



# The Heap II

CS2263 – Systems Software Development

## References

Lu, Yung-Hsiang. 2015. CRC Press. New York. Pp 9-27 (Chapter 8)



# Lecture Learning Outcomes

At the conclusion of this presentation students should be able to:

- List and explain the two pitfalls of programming using the heap.
- Describe the difference between shallow and deep copying



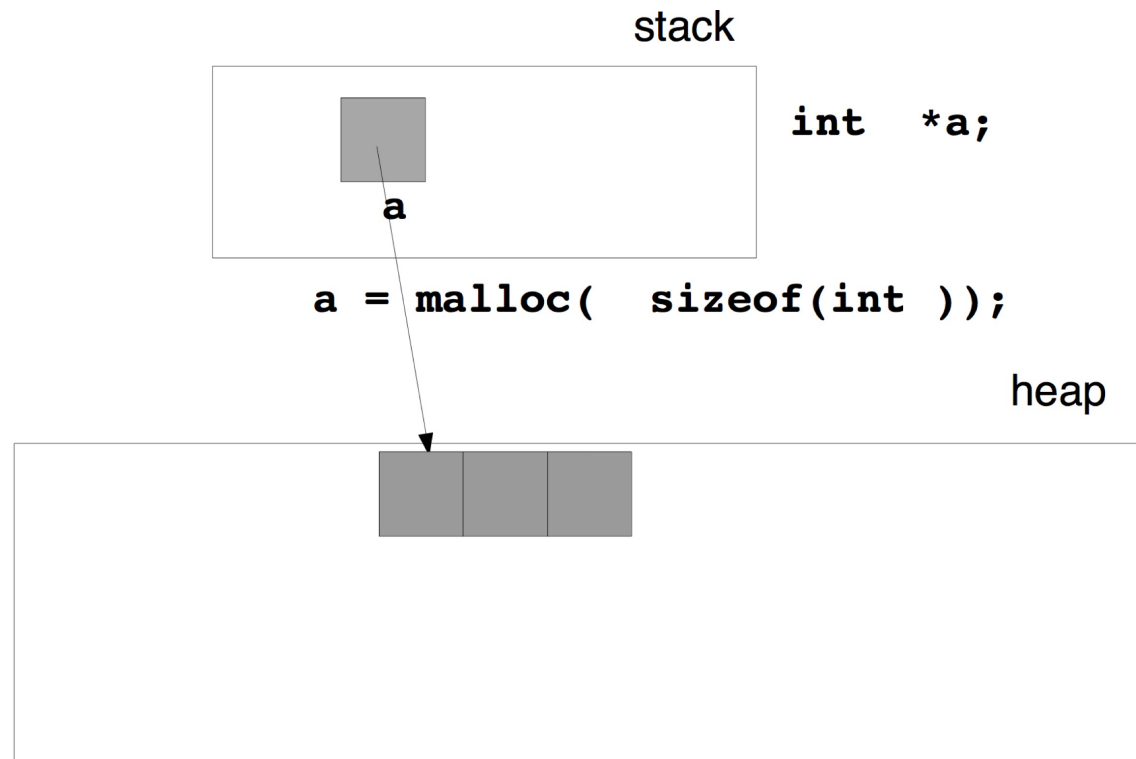
# Review of the Heap

- A programmer-managed region of process memory, just above the stack memory region
- Memory is allocated through `malloc()`
  - Tell it how much contiguous memory you wish (bytes)
  - It returns the address of the start of that memory.
- Memory is deallocated through `free()`
  - Give it the previously allocated address (pointer)



