WG14/N302
X3J11/93-049

X3J11.1 Technical Report — Designated Initializers and Compound Literals

X3J11.1/93-024

D. PROSSER

1. Introduction

1.1 Purpose

This document specifies the form and interpretation of a set of pure extensions to the language portion of the C standard, that provide important additional flexibility to initializers in general and 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. ANSI X3.159-1989, *American National Standard for Information Systems — Programming Language — C.*
2. ISO/IEC 9899:1990, *Programming Languages — C.*

All references to sections of the first document and to clauses of the second will be presented in pairs. For example, §3.4 or subclause 6.4 references constant expressions.

2. Language

2.1 Compound Literals

The syntax, constraints, and semantics for *postfix-expression* (§3.3.2 or subclause 6.3.2) is augmented by the following:

Syntax

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

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 postfix expression occurs at file scope, 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 known as a *compound literal*. It provides an unnamed object with value given by the initializer list.[2]

1. This is the "There shall be no more initializers..." constraint modified to take into account designated initializers.
2. Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types only, and the result of a cast expression is not an lvalue.

If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in §3.5.7 or subclause 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 postfix expression occurs at file scope; otherwise, it has automatic storage duration associated with the enclosing function body.

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 (§3.5.7 or subclause 6.5.7) are applicable to compound literals.[3] The order in which any side effects occur within the initialization list expressions is unspecified.[4]

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

Examples

The file scope construction

```
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.

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.

Designated initializers (see below) readily combine 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});
```

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

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

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 later in this document proscribes 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, even if the literals are of different types.

The following three expressions have different meanings:

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

The first is always static and has type `char[]` but may not be assigned to; the last two may be automatic, and their types accurately reflect whether they may be assigned to.

Compound literals with automatic storage duration are associated with the function body that textually contains them. This is not necessarily the immediately containing block. For example, after the following code is executed:

```
struct pt { float x, y, z; } *p[4], *q[4];
int i;
for (i = 0; i < 4; i++) {
      struct pt zero = (struct pt){0, 0, 0};
      p[i] = &zero;
      q[i] = &(struct pt){0, 0, 0};
}
```

`p` contains pointers whose values are indeterminate, whereas `q` contains distinct pointers to newly allocated automatic storage.

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"} == 1+"Xabc"
```

might yield 1 if the literals' storage is shared. Such sharing can occur across types, and across automatic and static storage. For example,

```
(char *)&(const int){0} == ""
```

might yield 1 if the corresponding storage is shared.

A single compound literal cannot specify a circularly linked object. For example, the following code cannot be rewritten to use only a compound literal to initialize the `endless_zeros` object:

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

2.2 Designated Initializers

The syntax for initializers (§3.5.7 or subclause 7.5.7) is changed to the following, and the constraints and semantics are augmented by the following:[6]

Syntax

initializer:
assignment-expression
{ *initializer-list* }
{ *initializer-list* , }

6. The = punctuator that ends the designation is unnecessary, strictly speaking. The redundancy can help an implementation recover from syntactic errors, and also simplifies any future extensions in this area (such as some form of repetition).

initializer-list:
designation$_{opt}$ *initializer*
initializer-list , *designation*$_{opt}$ *initializer*

designation:
designator-list =

designator-list:
designator
designator-list designator

designator:
[*constant-expression*]
. *identifier*

Constraints

No initializer shall attempt to provide a value for an object not contained within the entity being initialized.[7]

If a designator has the form

[*constant-expression*]

then the current object (defined below) shall have array type and the expression shall be an integral constant expression that shall evaluate to a valid index for that array. If the array is of unknown size, any nonnegative index value is valid.

If a designator has the form

. *identifier*

then the current object (defined below) shall have structure or union type and the identifier shall be a member of that type.

Semantics

Each brace-enclosed initializer list has an associated *current object*. When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first member of a union.[8] In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. Initialization continues forward, as normal, after the designated initializer.[9] (The designation can be thought of as a "goto" within the current object.)

Each designator list begins its description with the current object associated with the closest-surrounding brace pair. Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.[10] The current object that results at the end of the designator list is the subobject to be initialized by the following initializer.

---

7. This replaces the current first constraint of "There shall be no more initializers in an initializer list than there are objects to be initialized."

8. If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists.

9. As usual, after a union member is initialized, the next object is not the next member of the union; instead, it is the next subobject of an object containing the union.

10. Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. Note, too, that each separate designator list is independent.

The initialization shall occur in initialization list order, any subsequent value provided for a particular subobject overriding the previous value, where the initial value for all subobjects is zero (converted to the appropriate type).

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer.[11]

Examples

Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```
enum { Mem_One, Mem_Two, /*...*/ };
const char *nm[] = {
      [Mem_Two] = "Mem Two",
      [Mem_One] = "Mem One",
      /*...*/
};
```

Structure members can be initialized to nonzero values without depending on their order:

```
div_t answer = { .quot = 2, .rem = -1 };
```

Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood:

```
struct { int a[3], b; } w[] = { [0].a = {1}, [1].a[0] = 2 };
```

Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

In the above, if `MAX` is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

Finally, any member of a union can be initialized:

```
union { /*...*/ } = { .any_member = 42 };
```

11. This encompasses the current "size determined by the number of initializers" rule.