Unspooky action at a close range: leaking declarations through expressions

Category: Defect Report
Audience: WG14
Author: Elijah Stone
Date: 2021-02-13

Abstract

The rules restricting the declaration part of a `for` statement are ambiguous. The most commonly implemented interpretation creates inconsistencies. I propose a disambiguation of the semantics, and a number of potential solutions to the inconsistency.

Problem

The C specification makes the following statement regarding `for` statements [ISO/IEC 9899:2011 §6.8.5p3]:

> The declaration part of a `for` statement shall only declare identifiers for objects having storage class `auto` or `register`.

This can be interpreted in at least two ways:

1. All identifiers declared by the declaration part of a `for` statement shall be identifiers for objects having storage class `auto` or `register`

2. All objects for which identifiers are declared by the declaration part of a `for` statement shall have storage class `auto` or `register`

Compilers vary in their interpretation of this clause; ICC, MSVC, and CompCert follow interpretation 2, while GCC and Clang are closest to interpretation 1[1]. *The New C Standard: An Economic and Cultural Commentary* also supports interpretation 1 [§1765], saying:

[1] Clang is inconsistent in its enforcement. GCC consistently follows the spec, but has odd interactions with statement expressions, a nonstandard extension.

Problem
Consider the code:

```
for (enum fred { jim, sheila = 10 } i = jim; i < sheila; i++)
        // loop body
```

Proposed Committee Response
The intent is clear enough; `fred`, `jim`, and `sheila` are all identifiers which do not denote objects with `auto` or `register` storage classes, and are not allowed in this context.

However, as I will show, interpretation 1 leads to inconsistencies.

It is possible to 'leak' tag definition through expressions, as in the following example:

```
sizeof(struct foo { int m; });
struct foo x = {5}; //ok
```

Thus, if the *New C Standard* is correct about the intent of the paragraph, it doesn't actually succeed in its secondary intent (avoiding unwanted definitions), and creates a major inconsistency. Why are the first two below code snippets valid, but the third is not?

```
// 1, ok
int i = sizeof(struct foo { int m; });

// 2, ok
int i;
for (i = sizeof(struct foo { int m; });;);

// 3, not ok
for (int i = sizeof(struct foo { int m; });;);
```

Proposed solutions

I propose a number of potential solutions to this problem. All involve disambiguating the originally contentious paragraph. None are likely to impact existing code; the first three keep complete compatibility, while the last breaks it in a way which is likely to have minimal impact.

Proposal 1: Disambiguate in favour of interpretation 2

One possibility is to change §6.8.5p3 to unambiguously indicate 'interpretation 2' mentioned above, making it read as follows:

> All objects for which identifiers are declared by the declaration part of a `for` statement shall have storage class `auto` or `register`.

This change maintains compatibility with interpretation 1; conformant code previously written in accordance with it will continue to be conformant. Meanwhile it still disallows `extern`, `static`, `typedef`, and `_Thread_local` declarations.

Proposal 2: Specifically disallow identifier-introducing declarators

An additional stipulation could be added on top of proposal 1, specifically disallowing the declaration of a tag or enumeration as part of declaration (but not initialization). That is, the following wording or similar should follow the replaced §6.8.5p3:

> The declarator part of the declaration part of a `for` statement, if present, shall not declare or forward declare any structure, union, or enumeration tag; nor shall it declare any enumeration.

Though this stipulation is likely to be closest to the original intent of the paragraph, I do not recommend its inclusion in the standard as-is (though see below). The guarantees it provides are weak, and it's complex and somewhat arbitrary.

Proposal 3: Remove the problematic clause entirely

Another possibility is to simply remove the originally problematic paragraph. This is strictly more expressive than the alternative and provides no maintenance burden to compiler writers (in fact, it alleviates it somewhat; a number of smaller C compilers I tested did not reject code in violation of any interpretation of the clause). While it is unlikely to be strictly *useful* to most application writers, it is also unlikely to cause bugs, and the default storage class remains `auto` (which is likely to be correct in most cases).

Proposal 4: Prevent type declarations from escaping

The original source of the inconsistency was the way in which declarations contained in expressions could leak into the surrounding environment. In particular, the following all contain type names and can thus leak declarations:

- Generic associations
- Compound literals

– `_Alignof` and `sizeof` expressions

– Cast expressions

This escaping behaviour is not widely used and is unlikely to be desirable. Though its removal would comprise a compatibility break, it would not be likely to break existing code. This is my favoured solution. I propose making all of the aforementioned expressions block scopes, so that declarations cannot leak from them. This entails the following changes:

Add to the 'semantics' section of §6.5.1.1:

> A generic selection is a block whose scope is a strict subset of the scope of its enclosing block.

Note that this means type declarations in generic associations are available from the entirety of the remainder of the generic selection, as in this example:

```
_Generic(0,
         struct foo { int x; }: 0,
         default: sizeof(struct foo))
```

This is intentional.

Add to the 'semantics' section of §6.5.2.5:

> A compound literal is a block whose scope is a strict subset of the scope of its enclosing block.

Add to the 'semantics' section of §6.5.3.4:

> When either of the `sizeof` and `_Alignof` operators is applied to a parenthesized type, the parentheses surrounding the type delineate a block whose scope is a strict subset of the scope of its enclosing block.

Add to the 'semantics' section of §6.5.4:

> A cast expression is a block whose scope is a strict subset of the scope of its enclosing block.

Note that the scope cannot be limited to the parentheses surrounding the type, otherwise there will be an inconsistency between cast expressions and compound literals:

```
(enum foo { x, y }){x} // if this is a valid expression
(enum foo { x, y })x   // then this should be, too
```

This proposal is orthogonal to the ambiguity of §6.8.5p3. Even if it were adopted, it would still remain ambiguous whether the following snippet were valid:

```
for (struct foo { int m; } x;;);
```

Thus, even if this proposal is adopted, one of the above proposals (1–3) needs to be adopted along with it. Note, though, that proposal 2 becomes much more tenable with this addition, since it is actually able to make interesting guarantees about declaration leakage.