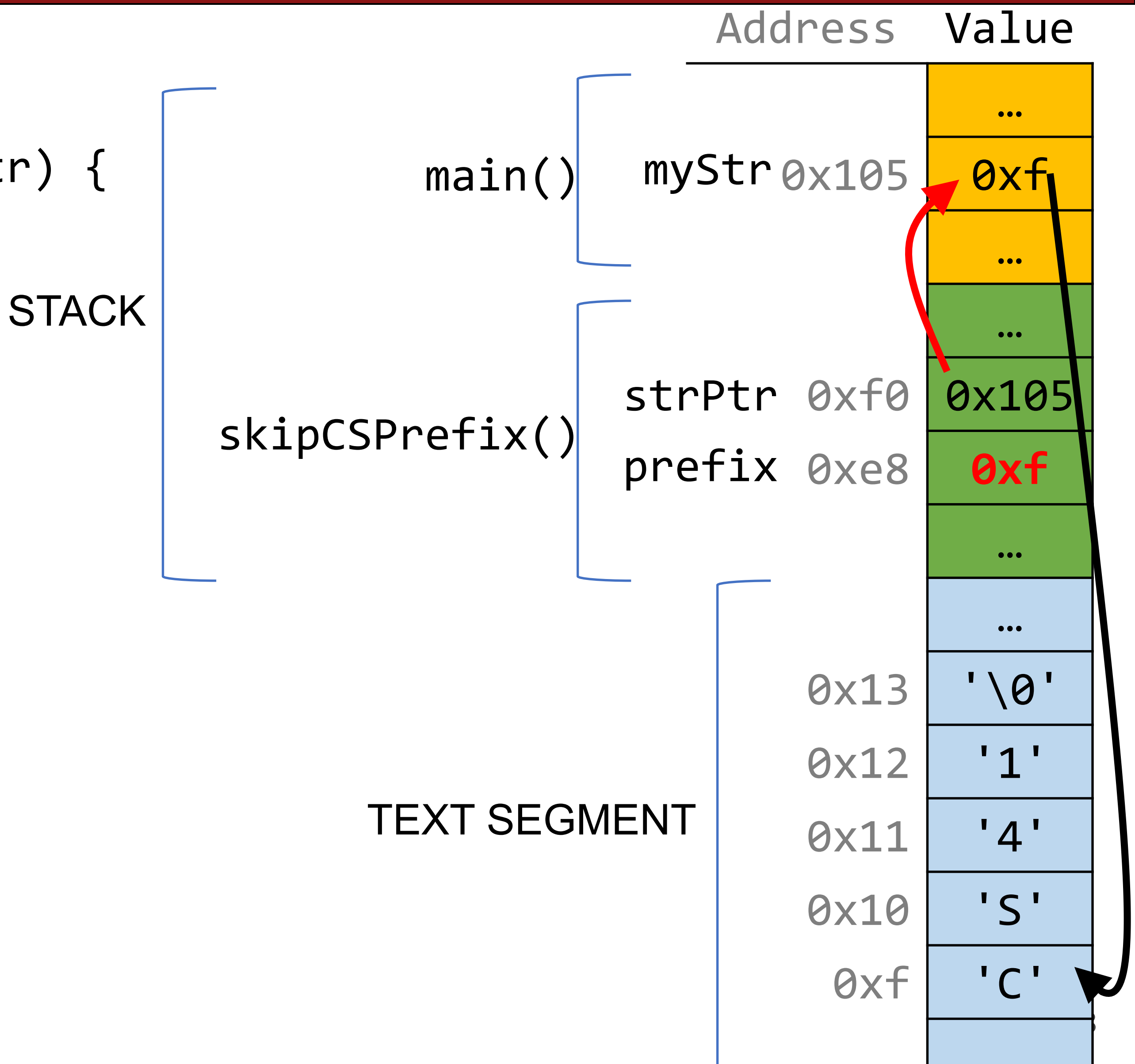
Pointers to Strings

```
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);        // 41
    return 0;
}
```



