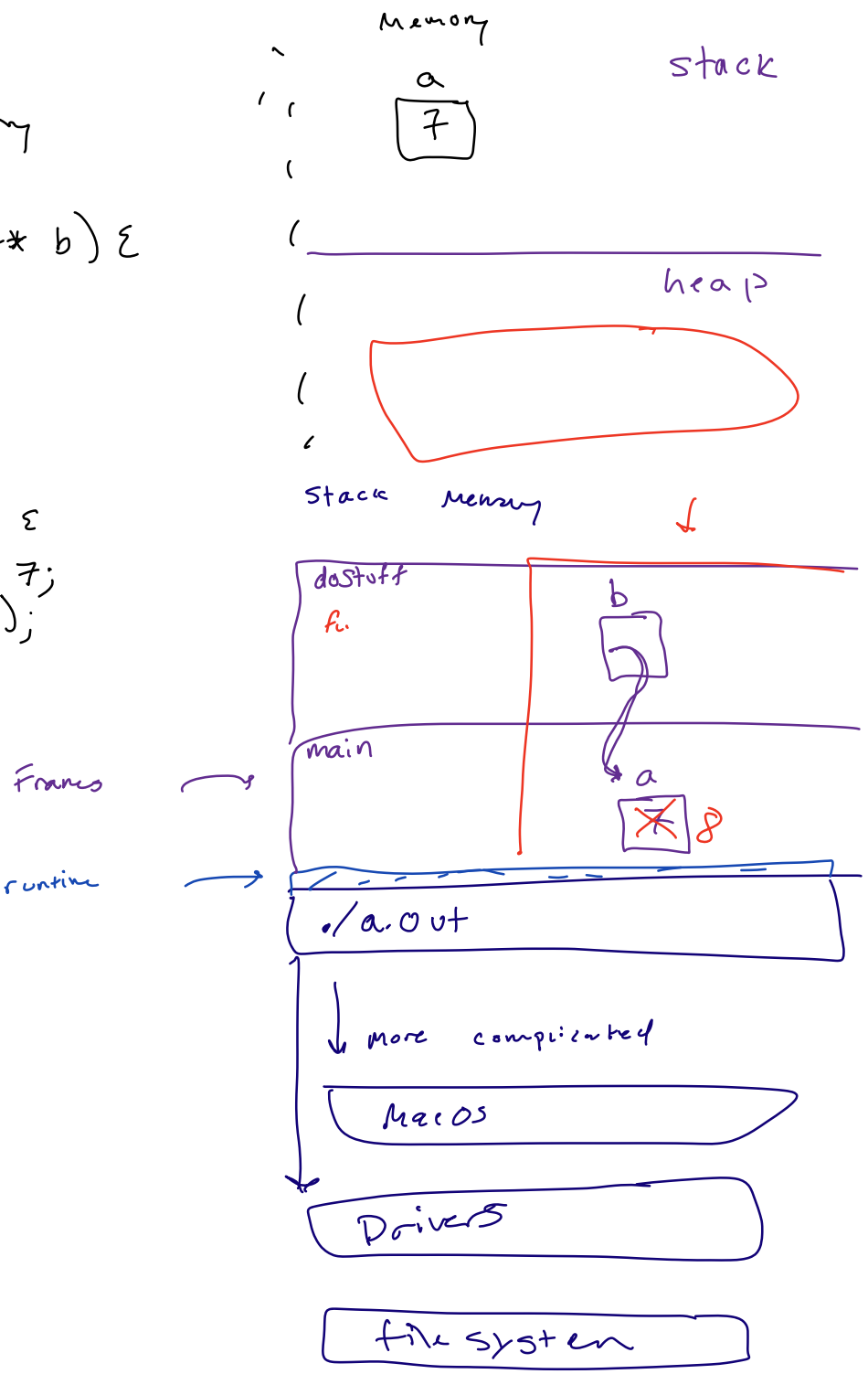
Arrays

→ Memory

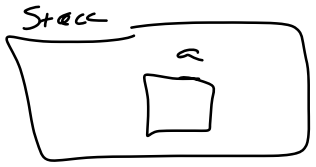
```
void doStuff(int* b) {
    *b = 8;
}

int main() {
    int a = 7;
    doStuff(&a);
}
```



