# CSE 374
# Programming Concepts & Tools

Brandon Myers

Winter 2015

Lecture 10 – C: Pointers, pointers, pointers

# Where we are

- Last time:
  - storage, scope, lifetime of variables
  - left values and right values in assignments
    - left value must have a location in memory, right value is just a value (number or address)
    - conversions between them
  - setting up the pointee / setting up the pointer
- Next:
  - review how to use pointers safely
  - pointers for passing data in/out of function calls
  - arrays and pointers
  - pointer arithmetic
  - examples

# What we learned from Binky

1. setup the pointee AND give the pointer a pointee
   – int v; int* x = &v;
2. dereference (*) a pointer to read (rvalue) or write (lvalue) its pointee
   – int v = *p
   – *p = 10;
3. assigning a pointer to another pointer makes them point to the same pointee
   – int* x; int* y; x = y;

# Dangling pointers

```
int* f(int x) {
   int *p;
   if(x) {
     int y = 3;
     p = &y; // ok
   } // ok, but p now dangling
   *p = 7;    // could CRASH! It is a bug
   return p;  // bad to return dangling pointer but will not crash
}
void g(int *p) { *p = 123; }
void h() {
   g(f(7));   // HOPEFULLY CRASHES! (but maybe not)
}
```

# Passing arguments by reference

- To pass data by reference, have the function take a pointer as an argument
- see capitalize.c

- Reassigning a pointer argument does not change the caller's pointer (the pointer itself is passed by value)
- see capitalize_use_argument.c

# Pointers to pointers to …

- Any level of pointer makes sense:
  - Example: argv, *argv, **argv, *(*argv+1)
  - Same example: argv, argv[0], argv[0][0], argv[0][1]
- But &(&p) makes no sense (&p is not a left-expression, the value is an address but the value is in no-particular-place)
- This makes sense (well, at least it's legal C):

```
void f(int x) {
  int* p = &x;
  int** q = &p;
  // ... can use x, p, *p, q, *q, **q, ...
  //        x == *p == **q
}
```

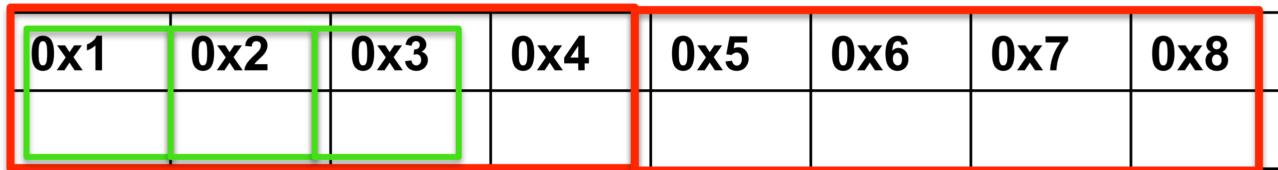
- Note: When playing, you can print pointers (i.e., addresses) with %p (just numbers in hexadecimal)

# Arrays and Pointers

- If p has type T* or type T[ ] :
  - *p has type T
  - If i is an int, p+i refers to the location of an item of type T that is i *items* past p (*not* +i storage locations unless each item of type T takes up exactly 1 unit of storage[1])
  - p[i] is defined to mean *(p+i)
  - if p is used in an expression (including as a function argument) it has type T*
    - Even if it is declared as having type T[ ]
    - One consequence: array arguments are always "passed by reference" (as a pointer), not "by value" (which would mean copying the entire array value)
    - see capitalize_array.c

*1: usually 1 byte*

# Pointer arithmetic




```
int i[2];      // i == 0x1
char* c = i;   // c == 0x1
int* j = i+1;   // j == 0x5
char* d = c+1;   // d == 0x2
```

# Arrays on the stack

- *A local variable that is an array* is allocated on the stack (that's why a size is required)
- its address is the same as that array variable's value
  - but they are different types

- see array_address.c and array_types.c

# Arrays revisited

- “Implicit array promotion”: a variable of type T[ ] becomes a variable of type T* in an expression

```
void f1(int* p) { *p = 5; }

int* f2() {
   int x[3];      /* x on stack */
   x[0] = 5;
/* (&x)[0] = 5; wrong */
   *x = 5;
   *(x+0) = 5;
   f1(x);
/* f1(&x); wrong – watch types! */
/* x = &x[2]; wrong – x isn’t really a pointer! */
   int *p = &x[2];
   return x;     /* wrong – dangling pointer – but type correct */
}
```