Paying the Sin Tax on Semantics: The Lowly Struct

Paul R. Potts

June 2006

This content started out as a post from 2006 in my now-retired Blogger blog, "Praise, Curse, and Recurse." The references to Scheme are because, at the time, I was frequently blogging about Scheme, and my posts were included in the aggregated blog Planet Scheme.

While this is not strictly about Scheme, I want to make a point about language design and the tax we pay (call it a *sin tax*, as the sins of the language designer are visited on generations of hapless language users) when there is no regularity in the syntax.

Programmers from the C/C++/Java world often look at languages like Scheme, with its relatively uniform syntax (or, I should say, almost a complete lack of syntax, where the language source is pretty much an textual form of an abstract syntax tree) and bemoan how strange it is.

But, as I will show, the alternative is actually far more complex: that alternative is to have *semantics* and *syntax* all jumbled together. The result is a jumbled mess that comes back to confuse generations of programmers again, and again, and again.

Said programmer with brains damaged from years of slogging away in C, C++, or Java might only really *see* this after working a bit with simple languages, such as Scheme, and then returning, as I've done with my embedded work. Going back is painful. To quote *Gollum's Song*:

> Where once was light
> now darkness falls
> where once was love
> love is no more

Let's look at one particular example which seems simple, but which is not, and which C and C++ programmers have come to accept. *The Unix-Hater's Handbook* describes this phenomenon with a quotation attributed to Ken Pier at Xerox PARC, which could also apply to the C family of languages:

> I liken starting one's computing career with Unix, say as an undergraduate, to being born in East Africa. It is intolerably hot, your body is covered with lice and flies, you are malnourished and you suffer from numerous curable diseases. But, as far as young East Africans can tell, this is simply the natural condition and they live within it. By the time they find out differently, it is too late. They already think that the writing of shell scripts is a natural act.

C has the concept of a **struct**, which is a sort of low-level record:

```
struct s_tag {
   int x;
   char y;
};
```

This defines an aggregate type. If you put names after the definition they define variables of that type:

```
struct s_tag {
   int x;
   char y;
} s_var_1, s_var_2;
```

Just like you define instances of an **int**:

```
int i_var_1;
```

You don't actually need a name for the type, unless you want to use it elsewhere; the following is legal:

```
struct {
    int x;
    char y;
} s_var_1, s_var_2;
```

However, even given the relative simplicity of this construct, we're already in a twisty maze of language features, all not quite alike! The **struct** is unusual in C in that it operates in a separate namespace reserved for **struct** and **union** tags, kind of like the way Common Lisp uses a distinct namespace for functions. While you normally define an instance of a variable like this (say, at file scope, or as a local variable inside a function):

```
int my_int;
```

Given the **struct** definition:

```
struct s_tag {
    int x;
    char y;
};
```

you cannot write:

```
s_tag s_var_1, s_var_2;
```

because the **struct** *tag* (the part that goes between the keyword **struct** and the opening curly bracket) is not a type in the general C namespace.

To access that namespace, you use the word **struct** again as a qualifier:

```
struct s_tag my_struct;
```

In practice, most C programmers ignore this second namespace, and create a *typedef*, which makes an *alias* of the **struct** tag in the general C namespace. This could be written:

```
struct s_tag {
   int x;
   char y;
};
typedef s_tag s_type;
```

but there is a commonly used shortcut, in which wrapping a **typedef** around the **struct** declaration changes its interpretation, and instead of defining variables, you are declaring a type:

```
typedef struct s_tag {
   int x;
   char y;
} s_type;
```

Since you aren't going to use the tag, you can leave it off altogether:

```
typedef struct {
   int x;
   char y;
} s_type;
```

and still use **s_type** as a type:

```
s_type my_struct;
```

Already, as you can see, the **struct** requires a kind of "little language" within C, with a lot of context-senstivity necessary to parse all this. And people thing Lisp's **format** is complicated! Other "little languages" include the way in which **for** loops and **switch** statements are written.

A "little language" is also known as a DSL, or *domain-specific language* — you see them used a lot in Lisp-style programming, and implemented using macros. But if we wanted to try to support these various syntactic and semantic variations using Lisp or Scheme-sytle macros, we couldn't do it — it isn't just a matter of replacing curly brackets with parentheses; there is no *underlying* common syntactic form. In other words, comprehending a language like C requires that we constantly change the behavior of the lowest level of the compiler, the *parser*, in a context-sensitive manner. This makes the full C language quite complex, and

don't even get me started on C++. It also imposes a burden on the programmer, which he or she will tend, over time, to forget is there — read the quotation from Ken Pier again!

