CS107, Lecture 6
More Pointers and Arrays

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

Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- **Announcements**
- Pointer Arithmetic

Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- **Announcements**
- Pointer Arithmetic

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

```
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

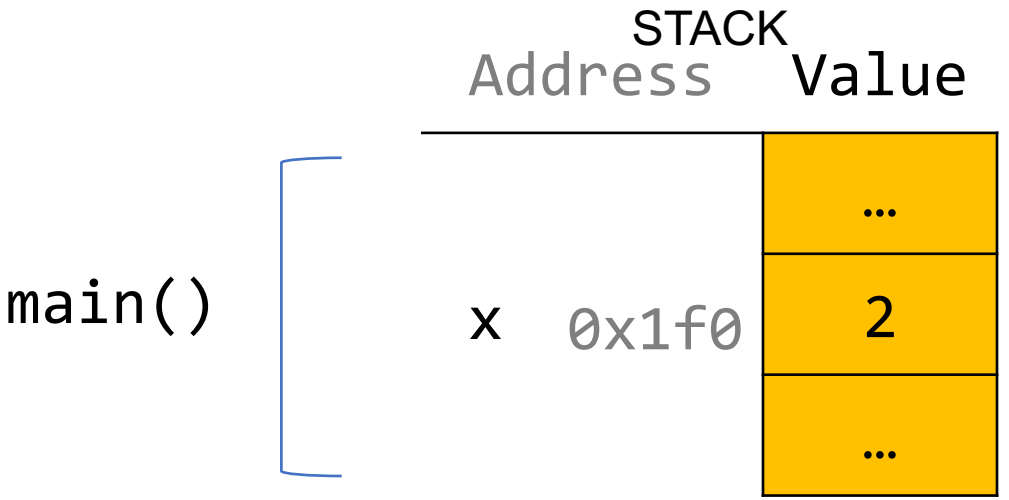
A pointer is just a variable that stores a memory address!

```
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!

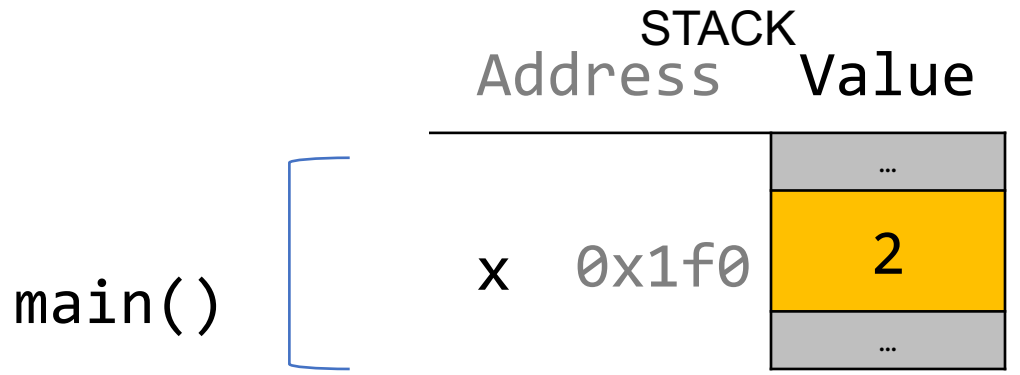
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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!

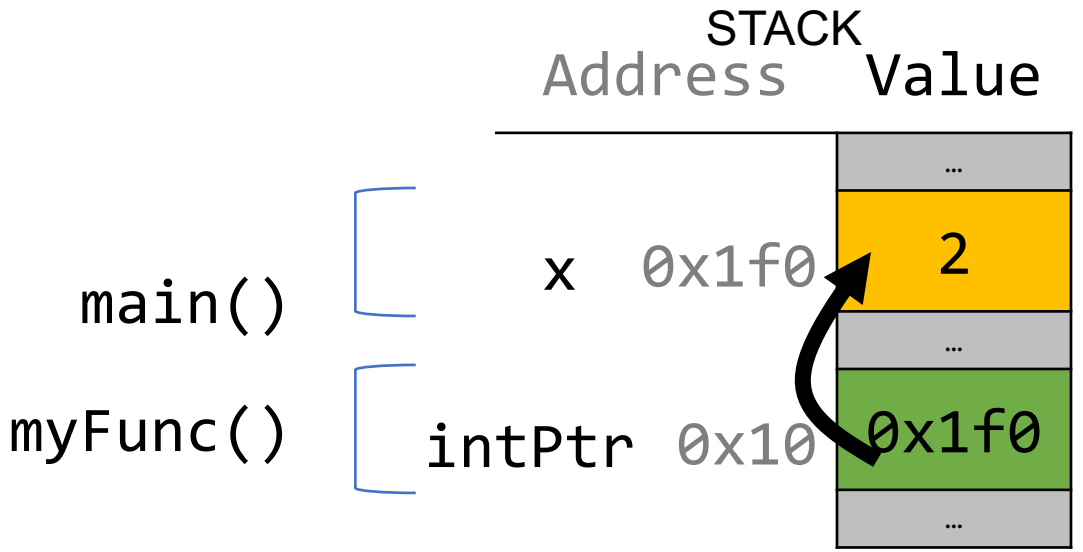
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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!

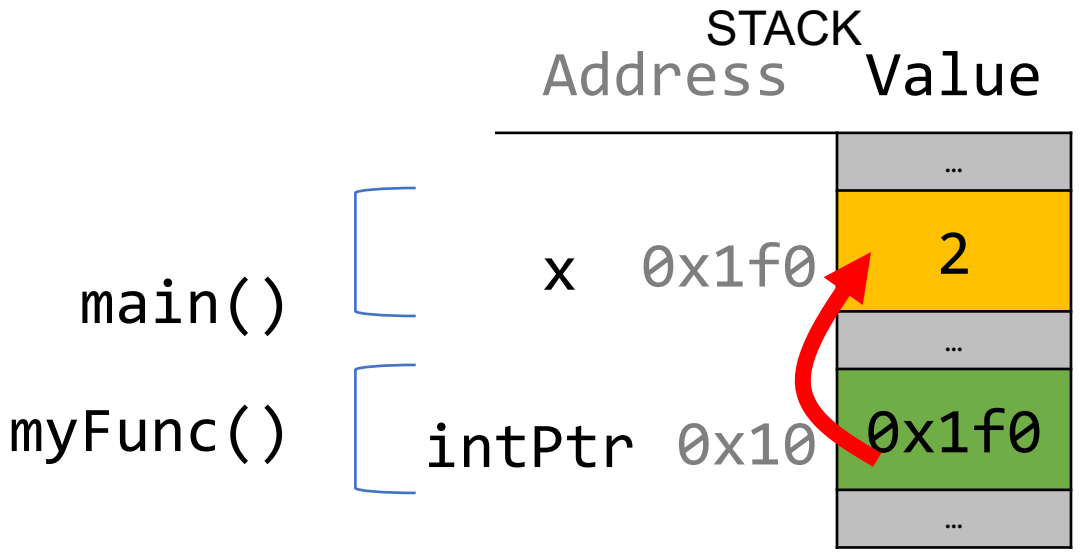
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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!

