Computer Systems

CS107

Cynthia Lee

Topics

LAST TIME:

- Pointers and arrays (including strings—arrays of char)
- Pointer arithmetic
- Strings
- Files, error() reporting

THIS TIME:

- Two tools/tricks that help us understand memory:
  - "sizeof"
  - When sizeof works on an array and when it doesn't
- Dynamic memory: malloc and free
- Where's my data?
  - Stack vs heap vs data segment
- Pointers to pointers

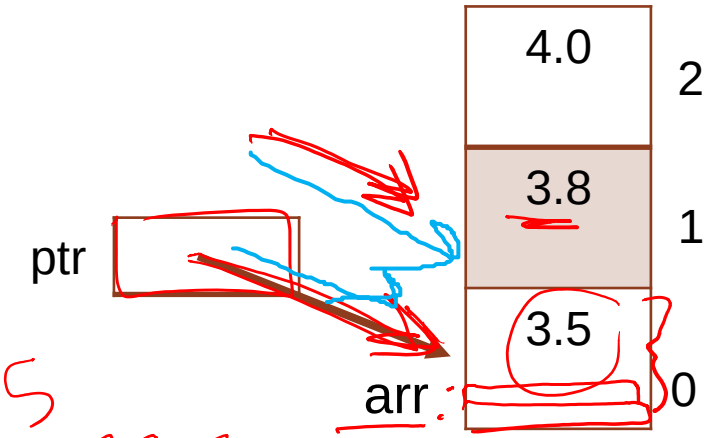
Code example: sizes.c, ptr.c

See samples/lect4 directory on myth for code

Applying this to an example from last time

We can add one to a pointer to access the next element in the array

```
int main(int argc, char *argv[]) {
    double arr[3];
    double *ptr = arr;
    ptr[0] = 3.5;
    ptr[1] = 3.8;
    ptr[2] = 4.0;
    printf("%p => %g\n", ptr, *ptr);
    printf("%p => %g\n", ptr+1, *(ptr+1));
    printf("%p => %g\n", ptr+1, ptr[1]);
```



- Important note: the last two lines are *completely equivalent.* C invented the array[index] notation as a shorthand version of *(array + index) notation, because it is so common to want to do that and the latter is clunky.

Dynamic memory: `malloc` and `free`

Arrays in C (on the heap)

```
int main(int argc, char *argv[]) {
    /* one-step process for stack */
    double arr1[3];
    /* two-step process for heap */
    double *ptr;
    ptr = malloc(3*sizeof(double)); //calloc similar but 0-fills
```

- All about **malloc**:
  - Like "new" in C++, but more basic
    - `void * malloc(int);`

Returns pointer to the location it has reserved for you

Takes an integer number of bytes to allocate (you need to do the math on how much you need)

Heap memory works like a hotel registration desk

(GOLDEN GLOBE WINNER GRAND BUDAPEST HOTEL)

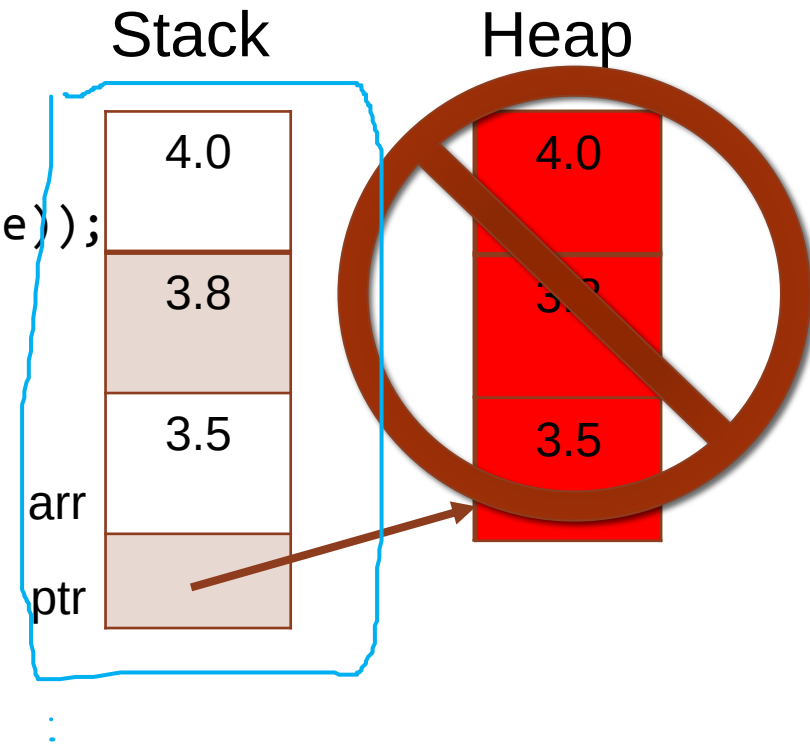
malloc's best friends: realloc and free

```
int main(int argc, char *argv[]) {
    double *ptr;
    ptr = malloc(3*sizeof(double));
    ptr[0] = 2.5;
    ptr = realloc(ptr, 5 * sizeof(double));
    free(ptr);
```

