Computer Systems and Networks

ECPE 170 – Dr. Pallipuram– University of the Pacific

C Programming 2

The slides are credited to Dr. Shafer

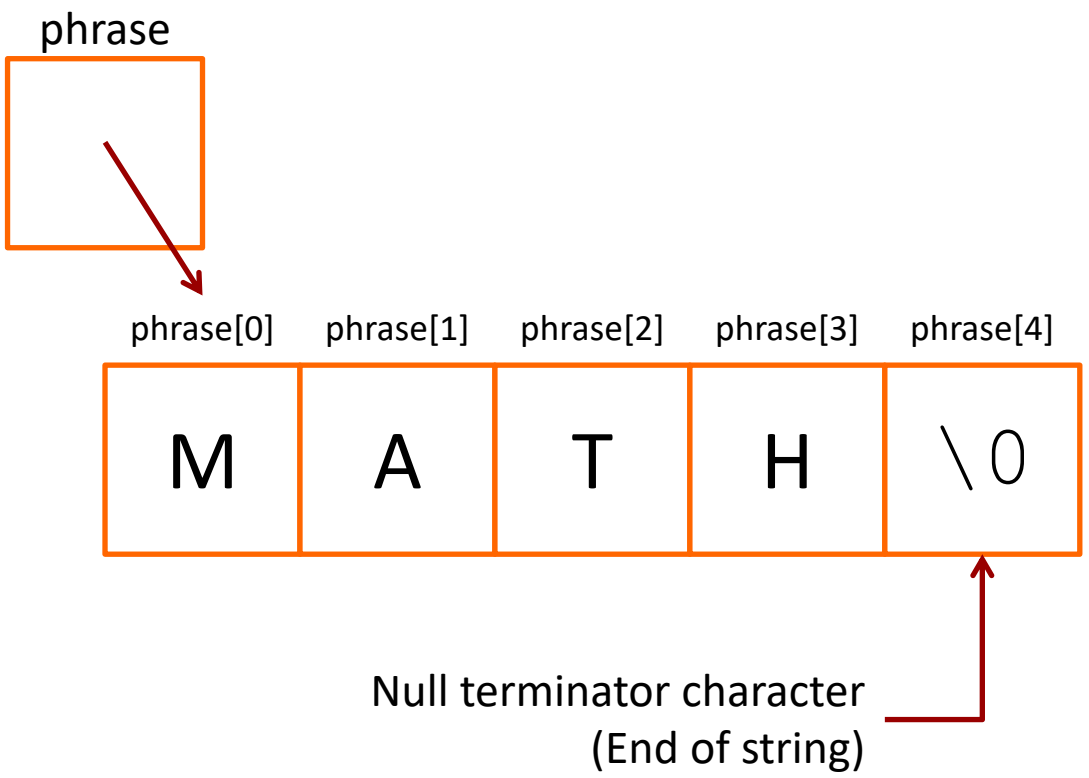
C-Strings (Arrays of Characters)

C Strings

- **There is no such thing as a “string” in C!**
- What do you get? **An array of characters**
  - Terminated by the null character `'\0'`
- Must manipulate element by element…
  - Not enough room in the array? Need a bigger array

