

Re: syntax error

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2005-12/msg00511.html

- *From:* Chris Torek <nospam@xxxxxxxx>
 - *Date:* 4 Dec 2005 18:38:35 GMT
-

>> Daniel Rudy <spamthis@xxxxxxxxxxxx> writes:

```
>>>typedef struct ImageT_tag__ {
>>> char magicNum[2];
>>> int width;
>>> int maxGrey'
>>> int pixels[ROW][COLUMN];
>>>} ImageT;
>>>ImageT ImageTvar;
>>>
>>>fscanf(infile, "%c", &ImageT.magicNum[0]);
```

>At about the time of 12/2/2005 1:17 AM, Keith Thompson stated the following:

```
>> This fixes the type error from the original code, but it has the same
>> problem that it uses ImageT as if it were a variable name, when in
>> fact it's a type name. Presumably you meant
>>
>> fscanf(stdin, "%c", &ImageTvar.magicNum[0]);
>>
>> It's always a good idea to compile your code before posting it; that
>> would have caught both this error and the syntax error in the
>> declaration of maxGrey.
```

In article <6wCkf.35980\$6e1.34486@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx>

Daniel Rudy <ZGNydWR5QHBhY2JlbGwubmV0> wrote:

```
>Yeah, I was dead tired and on my way to bed when I wrote that. I make
>no excuse though.
```

"foo.c, line 27: warning: code written and posted by tired programmer" :-)

(Or, as another joke in math / physics goes, "don't drink and derive", although in that case, the brain fatigue is from something other than inappropriate neurotransmitter levels.)

```
>>>... I've actually
>>>had code that refused to compile unless I placed an identifier after the
>>>struct. Is that part of the standard, or is it just a gcc thing?
```

```
>> As far as I know, it's neither. Can you show an example?
```

Re: syntax error

>I stand corrected, it was a warning, but the type name didn't register
>properly which caused errors later on. Look at the struct above, but
>omit the ImageT_tag__ identifier. I was using gcc at the time.
>Something about useless statement.

I think you are mixing up a couple of concepts again here. GCC (and some other compilers) will gripe -- usually with a "warning" rather than an "error" -- if you do something like this:

```
static int;
```

```
and:
```

```
typedef double;
```

both of which are useless since they do not declare any names to be "int"s (in the first case) or "double" aliases (in the second).

Now, as it happens, there are a lot of people who -- for whatever reason -- internalise the "struct" definition syntax incorrectly. The actual syntax is:

```
struct some_tag_name {  
... contents ...  
} /* optionally, variable names here too */;
```

where the keyword "struct" and the braces are required, the "structure tag" name is sometimes optional, and the "contents" need to include at least one type-and-member-name.

The tag is in fact *required* whenever any structure is to contain a pointer to this structure (which has not yet been completely defined, because we have not yet reached the close "}" that ends the definition of the members). One very common case in which this structure is to contain a pointer to another instance of itself, as in a linked-list:

```
struct foo_list {  
struct foo_list *next;  
...  
};
```

Here, regardless of the use (or lack thereof) of typedef-aliases, the tag is required.

Because a tag is sometimes required, I encourage neophyte C programmers to use one every time, not just when it is required. This eliminates the need to think about whether the tag name is required. Of course, it replaces this need with the need to think of a good name, which may be even harder; but thinking of good

Re: syntax error

names is good practice, too. :-)

Where people seem to become confused is when they start using "typedef".

The syntax for C's typedef is quite odd. In other languages, variable and type declarations tend to have wildly different syntaxes:

```
var i, j: integer;
type my_type_index is integer range 0 .. 999;
type my_type_struct is record
... list of "record" entries ...
end record;
var x: my_type_index;
var s: array [my_type_index] of my_type_struct;
```

Here a variable declaration lists the variables, then uses a punctuator (":"), then gives the type of those variables; but a type-definition names the type, then uses a keyword ("is"), then states what the type is supposed to be.

C, on the other hand, declares variables by having the programmer write down a type-name, then list the variables along with any modifiers. To declare a type alias (because typedef does not create a new type, just an alias for an existing one), C has the programmer write the type and the alias as if it were a variable declaration, then prefix the whole shebang with the keyword "typedef":

```
int i, j;
typedef int my_type_index; /* C does not have subranges */
typedef struct my_type_struct {
... list of entries ...
} my_type_struct;
my_type_index x;
my_type_struct s[1000]; /* can't use a subrange to define the range */
```

This makes typedef act, syntactically speaking, like a storage-class specifier ("register", "static", or "auto"):

```
void f(void) {
register int i, *j;
static int k, *m;
typedef int n, *p, q;
...
}
```

Here n, *p, and q are all type-aliases for "int". Since *p is an alias for "int", p must be an alias for "pointer to int".