- All about **realloc**:
  - It gives you a larger (or smaller) space, still contiguous!
  - If the adjacent space was unused, will give you that
    - Otherwise will copy values over for you to a new, bigger space
- All about **free**:
  - Like new/delete in C++, malloc/free always needs to come in pairs!
  - Failing to free something you malloc-ed when you are done using it is a **memory leak**
- Of course, after you realloc or free memory, you never try to access it again….

malloc + free example

```
int main(int argc, char *argv[]) {
    double arr[] = {3.5, 3.8, 4.0};
    double *ptr = malloc(3*sizeof(double));
    ptr[0] = 3.5;
    ptr[1] = 3.8;
    ptr[2] = 4.0;
    free(ptr);
```



Only a someone like Norman Bates would access a hotel room that isn't theirs (either never was, or was but checked out)

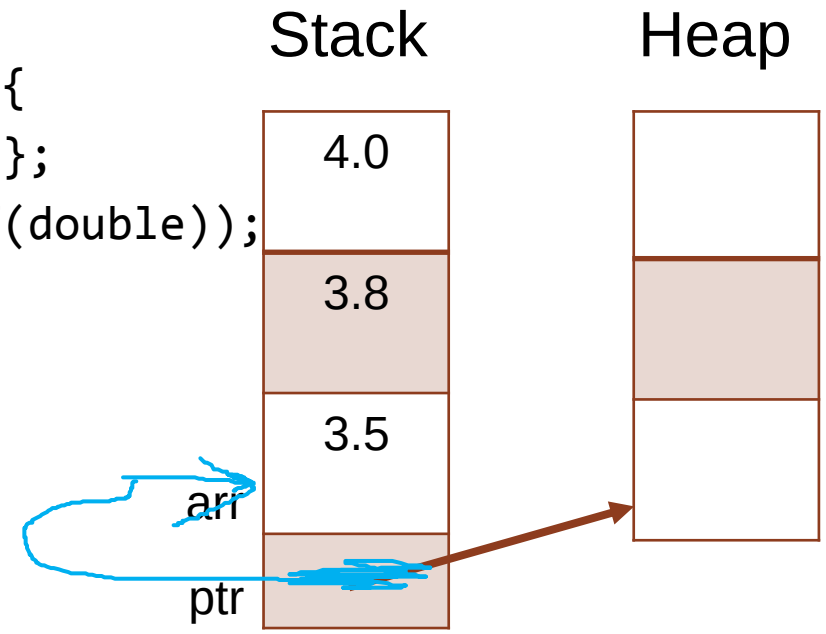
(DON'T BE A NORMAN BATES!!)

Arrays and Pointers

Pointers and arrays

```
int main(int argc, char *argv[]) {
    double arr[] = {3.5, 3.8, 4.0};
    double *ptr = malloc(3*sizeof(double));
    ptr = arr;  /* is this ok? */
```

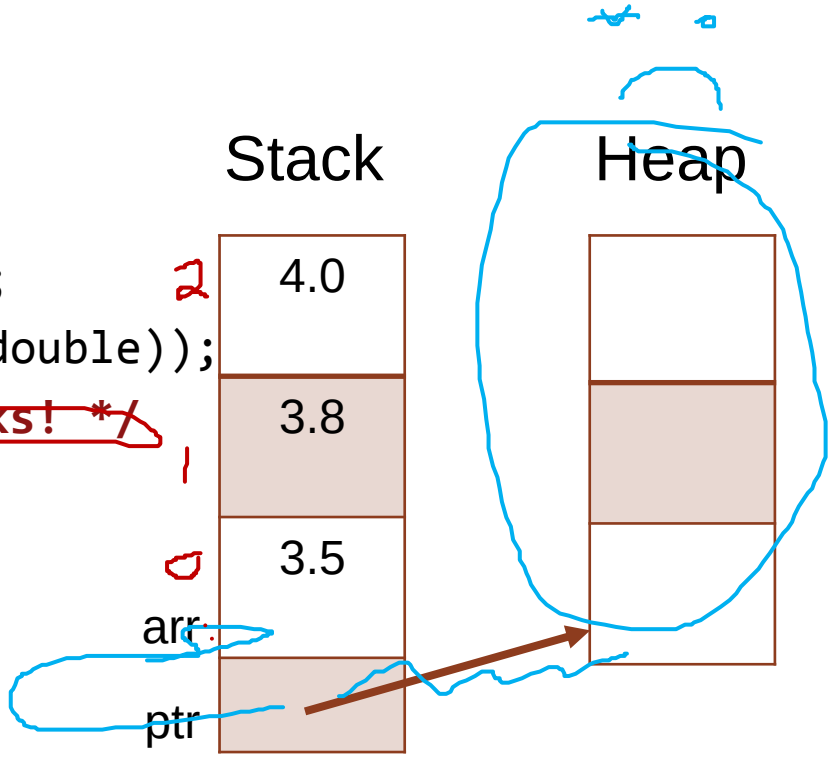
- Hmmm…what happens in this case?