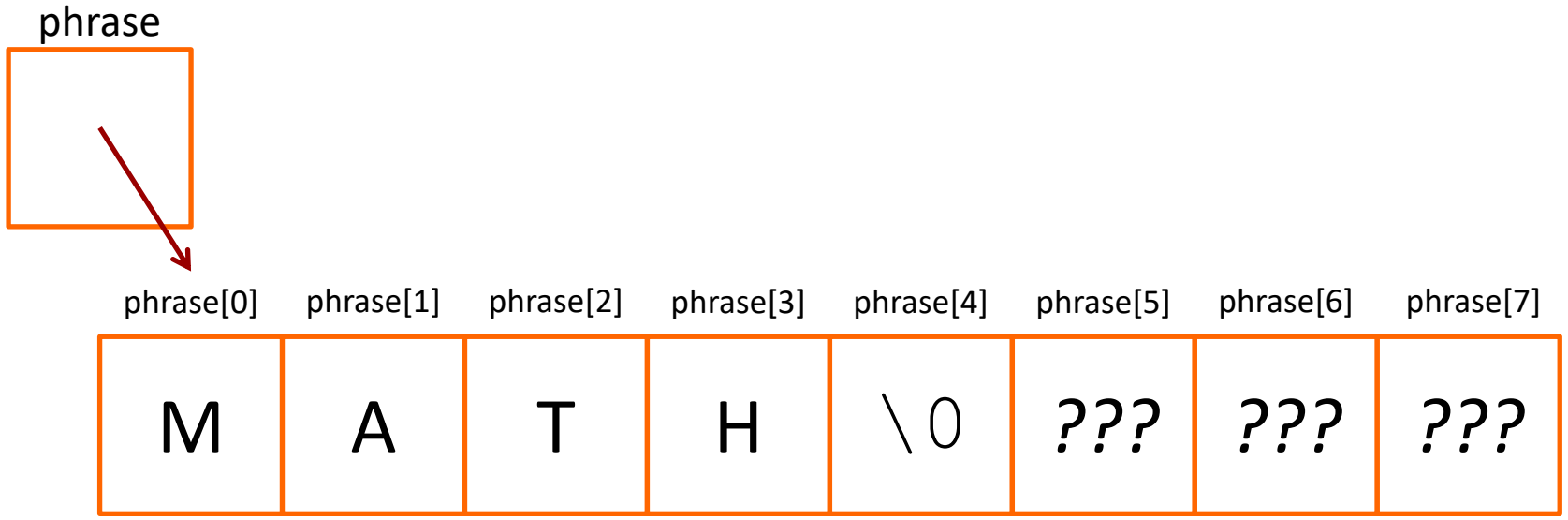
Arrays of Characters

- `char phrase[]="Math";`



Arrays of Characters

- `char phrase[8]="Math";`



`printf("%s\n", phrase);`

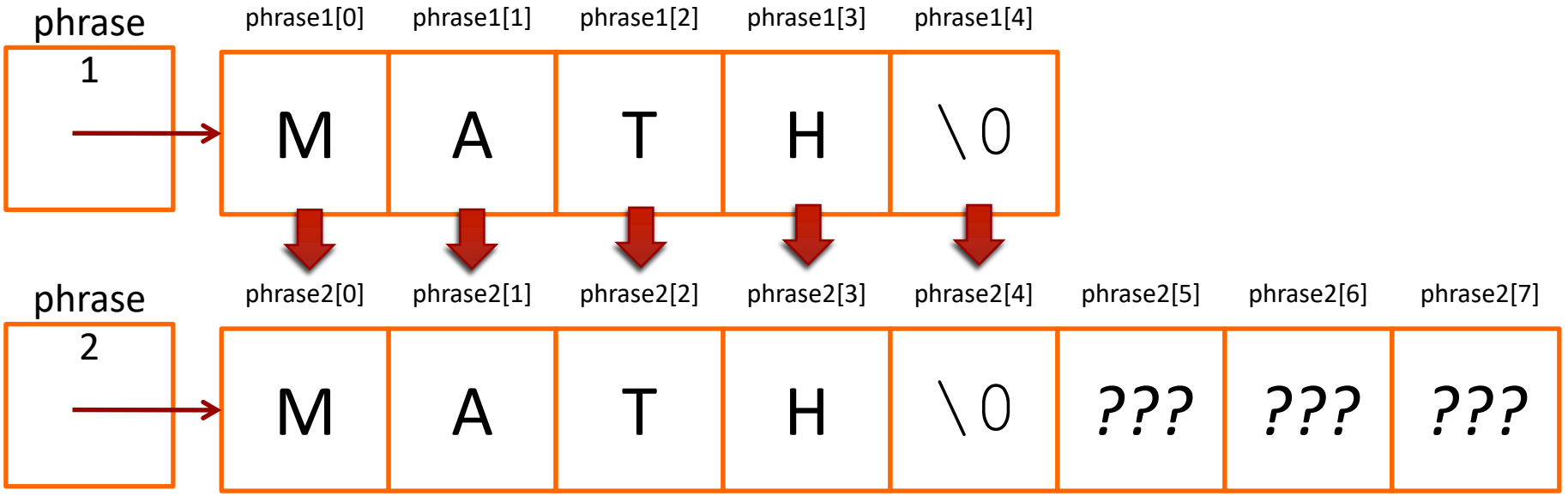
Prints until it reaches the \0 character!

Helpful Library for Character Arrays

- **`#include <string.h>`**
- Useful functions
  - `strcpy` - String copy
  - `strcmp` - String compare
  - `strlen` - String length
  - `strcat` - String concatenate

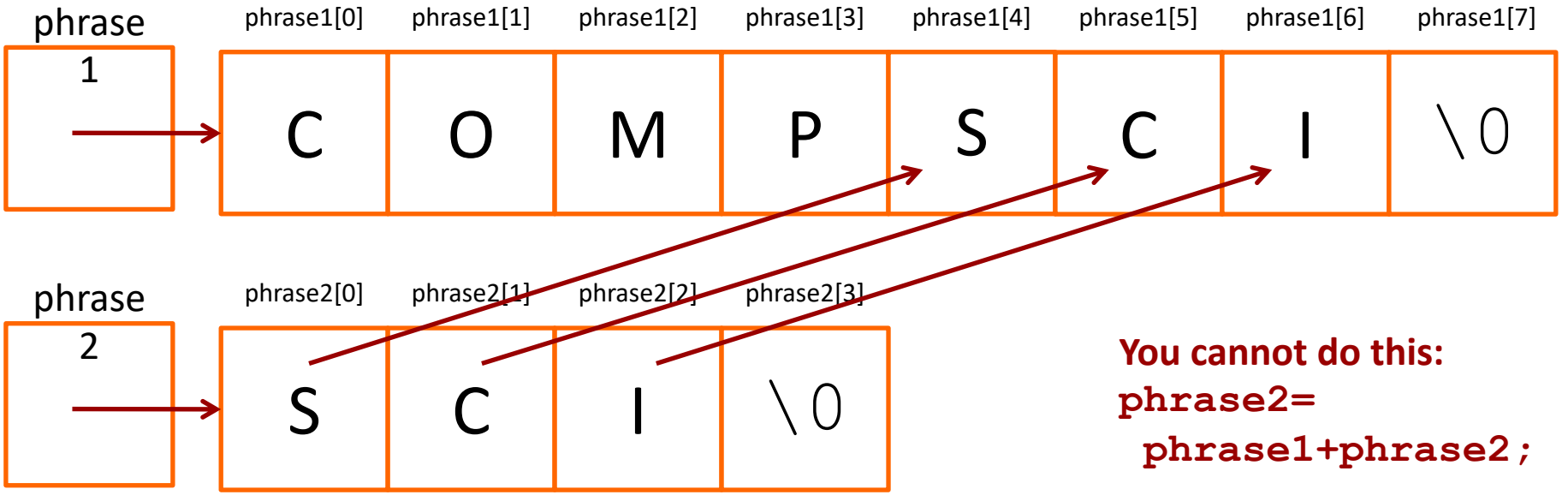
String Copy

- `char phrase1[] = "Math";`
- `char phrase2[8];`
- **`strcpy(phrase2, phrase1);`**



String Concatenation

- `char phrase1[8] = "Comp";`
- `char phrase2[] = "Sci";`
- **`strcat(phrase1, phrase2);`**



You cannot do this:

```
phrase2=
  phrase1+phrase2;
```

ctype Library

- Useful for character manipulation
- `#include <ctype.h>`
- **`toupper(char)`** / **`tolower(char)`** – Converts character to uppercase or lowercase
  - Example:

```
char c = toupper('a');
printf("%c", c);  // A
```

ctype Library

- **`isalpha(char)`** – Is the character a letter?
- **`isdigit(char)`** – Is the character a number 0-9?
- **`isspace(char)`** – Is the character whitespace? (space or newline character)
- **`ispunct(char)`** – Is the character punctuation? (technically, a visible character that is not whitespace, a letter, or a number)
- … and several other variations

File I/O

File I/O Functions

- fopen – opens a text file
- fclose – closes a text file
- feof – test for end-of-file
- fgets – reads a string from a file, stopping at EOF or newline
- fwrite – writes array of characters to a file
- fgetc – reads a character from a file
- fputc – prints a character to a file

```
#include <stdio.h>

int main()
{
      FILE *ptr_file;
      char buf[1000];

      ptr_file = fopen("input.txt","r");
      if (!ptr_file)
         return 1;

      while (fgets(buf,1000, ptr_file)!= NULL)
         printf("%s", buf);

      fclose(ptr_file);
      return 0;
   }
```

Pointer Arithmetic

- Only addition and subtraction are allowed with pointers
- All pointers increase and decrease by the length of the data-type they point to
- Example
  - If an integer pointer, `iptr` holds address 32, then after the expression `iptr++`, `iptr` will hold 36 (assuming integer is 4 bytes).

