Tip 63: Seamlessly extend C structures

Ben Klemens

5 February 2012

level: Compose structs
purpose: Inherit

[January 2015: It seems I have to add a caveat for this tip. Before I do, I'd like to point out that out of 400pp of useful things related to C that I've written about over the last few years, this tip is in the top two or three favorites (after optional and named function arguments (entry #100)). A lot of people really want this to work.]

[The caveat: the C11 standard and the Microsoft extension versions of this technique differ; an earlier version of this tip was somewhere between unclear and wrong about this point. Second, we could check MSFT documentation[1] and see that anonymous structs are defined, and verify that their use within a struct below is in line with that, but a documentation page is not exactly a standards document. Standards by MSFT are sometimes defined by instructions to do whatever the MSFT product does; see the ISO standardization of OOXML for examples.]

[This leaves compiler authors greater latitude to implement the standard how they see fit. This is reliable under gcc, as far as I can tell, and it works fine for my versions of clang, but I get reports from other clang users that it doesn't work for them, even with all the flags. If one of the authors of clang thinks that this tip doesn't conform with his or her reading of MSFT's documentation, there's no structured way to debate the point. But if any compiler authors are out there reading this: I can report that this is a very in-demand feature. Please take care to ensure that it works in a standardized way in future releases of your compiler. Thx.]

C11 allows us to include anonymous elements of a structure. Although this just got added to the standard, a variant of this was a Microsoft extension for a long time, and the GNU, ever the peacemakers, allows code to do this given the `-fms-extensions` flag on the gcc command line. Clang strives for gcc compatibility, and so joins the train as well.

After you set `CFILE`=your source file, both of these commands should compile the code via:

```
gcc -fms-extensions $CFILE -lm
# or
clang -fms-extensions -Wno-microsoft $CFILE -lm
```

Clang users, see the above caveat about how it may not work for some versions of clang. Also, it seems as if `-fms-extensions` should imply turning off warnings

[1] `https://msdn.microsoft.com/en-us/library/aa270923%28v=vs.60%29.aspx`

about MSFT extensions, but as of this writing you still need the `-Wno-microsoft` flag for clang.

The syntax: include another struct in the declaration of the new structure. All of the elements of the other structure are included in the new structure as if they were declared in place. Here's a textbook example, wherein we extend a 2-D point into a 3-D point. So far, it is pretty boring.

```
#include <stdio.h>
#include <math.h>

typedef struct pstruct {
   double x, y;
} point;

typedef struct {
    struct pstruct; //This is anonymous.
    double z;
} threepoint;

double threelength (threepoint p){
    return sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
}

int main(){
    threepoint p = {.x=3, .y=0, .z=4};
    printf("p is %g units from the origin\n", threelength(p));
}
```

OK: we've extended a structure seamlessly, to the point where users of the `threepoint` don't even know that its definition is based on another struct.

Before I get to the really awesome part, let me point out the first annoyance: it's not in the standard to use a typedef in the nested anonymous declaration, and the compiler authors seem to have extended that to this rendition of nested anonymous declarations as well. So you will see above that I include a name for the structure (`pstruct`), so we can use the typedef name in 100% of all other uses of the structure, and the `struct point` form for the folding-in.

Now for the trick that really makes this useful. Say that there is another function `length` that accompanied the a 2-D `point` structure. How are we going to use that function, now that we don't have a name for that subpart of the larger structure?

The solution is to use an anonymous `union` of a named `point` and an unnamed `point`. Being the union of two identical structures, the two structures share absolutely everything, and the only distinction is in the naming.

```
#include <stdio.h>
#include <math.h>

typedef struct pstruct{
   double x, y;
} point;
```

```
typedef struct {
    union {
        struct pstruct; //anonymous structure
        point p2; //structure named p2
    };
    double z;
} threepoint;

double length (point p){
    return sqrt(p.x*p.x + p.y*p.y);
}

double threelength (threepoint p){
    return sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
}

int main(){
    threepoint p = {.x=3, .y=0, .z=4};
    printf("p is %g units from the origin\n", threelength(p));
    printf("Its projection onto the XY plane is %g units from the origin\n", length(p.p2));
}
```

From here, the possibilities span $\Re^3$. You can extend any structure, and still use all of the functions associated with the original.

Multiple inheritance, if you're lucky If you want to inherit from multiple structures, just keep adding `unions`—but there is one more annoying restriction: we have to avoid name clashes. Here is a quick redux of the definitions of the GSL's vector and matrix structures, and a structure that inherits from both (so we can have the dependent and independent variables in a regression, or a vector with its covariance matrix):

```
typedef struct _gsl_matrix {
    size_t size1, size2;
    double *data;
} gsl_matrix;

typedef struct _gsl_vector {
    size_t size;
    double *data;
} gsl_vector;

typedef struct{ //Alas, this will fail.
    union {
        struct _gsl_vector;
        gsl_vector vector;
    };
    union {
        struct _gsl_matrix;
        gsl_matrix matrix;
    };
```

```
} data_set;

data_set d;
```

When we refer to `d.data`, are we referring to the `data` element of the matrix or the vector? Especially given that this sort of composition became possible in C a few weeks ago, the typical C structure is not particularly designed for composition, and we have no syntax for selective inclusion. The best we can do is pick the matrix or vector as primary and the other as secondary, callable only by its subelement name.

```
typedef struct{
    union {
        struct _gsl_vector;
        gsl_vector vector;
    };
    gsl_matrix matrix;
} data_set; //vector with supporting matrix.

data_set d;
```

Addendum: an alternative After adding that caveat at the top of this entry (here in January 2015), I feel I should offer one more alternative that sort of works everywhere.

There can't be padding at the head of a struct. Therefore, if you have a pointer to a larger struct whose initial definition is identical to a smaller struct, then you are guaranteed that a cast to the smaller struct will have all its bits in the right place. The new part in the following example is the pair of macros: one to define the guts of the smaller struct, and one to cast a pointer to the larger struct to a pointer to a smaller struct.

```
#include <stdio.h>
#include <math.h>

#define Pointguts double x, y;
#define As_point(instruct) (*(point*)(&instruct))

typedef struct pstruct{
    Pointguts
} point;

typedef struct {
    Pointguts
    double z;
} threepoint;

double length (point p){
    return sqrt(p.x*p.x + p.y*p.y);
}

double threelength (threepoint p){
```

```
    return sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
}

int main(){
    threepoint p = {.x=3, .y=0, .z=4};
    printf("p is %g units from the origin\n", threelength(p));
    printf("Its projection onto the XY plane is %g units from the origin\n", length(As_point(p)
        ));
}
```

The cast to a different type works safely, as long as the types remain in sync. Here I was able to do this with a macro, because I am the author of both elements. If you are extending existing code, you just have to hope that nobody goes back and changes the original struct definition. Also, the type cast is your way of telling the compiler that you know what you're doing and that the compiler should not check input types. If you tried `char x[]="hi."; As_point(x)`, the compiler won't complain and you'll get wrong answers in production.

So this works everywhere and conforms to the standard, but has risks associated with it which make clear why compiler authors would want something safer.