

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required.

Take the 2-minute tour x

Strange compiler warning C: warning: 'struct' declared inside parameter list

CAREERS 2.0
by stackoverflow



Easily apply for your dream job
No formatting needed!

I just found a quirk in C that I find really confusing. In C it's possible to use a pointer to a struct before it has been declared. This is a very useful feature that makes sense because the declaration is irrelevant when you're just dealing with a pointer to it. I just found one corner case where this is (surprisingly) not true, though, and I can't really explain why. To me it looks like a mistake in the language design.

Take this code:

```
#include <stdio.h>

#include <stdlib.h>

typedef void (*a)(struct lol* etc);

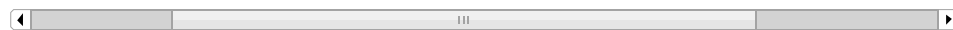
void a2(struct lol* etc) {

}

int main(void) {
    return 0;
}
```

Gives:

```
: 'struct lol' declared inside parameter list [enabled by default]
: its scope is only this definition or declaration, which is probably not what you
: 'struct lol' declared inside parameter list [enabled by default]
```



To remove this problem we can simply do this:

```
#include <stdio.h>

#include <stdlib.h>

struct lol* wut;

typedef void (*a)(struct lol* etc);

void a2(struct lol* etc) {

}

int main(void) {
    return 0;
}
```

The unexplainable problem is now gone for an unexplainable reason. Why?

Note that this question is about the behavior of language C (or possible the compiler behavior of gcc and clang) and not the specific example I pasted.

EDIT:

I won't accept "the order of declaration is important" as an answer unless you also explain why C would warn about using a struct pointer for the first time in a function argument list but allow it in any other context. Why would that possibly be a problem?

c

edited May 30 '13 at 9:06

asked May 30 '13 at 8:44

 Hannes Landeholm
166 3 15

2 You should still tell it the struct exists in advance using `struct lol;` – Dave May 30 '13 at 8:47

You should first tell the compiler the exists such type as a struct lol and then you can use struct lo or a pointer to the struct in the declaration of a new function – hetepeperfan May 30 '13 at 8:47

2 Also it looks like the compiler explained the exact problem to you: scope. Read the warnings! – Dave May 30 '13 at 8:48

[add a comment](#)

3 Answers

To understand why the compiler complains, you need to know two things about C "struct"s:

- they are created (as a declared, but not yet defined, type) as soon as you name them, so the very first occurrence of `struct lol` creates a declaration
- they obey the same "declaration scope" rules as ordinary variables

(`struct lol {` declares and then begins defining the structure, it's `struct lol;` or `struct lol *` or something else that does not have the open-brace that stops after the "declare" step.)

A struct type that is declared but not yet defined is an instance of what C calls an "incomplete type". You are allowed to use pointers to incomplete types, as long as you do not attempt to follow the pointer:

```
struct lol *global_p;
void f(void) {
    use0(global_p);    /* this is OK */
    use1(*global_p);   /* this is an error */
    use2(global_p->field); /* and so is this */
}
```

You have to complete the type in order to "follow the pointer", in other words.

In any case, though, consider function declarations with ordinary `int` parameters:

```
int imin2(int a, int b); /* returns a or b, whichever is smaller */
int isum2(int a, int b); /* returns a + b */
```

Variables named `a` and `b` here are declared inside the parentheses, but those declarations need to get out of the way so that the the *next* function declaration does not complain about them being re-declared.

The same thing happens with `struct` tag-names:

```
void gronk(struct sttag *p);
```

The `struct sttag` declares a structure, and then the declaration is swept away, just like the ones for `a` and `b`. But that creates a big problem: the tag is gone and now you can't name the structure type ever again! If you write:

```
struct sttag { int field1; char *field2; };
```

that defines a new and different `struct sttag`, just like:

```
void somefunc(int x) { int y; ... }
int x, y;
```

defines a new and different `x` and `y` at the file-level scope, different from the ones in `somefunc`.

Fortunately, if you declare (or even define) the struct *before* you write the function declaration, the prototype-level declaration "refers back" to the outer-scope declaration:


```
struct sttag;
void gronk(struct sttag *p);
```

Now both `struct sttag`s are "the same" `struct sttag`, so when you complete `struct sttag` later, you're completing the one inside the prototype for `gronk` too.

Re the question edit: it would certainly have been possible to define the action of struct, union, and enum tags differently, making them "bubble out" of prototypes to their enclosing scopes. That would make the issue go away. But it wasn't defined that way. Since it was the ANSI C89 committee that invented (or stole, really, from then-C++) prototypes, you can blame it on them. :-)

edited May 30 '13 at 9:13

answered May 30 '13 at 9:02

 torek
25.3k 2 22 38

Thanks. I think this was the most insightful answer. – Hannes Landeholm May 30 '13 at 9:15

2 I was around when they invented these (prototype scopes, and all that). People discovered the problem with

struct declarations inside prototypes when they started using prototypes—often getting incomprehensible error messages from compilers that did the type checking, but just said `struct foo *` is not compatible with `struct foo *` without explaining *why*. That was... interesting. :-)) — [torek](#) May 30 '13 at 9:19

[add a comment](#)

CAREERS 2.0
by stackoverflow



+



Have projects on SourceForge?
Import them easily to your profile

This is because, in the first example, the `struct` is previously undefined and so the compiler tries to treat this first reference to that `struct` as a definition.

In general, C is a language where the order of your declarations matters. Everything you use should be properly declared in advance in some capacity, so that the compiler can reason about it when it's referenced in other context.

This is not a bug or a mistake in the design of the language. Rather, it's a choice that I believe was made to simplify the implementations of the first C compilers. Forward declarations allow a compiler to translate the source code serially in one pass (as long as some information such as sizes and offsets is known). If this weren't the case, the compiler would have to be able to go back and forth in the program whenever it meets an unrecognized identifier, requiring its code emission loop to be much more complex.

[edited May 30 '13 at 9:11](#)

[answered May 30 '13 at 8:47](#)



[Theodoros Chatzigiannakis](#)

7,065 2 6 34

1 Technically, you can still do it all in "one pass", you just have to keep a lot more data in memory. Works out similarly in terms of simple vs complex in the compiler, of course. :-)) — [torek](#) May 30 '13 at 9:08

@torek True. I'll see if I can phrase it more accurately. — [Theodoros Chatzigiannakis](#) May 30 '13 at 9:09

[add a comment](#)

The compiler is warning you about a *forward declaration* of `struct lol`. C allows you to do this:

```
struct lol;    /* forward declaration, the size and members of
               struct lol are unknown */
```

This is most used when defining self-referencing structs, but it is also useful when defining private structs that are never defined in the header. Because of this latter use case, it is allowed to declare functions that receive or return pointers to incomplete structs:

```
void foo(struct lol *x);
```

However, just using an undeclared struct in a function declaration, as you did, will be interpreted as a *local* incomplete declaration of `struct lol` whose scope is constrained to the function. This interpretation is mandated by the C standard, but it is not useful (there is no way to construct the `struct lol` to pass to the function) and is almost certainly not what the programmer intended, so the compiler warns.

[answered May 30 '13 at 8:58](#)



[user4815162342](#)

26.1k 1 20 47

[add a comment](#)

Not the answer you're looking for? Browse other questions tagged [c](#) or [ask your own question](#).