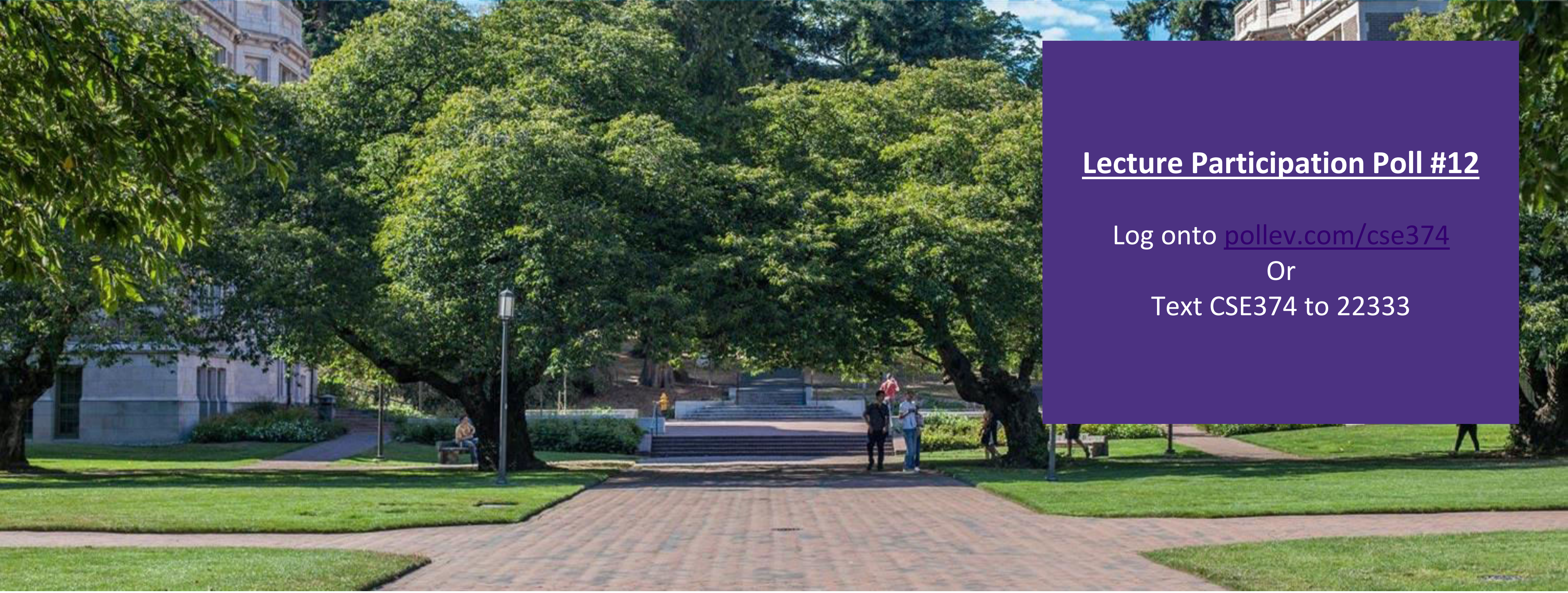


Lecture Participation Poll #12

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 12: Structs and Multi File C

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Assignments

- Klaatu should be back up
- HW2 & HW3 due Thursday
- HW4 releasing on Wednesday after lecture

Midterm on Friday

Data Types in C

- void - a place holder
- numbers – int, short, long, double, float (signed, unsigned)
- char – a very short int (1 byte) interpreted as a printable character
- pointers (T*) – stores address of where a value is stored in memory
- arrays (T[]) – implicit promotion to pointer when passed as an argument to a function or returned from a function
- booleans – not defined in C so instead we use values, 0 or NULL is interpreted as false, anything else true
- Advanced: Union T, Enum E, Function Pointers, Structs

Typedef

- A function that creates an alias for an existing type

```
typedef <type> <name>;
```

