The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.2.5 Compound literals

C90

Support for compound literals is new in C99.

C++

Compound literals are new in C99 and are not available in C++.

Usage

No usage information is provided on compound literals because very little existing source code contains any use of them.

Constraints

The type name shall specify an object type or an array of unknown size, but not a variable length array type. 1054

Commentary

abstract declarator syntax

The type of the compound literal is deduced from the type name, not from the parenthesized list of expressions. Note that here *type name* refers to a syntactic rule.

The number of elements in the compound literal are required to be known at translation time. This simplifies the handling of their storage requirements.

Compound literals are not usually thought of in terms of scalar types. In those cases where the address of an object is needed simply to fill an argument slot, using an unnamed object may be simpler than defining a dummy variable.

```
extern void glob(int, int, int *);

void f(void)
{
int loc = (int){9}; /* Surprising?  Unusual?  Machine generated? */
double _Complex C = (double _Complex){2.0 + I * 3.0};

glob(1, 2, &((int){0}));
}
```

Other Languages

Some languages do not require a type name to be given. The type of the parenthesized list of expressions is deduced from the context in which it occurs.

Example

```
extern int n;
struct incomplete_type;

void f(void)
{
typedef int a_n[n];
typedef int a_in[];

(a_n){1, 2, 3}; /* Constraint violation. */

/*
 * Number of elements in array deduced from number of
 * expressions in the parentheses; same as for initializers.
 */
(a_in){4, 5, 6, 7};

(void){1.3, 4};                  /* Constraint violation. */
(struct incomplete_type){'a', 3}; /* Constraint violation. */
}
```

1055 No initializer shall attempt to provide a value for an object not contained within the entire unnamed object specified by the compound literal. initializer value not contained in unnamed object

Commentary

initializer value not contained in object

This is the compound literal equivalent of a constraint on initializers for object definitions. The object referred to here is the object being initialized, not other objects declared within the translation unit (or even unnamed object). For instance:

```
void f(int p)
{
struct s_r {
           int mem_1,
               mem_2;
           int *mem_3;
           };

(int []){ p = 1, /* Also provide a value for p. */
              2};

(struct s_r){1, 7,
             &(([2]){5, 6})}; /* Provide a value for an anonymous object. */
}
```

Example

```
struct TAG {
           int mem;
           };

void f(void)
{
/*
 * More expressions that do not have elements in the array.
 */
(int [3]){0, 1, 2, 3}; /* Constraint violation. */
(int [3]){0, [4] = 1}; /* Constraint violation. */
(struct TAG){5, 6};    /* Constraint violation. */
}
```

1056 If the compound literal occurs outside the body of a function, the initializer list shall consist of constant expressions.

Commentary

1065 compound literal outside function body

static storage duration

static storage duration initialized before startup

initializer static storage duration object

All objects defined outside the body of a function have static storage duration. The storage for such objects is initialized before program startup, so can only consist of constant expressions. This constraint only differs from an equivalent one for initializers by being framed in terms of “occurring outside the body of a function” rather than “an object that has static storage duration.”

Semantics

1057 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. compound literal

Commentary

initialization syntax

This defines the term *compound literal*. A compound literal differs from an initializer list in that it can occur outside of an object definition. Because their need be no associated type definition, a type name must be specified (for initializers the type is obtained from the type of the object being initialized).

Other Languages

A form of compound literals are supported in some languages (e.g., Ada, Algol 68, CHILL, and Extended Pascal). These languages do not always require a type name to be given. The type of the parenthesized list of expressions is deduced from the context in which it occurs.

Coding Guidelines

compound literal inside function body 1066
compound literal is lvalue 1061

From the coding guideline point of view, the use of compound literals appears fraught with potential pitfalls, including the use of the term *compound literal* which suggests a literal value, not an unnamed object. However, this construct is new in C99 and there is not yet sufficient experience in their use to know if any specific guideline recommendations might apply to them.

