

---

CSE 374

# Programming Concepts & Tools

Hal Perkins

Winter 2012

Lecture 9 – C: Locals, lvalues and rvalues, more pointers

---

# The story so far...

---

- The low-level execution model of a process (one address space)
- Basics of C:
  - Language features: functions, pointers, arrays
  - Idioms: Array-lengths, strings with '\0' terminators
  - Control constructs and int guards
- Today, more features:
  - Local declarations
  - Storage duration and scope
  - Left vs. right expressions; more pointers
  - Dangling pointers
  - Stack arrays and implicit pointers (confusing)
- Later: structs; the heap and manual memory management  
(and some hacking)

# Storage, lifetime, and scope

---

- At run-time, every variable needs space
  - When is the space allocated and deallocated?
- Every variable has scope
  - Where can the variable be used (unless another variable shadows it)?
- C has several answers (with inconsistent reuse of the word static)
- Some answers rarely used but understanding storage, lifetime, and scope is important
- Related: Allocating space is separate from initializing that space
  - Use uninitialized bits? Hopefully crash but who knows?
  - Unlike Java, which zeros out objects and complains about uninitialized locals

# Storage, lifetime, and scope

---

- *Global variables* allocated before main, deallocated after main. Scope is entire program
  - Usually bad style, kind of like public static Java fields
  - But can be OK for truly global data like conversion tables, physical constants, etc.
- *Static global variables* like global variables but scope is just that source file, kind of like private static Java fields
  - Related: static functions cannot be called from other files
- *Static local variables* like global variables (!) but scope is just that function, rarely used (We *won't* use them)
- *Local variables* (often called *automatic*) allocated “when reached” deallocated “after that block”, scope is that block
  - With recursion, multiple copies of same variable (one per stack frame/function activation)
  - Like local variables in Java

# lvalues vs rvalues

---

- In intro courses we are usually fairly sloppy about the difference between the left side of an assignment and the right. To “really get” C, it helps to get this straight:
  - Law #1: Left-expressions get evaluated to locations (addresses)
  - Law #2: Right-expressions get evaluated to values
  - Law #3: Values include numbers and pointers (addresses)
- The key difference is the “rule” for variables:
  - As a left-expression, a variable *is* a location and *we are done*
  - As a right-expression, a variable gets evaluated to its location’s *contents*, and *then* we are done
  - Most things do not make sense as left expressions
- Note: This is true in Java too

# Function arguments

---

- Storage and scope of arguments is like for local variables
- But initialized by the caller (“copying” the value)
- So assigning to an argument has no affect on the caller
- But assigning to the space *pointed-to* by an argument might

```
void f() {  
    int i=17;  
    int j=g(i);  
    printf("%d %d",i,j);  
}  
  
int g(int x) {  
    x = x+1;  
    return x+1;  
}
```

# Function arguments

---

- Storage and scope of arguments is like for local variables
- But initialized by the caller (“copying” the value)
- So assigning to an argument has no affect on the caller
- But assigning to the space *pointed-to* by an argument might

```
void f() {
    int i=17;
    int j=g(&i);
    printf("%d %d",i,j);
}

int g(int* p) {
    *p = (*p) + 1;
    return (*p) + 1;
}
```

# Function arguments

---

- Storage and scope of arguments is like for local variables
- But initialized by the caller (“copying” the value)
- So assigning to an argument has no affect on the caller
- But assigning to the space *pointed-to* by an argument might

```
void f() {  
    int i=17;  
    int j=g(&i);  
    printf("%d %d",i,j);  
}
```

```
int g(int* p) {  
    int k = *p;  
    int *q = &k;  
    p = q;  
    (*p) = (*q) + 1;  
    return (*q) + 1;  
}
```

# Pointers to pointers to ...

---

- Any level of pointer makes sense:
  - Example: `argv`, `*argv`, `**argv`
  - Same example: `argv`, `argv[0]`, `argv[0][0]`
- 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, ...  
}
```
- Note: When playing, you can print pointers (i.e., addresses) with `%p` (just numbers in hexadecimal)

# Dangling pointers

---

```
int* f(int x) {
    int *p;
    if(x) {
        int y = 3;
        p = &y; /* ok */
    } /* ok, but p now dangling */
    /* y = 4 does not compile */
    *p = 7; /* could CRASH but probably not */
    return p; /* uh-oh, but no crash yet */
}

void g(int *p) { *p = 123; }
void h() {
    g(f(7)); /* HOPEFULLY YOU CRASH (but maybe not) */
}
```

# 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)
  - $p[i]$  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)

# 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 */  
}
```