Example: In C, strings are ”char*” but we can rename them to “string”

```
typedef char* string;
int main(int argc, string *argv)
{
   string s = “hello, world”;
   printf(“%s\n”, s);
}
```

Type-casting

- **casting** – converting one type to another

```
(T)E
```

* same as Java

```
main ()
{
    int sum = 17, count = 15;
    double mean;
    mean = (double) sum / count;
    printf(Value of mean: %f\n", mean);
}
```

If E is a numeric type and T is a numeric type:

- To wider type, get same value
- To narrower type, may not get same value (employs mod operator)
- From floating point to int, will round (may overflow)
- From int to floating point, may round (int to double is exact on most machines)

Pointer-casting

- If be has type `T1*`, then `(T2*)E` is a (pointer)cast
- Does not alter the address stored, but used to manage types

```
void evil (int **p, int x)
{
   int *q = (int*)p;
   *q = x;
}

void f(int **p)
{
   evil(p, 345);
   **p = 17; // writes 17 to address 345 - best case crash
}
```

Structs

- **structs** are a method of constructing new datatypes
  - store a collection of values together in memory, fields
  - similar to a Java class, but no methods
  - individual values are referred to using the "." operator
  - can use typedef to rename and turn struct tag into a "type"

```
typedef struct Cat Cat;
```

or

```
typedef struct Cat {
    …
} Cat;
```

Then you don't need keyword "struct"

`Cat mercy;` instead of `struct Cat mercy;`

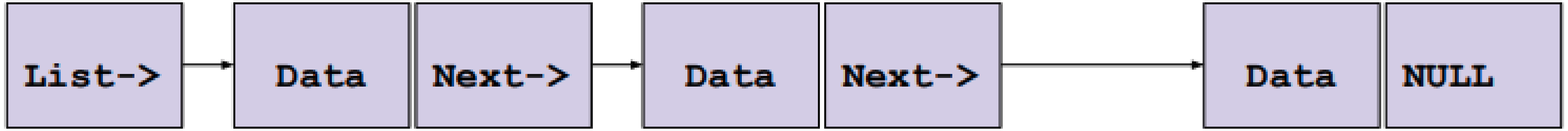
```
struct Cat
{
   char *name;
   int age;
   char *breed;
}
int main()
{
   struct Cat mercy;
   mercy.name = "Iron Fist No Mercy";
   mercy.age = 6;
   mercy.breed = "Pixie Bob";
}
```

Parameters / Arguments

- Function parameters are initialized with a copy of corresponding argument
  - If the argument is a pointer, the parameter value will point to the same thing (pointer is copied)
  - arrays are passed as pointers
  - Structs are passed as a copy by default, so it is more common to intentionally pass as pointers
    - avoids copying large objects
    - allows manipulation of original struct <- allows creation of methods that manipulate new type, like Java
    - to access members you must dereference the pointer (*) and access the field (.) – use parentheses to ensure dereference happens first
    - `(*ptr).` has a shortcut: `ptr->`

```
Cat (*ptr) = (Cat*)malloc(sizeof(Cat));
(*ptr).age = 6;
…
(*ptr).age++;
ptr->age;
```

Linked Lists




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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next) {
    Node *node = (Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

```c
int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    printf(
        "%d%d%d\n",
        n3->value,
        n3->next->value,
        n3->next->next->value
    );

    free(n3);
    free(n2);
    free(n1);
}
```

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

#include "linkedlist.h"

IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode* n = (IntListNode*) malloc(sizeof(IntListNode));
  n->data = data;
  n->next = next;
  return n;
}

IntListNode* fromArray(int* array, int length) {
  IntListNode* front = NULL;
  for (int i = length - 1; i >= 0; i--) {
    front = makeNode(array[i], front);
  }
  return front;
}

void freeList(IntListNode* list) {
  while (list != NULL) {
    IntListNode* next = list->next;
    free(list);
    list = next;
  }
}

void printList(IntListNode* list) {
  printf("[");
  while (list != NULL) {
    printf(" %d", list->data);
    list = list->next;
  }
  printf(" ]\n");
}
```

linkedlist.c