## Stack/Heap Interaction: 1D Array

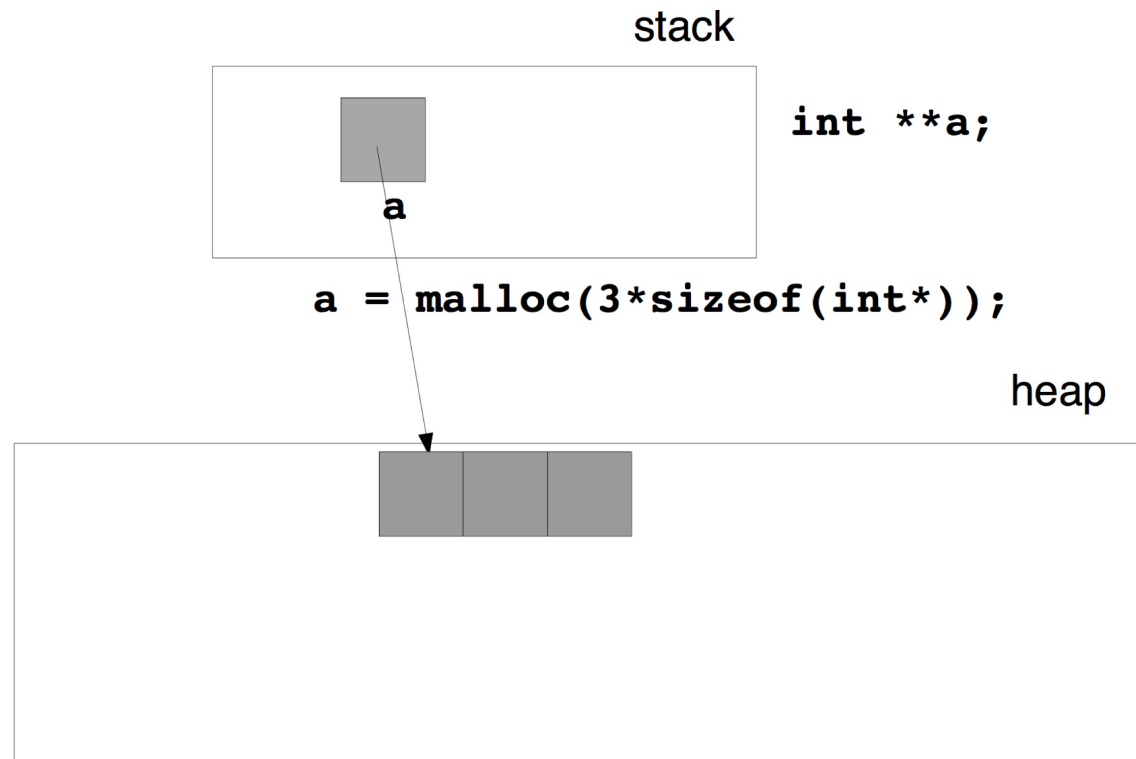
Pointer is declared on the stack, holds the heap address for the start of the allocated array of int values