---

compound literal unnamed object

It provides an unnamed object whose value is given by the initializer list.[81)] 1058

Commentary

The difference between this kind of unnamed object and that created by a call to a memory allocation function (e.g., `malloc`) is that its definition includes a type and it has a storage duration other than allocated (i.e., either static or automatic).

Other Languages

Some languages treat their equivalent of compound literals as just that, a literal. For instance, like other literals, it is not possible to take their address.

Common Implementations

In those cases where a translator can deduce that storage need not be allocated for the unnamed object, the as-if rule can be used, and it need not allocate any storage. This situation is likely to occur for compound literals because, unless their address is taken (explicitly using the address-of operator, or in the case of an array type implicit conversion to pointer type), they are only assigned a value at one location in the source code. At their point of definition, and use, a translator can generate machine code that operates on their constituent values directly rather than copying them to an unnamed object and operating on that.

Coding Guidelines

Guideline recommendations applicable to the unnamed object are the same as those that apply to objects having the same storage duration. For instance, the guideline recommendation dealing with assigning the address of objects to pointers.

object address assigned ??

Example

The following example not only requires that storage be allocated for the unnamed object created by the compound literal, but that the value it contains be reset on every iteration of the loop.

```
struct s_r {
           int mem;
           };

extern void glob(struct s_r *);

void f(void)
{
struct s_r *p_s_r;

while (p_s_r->mem != 10)
   {
   glob(p = &((struct s_r){1});
/*
 * Instead of writing the above we could have written:
 *    struct s_r unnamed_s_r = {1};
 *    glob (p_s_r = &unnamed_s_r);
 * which assigns 1 to the member on every iteration, as
 * part of the process of defining the object.
 */
   p_s_r->mem++; /* Increment value held by unnamed object. */
   }
}
```

1059 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.8, and the type of the compound literal is that of the completed array type.

Commentary

This behavior is discussed elsewhere.

array of unknown size initialized

Coding Guidelines

The some of the issues involved in declaring arrays having an unknown size are discussed elsewhere.

array incomplete type

1060 Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name.

Commentary

Presumably this is the declared type of the unnamed object initialized by the initializer list and therefore also its effective type.

effective type

1061 In either case, the result is an lvalue.

compound literal is lvalue

Commentary

While the specification for a compound literal meets the requirements needed to be an lvalue, wording elsewhere might be read to imply that the result is not an lvalue. This specification clarifies the behavior.

lvalue

lvalue converted to value

Other Languages

Some languages consider, their equivalent of, compound literals to be just that, literals. For such languages the result is an rvalue.

rvalue

1062 81) Note that this differs from a cast expression.

footnote 81

Commentary

A cast operator takes a single scalar value (if necessary any lvalue is converted to its value) as its operand and returns a value as its result.

Coding Guidelines

Developers are unlikely to write expressions, such as `(int){1}`, when `(int)1` had been intended (on standard US PC-compatible keyboards the pair of characters *({* and the pair *) }* appear on four different keys). Such usage may occur through the use of parameterized macros. However, at the time of this writing there is insufficient experience with use of this new language construct to know whether any guideline recommendation is worthwhile.

Example

The following all assign a value to `loc`. The first two assignments involve an lvalue to value conversion. In the second two assignments the operand being assigned is already a value.

```
extern int glob = 1;

void f(void)
{
int loc;

loc=glob;
loc=(int){1};

loc=2;
loc=(int)2;
}
```

For example, a cast specifies a conversion to scalar types or **void** only, and the result of a cast expression is not an lvalue. 1063

Commentary

cast scalar or void type footnote 85

These are restrictions on the types and operands of such an expression and one property of its result.

Example

```
&(int)x;   /* Constraint violation. */
&(int){x}; /* Address of an unnamed object containing the current value of x. */
```

The value of the compound literal is that of an unnamed object initialized by the initializer list. 1064

Commentary

The distinction between a compound literal acting as if the initializer list was its value, and an unnamed object (initialized with values from the initializer list) being its value, is only apparent when the address-of operator is applied to it. The creation of an unnamed object does not mean that locally allocated storage is a factor in this distinction. Implementations of languages where compound literals are defined to be literals sometimes use locally allocated temporary storage to hold their values. C implementations may find they can optimize away allocation of any actual unnamed storage.

