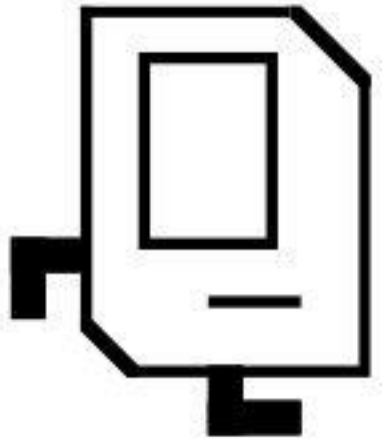
CS107, Lecture 9
Pointers and Arrays

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

CS198 Section Leading



cs198@cs.stanford.edu

Who should section lead?

For this round of applications, we are looking for applicants have completed the equivalent of CS106B… and that's you!

We are looking for section leaders from all backgrounds who can relate to students and clearly explain concepts.

What do section leaders do?

- Teach a weekly 50 minute section
- Help students in the LaIR
- Grade CS106 assignments
- Hold IGs with students
- Grade midterms and finals
- Get paid $18.50/hour (more with seniority)
- Have fun!

Time and requirements

You'll need to:

- Section lead for **two quarters!**
- Take CS198 for 3-4 units (1st quarter only)
- Attend staff meetings (Monday, 4:30-5:30PM)
- Attend Monday workshops (7:30-9pm) for first 4 weeks of first quarter
- Attend Wednesday workshops (based on availability) for first 4 weeks of first quarter
- Fulfill all teaching, LaIR, and grading responsibilities

Why section lead?

- “Learn to teach; teach to learn”
- Work directly with students
- Participate in fun events
- Join an amazing group of people
- Leave your mark on campus

Participate in fun events

- LaIR Formal
- Special D
- BAWK
- Lecturer Hangouts
- New SL Picnic
- Swag
- And more!

Apply Now

Application is open now!

Deadline: **Thursday, April 24th at 11:59PM PT**

Online application: cs198.stanford.edu

Contact us: cs198@cs.stanford.edu

CS107 Topic 3: How can we effectively manage all types of memory in our programs?

CS107 Topic 3

How can we effectively manage all types of memory in our programs?

Why is answering this question important?

- Shows us how we can pass around data efficiently with pointers (this time)
- Introduces us to the heap and allocating memory that we manually manage (next time)
- Helps us better understand use-after-free vulnerabilities, a common bug (next week)

assign3: implement a function using resizable arrays to read lines of any length from a file and write 2 programs using that function to print the last N lines of a file and print just the unique lines of a file. These programs emulate the **tail** and **uniq** Unix commands!

Learning Goals

- Learn about pointers and how they help us access data without making copies
- Understand arrays and how they relate to pointers
- Get more practice using memory diagrams to understand code behavior

Lecture Plan

- **Review:** Pointers
- Double Pointers
- Pointer Arithmetic
- Arrays in Memory

```
cp -r /afs/ir/class/cs107/lecture-code/lect9 .
```

Lecture Plan

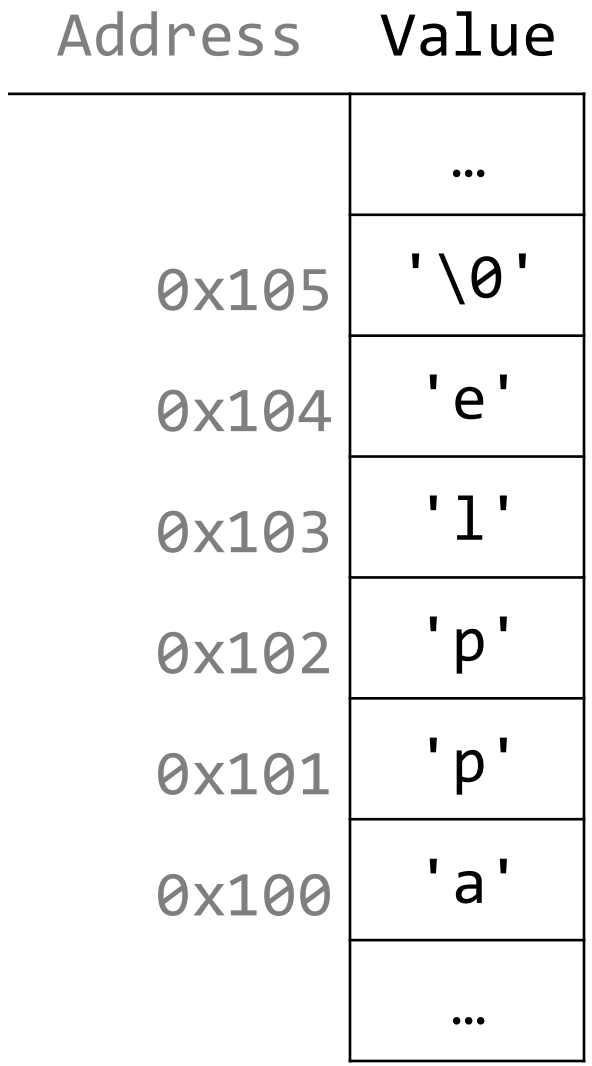
- **Review: Pointers**
- Double Pointers
- Pointer Arithmetic
- Arrays in Memory

```
cp -r /afs/ir/class/cs107/lecture-code/lect9 .
```

Pointers and Memory

A *pointer* is a variable that stores a memory address.

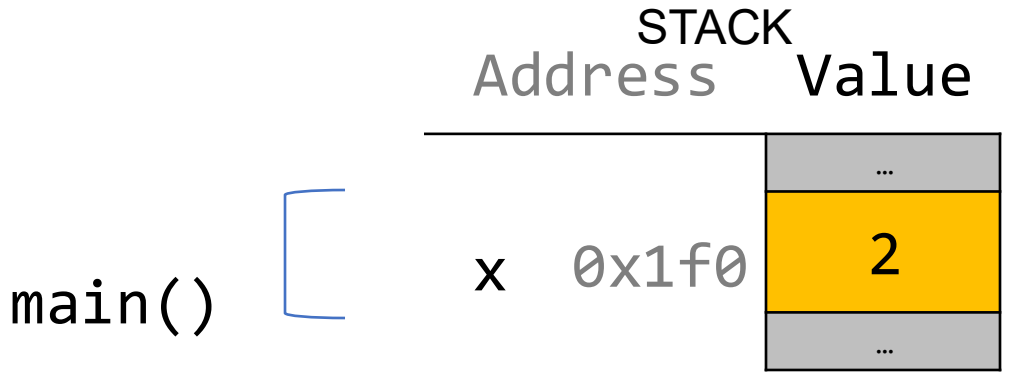
- Memory is a big array of bytes, and each byte has a unique numeric index that is commonly written in hexadecimal. A pointer stores one of these "indexes".
- Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
- Pointers are also essential for allocating memory on the heap, and to refer to memory generically, both of which we will cover later.



Pass By Value

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

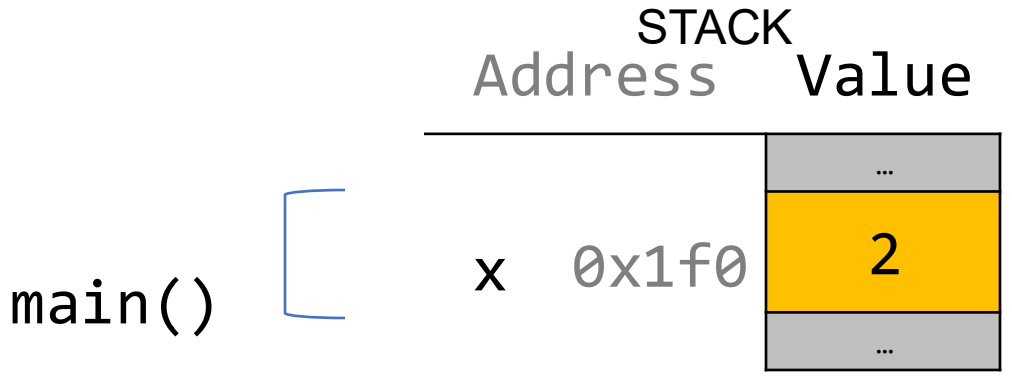
```
void myFunc(int val) {
     val = 3;
}

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



Pass By Value

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

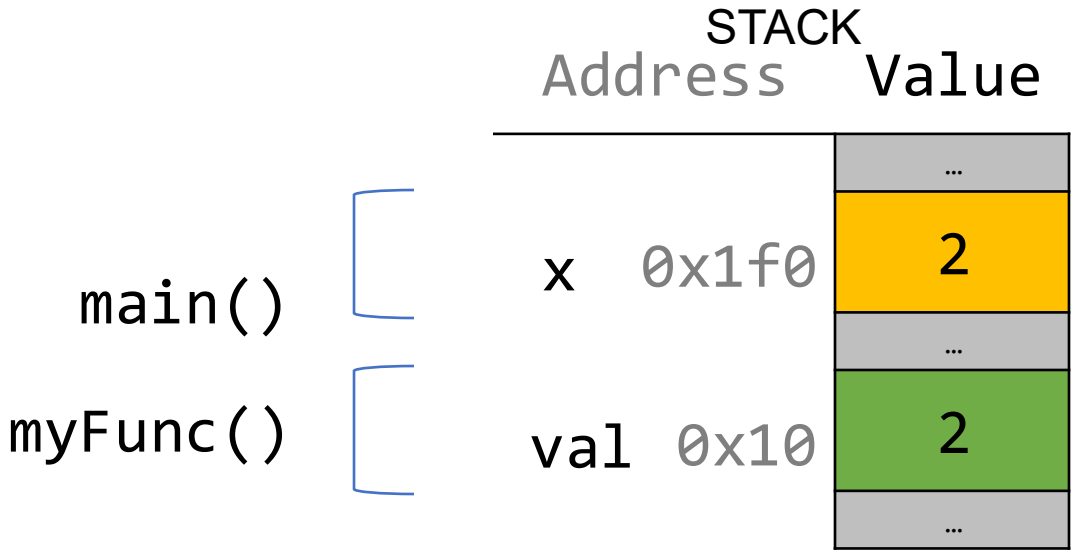
```
void myFunc(int val) {
     val = 3;
}

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



Pass By Value

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

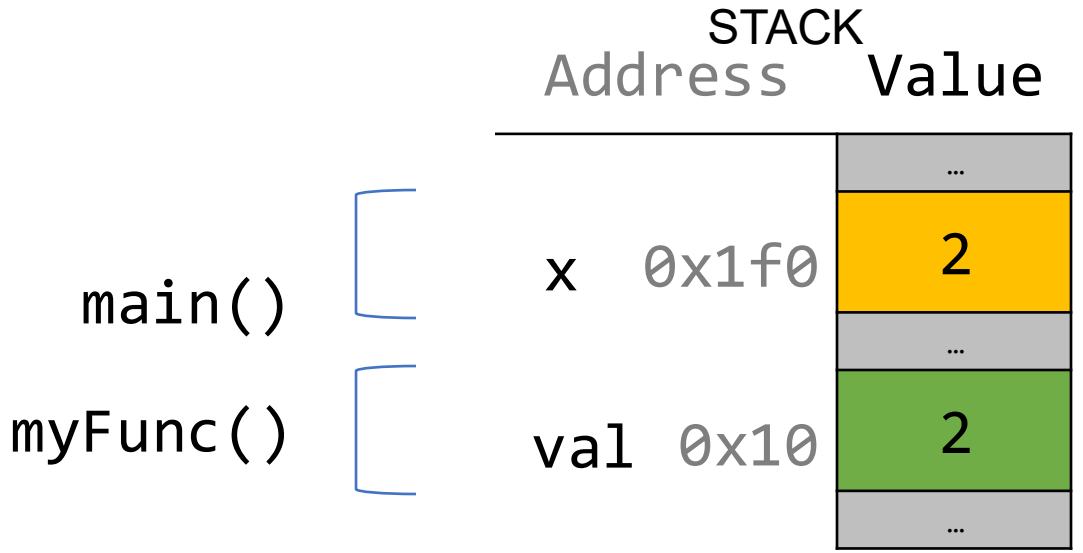
```
void myFunc(int val) {
     val = 3;
}

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



Pass By Value

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

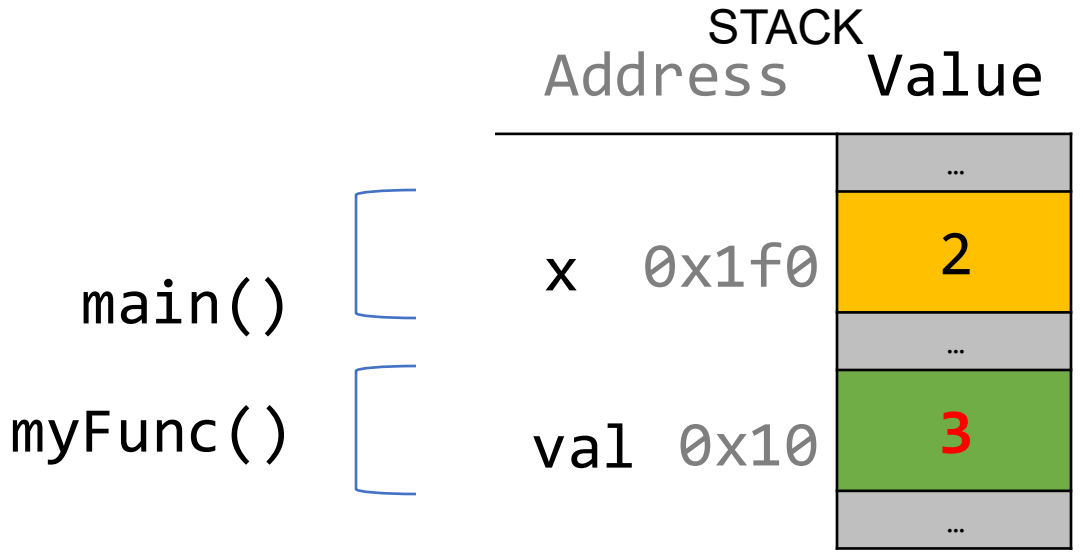
```
void myFunc(int val) {
     val = 3;
}