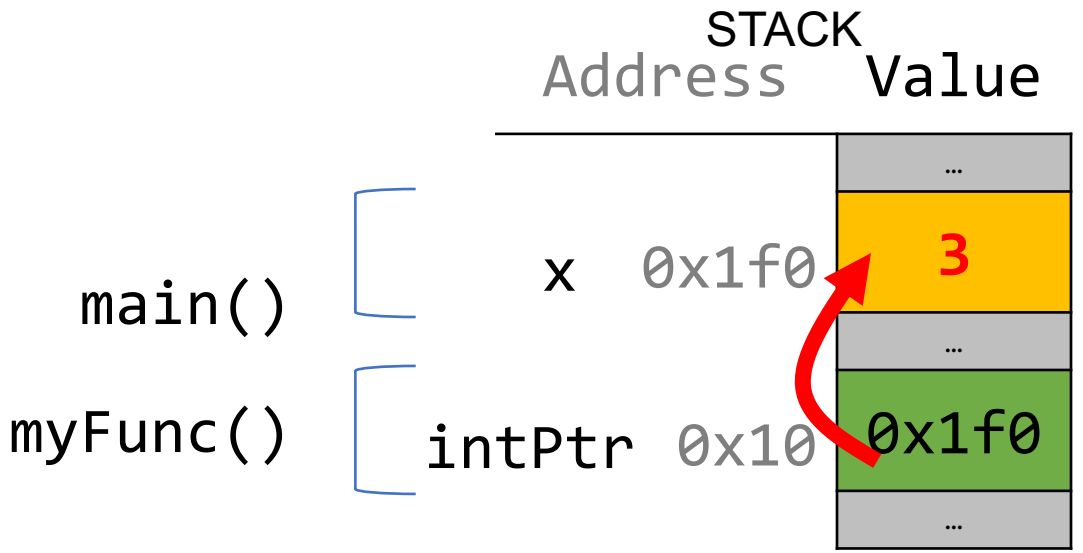
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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!

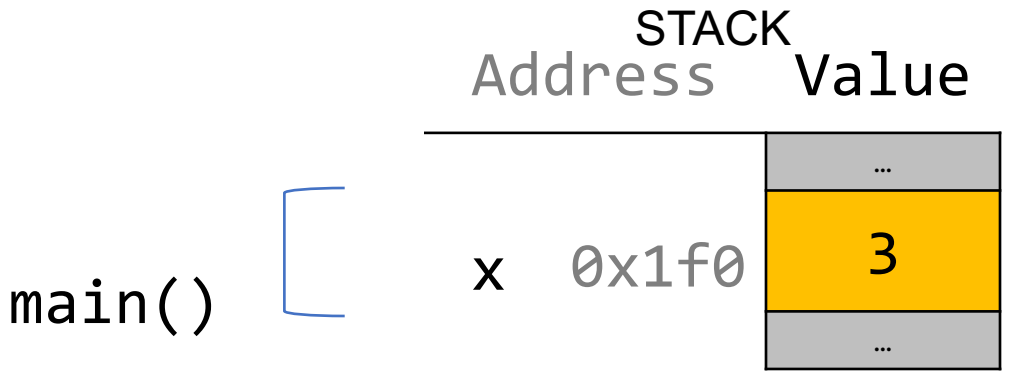
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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!

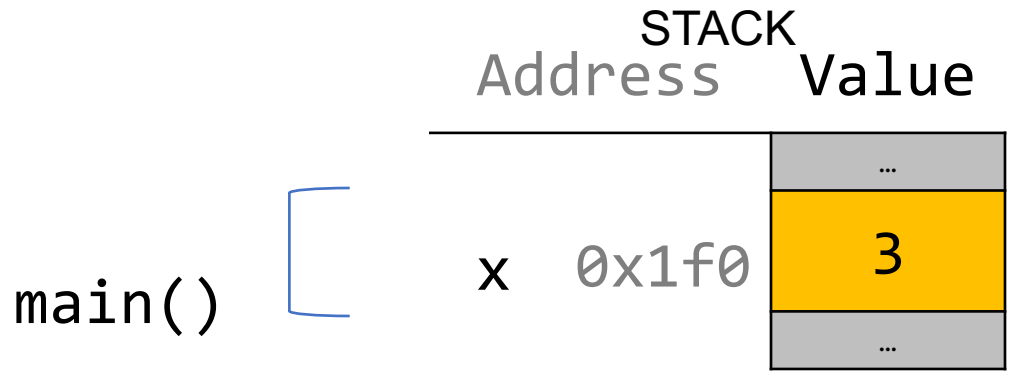
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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!
     ...
}
```



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;
}
```

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?

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

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

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

Exercise 2

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

```
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!
}
```

Exercise 3

We want to write a function that advances a string pointer past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(__?__) {
     int numSpaces = strspn(__?__, " ");
     __?__ += numSpaces;
}

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

Exercise 3

We want to write a function that advances a string pointer past any initial spaces. What should go in each of the blanks?

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

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

We want to write a function that advances a string pointer past any initial spaces. What should go in each of the blanks?