Pointers and arrays

```
int main(int argc, char *argv[]) {
    double arr[] = {3.5, 3.8, 4.0};
    double *ptr = malloc(3*sizeof(double));
    ptr = arr;  /* last slide: leaks! */
    arr = ptr;  /* is this ok? */
```

- Hmmm…what happens in this case?



Strings in C

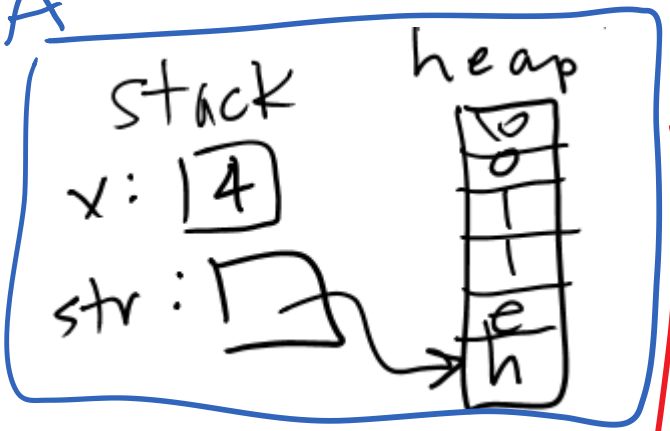
Recap of memory diagrams of their possible locations in memory

Strings in C: just an array of chars, but with a special ending sentinel value

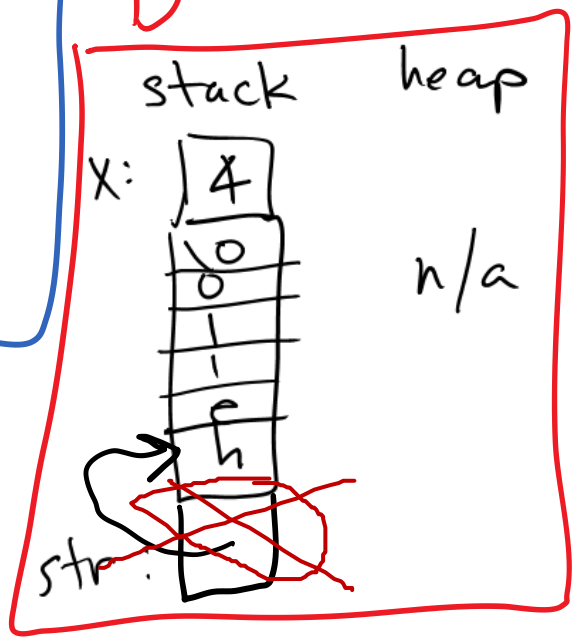
```
int main(int argc, char *argv[]) {
    int x = 4;
    char str[] = "hello";
```

