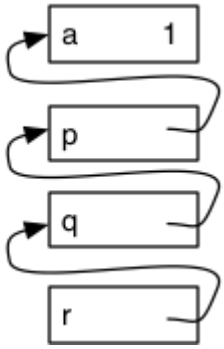
Pointers to Pointers

We can have pointers to pointers (or pointers to pointers to pointers…*etc*). For example, an **int****is a pointer to a pointer to an int. An **int*****is a pointer to a pointer to a pointer to an int. We can have as many levels of "pointer" as we want (or need), however, the usefulness drops of quite quickly (**int**** is quite common, **int***** moderately common, but neither author has ever had a use for an **int**********). The rules for pointers to pointers are no different from anything else we have seen so far: the * operator dereferences a pointer (follows an arrow), and the & operator takes the address of something (gives an arrow pointing at that thing).

The figure above illustrates pointers to pointers. Here we have 4 variables, *a* (which is an **int**), *p* (which is an **int***), *q* (which is an **int** **), and *r* (which is an **int*****)**. If we were to write **r,* it would refer to *q*'s box (because we would follow the arrow from **r**'s box, and end up at **q**'s box. We could write **r, which would refer to *p*'s box—because **r* is an arrow pointing at p, and the second * dereferences that pointer. Likewise, ****r* would refer to *a*'s box. It would be a compiler error to write *****r*, because that would attempt to follow the arrow in a's box, which is not an arrow, but rather a number (sure, everything is a number —but our types have told the compiler that a is just a plain number, not a number that means an arrow).

```
int      a = 1;
int *    p = &a;
int **   q = &p;
int *** r = &q;
```



You may wonder why we might want pointers to pointers. One answer to this question is "for all the same reasons we want pointers"—a pointer gives us the ability to refer to the location of a thing, rather than to have a copy of that thing. Anytime we have a variable that tells us the location of a thing, we can change the original thing through the pointer. Just as we might want to write swap for integers, we might also want to write swap for **int** *s (in which case, our swap function would take **int** **s as parameters).

Notice how the types work out (*i.e.* match up so that the compiler type check the program). Whenever we take the address of an lvalue of type *T*, we end up with a pointer of type *T** (*e.g.*, *p* has type **int** *, so *&p* has type **int** **). Whenever we take the address of something, we "add a star" to the type. This rule makes intuitive sense, because we have a pointer to whatever we had before. Whenever we dereference an expression of type *T**, we end up with a value of type *T*—we "take a star off the type", because we followed the arrow.

For the expression with variable e:	*e	&e

For the expression with variable e:	*e	&e
e must be...	a pointer	an lvalue
if e's type is..... then the resulting type is	T*T	TT*
conceptually, this means:	follow the arrow that is e's value	give me an arrow pointing at e

The table above summarizes these rules about pointers. Note that * and *&* are inverse operations—if we write **&e* or *&*e,* they both just result in e (whenever they are legal). The first would mean "give me an arrow pointing at *e*, then follow it back (to *e*)", while the second would mean "follow the arrow that is 's value, then give me an arrow pointing at wherever you ended up."

Completed