Arrays

declaring → asking for spot in stack memory.

```
int a;
a = 7;
```



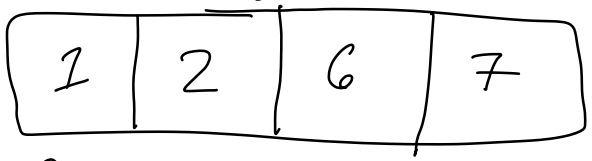
declare an array
fixed length array

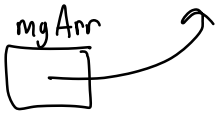
```
int myArr[4];
```

← cannot variable. Must be hard coded.
← Declare an array

```
myArr[2] = 6;
myArr[3] = 7;
myArr[0] = 1;
myArr[1] = 2;
```

initialization





0 1 2 3

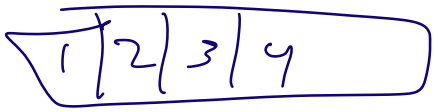
```
for (int i = 0; i < 4; i++) {
    myArr[i] = 0;
}
```


```
int sum = myArr[2] + myArr[3];
```

literally just an integer.

declare and initialize at same time

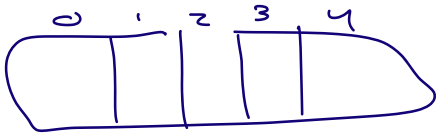
```
int myArr[] = {1, 2, 3, 4};
```



Example) STACK-BASED Fixed Length Hard-coded Arrays.

```
double mean(double[] myArr, int size_myArr)
    double sum = 0;
    for (int i = 0; i < size_myArr; i++) {
        sum += myArr[i];
    }
    return (sum / size_myArr);
}
```

```
int main(){
    double myArr[5];
    for (int i=0; i<5; i++){
        scanf("%lf", &(myArr[i]));
    }
    double res = mean(myArr, 5);
}
```

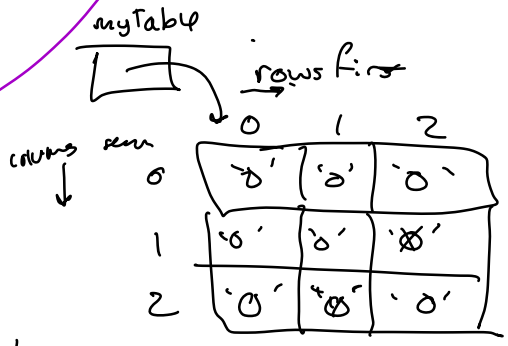


Multidimensional Arrays

2D array of length 5

2D arrays are "tables"

3D arrays are tensors




```
char myTable[3][3];
```

row col

```
myTable[2][1] = 'X';
myTable[1][2] = 'X';

for(int i=0; i<3; i++){
    for(int j=0; j<3; j++){
        myTable[i][j] = '0';
    }
}
```