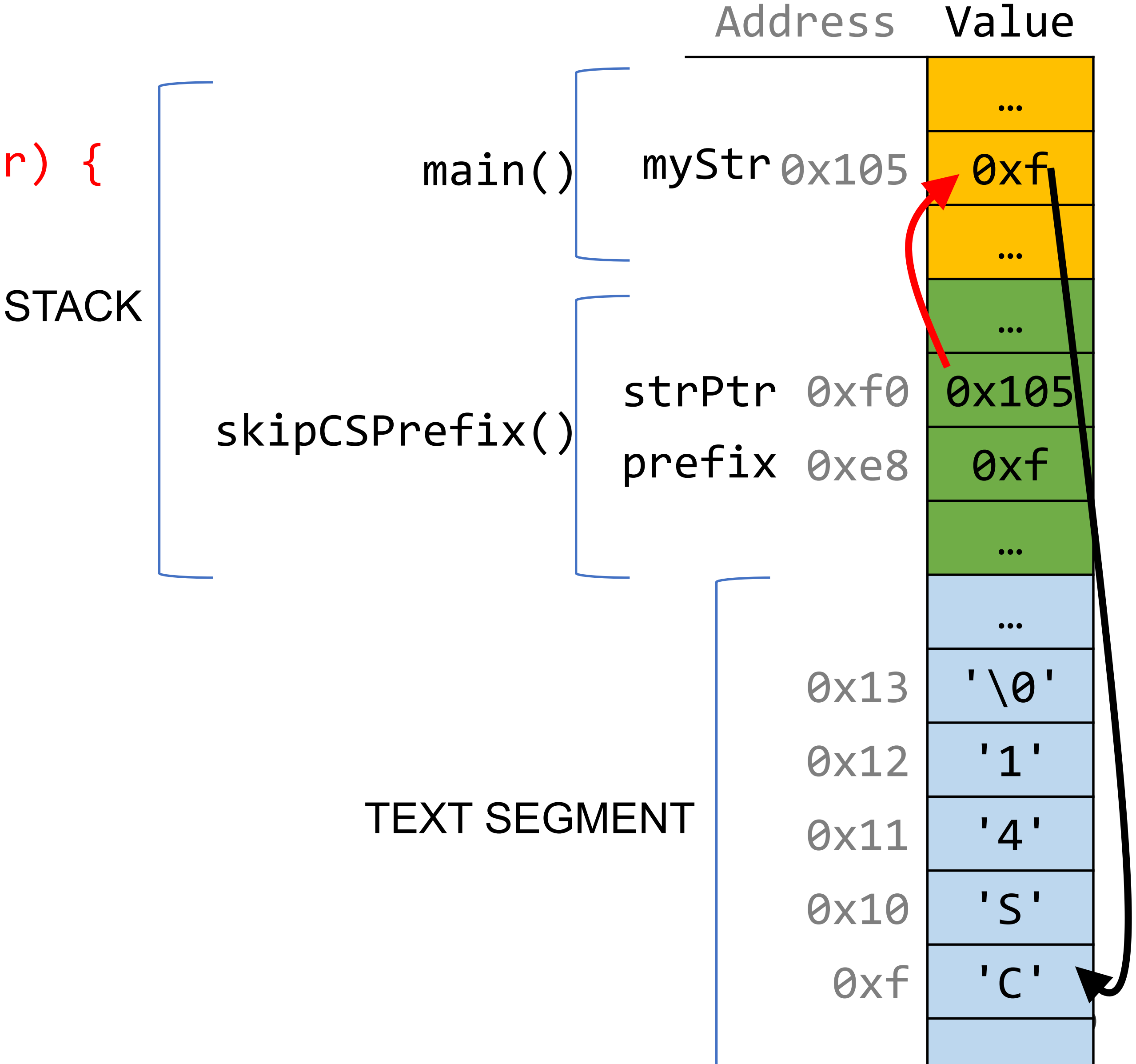
Pointers to Strings

```c
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);        // 41
    return 0;
}
```