- **What does memory look like?**

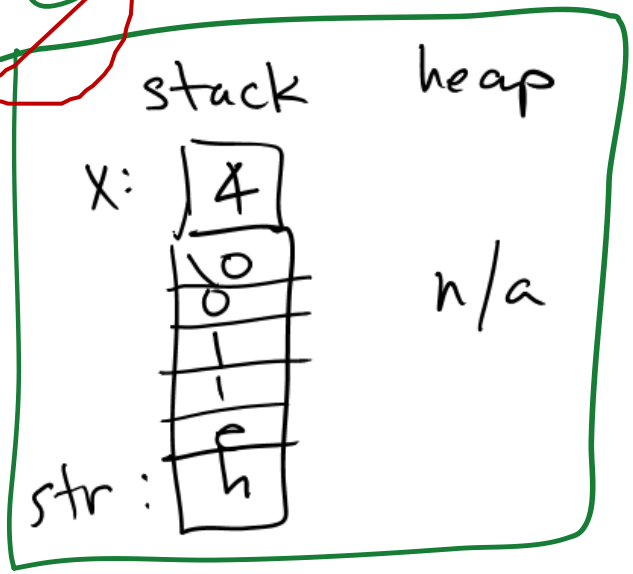
A



B



C



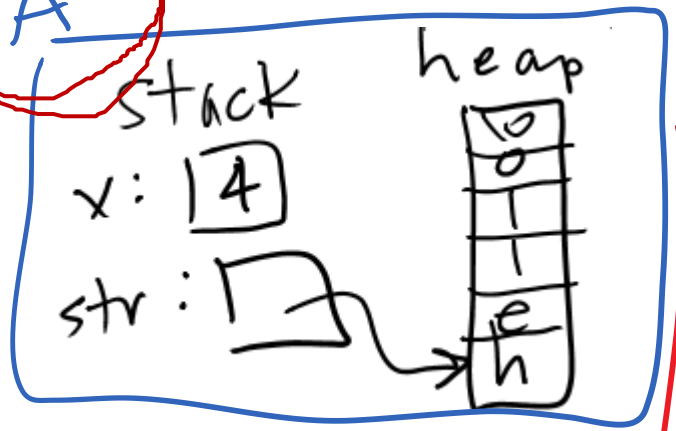
D (something else)

Strings and strdup: the gory details

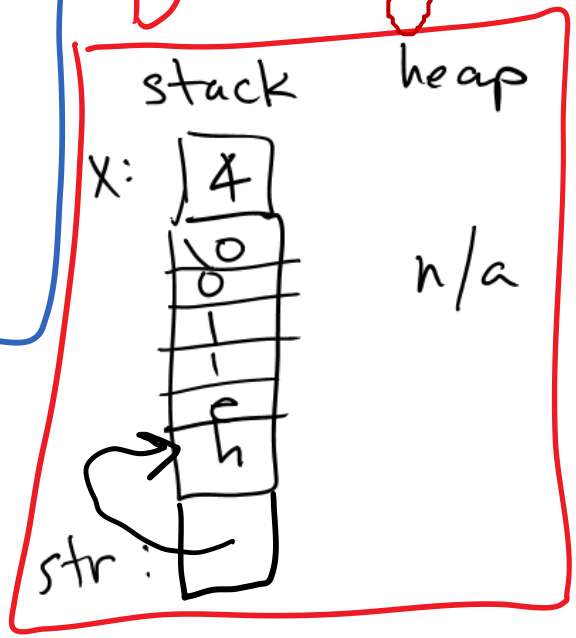
```
int main(int argc, char *argv[]) {
    int x = 4;
    char *str = strdup("hello");
```

- What does memory look like?

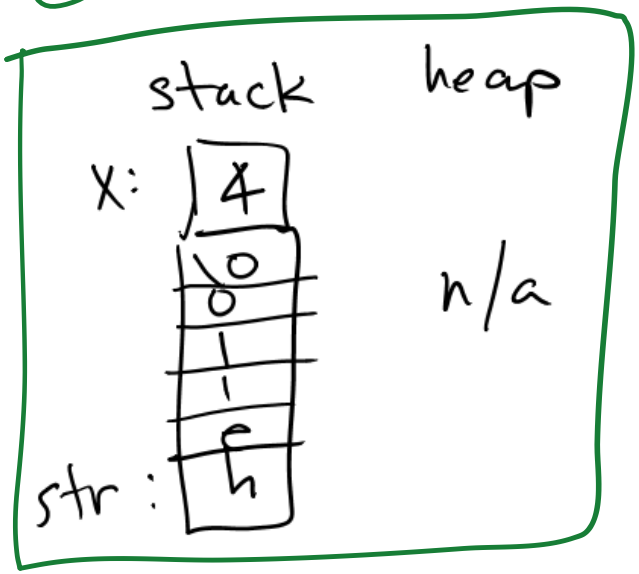
A



B



C



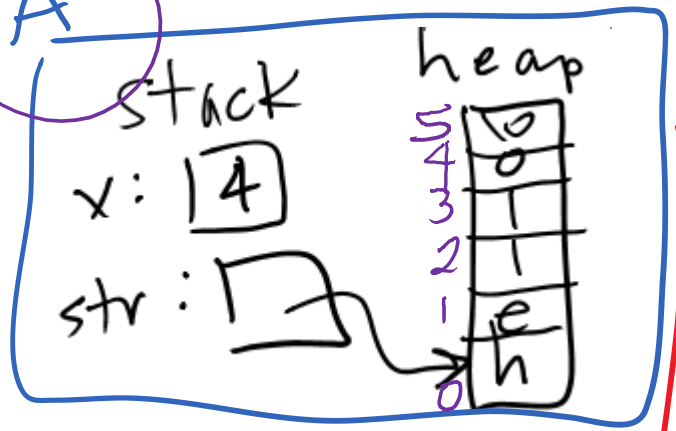
D (something else)

