COMPUTER SYSTEMS AND ORGANIZATION

Function Pointers

Daniel G. Graham Ph.D

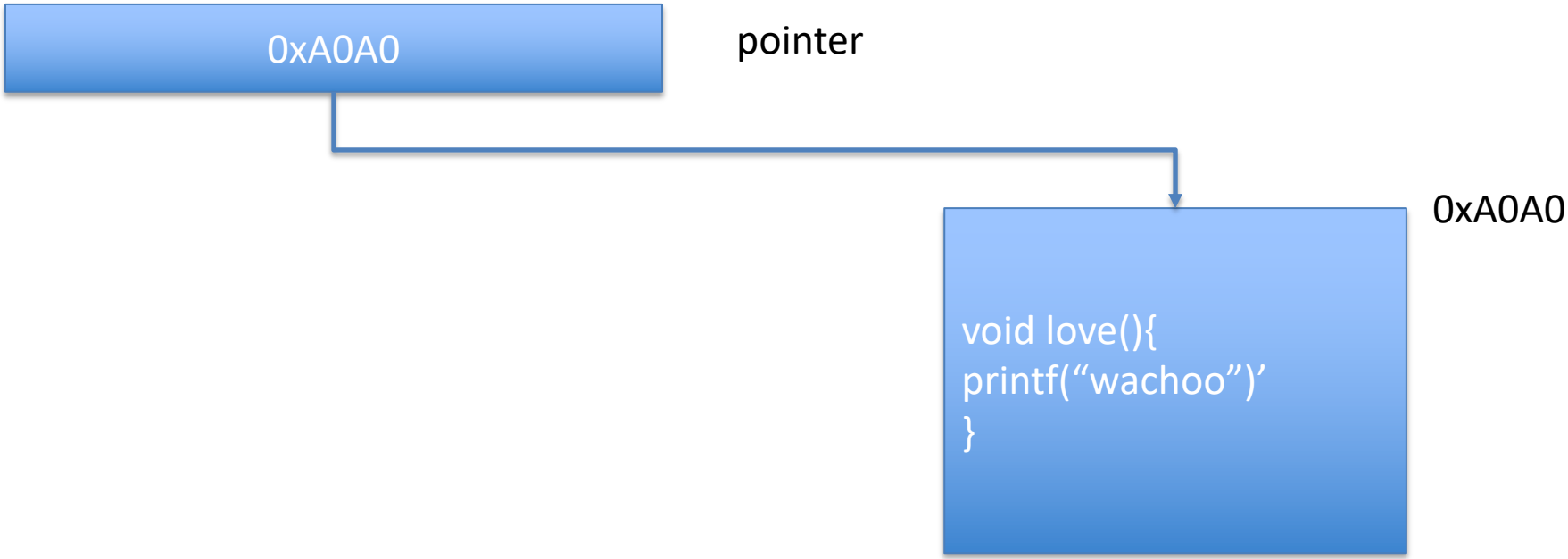
1. Warmup/Review Memory Leaks
2. Function Pointers

```
1.  // Determine if a number is odd
2.  int isOdd(int *x) {
3.      return (*x) % 2;
4.  }
5.
6.  // Sum up to the first n even numbers
7.  int sumFirstEvens(int *array, int n) {
8.      int *cpy = (int *)malloc(sizeof(n));
9.      int *sum = (int *)malloc(sizeof(int));
10.     int *cpy2 = cpy;
11.     int *sum2 = sum;
12.     for (int i = 0; i < n; i++)
13.         cpy[i] = array[i];
14.     while (!isOdd(cpy)) {
15.         *sum += *cpy;
16.         cpy += 1;
17.     }
18.     free(sum);
19.     return *sum2;
20. }
```