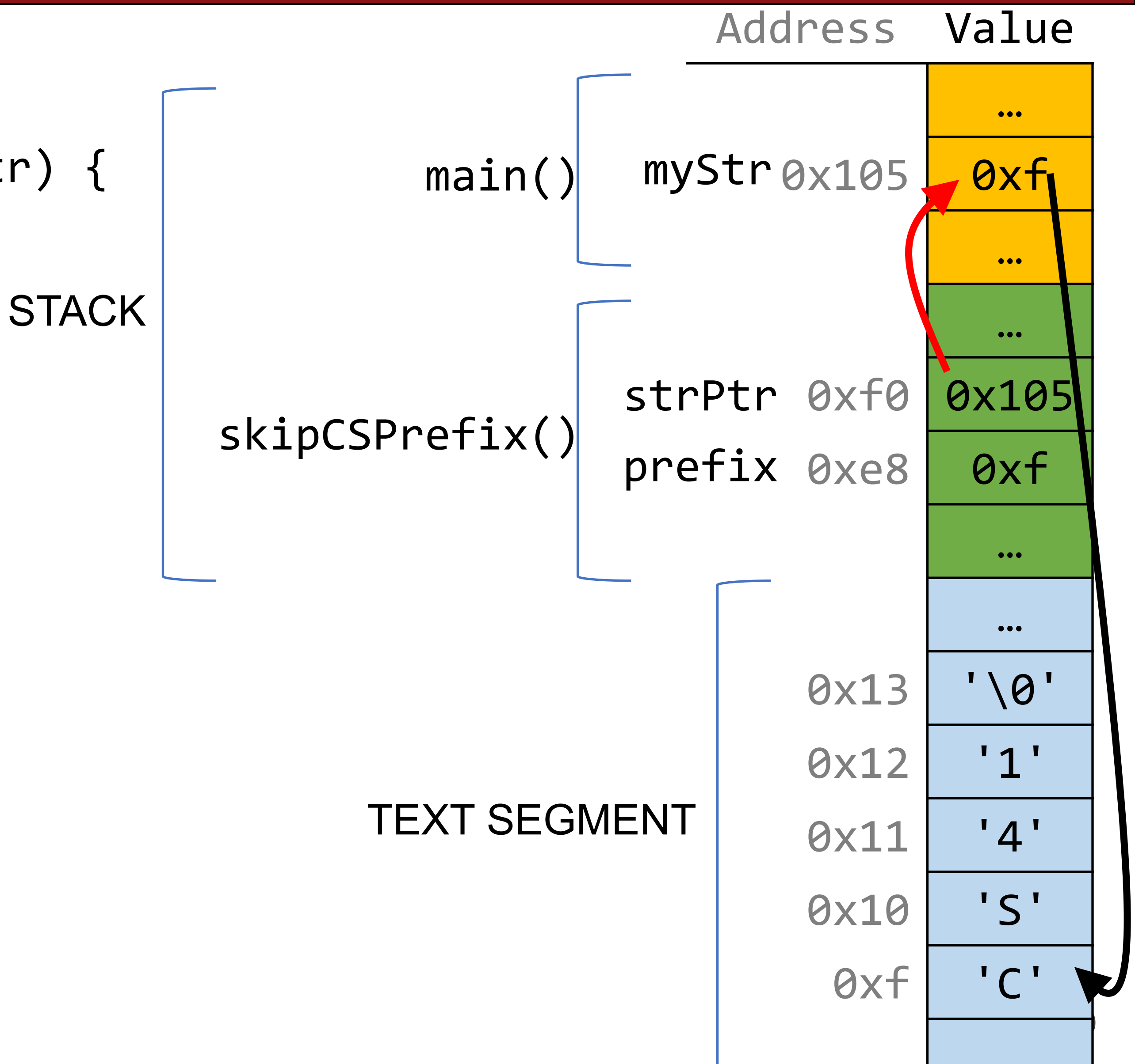
Pointers to Strings

```c
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);        // 41
    return 0;
}
```



