Lecture 7 : More on Memory, Pointers and Structures

Professor Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

A Few Administrative Announcements:

Don’t Cheat.

Paid Summer Research Program in Security at Uconn:
https://khan.engr.uconn.edu/reu_site/index.html

Welcome to the

BEACH

Belonging, Equity, Affinity, Community Hangout

ITE Room 360 (follow the signs!)

CSE is committed to building a culture of **belonging and inclusion** for our students. The BEACH is a safe, **open space** dedicated to collaboration for students with the key purpose of **constructing your own community** of support in the department and at UConn.

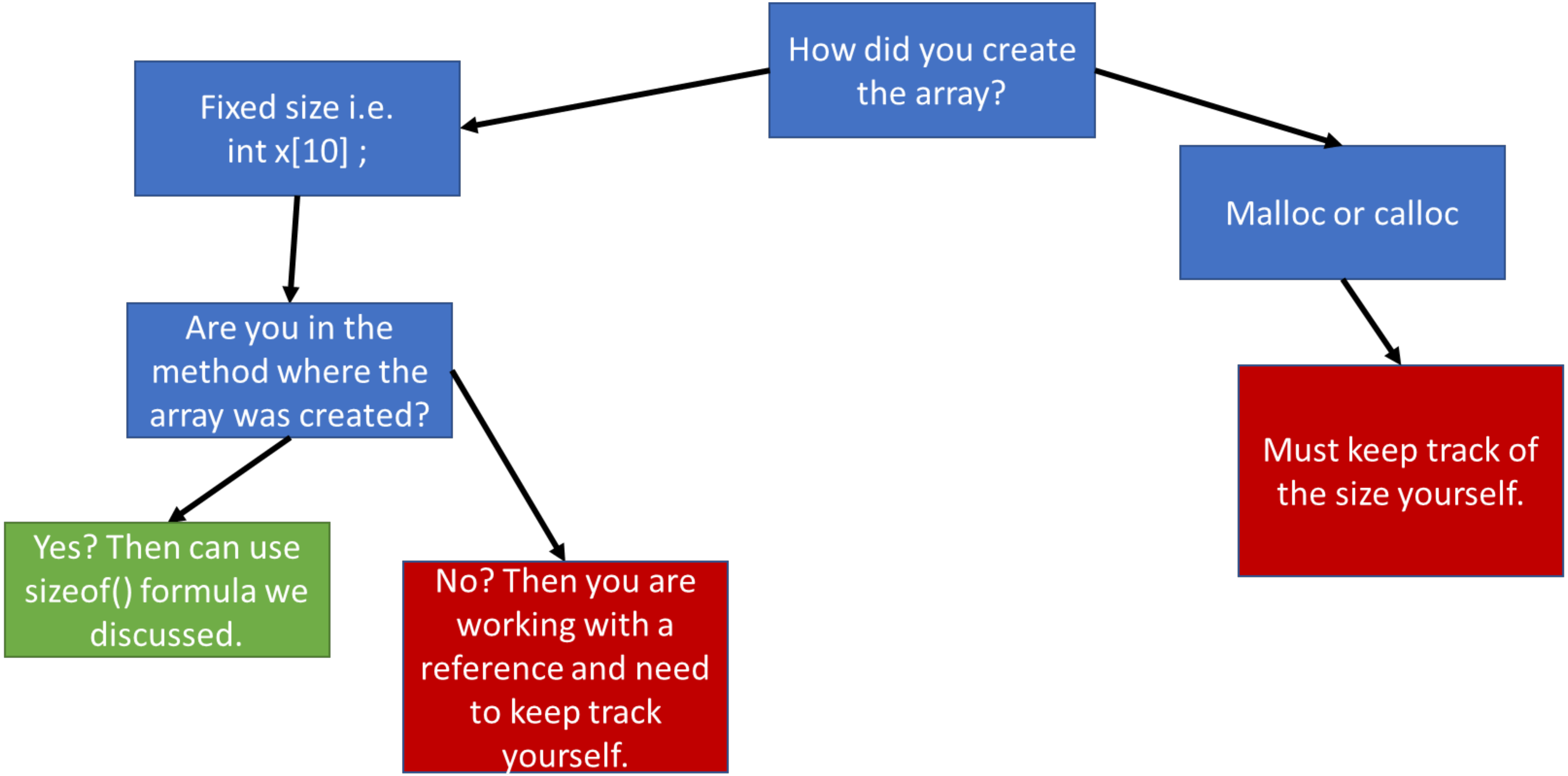
The BEACH offers the following:

- Dedicated **open space** for students to organically come together during drop-in hours, separate from “traditional” faculty or TA office hours. **Collaborate**, share research and internship advice, or **hang out** over our collection of board games!
- Head TAs, or “**Lifeguards,**” to support students and other CSE TAs.
- Affinity blocks for groups and clubs to **reserve meeting and event space.**
- **Collab Lab hours** for grads and undergrads to mingle while working and introduce UGs to research through grad student seminars and informal discussions.