Problem 1

The name of the array is actually a pointer pointing to the first element of the array.

Subscript	[0]	[1]	[2]	[3]	[4]
Value	5	6	4	8	2
Address	65528	65532	65536	65540	65544

Consider an integer array named `array`.

```
printf("\n %u:", array); //prints 65528
printf("\n %u:", array+2); //prints 65536
printf("\n %u:", *(array+1));
  //literally translates to array[1]. Prints 6
```

```
printf("\n", %u:", array+3); //prints?_____
printf("\n", %u:", *(array+3)); //prints?_____
```

P1

Pointers and Functions: Call by value vs. Call by reference

Call by value

```
main(){
    a=5,b=6;
    update(a,b);
    printf("%d",a);
}

update(int a, int b) {
    a=a-b;
}
```

These are just copies. No change to original variables

Call by reference (pointer)

```
main(){
    a=5,b=6;
    update(&a,&b);
    printf("%d",a);
}

update(int *a,int *b) {
    *a=*a-*b;
}
```

Modification to actual variable

Dynamic Memory Management

Memory Allocation with malloc()

- `#include <stdlib.h>`
- `void * malloc(int size)`
  - **Allocate** region in memory (aka “new”)
  - Argument: Size of region in bytes to allocate
  - Return value: Pointer to the region
- `void free(void * ptr)`
  - **De-allocate** region in memory (aka “delete”)
  - Argument: Pointer to the region

Memory Allocation with malloc()

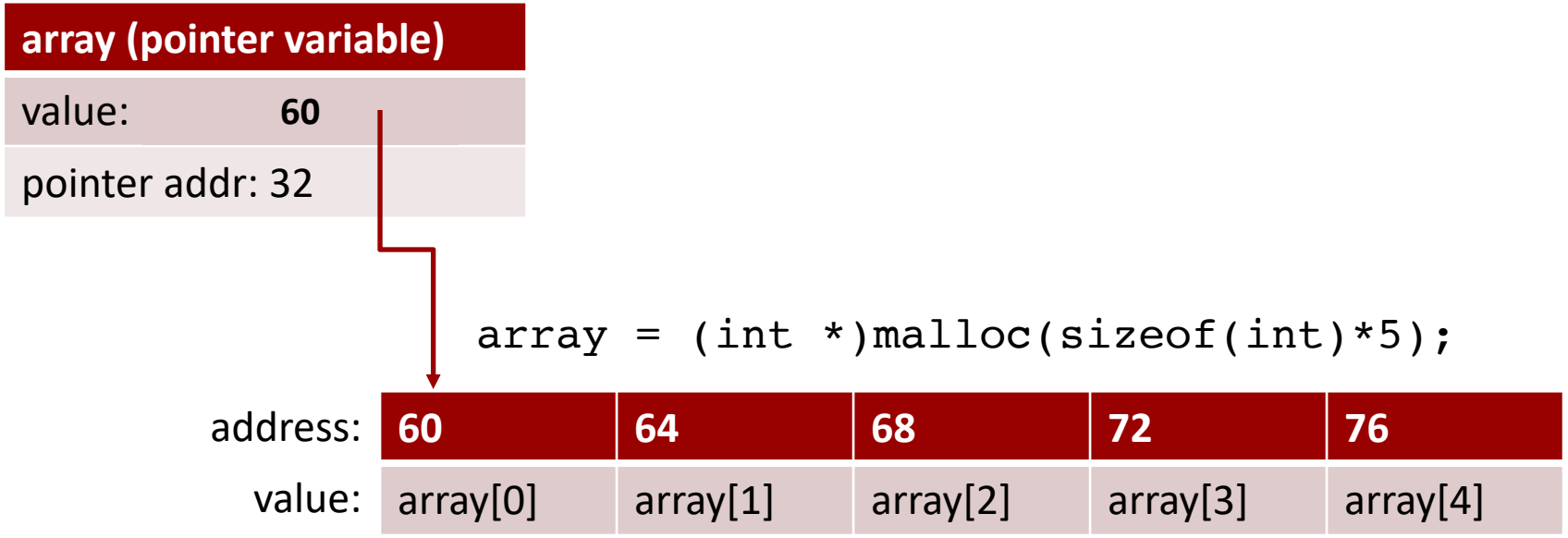
- `void * calloc(int count, int size)`
  - Basically the same as malloc!
    - Imagine you want an array of elements…
  - Argument 1: # of elements to allocate
  - Argument 2: Size of each element in bytes
  - Return value: Pointer to the region

Memory Allocation with malloc()

- `void * realloc(void *ptr, int size);`
  - **Resize** a dynamic region of memory
    - Note that it might **move** to a new address!
  - Argument: Pointer to the original region
  - Argument 2: Desired size in bytes of new region
  - Return value: Pointer to the new region
    - It might be at the same address if you made it smaller
    - It might be at a new address if you made it larger