Strings and malloc: the gory details

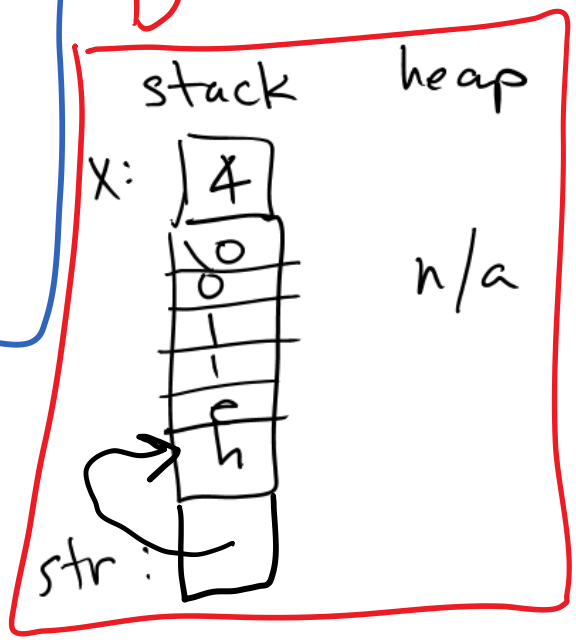
```
int main(int argc, char *argv[]) {
    int x = 4;
    char *str = malloc(6); //why not 5?
    strcpy(str, "hello");
```

- **What does memory look like?**

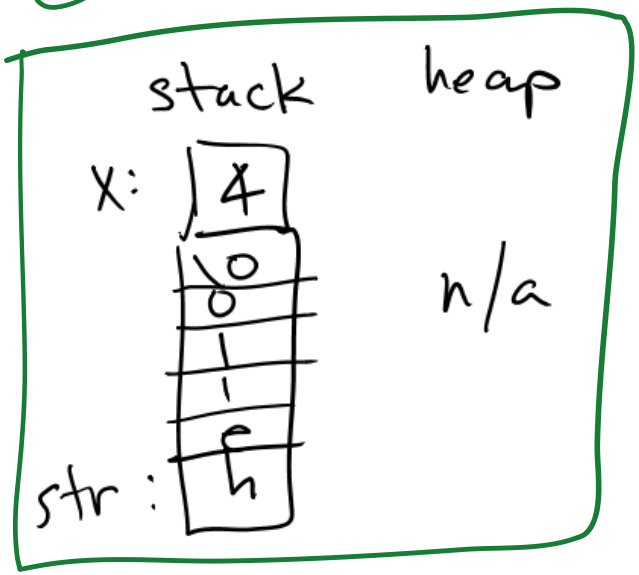
A



B



C



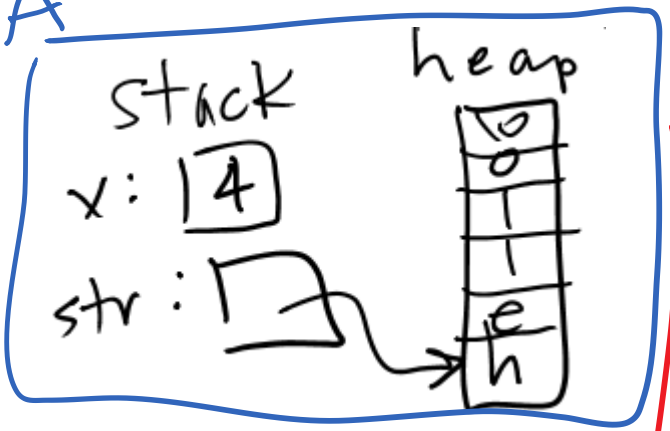
D (something else)

Strings in C: **even gorier** details

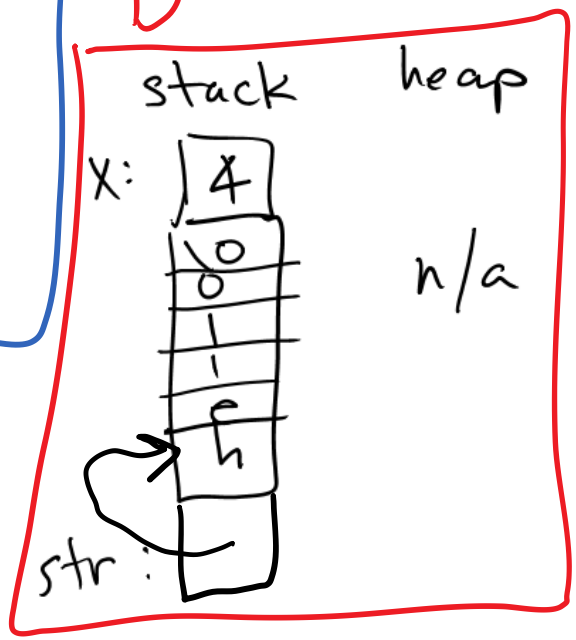
```
int main(int argc, char *argv[]) {
    int x = 4;
    char *str = "hello";
```

