CSE 303: Concepts and Tools for Software Development

Dan Grossman

Winter 2006

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
  - Left vs. right expressions
  - Stack arrays and implicit pointers (confusing)
    - dangling pointers

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.
  - Anything but 0 (or `NULL`) is true.
  - 0 and `NULL` are false.
- `goto` much maligned, but makes sense for some tasks (more general than Java's labeled break).

Local declarations

- 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`)
  - Difference between similar notions: *scope* and *lifetime*
  - If you "goto into scope", YPMSTCOF[a]
- You can also allocate arrays on the stack, but:
  - Size must be a constant expression (slowly changing (!))
  - 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)

[a] Your Program Might Set The Computer On Fire.

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.

The address-of and dereference operators

```
void f() {
  int x;
  int y;
  int *p;
  int *q;
  x = 3;
  y = x+1;
  p = &x;
  q = p;
  q = &y;
  *q = *p;
  q = 0; /* i.e., NULL */
  *q = 4; /* YPMSTCOF */
}
```

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; /* YPMSTCOF, but probably not */
  return p; /* uh-oh */
}
void g(int *p) { *p = 123; }
void h() {
  g(f(7)); /* YPMSTCOF, and likely a problem */
}
```

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];
}
```

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;`, ...
- But `int *x, y;` means `int *x; int y;` — you usually mean `int *x, *y;`

No forward references:

- A function must be defined and/or declared before it is used. (Lying: "implicit declaration" warnings, return type assumed to be `int`, ...)
- You get a *linker error* if something is declared but never defined (or main is not defined).
- You can still write mutually recursive functions, you just need a declaration.