CS107, Lecture 9
Pointers and Arrays

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

Announcement: CS198 Section Leading

CS198 Slides

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

- **Finishing up:** Strings and Pointers
- Double Pointers

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

Lecture Plan

- **Finishing up: Strings and Pointers**
- Double Pointers

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

C Parameters

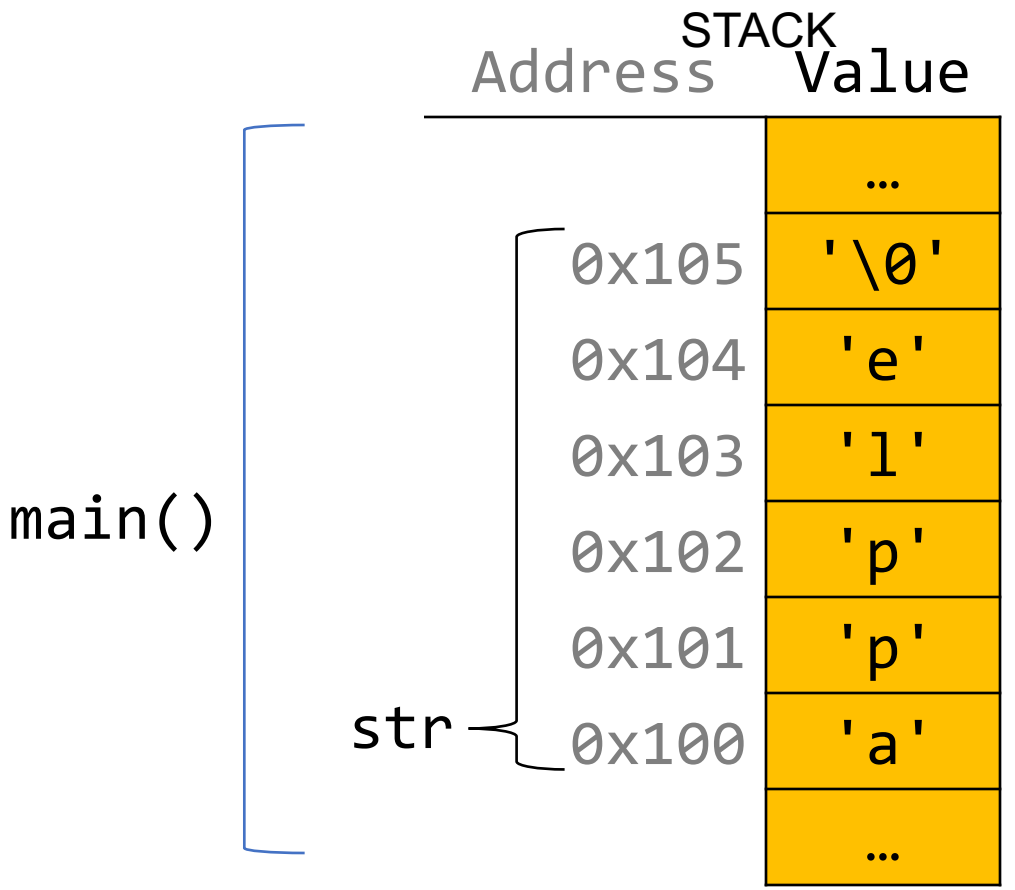
- If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.
- If you are modifying a specific instance of some value, pass the *location* of what you would like to modify and dereference that location to access what's there.
- If a function 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.

char[]

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

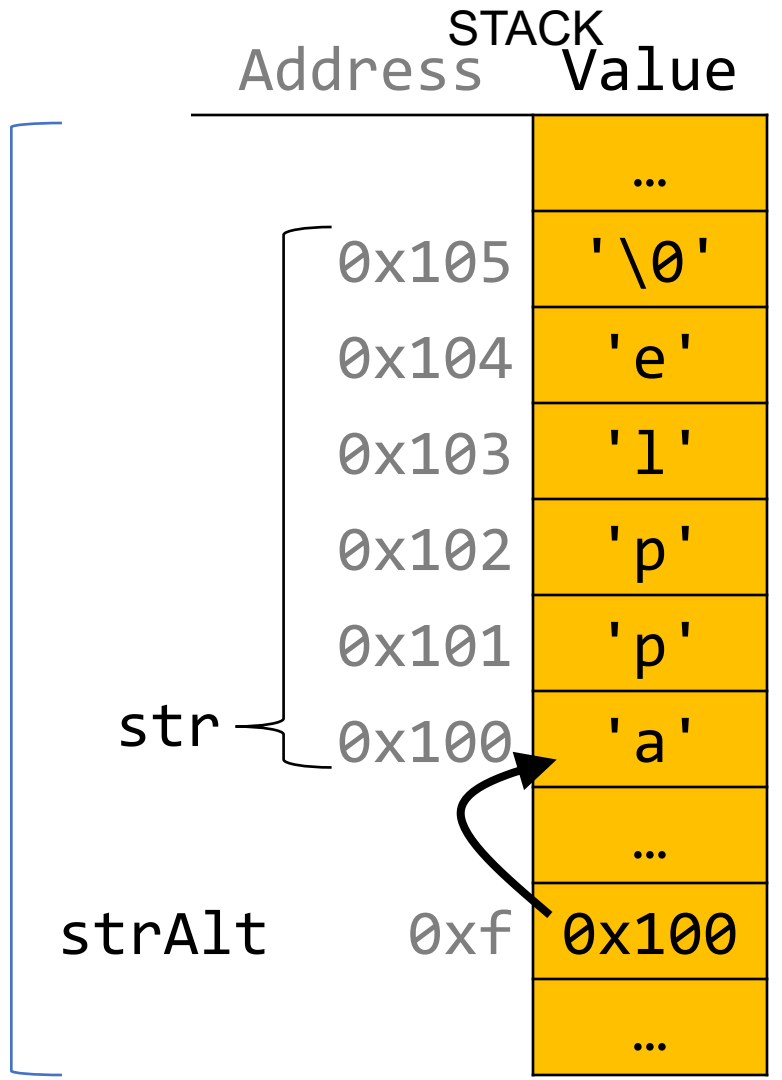
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      ...
```



char *

When we declare a **char** *, we allocate space on the stack to store an address, not actual characters. But we can still generally use **char** * the same as **char[]**.

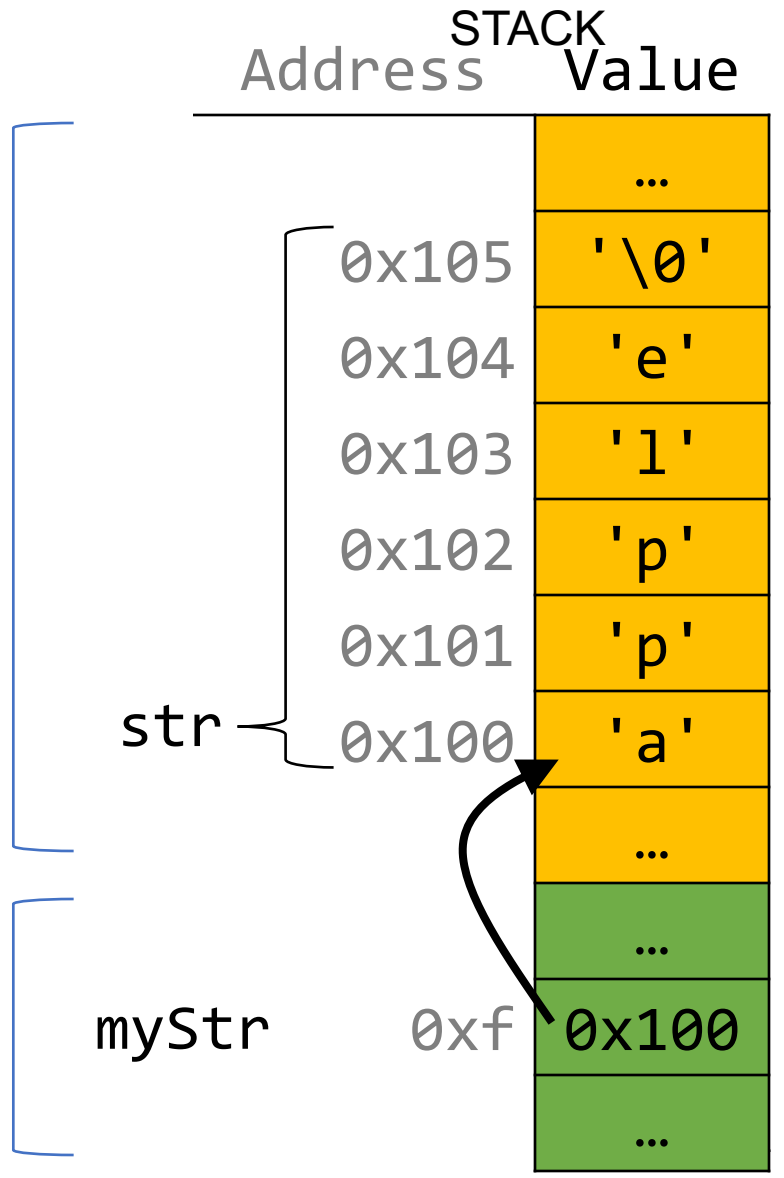
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *strAlt = str;
      ...
```