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

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: SkipSpaces

Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- **Announcements**
- Pointer Arithmetic

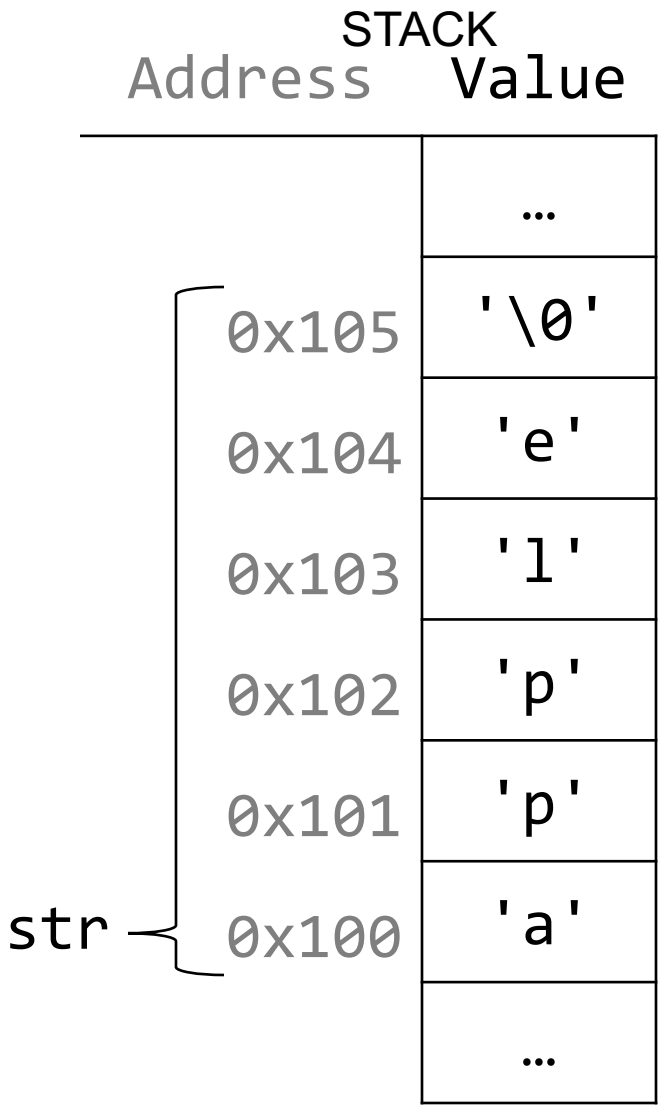
Arrays

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

```
char str[] = "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.

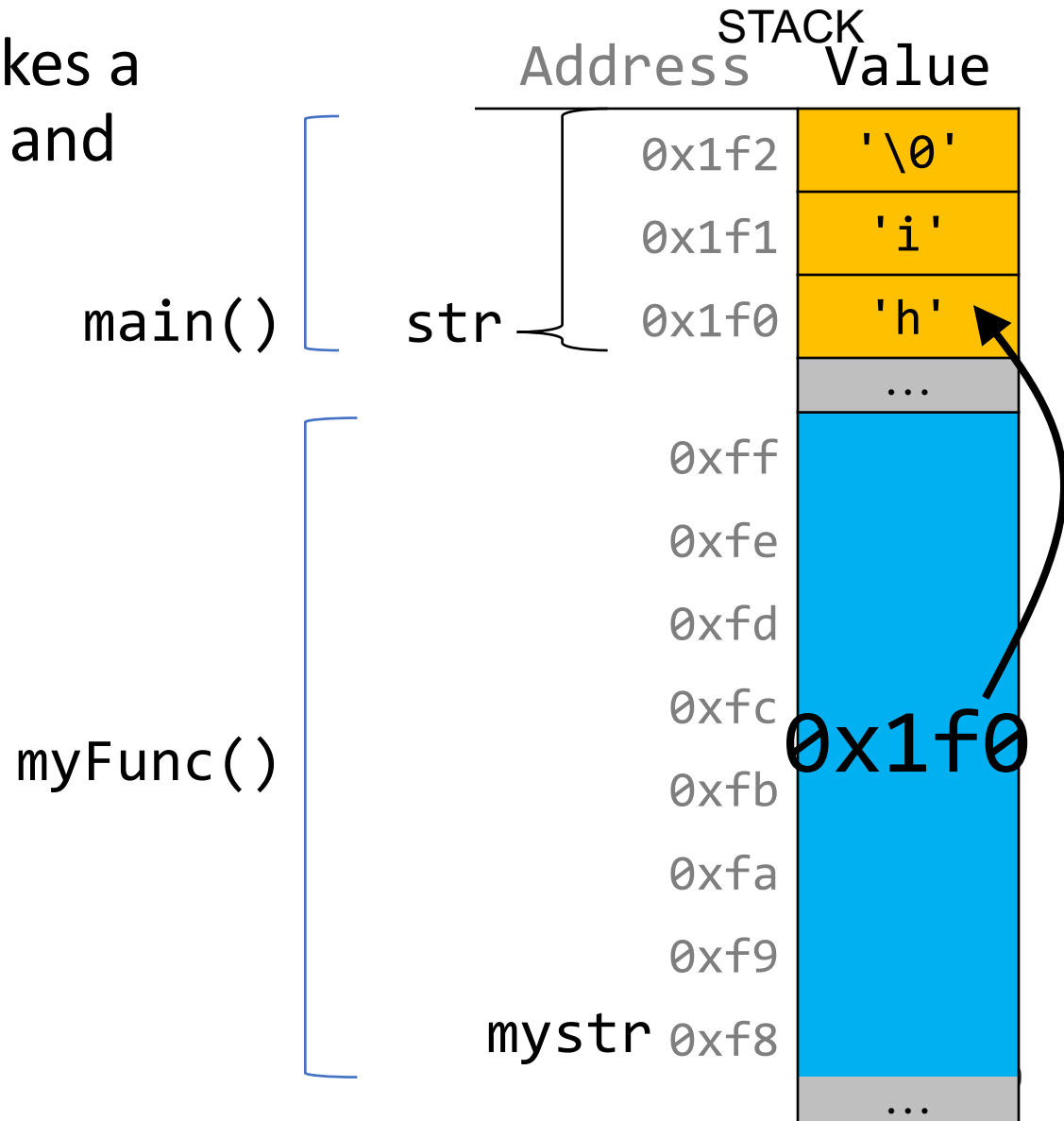
```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

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 (as a pointer) to the function.

