Tip 55: base your code on pointers to objects

Ben Klemens

23 January 2012

level: Your code depends on a class of objects
purpose: don't worry about scope when you don't want to

So you've seen several examples at this point of an object setup, with a typedef and new/copy/free functions. There was the `fstr*`, a pointer to a file-as-string, the `group*`, a pointer to a social group which our imaginary agents joined and left, and the `apop_model *`, a pointer to a statistical or simulation (or other) model.

Why did I base all of these things on pointers to data structures, instead of just passing around data structures directly? Using a plain struct is as easy as rolling out of bed. If you use a pointer-to-struct, you require a new/copy/free function; if you use a plain struct, then:

new Use designated initializers on the first line where you need a struct.

copy The equals sign does this.

free Don't bother; it'll go out of scope soon enough.

So we're making things more difficult for ourselves with pointers. Yet from what I've seen, there's consensus on using pointers to objects as the base of our designs.

Pros to using pointers:

- Copying a single pointer is cheaper than copying a full structure, so you save a picosecond on every function call with a struct as an input. Of course, this only adds up after a few hundred million function calls.
- Data structure libraries (your trees and linked lists) all have hooks for a void pointer.[1]

[1] A side note on the void pointer thing: using a void pointer is giving up the type-checking that is decidedly a good thing and which has saved all of us heartache at some time or another. With regards to linked lists and such, however, I've never perceived a problem. If I have a linked list named `active_groups` and another list named `persons`, both of which take in a void pointer, then the compiler would let me append `person` to the `active_groups` list, but it is obvious to me as a human that I'm touching the wrong list. As a practical matter, it doesn't take all that much care to check that you don't hook the wrong thing onto a data structure's void pointer hook.

Things get much worse when you have a void pointer as input to a function, and when using that kind of mechanism, I try to keep the function and the calls to the function close together.

- Now that you're filling a data structure, having the system automatically free the struct at the end of the scope in which they were created may be an annoyance.
- Many of your functions that take in a struct will modify the struct's contents, meaning that you've got to pass a pointer to the struct anyway. Having some functions that just take in the struct and some that take in a pointer to struct is confusing (I have one interface like this and I regret it), so you might as well just send a pointer every time.
- Once you have a pointer inside the struct, then the convenience bonus from using a plain struct evaporates anyway: if you want a deep copy (wherein the data pointed to is copied, not just the pointer) then you need a copy function, and you will probably want a free function to make sure the internal data is eliminated.

As your project gets bigger, and a throwaway struct grows into a core of how your data is organized, the pros for pointers wax and the pros for non-pointers wane. That's why these tips, taken together, don't provide a one-size-fits-all set of rules for using structs. The best techniques for using the throwaway structs are different from those for using structs as the core of your data organization.