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



Pass By Value

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

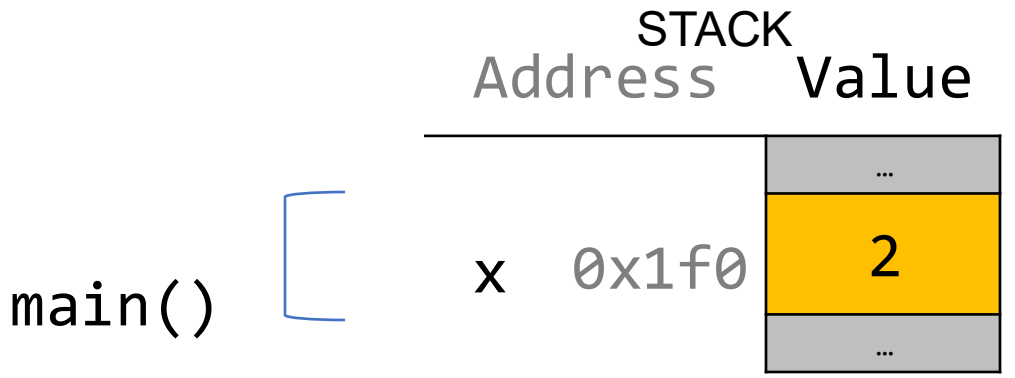
```
void myFunc(int val) {
     val = 3;
}

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



Pass By Value

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

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

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



Pointers

Pointers allow us to pass around the *location* of data so that the original data can be modified in other functions.

Example: I want to write a function *myFunc* that can change the value of an existing integer to be 3.

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

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
```

`*`

If **declaration**: “pointer”
ex: `int *` is "pointer to an int”
If **operation**: "dereference/the value at address”
ex: `*num` is "the value at address num"

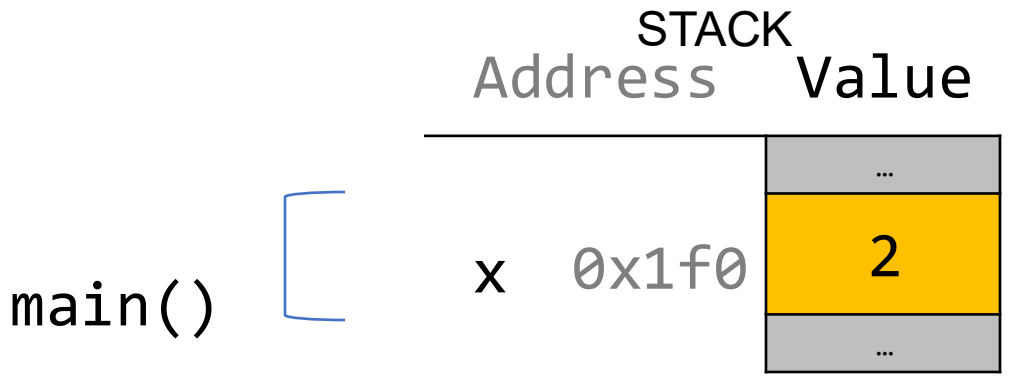
Pointers

```
int x = 2;

// & adds a * to the type (e.g. int -> int *)
int *xPtr = &x;

// Dereferencing removes a * from the type (e.g. int * -> int)
int y = *xPtr;
```

Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

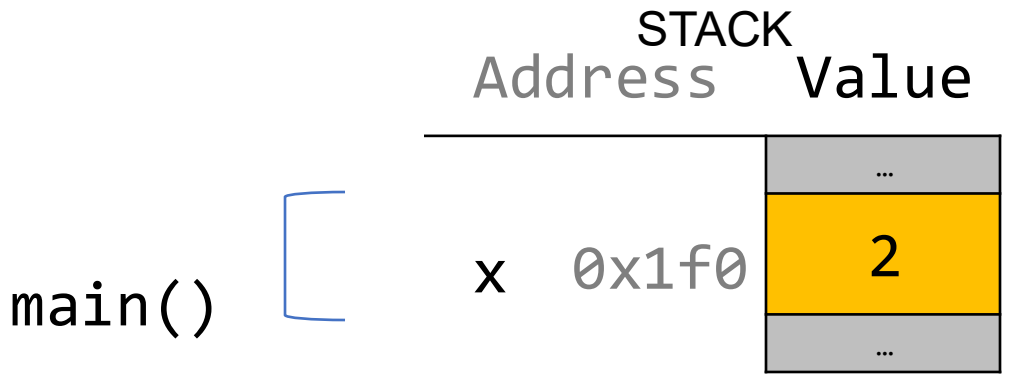
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

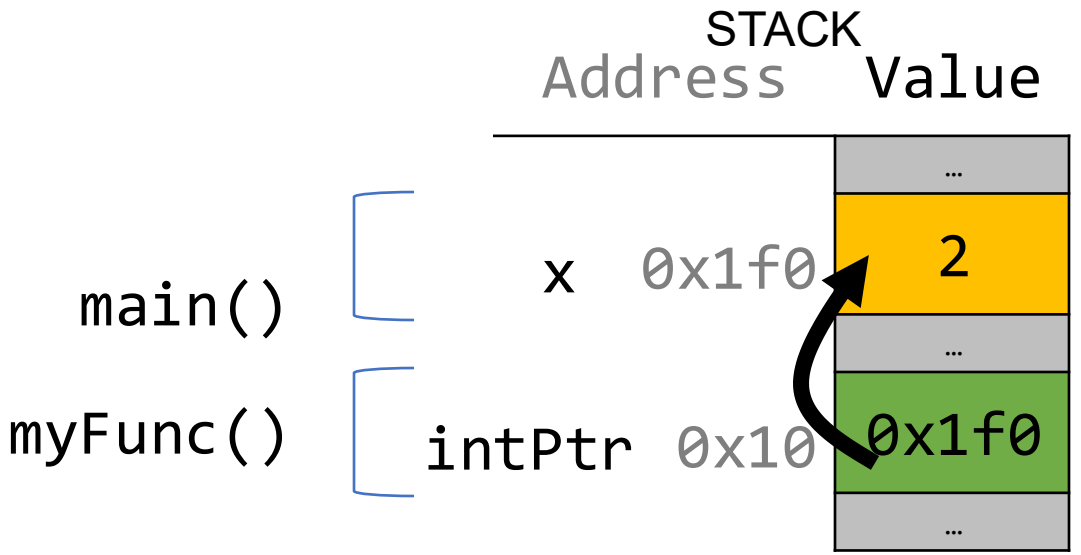
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

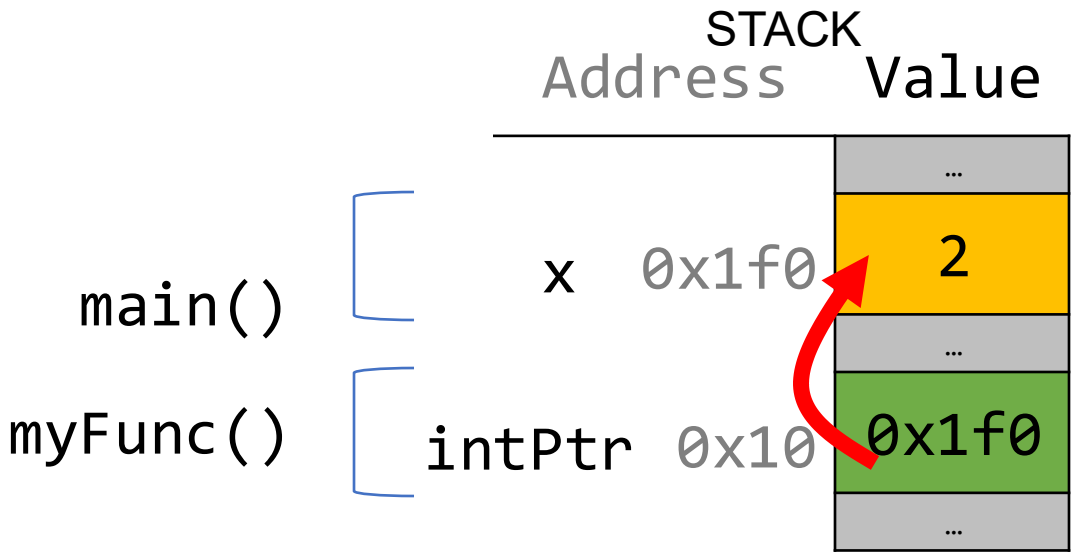
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

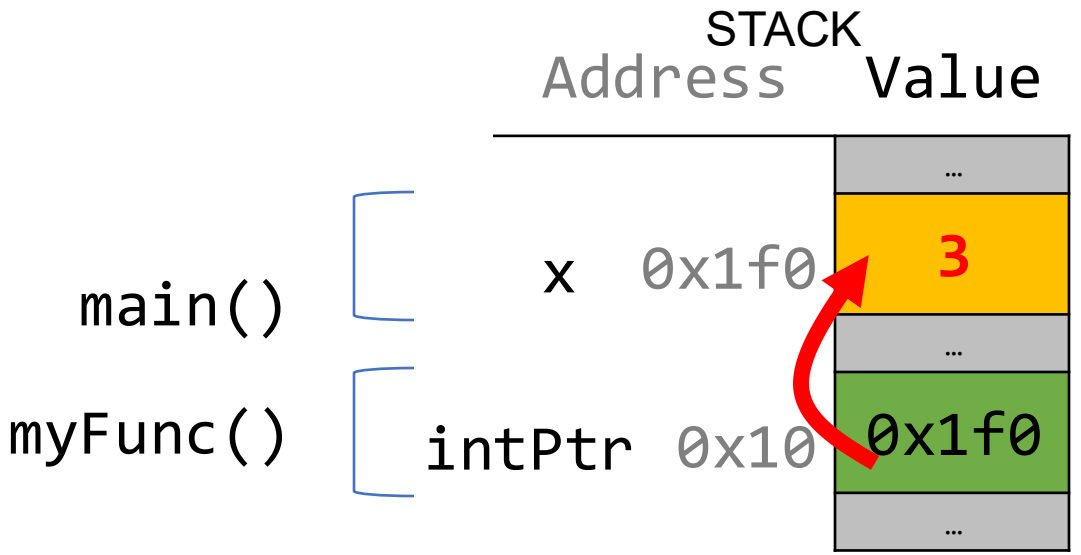
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

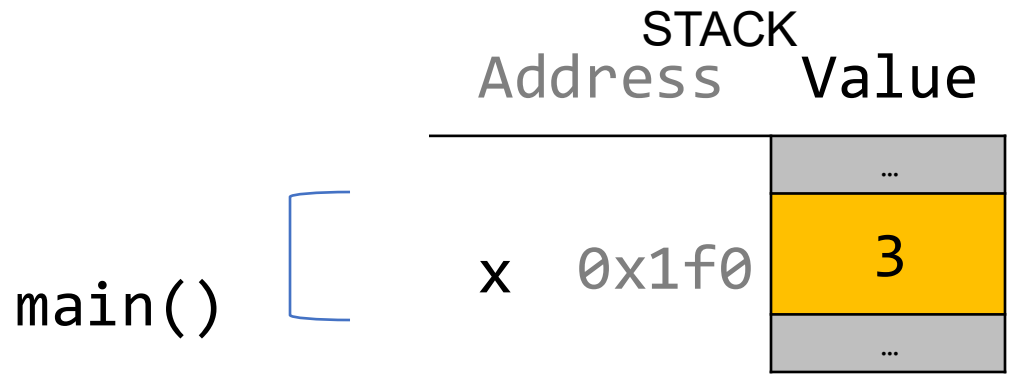
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

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



Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

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



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.
- If you are modifying a specific instance of some value, pass the *location* of what you would like to modify and dereference that location to access what's there.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

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.

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) {
     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.

Pointers Practice

```
void makeUpper(char *ptr) {
      __1__ = toupper(__2__);
}

int main(int argc, char *argv[]) {
      char ch = 'h';

      // want to modify ch to be capital
      makeUpper(__3__);
      printf("%c\n", ch); // should print 'H'
      return 0;
}
```

What should go in each of the blanks so that this code correctly modifies **ch** to be capitalized?

Pointers Practice

```
void makeUpper(char *ptr) {
      *ptr = toupper(*ptr);
}

int main(int argc, char *argv[]) {
      char ch = 'h';

      // want to modify ch to be capital
      makeUpper(&ch);
      printf("%c\n", ch); // should print 'H'
      return 0;
}
```

We are modifying a specific instance of the letter, so we pass the *location* of the letter we would like to modify.

Pointers Summary

```
void doSomething(TYPE *ptr) {
      // access/change val with *ptr
}

int main(int argc, char *argv[]) {
      TYPE myVariable = ...

      // want to modify myVariable
      makeUpper(&myVariable);
      // now myVariable should be different
      return 0;
}
```

General rule: if we want to modify a variable with type TYPE in another function, we pass in TYPE * as a parameter. We can then dereference it within the function to get/set the value pointed to.

Lecture Plan

- **Review**: Pointers
- **Double Pointers**
- Pointer Arithmetic
- Arrays in Memory

```
cp -r /afs/ir/class/cs107/lecture-code/lect9 .
```

Exercise 3

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

Respond on PollEv:
pollev.com/cs107

```
void skipSpaces(__1__) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(__2__);
    printf("%s", str);      // should print "hello"
}
```

What should go in each of the blanks so that this code correctly modifies str to skip leading spaces?