Malloc – 1D

```
int *array;  //array of integers
```

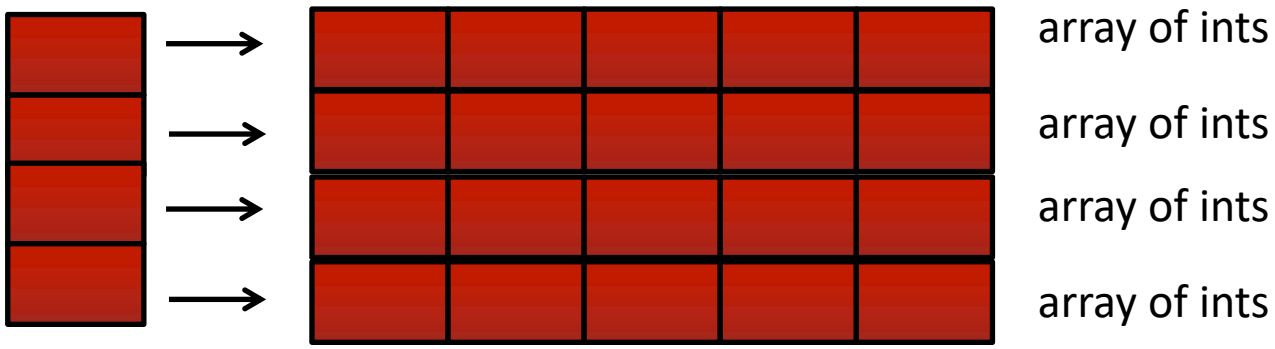


Malloc – 2D
Allocate 4x5 integers *(Hint for lab 4)*

```
int **array; //a double pointer

array = (int **)malloc(sizeof(int *)*4);

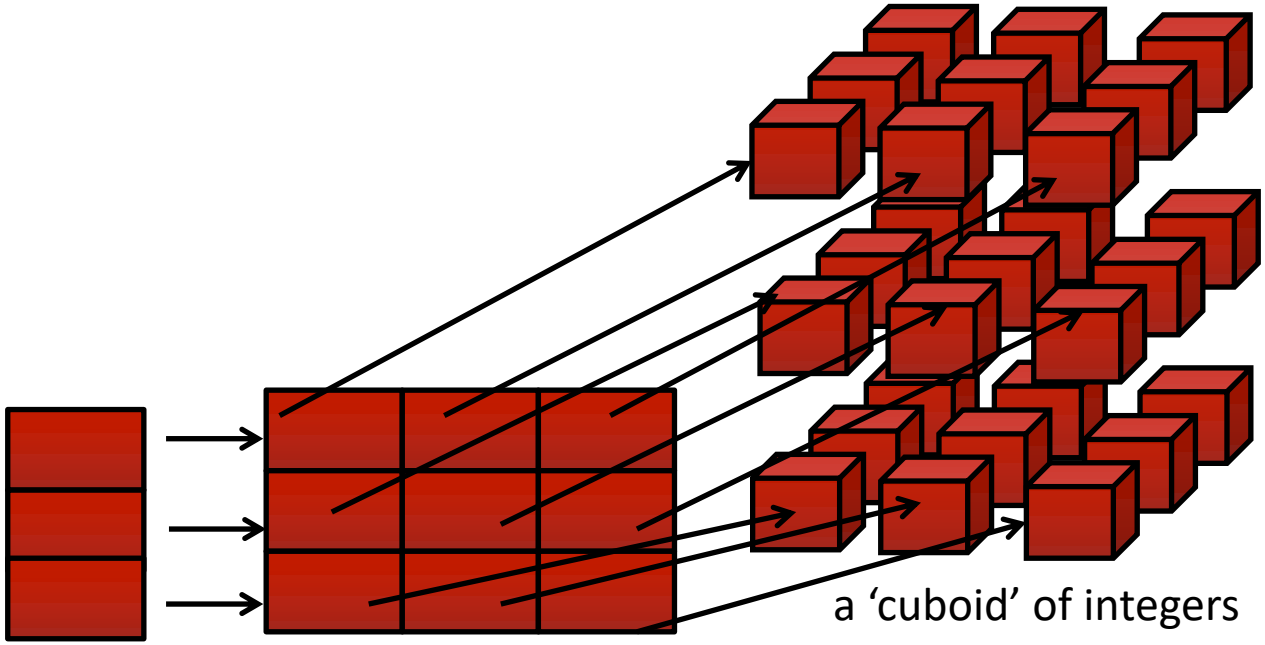
for(i=0;i<4;i++)
  array[i] = (int *)malloc(sizeof(int)*5);
```



an array of integer pointers

Malloc – 3D

```
int ***array; //a triple pointer
```



Problem 2

- Dynamically allocate space for a 3-D color image of width, w; height, h; color channel, c. Any pixel is accessed as `image[height][width][c]`.

Memory Management Internals

Memory Management

- **Who implemented `malloc()`?**
- **C Standard Library:** `#include <stdlib.h>`
- There are different C Standard Library implementations!
  - Android: Bionic
  - Apple: BSD-based / Proprietary
  - Microsoft: Proprietary C Runtime Library
  - Linux: GNU C Library (glibc) http://www.gnu.org/software/libc/

Memory Management

- **Where does the `malloc()` memory come from?**
- The **Heap**:
  - A region of memory for dynamic memory allocation
  - Per-process – each program gets its own heap
  - Managed by malloc() and related functions
  - Different from the **stack**, which is for static variables (known at compile-time)

Memory Management

- **`malloc()` outline:**

1. Call `malloc()` and request memory
2. `malloc()` checks existing heap size
   - Sufficient? Update bookkeeping to mark space as "used" and return address to your program
   - Insufficient?
     1. **Call operating system** via `brk()/nmap()` to grow the heap (plus a little extra for future requests)
     2. Update bookkeeping and return address to your program

Memory Management

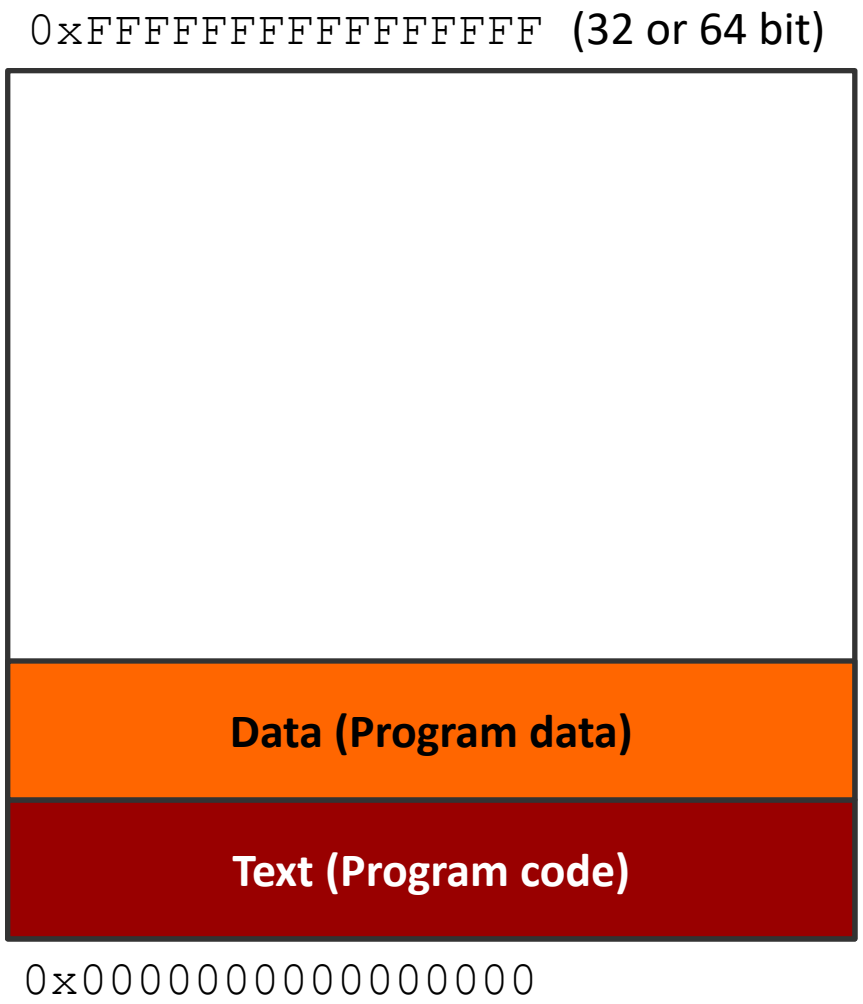
- **Why do we need to call `free()` after calling `malloc()`?**
  - Memory leak
  - `malloc()` cannot re-use that space ever, because its internal bookkeeping still thinks that region is used
  - Will only be recovered upon terminating program
    - Operating system wipes out all the memory allocated to your process (stack, heap, etc…)