1	2	3
4	5	6
7	8	9

```
int myTable[][] = {{1,2,3},{4,5,6},{7,8,9}};
```

Strings (basically just arrays...)

```
char myArr[];
```

(1) Make string "hi" ← string literally

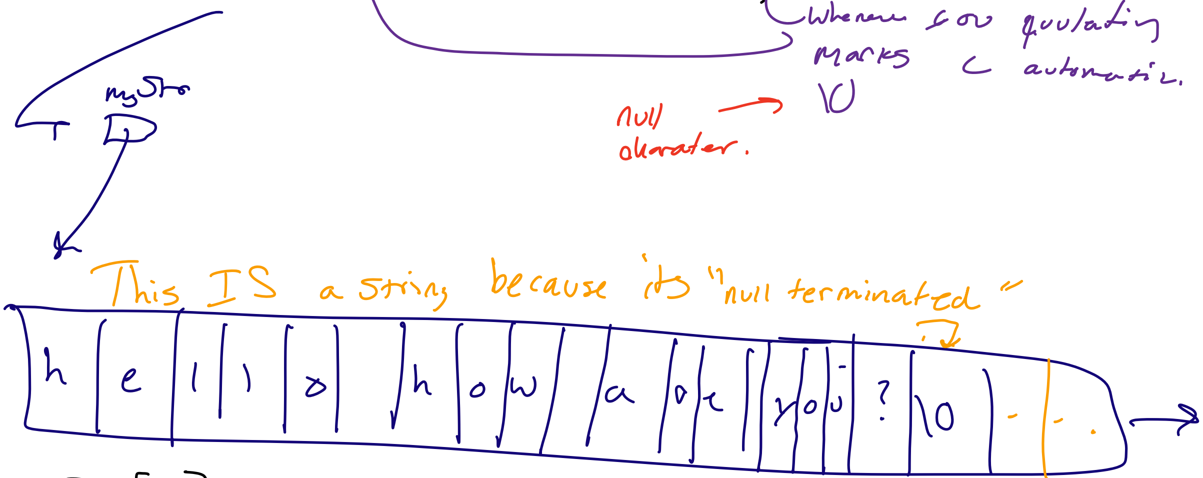
```
char[] myStr = "hello how are you?"
```

Whenever you quotating marks c automatic \0

null character → \0

Tragic Story

> Strings are character arrays
> BUT
> they must end with '\0' in order to be called a string.



myStr[10] ← 10th character

```
int strlen(char[] str) {
    int count = 0;
    while(str[count] != '\0') {
        count++;
    }
    return (count);
}
```

this already exists for you in <string.h>.

"loops forever and Segfault"

```
char myChars[] = {'a', 'b', 'c', 'd'};
```

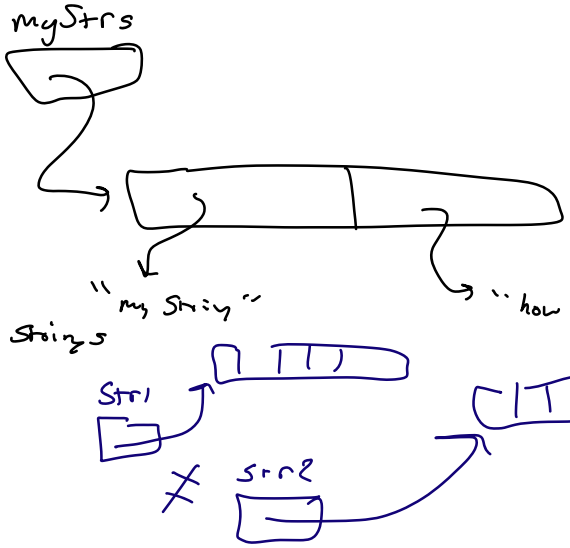
← NOT A String

```
char myStrs[][] = {"my string", "how are you"};
```

← not a string

myStrs[0] ← string

myStrs[1] ← string



in <string.h>

strcmp ← this used to compare strings


```
char[]  str1 = "hello";
char[]  str2 = "goodbye";
if  (str1 == str2 ) {
```

3

if are equal ↓

if (strcmp(str1, str2) == 0) → strcmp { -1 means left < right; 0 equal; 1 mean right > left

string copy ↓

strcpy(str1, str2) ← might get error because str2 is longer than str2 and str1 may only len str2 boxes

dst src

but

strcpy(str2, str1) ← okay because str2 has at least as many boxes as str 2,

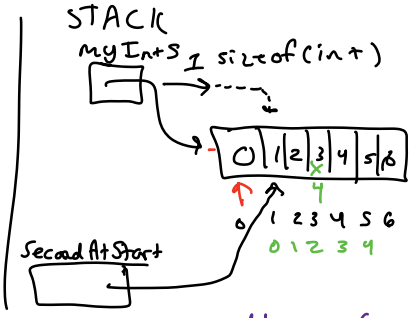
strncpy(str2, str2, 5) ← okay str2 has at least 5 boxes.

