C9X Addition **WG14/N716 X3J11/97-079**

Compound Literals

David Prosser (dfp@sco.com)
David Keaton (dmk@dmk.com)

20 June, 1997

1. Introduction

1.1 Purpose

This document specifies the form and interpretation of a pure extension to the language portion of the C standard to provide important additional flexibility to literals in expressions.

1.2 Scope

This document, although extending the C standard, still falls within the scope of that standard, and thus follows all rules and guidelines of that standard except where explicitly noted herein.

1.3 References

1. ISO/IEC 9899:1990, *Programming Languages — C*.
2. WG14/N494 X3J11/95-095, Prosser & Keaton. C9X Addition, *Designated Initializers*, 8 December, 1995.

All references to the ISO C standard will be presented as subclause numbers. For example, §6.4 references constant expressions.

1.4 Rationale

Compound literals provide a mechanism for specifying constants of aggregate or union type. This eliminates the requirement for temporary variables when an aggregate or union value will only be needed once.

Compound literals integrate easily into the C grammar and do not impose any additional run-time overhead on a user's program. They also combine well with designated initializers (see [2]) to form an even more convenient aggregate or union constant notation. Their initial C implementation appeared in a compiler by Ken Thompson at AT&T Bell Laboratories.

2. Language

2.1 Compound Literals

The syntax for *postfix-expression* in §6.3.2 is augmented by the following:

Syntax

postfix-expression:
    (*type-name* **)** **{** *initializer-list* **}**
    (*type-name* **)** **{** *initializer-list* **,** **}**

A new subclause is added, §6.3.2.5, as follows:

6.3.2.5 Compound literals

Constraints

The type name shall specify an object type or an array of unknown size.

No initializer shall attempt to provide a value for an object not contained within the entire unnamed object specified by the compound literal.[1]

If the compound literal occurs outside the body of a function, the initializer list shall consist of constant expressions.

Semantics

A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose value is given by the initializer list.[2]

If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in §6.5.7, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

The value of the compound literal is that of an unnamed object initialized by the initializer list. The object has static storage duration if and only if the compound literal occurs outside the body of a function; otherwise, it has automatic storage duration associated with the enclosing block.

Except that the initializers need not be constant expressions (when the unnamed object has automatic storage duration), all the semantic rules and constraints for initializer lists in §6.5.7 are applicable to compound literals.[3] The order in which any side effects occur among the initialization list expressions is unspecified.[4]

String literals, and compound literals with const-qualified types, need not designate distinct objects.[5]

Examples

1. The file scope definition

   ```
   int *p = (int []){2, 4};
   ```

   initializes **p** to point to the first element of an array of two **int**s, the first having the value two and the second, four. The expressions in this compound literal must be constant. The unnamed object has static storage duration.

2. In contrast, in

   ```
   void f(void)
   {
           int *p;
           /*...*/
           p = (int [2]){*p};
           /*...*/
   ```

   p is assigned the address of an unnamed automatic storage duration object that is an array of two **int**s, the first having the value previously pointed to by **p** and the second, zero.

1. This is the “There shall be no more initializers...” constraint modified to take into account designated initializers [2].
2. Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or void only, and the result of a cast expression is not an lvalue.
3. For example, subobjects without explicit initializers are initialized to zero.
4. In particular, the evaluation order need not be the same as the order of subobject initialization. The extensions to initializers described in [2] prescribe an ordering for the implicit assignments to the subobjects that comprise the unnamed object.
5. This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

3. Initializers with designations can be readily combined with compound literals. On-the-fly structure objects can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
         (struct point){.x=3, .y=4});
```

Or, if **drawline** instead expected pointers to **struct point**:

```
drawline(&(struct point){.x=1, .y=1},
         &(struct point){.x=3, .y=4});
```

4. A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

5. The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char[]){"/tmp/fileXXXXXX"}
```

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

6. Like string literals, const-qualified compound literals can be be placed into read-only memory and can even be shared. For example,

```
(const char[]){"abc"} == "abc"
```

might yield 1 if the literals' storage is shared.

7. Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object **endless_zeros** below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

8. Outside the body of a function, a compound literal is an initialization of a static object; however, inside a function body, it is an assignment to an automatic object. Therefore, the following two loops produce the same sequence of values for the objects associated with their respective compound literals.

```
for (i = 0; i < 10; i++) {
      f((struct foo){.a = i, .b = 42});
}

for (i = 0; i < 10; i++)
      f((struct foo){.a = i, .b = 42});
```

2.2 Design Discussion

There has been some discussion that perhaps compound literals could be made to have C++ style expression scope, a concept that does not currently exist in C. However, the principle of least surprise dictates that the following should work.

```
void f(void)
{
      int *p = (int []){1, 1, 2, 3, 5, 8, 13};

      for (; *p < 10; p++) {
            /*...*/
      }
      /*...*/
```

Therefore, the scope of a compound literal inside a function body should encompass the enclosing block.