The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.8 Initialization

initialization syntax

1641

```
initializer:
                assignment-expression
                { initializer-list }
                { initializer-list , }
initializer-list:
                designation_opt initializer
                initializer-list , designation_opt initializer
designation:
                designator-list =
designator-list:
                designator
                designator-list designator
designator:
                [ constant-expression ]
                . identifier
```

Commentary

The term *designated initializer* is sometimes used in discussions by members of the committee. However, the C Standard does not use or define this term, it uses the term *designation initializer*.

Rationale

A new feature of C99: Designated initializers provide a mechanism for initializing sparse arrays, a practice common in numerical programming. They add useful functionality that already exists in Fortran so that programmers migrating to C need not suffer the loss of a program-text-saving notational feature.

This feature also allows initialization of sparse structures, common in systems programming, and allows initialization of unions via any member, regardless of whether or not it is the first member.

C90

Support for designators in initializers is new in C99.

C++

Support for designators in initializers is new in C99 and is not specified in the C++ Standard.

Other Languages

Ada supports the initialization of multiple objects with a single value.

```
Total_val,
Total_average : INTEGER : = 0;
```

Ada (and Extended Pascal) does not require the name of the member to be prefixed with the **.** token and uses the token **=>** (Extended Pascal uses **:**), rather than **=**.

Extended Pascal supports the specification of default initial values in the definition of a type (that are then applied to objects defined to have that type).

The Fortran **DATA** statement is executed once, prior to program startup, and allows multiple objects to be initialized (sufficient values are used to initialize each object and it is the authors responsibility for ensuring that each value is used to initialize the intended object):

```
      CHARACTER*5 NAME
      INTEGER I, J, K
      DATA NAME, I, J, K / 'DEREK', 1, 2, 3/
```

Ada, Extended Pascal, and Fortran (since the 1966 standard) all provide a method of specifying a range of array elements that are to be initialized with some value.

Algol 68 uses the token `=` to specify that the declared identifier is a constant, and `:=` to specify that declared identifier is a variable with an initial value. For instance:

```
INT c = 5;       # constant #
INT d = c + 1;   # constant #
INT e := d;      # initialized variable #
```

BCPL supports parallel declarations, for instance:

```
LET a, b, c = x, y, z; // declare a, b, and c, and then in some order assign x to a, y to b, and z to c
```

Common Implementations

Some prestandard implementations (e.g., `pcc`) parsed initializers bottom-up, instead of top-down (as required by the standard). A few modern implementations provide an option to specify a bottom-up parse (e.g., the Diab Data C compiler[1] supports the `-Xbottom-up-init` option). For instance, in:

```
struct T { int a, b; };
struct  {
         struct T c[2];
         struct T d[2];
        } x = {
                {1, 2},
                {3, 4}
               };
```

the initialization of `x` is equivalent to:

```
{ { {1, 2}, {0, 0} },
  { {3, 4}, {0, 0} } };
```

An implementation performing a bottom-up parse would treat it as being equivalent to:

```
{ { {1, 2}, {3, 4} },
  { {0, 0}, {0, 0} } };
```

`gcc` supports the use of a range notation in array initializer designators. For instance:

```
int my_array[100] = { [0 ... 99] = 1 };
int my_parts[PART_1 + PART_2] = { [0 ... PART_1-1] = 1,
                                  [PART_1 ... PART_1+PART_2-1] = 2 };
```

Coding Guidelines

Constant values often occur in initializer lists. Such occurrences may be the only instance of the constant value in the source, or the values appearing in the list may have semantic or mathematical associations with each other. In these cases the guideline recommendation that names be given to constants may not have a worthwhile benefit.

Dev ?? An integer constant may appear in the list of initializers for an object having an array or structure type.

Dev ?? A floating constant may appear in the list of initializers for an object.

statement visual layout

declaration visual layout

What are the costs and benefits of organizing the visible appearance of initializers in different ways (like other layout issues, experience suggests that developers have an established repertoire of layout rules which provide the template for laying out individual initializers, e.g., a type based layout with array elements in columns and nested structure types indented like nested conditionals)?

- *Cost*— the time taken to create the visual layout and to maintain it when new initializers are added, or existing ones removed. Experience suggests that developers tend to read aggregate initializers once (to comprehend what their initial value denotes) and ignore them thereafter. Given this usage pattern, initializers are likely to be a visual distraction most of the time (another cost).
- *Benefit*— the visual layout may reduce the effort needed by readers to comprehend them.

