CSE 374
Programming Concepts & Tools

Hal Perkins

Spring 2009

Lecture 8 – 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, '\0' terminators
- Today, more features:
  - Control constructs and int guards
  - Local declarations
  - Source file structure; storage duration and scope
  - Left vs. right expressions; more pointers
  - Dangling pointers
  - Stack arrays and implicit pointers (confusing)
- Next time: structs; the heap and manual memory management

  (and some hacking)

Control constructs

- while, if, for, break, continue, switch: much like Java
- Key difference: No built-in boolean type; use ints (or pointers)
  - Anything but 0 (or NULL) is true.
  - 0 and NULL are false.
  - C99 did add a bool library but use is still sporadic/ optional
- goto much maligned, but makes sense for some tasks (more general than Java's labeled break)
- Gotcha: switch cases fall-through unless there is an explicit transfer (typically a break), just like Java

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, 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.
  - So with recursion, multiple copies of same variable (one per stack frame/function activation).
  - Like local variables in Java.

Typical file layout

- Not a formal rule, but good conventional style

```
// includes for functions & types defined elsewhere
#include <stdio.h>
#include …
// global variables (if any)
static int days_per_month[ ] = { 31, 28, 31, 30, …};
// function prototypes (to handle “declare before use”)
void some_later_function(char, int);
// function definitions
void do_this( ) { … }
char * return_that(char s[ ], int n) { … }
int main(int argc, char ** argv) { … }
```

Some glitches

- Declarations must precede statements in a “block”
  - But any statement can be a block, so use { … } if you need to
  - Or use --std=c99 gcc compiler option
- Array variables in code must have a constant size
  - So the compiler knows how much space to allocate
  - (C99 has an extension to relax this; rarely used)
  - Arrays whose size depends on runtime information are allocated on the heap (next time)
  - Large arrays are best allocated on the heap also, even if constant size, although not required
- Array types in function arguments are pointers(!)
- Referring to an array doesn’t mean what you think (!)
  - “implicit array promotion” (later)

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

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 g(int* p) {
   int i=17;                       *p = (*p) + 1;
   int j=g(&i);                    return (*p) + 1;
   printf("%d %d",i,j);         }
}
```

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 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) */
}
```

More gotchas

- Declarations in C are funky:
  - You can put multiple declarations on one line, e.g., int x, y; or int x=0, y; or int x, y=0;, or ...
  - But int *x, y; means int *x; int y; – you usually mean int *x, *y;
  - Common style rule: *one* declaration per line (clarity, safety)
- No forward references:
  - A function must be defined or declared before it is used. (Lying: “implicit declaration” warnings, return type assumed int, ...)
  - Linker error if something is used but not defined (including main)
    - Use -c to not link yet (more later).
  - To write mutually recursive functions, you just need a (forward) declaration.
- Variables holding arrays have super-confusing (but convenient) rules…

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[0] = 5;
/* (&x)[0] = 5; wrong */
   *x = 5;
   *(x+0) = 5;
   f1(x);
/* f1(&x); wrong */
/* x = &x[2]; wrong */
   int *p = &x[2];
}
```