```c
#ifndef LL_H
#define LL_H

// A single list node that stores an int as
data
typedef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;


// Allocates a new node on the heap.
IntListNode* makeNode(int data, IntListNode*
next);

// Builds a heap-allocated linked list with the
values in the array.
IntListNode* fromArray(int* array, int length);

// Frees all nodes in the linked list.
void freeList(IntListNode* list);

// Prints the contents of the linked list.
void printList(IntListNode* list);
```

linkedlist.h

```c
#include <stdlib.h>

#include "linkedlist.h"

int main(int argc, char **argv) {
  int arr1[3] = {1, 2, 3};
  IntListNode* list1 = fromArray(arr1, 3);
  printList(list1);

  int arr2[4] = {4, 3, 2, 1};
  IntListNode* list2 = fromArray(arr2, 4);
  printList(list2);

  freeList(list1);
  freeList(list2);
  return EXIT_SUCCESS;
}
```

linkedlistclient.c

Multi-File C Programming

- You can split C into multiple files!
  - What if we wanted to use Linked List code in a different project?
  - If the linked list code is long, it can make files unwieldy
  - What if we want to separate our "main" from the struct definitions
- Pass all ".c" files into gcc:

```
gcc -o try_lists ll.c main.c
```

Must include code header files to enable one file to see the other, otherwise you have linking errors

```
$ gcc -g -Wall -o try_lists ll.c main.c
main.c: In function 'main':
main.c:5:5: error: unknown type name 'Node'
    5 |     Node *n1 = make_node(4, NULL);
      |     ^~~~
main.c:5:16: warning: implicit declaration of function 'make_node' [-Wimplicit-function-declaration]
    5 |     Node *n1 = make_node(4, NULL);
      |                ^~~~~~~~~
```

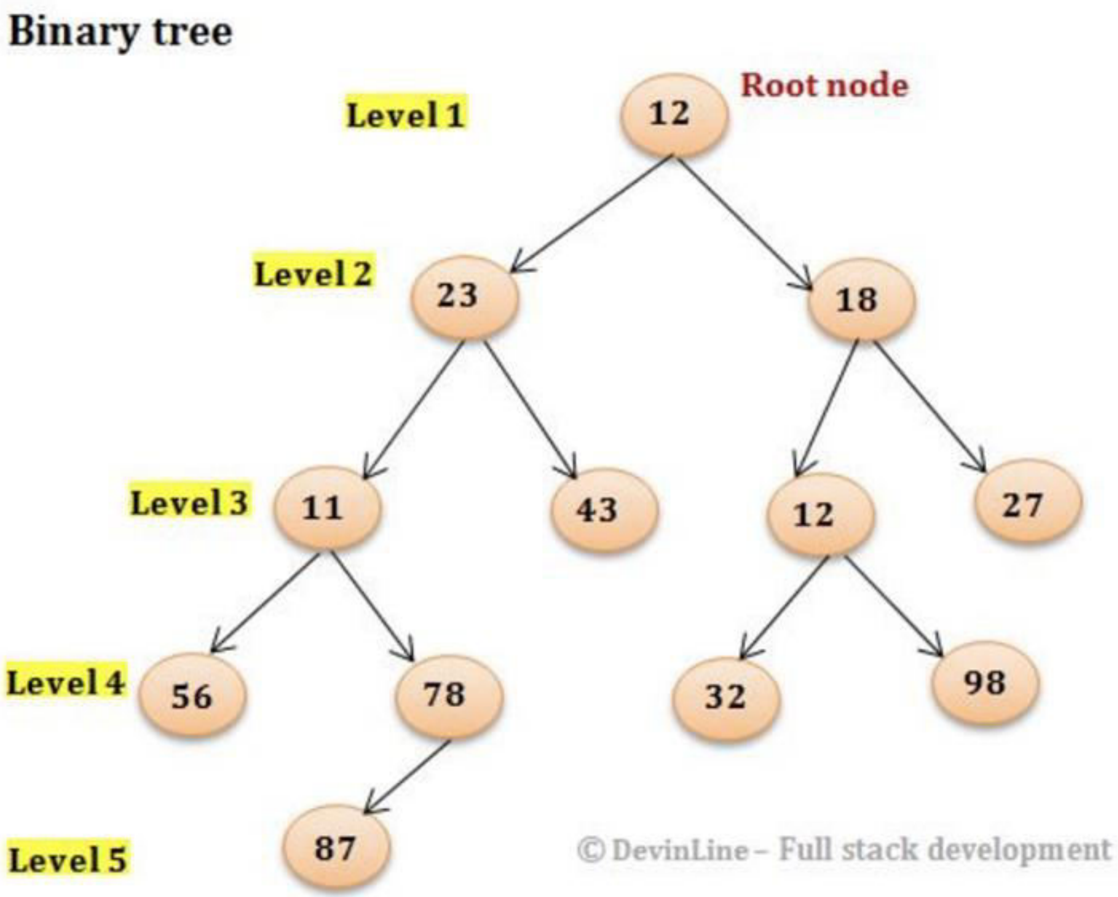
Appendix

Example: Pointer.c