There's another way in which structs are special. C supports a special form of initializer, called an *aggregate initializer*. This lets you specify the contents of a **struct** on initialization — that is, when it comes into scope, whether it is file scope or function scope — but not later, upon assignment, also known as *mutation*. In other words, the legal syntax changes *again* depending on context!

This makes it very clear to me that it is highly valuable to teach Scheme first, so that this tortured mess can be understood as it is, rather than as a set of *givens* about programming in general.

This aggregate initializer syntax can also be used for arrays, unions, and various nested combinations of the above. It also can in particular be used for **const** variables. For our example:

```
const s_type my_struct = { 1, 'a' };
```

If you leave off some of the values, the remainder of the **struct** will be initialized to zero; aggregate initializers may be nested; dimensions must be specified, or the compiler won't know how much data to supply. All well and good, and commonplace to C programmers. Simple, useful, and aside from some caution you must exercise about alignment and possible holes in the runtime representation, structs generally go about their business quite happily and never hurt anyone. (Of course, this is a lie; this shit causes unanticipated bugs and crashes every day. It would be more accurate to say that at this point we haven't introduced *that* much complexity, and so experienced developers who have been around the block have learned how to use these language elements with reasonable safety by applying some commonplace "best practices." This isn't the same as claiming that the language truly supports or helps to enforce this reasonable safety).

And then C++ came along.

One of the ways you can initialize objects in C++ is to provide an *initializer list* in your constructor. This is in fact the required method for initializing members of a class that are **const** or that are references, which are inherently **const**. In other cases, it is not required that you initialize members that way, but due to the rules about default member initialization this could mean that if you don't, but assign an initial value in the constructor body, you're wasting effort. Chalk this up to the distinction C and C++ makes between initialization and assignment; somewhat baffling to Scheme programmers, who would understand these both as variants on the underlying concept of *binding*.

The class could be declared like this:

```
class my_class {
    const int a;
```

```
    const int b;
};
```

The constructor is declared like this:

```
my_class( int a, int b );
```

and defined like this:

```
my_class::my_class( int a, int b ) :
   member_a( a ),
   member_b( b )
{
    /* body, which is often empty */
}
```

The initializer list is, essentially, made up of constructor calls.

What do you do if you have a **struct** in your class?

```
class my_class {
    const s_type my_struct;
}
```

Can you initialize it like this?

```
my_class::my_class( int a, int b ) :
   my_struct( { a, b } )
{
    /* body, which is often empty */
}
```

Naturally, after using Scheme for a while, I tend to presume that programming languages are at least somewhat regular. There's an aggregate initializer syntax (really, another tiny DSL) that you can use to initialize a **struct**, right? So can I use it here?

The answer is "no, that would be too easy!"

If you are accustomed to Scheme, you have become used to a language that is relatively *regular* — different language forms, which return values, can be used just about everywhere it might make sense to do so. But C++ was not designed like that. There is no *underlying* syntax that is common despite the changed context.

In particular, although compatibility with C was a major goal of the language design, C++ broke this compatibility in several ways:

1. ***struct** types can be declared directly in the common namespace.*

In other words, you can just say

```
struct s_type {
  /* ... */
};
```

without having to **typedef** it, and simply use **s_type** freely as a type. The **typedef struct** form still works, but you've introduced an *alias* for the type; this is not usually an issue since C programmers tend to use prefixes, or suffixes, or some Hungarian-like name mangling to try to keep them straight. This has the side benefit of allowing you to use header files containing the same **struct** declarations in plain old C, assuming you keep those headers free of other C++-specific code.

2. Structs are classes.

It seems like a strange design decision if a design goal was really to maintain as much compatibility with C++ as possible, but when Bjarne Stroustrup designed C++, he apparently decided that a **struct** was just a **class**, with different default access rules (all members are public by default). The C construct known as the **union** is also a class now, but that's another story for another day.

But despite this surface uniformity, C++ is still a *hybrid* language. It isn't "objects all the way down," like Ruby, or Dylan, where even an integer is an object, with a class. Which means that a lot of other language constructs act kind of like classes, in some contexts — except when they don't.

When you read Stroustrup's book *The C++ Programming Language, Special Edition*, when you are expecting to look at examples of construction and initialization in classes, he often throws in a **struct** instead. The C++ Standard document often does the same thing. (Warning: reading the C++ Standard document can cause persistent brain damage).

This also means it is perfectly legal to adorn the good old **struct** with **public**, **protected**, and **private**, and to provide constructors, assignment operators, and various other methods.