Until more is known about the frequency with which individual initializers are read for comprehension, as opposed to being given a cursory glance (almost treated as distractions) it is not possible to reliably provide cost effective recommendations about how to organize their layout. The following discusses some possibilities.

gestalt principles

The gestalt principle of organization suggest that related initializers be visually grouped together. It is not always obvious what information needs to be visually grouped. For instance, for an array of structure type initializer might be grouped by array element or by structure member, or perhaps an application oriented grouping:

```
/* Low visual distraction to rest of source. */
{ {1, 'w'}, {2, 'q'}, {3, 'r'} {4, 'a'} }

/* Grouped by array element. */
{
 {1, 'w'},
 {2, 'q'},
 {3, 'r'},
 {4, 'a'}
}

/* Application may suggest an odd/even array element grouping. */
{
 {1, 'w'},
             {2, 'q'},
 {3, 'r'},
             {4, 'a'}
}

/* Perhaps initializers should be grouped by member. */
{
 {1,     'w'},
 {2,     'q'},
 {3,     'r'},
 {4,     'a'}
}
```

Other considerations on initializer layout include making it easy to check that all required initializers are present and visually exposing patterns in the constants appearing in the initializer (with the intent of reducing the effort needed, by subsequent, to deduce them):

```
int bit_pattern[] = {
                      0001, /* octal constant */
                      0010,
                      0100,
                      1000, /* Context may cause reader treat this as an octal constant. */
                     };
```

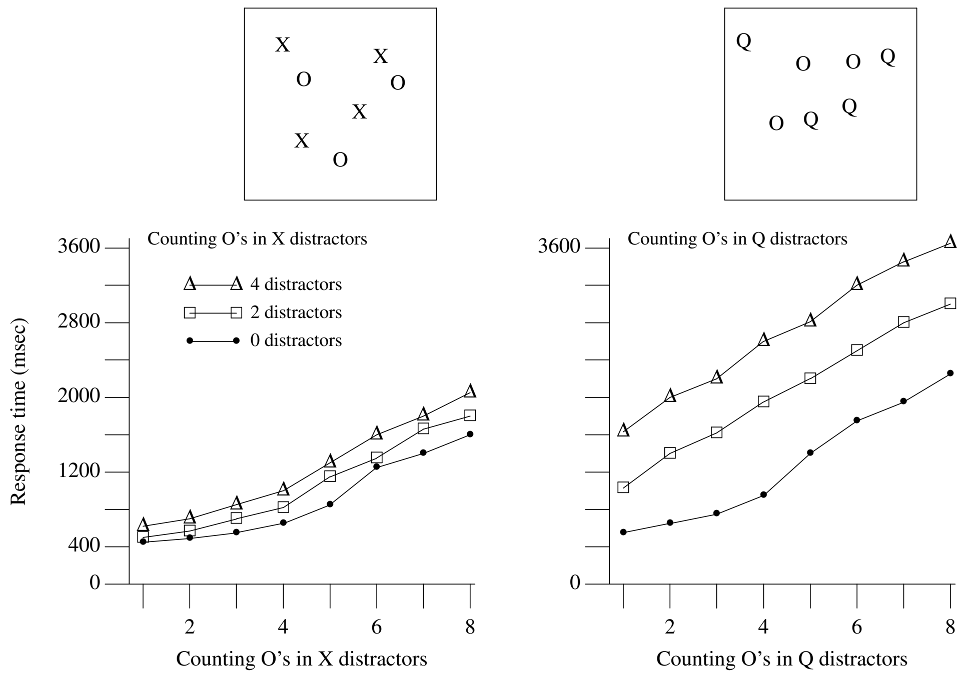

Figure 1641.1: Average time (in milliseconds) taken for subjects to enumerate O's in a background of X or Q distractors. Based on Trick and Pylyshyn.[5]

Other cognitive factors include subitizing and the Stroop effect.

subitizing

When people are asked to enumerate how many dots, for instance, are visible in a well defined area their response time depends on the number of dots. However, when there are between one and four dots performance varies between 40 ms to 100 ms per dot. With five or more dots performance varies between 250 ms to 350 ms per dot. The faster process used when there are four or fewer dots is called *subitizing* (people effortlessly *see* the number of dots), while the slower process is called *counting*.

