CSE 374: Programming Concepts and Tools

Eric Mullen

Spring 2017

Lecture 10: Locals, lvalues vs rvalues, more pointers

Administrivia

- Homework 3 is out: start working TODAY if you haven't already
- Midterm is a week from Friday, during class time
    - Would people like a review session?

Process Execution

- Recall the single address space
- This holds everything during execution
  - But *when*, exactly?

0x0000 0xFFFF

	Code	Globals	Heap ->		<- Stack

Scope

- The *scope* of a variable describes when it will exist, programmatically
- At runtime, everything needs memory space
- C has several different ways to declare the scope of something
    - Most all reuse the keyword *static* in different ways
- Reminder: Allocating space is separate from initializing that space

Scope

- **Global Variables**: declared outside any function, allocated before main called, deallocated after main returns
    - usually bad style, can be ok for truly global data
- **Static Global Variables**: declared just like globals, but use the *static* keyword, restricted to use within one file
    - related: static functions are also limited to within one file
- **Static Local Variables**: lifetime like globals, but use restricted to one function. NOT USED IN THIS CLASS
- **Local Variables**: allocated when reached, deallocated at end of block

lvalues and rvalues

- assignment in C: `<lvalue> = <rvalue>`
  - `<lvalue>` evaluates to a location
  - `<rvalue>` evaluates to a value
- Key difference is with variables
  - On the left, variable evaluates to a location
  - On the right, variable is accessed, and we get the contents of that location
  - Recall in Bash: we used $foo on right side

Function Arguments

- Storage and Scope of arguments is like for local variables
  - Except: arguments are initialized by copying their value
  - Assigning to an argument has no effect on caller
    - Except: assigning to space pointed to by argument might affect the caller

Example

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

i	17

j	19

x	18

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

19

Example

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

i	18

j	19

p	

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

19

Example

```
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;
}
```

i 18

j 18

p

k 17

q

18

Pointers to pointers to…

- You can construct this as deep as you want:
  - Example: argv, *argv, **argv
- However, &(&p) makes no sense: (&p) is not an lvalue, the value is an address, but the value is not in a particular place
- Note: When playing, the %p format string will let you print out the value of pointers

Example

```
void f(int x) {
  int* p = &x;
  int** q = &p;
  //at this point x, p, *p, *q, and **q
  //make sense as rvalues
}
```

Dangling Pointers

- aka how to shoot yourself in the foot
- If you have a pointer to something, and what it points to goes out of scope, the pointer you have is now *dangling*
- Be careful of this!

Dangling Pointers

```
//always returns
//a dangling pointer
int* foo() {
  int x = 5;
  return &x;
}
```

x 5

```
void bar() {
  int* x = foo();
  *x = 7;
}
```

x

Arrays and Pointers

- If p has type T* or 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 bytes, unless each T takes only one byte)
  - p[i] is defined to mean *(p+i)
  - if p is used in an expression, it has type T*
    - even if it is declared to have type T[]