Memory Management

- OS creates **virtual memory** space for process when started
- Region is huge (full 32 or 64 bit space)
  - **Not** fully mapped to physical memory
  - Otherwise you could only fit 1 program in memory

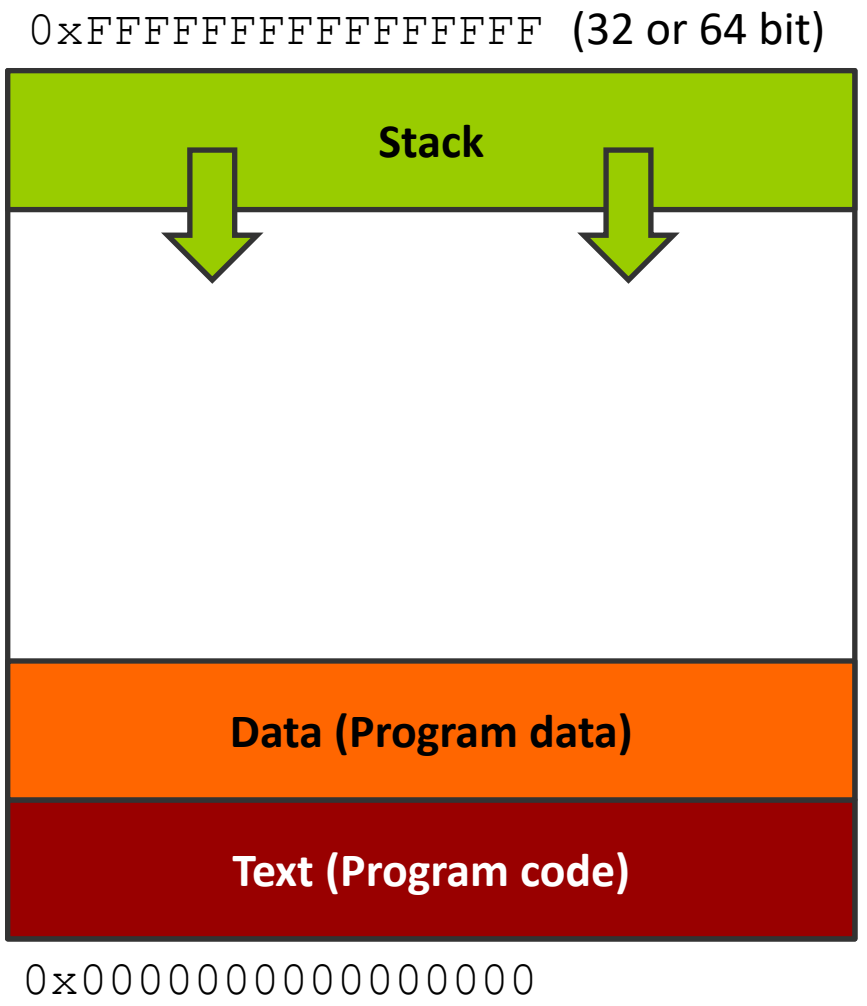
Memory Management

- OS loads in the program from disk
- "Text" region
  - Program **code**
- "Data" region
  - Program fixed **data**