Common Implementations

If a compound literal occurs in a context where its value is required (e.g., assignment) there are obvious opportunities for implementations to use the values of the initializer list directly. C99 is still too new to know whether most implementations will make use of this optimization.

Coding Guidelines

The distinction between the value of a compound literal being an unnamed object and being the values of the initializer list could be viewed as an unnecessary complication that is not worth educating a developer about. Until more experience has been gained with the kinds of mistakes developers make with compound literals, it is not possible to recommend any guidelines.

Example

```
#include <string.h>

struct TAG {
            int mem_1;
            float mem_2;
            };

struct TAG o_s1 = (struct TAG){1, 2.3};

void f(void)
{
memcpy(&o_s1, &(struct TAG){4, 5.6}, sizeof(struct TAG));
}
```

1065 If the compound literal occurs outside the body of a function, the object has static storage duration;

compound literal outside function body

Commentary

This specification is consistent with how other object declarations, outside of function bodies, behave. The storage duration of a compound literal is based on the context in which it occurs, not whether its initializer list consists of constant expressions.

storage duration object

```
struct s_r {
            int mem;
            };

static struct s_r glob =             {4};
static struct s_r bolg =  (struct s_r){4}; /* Constraint violation. */
static struct s_r *p_g = &(struct s_r){4};

void f(void)
{
static struct s_r  loc =             {4};
static struct s_r  col =  (struct s_r){4}; /* Constraint violation. */
static struct s_r *p_l = &(struct s_r){4}; /* Constraint violation. */
}
```

Other Languages

The storage duration specified by other languages, which support some form of compound literal, varies. Some allow the developer to choose (e.g., Algol 68), others require them to be dynamically allocated (e.g., Ada), while in others (e.g., Fortran and Pascal) the issue is irrelevant because it is not possible to obtain their address.

1066 otherwise, it has automatic storage duration associated with the enclosing block.

compound literal inside function body

Commentary

A parallel can be drawn between an object definition that includes an initializer and a compound literal (that is the definition of an unnamed object). The lifetime of the associated objects starts when the block that contains their definition is entered. However, the objects are not assigned their initial value, if any, until the declaration is encountered during program execution.

object lifetime from entry to exit of block

initialization performed every time declaration reached

The unnamed object associated with a compound literal is initialized each time the statement that contains it is encountered during program execution. Previous invocations, which may have modified the value of the unnamed object, or nested invocations in a recursive call, do not affect the value of the newly created object. Storage for the unnamed object is created on block entry. Executing a statement containing a compound literal does not cause any new storage to be allocated. Recursive calls to a function containing a compound literal will cause different storage to be allocated, for the unnamed object, for each nested call.

object initializer evaluated when

function call recursive

1078 EXAMPLE compound literal single object

```
struct foo {
            struct foo *next;
            int i;
```

```
           };

void WG14_N759(void)
{
struct foo *p,
           *q;
/*
 * The following loop ...
 */
p = NULL;
for (int j = 0; j < 10; j++)
   {
   q = &((struct foo){ .next = p, .i = j });
   p = q;
   }
/*
 * ... is equivalent to the loop below.
 */
p = NULL;
for (int j = 0; j < 10; j++)
   {
   struct foo T;

   T.next = p;
   T.i = j;
   q = &T;
   p = q;
   }
}
```

Common Implementations

To what extent is it worth trying to optimize compound literals made up of a list of constant expressions; for instance, by detecting those that are never modified, or by placing them in a static region of storage that can be copied from or pointed at? The answer to these and many other optimization issues relating to compound literals will have to wait until translator vendors get a feel for how their customers use this new, to C, construct.