UG Open Hours
Mon-Fri
4pm-8pm

Computer Science & Engineering
ITE Room 360 (follow the signs!)

Review from last lecture: Memory Allocation



Review from last lecture: Memory Allocation

- Two ways to create arrays:

1. int x[3];
2. int *x = malloc(sizeof(int) * 3)

In functions why would we want to use malloc over the "[]" approach?

In functions why would we want to use malloc over the "[]" approach?

Car Salesman: *slaps roof of heap*
"This bad boy can hold so many arrays in it"

- When we declare arrays in functions with the "[]" approach that memory is allocated from the stack.
- The stack is a FIXED size, so we could actually run out of memory in a function call if we use an especially large "[]" array.
- The heap is not a fixed size and in general larger.
- Also extremely important: we cannot take non-static stack variables outside of the function call, like we can with malloc…

Question: What will this code print out

```c
#include <stdio.h>
#include <stdlib.h>
//try to return a stack variable
int* AddThreeToArray(int* x) {
    int solution[3];
    for (int i = 0; i < 3; i++) {
        solution[i] = x[i] + 3;
    }
    return solution;
}

int main() {
    //create an array of size 3
    int* x = malloc(sizeof(int) * 3);
    x[0] = 1;
    x[1] = 2;
    x[2] = 3;
    // call AddThreeToArray
    int* solution = AddThreeToArray(x);
    //Print solution
    for (int i = 0; i < 3; i++) {
        printf("solution[%d]=%d\n", i, solution[i]);
    }
}
```

Ideally we want: the solution to be {4,5,6} right?

```
solution[0]=4
solution[1]=188
solution[2]=-858993460
```

When you try to return stack variables from functions:

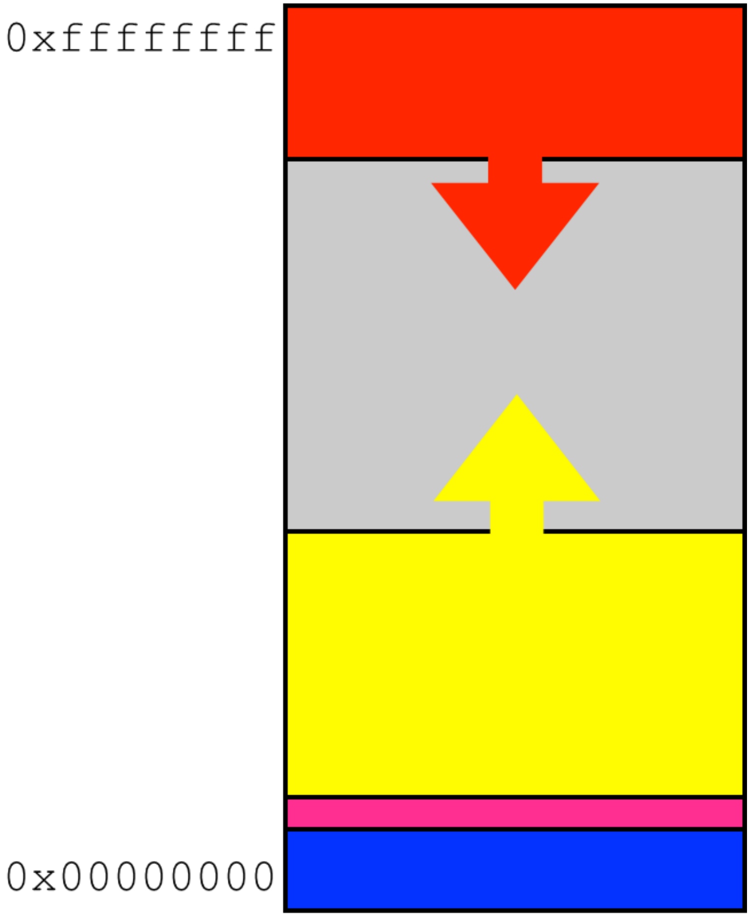
Will this code work? If so why?

```c
#include <stdio.h>
#include <stdlib.h>

int* AddThreeToArray(int* x) {
    int* solution = malloc(sizeof(int) * 3);
    for (int i = 0; i < 3; i++) {
        solution[i] = x[i] + 3;
    }
    return solution;
}

int main() {
    //create an array of size 3
    int* x = malloc(sizeof(int) * 3);
    x[0] = 1;
    x[1] = 2;
    x[2] = 3;
    // call AddThreeToArray
    int* solution = AddThreeToArray(x);
    //Print solution
    for (int i = 0; i < 3; i++) {
        printf("solution[%d]=%d\n", i, solution[i]);
    }
}
```