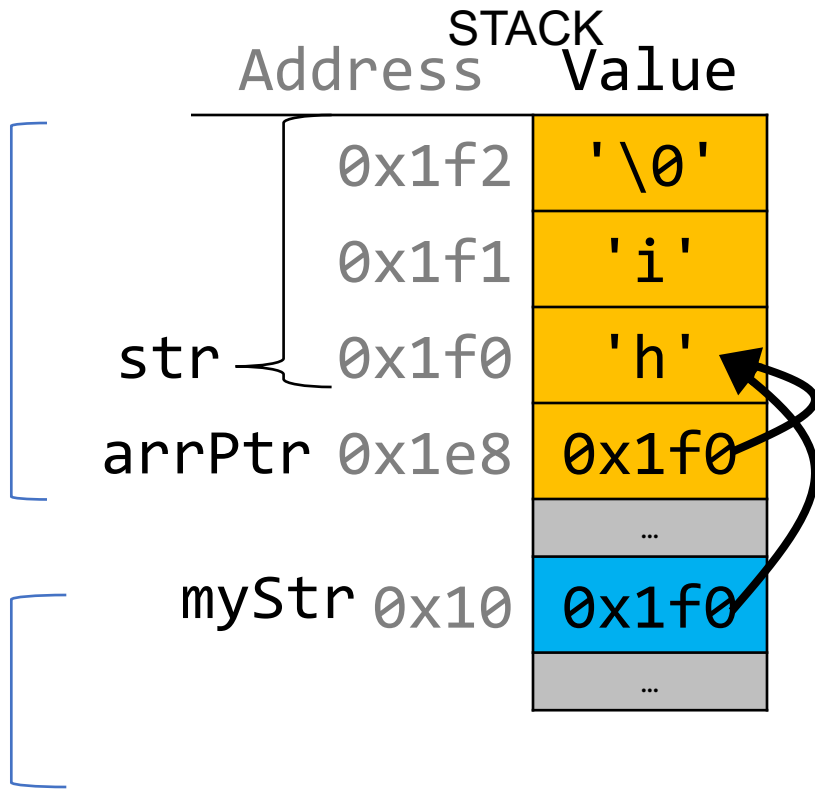
```
void myFunc(char *myStr) {
      …
}

int main(int argc, char *argv[]) {
      char 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 (as a pointer) to the function.

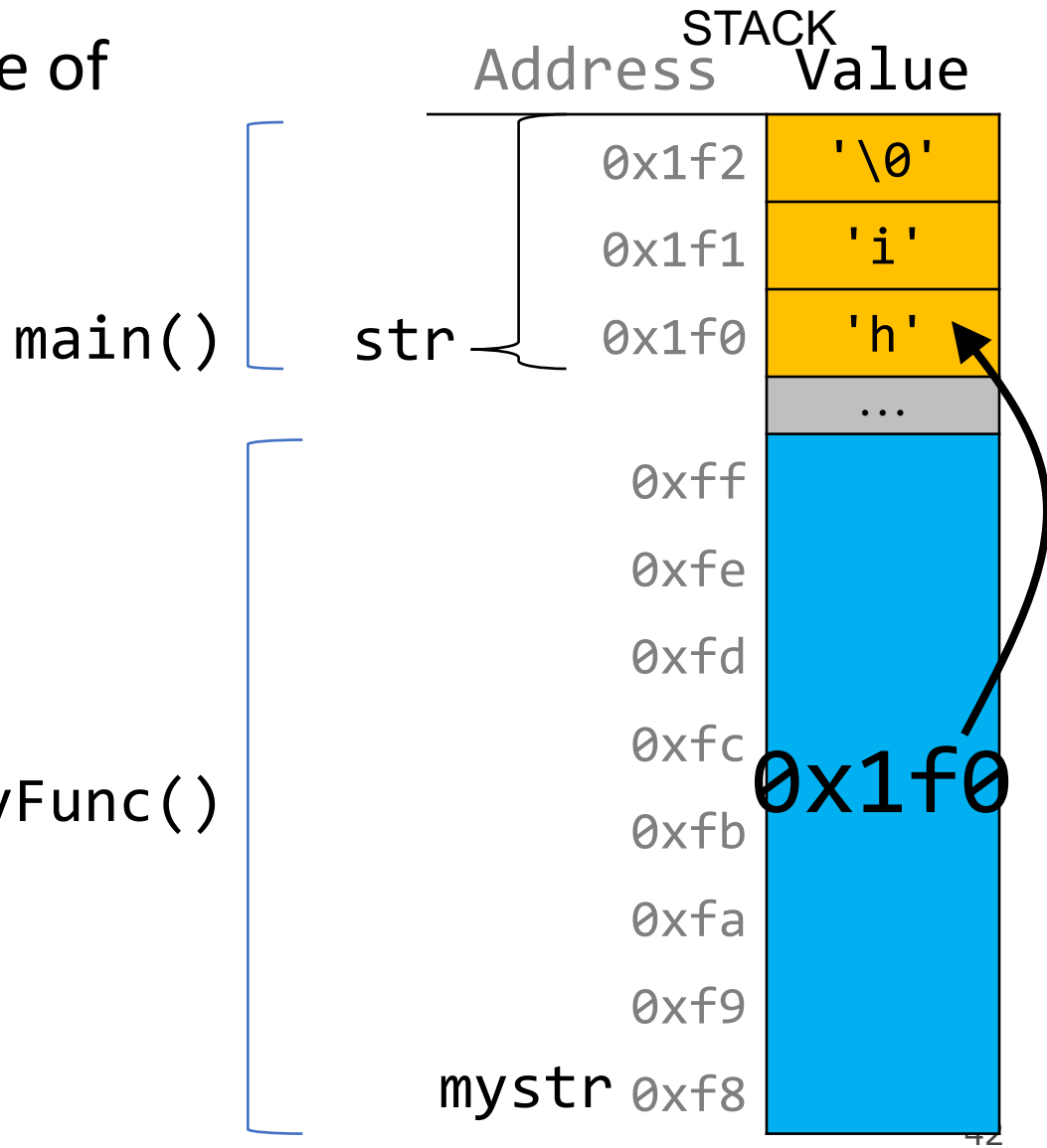
```
void myFunc(char *myStr) {
      …
}

