CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Winter 2009

Lecture 8— C: locals, left vs. right expressions, dangling pointers, ...

Where are We

- 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
  - 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.

Control constructs

- `while`, `if`, `for`, `break`, `continue`, `switch` all 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`).
- See `sums.c`; should be understandable on your own (with help from the book, web, etc.)

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.

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.
- *Static global variables* like global variables but scope is just that *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.
- *Local variables* allocated "when reached" deallocated "after that block", scope is that block.
  - So with recursion, multiple spaces for same variable (one per stack frame).
  - Like local variables in Java.

A typical file layout

No *rules* on this order, but good conventional style

```
// includes for functions, types defined elsewhere (just prototypes)
#include <stdio.h>
#include ...
// global variables (usually avoid them)
int some_global;
static int this_file_arr[7] = { 0, 2, 4, 5, 9, -4, 6 };
// function prototypes for forward-references (to get around
// uses-follow-definition rule)
void some_later_fun(char, int); // argument names optional
// functions
void f() { ... }
void some_later_fun(char x, int y) {...}
int main(int argc, char**argv) {...}
```

Some glitches

- Silly almost-obsolete syntax restriction not in Java or C++: declarations only at the beginning of a "block" – but any statement can be a block.
  - Just put in braces if you need to (see `main` in `sums.c`)
  - Or use `--std=c99` compiler option (gcc)
- (Local or global) variables holding arrays must have a constant size
  - So the compiler knows how much space to give.
  - (C99 has an extension to remove this limitation; rarely used.)
  - So for arrays whose size depends on run-time information, allocate them in the heap and *point* to them (next time)
- Array types as function arguments don't mean arrays (!)
- Referring to an array doesn't mean what you think it does (!)
  - "implicit array promotion" (come back to this)

Function arguments

- Storage and scope of arguments is like for local variables.
- But intialized 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;
}
```

Left vs. right

We have been fairly sloppy in 142, 143, and so far here 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 intialized 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 intialized 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:

```
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;`

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 declaration.

Variables holding arrays have super-confusing (but convenient) rules...

Stack Arrays Revisited

A very confusing thing about C: "implicit array promotion (in right-expressions)"

```
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];
}
```