## Stack/Heap Interaction II 2D Array

version 1, part 1

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int pointers

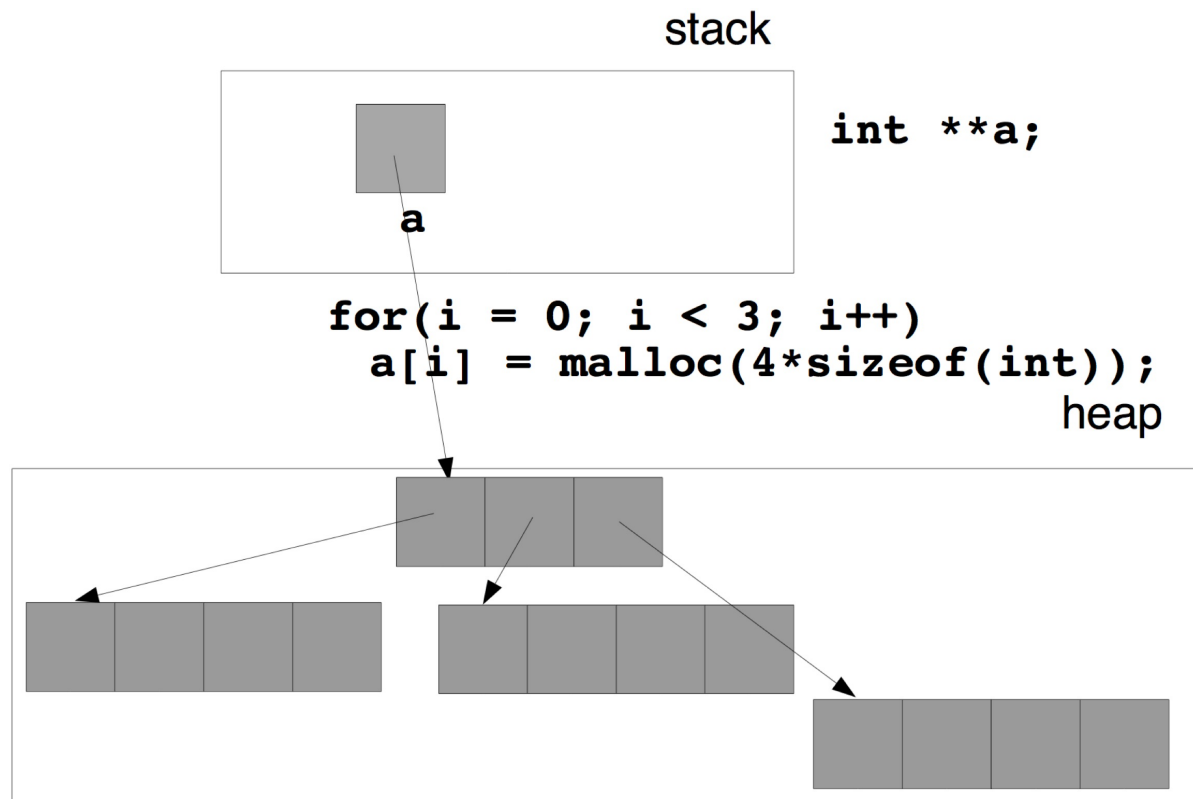


## Stack/Heap Interaction II 2D Array

version 1, part 2

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int pointers

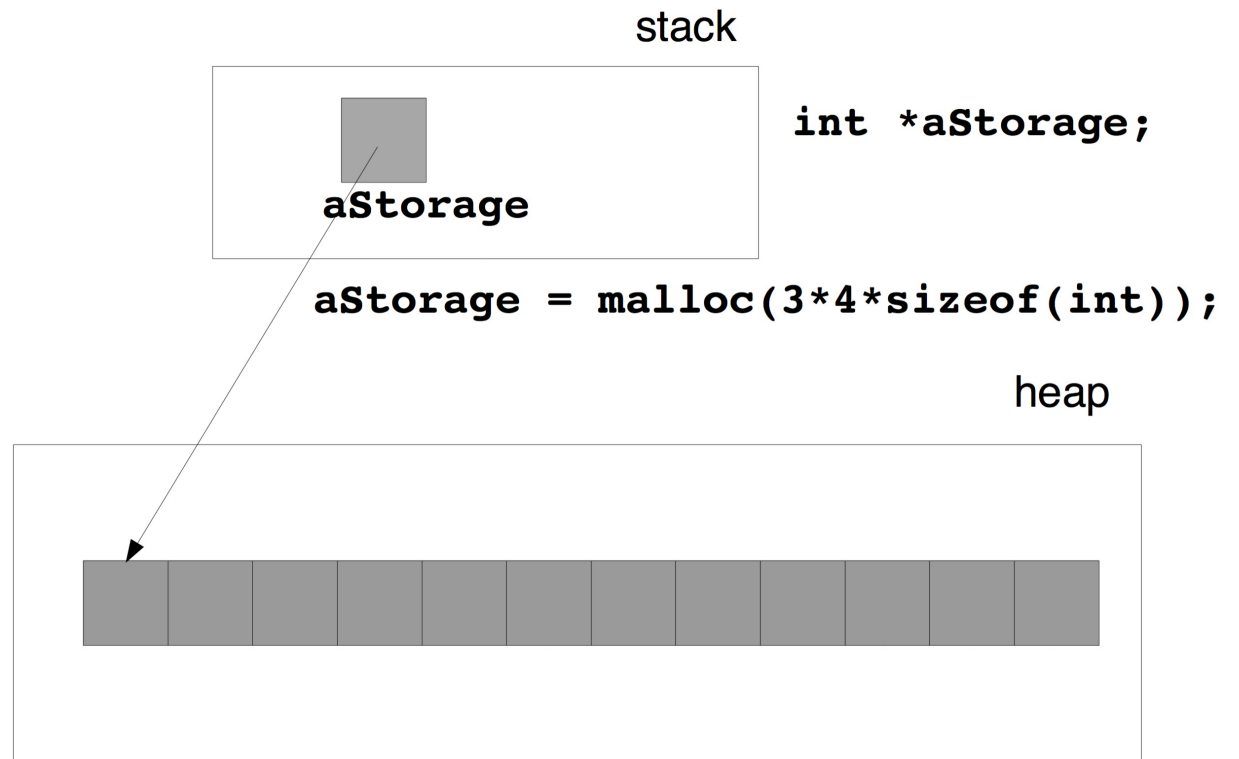
Each pointer in the array holds the heap address of the allocated array of int values