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



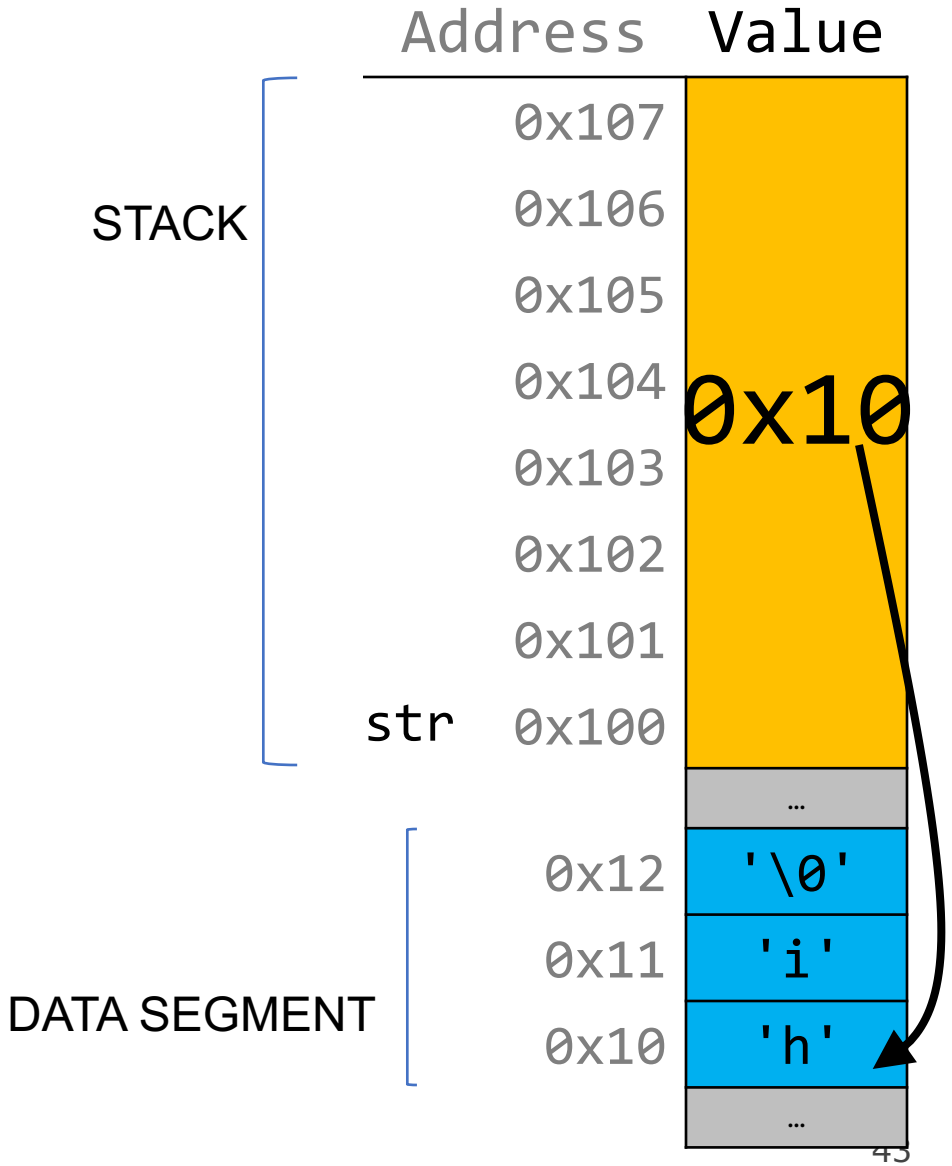
char *

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 "Data 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 data segment*. Since this variable is just a pointer, **sizeof** returns 8, no matter the total size of the string!

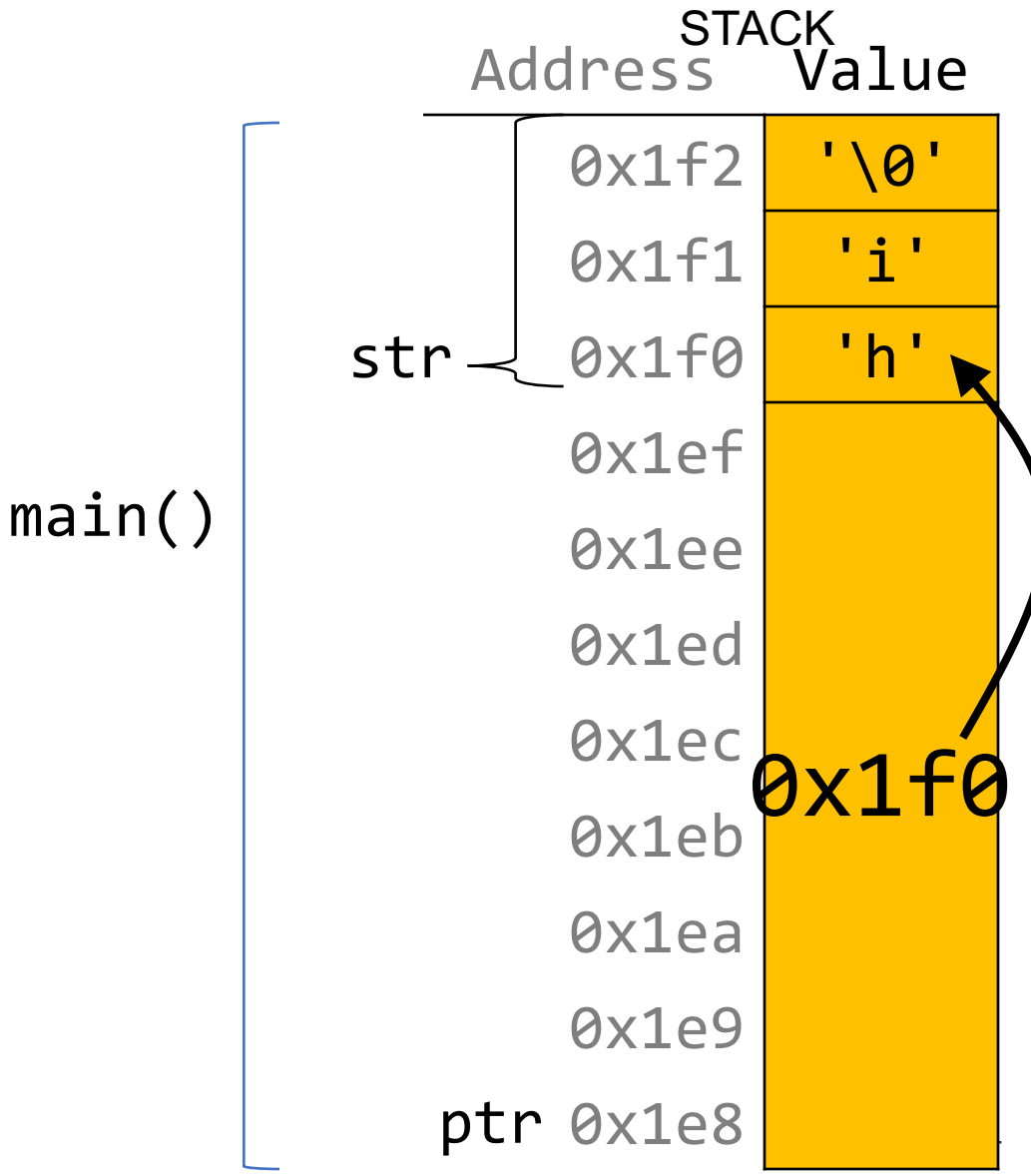
```
int stringBytes = sizeof(str);   // 8
```



