CSE 374
Programming Concepts & Tools

Brandon Myers
Winter 2015
Lecture 9 – C: Locals, lvalues and rvalues, more pointers
(Thanks to Hal Perkins)
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
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?
- Allocating space is separate from initializing that space
  - Use uninitialized bytes? Hopefully crash but undefined.
  - Unlike Java, where object references default to null and numbers default to zero; and complains about uninitialized local variables
## Storage, lifetime, and scope

<table>
<thead>
<tr>
<th>type</th>
<th>lifetime</th>
<th>scope</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>global variables</td>
<td>before main to after main</td>
<td>entire program</td>
<td>often bad style but OK for truly global data like constants; kind of like public static fields in Java</td>
</tr>
<tr>
<td>static global variables</td>
<td>before main to after main</td>
<td>source file where it appears</td>
<td>kind of like private static fields in Java; static functions also not visible to other files</td>
</tr>
<tr>
<td>static local variables</td>
<td>before main to after main</td>
<td>function where it appears</td>
<td>multiple copies of same variable (as in recursion); like local variables in Java</td>
</tr>
<tr>
<td>local variables (“automatic”)</td>
<td>where declared to after the current block</td>
<td>the block where it appears</td>
<td>multiple copies of same variable (as in recursion); like local variables in Java</td>
</tr>
</tbody>
</table>
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 locations:
  - As a left-expression, a we have a location and are done
  - As a right-expression, we get the location’s contents
- Most things do not make sense as left expressions
- Note: This is true in Java too
Conversions

- lvalue can be implicitly converted to rvalue, by evaluation
  - e.g. x = z; lvalue z is converted to an rvalue
- rvalue can be explicitly converted to lvalue, by dereference operator (*)
  - e.g., *(y+1) = 5; rvalue (y+1) is converted to lvalue
  - using dereference on a non pointer type results in a type error
- lvalue can be explicitly converted to rvalue, by address-of operator (&)
  - e.g., mypointer = &x; lvalue x is converted to rvalue
  - using address-of on an rvalue is an error
Rvalue to lvalue conversion example

- int *y;
- ...
- *(y+4) = 1

<table>
<thead>
<tr>
<th>0x10(y)</th>
<th>0x18</th>
<th>0x1c</th>
<th>0x20</th>
<th>0x24</th>
<th>0x28</th>
<th>0x2c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

y points to location 0x1c. 0x1c + 4*4 = 0x2c (4*4 because y is an integer pointer)
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 does affect the caller

```c
int f(int x) {
    x = x + 1;
    return x;
}
```
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 does affect the caller

```c
int f(int x) {
    x = x + 1;
    return x;
}

void g(int* z) {
    *z = *z + 1;
}

int y = 10;
int fy = f(y);
// y = 10
int fy = f(y);
// y = 10
```
Pointer video

- Binky
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):
  
  ```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

```c
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)
}
```
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)
“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 */
}