Tip 39: Know the constraints of C structs

Ben Klemens

18 December 2011

level: beginner in C, versed in other languages
purpose: add elements to your structures

Adding elements to your structures is, it turns out, a really hard problem, with an abundance of different solutions, none of which are all that pleasing. This entry isn't too much of a direct tip, to tell you the truth, but is about the means of thinking about all this.

We'll start with the problem statement, which is a restatment of the previously-stated principle (Entry #073) that C really likes to think in base-plus-offset terms. If you have an array of ints declared via `int *aa`, then that's a specific location in memory, and you refer to `int[3]`, that is a specific location in memory `3 * sizeof(int)` past the point where `aa` is. Structured data works the same: if I declare a type and an instance of that type like this:

```
typedef struct {
    char *name;
    void *value;
} list_element_t;

list_element_t *LL;
```

then I know that `LL` is a point in memory, and `LL->value` is a point in memory just far enough past that to get past the name, i.e. `sizeof(char*)` further down from `LL`.

In terms of processing speed, this base-plus-offset setup is as fast as it gets, but it more-or-less requires knowing the offsets at compile-time. If you come in later in the game and somehow redefine `list_element_t` as not having that first `name` element, then the whole system breaks down. You could perhaps imagine extending the structure to have more elements after the fact, but now all your arrays of stucts are broken. Your structure is fixed at compile time, at which point the various offsets are measured and your source code converted to express offsets (`LL + sizeof(int)`) instead of the names we recognize (`LL->value`).

Pro: we get lots of compile-time checks, and what is probably the fastest way possible to get to your data. Con: the structrue is fixed at compile time.

There are many ways to get around the fixed structures.

C++, Java, &c Develop a syntax for producing a new type that is an instance of the type you want to extend, but which inherits the old type's elements. Pros: you can still get base-plus-offset speed, and compile-time checking; once the child structure is set up, you can use it as you would the parent. Cons: so much paperwork, so many added keywords to support the structure (where C has `struct` and its absurdly simple scoping rules (Entry #047), Java has `implements`, `extends`, `final`, `instanceof`, `class`, `this`, `interface`, `private`, `public`, `protected`).

Perl, Python, &c A structure in any of these languages is really a list of named elements—the `struct` above would be a fine prototype for a rudimentary implementation. When you need an item, the system would traverse the list, searching for the item name the programmer gave. Pros: fully extensible by just adding a new named item. Cons: ¿when you refer to a name not in the list, is it a typo or are you adding a new element?; you can improve the name search via various tricks, but you're a long ways from the speed of a single base-plus-offset step; you'll need more C++-like syntax go guarantee that a list-as-structure has certain elements.

C All the machinery you have in C for extending a structure is to wrap it in another structure. Say that the above type is already packaged and can not be changed, but we'd like to add a type marker. Then we'll need a new structure:

```
typedef struct {
    list_element_t elmt;
    int typemarker;
} list_element_w_type_t;
```

Pros: this is so stupid easy, and you still get the speed bonus. Cons: Now, every time you want to refer to the name of the element, you need `your_typed_list->elmt->name` instead of what you'd get via a C++/Java-like extension: `your_typed_list->name`. Add a few layers to this and it starts to get annoying. You still don't get to add to the list during run-time; the only way to do this is via a list like the one the struct here describes.

If you come from one of the other traditions, don't expect the C struct to do what a list or hash does in Ruby or Perl—but you can get such a struct via the Glib library. Instead, revel in the simplicity of building structures using other structures as elements. Several of the tips to follow will help you with using nested structures without annoyance. You already saw how using aliases (Entry #087) can help here.