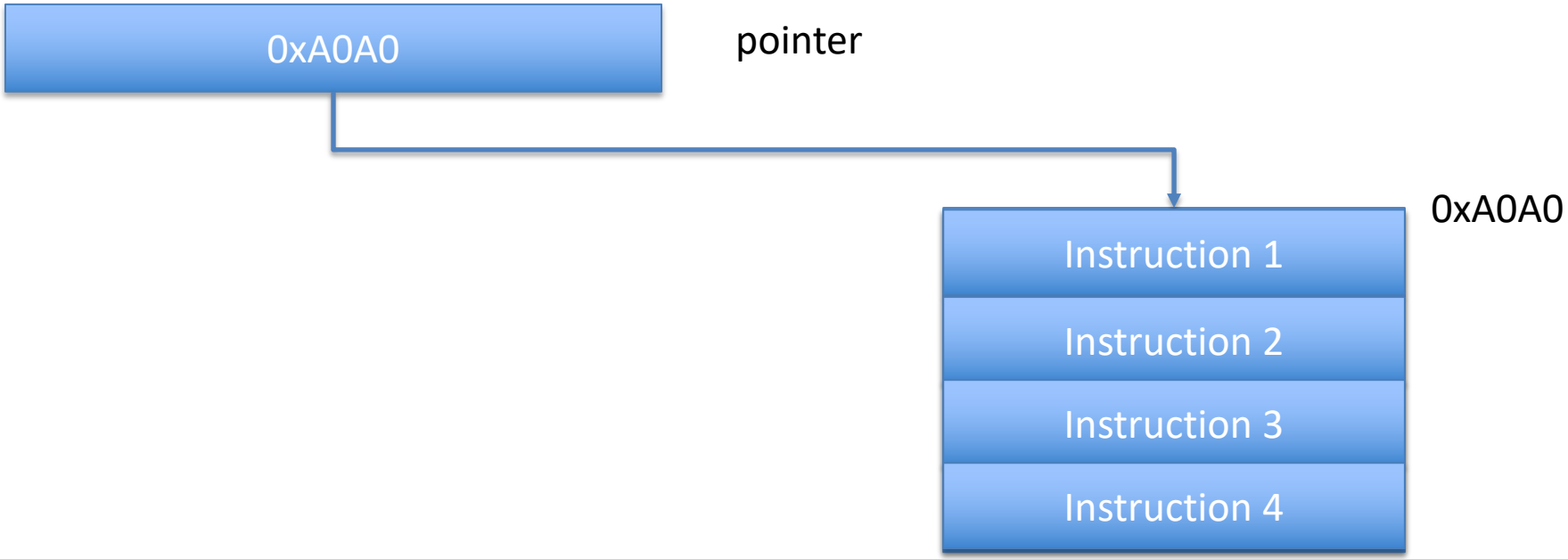
WARM UP

Where should we add free and what should we free.

WHAT IS A FUNCTION POINTER



WHAT IS A FUNCTION POINTER



HOW TO DECLARE A POINTER TO A FUNCTION

Need to discuss operator precedence

OPERATOR PRECEDENCE

How do we declare a variable that is an array of ten integers?

```
int a[10];
```

OPERATOR PRECEDENCE

How to declare a pointer to an array of ten integers

```
int *ptr[10];
```

Wrong

OPERATOR PRECEDENCE

How to declare a pointer to an array of ten integers

```
int *ptr[10];
```

Wrong

[] is higher precedence that *.
So the identifier ptr is associated with []

OPERATOR PRECEDENCE

How to declare a pointer to an array of ten integers

```
int *ptr[10];
```

Wrong

[] is higher precedence that *.
So the identifier ptr is associated with []

So it is an array of 10 integer pointers

BUT WE WANT A POINTER TO ARRAY OF 10 INTEGERS

```
int (*ptr)[10];
```

So we add brackets around the identifier

BUT WE WANT A POINTER TO ARRAY OF 10 INTEGERS

```
int (*ptr)[10];
```

So we add brackets around the identifier

Now we have a pointer to an array of 10 integers

BUT WE WANT A POINTER TO ARRAY OF 10 INTEGERS

```
int (*ptr)[10];
```

So we add brackets around the identifier

Now we have a pointer to an array of 10 integers

DECLARING A FUNCTION POINTER

```
float div(int a, int b){
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
}
```

DECLARING A FUNCTION POINTER

```
float div(int a, int b){
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
}
```

pointer

DECLARING A FUNCTION POINTER

```
float div(int a, int b){
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
}
```

A function that takes two ints and returns a float

GENERAL FORM

```
float (*ptr) ( int, int)
```

Return type

GENERAL FORM

```
float (*ptr) ( int, int)
```

Function name

GENERAL FORM

```
float (*ptr) ( int, int)
```

parameters

GENERAL FORM

```
[return type] (*[name])([parameters])
```

ASSIGNING A FUNCTION POINTER

```
float div(int a, int b){
    return a/b;
}
int main(){
    float (*ptr) ( int, int);
    ptr = &div;
}
```

USING FUNCTION PTR

```
float div(int a, int b)
{
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
    ptr = &div;
    float result = *ptr(10,20);
    printf("%f", result);
}
```

USING FUNCTION PTR

```
float div(int a, int b)
{
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
    ptr = &div;
    float result = *ptr(10,20);
    printf("%f", result);
}
```

We don't need & is optional for function names since names already represent address.

