



CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 40

Apr 29, 2013

Review for Final Exam Part I

Lecture Goals

1. Final Exam Info
2. Find Bugs = Extra Credit
3. Pointers
4. Recursion

Upcoming Homework Assignment

HW #12 **Due: Friday, May 3**

Game AI

Create an AI for last week's project.

This is entirely extra credit and is worth 15 points.

Submission info is in the README.md on GitHub.

Final Exam

Where: Ramaley C250 (our classroom)

Date: May 8

Time: 1:30—4pm

The exam is designed to last about an hour, so it is only slightly longer than the last two. We will not need the full time period.

Final Exam

Final is worth **75 points**.

Approximately half of this is directly related to pointers and recursion. Other topics include:

Linked Lists
Binary Trees
Complexity

Graphs + Related Algorithms
Finite State Machines
Debugging

Final Exam Format

Pointer questions: short answer

Recursion questions: code

Everything else: short answer or diagram

Find Bugs, Get Extra Credit

Send mail to Gabe with the subject line **[cs2270 bug]** with legitimate technical (or grammatical) errors in any of my lecture slides or associated PDFs, and I will give you some amount of extra credit.

Time limit: You have to do this before May 6.

Going through old slides might be a good way to study, and debugging is certainly a good way to see if you know what you're doing.

Pointers are Everywhere

Languages like Java and Python hide them so you don't have to worry about punctuation like

& * . ->

... and use conventions so you always know that an item is passed by reference or by value.

Pointers in Java

Primitive types (ints and such) are passed with **values**.
Integral types (arrays, objects) are passed by **pointers**.

```
public static void add(int[] numbers, int val) {  
    for (int i=0; i < numbers.length; i++) {  
        numbers[i] = numbers[i] + val;  
    }  
}
```

```
public static void add(int change_me, int val) {  
    change_me = change_me + val;  
}
```

Pointers in Java

```
public static void main (String[] args) {  
    int[] nums = { 4, 3, 9 , 10 };  
    add(nums, 10);  
    out(nums); // prints [14, 13, 19, 20]  
    int x = 50;  
    add(x, 20);  
    out(x);    // prints 50  
}
```

Pointers in Python

You can do the exact same test in Python. When passing primitive types (like int) into a function, they are copied and put into a new variable. Anything you do to that new variable has nothing to do with the original.

Objects (including Lists) are passed by reference. You can pass a List and manipulate its contents, and those changes are persistent after the function returns.

Why Pointers Matter

Common complaint:

“Language XYZ doesn’t use pointers. You don’t need to know this to program in any modern language. This is a waste of time.”

I guarantee your language uses pointers. Learning about pointers forces you to think deeply about how the computer works. Understanding pointers, memory, and values helps demystify the machine—you *can’t engineer with something that you think of as magic.*

Pointer/Memory Review

The following slides were adapted from Lecture 35.
I've tightened up the language a bit.

Memory

Lets say we have a memory as below— 100 slots, going sequentially from left to right, top to bottom.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30										
40										
50										
60										
70										
80										
90										

Memory

When we **declare** a new variable, the operating system has to allocate memory to store it. Let's make an int:

```
int x = 9001;
```

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30										
40										
50										
60										
70										
80										
90										

Memory

First the OS finds a spot in memory that is the correct size to store the variable.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60										
70										
80										
90										

Memory

37 is that cell's **address**.

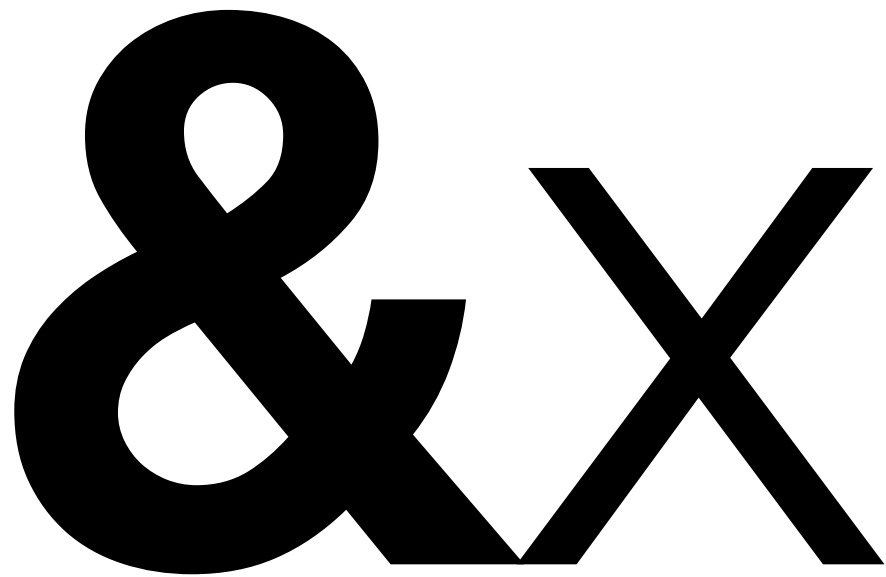
	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60										
70										
80										
90										