Answer: Yes because malloc takes memory from the heap NOT the stack. Once allocated by us, the heap memory is only freed by us.

```
solution[0]=4
solution[1]=5
solution[2]=6
```

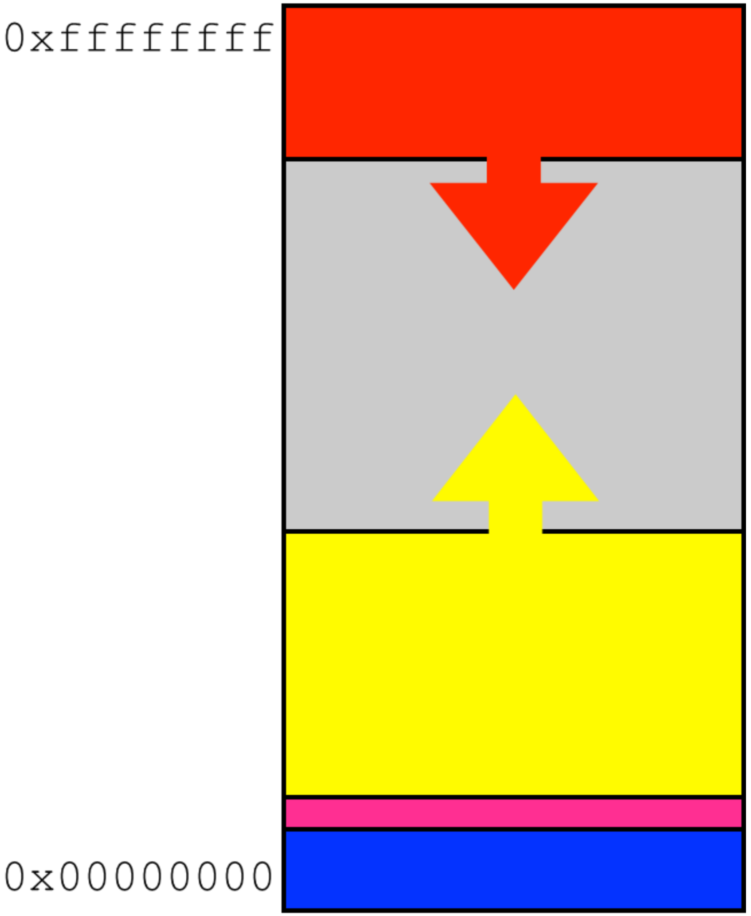
So should we just never use “[]” arrays that come from the stack?



- The answer is not that simple.
- In this class we treat memory as one big block (with three different sections).
- So it seems like it doesn’t matter if we are taking memory from the heap or the stack as both are just pieces from the same block.
- In reality stack memory can be accessed faster than the heap. You’ll learn about this more in an operating system class probably.

Pointer Arithmetic

Pointers are addresses



- The **value** of a pointer is a **byte address**
  - Unsigned integer used to number bytes in memory
  - Range is between

    0x00000000 and 0xFFFFFFFF [32-bit]

    0x0000000000000000 and 0xFFFFFFFFFFFFFFFF [64-bit]

- If a pointer is an integer, you can do ***arithmetic***...
- To ***compute*** other addresses

Simple Pointer Example: What value is p pointing to?