- **What does memory look like?**

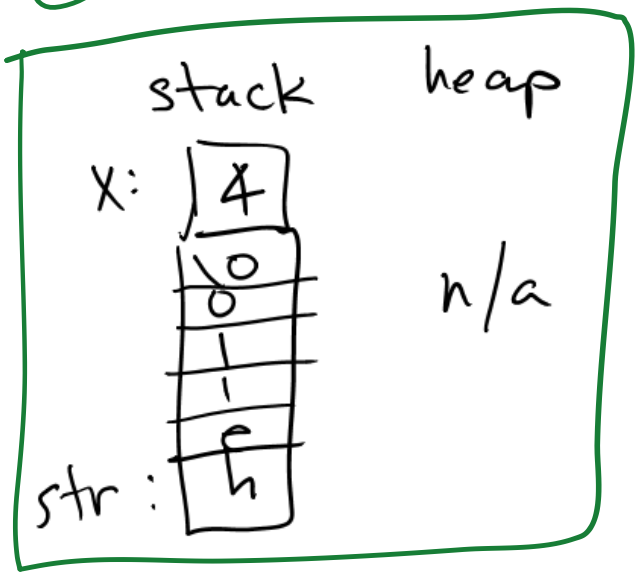
A



B



C

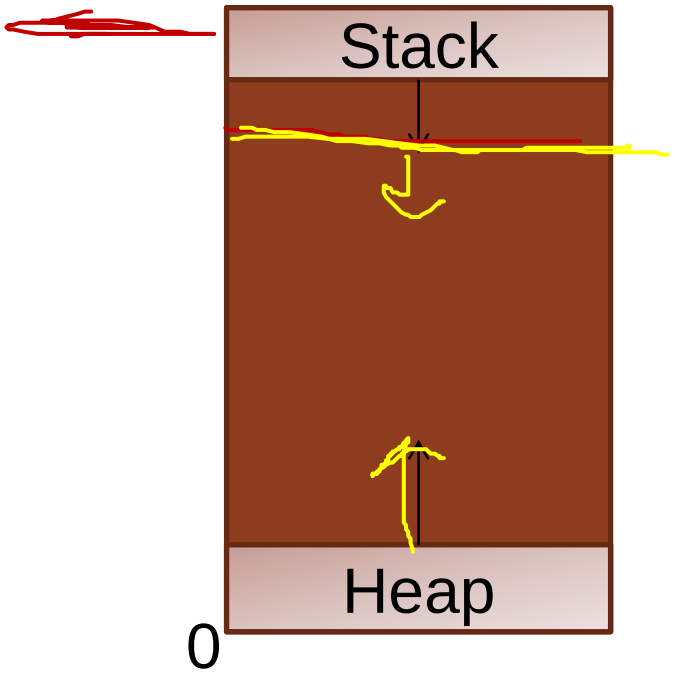


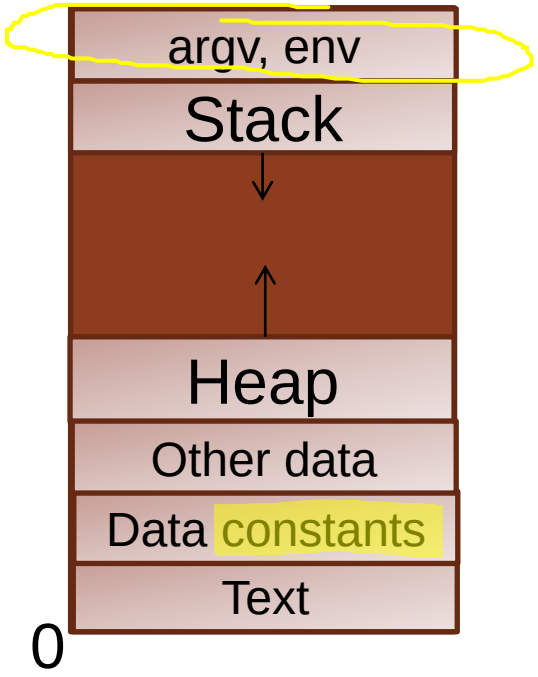
D (something else)

Strings in C: more gory details

```
int main(int argc, char *argv[]) {
    int x = 4;
    char *str = "hello";
```