Arrays and Pointers

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

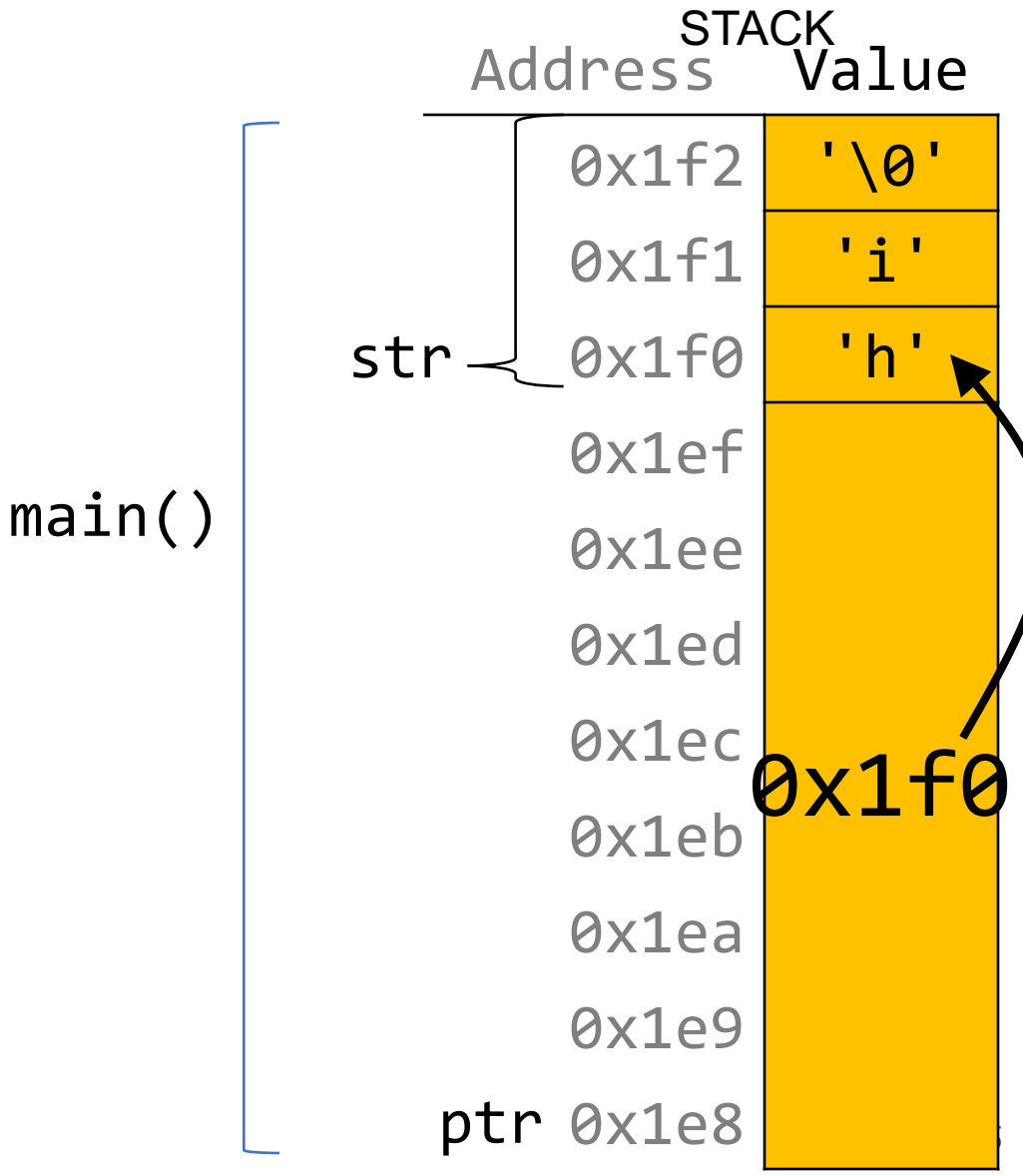
```
int main(int argc, char *argv[]) {
      char str[] = "hi";
      char *ptr = str;
      ...
}
```



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[] = "hi";
      char *ptr = str;

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

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



Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- **Announcements**
- Pointer Arithmetic

Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:

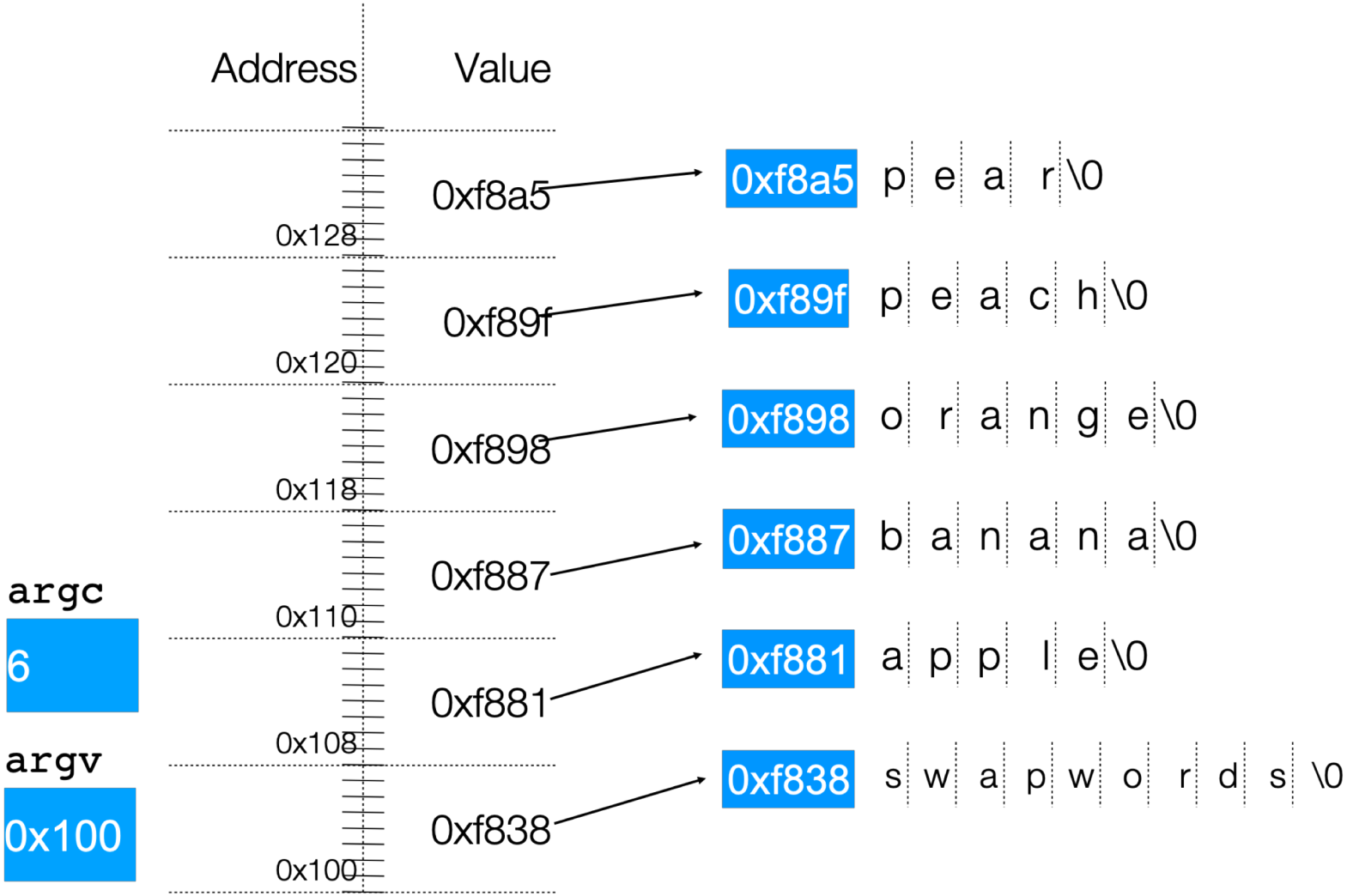
```
char *stringArray[5];   // space to store 5 char *s
```