```
// constructor for a new Point
Point newPoint()
{
   Point p; p.x = 0; p.y = 0; return p;
}
// translateX moves one point horizontally by deltax
void translateX(Point* p, int deltaX)
{
   p->x += deltaX; // OR (*p).x += deltaX;
}
// translateX_wrong won't move the original point
void translateX_wrong(Point p, int deltaX)
{
   p.x += deltaX;
}
// print out the point.
void print(Point* p)
{
   printf("p = (%d, %d)\n", p->x, p->y);
}
// note: here we could pass by value
void print_point(Point p)
{
   printf("p = (%d, %d)\n", p.x, p.y);
}
```

```
// main tests the Point struct
int main(int argc, char **argv)
{
   Point p = newPoint();
   printf ("Show point.\n");
   print(&p); // pass by reference
   translateX(&p, 12);
   print(&p);
   printf ("Show incorrectly translated point.\n");
   translateX_wrong(p, 12);
   print(&p);
   printf ("But pass by value works for print.\n");
   print_point (p);
}
// constructor for a new Point
Point newPoint()
{
   Point p;
   p.x = 0;
   p.y = 0;
   return p;
}
```

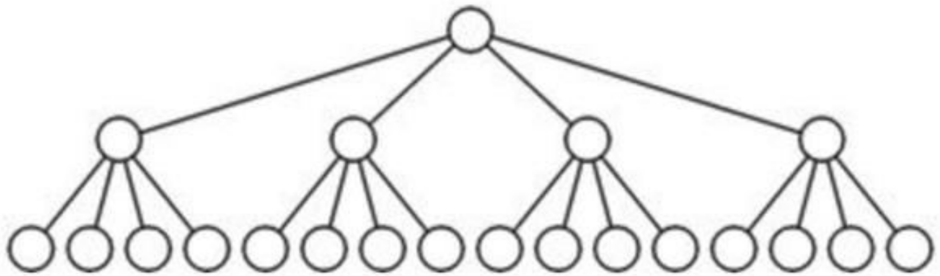
Binary Trees




```
struct BinaryTreeNode {
  int data;
  struct BinaryTreeNode* left;
  struct BinaryTreeNode* right;
}

struct BinaryTree {
  Struct BinaryTreeNode* root;
}
```

N-ary Trees



```
struct TrinaryTreeNode {
  char* data;
  struct TrinaryTreeNode* left;
  struct TrinaryTreeNode* middle;
  struct TrinaryTreeNode* right;
}
```

```
struct QuadTreeNode {
  char* data;
  struct QuadTreeNode* children[4];
}
```

Binary trees just one form; can have any “branching number”.

Trinary trees have branching number of three.

For arbitrarily large branching numbers, arrays can make more sense than lists of named pointers.