


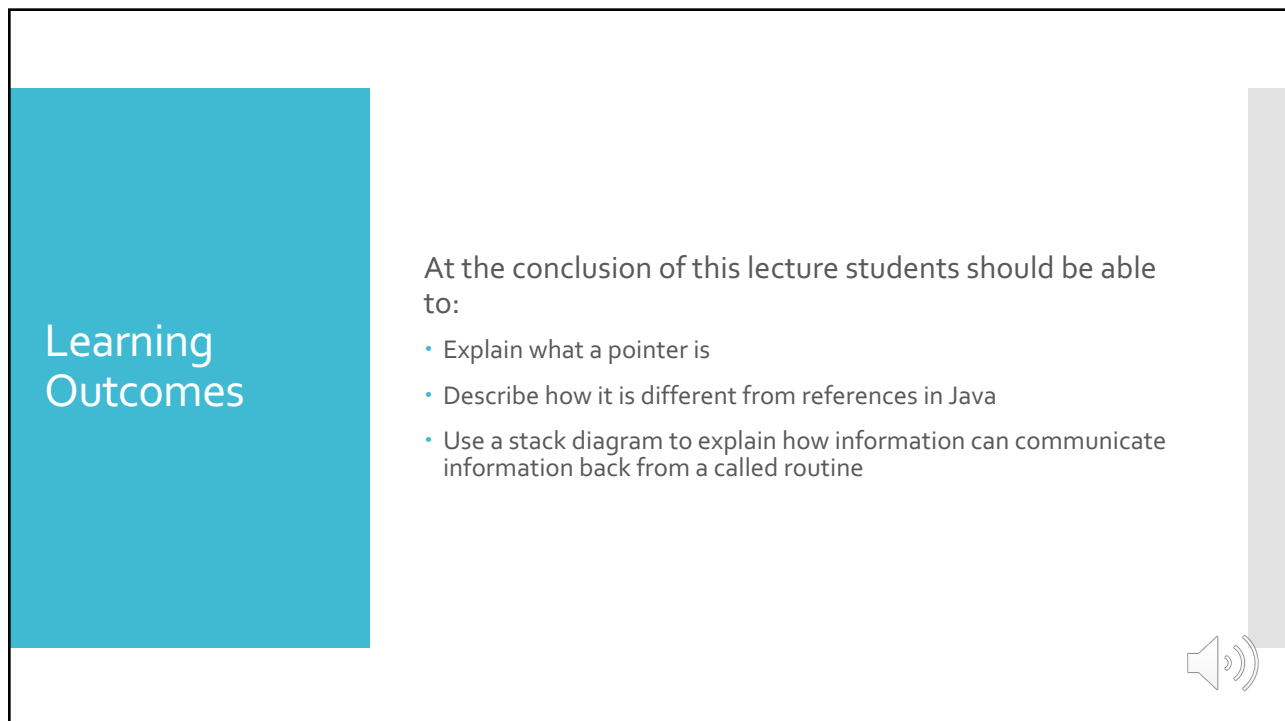
Pointers I

CS2263 – Systems Software Development

Everybody Hurts, R.E.M. (Automatic for the People, 1992) https://www.youtube.com/watch?v=5rOiW_xY-kc




1



Learning Outcomes

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

- Explain what a pointer is
- Describe how it is different from references in Java
- Use a stack diagram to explain how information can communicate information back from a called routine



2

References

- Lu, Yung-Hsiang. 2015. Intermediate C Programming. CRC Press. New York. Pp 9-27 (Chapter 4.1-4.3)
- Frantisek Franek. 2004. Memory as a Programming Concept in C and C++. Cambridge U.P.



3

Scope

- What is it?
- Why does it matter? (the “when is `i` not `i`?” edition)
 - `ifscope.c`
 - `forscope.c`
- Different scopes exist, but in the same stack frame
 - These are termed *shadow variables*



4

Pointers I

- `swap.c`: Why does this not work?
 - "Do the map"

```
int main(int argc, char* argv[]) {
    int i = 10;
    int j = 99;
    printf("i = %d; j = %d\n", i, j);
    swap(i, j);
    printf("i = %d; j = %d\n", i, j);
    return EXIT_SUCCESS;
}

void swap(int i, int j) {
    int swap;
    swap = i;
    i = j;
    j = swap;
}
```

```
Stack memory for swap() (end) -----
Var      Address Value
swap:    0x7ffeea722964 10
j:       0x7ffeea722968 10
i:       0x7ffeea72296c 99
-----
```

```
Stack memory for main() -----
Var      Address Value
j:       0x7ffeea7229a8 99
i:       0x7ffeea7229ac 10
-----
```

i = 10; j = 99

i = 10; j = 99



5

Pointers II

- There are times when we need to access memory outside of the current stack frame
 - Like when we need to "return" multiple values
 - Like when we use `scanf ()`
 - Review of `scanf ()`



6

Pointers III

- We need to store addresses
 - Address variables
 - Typed for the datatype they point to
 - Aka Pointers
- Declaring pointers
 - `<datatype>*`
 - `int* pi;`
 - `char* pc;`
 - `double* pd;`
- What would `sizeof(pc)` show?
 - `sizeof(pi)`?
 - `sizeof(pd)`?
- These are all the same
 - `int* p;`
 - `int * p;`
 - `int *p;`
- This is a trap though. What are the types?
 - `int* p, t, h, d;`
- Let me fix this for you
 - `int* p;`
 - `int t;`
 - `int h;`
 - `int d;`



7

Setting Pointers

- Through the unary operator `&`

```
int i; //value?
int* pi; //value?
pi = &i;
```
- Using dynamic memory allocation (more on this later in the course) to use the process's heap memory.
- Through calculation (pointer arithmetic)
 - e.g. `pi++`, `pi+4`, `pi-i`
 - Not a great idea



8

Dereferencing Pointers

- We know how to get the address of a variable: `&`
- Through the indirection operator `*`

```
int i = 2;
int* pi;
pi = &i;
printf("p: %i", &i, i);
printf("p: %i\n", pi, *pi);
```
- Can also use it to assign values


```
*pi = 4;
```



9

Pointer Pitfalls

- There are many
- No worries, you'll find them


```
int *p;
*p = 2;
```
- Where does `p` point to?
 - Memory belonging to the program, causing it to behave erratically
 - Memory belonging to another process, causing a segmentation fault.



10

Pointer Assignment

```
int i, j, *p, *q; //Don't outsmart yourself
p = &i;
q = p;
```

- Don't confuse `q=p` with `*q=*p`
 - What's the difference?
 - Naming conventions are helpful
 - Prefix with "p"



11

Pointers As Function Arguments I

Can I do this?

- `fl (&m) ;`
- What does it mean?
 - Passing the address of the variable
 - Allows changing the value at that address within the function
- How?
 - C is pass-by-value (Java too!)
 - C passes the addresses as values
 - The addresses can't change, but the values at the address can.



12

Pointers As Function Arguments II

Consider:

```
/* the value at the address in pi is  
 * set to zero.  
 */  
void zero(int* pi){  
    *pi = 0;  
}
```