Strings as Parameters

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

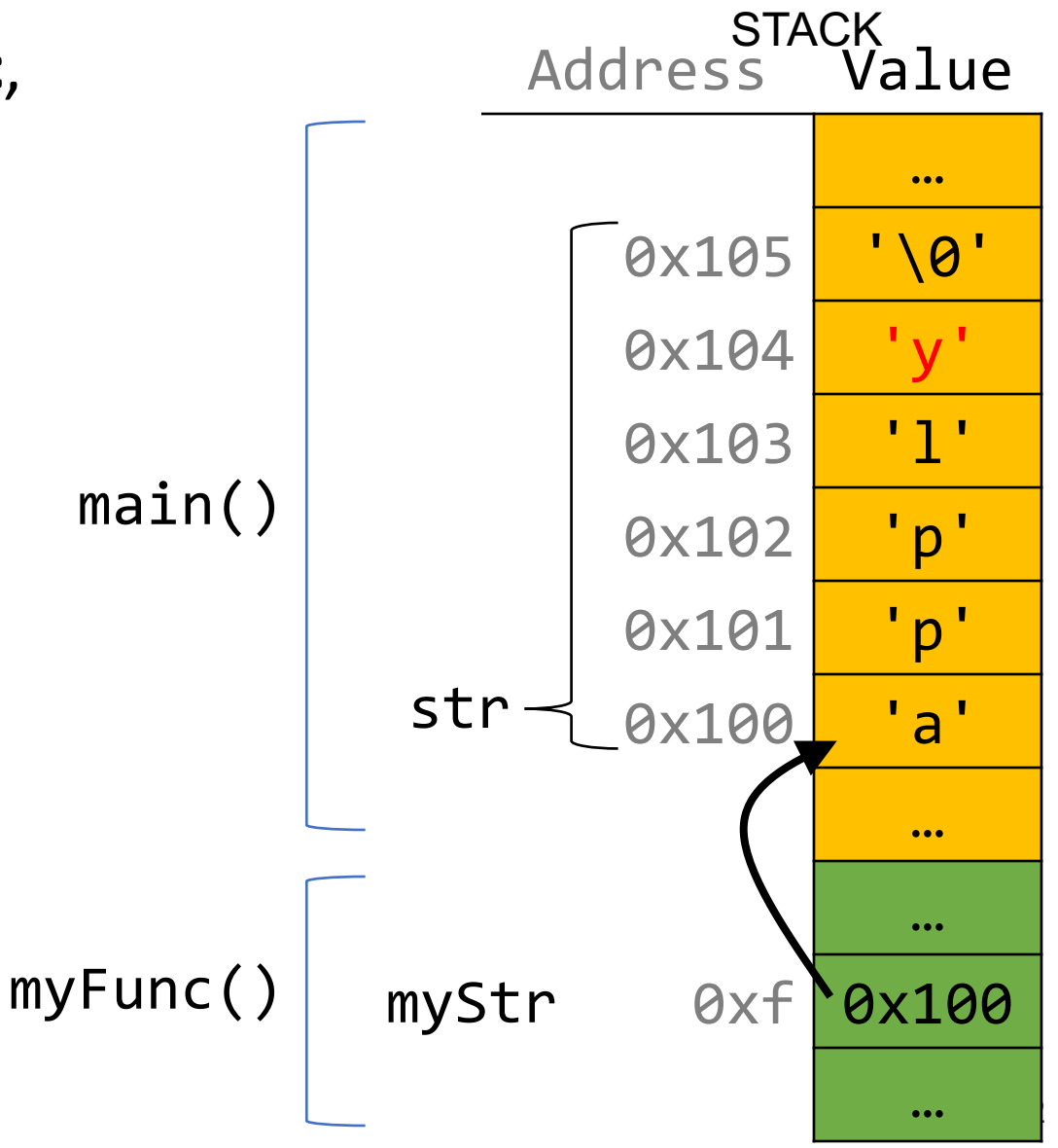
```
void myFunc(char *myStr) {
      ...
}

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



Strings as Parameters

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

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

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



char *

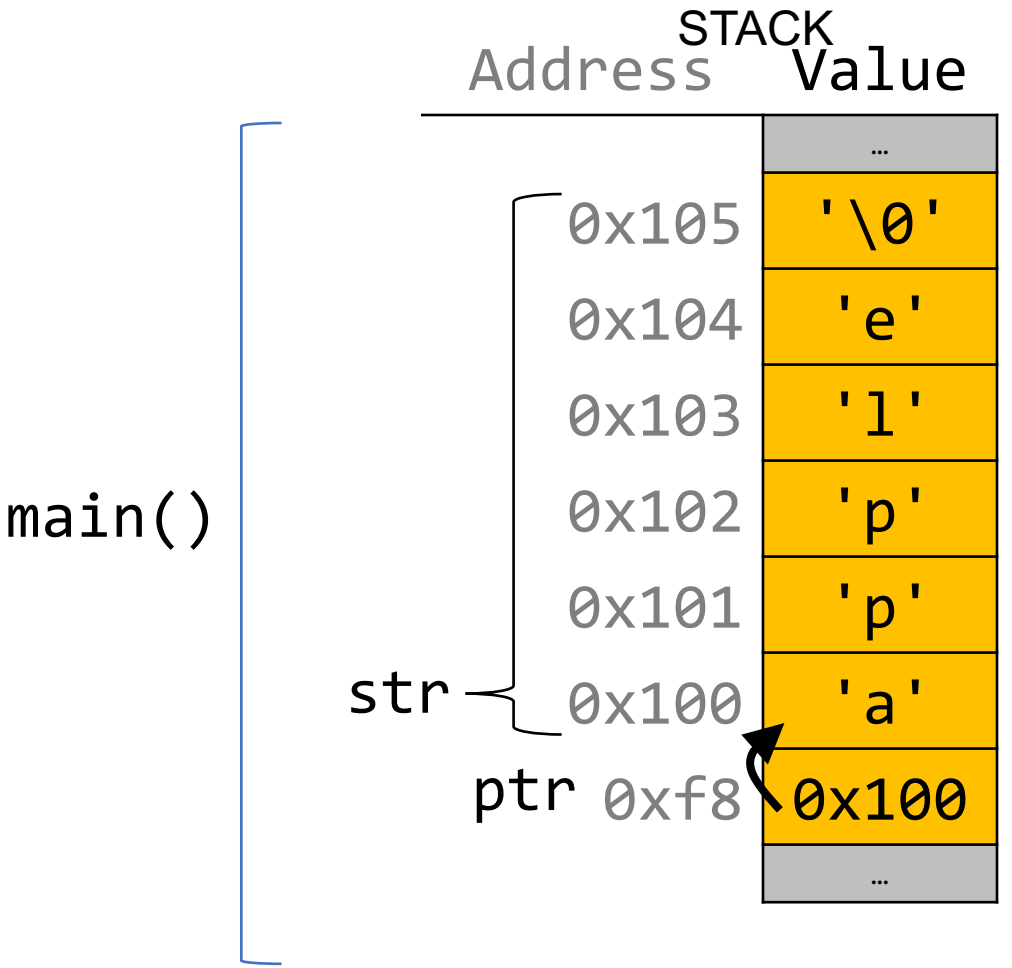
A **char *** variable refers to a single character. We can reassign an existing **char *** pointer to be equal to another **char *** pointer.

```
char *str = "apple";          // e.g. 0xfff0
char *str2 = "apple 2";       // e.g. 0xfe0
str = str2;   // ok!  Both store address 0xfe0
```

Arrays and Pointers

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

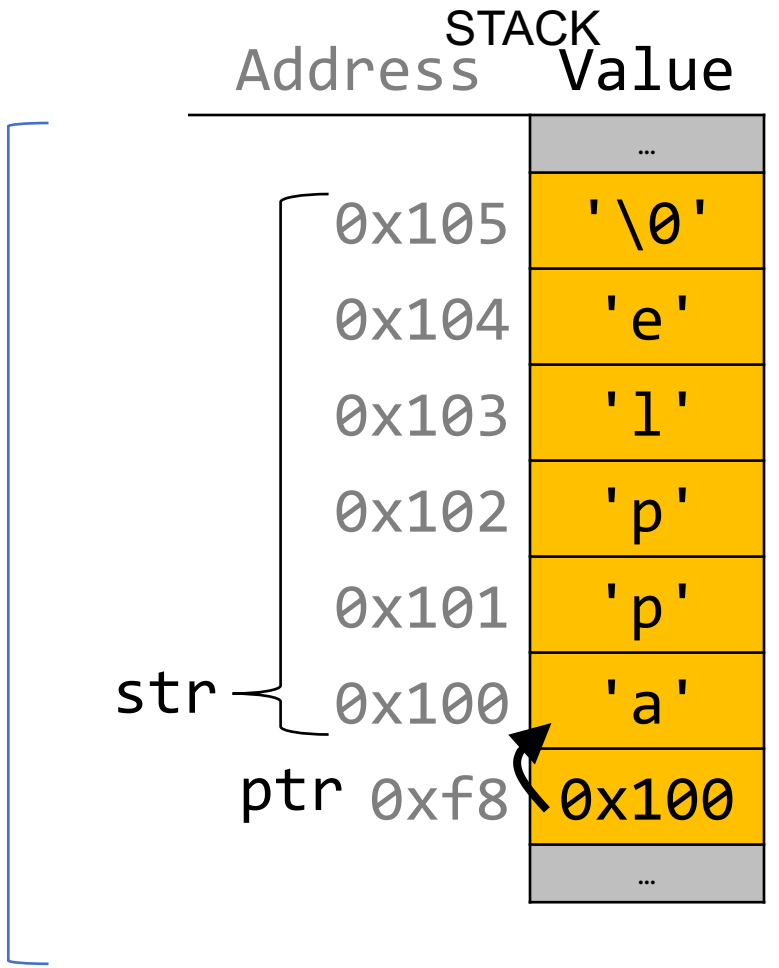
```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;
      ...
}
```



Arrays and Pointers

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

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

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

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



Read-only Strings

There is another convenient way to create a string if we do not need to modify it later. We can create a `char *` and set it directly equal to a string literal.

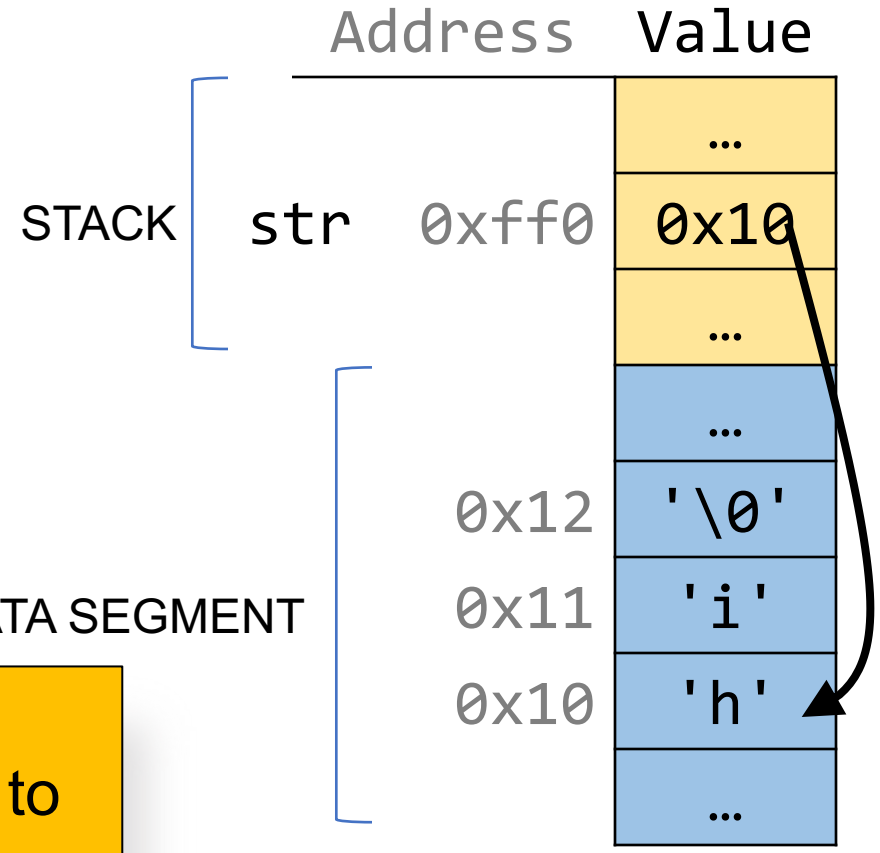
```
char *myString = "Hello, world!";
...
printf("%s", myString);                    // Hello, world!
```

Read-only Strings

When we declare a char pointer equal to a string literal, the characters are *not* stored on the stack. Instead, they are stored in a special area of memory called the "data segment". We *cannot modify memory in this segment*.