Everything is a Pointer (except primitive data types)

thrid
12

```
int myInts[7];

myInts == &(myInts[0]);
*myInts == myInts[0];

*(myInts + 1) == myInts[1];

int thrid = *(myInts + 2); // 2
```

says take address of myInts and add 2 spots then dereference which is equiv to myInts[2];

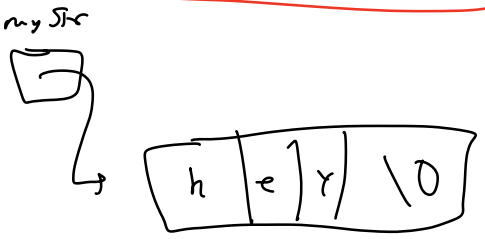



```
int* startAtSecond = myInts + 1; // don't dereference
startAtSecond[0] == myInts[1];
startAtSecond[2] = 4;
printf("%d", myInts[3]); // 4
```

```
myArr[X] == *(myArr + X)
```

Strings are pointers too.

```
char *myStr = "hey";

*myStr == 'h' == myStr[0];
```



Office hours
7 - 8pm

HEAP

how do I access the heap?

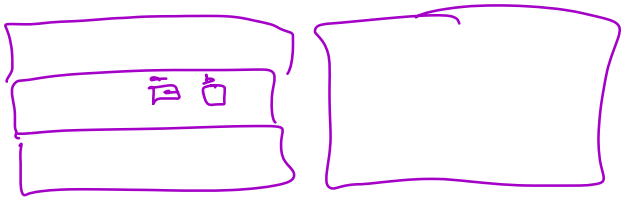
malloc ← gateway to the heap.

memory ↗ ← allocate

to delete from the heap you use "free".

stack | heap

wisedom ## malloc should equall number frees.



heap exists everywhere and for the duration of your program.

Stack Array

```
int myInts[7];
```

← stack array are stupid because you have to hardcode the size AND you are limited by the stack

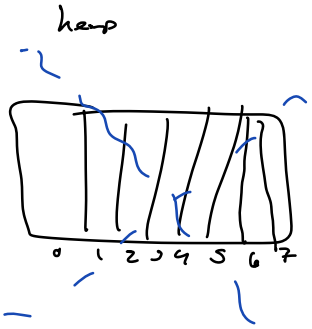
in their defense stack memory is a couple (CPU clock cycles) faster.

```
int * myArray = NULL;
myArray = (int*) malloc(8 * sizeof(int));
```

count: malloc's return type is void* stack

myArray

↖ doesn't have to be hard coded. could be a variable



myArray[7] ← last

*myArray ← first spot

*(myArray + 7) ← last spot

*(myArray + 8) ← maybe a segfault or something else bad.

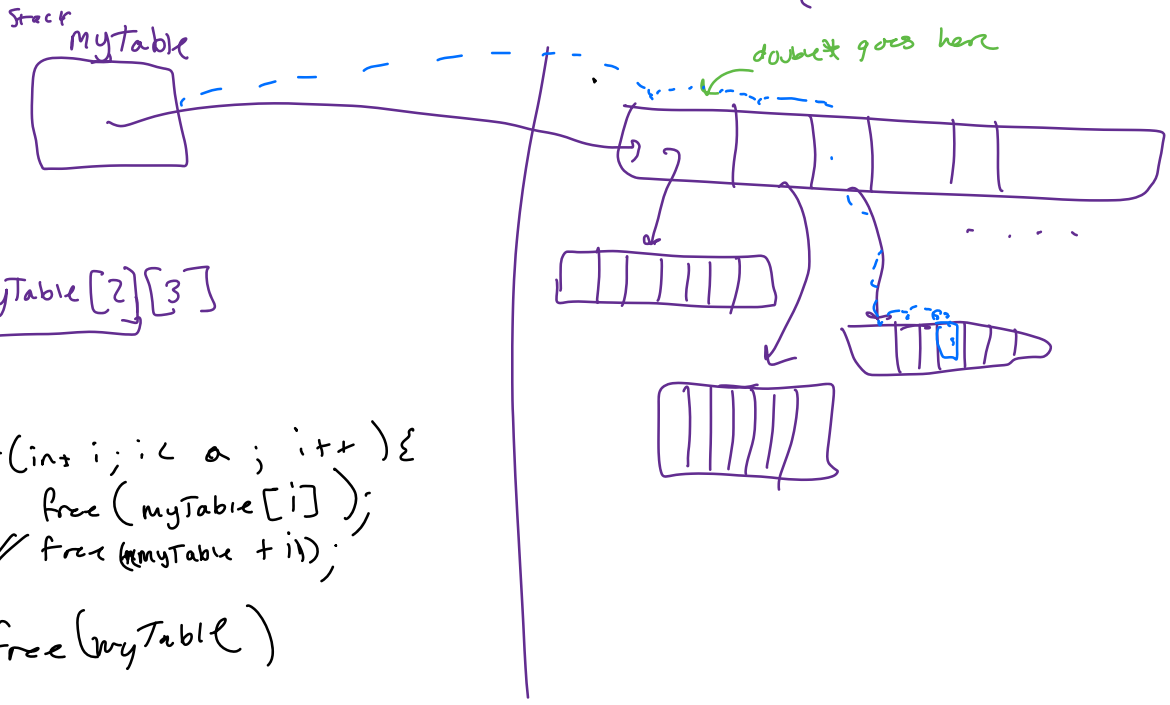
free(myArray)