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
	//create an array of size 3
	int* x = malloc(sizeof(int) * 3);
	x[0] = 7;
	x[1] = 8;
	x[2] = 9;
	//create another pointer
	int* p = x + 1;
	printf("Value at pointer p=%d\n", *p);
}
```

```
Value at pointer p=8
```

There may be some confusion…

- Recall x is an integer pointer to an address in memory.
- p = x + 1
- It seems if we add 1 to it we should get 993.
- But actually p is pointing to 996.

p

x

Address	Value
1020	
1016	
1012	
1008	
1004	
1000	9
996	8
992	7

Adding a pointer and an integer

- Add an integer to a pointer, the result is a pointer of the same type
  - It is different from regular integer addition
  - The integer is **automatically scaled** by the size of the type pointed to

- Suppose p is a pointer to type T, and k is an integer

Then both `p + k` and `k + p`

- Are valid expressions that evaluate to a pointer to type T
- Have a byte address equal to

```
(unsigned long)(address stored in p) + k *
sizeof(T)
```

Can we actually see the addresses?

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //create an array of size 3
    int* x = malloc(sizeof(int) * 3);
    x[0] = 777;
    x[1] = 888;
    x[2] = 999;
    //create another pointer
    int* p = x + 1;
    printf("Address x points to %x\n", x);
    printf("Address p points to %x\n", p);
}
```

```
Address x points to 41ee0800
Address p points to 41ee0804
```

Pointer Addition

Suppose p is a pointer to an int, and its value is 1000

p + 1 is not the next byte address.

It is the address of next item (of type int)

Address	Value		To access values
1020		p + 5	*(p+5) OR p[5]
1016		p + 4	*(p+4) OR p[4]
1012		p + 3	*(p+3) OR p[3]
1008		p + 2	*(p+2) OR p[2]
1004		p + 1	*(p+1) OR p[1]
1000		p = 1000	*p OR p[0]
996		p - 1	*(p-1) OR p[-1]
992		p - 2	*(p-2) OR p[-2]

Pointers Subtraction

- Subtract one pointer from another: both must have the same type
- The result is the number of data items between the two pointers!
  - Not the number of bytes

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
      int *p = malloc(sizeof(int)*10);
      int *last = p + 9;
      int  dist = last - p; // both are
int *
      printf("Distance is %d\n",dist);
   free(p);
   return 0;
}
```

Output

```
$ gcc
ptrsub.c
$ ./a.out
Distance is 9
$
```

Pointer comparisons

- You can also compare two pointers

  `<` `>` `<=` `>=` `!=` `==`

- Purpose
  - Check boundary conditions in arrays
  - Manually manage memory blocks
- Semantics
  - Simply based on memory layout!
  - Compare bits in pointers as unsigned integers!

Effect of casting types

- If you cast a pointer type…
  - Any subsequent pointer arithmetic will use the type you chose
- Do not want scaling? Casting a pointer to (**char** *)
  - Because sizeof(char) is 1

```
int * t;
char * p = (char *) t + 8;       // 8 is not scaled
char * q = (char *) (t + 8); // 8 is scaled
```

Arrays and Pointers 1

- Arrays and pointers can often be used interchangeably

```
int  a[10];
int *p = a;

// all of the following evaluate to the value of a[0]
*p               p[0]         *a           a[0]

// all of the following evaluate to the value of a[1]
*(p+1)           p[1]         *(a+1)       a[1]

// all of the following evaluate to the address of a[0]
// type is int *
p                &p[0]         a           &a[0]
```

"array of int" becomes "pointer to int" (array decay)

Arrays and Pointers 2

- Equivalent ways of initializing an array

```
int a[10], *p = a;  // not *p = a; it is int *p; p = a;

for(int i=0; i<10; i++) a[i] = i;  // array indexing

for(int i=0; i<10; i++) p[i] = i;  // indexing via pointer

for(int i=0; i<10; i++) *(p+i) = i; // explicit pointer arithmetic

for(i=0; i<10; i++) i[p] = i;   // obfuscated but valid C!

for(i=0; i<10; i++) *p++ = i;   // common pointer use idiom
```

```
int * tp = p;
p ++;
*tp = i;
```

Are arrays and pointers the same?

Visualization of Pointer vs Array

- In this visualization x in an array and p is a pointer.

	Address	Value
	1020	
p	1016	
	1012	992
	1008	
	1004	
x	1000	9
	996	8
	992	7

Arrays and pointers are NOT the same

```
int   a[10];
int   *p = a; // a is converted to int *

// a is still an array after &
&a  // pointer to array of 10 int's  int (*)[10]
&p   // pointer to a pointer to int   int **

// a is still an array after sizeof
sizeof(a) // 40 because a is an array of 10 int's
sizeof(p) // 8  because p is a pointer

p++;   // can increment p
a++;   // cannot increment a; this will not compile
       // Similar to n++ vs 2++
```

Code example to further illustrate the point…

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //create array
    int a[] = { 1, 2, 3 };
    //create pointer to array
    int* p = a;
}
```