```
char *str = "hi";
```

The pointer variable (e.g. **str**) refers to the *address of the first character of the string in the data segment*.



NOTE: not all char * strings are read-only. Only ones that point to characters in the data segment are read-only.

Read-only Strings

Read-only strings are convenient to use, but make sure to not use a read-only string in code that tries to modify its characters – it will crash!

```
char *myString = "Hello, world!";
myString[0] = 'h';  // crashes!
```

There's no way in code to check if a string is read-only; it's up to the programmer to properly use strings to avoid these crashes.

- E.g. don't pass in a read-only string as the **dst** to **strcpy**

```
strcpy(myString, "Hi");  // crashes!
```

Read-only Strings

A string is read-only if it points to characters that live in the data segment, rather than memory we can modify.

```
char *readOnly = "Hi";

char modifiable[6];
strcpy(modifiable, "Hi");

// is ptr read-only?
char *ptr = modifiable;
// no, because it points to characters on the stack
ptr[0] = 'h';  // ok!
```

Strings In Memory

1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.
4. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
5. If we create a new string with new characters as a **`char *`**, we cannot modify its characters because its memory lives in the data segment.
6. Adding an offset to a C string gives us a substring that many places past the first character.
7. If we change characters in a string parameter, these changes will persist outside of the function.

Lecture Plan

- **Finishing up:** Strings and Pointers
- **Double Pointers**

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

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.

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.

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
or text CS107 to 22333
once to join.

```
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

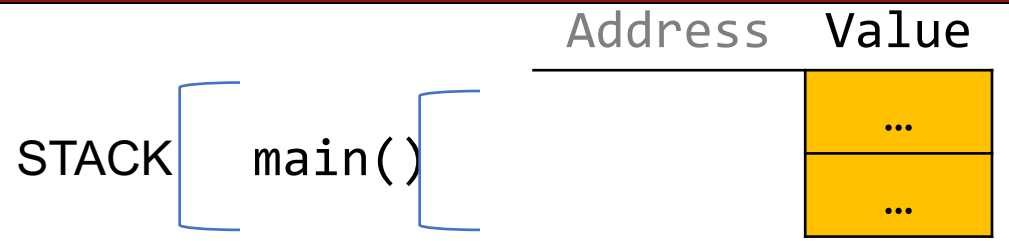
Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function **skipSpaces** that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
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.

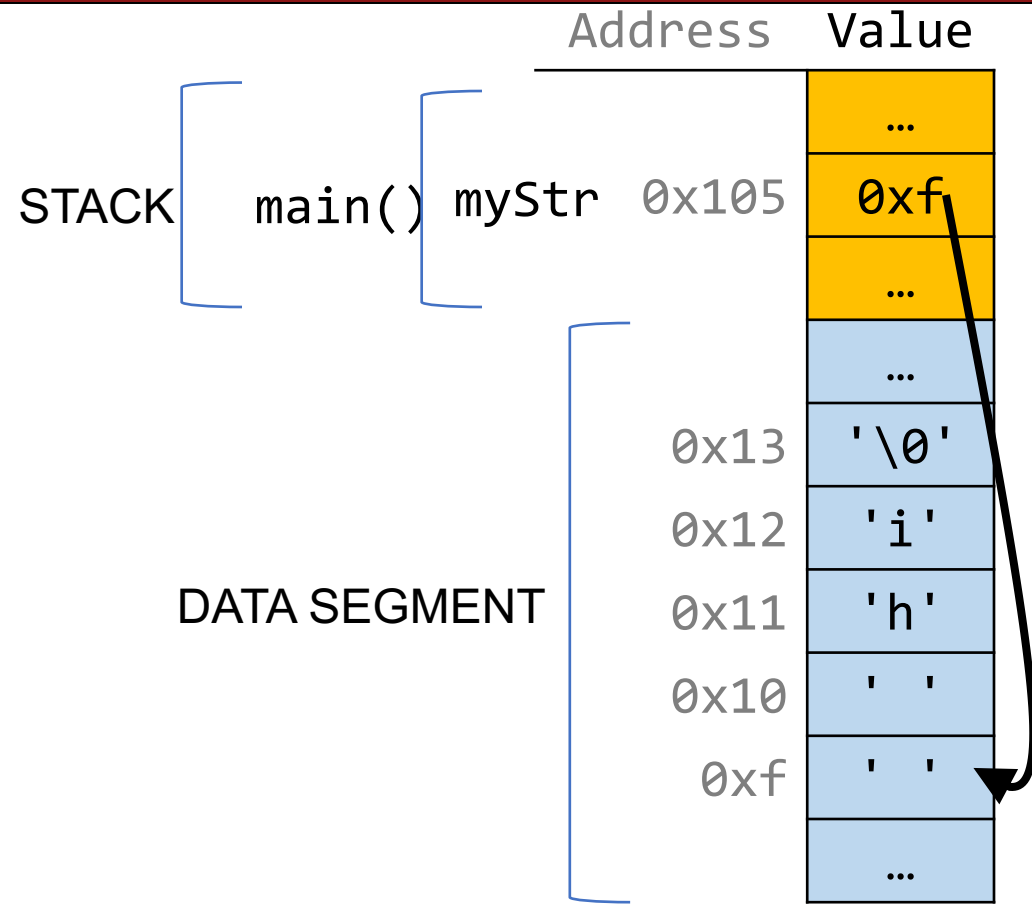
Pointers to Strings

```
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;
}
```



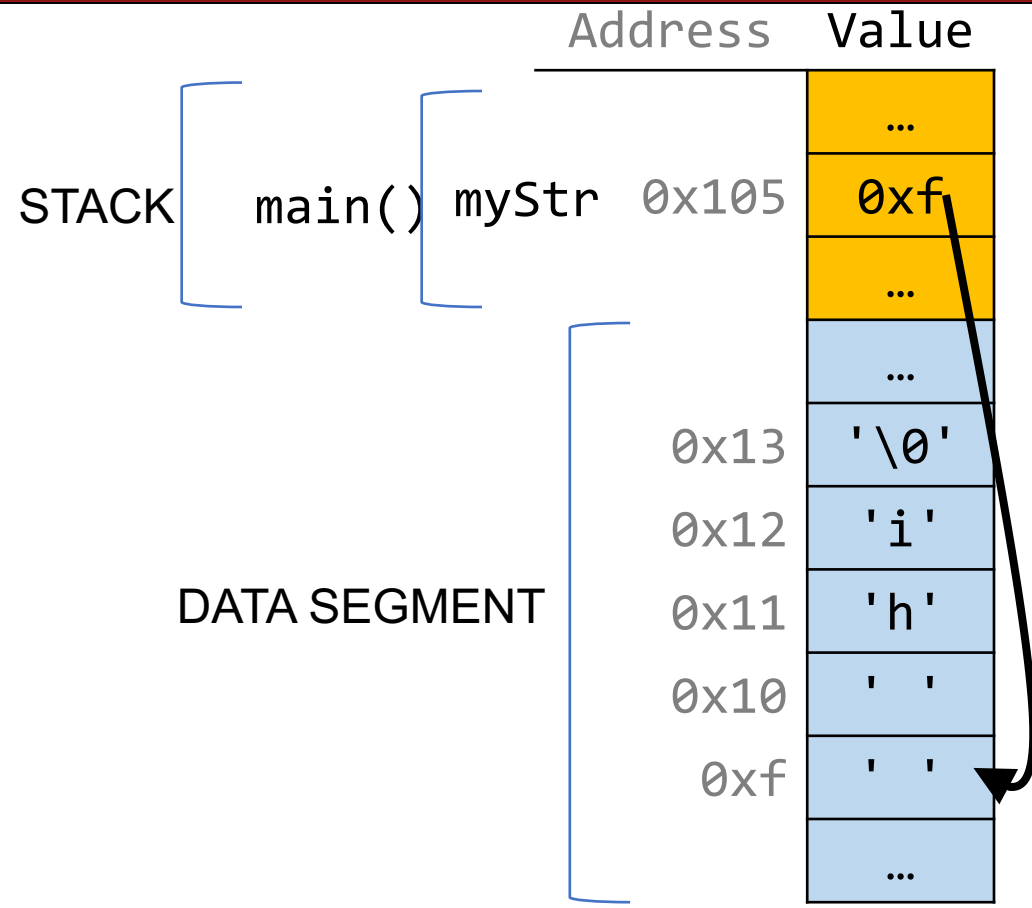
Pointers to Strings

```
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;
}
```