Coding Guidelines

Parallels can be drawn between the unnamed object associated with a compound literal and the temporaries created in C++. Experience has shown that C++ developers sometimes assume that the lifetime of a temporary is greater than it is required to be by that languages standard. Based on this experience it is to be expected that developers using C might make similar mistakes with the lifetime of the unnamed object associated with a compound literal. Only time will tell whether these mistakes will be sufficiently common, or serious, that the benefits of being able to apply the address-of operator to a compound literal (the operator that needs to be used to extend the range of statements over which an unnamed object can be accessed) are outweighed by the probably cost of faults.

object ?? address assigned

The guideline recommendation dealing with assigning the address of an object to a pointer object, whose lifetime is greater than that of the addressed object, is applicable here.

```
#include <stdlib.h>

extern int glob;
struct s_r {
           int mem;
           };

void f(void)
{
```

```
struct s_r *p_s_r;

if (glob == 0)
   {
   p_s_r = &((struct s_r){1});
   }
else
   {
   p_s_r = &((struct s_r){2});
   }
/* The value of p_s_r is indeterminate here. */

/*
 * The iteration-statements all enclose their associated bodies in
 * a block.  The effect of this block is to start and terminate
 * the lifetime of the contained compound literal.
 */
p_s_r=NULL;
while (glob < 10)
   {
   /*
    * In the following test the value of p_s_r is indeterminate
    * on the second and subsequent iterations of the loop.
    */
   if (p_s_r == NULL)
      ;
   p_s_r = &((struct s_r){1});
   }
}
```

1067 All the semantic rules and constraints for initializer lists in 6.7.8 are applicable to compound literals.[82)]

Commentary

They are the same except

- initializer lists don't create objects, they are simply a list of values with which to initialize an object; and
- the type is deduced from the object being initialized, not a type name.

Coding Guidelines

Many of the coding guideline issues discussed for initializers also apply to compound literals.

initialization syntax

1068 String literals, and compound literals with const-qualified types, need not designate distinct objects.[83)]

string literal distinct object compound literal distinct object

Commentary

A strictly conforming program can deduce if an implementation uses the same object for two string literals, or compound literals, by performing an equality comparison on their addresses (an infinite number of comparisons would be needed to deduce whether an implementation always used distinct objects). This permission for string literals is also specified elsewhere.

1076 EXAMPLE string literals shared

string literal distinct array

The only way a const-qualified object can be modified is by casting a pointer to it to a non-const-qualified pointer. Such usage results in undefined behavior. The undefined behavior, if the pointer was used to modify such an unnamed object that was not distinct, could also modify the values of other compound literal object values.

pointer converting qualified/unqualified

Other Languages

Most languages do not consider any kind of literal to be modifiable, so whether they share the same storage locations is not an issue.

Common Implementations

The extent to which developers will use compound literals having a const-qualified type, for which storage is allocated and whose values form a sharable subset with another compound literal, remains to be seen. Without such usage it is unlikely that implementors of optimizers will specifically look for savings in this area, although they may come about as a consequence of optimizations not specifically aimed at compound literals.

Example

In the following there is an opportunity to overlay the two unnamed objects containing zero values.

```
const int *p1 = (const int [99]){0};
const int *p2 = (const int [20]){0};
```

EXAMPLE 1 The file scope definition 1069

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

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

Commentary

This usage, rather than the more obvious `int p[] = {2, 4};`, can arise because the initialization value is derived through macro replacement. The same macro replacement is used in noninitialization contexts.

EXAMPLE 2 In contrast, in 1070

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

p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by **p** and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

Commentary

The assignment of values to the unnamed object occurs before the value of the right operand is assigned to `p`.

Example

The above example is not the same as declaring `p` to be an array.

```
void f(void)
{
int p[2]; /* Storage for p is created by its definition. */

/*
 * Cannot assign new object to p, can only change existing values.
 */
p[1]=0;
}
```