vision
preattentive

Subitizing has been shown[5] to rely on information available during the preattentive stage of vision. Items that rely on later stages of visual processing (e.g., those requiring spatial attention, such as enumerating the number of squares along a given line) cannot be subitized, they have to be counted. The limit on the maximum number of items that can be subitized is thought to be caused by capacity limits in the preattentive stages of vision.[6] The extent to which other items, distractors, visible on the display reduce enumeration performance depends on the number of distractors and whether it is possible to discriminate between the visible items during the visual systems preattentive stage. For instance, it is possible to subitize the letter *O* when the distractors are the letter *X*, but not when the distractors are the letter *Q* (see Figure 1641.1).

stroop effect

A study by Stroop[4] asked subjects to name the color of ink that words were written in. For instance, the word *red* was printed in black, blue, and green inks and the word *blue* was printed in black, red, and green inks. The results showed that performance (response time and error rate) suffered substantial interference from the tendency to name the word, rather than its color.

The explanation for what has become known as the *Stroop* effect is based on interference, in the human brain, between the two tasks of reading a word and naming a colour (which are performed in parallel; in general words are read faster, an automatic process in literate adults, than colors can be named). Whether the interference occurs in the output unit, which is serial (one word arriving just before the other and people only being able to say one word at a time), or occurs between the units doing the naming and reading, is still an open question.

Experiments using a number of different kinds of words and form of visual presentation had replicated the effect. For instance, the Stroop effect has been obtained using lists of numbers. Readers might like to try counting the number of characters occurring in each separate row appearing in the margin.

3 3 3 3
a a a a a a
8 8 8
z z
1 1
t t t t
6 6 6 6 6

The effort of counting the digit sequences is likely to have been greater and more error prone than for the letter sequences.

Studies[3] have found that when subjects are asked to enumerate visually presented digits, the amount of Stroop-like interference depends on the arithmetic difference between the magnitude of the digits used and the number of those digits displayed. Thus a short, for instance, list of large numbers is read more quickly and with fewer errors than a short list of small numbers. Alternatively a long list of small numbers (much smaller than the length of the list) is read more quickly and with fewer errors than a long list of numbers where the number has a similar magnitude to the length of the list.

Initializers often contain lists of similar numbers. The extent to which initializer layout interacts with readers using subitizing/counting and the Stroop effect is not known.

Example

If the `wchar_t` type is different from type **char**, then:

```
#include <stddef.h>

char str1[] = L"abc";               /* Constraint violation, if (wchar_t != char) */
char str2[] = {L'a', L'b', L'c'};   /* OK    */

wchar_t wstr1[] = "abc";            /* Constraint violation, if (wchar_t != char) */
wchar_t wstr2[] = {'a', 'b', 'c'};  /* OK    */
```

Table 1641.1: Occurrence of object types, in block scope, whose declaration includes an initializer (as a percentage of the type of all such declarations with initializers). Based on the translated form of this book's benchmark programs. Usage information on the types of all objects declared at file scope is given elsewhere (see Table **??**).

Type	%	Type	%
struct *	39.5	**long**	2.6
int	22.6	**char**	2.5
other-types	9.1	**unsigned short**	2.4
unsigned int	4.5	**unsigned char**	1.5
union *	4.3	**unsigned char** *	1.4
char *	4.0	**unsigned int** *	1.2
unsigned long	3.4	**enum**	1.1

Table 1641.2: Occurrence of object types with internal linkage, at file scope, whose declaration includes an initializer (as a percentage of the type of all such declarations with initializers). Based on the translated form of this book's benchmark programs. Usage information on the types of all objects declared at file scope is given elsewhere (see Table **??**).

Type	%	Type	%
const char []	22.5	**char** *	2.2
const struct	14.7	**int** []	2.1
int	11.1	**char** []	2.0
struct	10.4	**unsigned char** []	1.7
other-types	10.4	**void** *()	1.3
struct []	8.3	**(char *)** []	1.3
struct *	2.9	**int** *()	1.2
(const char * const) []	2.9	**const unsigned char** []	1.2
unsigned short []	2.5	**const short** []	1.2

Constraints

1642 No initializer shall attempt to provide a value for an object not contained within the entity being initialized.

initializer value not contained in object

Commentary

This constraint can apply to any initializer where a designator is not used and more values are given than are available to be initialized in the type. It can also apply to an array being initialized with a designator that specifies an element not contained within the array type (structure member designators are covered by another constraint).