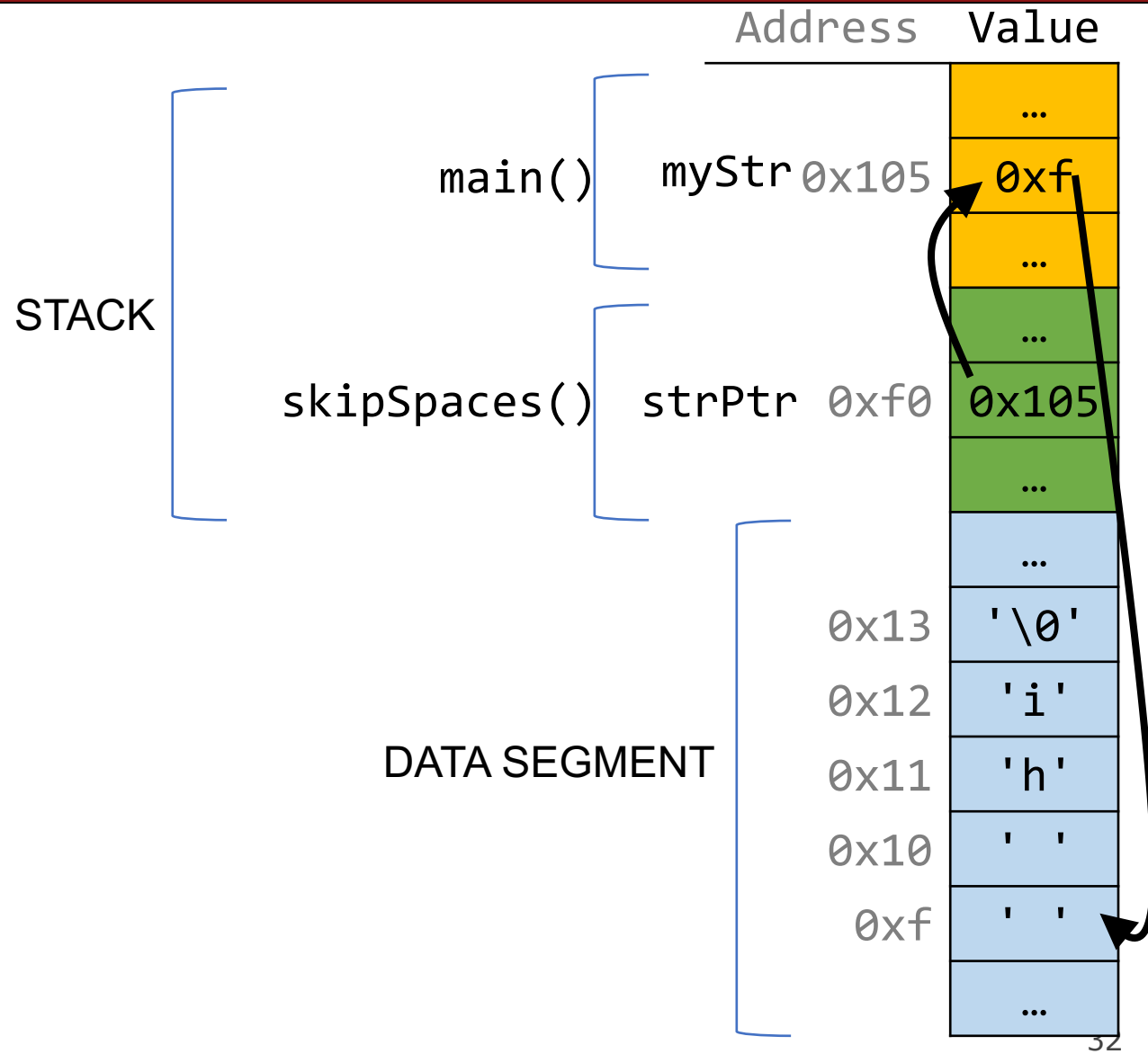
Pointers to Strings

```
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;
}
```



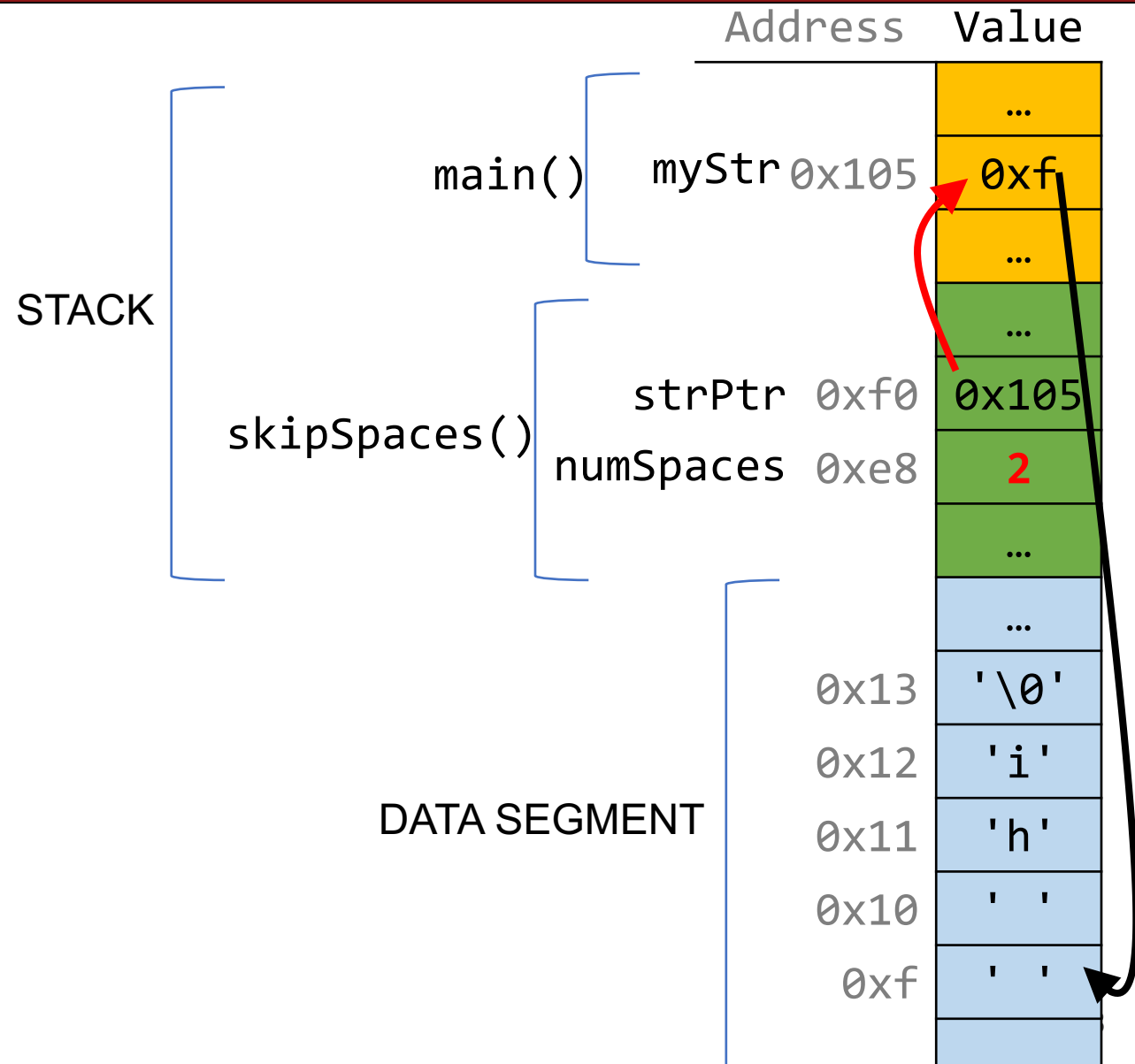
Pointers to Strings

```
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;
}
```