Memory

9001 is that cell's **value**.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60										
70										
80										
90										

Address Of Operator

A large, bold, black ampersand (&) followed by a large, bold, black 'X'. The ampersand is stylized with a thick stroke, and the 'X' is also bold and black.

An ampersand in front of a variable like this says “I demand you give me the address of this variable!”

Memory

We can prove to ourselves this is true:

```
cout << x << endl; // outputs 9001
```

```
cout << &x << endl; // outputs 37
```

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60										
70										
80										
90										

Pointer Variables

 $x_p = \&x;$

The address-of operator produces a *pointer*. We write pointers as `<data_type>*`. **Since `x` is an `int`, we know `&x` produces an `int*`.**

Pointer Variables

int* xp = &x;

Memory

Declare a pointer and set its value to x's address:

```
int* xp = &x;
```

All variable declarations result in memory allocation:

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60			37							
70										
80										
90										

Dereference Operator

```
int z = *xp;
```

We can *dereference* a pointer by putting an asterisk in front of it. It does this:

1. Read the value in xp's cell. This is an address.
2. Find the cell associated with that address.
3. Read that cell's value.

Memory

Lets create a pointer variable to x called z.

```
int z = *xp;
```

Note: two independent cells with 9001 in them.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30								9001		
40										
50										
60			37							
70							9001			
80										
90										

Pointer Syntax

values
pointers
addresses

Pointer Syntax

```
int x=42;
```

`x` is an integer that has the *value* 42. It is declared with the data type `int`, and stores the value 42.

Pointer Syntax

```
int x=42;
```

```
int* x_ptr;
```

`x` is an integer that has the *value* 42. It is declared with the data type `int`, and stores the value 42.

`x_ptr` is a *pointer to an int*. It is declared with the data type `int*`, but its value is unspecified.

Pointer Syntax

```
x_ptr = &x;
```

The value of **x** is the integer 42.

The *value* of a *pointer variable* is an *address*.

In the above code, we take the *address of x*, and store it in **x_ptr**.

Pointer Syntax

X ← This is our int x. Value is 42.

&X ← Memory address of x.

x_ptr ← int pointer. Value is the address of int x.

***x_ptr** ← Dereference an int pointer. Value is 42.

Pointers in Memory

```
int x = 42; // given memory cell 23
```

	0	1	2	3	4	5	6	7	8	9
0										
10										
20				42						
30										
40										

var	type	addr
x	int	23

Pointers in Memory

```
int x = 42; // given memory cell 23
int* x_ptr = &x; // given memory cell 30
```

	0	1	2	3	4	5	6	7	8	9
0										
10										
20				42						
30	23									
40										

var	type	addr
x	int	23
x_ptr	int*	30

Pointers in Memory

```
int x = 42; // given memory cell 23
int* x_ptr = &x; // given memory cell 30
int* other = &x; // cell 18
```

	0	1	2	3	4	5	6	7	8	9
0										
10									23	
20				42						
30	23									
40										

var	type	addr
x	int	23
x_ptr	int*	30
other	int*	18

Pointers in Memory

```
int x = 42; // given memory cell 23
int* x_ptr = &x; // given memory cell 30
int* other = &x; // cell 18
int** crazy = &x_ptr; // cell 49
```

	0	1	2	3	4	5	6	7	8	9
0										
10									23	
20				42						
30	23									
40										30

var	type	addr
x	int	23
x_ptr	int*	30
other	int*	18
crazy	int**	49

Pointers: How To Study

You'll need to know all of the punctuation.

&x	address of x (yields pointer)
x.y	member access of integral type
x->y	member access via pointer
*x	dereference pointer x
foo* x	declare pointer to foo

Pointers: How To Study

You'll need to know:

- how to declare 'regular' variables and pointers.
- how to create a pointer to a variable.
- how to read the value of a cell given a pointer to it.

Recursion

The review for recursion is in **Recursion.pdf**
(currently in the lecture_slides directory).