Address	Value
1020	
1016	
1012	
1008	
1004	
1000	
996	
992	

Code example to further illustrate the point…

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //create array
    int a[] = { 1, 2, 3 };
    //create pointer to array
    int* p = a;
}
```

	Address	Value
	1020	
p	1016	
	1012	
	1008	
	1004	
	1000	3
x	996	2
	992	1

Code example to further illustrate the point...

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //create array
    int a[] = { 1, 2, 3 };
    //create pointer to array
    int* p = a;
}
```

	Address	Value
	1020	
p	1016	
	1012	992
	1008	
	1004	
x	1000	3
	996	2
	992	1

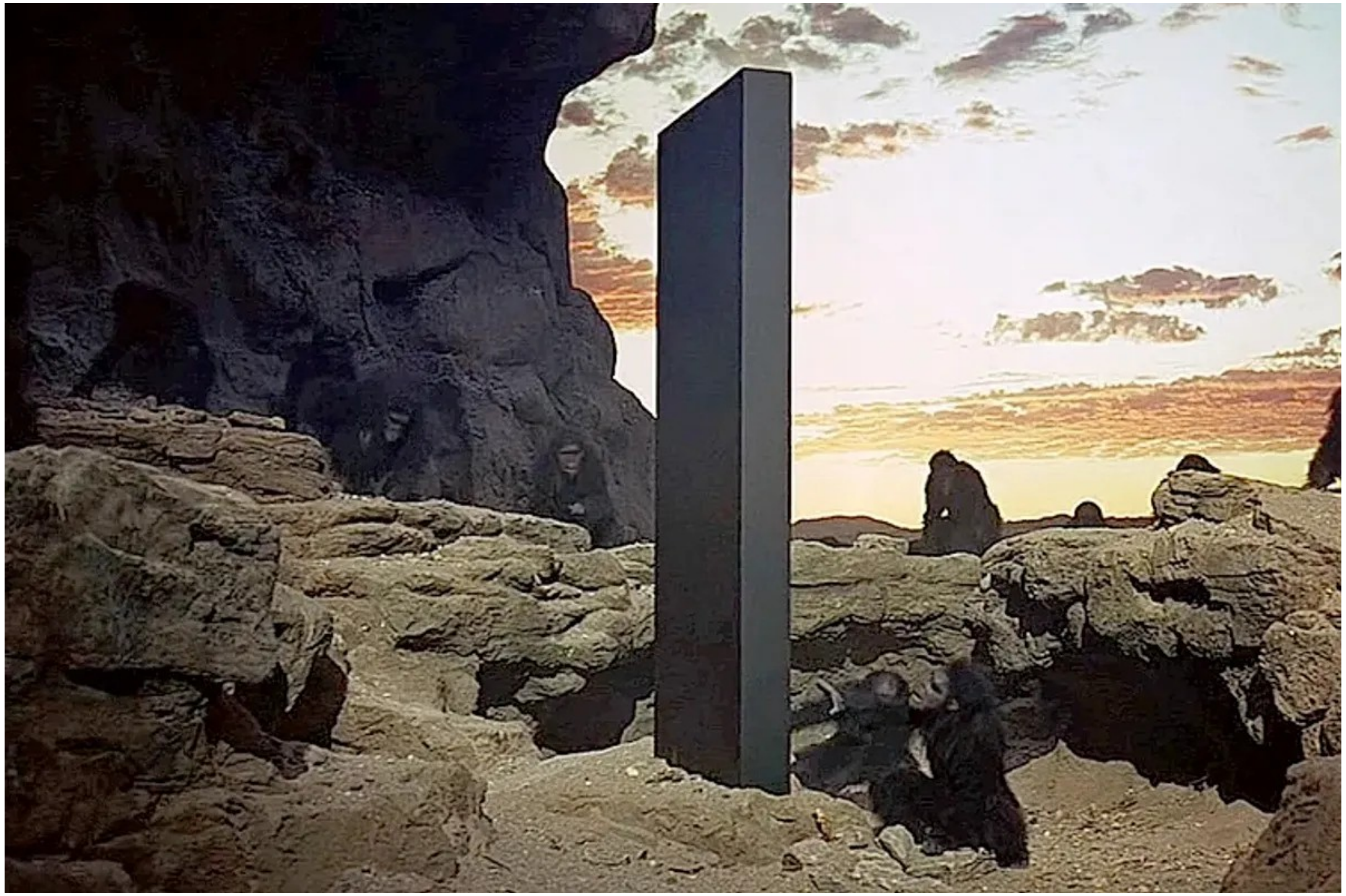
Showing the Address for Further Proof

- We can actually print out the address to see that the pointer is point to the address AND the array value is stored at that address:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //create array
    int a[] = { 1, 2, 3 };
    //create pointer to array
    int* p = a;
    printf("Address of a[0]:%x\n", &a[0]);
    printf("Address stored in pointer p:%x\n", p);
}
```

