



# CS107, Lecture 10

## Arrays and Pointers, Take II

Reading: K&R (5.2-5.5) or Essential C section 6

Ed Discussion: <https://edstem.org/us/courses/46162/discussion/3643757>

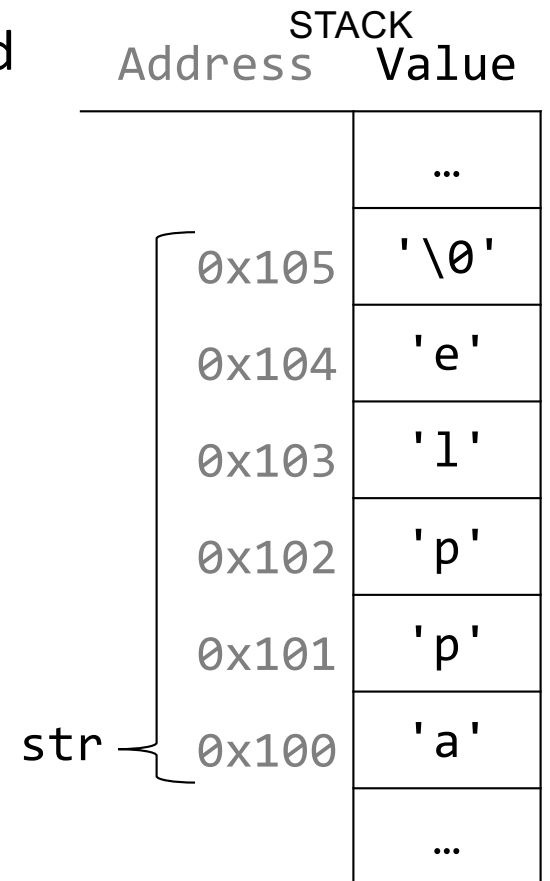
# Arrays

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!

```
size_t 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.

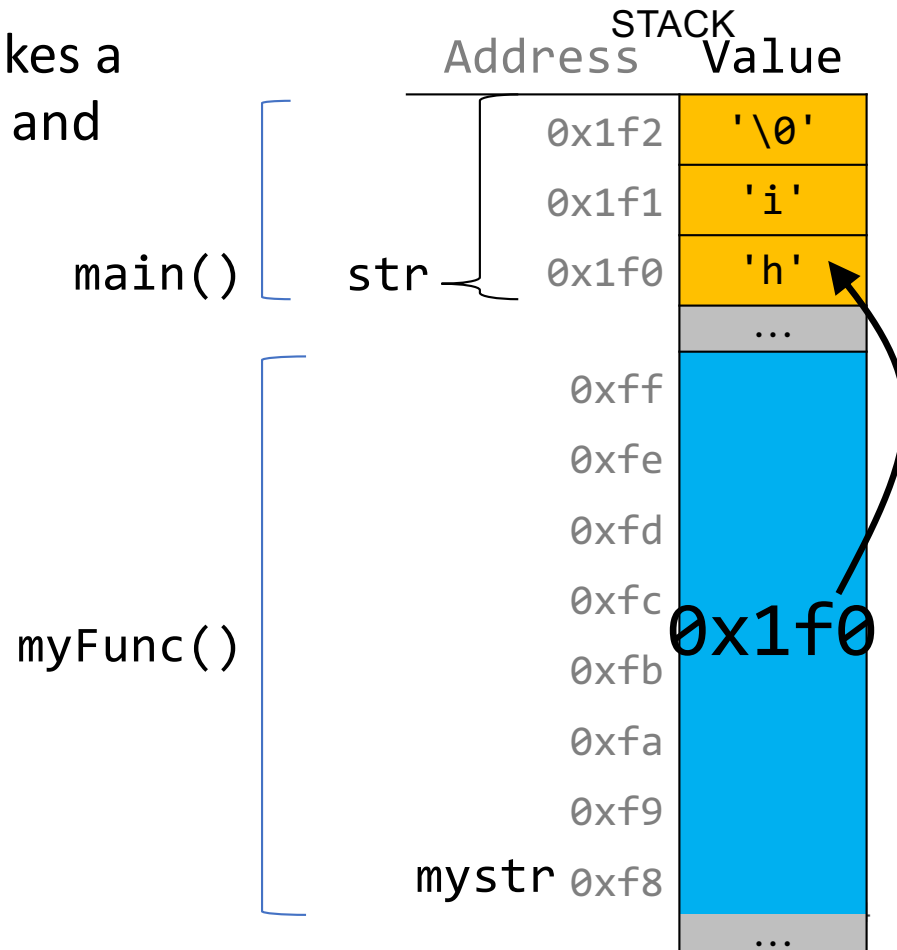
```
int nums[] = {1, 2, 3};  
int nums2[] = {4, 5, 6, 7};  
nums = nums2; // not allowed!
```

An array's size cannot be changed once you create it. You must create another 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);  
    ...  
}
```