## Stack/Heap Interaction II 2D Array

version 2, part 1

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int values



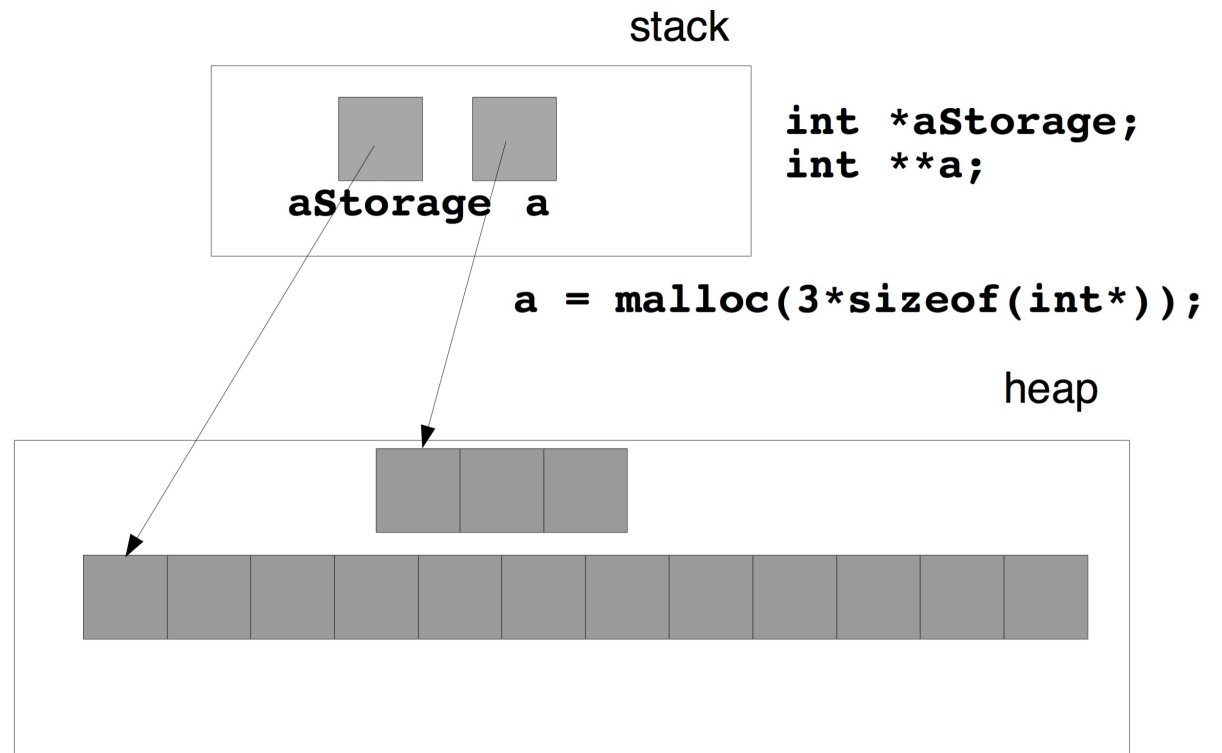


## Stack/Heap Interaction II 2D Array

version 2, part 2

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int values

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int pointers



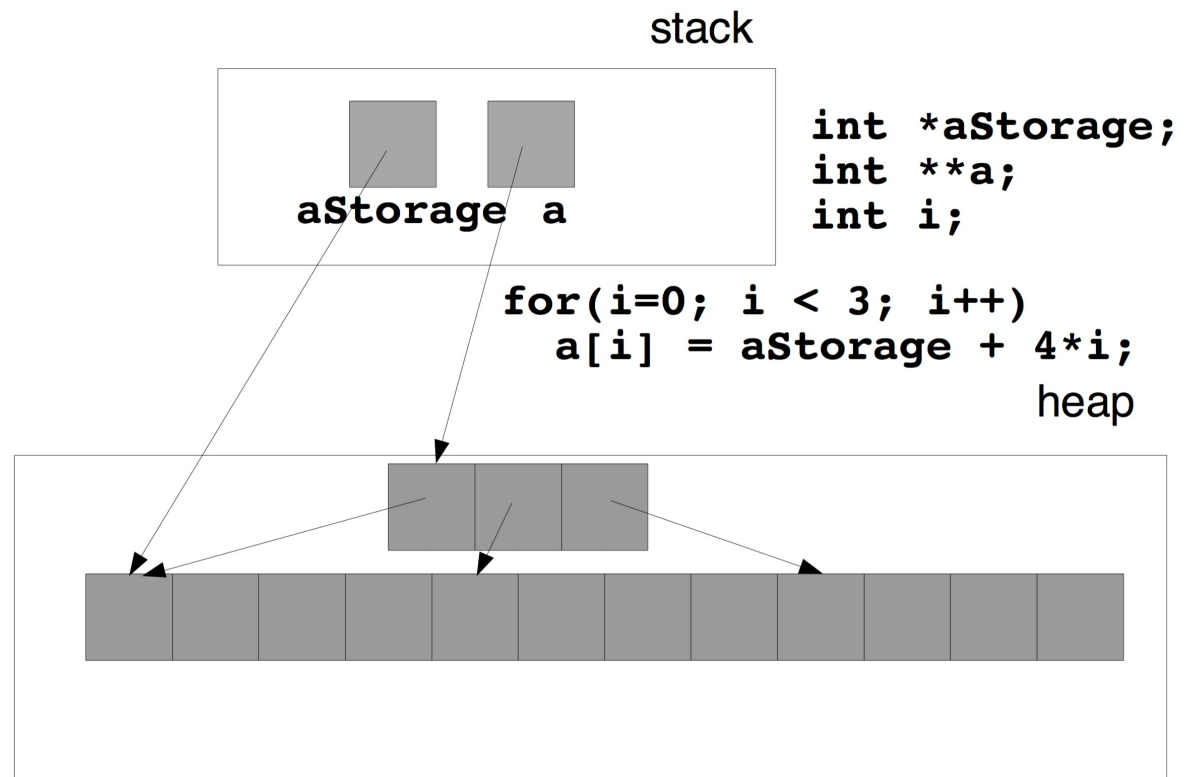
## Stack/Heap Interaction II 2D Array

### version 2, part 3

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int values

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int pointers

Each pointer in the array holds the heap address of the allocated array of int values