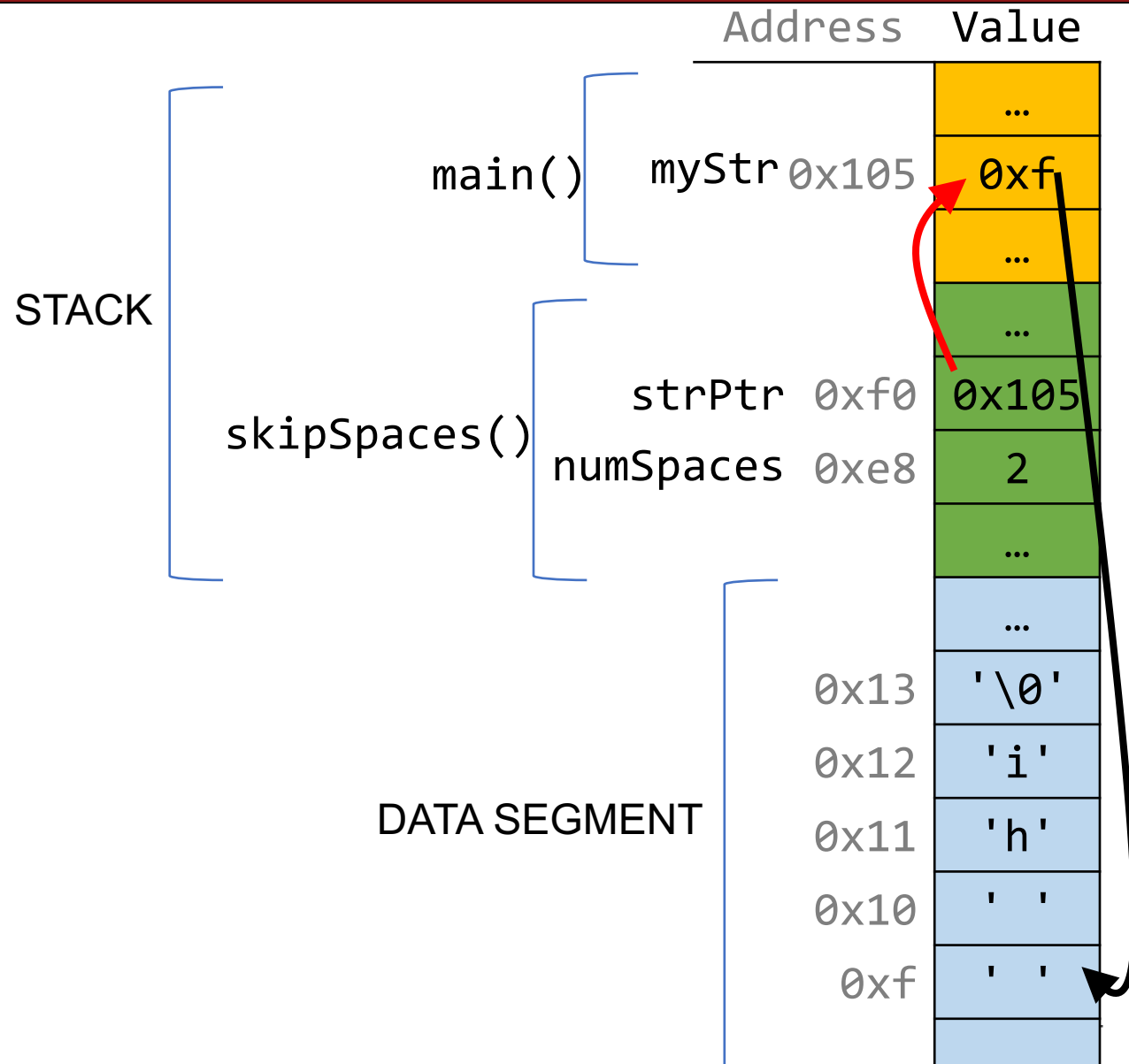
Pointers to Strings

```
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;
}
```



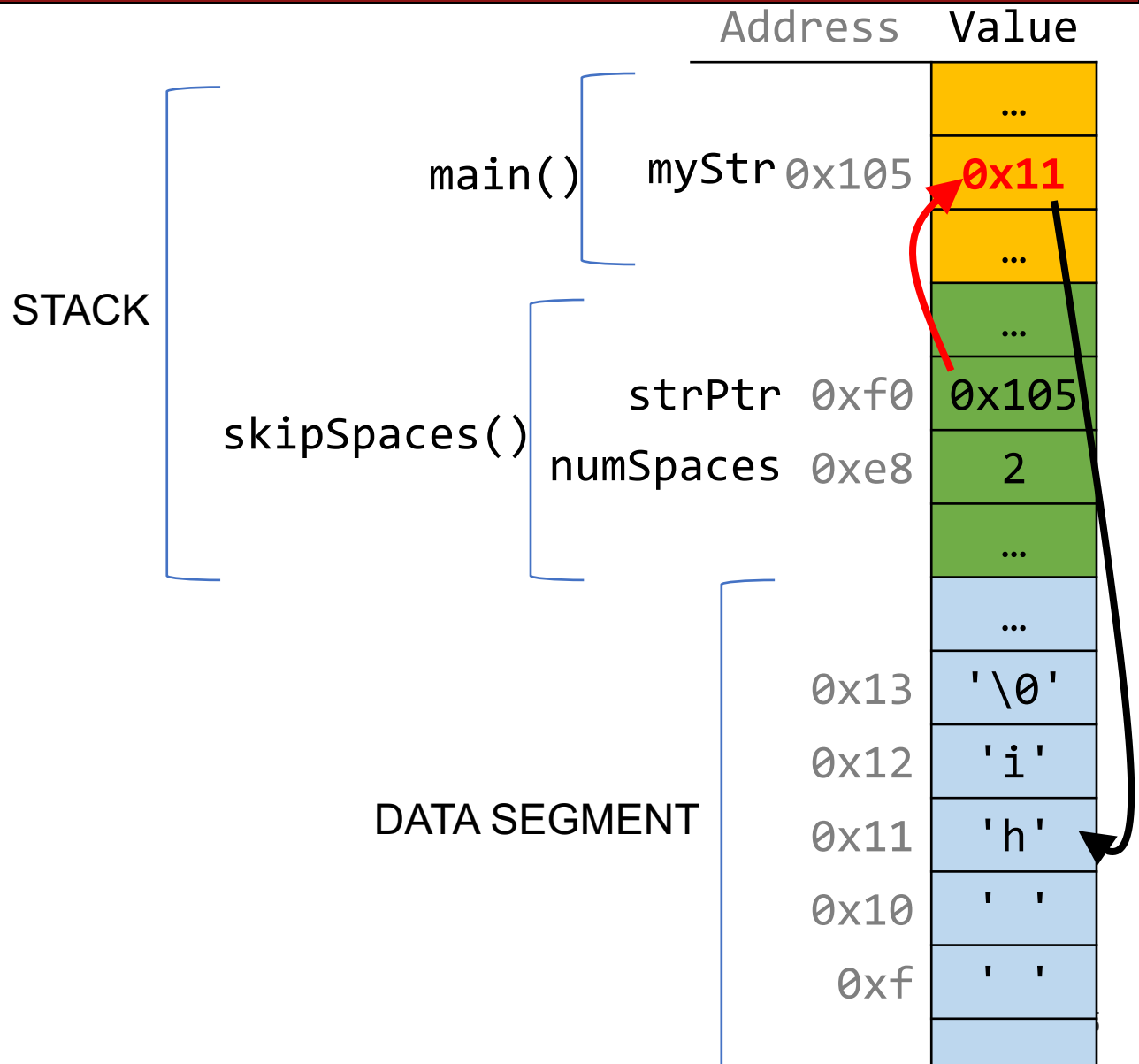
Pointers to Strings

```
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;
}
```



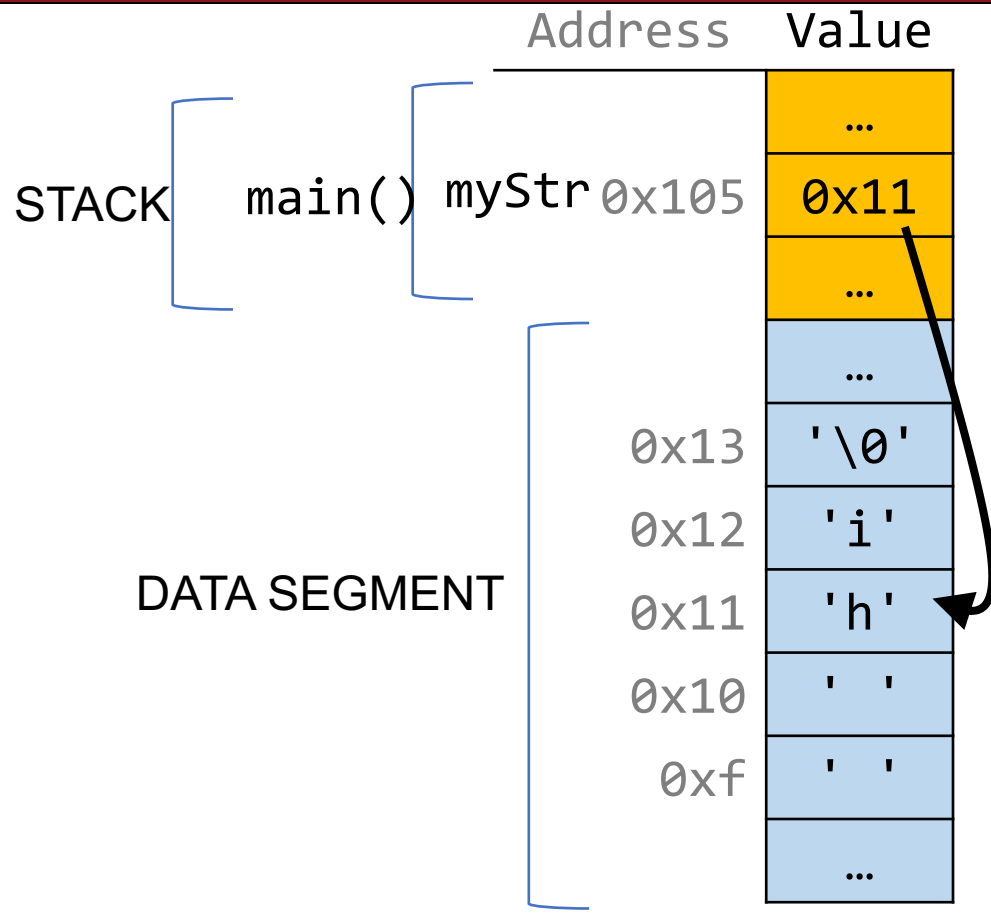
Pointers to Strings

```
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;
}
```