Exercise 3

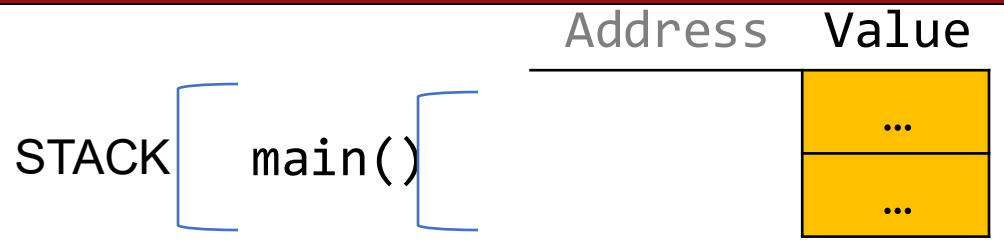
Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(char **strPtr) {
     ...
}

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

We are modifying a specific instance of the string pointer, so we pass the *location* of the string pointer we would like to modify.

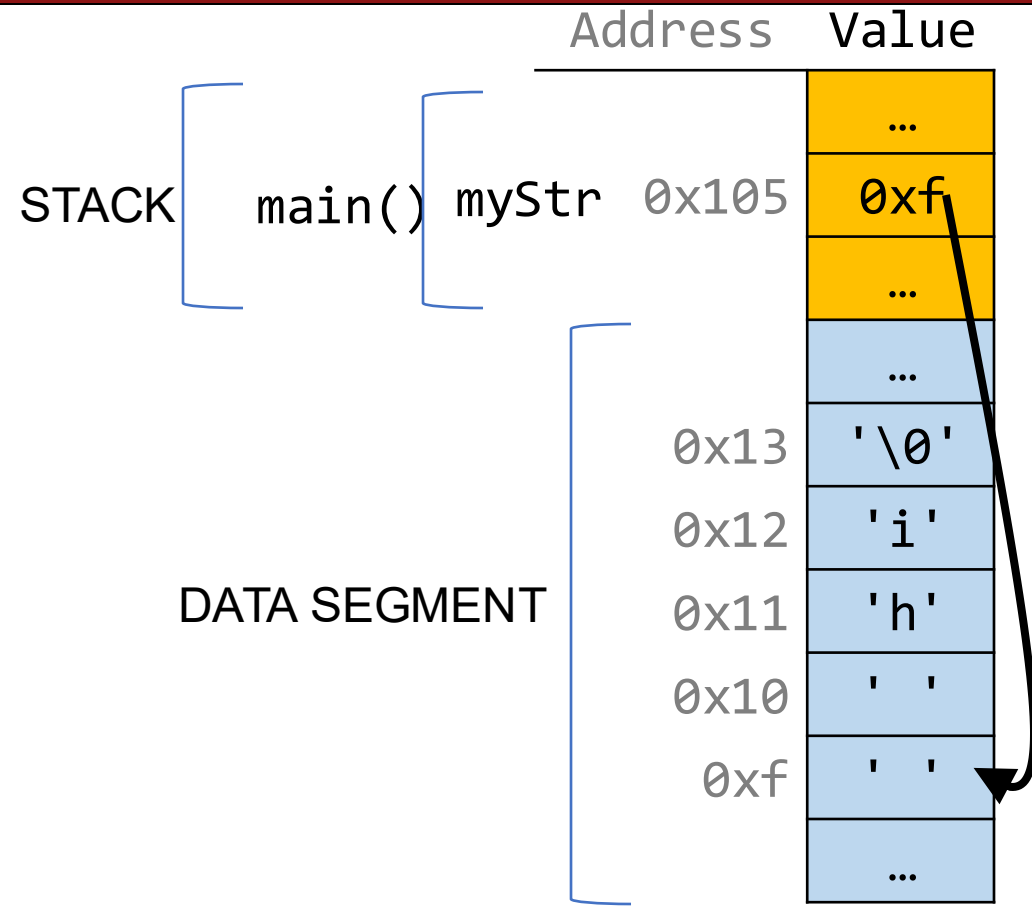
Pointers to Strings

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

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



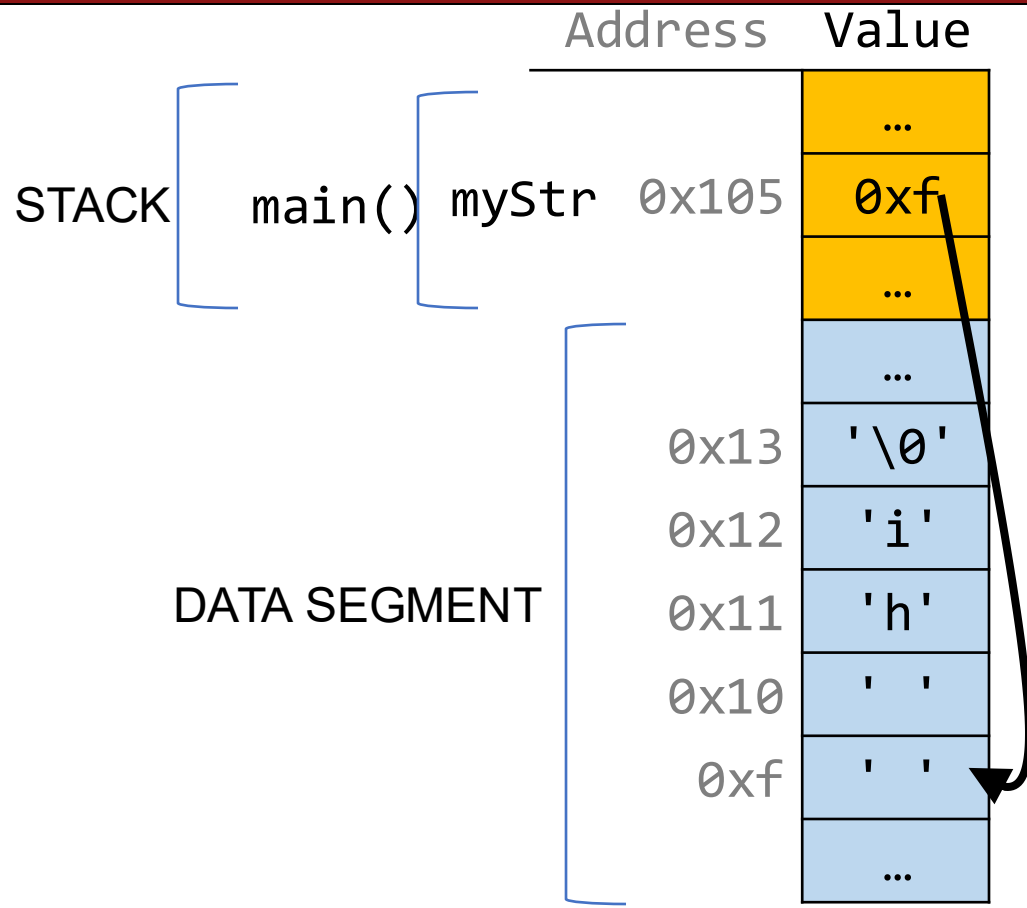
Pointers to Strings

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

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



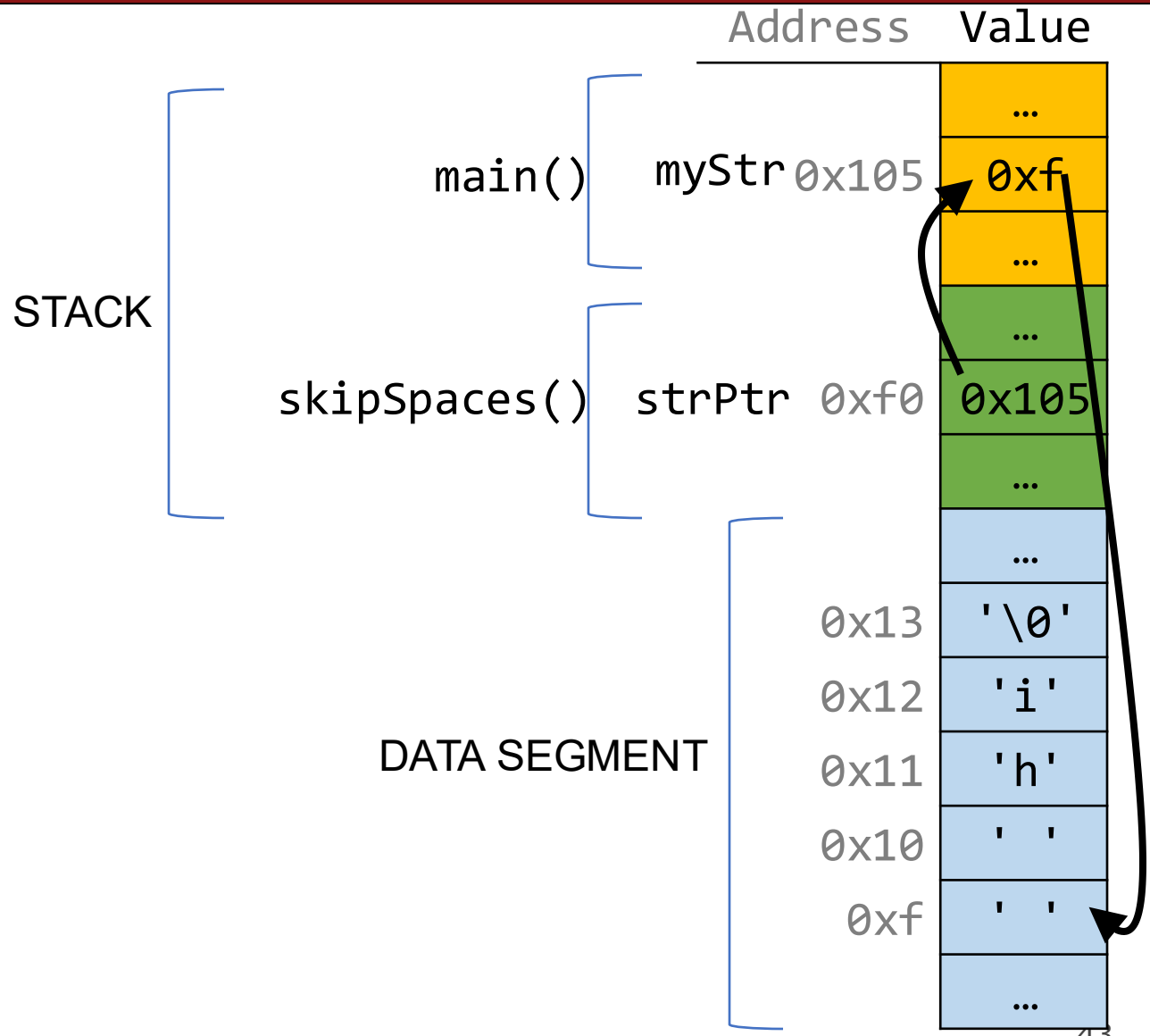
Pointers to Strings

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

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



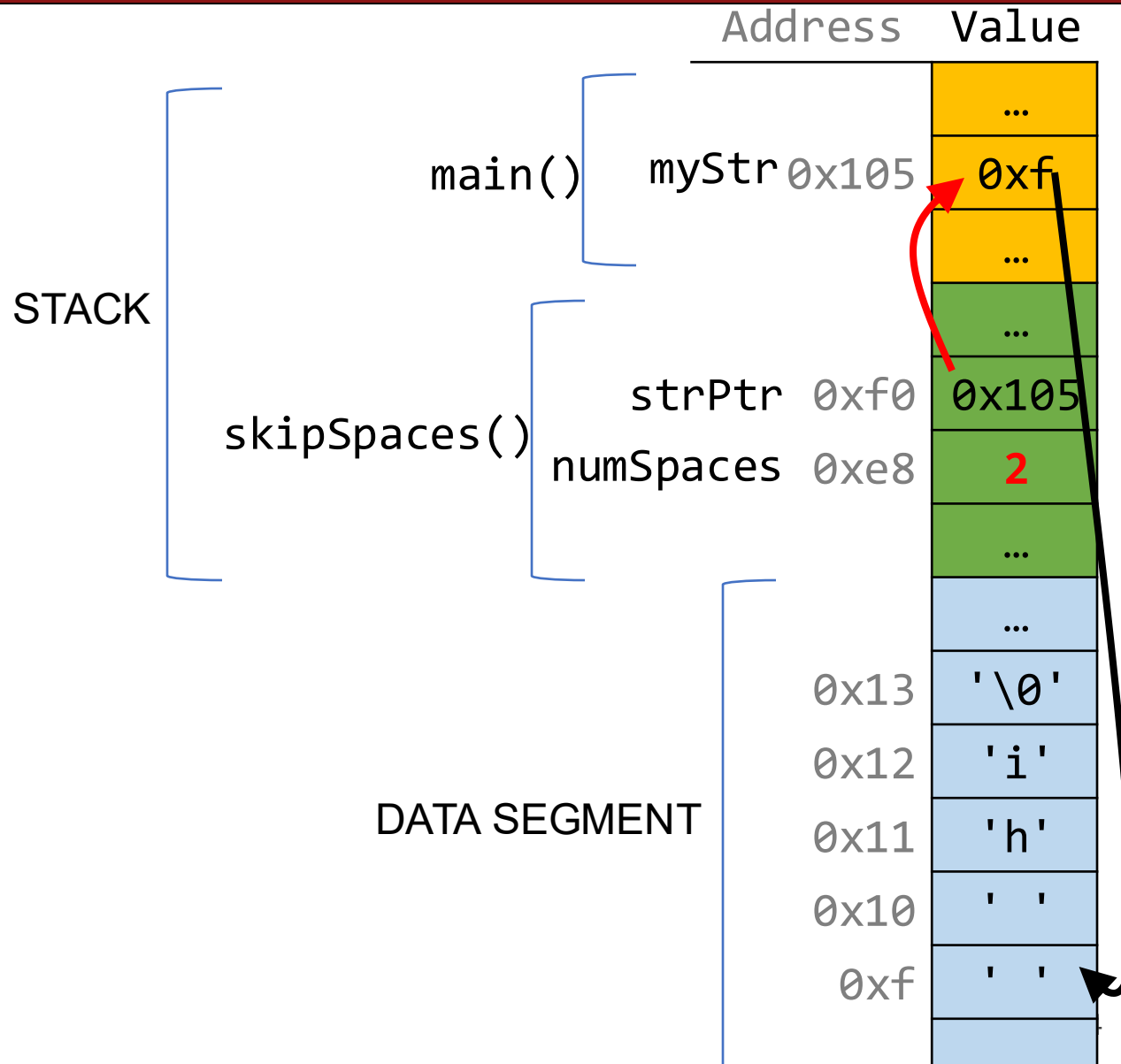
Pointers to Strings

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

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



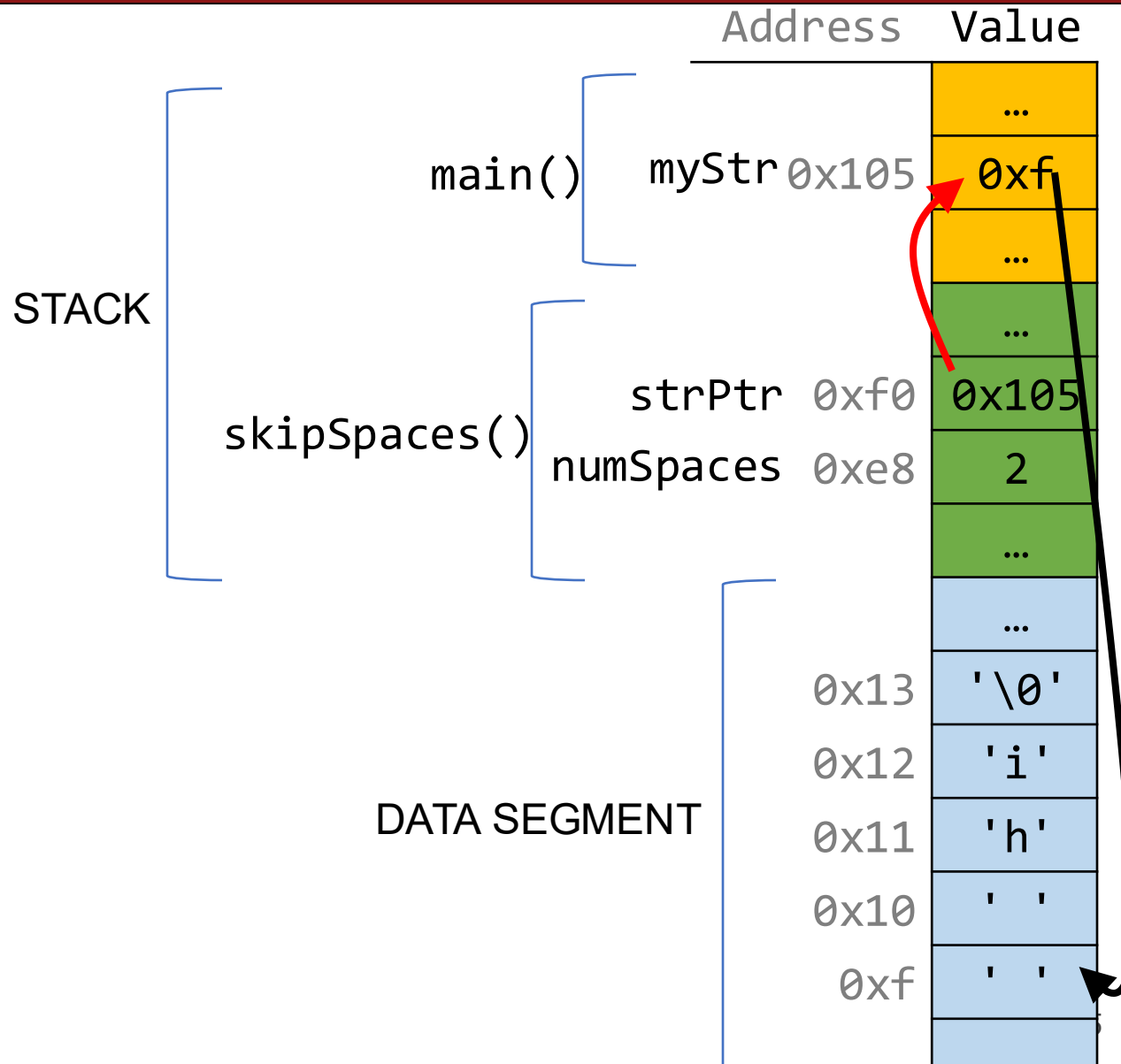
Pointers to Strings

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

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



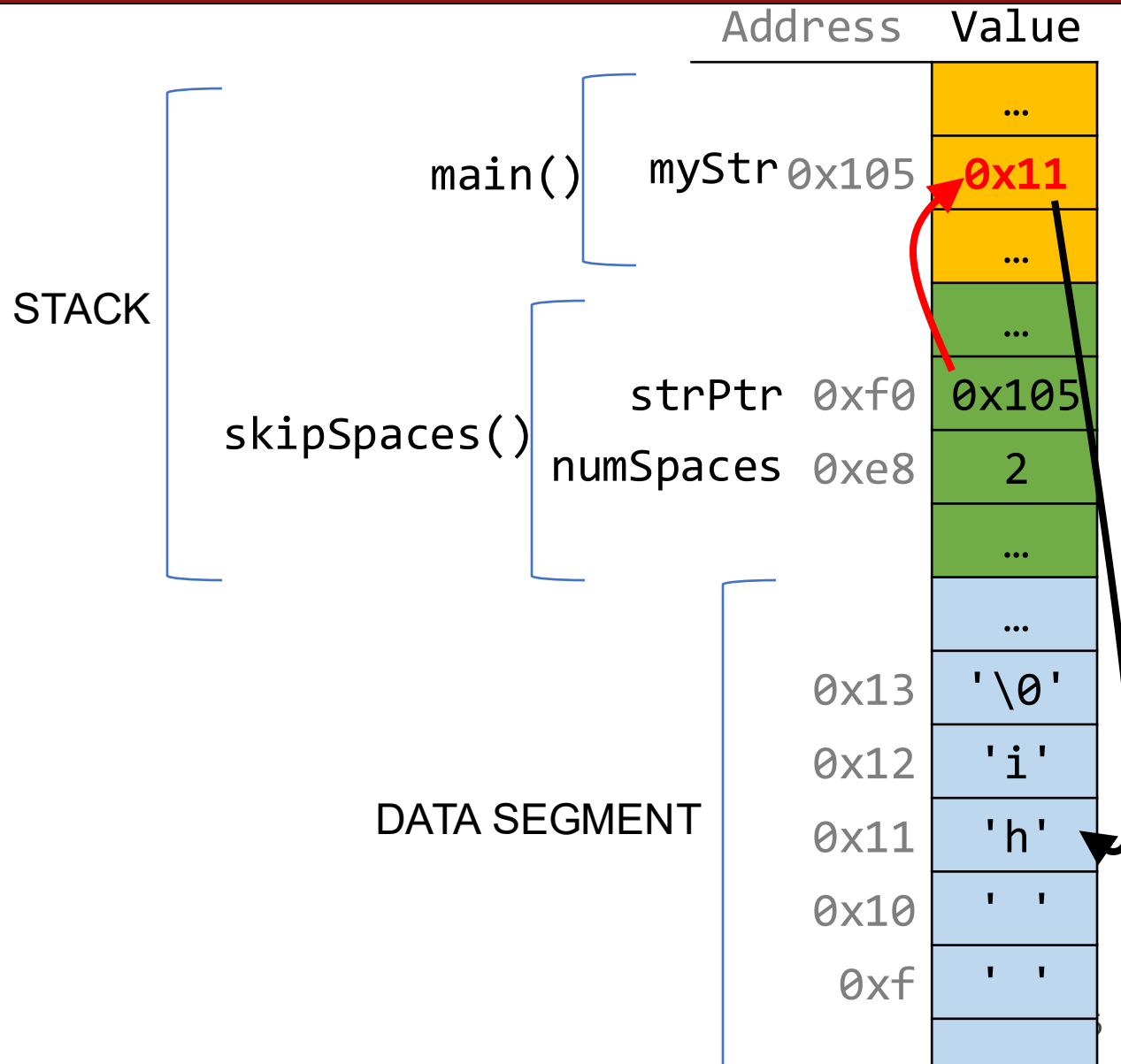