## Stack/Heap Interaction II 2D Array

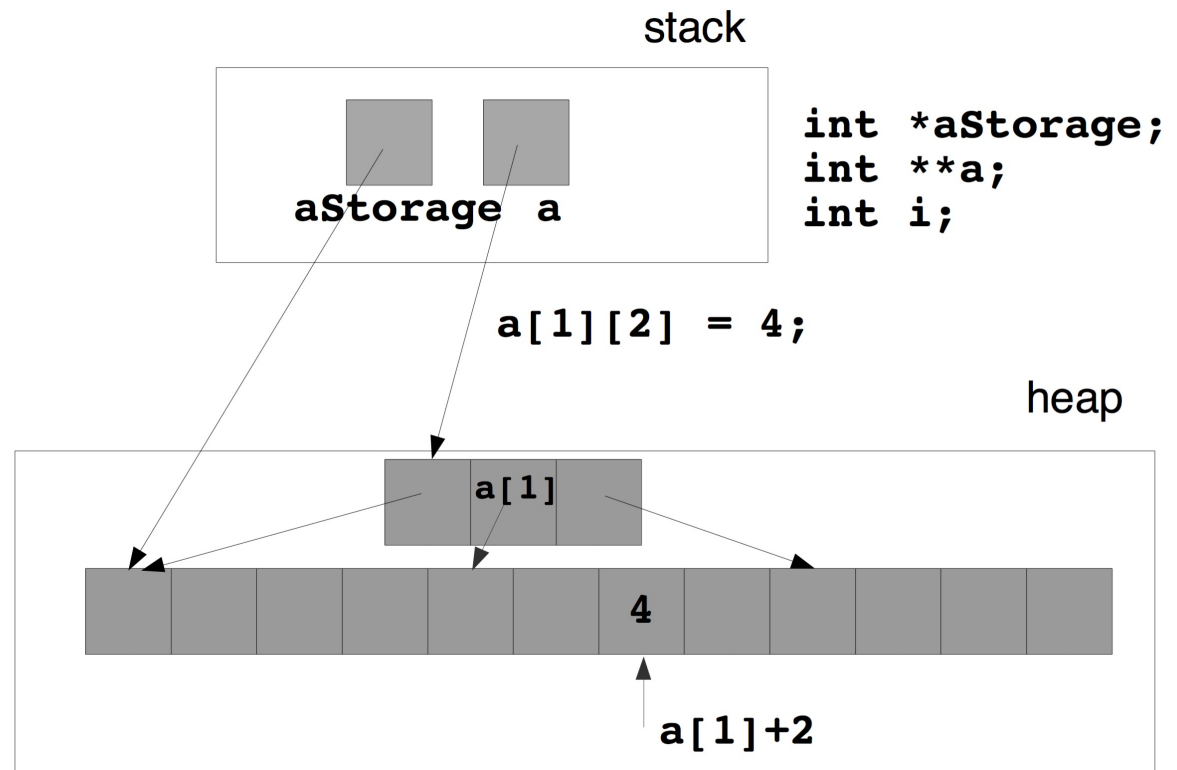
version 2, part 4

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int values

Pointer is declared on the stack, holds the heap address for the start of the allocated array of int pointers

Each pointer in the array holds the heap address of the allocated array of int values

Note the identification of a value.



# Common Heap Memory Pitfalls I

- Moving the “anchor pointer” that the memory was allocated to.

