

Re: struct type completion

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2005-04/msg02100.html

- *From:* "S.Tobias" <siXtY@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* 26 Apr 2005 00:05:25 GMT
-

Lawrence Kirby <lknews@xxxxxxxxxxxxxxxx> wrote:
> On Mon, 18 Apr 2005 19:03:54 +0000, S.Tobias wrote:

>> Lawrence Kirby <lknews@xxxxxxxxxxxxxxxx> wrote:
>>> On Mon, 18 Apr 2005 15:12:46 +0000, S.Tobias wrote:

(I gave more context for the first quote.)

```
# [#23] An array type of unknown size is an incomplete type.  
# It is completed, for an identifier of that type, by  
# specifying the size in a later declaration (with internal or  
# external linkage). A structure or union type of unknown  
# content (as described in 6.7.2.3) is an incomplete type. It  
# is completed, for all declarations of that type, by  
# declaring the same structure or union tag with its defining  
# content later in the same scope.
```

What I think what this says regarding the structs is this:

With array types:

```
/* file scope */  
int a[], b[];  
(`a' and `b' have same type here)  
int a[5];  
int b[7];  
Now we have re-declared and the identifiers `a' and `b' have different  
types. We need to re-declare the identifiers to complete their type.
```

With struct types this is a bit different:

```
struct s *p;  
struct s o; /* in file scope only */  
(*p' and `o' expressions have same, incomplete type)  
struct s { ... };  
The above definition of `struct s' also causes all previous identifiers  
to be *implicitly* re-declared with the new complete type. So from now on  
`o' and `*p' are complete types.
```

I think the following change would improve that fragment:
[...] It is completed, for all declarations [of identifiers]

Re: struct type completion

of that type, by [...]

```
# [#3] All declarations of structure, union, or enumerated
# types that have the same scope and use the same tag declare
# the same type. The type is incomplete until the
# closing brace of the list defining the content, and complete
# thereafter.
```

I have thought about this one a lot, but I can't reach any conclusion.

```
>>>> So (eg. in a file scope):
>>>> /* 1 */ struct mystruct object;
>>>> /* 2 */ struct mystruct { /*...*/ };
>>>> line 1 defines `object' (although at this point the body of `struct
>>>> mystruct' is not yet known(?)); and line 1 does not work without
>>>> line 2.
>>
>>> Line 1 is invalid.
[snip]
> Yes, my error. Tentative definitions are allowed to have incomplete type
> as long as they don't have internal linkage, I overlooked the last part.
```

Mine too – I thought it would always work, but I hadn't checked it (thanks, Old Wolf). In fact your previous example with a pointer to struct was better (more general) – I'm sorry for snipping it away too hastily.

```
[...]
> I guess it comes down to what it means by "the same type".
```

8–O You aren't a lawyer, are you? ;–)

```
[snip]
```

```
>>> At the point that line 1 is translated its type is incomplete. For any
>>> code after line 2 it would appear complete (except of course that it is
>>> invalid).
>>
>>> I thought so too. But this seems to contradict eg. the second quote
>>> (both in fact): both declarations use the same tag, therefore they
>>> declare the *same* type; a type cannot be both complete and incomplete.

> Not at a particular point in the code, but it can at different points in
> the code. It is much the same issue as with function prototypes. If you
> have
```

Re: struct type completion

Re: struct type completion

> void foo();

> foo(x);

> void foo(void);

> foo(x);

> then the rules that apply to the first foo(x); are different to the
> second: the second sees the composite type that includes the prototype.

I don't see what you're trying to show me, I don't see anything strange in here: `foo` is simply re-declared, it had one type in the first case, and a different type in the second case (it's allowed for identifiers that have linkage).

It's similar as in my first example a few pages up: both `a` and `b` start with the same incomplete type, but then each identifier is re-declared (in different ways) and acquires a different (complete) type (actually: composite type). The initial and final types are different.

(It's impossible to make similar parallel for structs, because their completion is one-way only – a single struct definition automatically completes all identifiers in the same scope.)

>>> A type can be incomplete at one point in the source file and then complete
>>> later on.

>> Hmmmm.... I don't have an answer to that, but I strongly disagree.

I think that an object type can either be complete or incomplete, but never both.

"Types are partitioned into object types (types that fully describe objects), function types (types that describe functions), and incomplete types (types that describe objects but lack information needed to determine their sizes)." (6.2.5p1)

For the first thing, I think the word "partitioned" determines it. For another, a type cannot *fully* describe an object and lack information about its size at the same time.

When the Standard says "a type is completed", I think it means a new type is created. I have grep'ed it for "complete" (to find "incomplete", "complete", "completed", "completely" etc.), and I haven't found anything to suggest unambiguously anything opposite.

Re: struct type completion

For the last thing: "The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit." (6.7.2.1p7), therefore I think that in:

```
struct s;  
struct s { ... };  
the types must be different (second type is a *new* type).
```

>> If one is incomplete, the other complete, then they're different types,
>> I thought that was pretty logical. Could you give some example, or
>> a reference (c&v), please?

> The wording of the standard you quote above depends on this. A type can
> be incomplete at one point in the source code and complete later on. "The
> same type" here is simply the opposite of "distinct types".

> Consider

```
> {  
> struct foo { int x } a;  
> }
```

```
> {  
> struct foo { int x } b;  
> }
```

> a and b have 2 distinct types here, and they are incompatible.

Yes, so what?

>Given your
> example

```
>>> /* 1 */ struct mystruct object;  
>>> /* 2 */ struct mystruct { /*...*/ };
```

[a small "enhancement":]

```
/* 1 */ struct mystruct object1;  
/* 1a */ struct mystruct object1a;  
/* 2 */ struct mystruct { /*...*/ } object2;
```

> the fact that the 2 lines refer to the same type means that the completing
> of the type in the second line affects the type of object because they are
> the same type. So after line 2 (sizeof object) is valid, which it wouldn't
> be if they weren't the same type. I think that is all it is saying.

In my view, `object' changes its type[*] after the second declaration, that's why sizeof object is okay after the second declaration. However, both declarations declare distinct types.

Re: struct type completion

Re: struct type completion

[*] not because the declarations are "the same type", but because the first declaration with the same tag is in the same scope, and this case is covered by the first quote (I think).

```
# [#3] All declarations of structure, union, or enumerated
# types that have the same scope and use the same tag declare
# the same type.
```

Following the letter, the above excerpt concerns only (type, not object) declarations 1 and 2, and not 1a, because 1a is not a declaration of a type (because previous declaration is visible – cf. 6.7.2.3p7,8), it merely specifies the same type as in 1. Both type declarations 1 and 2 have the same scope and the same tag, therefore (according to the quote) they declare the same type. (I think that's clear – my understanding and deduction skills are still in order... right?).

1. I don't agree with that (which I have outlined above).
2. I don't see any need for that. Can you show what would change if that particular sentence was not there?

Maybe it's just an error in the Standard, or maybe I need a life.

—
Stan Tobias
mailx `echo siXtY@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx | sed s/[[:upper:]]//g`
.

• **References:**

- ◆ **struct type completion**
 ◇ From: S.Tobias
- ◆ **Re: struct type completion**
 ◇ From: Lawrence Kirby
- ◆ **Re: struct type completion**
 ◇ From: S.Tobias
- ◆ **Re: struct type completion**
 ◇ From: Lawrence Kirby

- Prev by Date: **Re: C's trig functions**
- Next by Date: **Re: C's trig functions**
- Previous by thread: **Re: struct type completion**
- Next by thread: **Reader writer problem**
- Index(es):
 - ◆ **Date**
 - ◆ **Thread**