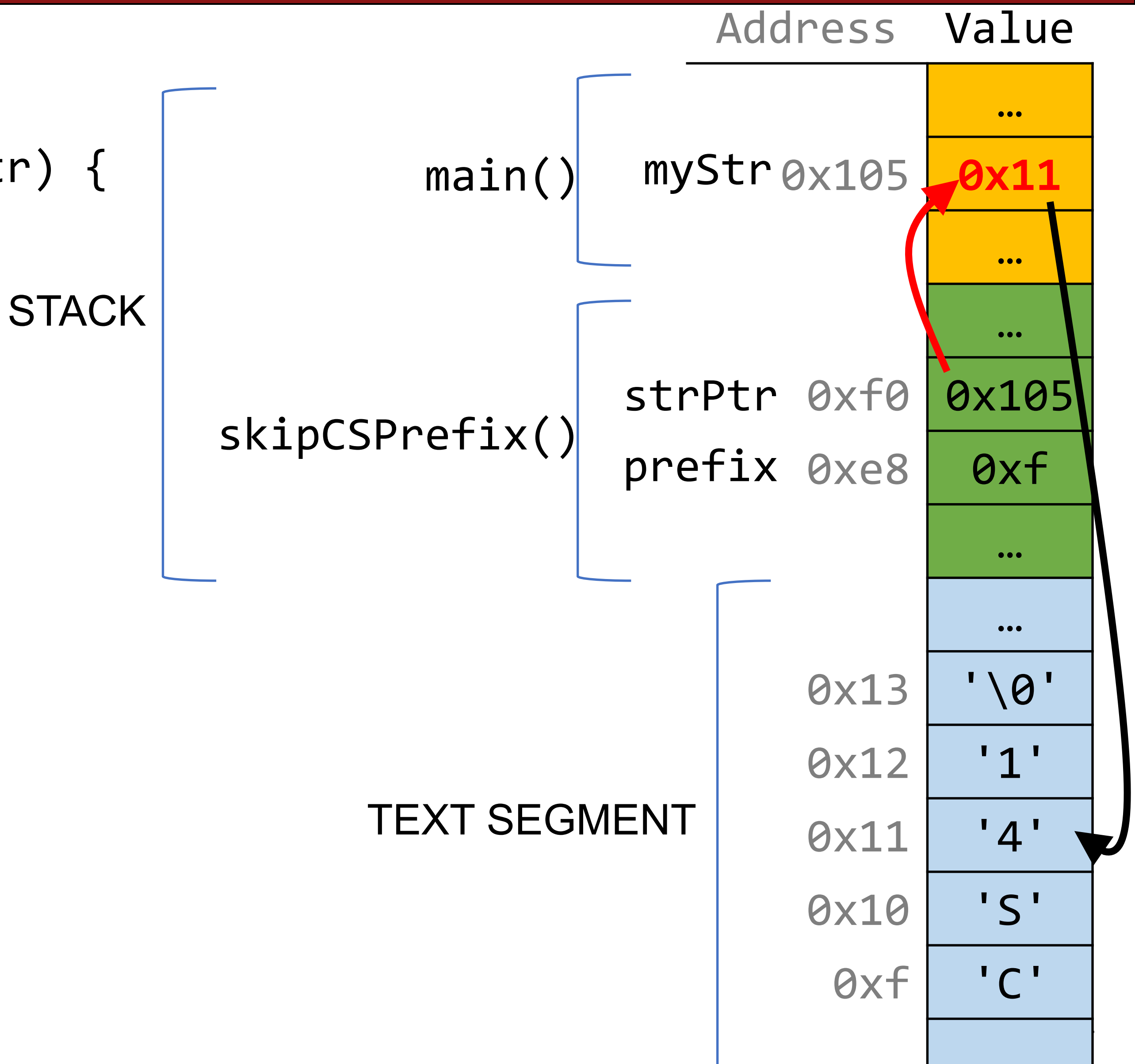
Pointers to Strings

```
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);        // 41
    return 0;
}
```



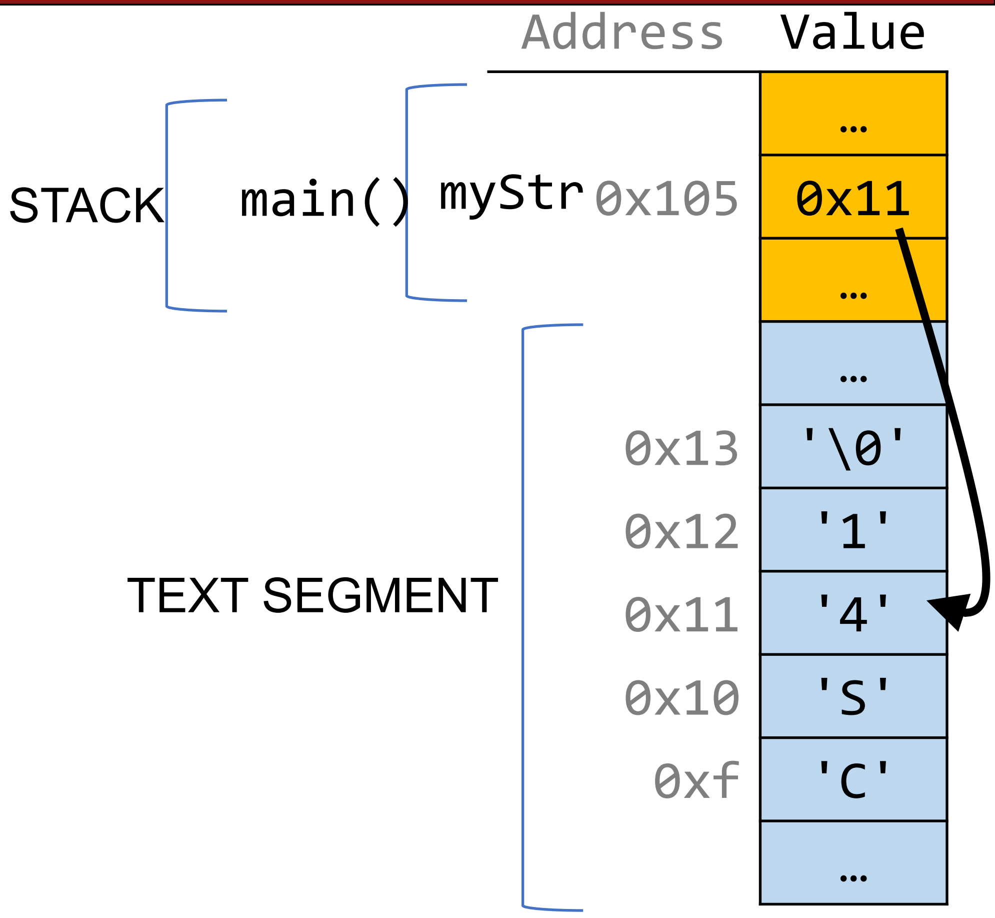
Pointers to Strings

```c
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);        // 41
    return 0;
}
```