Say

[][]

```
int a = 6, b = 7;
double** myTable = (double**)malloc(a * sizeof(double*));
for(int i = 0; i < a; i++) {
    myTable[i] = (double*)malloc(b * sizeof(double));
}
```

has double*




```
for(int i; i < a; i++){
    free(myTable[i]);
}// free(*myTable + i));
free(myTable)
```

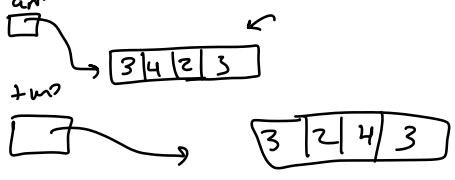
---

ex)

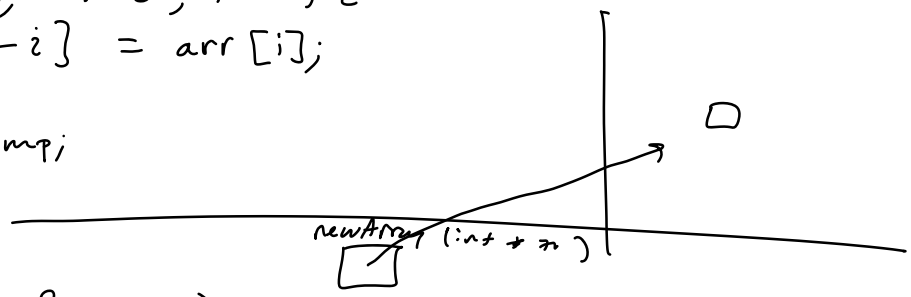
```
void reverseArray(int* arr, int len) {
    int* tmp = (int*)malloc(len * sizeof(int));
    for (int i = len-1; i >= 0; i--) {
        tmp[len-1-i] = arr[i];
    }
    memcpy(arr, tmp, len*sizeof(int));
    free(tmp);
}
```

} fill array backwards

dst src amount



```
Void returnRemoveArray(int *arr, int size, int** newArray) {
    int* tmp = (int*)malloc(len * sizeof(int));
    for (int i = len-1; i >= 0; i--) {
        tmp[len-1-i] = arr[i];
    }
    *newArray = tmp;
}
```



3.

```
int main() {
    int* newarray;
    returnRemoveArray(arr, &newarray);

}
```

```
int* returnNewArray(. ---)
    int* tmp =
    do some thing

    return tmp;
```

FILE pointer

{ "r" → read
  "w" → write
  "a" → append

```
void readFile(char* filename) {
    FILE *fp = fopen(filename, "r")
    // file was opened
    if(fp == NULL) {
        // when file did not open... maybe wrong path.
    }

    int read = 0;
    int len = 0;
    char* line;
    while((read = getline(&line, &len, fp)) != -1) {
        line ← a line of the file
        if line[0] == 'c'
    } } ... do something
```

if -1 the end of file