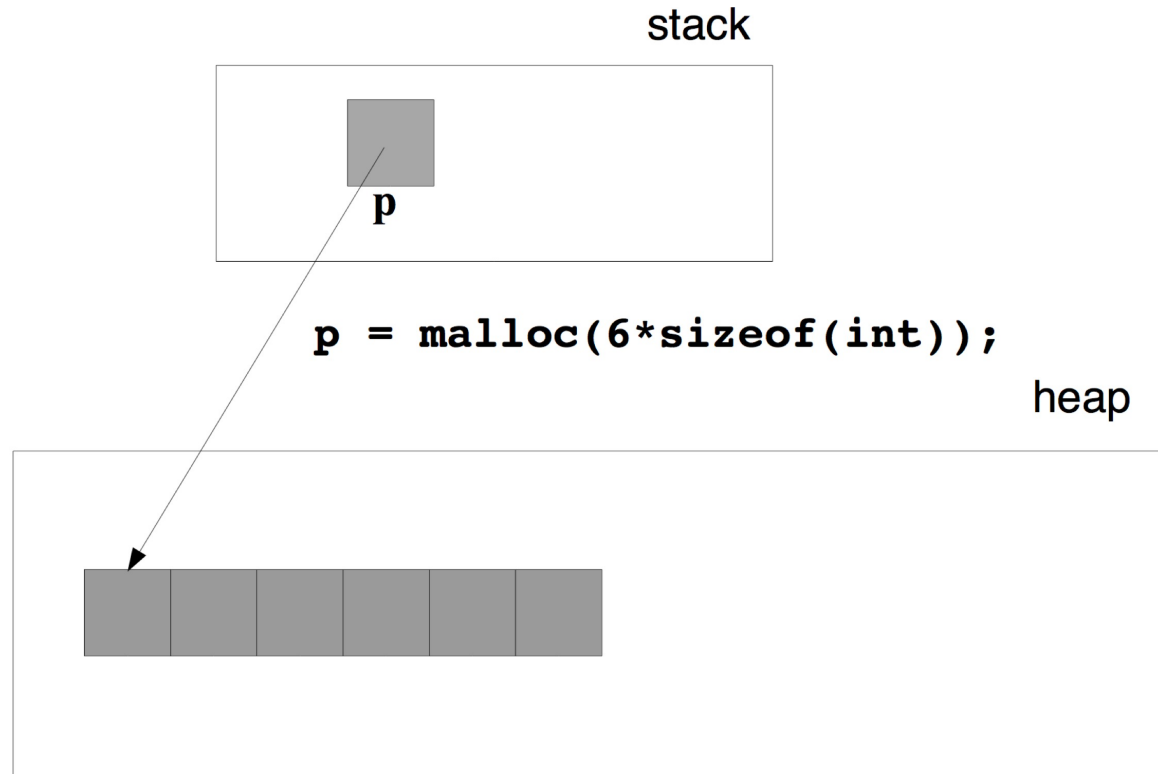
```
#define MAXLEN 10

int main(int argc, char* argv){

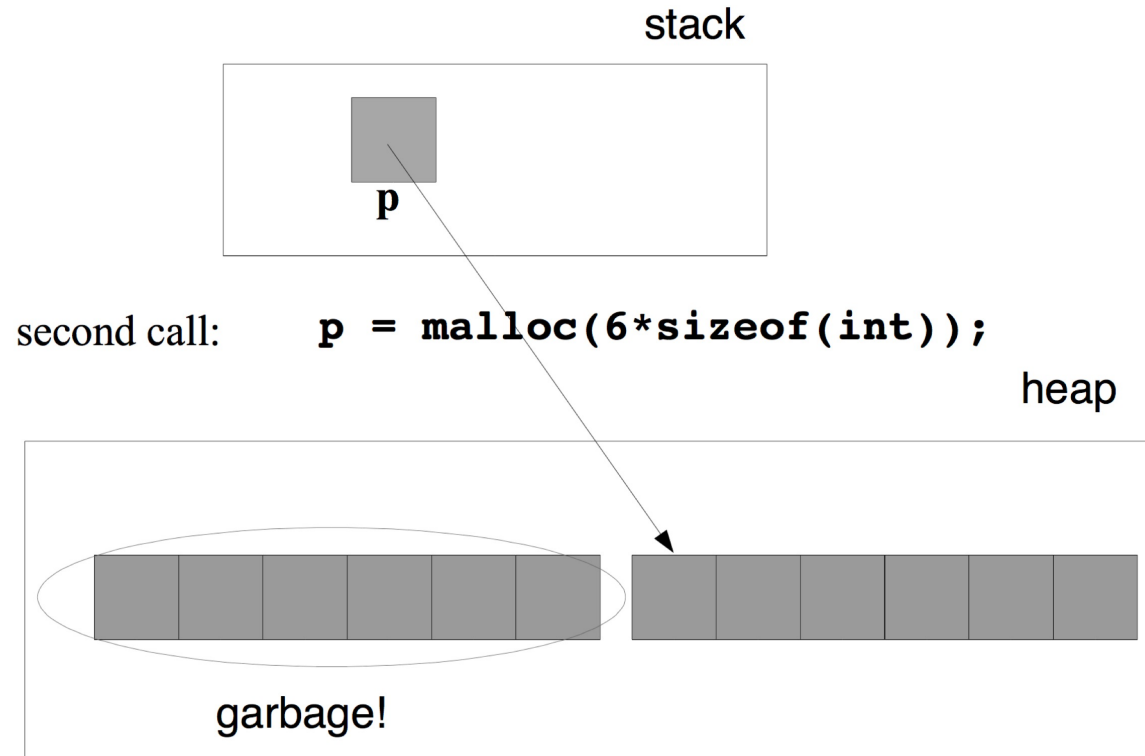
    char* s = (char*)malloc(sizeof(char)*(MAXLEN+1) );
    // stuff here
    while(*s != (char)NULL){
        printf("%c", *s);
        s++;
    }
    printf("\n");
    // So where is the beginning of the string held?
    return EXIT_SUCCESS;
}
```



# Common Heap Memory Pitfalls I



# Common Heap Memory Pitfalls I



# Common Heap Memory Pitfalls II

- Losing the pointer to allocated memory
  - memory leak

```
#define MAXLEN 10

char* mallocString(unsigned int stringLength);

int main(int argc, char* argv[]){
    char* s;
    for(int i=0; i< argc; i++){
        // heap memory previously pointed to by s is about to be lost!
        s = mallocString(strlen(argv[i]) );
        if(s==(char*)NULL){
            fprintf(stderr, "Memory failure\n");
            return EXIT_FAILURE;
        }
        strcpy(s,argv[i]);
        printf("%s\n", s);
    }
    return EXIT_SUCCESS;
} //End main()
char* mallocString(unsigned int stringLength){
    return (char*)malloc( (stringLength+1) * sizeof(char));
}
```