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

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

Arrays Of Pointers

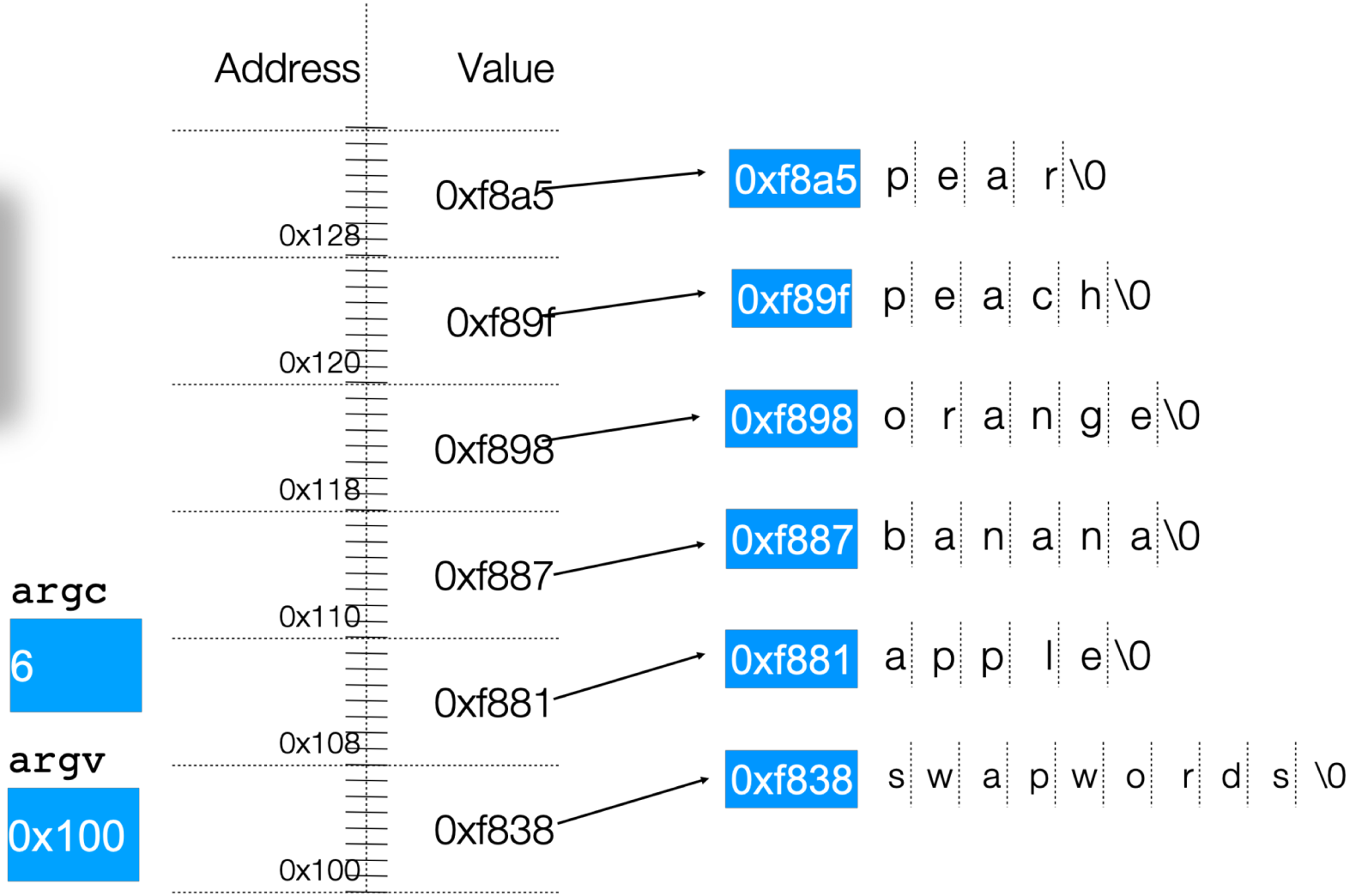
```
./swapwords apple banana orange peach pear
```



Arrays Of Pointers

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

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



Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- **Announcements**
- Pointer Arithmetic

Announcements

- Nick's New Office Hours Location
- Assignment 2 Testing
  - scan_token buffer size: test by modifying tokenize.c or hardcoding into your code
  - env and custom tests: can't use env on custom_tests. Test in terminal instead

Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- **Announcements**
- 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
```

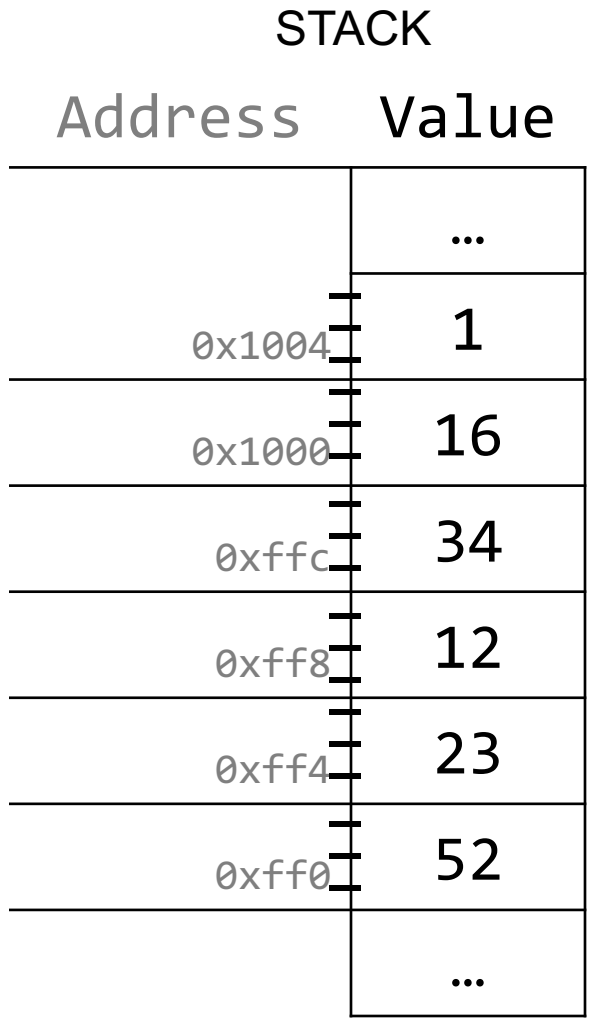
DATA SEGMENT

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

Pointer Arithmetic

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

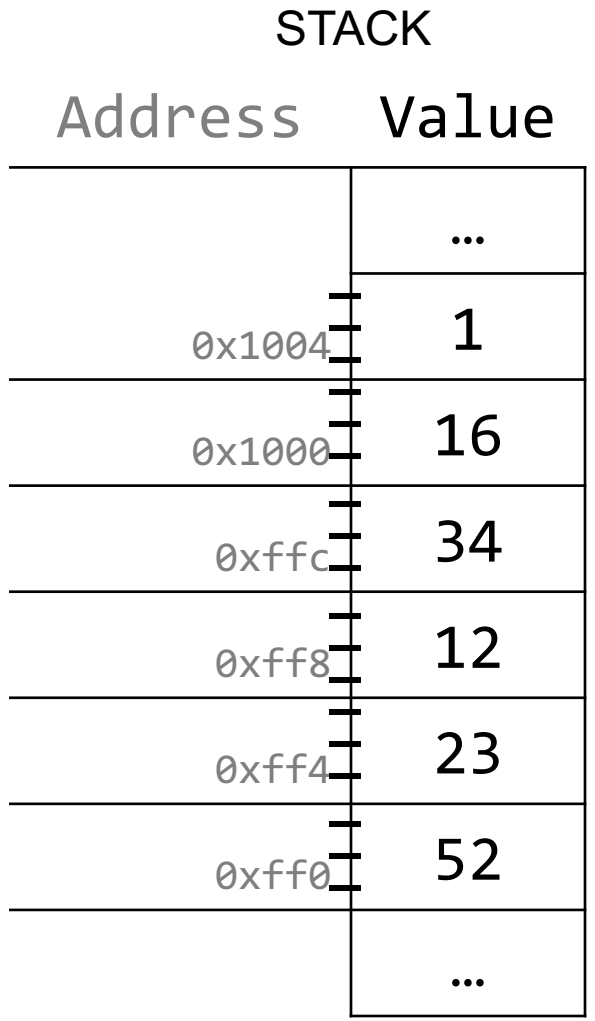
```
// nums points to an int array
int *nums = …                 // e.g. 0xff0
int *nums1 = nums + 1;        // e.g. 0xff4
int *nums3 = nums + 3;        // e.g. 0xffc

printf("%d", *nums);    // 52
printf("%d", *nums1);   // 23
printf("%d", *nums3);   // 34
```



Pointer Arithmetic

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. 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
```



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'
```

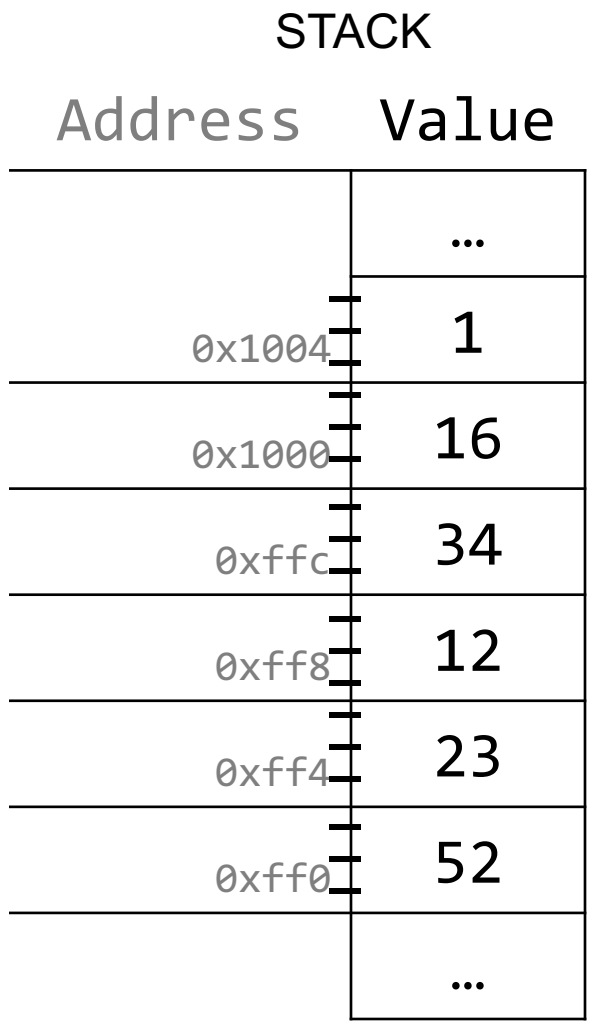
DATA SEGMENT

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
```



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[] = "CS107";

// 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.

Recap

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- **Announcements**
- Pointer Arithmetic

Next time: dynamically allocated memory