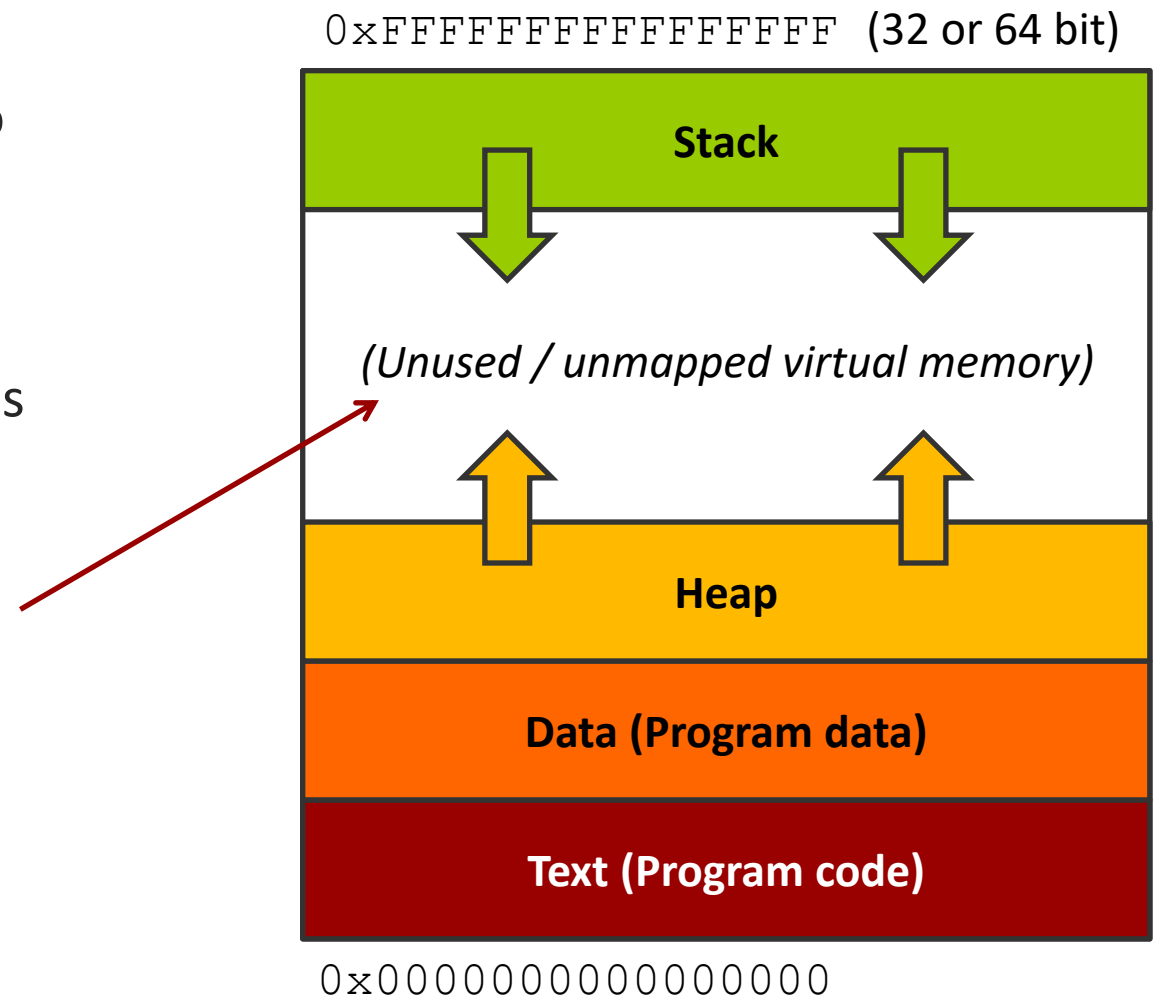
Memory Management

- **Stack** created to track program function calls and local variables



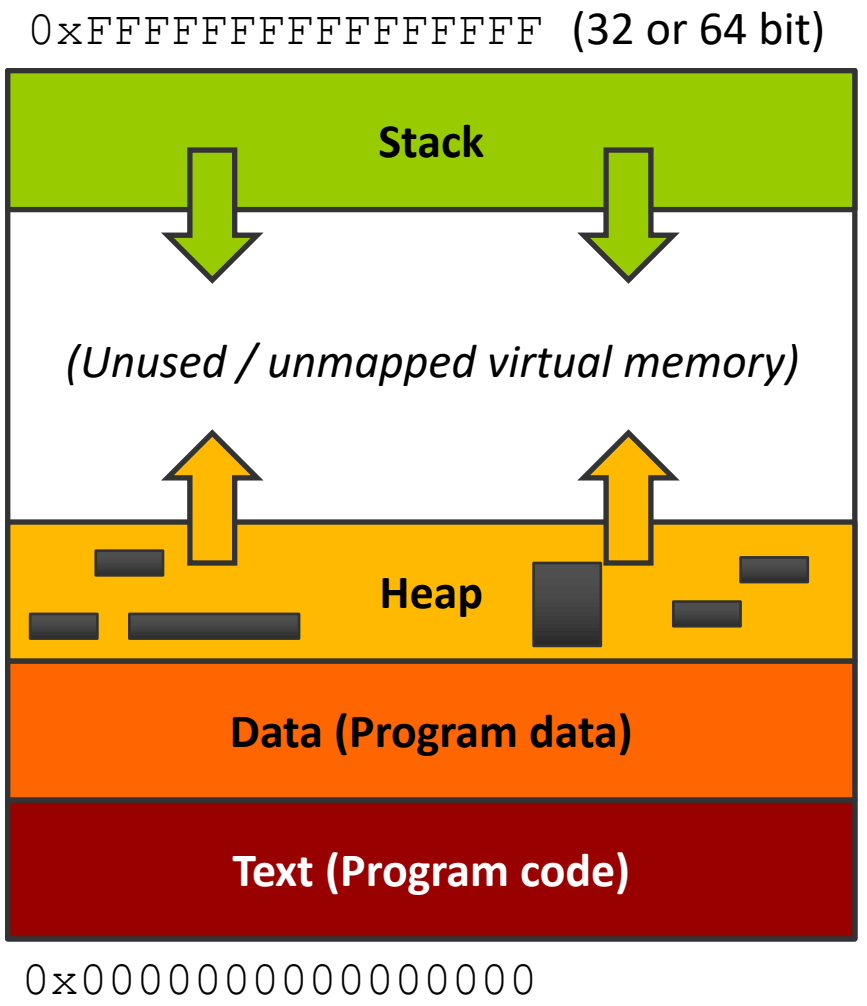
Memory Management

- **Heap** created to store dynamic memory from `malloc()` and related functions
- Not to scale – this unused region is **huge!**



