Structs versus dictionaries

Ben Klemens

25 March 2011

This continues the entry #038 giving a simplified (but sufficient) view of compound types across all languages. Every language has lots of cute tricks, but if you know what sort of means your language has of representing lists of numerically indexed elements, and how it deals with lists of named elements, then you can fake yourself a pretty long way along.

In the last episode, I equated `structs` and dictionaries, which may seem odd to those of you who have used both. Their intent tends to be different, as revealed by their names: `structs` are for structured collections of data, while dictionaries are for long lists of named elements.

[The internal workings are certainly different, but the point of this post is that this paragraph on internals is a digression which you can skip. A `struct` is a variant of an array, in that it's a sequence of items at some spot in memory; the only difference is that the position has a name instead of a number, and the distance from one item to the next, in terms of transistors on the memory chip, isn't constant. A dictionary or hash is a higher-level structure, maybe a linked list or something like what I sketch out below, which somehow associates a name (a text string) with each item. Comparing two strings is computationally expensive, so the strings are typically munged into a more easily compared number—a hash. So the `struct` is a variant of the array that allows variable-length elements and names in your source code; the dictionary is a high-level data structure that happens to use strings as labels.]

All those differences aside, they do share much in common. You'll notice, for example, that none of the languages in the list from last time have both a fixed `struct` and a dictionary built in to the language: if there's a dictionary or hash, then that serves as the vehicle by which complex types get constructed. In the other direction, though, you can't use a `struct` to generate an especially long list of named elements, like a *bona fide* dictionary of English words and their definitions.

Or to put this another way:

	array	`struct`	dictionary
many homogeneous elements	yes	no	yes
some heterogeneous elements	don't	yes	yes

I explained the *don't* entry last time: your language may allow a numerically indexed array to hold a long list of heterogeneous elements, but this is lousy form; more below. The *no* entry is because `struct` declarations can only be so long here in practical reality.

At this point, some of the fans of the newer languages declare victory—the dictionary does more than the `struct`. But this is using only what is built in to the grammar of the language.

A dictionary is an easy structure to generate given what we have in the static-struct languages. Here's some C code; for consistency with the awk example from last time, you can cut and paste it onto your command line.

```
echo '
#include <stdio.h>

typedef struct {
   char *key;
   void *value;
} keyval;

int main(){
   int zero = 0;
   float one = 1.0;
   char two[] = "two";

   keyval dictionary[] = {{.key="zeroth", .value=&zero},
                          {.key="first", .value=&one},
                          {.key="second", .value=&two}};

   printf("keyval %s: %i\n", dictionary[0].key,
                     *(int*)dictionary[0].value);
   printf("keyval %s: %g\n", dictionary[1].key,
                     *(float*)dictionary[1].value);
   printf("keyval %s: %s\n", dictionary[2].key,
                     (char*)dictionary[2].value);
}
' | gcc -xc '-'; ./a.out
```

Once you write a `find_key` function, this can work as a full-blown dictionary. [The thing about knowing the types on output can also be worked around via creative macros, but for most applications you don't need to.] Writing this function is left as an exercise to the reader, but it's just an instructional exercise, because fleshing this out and making it bulletproof has already been done by other authors; see the GLib's keyed data tables or `GHashTable`, for example. The point here is simply that having compound structs plus simple arrays equals a short hop to a dictionary. If you are coming from a dictionary-heavy idiom to the C family, then you'll have to split your dictionary uses into `struct`-like short lists and long lists, and use a structure out of GLib (or Boost, or whatever is appropriate) for the long homogeneous lists with a name index.

OK, so we've seen (last time) how Awk uses named lists to fake numbered lists. We put named lists into numbered arrays to generate key-value lists. What if we have only simple numbered arrays?

FORTRAN 77 (which is not Fortran 90 or later versions) lacks the ability to declare complex types. This is true of many of the punched card languages first developed in

the ‘60s and 70s. If you want a structure listing dimensions one through three, population count, and workspace size, then declare an integer array of size 5 and just remember that `iv[1]` through `iv[3]` are the dimensions, `iv[4]` represents population, `iv[5]` represents workspace size, and so on. This is what I’d above referred to as bad form, and the language all but forces you to do it. And now that you have your array of integers, set up another array `fv` for the floating-point values. [Exercise: given only numerically-indexed arrays of homogeneous types, how would you set up a key/value structure? If you wind up with four or five arrays, is there any way to bind them into one parent structure, so you don’t have to send all those arrays to every function that uses the structure? [answer: no, not in F77.]] If you want a linked list where item 4 points to item 2 which points to item 6, then declare an array of integers and write 2 in location 4 and 6 in location 2, and perhaps -1 in location 4 to indicate the start of the list. If that sentence confused you, try writing (or debugging) a whole program in that style.

Here’s an actual code snippet, a function call cut and pasted from archives of FORTRAN routines (I ran it through `f2c`; the R project uses this in the orignal FORTRAN):

```
ehg131(xx, yy, ww, &trl, diagl, &iv[20], &iv[29], &iv[3], &iv[2],
&iv[5], &iv[17], &iv[4], &iv[6], &iv[14], &iv[19], &wv[1] ,
&iv[iv[7]], &iv[iv[8]], &iv[iv[9]], &iv[iv[10]], &iv[iv[22]],
& iv[iv[27]], &wv[iv[11]], &iv[iv[23]], &wv[iv[13]], &wv[iv[12]],
& wv[iv[15]], &wv[iv[16]], &wv[iv[18]], &i__1, &wv[3], &wv[iv[26]],
&wv[iv[24]], &wv[4], &iv[30], &iv[33], &iv[32], &iv[41], &iv[iv[25]]
&wv[iv[34]], &setlf);
```

I could explain what `iv[1]` through `iv[41]` stand for, but it wouldn’t help. This is as write-only as code gets.

I’ve made some effort to translate some such code to C, and it’s something I really regret. The problem with code like this is not that it’s in a now-unpopular language, but that it’s in a language that doesn’t support named, heterogeneous data structures.

There’s more: objects (structs/dictionaries with functions in them), variants like Python’s tuples, sets, bags, and everything else you learned about in your data structures textbook. But if you’re a tourist to a new language and have to get things done fast, the above will be a good start. In an episode or two I’ll really expand this point.