The critical thing to remember here is that typedef is just pasted

Re: syntax error

on in front of a regular variable declaration. Inside the compiler, the "typedef" keyword sets a flag:

"Whenever you are about to enter a variable name into your tables, instead of marking it as `x is a variable that has type T', mark it instead as `x is an alias for type T'."

Thus, if p would have been a variable of type "pointer to int", it gets marked as "alias for pointer to int" instead.

Now, suppose we define a new structure type:

```
struct mylist {
struct mylist *next;
char *name;
int value;
};
```

We can then define some variables with that type:

```
struct mylist second_item; /* forward declaration */

struct mylist first_item = { &second_item, "hello", 1 };
struct mylist second_item = { NULL, "world", 2 };
```

But we can sneak those in with the definition of the structure type too (and at the same time, reverse the order to get rid of the forward declaration):

```
struct mylist {
struct mylist *next;
char *name;
int value;
} second_item = { NULL, "world", 2 },
first_item = { &second_item, "hello", 1 };
```

But if we can declare variables, we can stick a typedef keyword on the front of the whole thing and turn the variables into type-names. (Of course, we have to get rid of the initializers: only variables, not type-aliases, can be initialized.) This gives something like:

```
typedef struct mylist {
struct mylist *next;
char *name;
int value;
} ALIAS1, *ALIAS2, ALIAS3[100];
```

With this monster combination construct, we have simultaneously declared a new type ("struct mylist") and defined it ("{ ... }"), then declared three aliases that are connected with it. The first

Re: syntax error

(ALIAS1) is an alias for "struct mylist"; the second (ALIAS2) is an alias for "struct mylist *"; and the third (ALIAS3) is an alias for "struct mylist [100]" -- a type-name that looks peculiar in C, because the name of the variable itself is missing; but this is how one turns variable-names into type-names (just drop the variable-name).

[Aside: this is why C's pointer-to-function types look so odd. If fp is a variable of type "pointer to function taking double and returning int", it is declared as:

```
int (*fp)(double);
```

which looks bad enough to start with. To turn that into a type name instead of a variable name, we drop the variable's name (and the semicolon):

```
int (*)(double)
```

Typically this type-name would be used in a cast, so the whole thing would get one additional set of parentheses, and be placed in front of some other expression. We might use that to convert a value stored in a variable of type "pointer to function (void) returning void" to the presumably-correct type, and call the resulting function:

```
int result;
...
result = ((int (*)(double))lookup("func"))(3.14159265358979323846);
```

This would likely be more readable and maintainable if we used several intermediate variables, though.]

Returning to structures and typedefs, though: for some reason, a lot of programmers seem to wind up thinking that it is "typedef" that defines types. (Perhaps it has to do with the fact that the word typedef is an obvious contraction for "type define", which logically ought to define types. But C is C: typedef instead means "define a new alias, not a new type at all", just as "static" has nothing to do with electricity and "auto" is not a car, which is good since you cannot "register" it with the motor vehicle department either. :-))

The crucial point here, really, is that typedef does not define new types. Instead, it turns what would have been variable declarations into type-alias declarations. Declare some variables, stick "typedef" on the front, and you have made some type-aliases instead. To define a truly *new* type in C, your best bet is the "struct" keyword:

```
struct my_type { double value; };
```

Re: syntax error

makes a new type that is different from every existing type, even one that has the same contents:

```
struct temperature { double value; };
struct pressure { double value; };

void f(void) {
  struct temperature hot;
  struct pressure high;
  ...
  hot = high; /* ERROR --- diagnostic required */
  ...
}
```

(You can also define new types with "union" and "enum", but the latter two are limited; "struct" is not. Being unlimited is often a mixed blessing — "goto" is unlimited while structured looping constructs like "for" and "while" are limited — but in this case, doing a cost/benefit analysis for struct vs union-or-enum usually results in choosing "struct", at least for me.)

—
In-Real-Life: Chris Torek, Wind River Systems
Salt Lake City, UT, USA (40°39.22'N, 111°50.29'W) +1 801 277 2603
email: forget about it <http://web.torek.net/torek/index.html>
Reading email is like searching for food in the garbage, thanks to spammers.

• *Follow-Ups:*

- ◆ **Re: syntax error**
◇ From: Daniel Rudy

• *References:*

- ◆ **syntax error**
◇ From: danu
- ◆ **Re: syntax error**
◇ From: Daniel Rudy
- ◆ **Re: syntax error**
◇ From: Keith Thompson
- ◆ **Re: syntax error**
◇ From: Daniel Rudy

- Prev by Date: **Re: A simple program with Monte Carlo**
- Next by Date: **Re: Regarding Q. 14-5**
- Previous by thread: **Re: syntax error**
- Next by thread: **Re: syntax error**
- Index(es):
 - ◆ **Date**
 - ◆ **Thread**