1648 designator . identifier

C90

There shall be no more initializers in an initializer list than there are objects to be initialized.

Support for designators in initializers is new in C99 and a generalization of the wording is necessary to cover the case of a name being used that is not a member of the structure or union type, or an array index that does not lie within the bounds of the object array type.

C++

The C++ Standard wording has the same form as C90, because it does not support designators in initializers.

8.5.1p6

An `initializer-list` is ill-formed if the number of initializers exceeds the number of members or elements to initialize.

1643 The type of the entity to be initialized shall be an array of unknown size or an object type that is not a variable length array type.

Commentary

An array of unknown size is an incomplete type and therefore not an object type.

array unknown size
object types

There is no mechanism for specifying a repeat factor for initializers in C, the number of elements is known at translation time. By the nature of their intended usage the number of elements in an object having a variable length array type is not known at translation time and is likely to vary between different instances of type instantiation during program executions. Providing support for initializations would involve specifying many different combinations of events, a degree of complexity that is probably not worth the cost. It is not possible to specify a single initial value, in the declaration of an object having a variable length array type, as a means of implicitly causing all other elements to be initialized to zero.

1682 initializer fewer in list than members

C90

Support for variable length array types is new in C99.

1644 All the expressions in an initializer for an object that has static storage duration shall be constant expressions or string literals.

initializer static storage duration object

Commentary

This requirement ensures that the initial value of objects is known at translation time. This simplifies program startup and avoids the complications involved in deducing dependencies between initialization values (needed to define an order of static initialization, on program startup, that gives consistent behavior).

C90

All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.

C99 has relaxed the requirement that aggregate or union types always be initialized with constant expressions.

Support for string literals in this context was added by the response to DR #150.

C++

8.5p2 *Automatic, register, static, and external variables of namespace scope can be initialized by arbitrary expressions involving literals and previously declared variables and functions.*

A program, written using only C constructs, could be acceptable to a conforming C++ implementation, but not be acceptable to a C implementation.

C++ translators that have an *operate in C mode* option have been known to fail to issue a diagnostic for initializers that would not be acceptable to conforming C translators.

Other Languages

Not all languages require the values of initializers to be known at translation time.

---

identifier linkage at block scope

If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier. 1645

Commentary

Existing code, prior to C90, contained declarations of identifiers with external linkage (but without initializers) in block scope and the C committee sanctioned its continued use. Providing an initializer for an object having external or internal linkage, is one method of specifying that it denotes the definition of that object. However, support for such usage has no obvious benefit in block scope.

object reserve storage

Block scope definitions that include the **static** storage-class specifier have no linkage and may contain an initializer.

no linkage block scope object

C++

The C++ Standard does not specify any equivalent constraint.

Coding Guidelines

If the guideline recommendation dealing with declaring identifiers with external or internal linkage at file scope is followed this situation can never occur.

identifier declared in one file ??

---

designator constant-expression

If a designator has the form 1646

```
[ constant-expression ]
```

then the current object (defined below) shall have array type and the expression shall be an integer constant expression.

Commentary

While it is possible for the initialization value to be a nonconstant, the designator of the element being initialized must be constant. Requiring support for nonconstant designators would have introduced a number of complexities. For instance, in the following example:

```
int n;
/* ... */
int vec_1[]   = { [n] = 2 };
int vec_2[10] = { [n] = 2 };
```

the declaration of vec_1 is essentially a new way of specifying a VLA, while in the initializer for vec_2 a translator cannot enforce the constraint that initializers only provide values for objects contained within the entity being initialized until program execution.

initializer value not contained in object 1642

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

Some of the coding guideline issues involved in using designators are discussed elsewhere.

1670 initialization using a designator

1647 If the array is of unknown size, any nonnegative value is valid.

Commentary

The number of elements in an array declarator that does not specify a size is deduced from its initializer. Hence, any nonnegative value is guaranteed to be permitted by the standard. However, an implementation may fail to translate a source file containing an object whose size exceeds the minimum required limits.

1683 array of unknown size initialized

limit minimum object size

Coding Guidelines

There is not sufficient experience available with the use of designators to know if any particular nonnegative value should be considered suspicious (e.g., very large values, or gaps in a consecutive range).

1648 If a designator has the form

designator . identifier

```
        . identifier
```

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

Commentary

This requirement mimics that specified for the `.` operator.

operator . first operand shall

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

The issue of two or more designators specifying a value for the same member of a subobject is discussed elsewhere.