Pointers to Strings

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

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



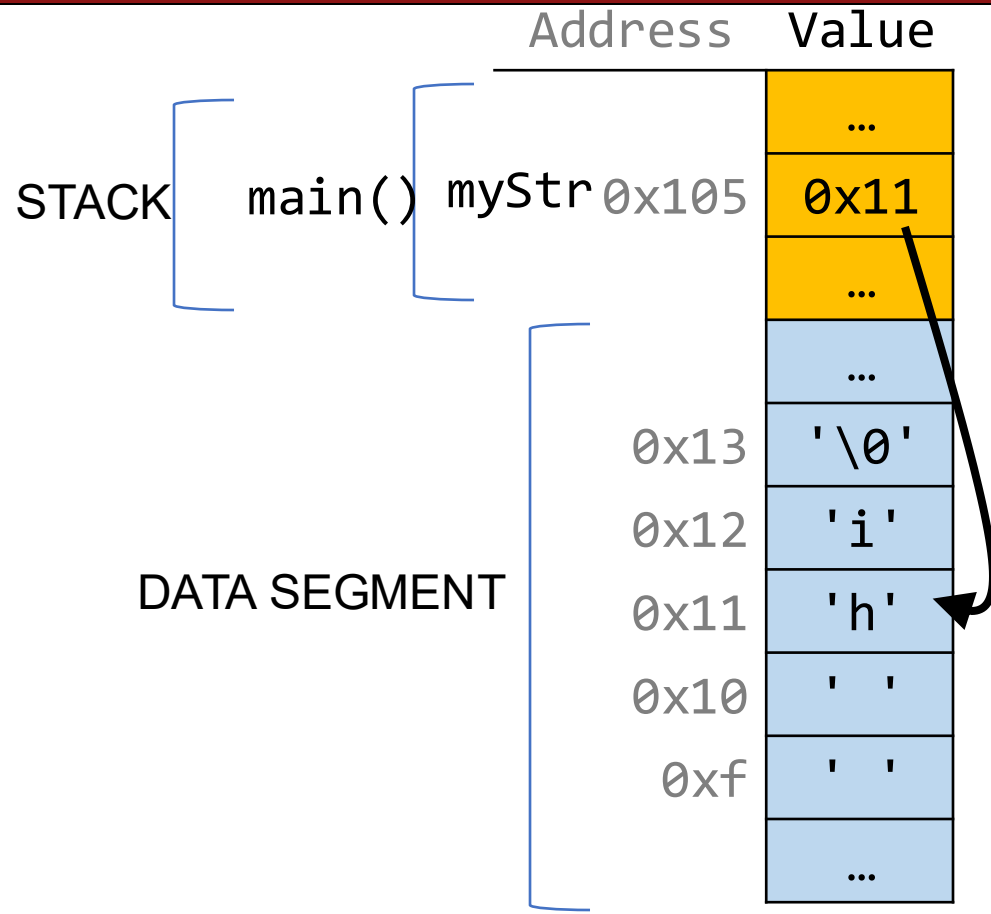
Pointers to Strings

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

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



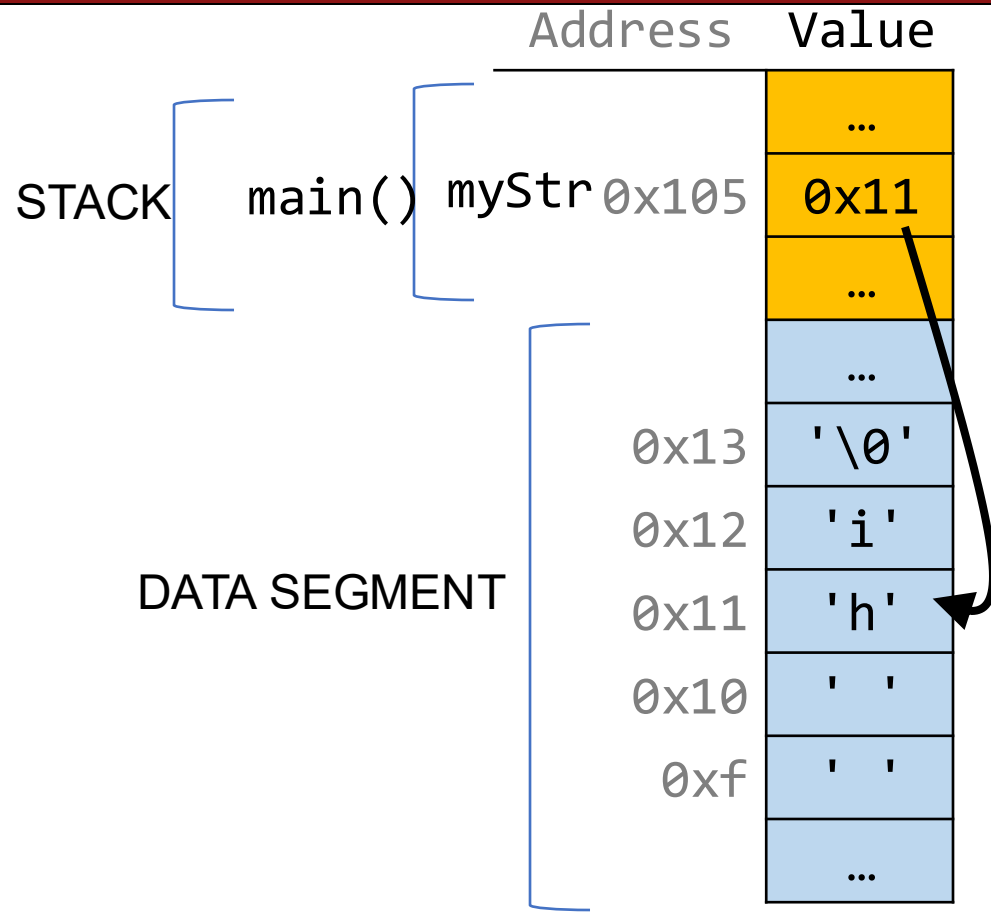
Pointers to Strings

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

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



Pointers to Strings

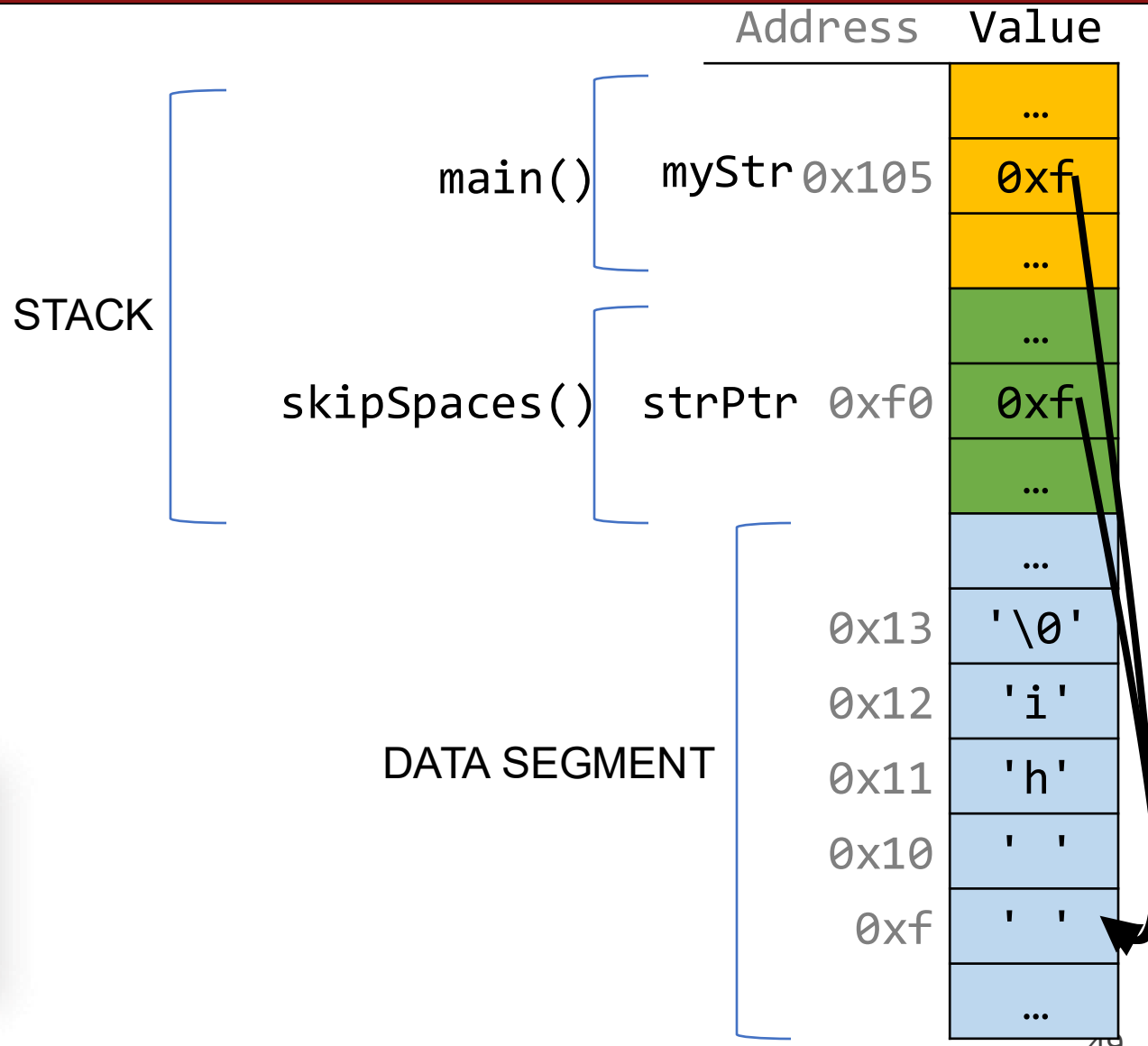
```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

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



Weird thought – **0x11** *is a string*.

Making Copies

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

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



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

Lecture Plan

- **Review:** Pointers
- Double Pointers
- **Pointer Arithmetic**
- Arrays in Memory

```
cp -r /afs/ir/class/cs107/lecture-code/lect9 .
```

Pointer Arithmetic

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

```
char *str = "apple";     // e.g. 0xff0
char *str1 = str + 1;    // e.g. 0xff1
