Lecture 16:
Memory access grab bag

CSE 29: Systems Programming and Software Tools

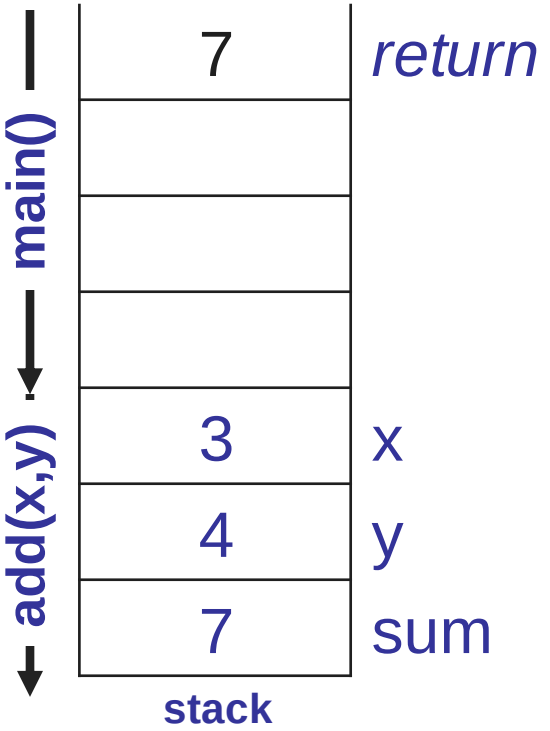
Aaron Schulman (Shalev)

UCSDCSE
Computer Science and Engineering

Review: How functions use memory

```
int add(int x, int y) {
  int sum = x + y;
  return sum;
}

int main() {
  int x = add(3,4);
}
```



What kinds of structs have we seen so far?

Statically allocated struct or array of structs (on stack)

- **struct person p[10];**
  - » Puts the array in a static-sized area on the stack when a function is called.
- *Pro:* Automatically deallocates when function ends
- *Con:* Need for struct needs to be known when programmer writes the code.

Dynamically allocated struct or array of structs (on heap)

- **struct person p_ptr* = malloc(sizeof(struct person) * 10);**
  - » Puts the array in a dynamically allocated area on the heap when malloc is called.
- *Pro:* Allows you to chose an array size during program execution.
- *Con: We can only change the size by allocating new area on the heap and copying into it.*

Many ways to access a struct

```
struct person {
  char name[100];
  unsigned int age;
};

struct person p;
p.name = “Aaron”;
p.age = 55;

struct person* p_ptr = malloc(sizeof(struct person));
p_ptr->name = “Aaron”;
p_ptr->age = 55;
```

Arrays of structs are still just arrays

```
struct person {
  char name[100];
  unsigned int age;
};

struct person people[10];
p[0].name = “Aaron”;
p[0].age = 55;

struct person* p_ptr = malloc(sizeof(struct person) * 10);
p_ptr[0].name = “Aaron”;
p_ptr[0].age = 55;
```

A useful tool when working with structs

memcpy(): Byte-by-byte copy of memory at a pointer to another pointer

```
#include <stdlib.h>


struct place p1 = {10, 1000};
struct place p2;
mempy(&p2, &p1, sizeof(struct place));
```

Functions are pointers

```
int add(int a, int b) {
  return a + b;
}
// Can point to any func like this
int (*operation)(int, int);

operation = add;
int ret = operation(1,2);
```