1676 initialization in list order

Semantics

1649 An initializer specifies the initial value stored in an object.

initializer initial value

Commentary

Here the term *initial* refers to the behavior from an executing programs perspective. When initialization occurs depends on the storage duration of the object being initialized. Objects with static storage duration are initialized on program startup, while objects with automatic storage duration are initialized when the objects declaration is encountered during program execution (although their lifetime starts when the block that contains the declaration is entered), and objects with allocated storage duration can be given an initial value to zero by calling the `calloc` library function.

static storage duration initialized before startup

object initializer evaluated when

lifetime of object

Common Implementations

The machine code generated to initialize scalar objects with automatic storage duration is usually the same as that used to assign a value in an expression statement.

It is much easier to generate efficient machine code for aggregate objects, with automatic storage duration, when an initializer is used compared to when the developer has used a sequence of assignment statements (the translator does not need to perform any analysis to deduce that the values being assigned are all associated with the same object). Using compound literals is likely to result in translators allocating additional storage for the unnamed object (unless it can be deduced that such storage is unnecessary).

compound literal
unnamed object

Coding Guidelines

Some coding guideline documents recommend that an initializer always be used to store an initial value in an object. The rationale being that providing an initial value in the definition of the object guarantees that the object is always initialized before use (the concern seems to be more oriented towards ensuring an object does not have an indeterminate value than that it have the correct value, which it might not be possible to assign at the point of definition).

One potential advantage, in providing an initializer, is that if the definition is moved from an inner to an outer scope (moving from an outer to an inner scope will cause a diagnostic to be issued for any references from outside the new scope), during program modification, its initialization is moved at the same time (if the initial value is assigned in a statement, then that statement also needs to be moved; forgetting to do this is a potential source of faults). The following lists several potential disadvantages to this usage:

- The values of the operands in the expression providing the initial value may not be correct, at the point in the source where the initializer occurs. It is also possible that the object represents some temporary value that depends on the value of other objects defined in the same scope.
- A strong case can be made for initializing objects close to their point of use, so the associated source forms a readily comprehensible grouping (this is also an argument for moving the definition closer to its point of use). This issue is discussed in more detail elsewhere.

statement
syntax

- Use of an initializer has been found to sometimes give developers a false sense of security, causing the assignment of a value to be overlooked. For instance, when two very similar sequences of source code occur in a function the second is often created by modifying a copy of the first one. A commonly seen mistake, when initializers are used, is to forget to give some object a new initial value.

```
struct T;
extern struct T *p_list;

void f(void)
{
struct T *walk_p_list = p_list;

while (walk_p_list)
   {
/* Walk p_list doing something. */
   }

/*
 * Copied above loop, but overlooked giving walk_p_list an initial value.
 */
while (walk_p_list)
   {
/* Walk p_list doing something similar (but loops cannot be merged). */
   }
}
```

In C99 it is possible to intermix declarations and statements. This means that the point of declaration, of an object, could be moved closer to where it is first used. The issue of where to declare objects is discussed elsewhere.

identifier
definition
close to usage

There is no evidence to suggest that the benefits of unconditionally providing an initializer in the definition of objects are greater than the costs. For this reason no guideline recommendation is given.

1650 Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization.

unnamed members initialization of

Commentary

This is explicitly stated otherwise in one other instance and is discussed there. The fact that a member is unnamed suggests the author did not intend any value it might happen to contain to be accessed. Providing a mechanism to specify their initial value has no obvious benefit.

1663 initializing including unnamed members

C90

The C90 Standard wording "All unnamed structure or union members are ignored during initialization." was modified by the response to DR #017q17.

C++

This requirement is not explicitly specified in the C++ Standard.

1651 Unnamed members of structure objects have indeterminate value even after initialization.

Commentary

Many developers are aware of the common implementation practice, if a sufficient number of members are initialized to zero, of zeroing all of the storage occupied by an object having a structure type. This C sentence points out that as far as the abstract machine is concerned unnamed members are not affected by the initializer in a definition.

abstract machine C

C90

This was behavior was not explicitly specified in the C90 Standard.

C++

This behavior is not explicitly specified in the C++ Standard.

Common Implementations

For objects having an aggregate type and where many members are initialized to zero, many implementations generate machine code to loop over the entire object, setting all bytes to zero; the nonzero values are then individually assigned.