```
Address of a[0]:e9b6faa8
Address stored in pointer p:e9b6faa8
```

Structures



Structures

```
struct student_grade {
         char *name;
         int  id;
         char grade[3];
};
…
struct student_grade student1;
struct student_grade student2,
student3;
struct student_grade all[200];
```

- Mechanism to define new types
  - Also known as “compound types”
  - Used to aggregate related variables of different types
- Structures type declaration
  - Structures can have a **type name**
  - Can have “members” of any type
    - Basic types
    - Pointers
    - Arrays
    - Other structures
- Structure variable definition
  - Specifies **variable name**

Structure Example

```c
struct Person {
   int    age;
   char gender;
};

int main(){
   struct Person p;

   p.age = 44;
   p.gender = 'M';

   struct Person q = {44, 'M'};

   return 0;
}
```

Structure *type declaration*

Structure *variable definition*

Syntax for field access similar to Java and Python

Structure *variable definition* and *initialization*

Example: Array of Structures

```c
#include <stdlib.h>

struct Person {
   char   name[32];
   int    age;
   char   gender;
};

int main()
{
   struct Person family[4] = {
      {"Alice",34,'F'},
      {"Bob",40,'M'},
      {"Charles",15,'M'},
      {"David",13,'M'}
   };
   int juniorAge = family[3].age;
   return 0;
}
```

- Member name is a char array
- Caveats
  - Names cannot be more than 31 characters long
  - Four persons in family
    - Indexed 0..3
- Array of structures for the whole family
  - Nested initializers

typedef

```c
struct Person {
   char    name[32];
   int     age;
   char    gender;
};
typedef struct Person TPerson;
int main()
{
    TPerson family[4];
    ...
    return 0;
}
```

- Struct names can become long
- C provide the ability to define type abbreviations
  - **typedef** declaration
    - Give existing type new type name
- Make code more readable
- Structure and typedef declarations often combined

```c
typedef struct Person {
   char    name[32];
   int     age;
   char    gender;
} TPerson;
```

Operations on struct

- Assignment
  - All struct members copied
- Can be passed to functions
  - **By value**
  - Even if some members are arrays!
- Can be returned from a function

- If pass by value, cannot change members in functions
- Passing or returning large structures can be costly

Use pointers to structures!

Structure Alignment

- Structure members are ***aligned*** for the natural types

Alignment requirements on x64 architecture	
char	1
short	2
int	4
long	8
float	4
double	8

```
struct struct_random {
    char    x[5]; // bytes 0-4
    int     y;    // bytes 8-11
    double  z;    // bytes 16-23
    char    c;    // byte 24
};      // Total size 32
```

```
struct struct_sorted {
    double  z;    // bytes 0 - 7
    int     y;    // bytes 8 - 11
    char    x[5]; // bytes 12 - 16
    char    c;    // byte  17
};       // Total size 24
```

Lecture Conclusions

- As a language C offers an extremely large amount of control over the code you program.
- This also makes it very challenging to learn and pick up.
- You don’t want to be like the meme shown on this slide.
- You should understand how to use pointers, arrays and which variables use which memory.
- Understand how to create structures in C.

Figure Sources

1. https://i.kym-cdn.com/photos/images/newsfeed/001/608/838/9f2.jpg
2. https://img.memegenerator.net/instances/62930796.jpg
3. https://i.kym-cdn.com/entries/icons/original/000/028/596/dsmGaKWMeHXe9QuJtq_ys30PNfTGnMsRuHuo_MUzGCg.jpg
4. https://mediaproxy.salon.com/width/1200/https://media.salon.com/2013/07/2001_monolith.jpg
5. https://programmerhumor.io/wp-content/uploads/2023/01/programmerhumor-io-python-memes-backend-memes-e1c8fbd8d3296f0-608x596.jpg