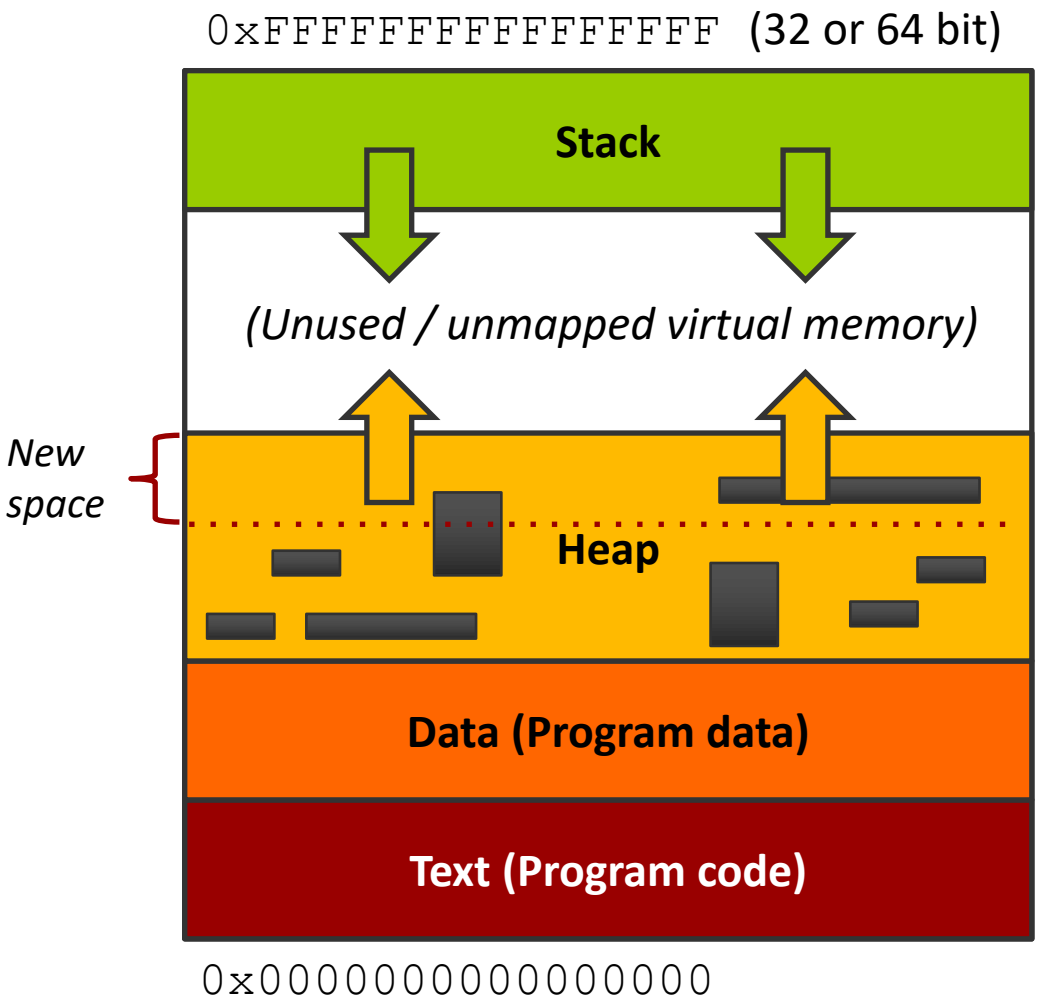
Memory Management

- Program starts running
- `malloc()` allocates some memory



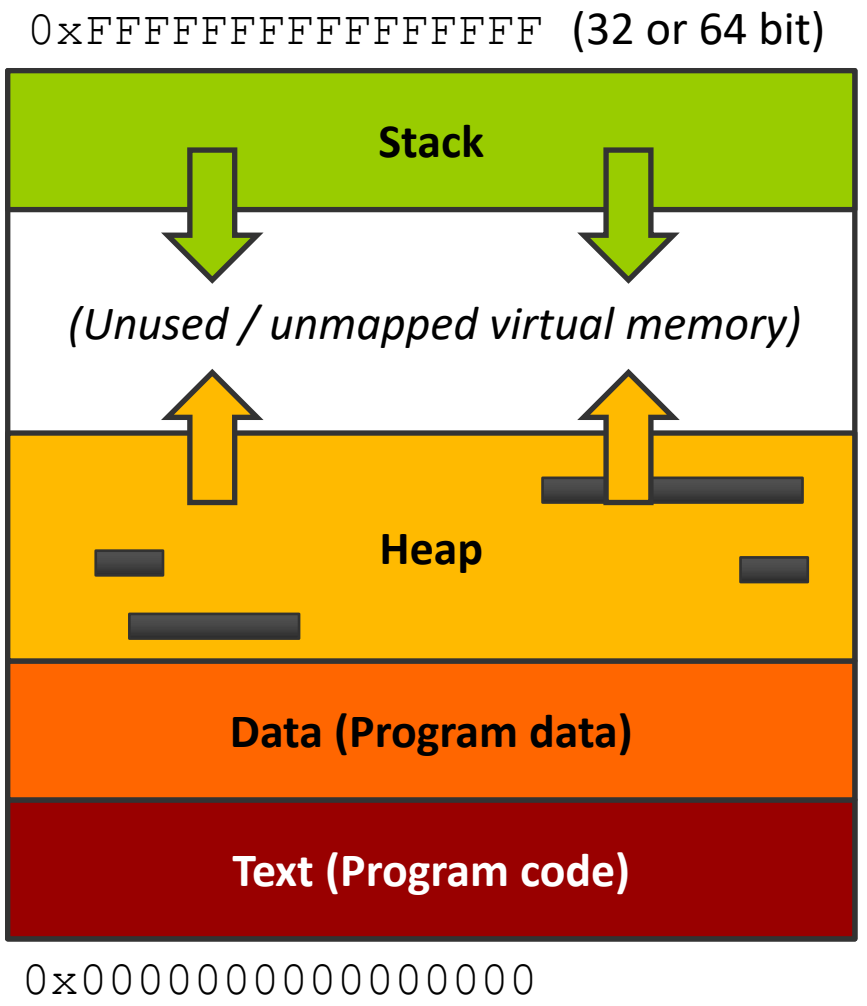
Memory Management

- Original heap space eventually fills up
- `malloc()` *requests* additional space from the kernel by using `brk()` system call



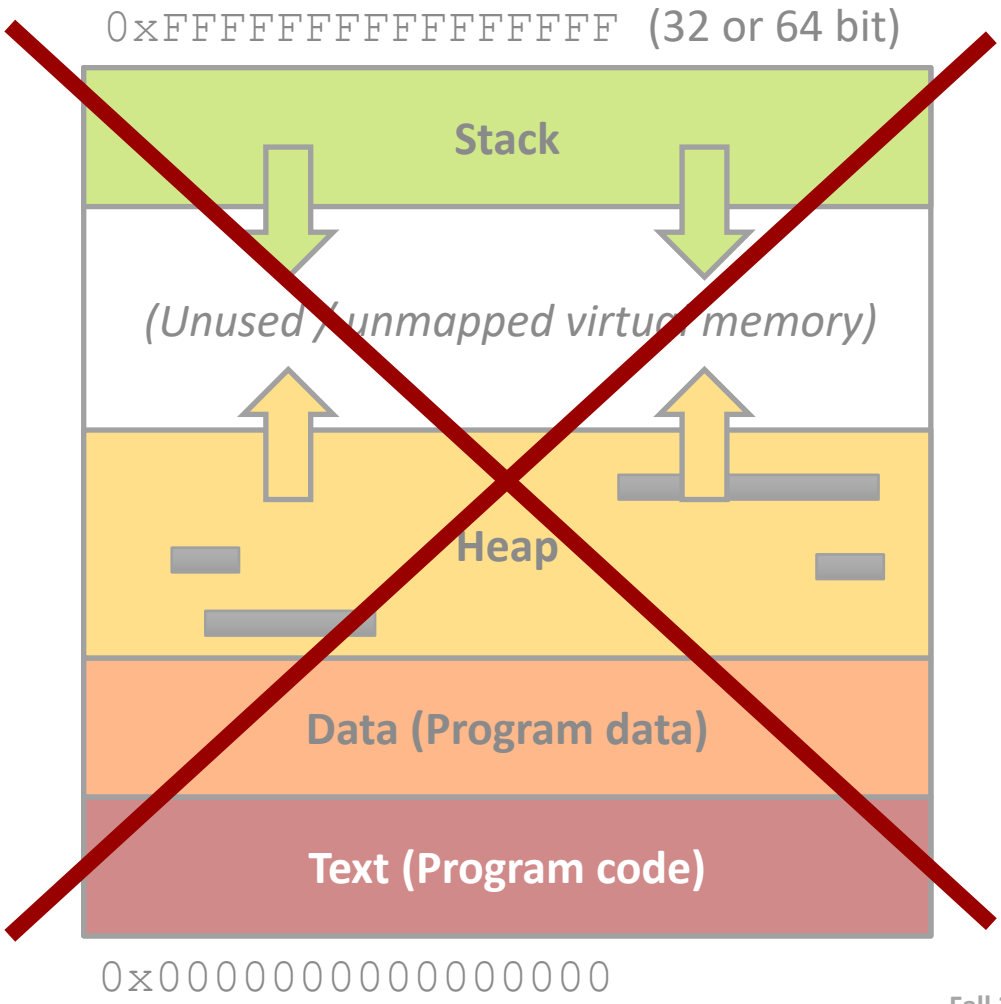
Memory Management

- `free()` deallocates blocks from the heap



Memory Management

- Program terminates
- OS expunges entire virtual address space
  - Everything is deleted



