Tip 37: Rename things with pointers

Ben Klemens

14 December 2011

level: just far enough to be confused by pointers
purpose: distinguish between aliasing and memory management issues

OK, where were we? At the outset, I went over some of the means (Entry #050) of compiling (Entry #059) your C programs, in case you're used to having an interpreter doing all your dirty work. I went over the subset you need to make strings tolerable (Tips 10 (Entry #060), 11 (Entry #061), 12 (Entry #062), and 17 (Entry #067)). I showed you some tricks with compound literals (Entry #074), with macros (Entry #075), and how far you can get using the two together (Entry #078). Underlying that was that you could do so while avoiding the dreaded `malloc` and all the associated memory management. I covered three types of memory (Entry #070) (there's a fourth type to come). I haven't even mentioned structs yet, though my number one favorite tip is about them. That'll come next.

But for now, let me pick up on that thread of segregating `malloc` and memory management issues to their proper place.

When I tell my computer *set* `A` *to* `B`, I could mean one of two things:

1. Copy the value of `B` into `A`. When I do `A++`, then `B` doesn't change.

2. Let `A` be an alias for `B`. When I do `A++`, then `B` also gets incremented.

The first conceptual tip is in no way C specific: every time your code says *set* `A` *to* `B`, you need to know whether you are making a copy or an alias.

For C, you are always making a copy, but if you are copying the address of the data you care about, a copy of the pointer is a new alias for the data. That's a fine implementation of aliasing. It doesn't get awkward until you start aliasing the the location of the data, which is the start down the chain of aliasing aliases.

Other languages have different customs: LISP family languages lean heavily on aliasing and have `set` commands to copy; Python generally copies scalars but aliases lists (unless you use `copy` or `deepcopy`). Again, knowing which to expect will clear up a whole lot of bugs all at once.

[By the way, you'll now and then meet a language that does not provide any mechanism at all for one of aliasing or copying. I hate to be negative, but such languages are braindead. Do not use them for serious work. I don't care if the language is well-funded and there are conferences about it; this is an absolutely basic requirement.]

We often wind up with structures within structures within structures. Let me use Apophenia as an example, and allocate an `apop_data` set via

```
apop_data  *adata = apop_data_alloc(1,1);
```

This is an `apop_data` set that has one element, which you may sometimes need; e.g., model parameters have to be of type `apop_data`, so if your model has one parameter, this is what you've gotta have. But this declaration wraps a `gsl_matrix`. If you're mostly working with the matrix, then name it:

```
apop_data  *adata = apop_data_alloc(1,1);
gsl_matrix  *matt = adata->matrix;

//Now matrix operations are clearer:
gsl_matrix *inv = apop_matrix inverse(matt);
```

[Inverting a 1-D matrix may seem like overkill, but in a larger routine, it sure beats writing a bunch of special cases that depend on the size of the input matrix.]

In fact, the matrix is really just an array of `doubles`, so the one element in that 1×1 matrix may also merit an alias:

```
apop_data  *adata = apop_data_alloc(1,1);
double *thedata = adata->matrix->data;

//Use this like any other scalar:
*thedata = 1.2;
```

Getting slices of matrices and vectors also work by generating aliases for the data.

So you've got a lot of ways of looking at your data, depending on whatever is conveninent and has meaning for you. The alias makes clear to human readers what your focus is and how you are thinking about the data.

And writing `thedata` sure is more readable than putting `adata->matrix->data` everywhere.

Oh, look: we've gotten through another tip on pointers without mentioning `malloc`—and that's really the point of this tip. The concept of manually-allocated memory and the concept of an alias are distinct, and you can readily use aliases without manual allocation.

If you manually allocate a block, you're going to feel pretty stupid if you don't have a means of referring to it, so you can't have manual allocation without a pointer aliasing the location. That's why the old school textbooks introduce pointers and manual memory management at the same time. But so what—point to whatever you want whenever you want to make something more readable.

To do:
I gave you that advice above that every time you have a line that says *set* `A` *to* `B`, you need to know whether you are asking for an alias or a copy. Grab some code you have on hand (in whatever language) and go through line by line and ask yourself which is which. Were there cases where you could sensibly replace a copying with an alias?