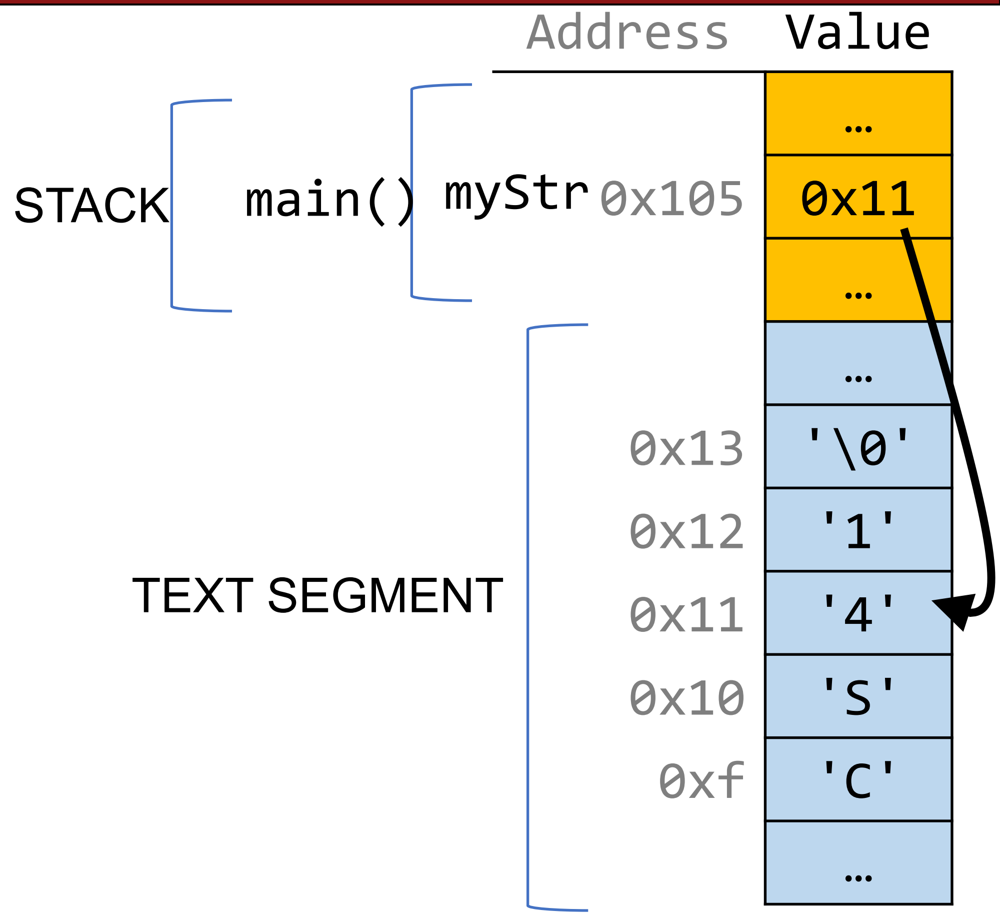
Pointers to Strings

```
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);        // 41
    return 0;
}
```



Pointers to Strings

```
void skipCSPrefix(char **strPtr) {
    char *prefix = strstr(*strPtr, "CS");
    if (prefix != NULL && prefix == *strPtr) {
        *strPtr += strlen("CS");
    }
}

int main(int argc, char *argv[]) {
    char *myStr = "CS41";
    skipCSPrefix(&myStr);
    printf("%s\n", myStr);       // 41
    return 0;
}
```



Recap

- **Recap:** String Operations
- **Demo:** Buffer Overflow and Valgrind
- Arrays of Strings
- **Practice:** Password Verification
- Pointers
- **Announcements**
- Strings in Memory
- Pointers to Strings

Next time: Arrays and Pointers