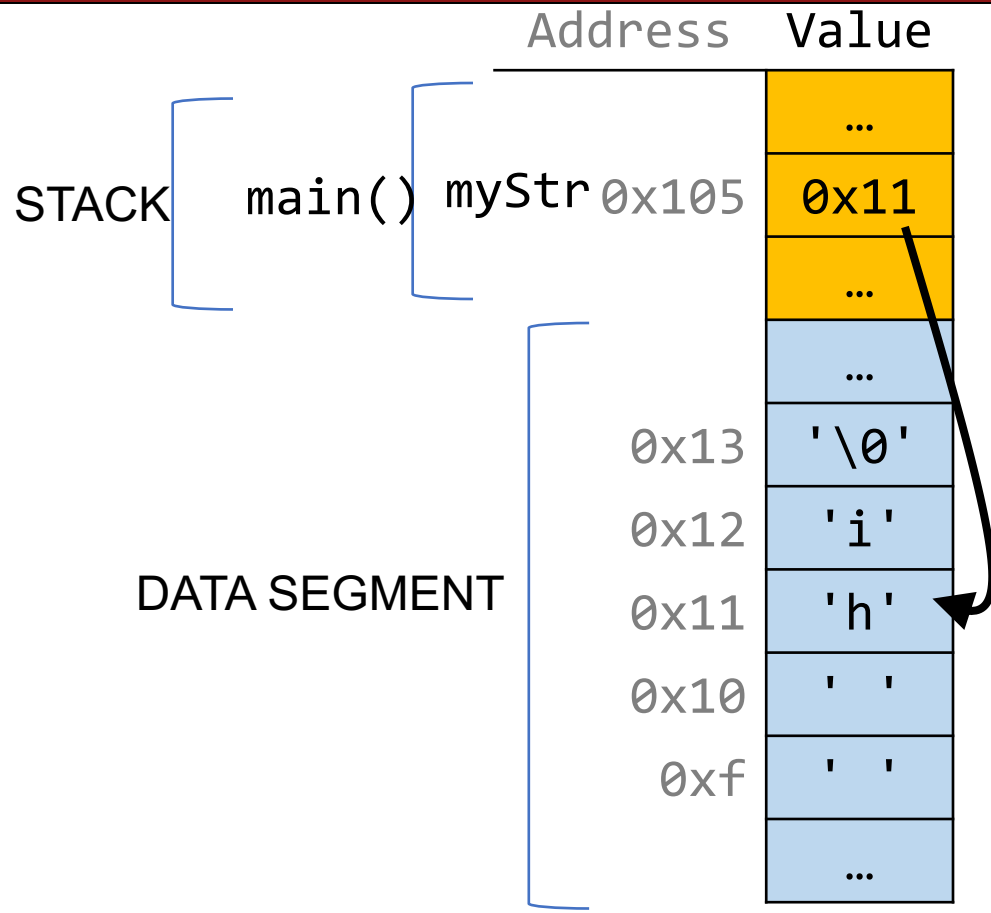
Pointers to Strings

```
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

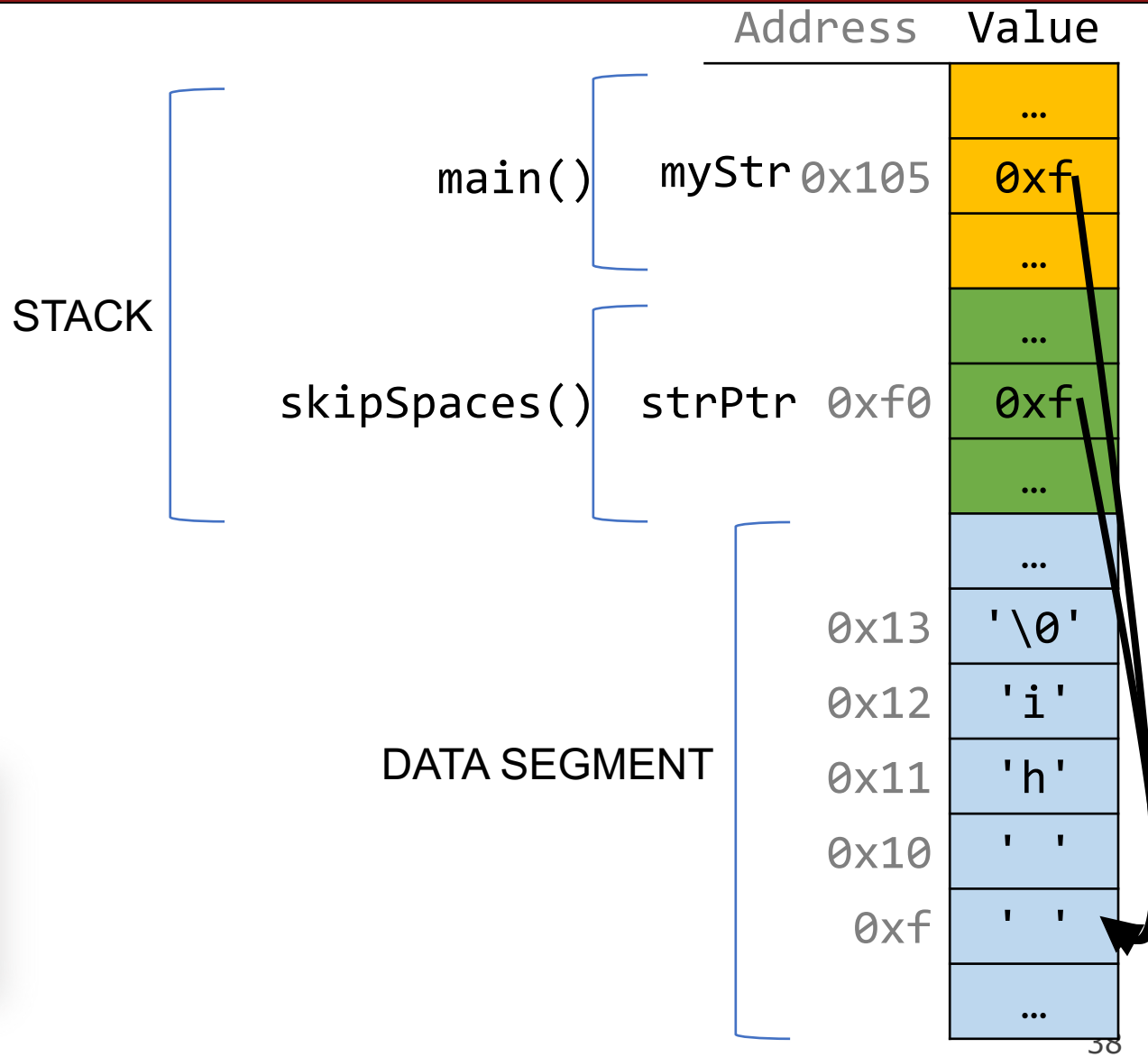
```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

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.

Recap

- **Finishing up:** Strings and Pointers
- Double Pointers

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.

Extra Practice

2. char* vs char[] exercises

Suppose we use a variable `str` as follows:

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

For each of the following initializations:

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

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

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

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

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

2. char* vs char[] exercises

Suppose we use a variable `str` as follows:

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

For each of the following initializations:

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

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

Line A: Compile error (cannot reassign array)

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

Line B: Segmentation fault (string literal)

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

Prints eulo3

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

Line B: Segmentation fault (string literal)

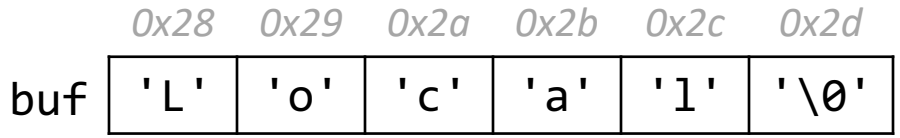
3. Bonus: Tricky addresses

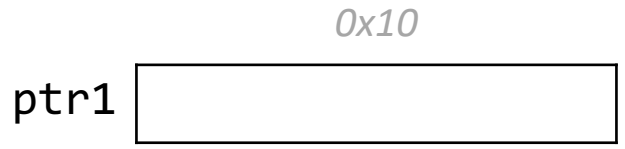
```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("        address:  %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);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

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

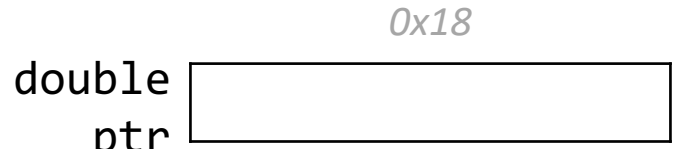
What is stored in each variable?

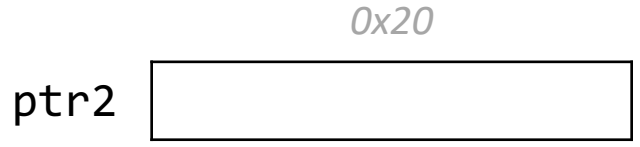
3. Bonus: Tricky addresses

