C pointers

(Reek, Ch. 6)
Review of pointers

- A pointer is just a memory location.
- A memory location is simply an integer value, that we interpret as an address in memory.
- The contents at a particular memory location are just a collection of bits – there’s nothing special about them that makes them `int`s, `char`s, etc.
- How you want to interpret the bits is up to you.

- Is this... an `int` value?
  - ... a pointer to a memory address?
  - ... a series of `char` values?
A pointer variable is just a variable, that contains a value that we interpret as a memory address.

Just like an uninitialized int variable holds some arbitrary “garbage” value, an uninitialized pointer variable points to some arbitrary “garbage address”

```c
char *m;
```

```
(char *) m
```
Following a “garbage” pointer

- What will happen? Depends on what the arbitrary memory address is:
  - If it’s an address to memory that the OS has not allocated to our program, we get a segmentation fault
  - If it’s a nonexistent address, we get a bus error
  - Some systems require multibyte data items, like ints, to be aligned: for instance, an int may have to start at an even-numbered address, or an address that’s a multiple of 4. If our access violates a restriction like this, we get a bus error
  - If we’re really unlucky, we’ll access memory that is allocated for our program – We can then proceed to destroy our own data!
How can you test whether a pointer points to something meaningful?

- There is a special pointer value `NULL`, that signifies “pointing to nothing”. You can also use the value `0`.

```c
char *m = NULL;
...
if (m) { ... safe to follow the pointer ... }  
```

- Here, `m` is used as a Boolean value
  - If `m` is “false”, aka `0`, aka `NULL`, it is not pointing to anything
  - Otherwise, it is (presumably) pointing to something good
  - Note: It is up to the `programmer` to assign `NULL` values when necessary
Indirection operator *

- Moves from address to contents

```c
char *m = "dog";
char result = *m;
```

- `m` gives an address of a char
- `*m` instructs us to take the contents of that address
- `result` gets the value 'd'
Address operator &

- Instead of contents, returns the address

```c
char *m = "dog",
    **pm = &m;
```

pm needs a value of type char **
- Can we give it *m? No – type is char
- Can we give it m? No – type is char *
- &m gives it the right value – the address of a char * value
C allows pointer values to be incremented by integer values

```c
char *m = "dog";
char result = *(m + 1);
```

- `m` gives an address of a `char`
- `(m + 1)` gives the `char` one byte higher
- `*(m + 1)` instructs us to take the contents of that address
- `result` gets the value ‘o’
A slightly more complex example:

```c
char *m = "dog";

char result = *++m;
```

$m$ gives an address of a char 
$++m$ changes $m$, to the address one byte higher, 
and returns the new address 
$*++m$ instructs us to take the contents of that location 
$result$ gets the value ‘o’
How about multibyte values?

Q: Each char value occupies exactly one byte, so obviously incrementing the pointer by one takes you to a new char value...

But what about types like int that span more than one byte?

A: C “does the right thing”: increments the pointer by the size of one int value

```c
int a[2] = {17, 42};
int m = a;
int result = *++m;
```
Example: initializing an array

```c
#define N_VALUES 5
float values[N_VALUES];

float *vp;
for (vp = &values[0]; vp < &values[N_VALUES]; )
    *(vp++) = 0;
```

(done!)
A note on assignment: Rvalues vs. Lvalues

What’s really going on in an assignment?
- Different things happen on either side of the assignment operator.

```c
int a = 17, b = 42;
b = a;
```

- `a` is the "rvalue" (right value)
  - We go to the address given by `a`...
  - and get the contents (17)

- `b` is the "lvalue" (left value)
  - We go to the address given by `b`...and get the contents?
  - No! We don’t care about 42!
  - We just want the address of `b` – to store 17 into
A note on assignment: Rvalues vs. Lvalues

This explains a certain “asymmetry” in assignments involving pointers:

```c
char *m = NULL, **pm = NULL;
m = "dog";
pm = &m;
```

Here, `m` is an lvalue – It’s understood that the address of `m` is what’s needed.

Once again, we need the address of `m` – but since it’s an rvalue, just plain `m` will give the contents of `m` – use `&` to get the address instead.
Example: `strcpy “string copy”`

```c
char *strcpy(char *dest, const char *src)
```

- (assume that) `src` points to a sequence of `char` values that we wish to copy, terminated by NUL
- (assume that) `dest` points to an accessible portion of memory large enough to hold the copied chars
- `strcpy` copies the char values of `src` to the memory pointed to by `dest`
- `strcpy` also gives `dest` as a return value
Example: `strcpy “string copy”`

```c
char *strcpy(char *dest, const char *src) {
    const char *p;
    char *q;
    for(p = src, q = dest; *p != '\0'; p++, q++)
        *q = *p;
    *q = '\0';
    return dest;
}
```
Pointer subtraction and relational operations

- Only meaningful in special context: where you have two pointers referencing different elements of the same array
  - \( q - p \) gives the difference (in number of array elements, not number of bytes between \( p \) and \( q \) (in this example, 2)
  - \( p < q \) returns 1 if \( p \) has a lower address than \( q \); else 0 (in this example, it returns 1)