USING FUNCTION PTR

```
float div(int a, int b)
{
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
    ptr = div;
    float result = *ptr(10,20);
    printf("%f", result);
}
```

This is valid C code

USING FUNCTION PTR

```
float div(int a, int b)
{
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
    ptr = div;
    float result = *ptr(10,20);
    printf("%f", result);
}
```

We don't need * is also optional.

USING FUNCTION PTR

```
float div(int a, int b)
{
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
    ptr = div;
    float result = ptr(10,20);
    printf("%f", result);
}
```

This is also valid c

ALTERNATIVE

```
float div(int a, int b)
{
    return a/b;
}
```

```
int main(){
    float (*ptr) ( int, int);
    ptr = div;
    float result = *ptr(10,20);
    printf("%f", result);
}
```

TALK TO YOUR NEIGHBOR

```
#include <stdio.h>

void greet() {
    printf("Hello, World!");
}

int main() {
    void (*funPtr)();
    funPtr = greet;
    (*funPtr)();
    return 0;
}
```

What will happen when I run the following program:

A. Compilation Error
B. Runtime Error
C. Hello, World!
D. No Output

TALK TO YOUR NEIGHBOR

```
#include <stdio.h>

void greet() {
    printf("Hello, World!");
}

int main() {
    void (*funPtr)();
    funPtr = greet;
    (*funPtr)();
    return 0;
}
```

What will happen when I run the following program:

A. Compilation Error
B. Runtime Error
C. Hello, World!
D. No Output

SAME IS TRUE FOR THIS

```
#include <stdio.h>

void greet() {
    printf("Hello, World!");
}

int main() {
    void (*funPtr)();
    funPtr = greet;
    (funPtr)();
    return 0;
}
```

What will happen when I run the following program:

A. Compilation Error
B. Runtime Error
C. Hello, World!
D. No Output

AND THIS

```
#include <stdio.h>

void greet() {
    printf("Hello, World!");
}

int main() {
    void (*funPtr)();
    funPtr = greet;
    funPtr();
    return 0;
}
```

What will happen when I run the following program:

A. Compilation Error
B. Runtime Error
C. Hello, World!
D. No Output

TALK TO YOUR NEIGHBOR

Which of the following is the correct way to implement a function that takes a function and calls it

```
int operate(int (*func)(int, int), int x, int y) {
    return func(x, y);
 }
```

```
int operate(int (*func)(int, int) div, int x, int y) {
    return div(x, y);
 }
```

```
int operate(int (*div), int x, int y) {
    return div(x, y);
 }
```

TALK TO YOUR NEIGHBOR

Which of the following is the correct way to implement a function that takes a function and calls it

```
int operate(int (*func)(int, int), int x, int y) {
    return func(x, y);
 }
```

```
int operate(int (*func)(int, int) div, int x, int y) {
    return div(x, y);
 }
```

```
int operate(int (*div), int x, int y) {
    return div(x, y);
 }
```

PASSING A FUNCTION POINTER

```
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int operate(int (*func)(int, int), int x, int y) {
    return func(x, y);
}

int main() {
    int (*funcPtr)(int, int) = add;
    int result = operate(funcPtr, 5, 3);
    printf("%d\n", result);
    return 0;
}
```

LET'S LOOK AT AN EXAMPLE FUNCTION THAT WOULD USE A FUNCTION POINTER

```
QSORT(3)                    Linux Programmer's Manual                    QSORT(3)

NAME
       qsort, qsort_r - sort an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));

       void qsort_r(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *, void *),
                  void *arg);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       qsort_r(): _GNU_SOURCE

DESCRIPTION
       The  qsort() function sorts an array with nmemb elements of size size.  The
       base argument points to the start of the array.
```

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


int main() {
    int arr[] = {10, 5, 15, 3, 12, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    return 0;
}
```

How do we call qsort

qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));

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


int main() {
    int arr[] = {10, 5, 15, 3, 12, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Using qsort to sort the array
    qsort(arr, n, sizeof(int), ____________);



    return 0;
}
```

Let's create the function

qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));

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

int compareInts(const void *a, const void *b) {

}

int main() {
    int arr[] = {10, 5, 15, 3, 12, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Using qsort to sort the array
    qsort(arr, n, sizeof(int), ________);

    return 0;
}
```

The compare function should subtract b from a and return the result. How would we do this?

qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));

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

int compareInts(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

int main() {
    int arr[] = {10, 5, 15, 3, 12, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Using qsort to sort the array
    qsort(arr, n, sizeof(int), compareInts);

    return 0;
}
```

THERE FUNCTION DECLARATION CAN GET COMPLICATED

```
int (*(*fun_one)(char *,double))[9][20];
```

Are there rules for reading these? Yes ☺

THE RIGHT-LEFT RULE

The 'right-left' rule simplifies interpreting and creating C declarations.
Symbols:
- '*' means 'pointer to' (left side).
- '[]' means 'array of' (right side).
- '()' means 'function returning' (right side).