Buffer Overflow Vulnerability

- **What is a buffer overflow bug?**
  - ```
    char buf1[8]="";
    char buf2[8]="";
    strcat(buf1, "excessive");
    ```
- End up overwriting two characters beyond `buf1`!
    ```

Buffer Overflow Vulnerability

- **Why is a buffer overflow bug dangerous?**
- What is beyond my buffer in memory?
  - Other variables and data? (probably `buf2`)
  - The stack? (further out)
  - **The return address to jump to after my function finishes?**
- If app is running as administrator, attacker now has full access!

Memory Management

- **Limitless opportunities in C** for errors regarding memory ☹
  - Forgetting to `free()` some dynamic memory
  - Trying to `free()` dynamic memory more than once
  - Losing a pointer to dynamic memory (memory is "lost")
  - Accessing array elements past the end of the array
  - Mis-calculating array pointers that miss their desired target
- **Will learn a tool (*Valgrind*) in Lab 5 to analyze your program and detect / trace errors**

What's the Error?

```
char *a = malloc(128*sizeof(char));
char *b = malloc(128*sizeof(char));
b = a;
free(a);
free(b);
```

http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html

What's the (Potential) Error?

```
char *a = malloc(128*sizeof(char));

dataLen = <some value...>

// Copy "dataLen" bytes
// starting at *data to *a
memcpy(a, data, dataLen);
```

http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html

What's the Error?

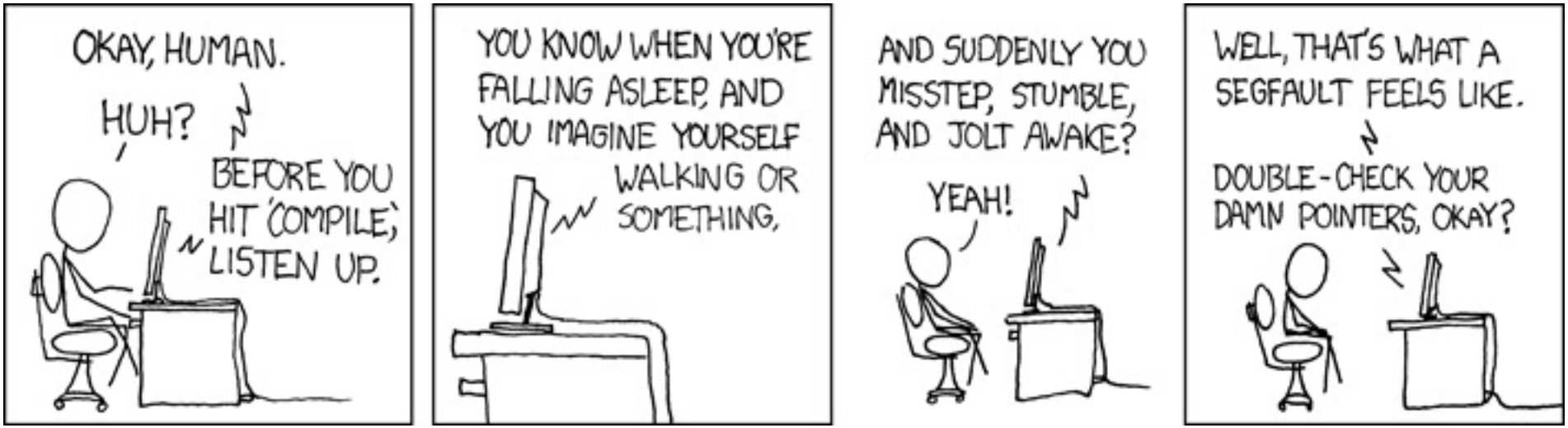
```
ptr = (char *) malloc(strlen(string_A));
strcpy(ptr, string_A);
```

http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html

What's the Error?

```
int *get_ii()
{
    int ii = 2;     // Local stack variable
    return &ii;
}
main()
{
   int *ii;
   ii = get_ii();
   ... Do stuff using ii pointer
}
```

http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html



http://xkcd.com/371/

Memory Management

- **What's a NULL pointer?**
  - Pointer value is 0x000000000
  - *Meaning* is that the pointer is not pointing anywhere
- **What happens if you dereference a NULL pointer?**
  - Telling the computer to read from (or write) to the value stored in the pointer, which is 0x000000000
  - Behavior undefined and generally unpleasant on various computer systems

Memory Management

- **"Segfault" = Segmentation Fault**
- Your program tried to read or write a *virtual memory address* that is not allowed
  - Tried to read memory outside of program bounds?
  - Tried to write read-only memory regions? (used for program data)
- "**Segmentation**" was the name of an old system (back before Intel 386 processors) used to divide physical computer memory into many virtual address regions, one per application process
  - The Segfault name stuck even though we now use **paging** to manage virtual memory

Structures

Structures

```
struct database
{
  int id_number;
  int age;
  float salary;
};

int main()
{
  struct database employee;
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

Useful way to group related variables!

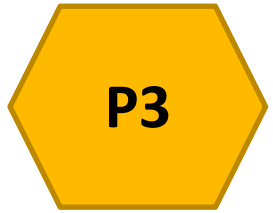
Problem

Declare a structure called **board** that contains:

- a double character pointer **matrix**
- two integer variables **height** and **width** denoting the number of rows and columns in the matrix.

Inside main, do the following:

1. Create a structure object called **myboard**, initialize **matrix** to NULL, set **height** to 7 and **width** to 7
2. Dynamically allocate **matrix** to hold **height x width** elements



Problem

Continue with the code from Problem 3.

free() is actually a reverse operation of malloc. The steps you use for free are opposite of the steps for malloc. Free the dynamically allocated 2D matrix you created in Problem 3.

You're ready to

for Lab 4!