In fact, if you want to use a **struct** as a **const** member of a class, which means that it must be initialized using the *initializer list* syntax described above, you must do so by giving the **struct** a constructor. You can do this inline as follows:

```
struct s_type {
   int x;
   char y;
   s_type ( const int init_x, const char init_y )
       { x = init_x; y = init_y; }
};
```

Or you can just declare the constructor in the header and place the constructor definition in your implementation file, and you even… wait for it… use the same *initializer-list* syntax:

```
struct s_type {
    int x;
    char y;
    s_type ( const int init_x, const char init_y );
}

s_type::s_type( const int init_x, const char init_y ) :
    x( init_x ),
    y( init_y )
{
    /* empty constructor body */
}
```

And *then*, finally, in your class-containing-a-struct's initializer list, you can do this:

```
my_class::my_class( int a, int b ) :
   my_struct( a, b )
{
    /* body, which is often empty */
}
```

All that syntax, just to initialize a structure member, because Stroustrup decided that initializer lists would actually be lists of constructor calls, except when they aren't quite.

But what if you don't want to bother with that, but just want to set up the value of your member **struct** in the constructor body, like this:

```
my_class::my_class( int a, int b )
    /* no initializer list */
{
    my_struct( a, b );
}
```

The answer is: *you can't!* Because this isn't initialization; it is assigment. This is true even if the **struct** member of your class is *not* **const**.

However, you can *assign* your **struct** in the constructor body:

```
my_class::my_class( int a, int b )
    /* no initializer list */
{
    s_type temp_struct = { 0, 'a' };
    my_struct = temp_struct;
}
```

and this is legal, because it isn't initialization.

If you want to maintain full compatibility with C and expose your header files to a standard C compiler, you will need to perform just this kind of awkward hack

to use structs as members. This is apparently what maintaining full backward compatibility with C meant to Stroustrup: that is, it didn't mean as much as indulging his trivial observation that structs behaved like classes with public data members. Sort of.

Or, if you don't need your **struct** instance to be per-member (and if it is const, you probably don't), you can make it **static**, which means it is initialized at file scope:

```
class my_class {
    static const s_type my_struct;
}
const my_class::my_struct = { 0, 'a' };
```

Do you have a headache yet?

It is surprisingly difficult to get this information out of C++ books, even Stroustrup's books, and the C++ standard document: see the end notes, below. Understanding the usage in C++ basically requires an archaeological excavation of hacks: the original, oddly designed two-namespace **struct** in C, the common workarounds, the semantics of classes, Stroustrup's somewhat odd mutation of **struct** into **class**, and the strict distinction between initialization and assignment. (Not that there aren't a few things in Common Lisp that feel similarly weighted with historic baggage!

I know this hasn't quite been about Scheme, but I am just trying to point out how C and C++ programmers have come to live with this strange mix of syntax-directed semantics, and semantic-specific syntax, where constructs are legal depening on their context. I've been using C and C++ on and off for about twenty years, but have only very recently fully understood *why* I can't initialize structs in what seems like the logical, backwards-compatible way. Maybe you think you can just look it up in the index. Good luck! And if you think you can just look this up in the language's BNF, think again!

C and C++ programmers live with these restrictions and special cases every day, but yet think that functional programming in a language like Scheme is somehow hard. It isn't — it is freedom!

Notes

The compiler wasn't cooperating, so I asked myself "does C++ really forbid initialization of a **struct** member using aggregate initializer syntax (with the curly brackets) in a constructor initialization list?"

Stroustrup, *The C++ Programming Language, Special Edition* p. 101, 234, 809, 818:

Stroustrup actually seems to advocate freely intermixing **struct** and **class** depending on your intentions for access control. Regarding initialization of

structs, he writes "using a constructor [to initialize a **struct**] is usually better." But, of course, that breaks compatibility with one of the simplest but most highly useful tools of C, and the struct was never designed to support object-oriented programming.

Since the **struct** tag is promoted to a type in the general namespace, in order to maintain compatibility with C, which is broken anyway, there is a further workaround: although there is not a distinct namespace for structs, you can have a **struct** and non-**struct** type with the same name declared *in the same scope*, but the non-**struct** will take precedence unless you disambiguate with the keyword **struct**. Stroustrup's examples complicate things even further by disambiguating using namespaces, which are another feature not present in C.

According to Stroustrup's somewhat limited version of the grammar of C++, there isn't any *aggregate initialization* in *mem_initializer_list* (p. 810), but the grammar is not complete. Page 247 seems to indicate that items in the *member initializer list* in the constructor definition are all *constructor calls*, given that even POD members (plain old data, like **int** and **char** and **float** and pointers to these types) can be initialized with this constructor call syntax.

In C++ Std. section 8.4: the grammar shows **ctor-initializer** and **mem-initializer-list** (where "mem" means "member.") This is described in 12.6.2 and seems to break down further into the formal grammar into *class names* and parameter lists, which seems to indicate that these are all constructors, but again this does not seem to be a complete syntax for C++. Good luck trying to find a complete grammar for C++! And happy programming!