Follow these steps:

1. Find the identifier (the variable or function name) and start with ‘<identifier> is'.
2. Check the symbols to the right of the identifier. For example, '()' means 'function returning', and '[]' means 'array of'. Continue right until there are no more symbols or you reach a right parenthesis ')'.
3. Look at the symbols to the left of the identifier. If it's a basic type (like 'int'), state it. Otherwise, use the translations above. Continue left until there are no more symbols or you reach a left parenthesis '(‘.
4. Repeat steps 2 and 3 as necessary

EXAMPLE 1

```
int *p[];
```

1) Find identifier. int *p[];
                    ^ "p is"

EXAMPLE 1

2) Move right until out of symbols or left parenthesis hit.

```
int *p[];
      ^^ "p is an array of"
```

EXAMPLE 1

3) Can't move right anymore (out of symbols), so move left and find:

```
int *p[];
    ^ "p is an array of pointers to"
```

EXAMPLE 1

4) Keep going left and find:

int *p[];

^^^ "p is an array of pointers to ints".

EXAMPLE 2

```
int *(*func())();
```

EXAMPLE 2

1) Find the identifier.

```
int *(*func())();
        ^^^^ "func is"
```

EXAMPLE 2

2) Move right.

```
int *(*func())();
          ^^ “func is a function returning"
```

EXAMPLE 2

Can't move right anymore because of the right parenthesis, so move left.

```
int *(*func())();
      ^ "func is a function returning a pointer to"
```

EXAMPLE 2

Can't move left anymore because of the left parenthesis, so keep going right.

```
int *(*func())();
              ^^ "func is a function returning a pointer to function returning"
```

Can't move right anymore because we're out of symbols, so go left.

```
int *(*func())();
    ^ "func is a function returning a
         pointer to function returning a pointer to"
```

EXAMPLE 2

And finally, keep going left, because there's nothing left on the right.

```
int *(*func())();
^ "func is a function returning a
         pointer to function returning a pointer to an int"
```

Take a second to think about this function.

IMPLEMENTATION

```
int *(*func())();
 "func is a function returning a
         pointer to function returning a pointer to an int"
```

Let's start here.

IMPLEMENTATION

```
int *(*func())();
 "func is a function returning a
         pointer to function returning a pointer to an int"
```

```
int *myInt(){
    int a = 3;
    return &a;
}
```

Why can't we just do this?

IMPLEMENTATION

```
int *(*func())();
 "func is a function returning a
         pointer to function returning a pointer to an int"
```

```
int *myInt(){
    int a = 3;
    return &a;
}
```

Why can't we just do this?

We want the int to still be there after the function call (once the stack frame is destroyed). So instead, we need to allocate it on the heap.

IMPLEMENTATION

```
int *(*func())();
 "func is a function returning a
         pointer to function returning a pointer to an int"
```

```
  int *myInt(){
      int* a = malloc(size(int));
      return a;
  }
```

IMPLEMENTATION

```
int *(*func())();
 "func is a function returning a
          pointer to function returning a pointer to an int"
```

```
int *myInt(){
    int* a = malloc(size(int));
    return a;
}
```

Now let's write this part

IMPLEMENTATION

```
int *(*func())();
 "func is a function returning a
          pointer to function returning a pointer to an int"
```

```
int *myInt(){
    int* a = malloc(size(int));
    return a;
}
```

```
int *(*func())() = myInt;
return func;
```

More the function that contains this line next time.

LAST ONE

```
int (*(*fun_one)(char *,double))[9][20];
```

Removed parameters to make it easier to read

```
int (*(*fun_one)())[9][20];
```

LAST ONE

```
int (*(*fun_one)(char *,double))[9][20];
```

Removed parameters to make it easier to read

```
int (*(*fun_one)())[9][20];
```

"fun_one is pointer to function expecting (char *,double) and returning pointer to array (size 9) of array (size 20) of int."

NEXT TIME

Returning a function pointer from a function.

Using typedef with function pointers

Right left rule

REFERENCES

https://www.youtube.com/watch?v=BRsv3ZXoHto

https://cseweb.ucsd.edu/~gbournou/CSE131/rt_lt.rule.html