char *str3 = str + 3;    // e.g. 0xff3

printf("%s", str);       // apple
printf("%s", str1);      // pple
printf("%s", str3);      // le
```

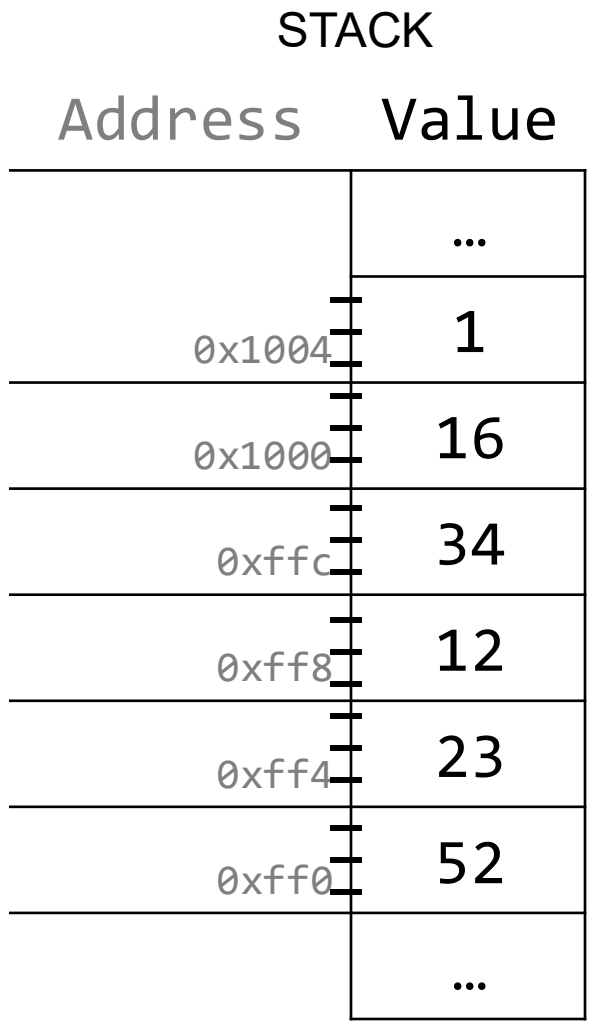
DATA SEGMENT

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

Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

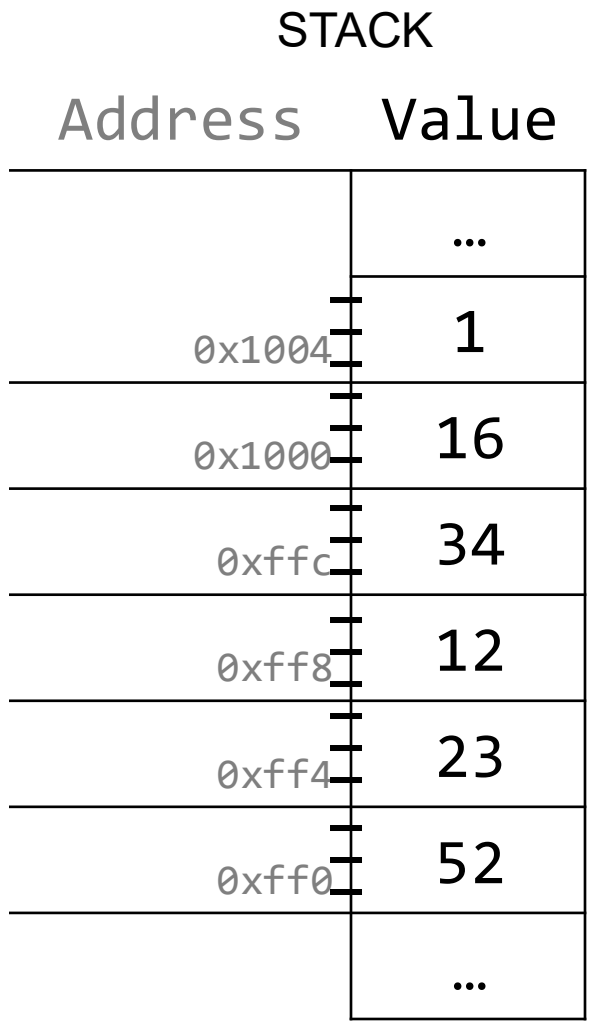
```
// nums points to an int array
int *nums = …                 // e.g. 0xff0
int *nums1 = nums + 1;        // e.g. 0xff4
int *nums3 = nums + 3;        // e.g. 0xffc

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



Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

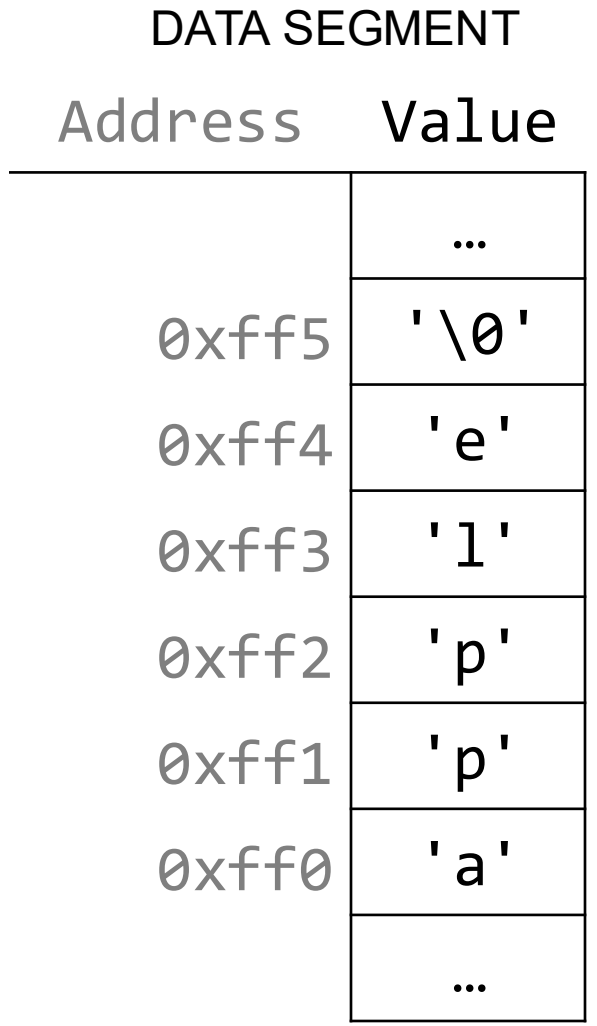
```
// nums points to an int array
int *nums = …                 // e.g. 0xff0
int *nums3 = nums + 3;  // e.g. 0xffc
int *nums2 = nums3 - 1; // e.g. 0xff8

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



Pointer Arithmetic

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

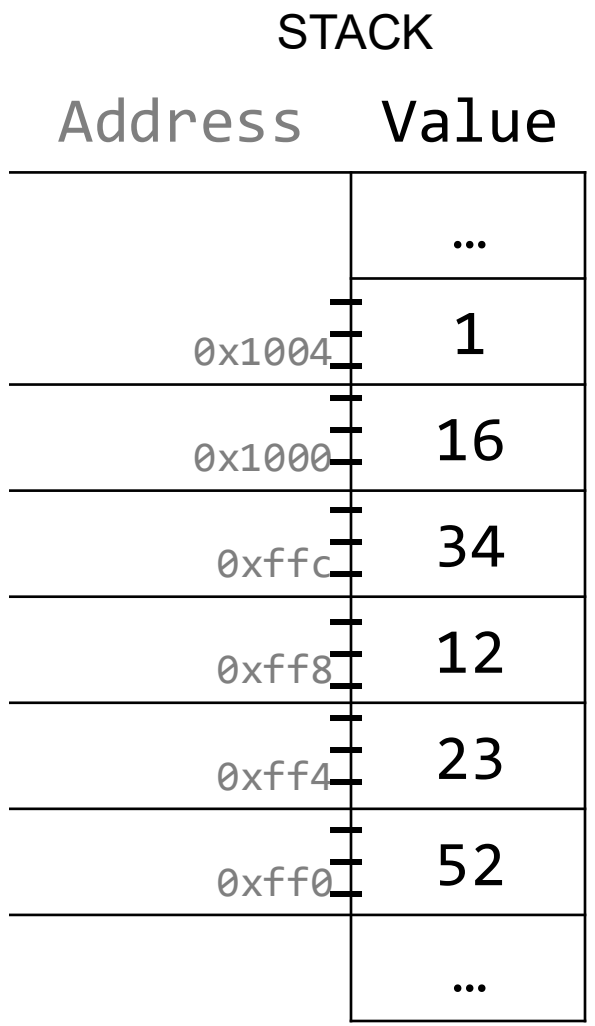
```
char *str = "apple";    // e.g. 0xff0

// 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'
```



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? At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.