```
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 }
```







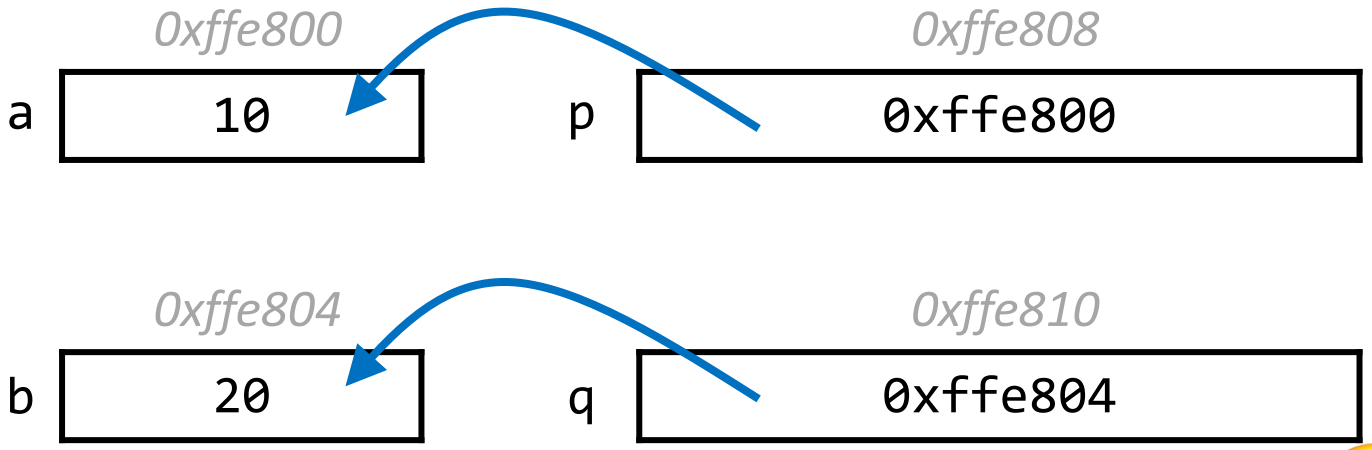


While Line 10 raises a compiler warning, functionally it will still work—because pointers are **addresses**.

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 }
```

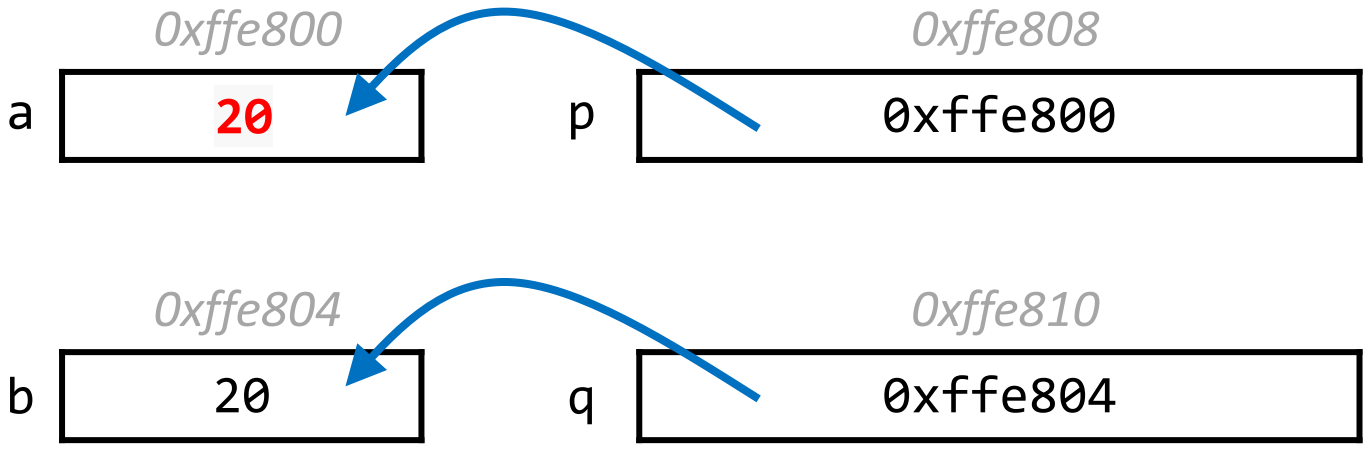
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



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 }
```

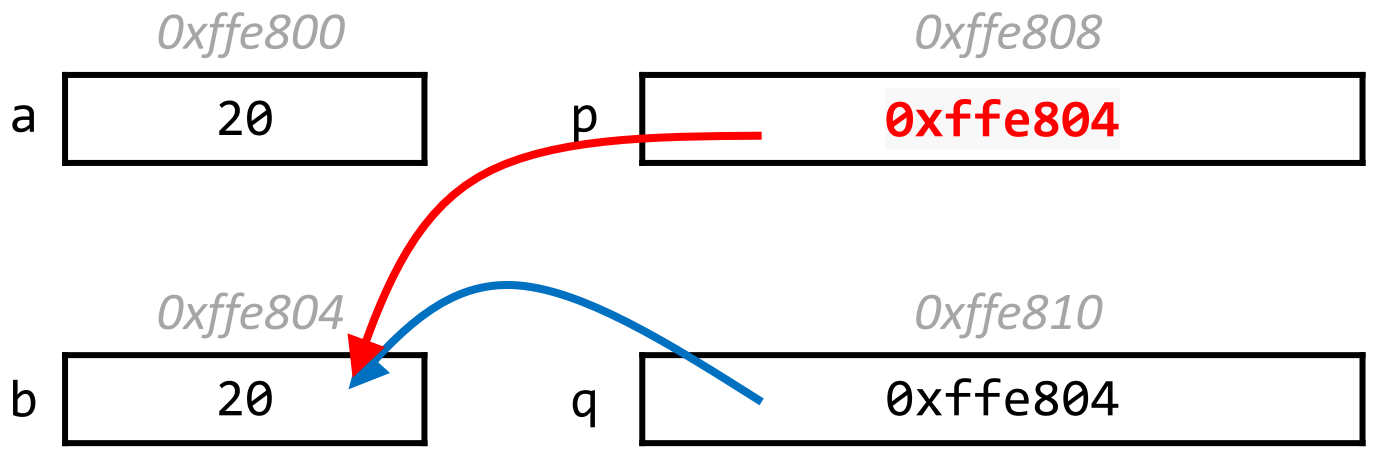
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



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';
```

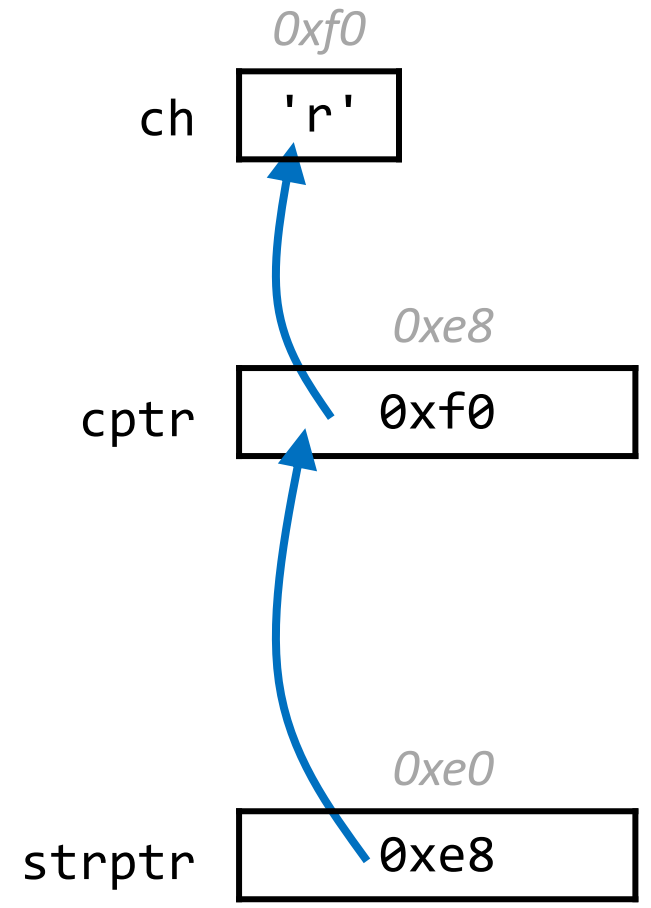
ch stores a char

```
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)

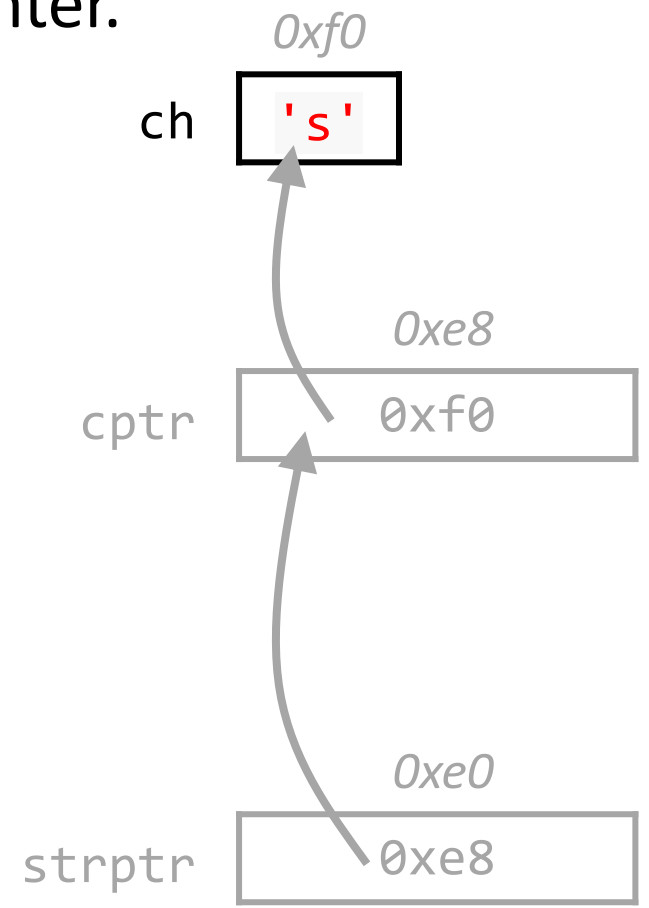
In **reading values from/storing values**, * **dereferences** a pointer.