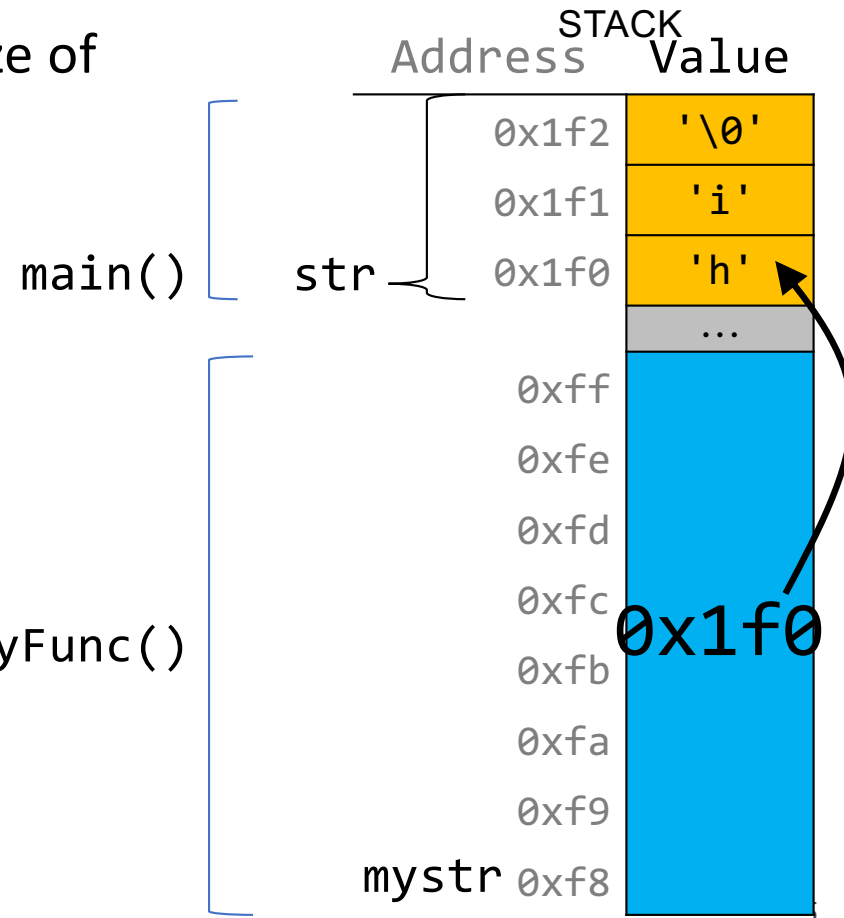



# Arrays as Parameters

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) {  
    size_t size = sizeof(myStr); // 8  
}
```

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



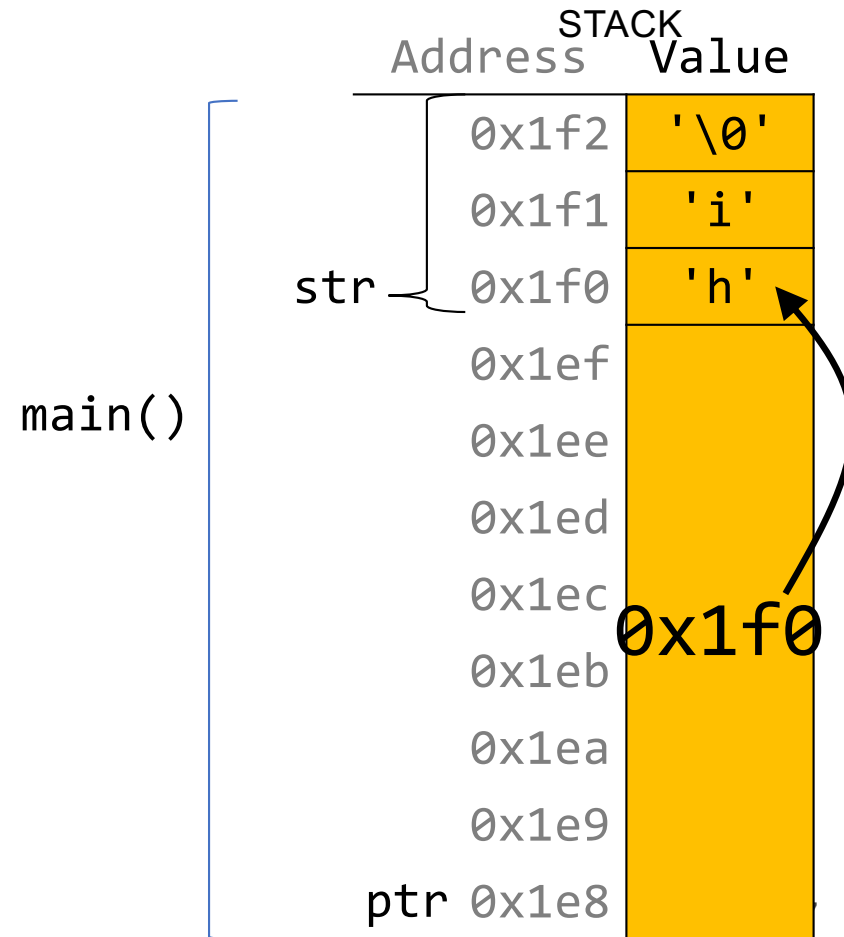


**sizeof** returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **sizeof** to get its full size.

# 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

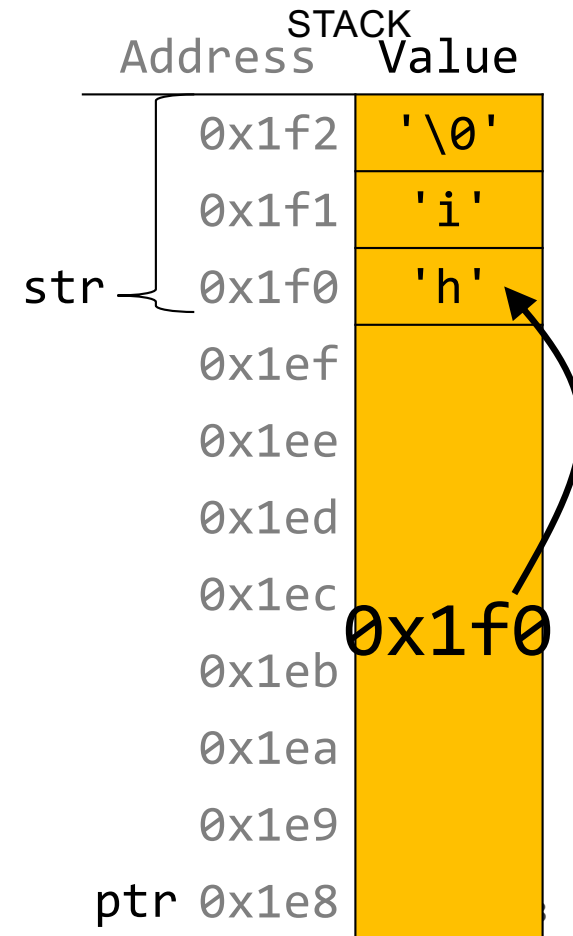
You can also make a pointer equal to an array; it will point to the first element in that array.

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

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

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

main()





# 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
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...

# 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 *nums3 = nums + 3;   // e.g. 0xffc
int *nums2 = nums3 - 1;  // e.g. 0xff8

printf("%d", *nums);     // 52
printf("%d", *nums2);    // 12
printf("%d", *nums3);    // 34
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...

# 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 thirdLetter2 = *(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
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...