```
int x = 2;
int *xPtr = &x;          // e.g. 0xff0

// C knows 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? At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.

```
int nums[] = {1, 2, 3};

// C knows to add 4 bytes here
int *intPtr = nums + 1;

char str[6];
strcpy(str, "CS107");

// C knows to add 1 byte here
char *charPtr = str + 1;
```

Lecture Plan

- **Review:** Pointers
- Double Pointers
- Pointer Arithmetic
- **Arrays in Memory**

```
cp -r /afs/ir/class/cs107/lecture-code/lect9 .
```

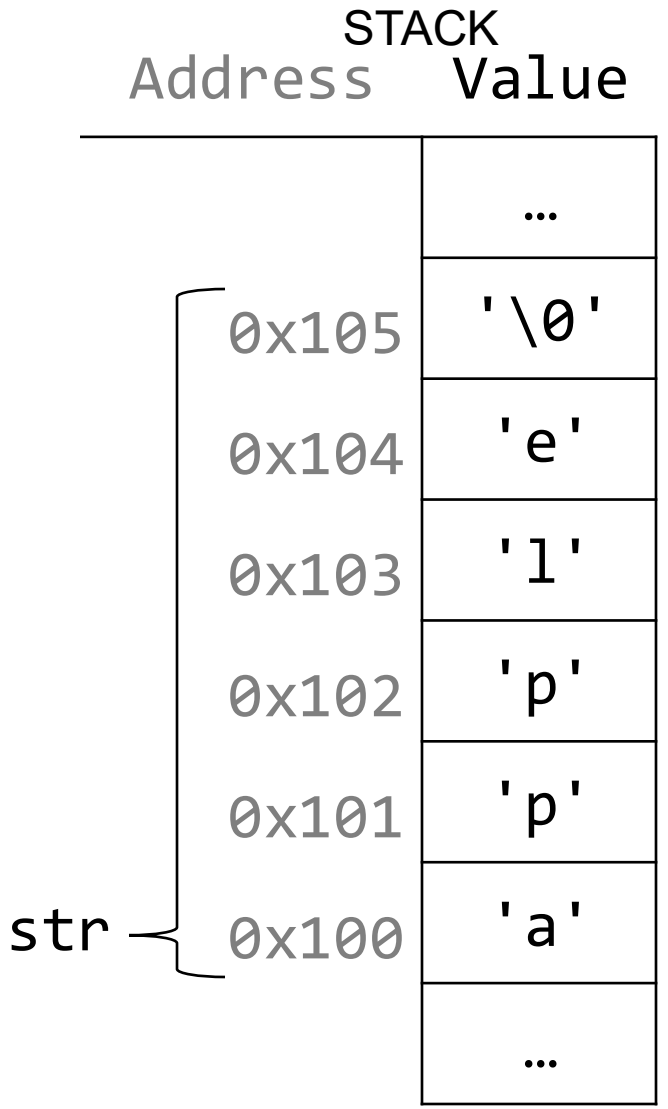
Arrays

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

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

The array variable (e.g. **str**) is not a pointer; it refers to the entire array contents. In fact, **sizeof** returns the size of the entire array!

```
int arrayBytes = sizeof(str);    // 6
```



Arrays

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

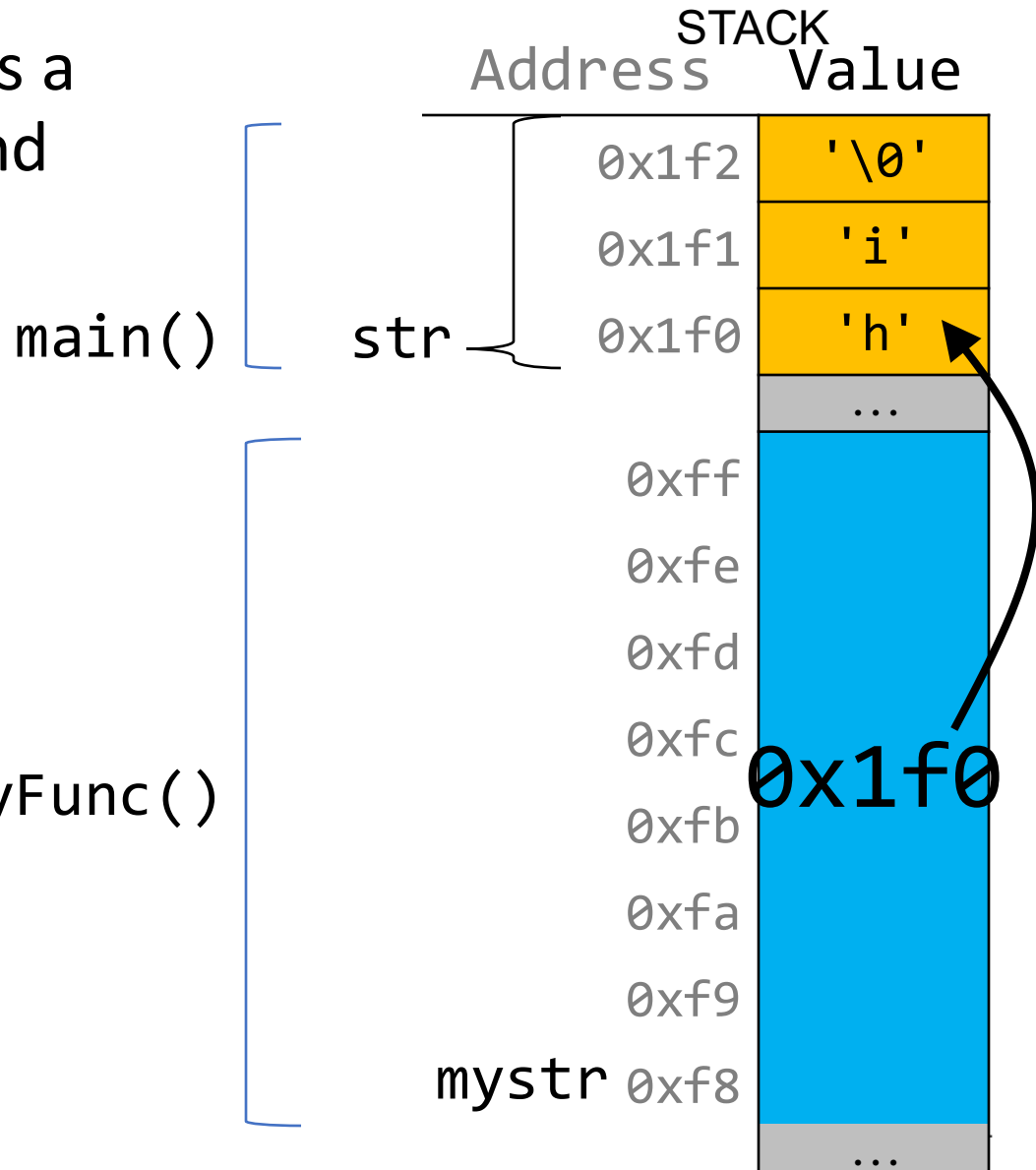
```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

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

Arrays as Parameters

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

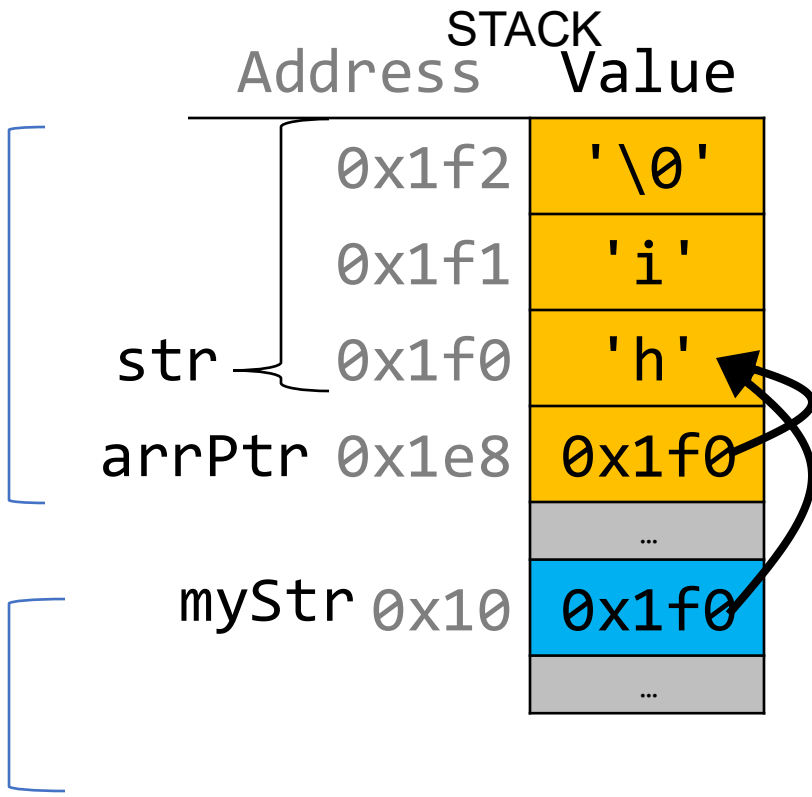
```
void myFunc(char *myStr) {
      ...
}

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



Arrays as Parameters

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

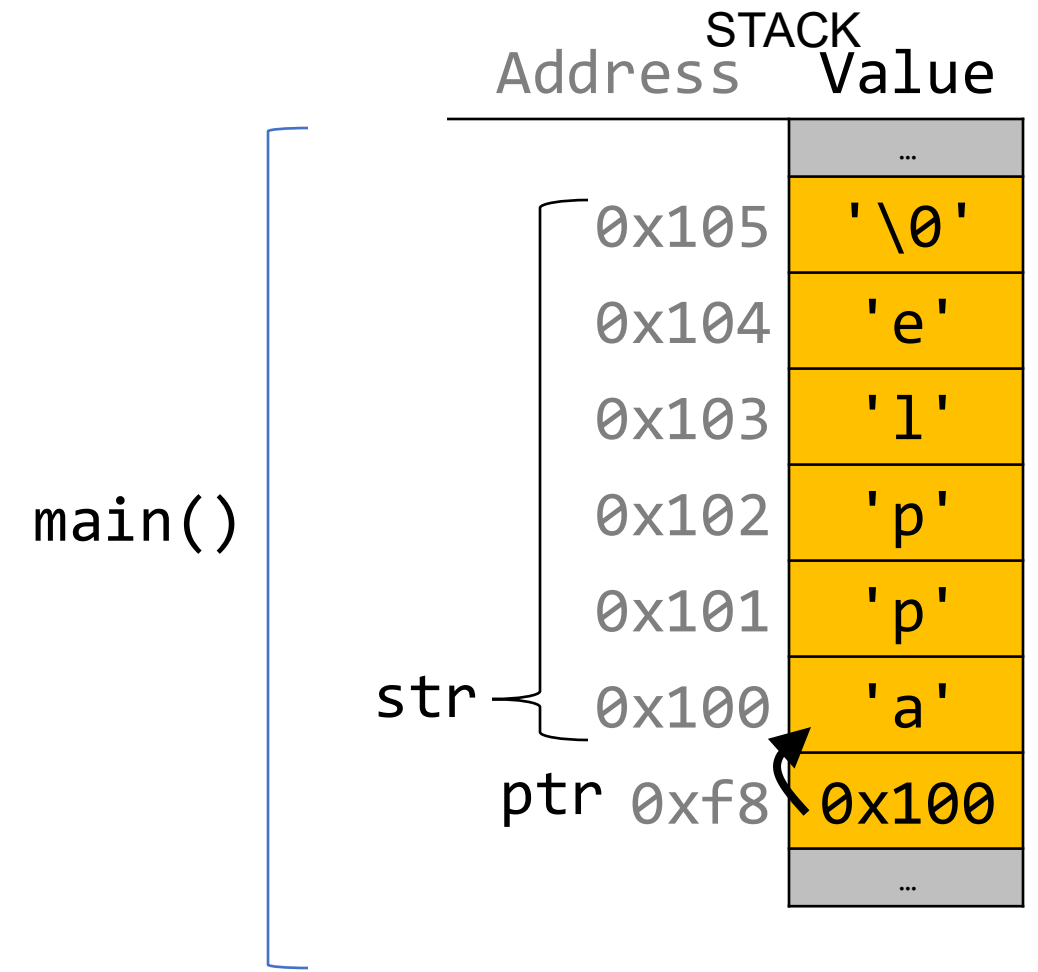
```
void myFunc(char *myStr) {
      …
}

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



Arrays and Pointers

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

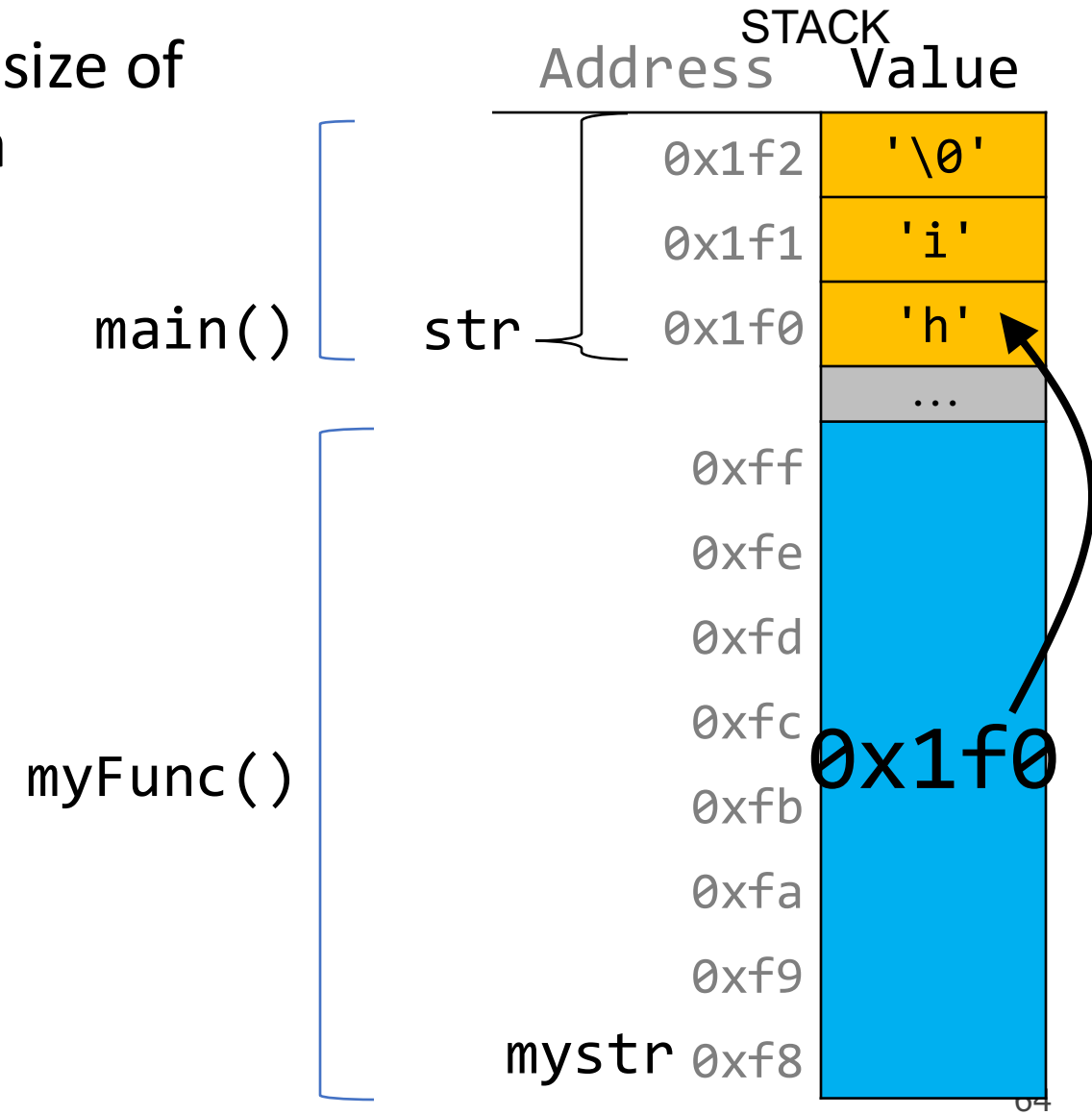
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;

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

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



Arrays as Parameters

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

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

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



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

Arrays Summary

- When you create an array, you are making space for each element in the array.
- When you create a pointer, you are making space for an 8 byte address.
- Arrays ”decay to pointers” when you perform arithmetic or pass as parameters.
- You can set a pointer equal to an array; that pointer will point to the array’s first element
- &arr does nothing on arrays, but &ptr on pointers gets its address
- sizeof(arr) gets the size of an array in bytes, but sizeof(ptr) is always 8
- Only strings have null terminating characters at the end – other arrays do not

Recap

- **Review**: Pointers
- Double Pointers
- Pointer Arithmetic
- Arrays in Memory

Lecture 9 takeaway: pointers let us store the addresses of data and pass them as parameters. We can use double pointers if we want to change the value of a pointer in another function.

Next time: stack vs. heap

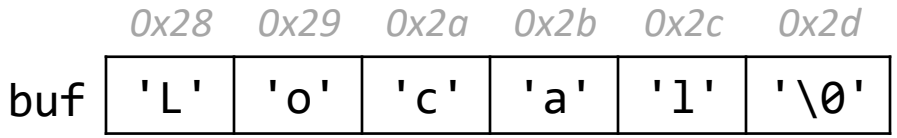
3. Bonus: Tricky addresses

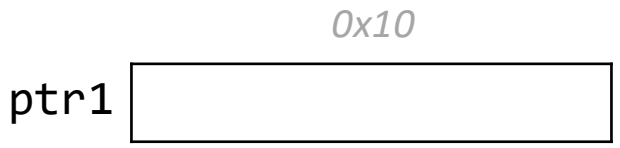
```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);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