# Common Heap Memory Pitfalls III

- Using memory after it's been deallocated.

```
#define MAXLEN 10

char* mallocString(unsigned int stringLength);

int main(int argc, char* argv[]){
    char* s;
    for(int i=0; i< argc; i++){
        // heap memory previously pointed to by s is about to be lost!
        s = mallocString(strlen(argv[i]) );
        if(s==(char*)NULL){
            fprintf(stderr, "Memory failure\n");
            return EXIT_FAILURE;
        }
        free(s);
        strcpy(s,argv[i]);
        printf("%s\n", s);
    }
    return EXIT_SUCCESS;
} //End main()

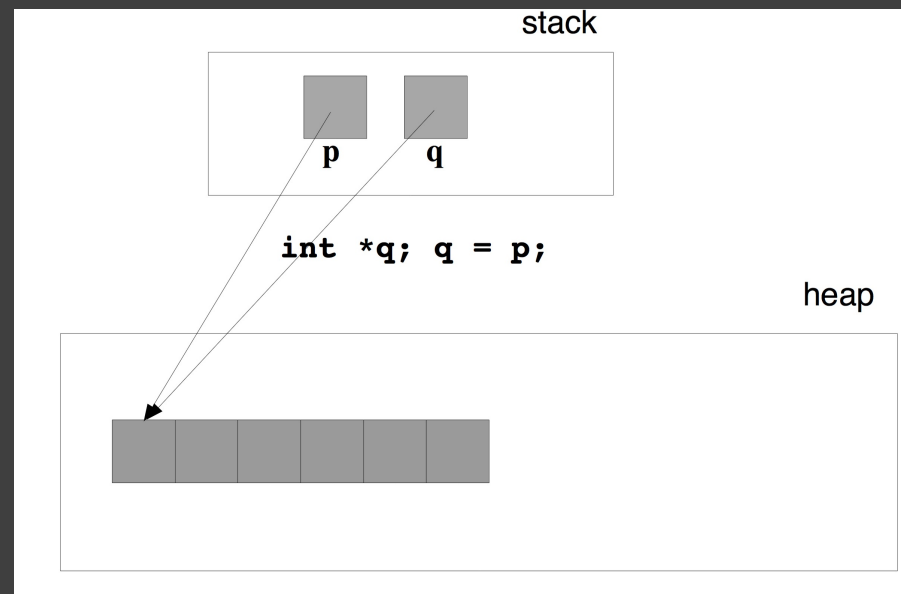
char* mallocString(unsigned int stringLength){
    return (char*)malloc( (stringLength+1) * sizeof(char));
}
```





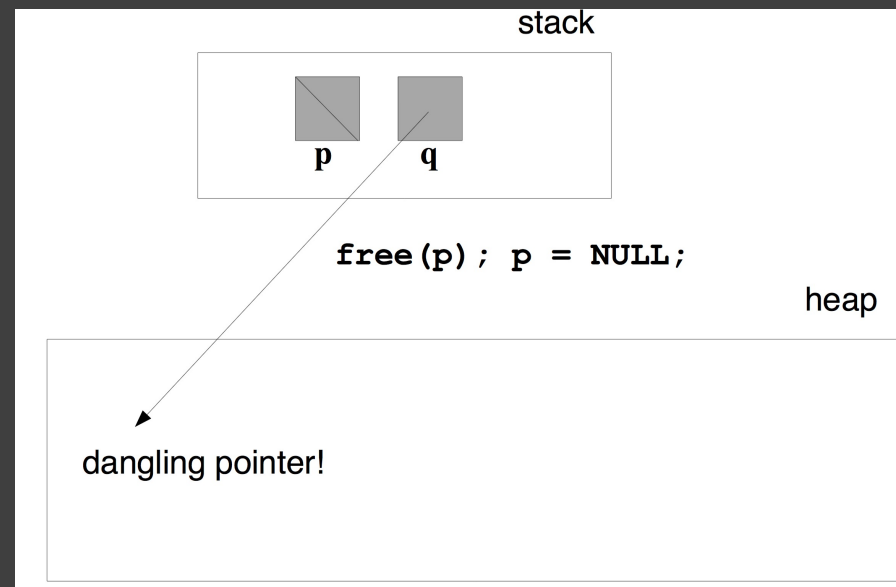
# Common Heap Memory Pitfalls IV

## Shallow Copy, part 1



# Common Heap Memory Pitfalls IV

## Shallow Copy, part 2



Will it work? Unfortunately, it just might. At least until a `malloc()` request assigns the memory again and it gets overwritten.



## Memory: "It's Complicated"

- Copying a pointer versus copying the structure
- Copying only part of a thing
- Write functions to copy things
  - hides the complexity
  - reduces the cognitive load