```
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.

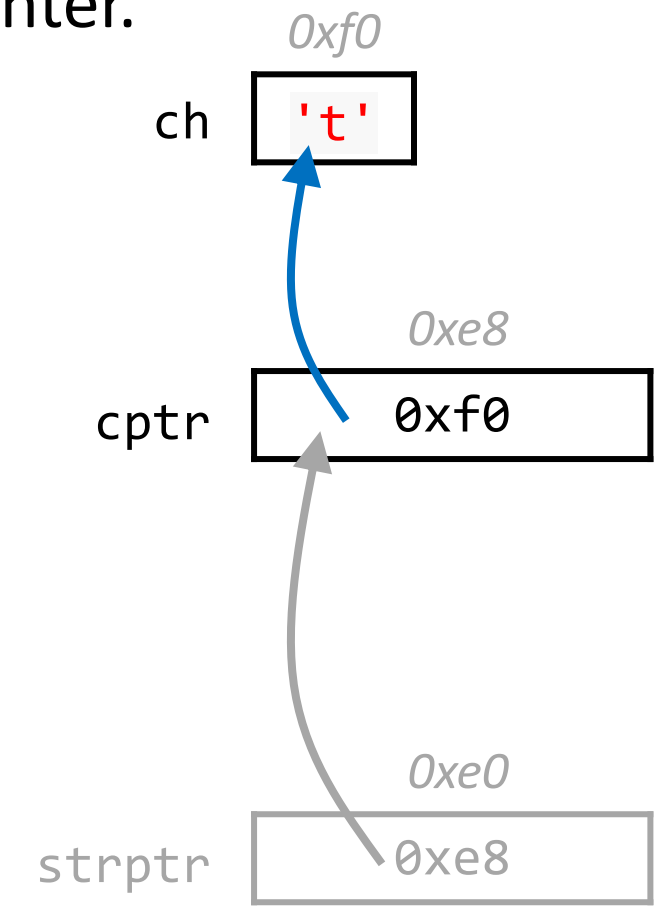
```
char ch = 'r';
ch = ch + 1;
```

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;
```

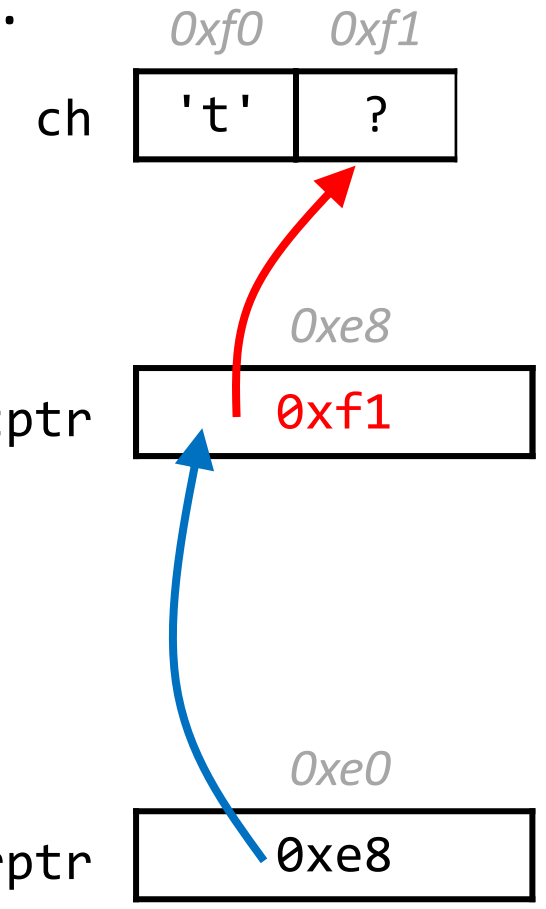
Increment value stored in ch

```
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**)

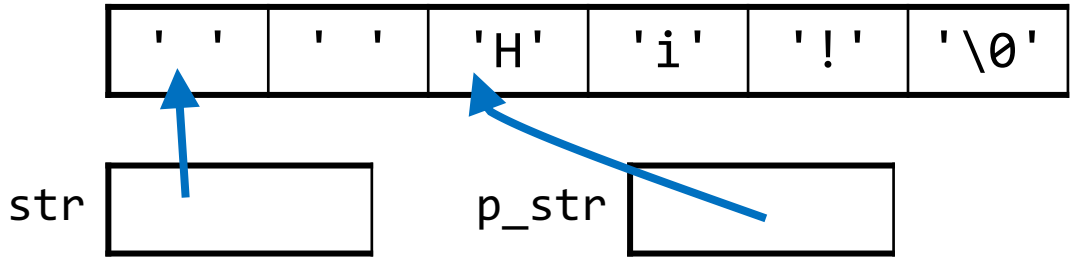


Skip spaces

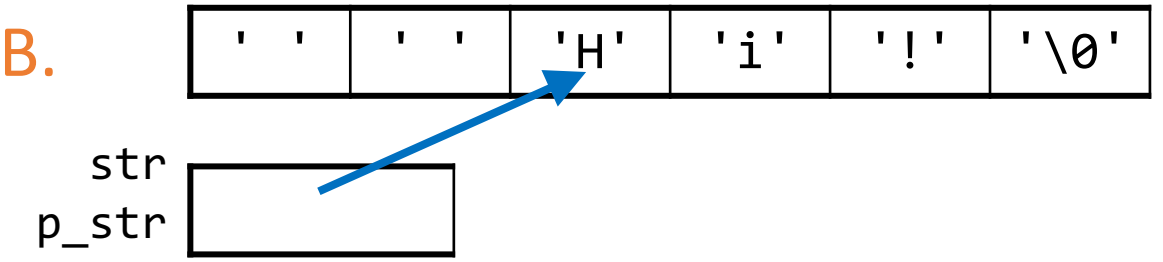
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

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

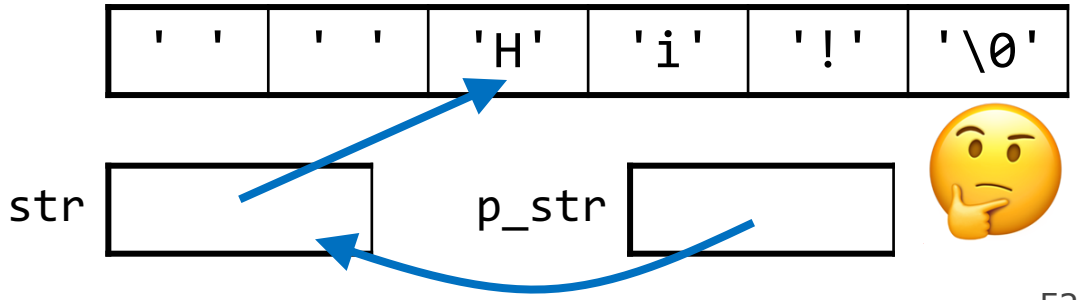
A.



B.



C.

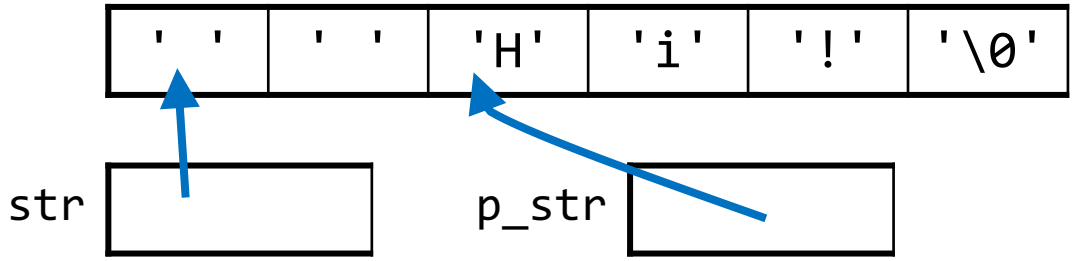


Skip spaces

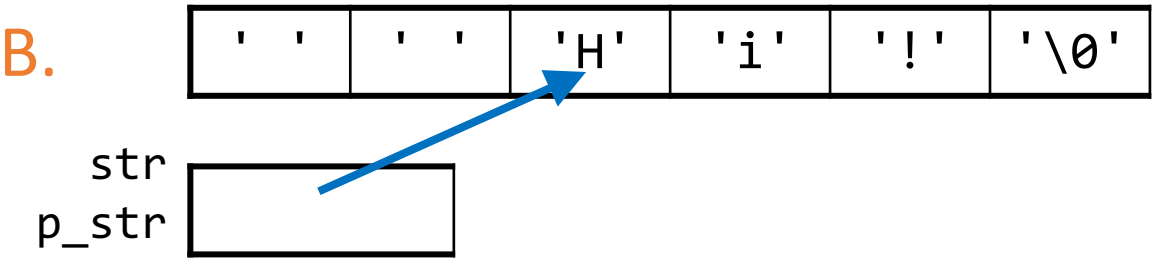
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

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

A.



B.



C.