- **What memory looks like, updated version with more detail:**



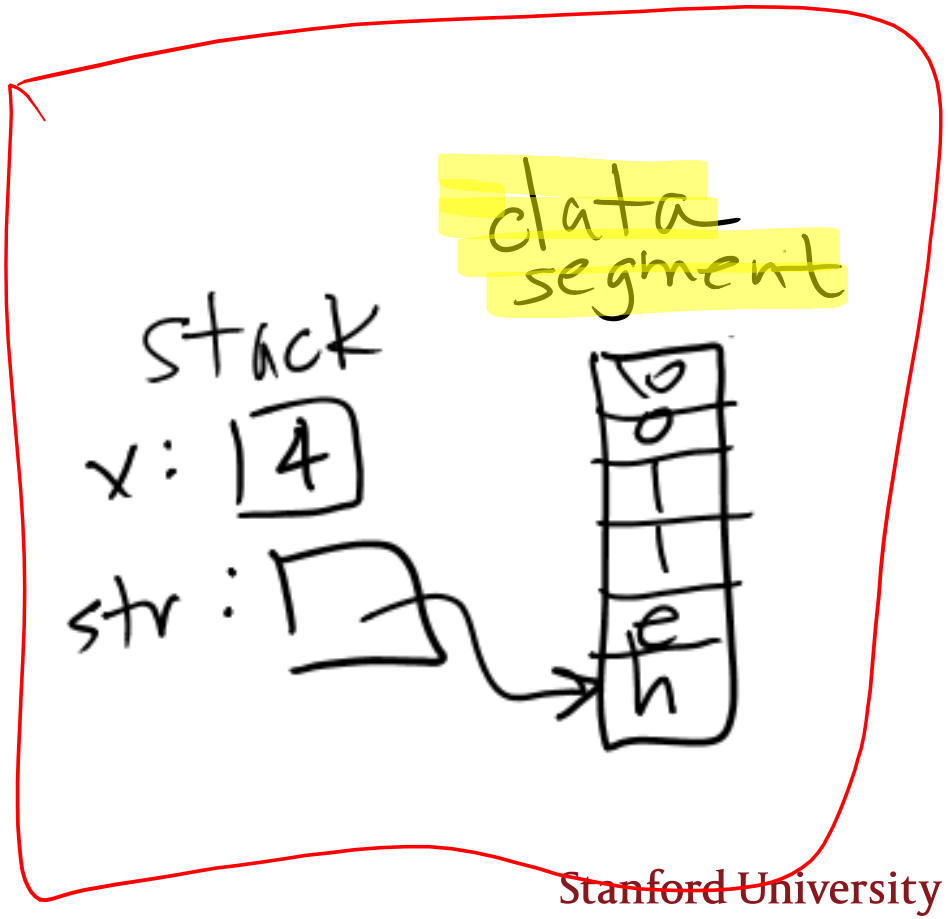


Strings in C: **even gorier** details [CORRECT ANSWER]

```
int main(int argc, char *argv[]) {
    int x = 4;
    char *str = "hello";
```

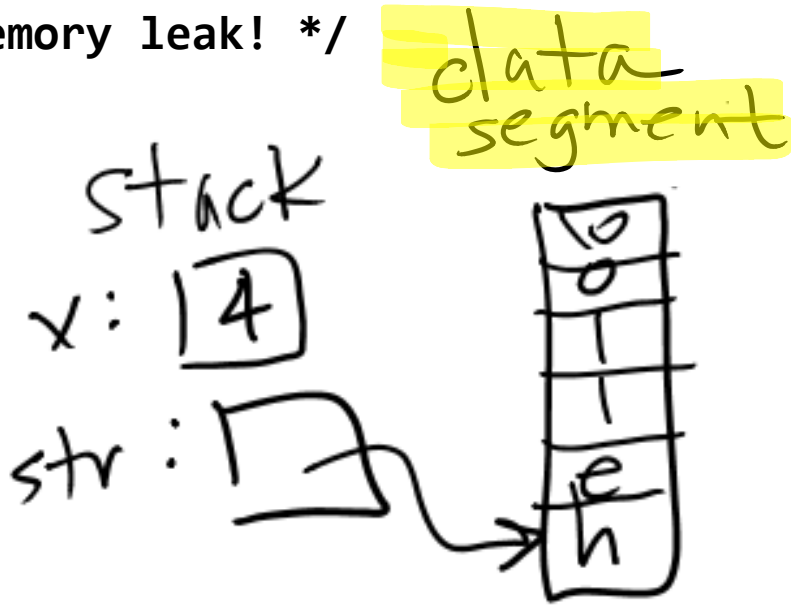
- **What does memory look like?**

D
(something else)



Strings in C: *Leonardo DiCaprio cauterizing his own wound in the Revenant level of gory* details*

```
int main(int argc, char *argv[]) {
    int x = 4;
    char *str = "hello";
    str[4] = 'a'; /* not allowed – read only */
    str = NULL;   /* ok! not a memory leak! */
```