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

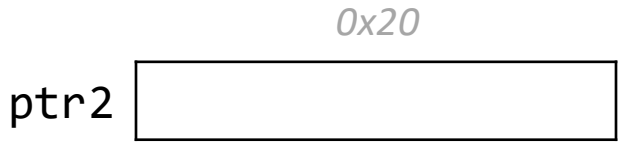
What is stored in each variable?

3. Bonus: Tricky addresses

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

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



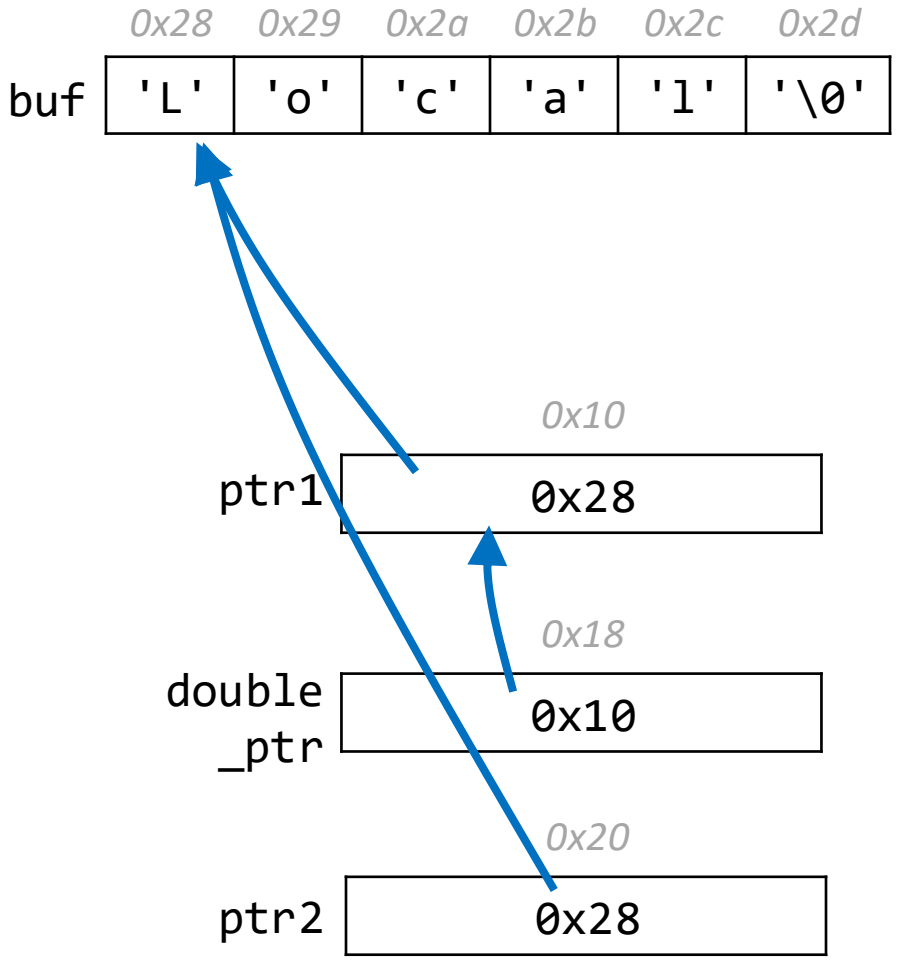




While Line 10 raises a compiler warning, functionally it will still work—because pointers are **addresses**.

3. Bonus: Tricky addresses

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

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

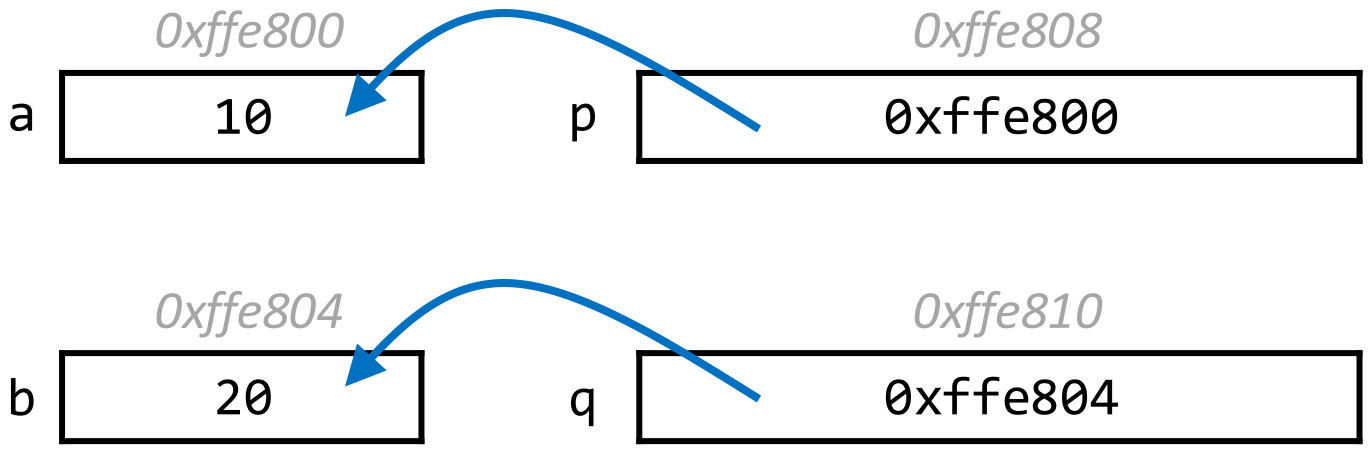


While Line 10 raises a compiler warning, functionally it will still work—because pointers are **addresses**.

Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

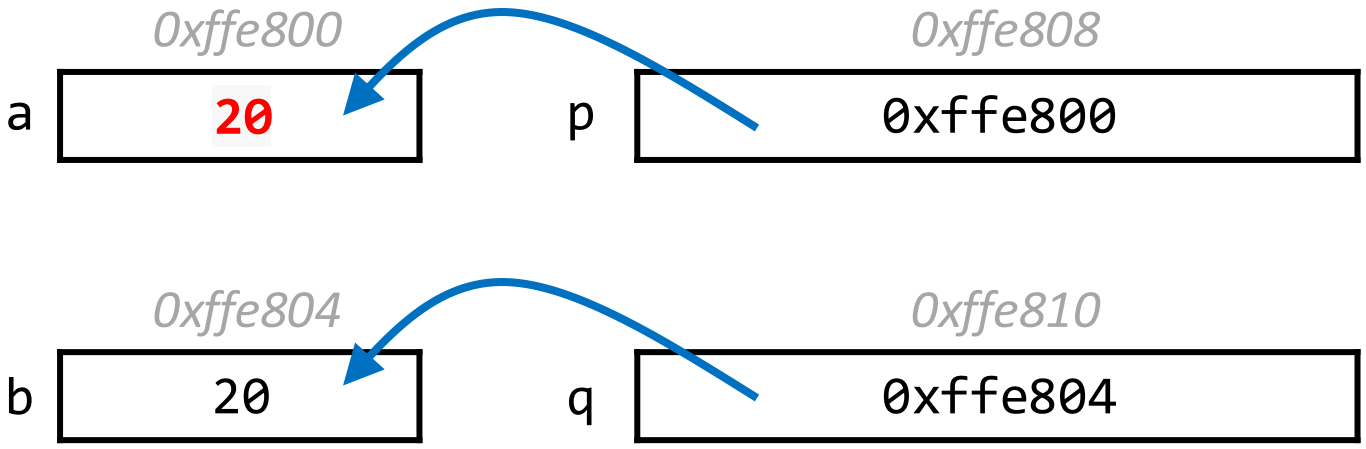
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

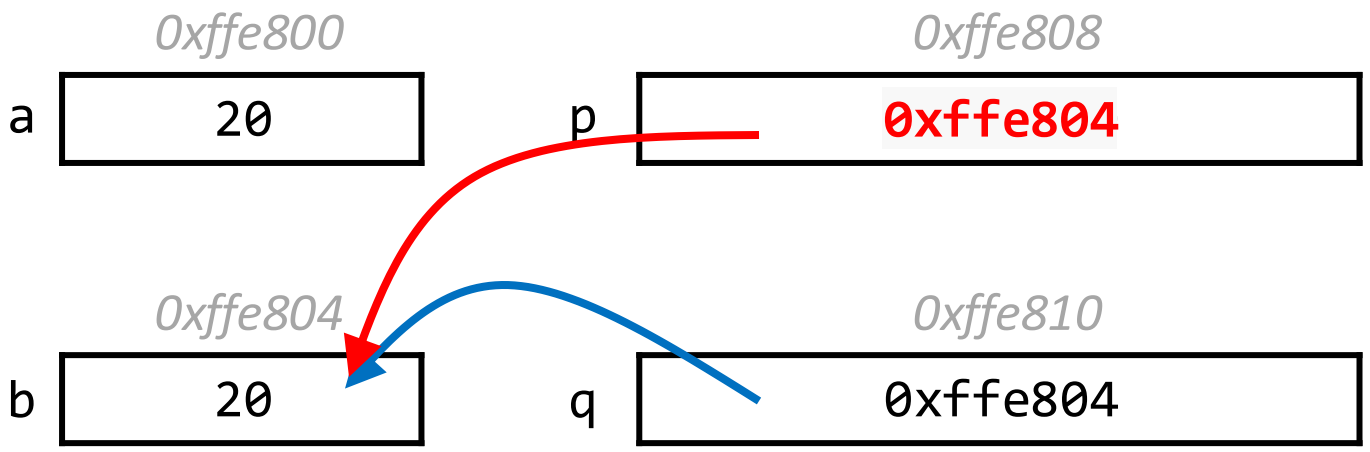
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