1071 EXAMPLE 3 Initializers with designations can be combined with compound literals. Structure objects created using compound literals 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});
```

Commentary

This usage removes the need to create a temporary in the calling function. The arguments are passed by value, like any other structure argument.

1072 EXAMPLE 4 A read-only compound literal can be specified through constructions like:

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

Commentary

An implementation may choose to place the contents of this compound literal in read-only memory, but it is not required to do so. The term *read-only* is something of a misnomer, since it is possible to cast its address to a non-const-qualified type and assign to the pointed-to object. (The behavior is undefined, but unless the values are held in a kind of storage that cannot be modified, they are likely to be modified.)

Other Languages

Some languages support a proper read-only qualifier.

Common Implementations

On some freestanding implementations this compound literal might be held in ROM.

1073 82) For example, subobjects without explicit initializers are initialized to zero. footnote 82

Commentary

This behavior reduces the volume of the visible source code when the object type includes large numbers of members or elements.

initializer fewer in list than members

Coding Guidelines

Some of the readability issues applicable to statements have different priorities than those for declarations. These are discussed elsewhere.

initialization syntax

1074 83) This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations. footnote 83

Commentary

The need to discuss an implementation's ability to share storage for string literals occurs because it is possible to detect such sharing in a conforming program (e.g., by comparing two pointers assigned the addresses of two distinct, in the visible source code, string literals). The C Committee choose to permit this implementation behavior. (There were existing implementations, when the C90 Standard was being drafted, that shared storage.)

EXAMPLE 5 The following three expressions have different meanings: 1075

```
"/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.

Commentary

EXAMPLE 1076 string literals shared

In all three cases, a pointer to the start of storage is returned and the first 16 bytes of the storage allocated will have the same set of values. If all three expressions occurred in the same source file, the first and third could share the same storage even though their storage durations were different. Developers who see a potential storage saving in using a compound literal instead of a string literal (the storage for one only need be allocated during the lifetime of its enclosing block) also need to consider potential differences in the number of machine code instructions that will be generated. Overall, there may be no savings.

EXAMPLE string literals shared

EXAMPLE 6 Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example, 1076

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

might yield 1 if the literals' storage is shared.

Commentary

compound literal 1068 distinct object
string literal distinct array

In this example pointers to the first element of the compound literal and a string literal are being compared for equality. Permission to share the storage allocated for a compound literal only applies to those having a const-qualified type (there is no such restriction on string literals).

Coding Guidelines

Comparing string using an equality operator, rather than a call to the `strcmp` library function is a common beginner mistake. Training is the obvious solution.

Usage

In the visible source of the `.c` files 0.1% of string literals appeared as the operand of the equality operator (representing 0.3% of the occurrences of this operator).

EXAMPLE 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: 1077

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

Commentary

A modification using pointer types, and an additional assignment, creates a circularly linked list that uses the storage of the unnamed object:

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

endless_zeros->cdr=endless_zeros; /* Let's follow ourselves. */
```

The following statement would not have achieved the same result:

```
endless_zeros = &(struct int_list){0, endless_zeros};
```

because the second compound literal would occupy a distinct object, different from the first. The value of `endless_zeros` in the second compound literal would be pointing at the unnamed object allocated for the first compound literal.

Other Languages

Algol 68 supports the creation of circularly linked objects (see the Other Languages subsection in the following C sentence).

1078 EXAMPLE 8 Each compound literal creates only a single object in a given scope:

EXAMPLE compound literal single object

```
struct s { int i; };

int f (void)
{
        struct s *p = 0, *q;
        int j = 0;

again:
        q = p, p = &((struct s){ j++ });
        if (j < 2) goto again;

        return p == q && q->i == 1;
}
```

The function **`f()`** always returns the value 1.
Note that if an iteration statement were used instead of an explicit **`goto`** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around **`p`** would have an indeterminate value, which would result in undefined behavior.

Commentary

Specifying that a single object is created helps prevent innocent-looking code consuming large amounts of storage (e.g., use of a compound literal in a loop).

Other Languages

In Algol 68 **`LOC`** creates storage for block scope objects. However, it generates new storage every time it is executed. The following allocates 1,000 objects on the stack.

```
MODE M = STRUCT (REF M next, INT i);
M p;
INT i := 0

again:
   p := LOC M := (p, i);
   i +:= 1;
   IF i < 1000 THEN
      GO TO again
   FI;
```

1079 **Forward references:** type names (6.7.6), initialization (6.7.8).

References