* confession: I haven't seen it, only heard about it

Strings in C: passing them as arguments

Passing strings as arguments: code demo key points

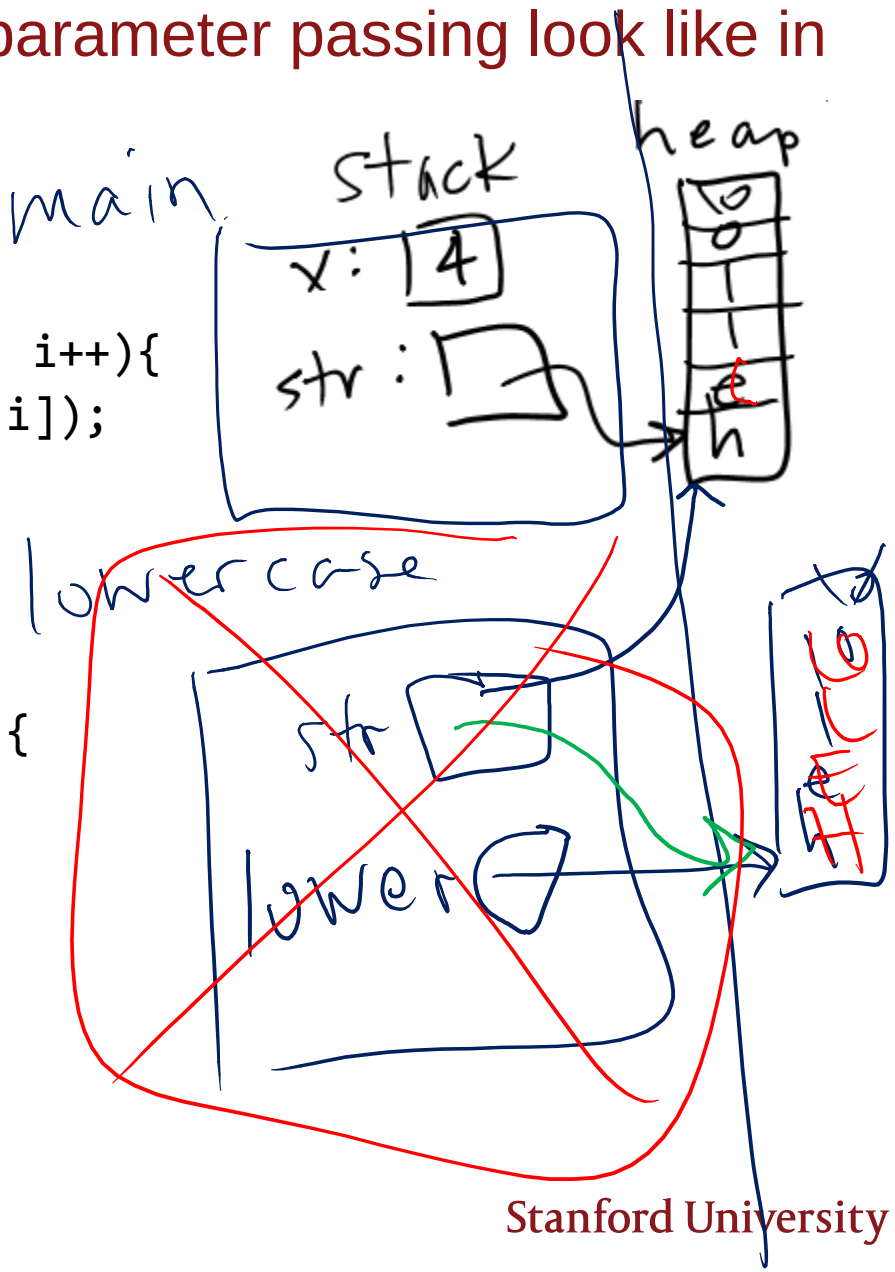
```
void lowercase(char *str) {
```

- You don’t need to pass length (be careful with this)
- You *may* alter the *contents* of a char* argument

Strings in C: what does char* parameter passing look like in memory?

```
void lowercase(char *str) {
    char * lower = strdup(str);
    for (int i=0; str[i] != '\0'; i++){
        lower[i] = tolower(lower[i]);
    }
    str = lower;
}

int main(int argc, char *argv[]) {
    int x = 4;
    char *str = strdup("HeLLo");
    lowercase(str);
    printf("%s\n", str);
    free(str);
}
```



- **What does memory look like?**

Passing strings as arguments

- You don't need to pass length (assuming the string is correctly set up with a null terminating character)
- You may alter the *contents* of a char* argument, but not redirect the pointer
  - For example, if you want to lengthen the string, you're out of luck with char*
  - If you want to do this, add a level of indirection that gives you access to the char* pointer itself: char** (this is essentially passing the pointer by reference), or return a char*

```
void lowercase(char **str) {
char* lowercase(char *str) {
```