* Wars: Episode I (of 2)

In variable **declaration**, * creates a **pointer**.

```
char ch = 'r';
```

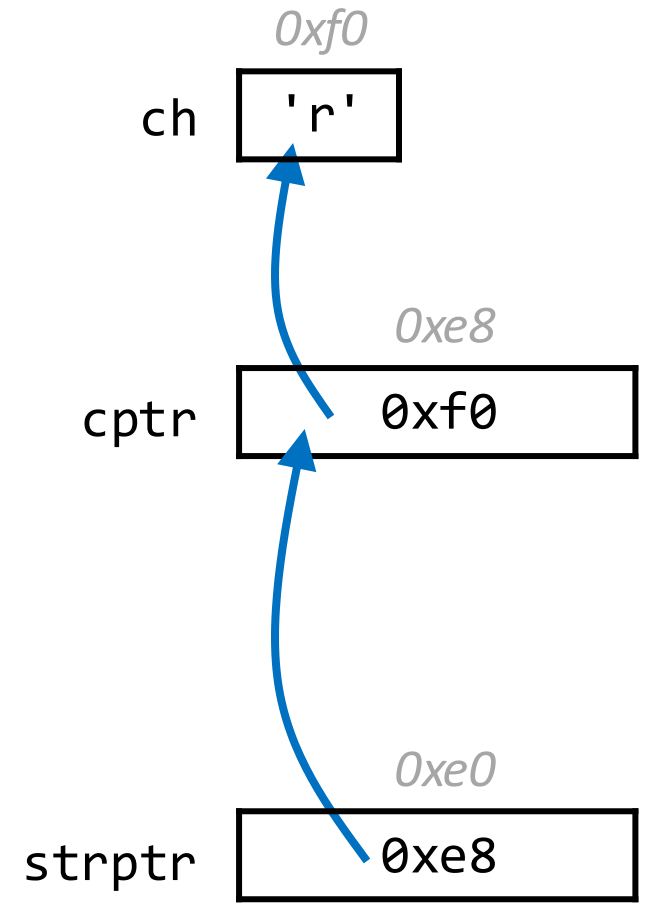
ch stores a char

```
char *cptr = &ch;
```

cptr stores an address of a char
(**points to** a char)

```
char **strptr = &cptr;
```

strptr stores an address of a char *
(**points to** a char *)



* Wars: Episode II (of 2)

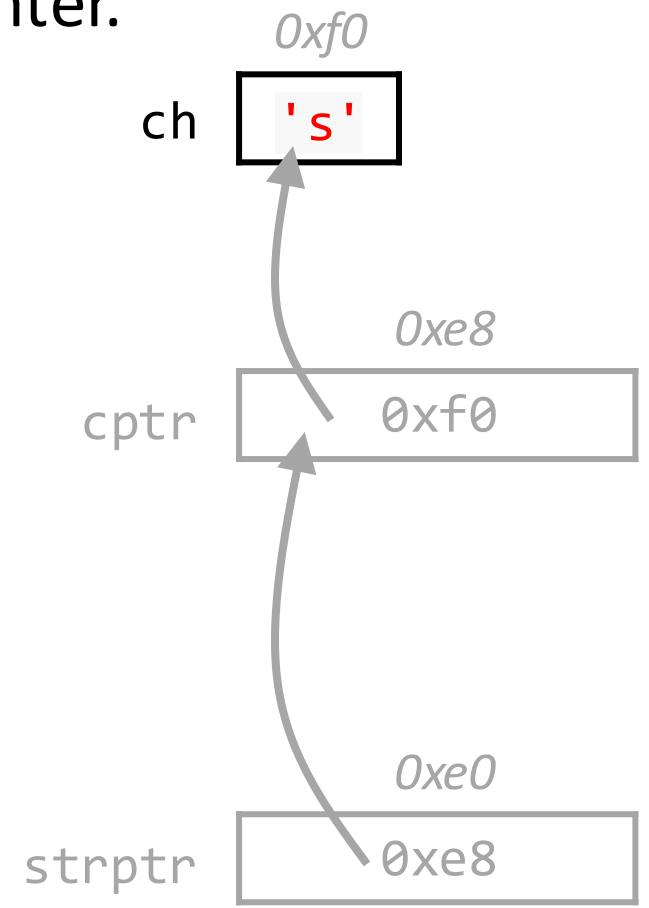
In **reading values from/storing values**, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in ch

```
char *cptr = &ch;
```

```
char **strptr = &cptr;
```



* Wars: Episode II (of 2)

In **reading values from/storing values**, * **dereferences** a pointer.

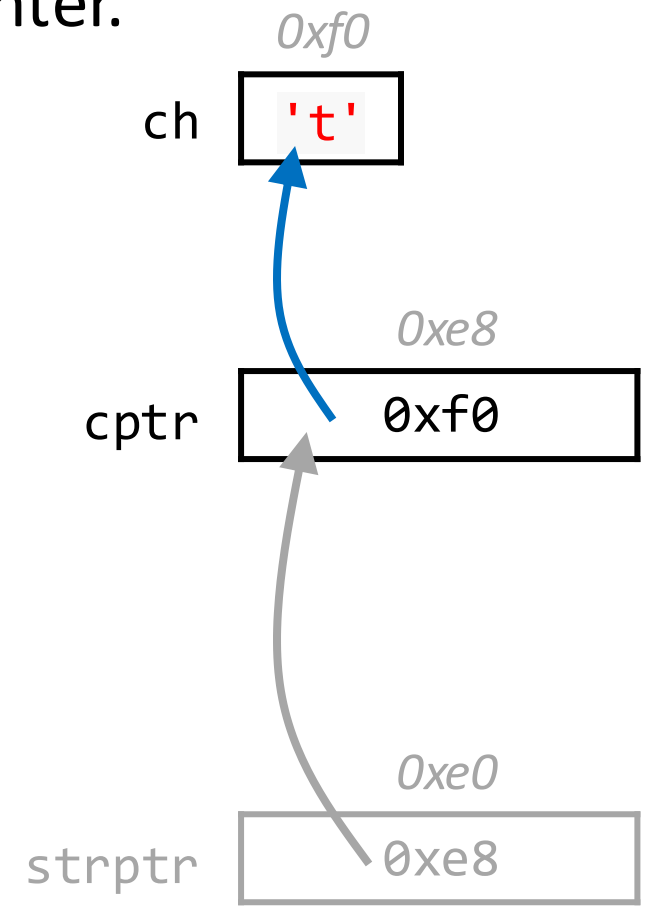
```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in ch

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in cptr (increment char **pointed to**)

```
char **strptr = &cptr;
```



* Wars: Episode II (of 2)

In **reading values from/storing values**, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

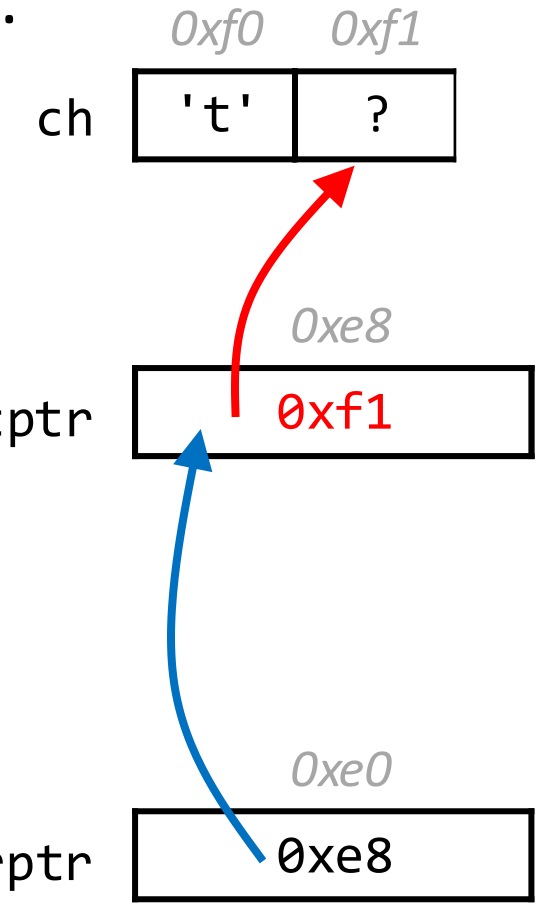
Increment value stored in ch

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in cptr (increment char **pointed to**)

```
char **strptr = &cptr;
*strptr = *strptr + 1;
```

Increment value stored at memory address in cptr (increment address **pointed to**)

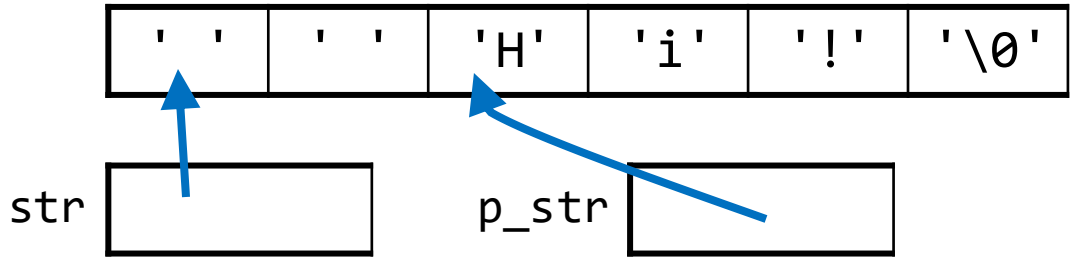


Skip spaces

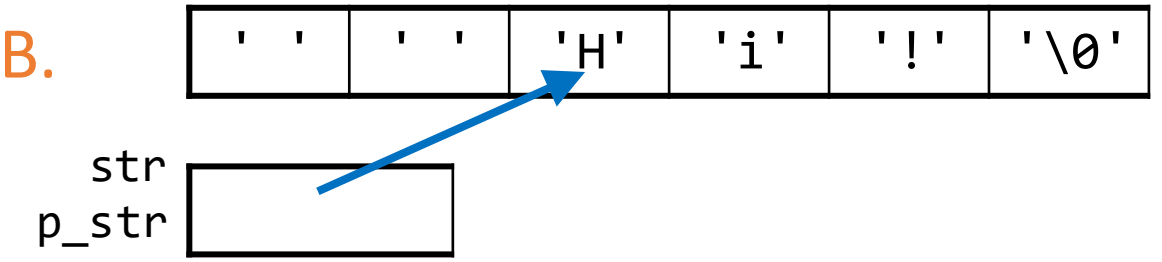
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

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

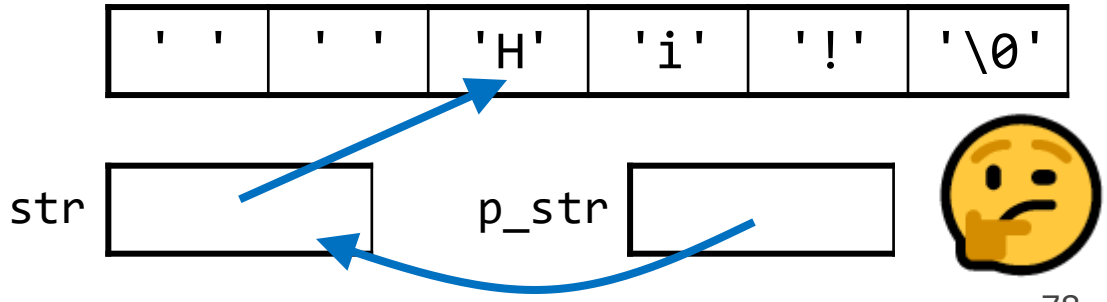
A.



B.



C.

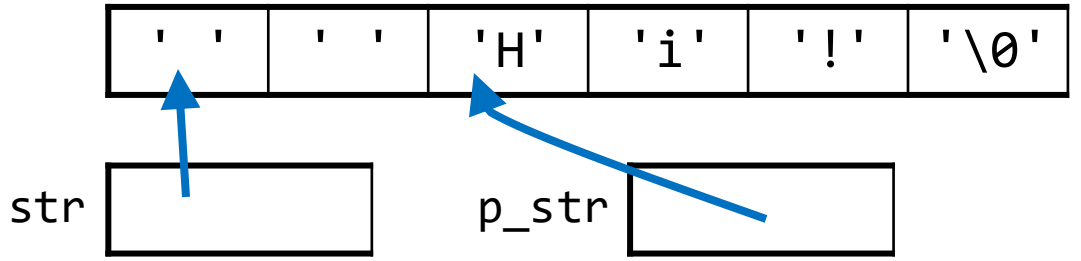


Skip spaces

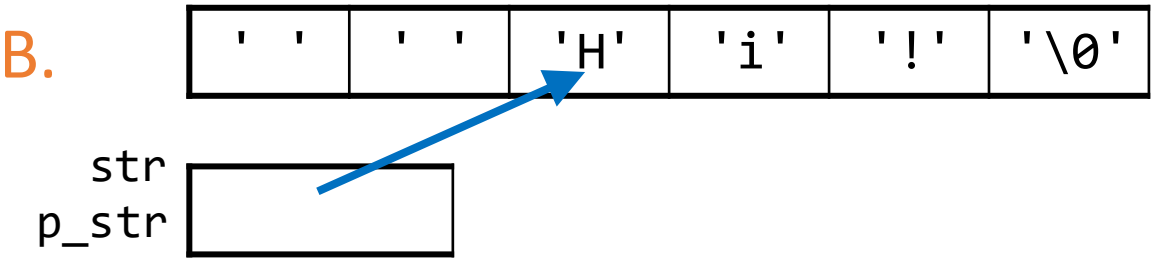
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

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

A.



B.



C.