1652 If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

object value indeterminate

Commentary

This is a conceptual indeterminate value, an implementation is not required to create and assign it (when the declaration of such an object is encountered during program execution).

indeterminate value

```
extern _Bool g(void);

void f (void)
{
loop: ;
int i = 42; /* Explicit initialization occurs every time declaration is encountered */
int j;      /* as does implicit initialization to indeterminate value. */
if (g())    /* j always has an indeterminate value here. */
   return;
i = 69;     /* Change value of i.   */
j = 0;      /* Assign a value to j. */
goto loop;
}
```

The behavior resulting from reading the value of an object having an indeterminate value depends on the type of the object. Objects having type **unsigned char** are guaranteed to contain a representable value (or an

unsigned char pure binary

trap representation

object having type array of **`unsigned char`**). In this case a read access results in unspecified behavior. The indeterminate value of objects having other types may be a trap representation (accessing an object having such a value results in undefined behavior).

Other Languages

Some languages (e.g., Perl) implicitly assign a value to all objects before they are explicitly assigned to (in the case of Perl this is the value `undef`, which evaluates to either 0 or the zero length string). Some languages (e.g., Ada and Extended Pascal) allow the definition of a type to include an initial value that is implicitly assigned to objects defined to have that type.

Common Implementations

The value of such objects is usually the bit pattern that happened to exist in the storage allocated at the start its lifetime. This bit pattern may have been created by assigning a value to an object whose lifetime has ended, or the storage may have been occupied by more than one object, or it may have been used to hold housekeeping information. A few implementations zero the storage area, reserved for defined objects, on function entry.

Example

```
{ float f=1.234; }
{ int i;  /* i will probably be allocated storage previous occupied by f */ }
```

static initialization default value

If an object that has static storage duration is not initialized explicitly, then: 1653

Commentary

In the past many implementations implicitly initialized the storage occupied by objects having static storage duration to all bits zero prior to program startup. This practice is codified in the C Standard here; where the intent of *all bits zero*, i.e., a value representation of zero, is specified. The issue of initializing objects with static storage duration is discussed elsewhere.

static storage duration initialized before startup
static storage duration when initialized

C++

The C++ Standard specifies a two-stage initialization model. The final result is the same as that specified for C.

8.5p6

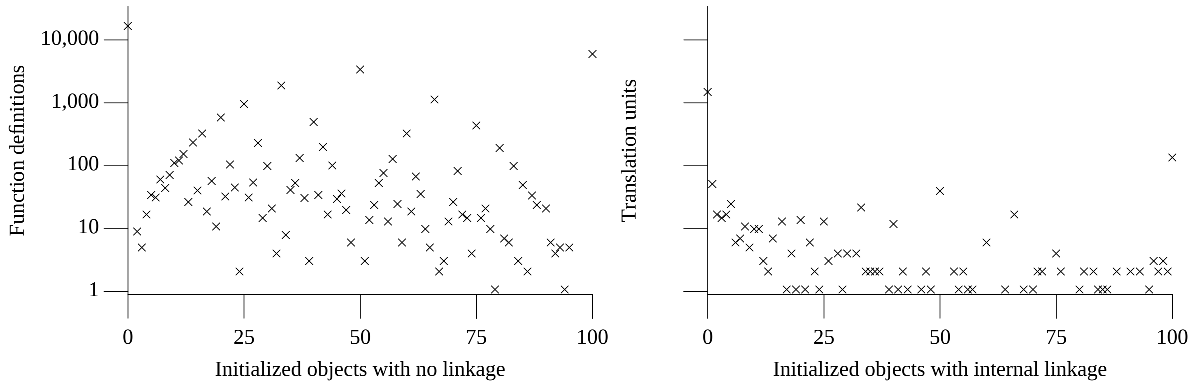

Figure 1652.1: Number of object declarations that include an initializer (as a percentage of all corresponding object declarations), either within function definitions (functions that did not contain any object definitions were not included), or within translation units and having internal linkage (while there are a number of ways of counting objects with external linkage, none seemed appropriate and no usage information is given here). Based on the translated form of this book's benchmark programs.

The memory occupied by any object of static storage duration shall be zero-initialized at program startup before any other initialization takes place. [Note: in some cases, additional initialization is done later.]

Other Languages

Most other languages do not treat one kind of uninitialized objects any differently than another kind.

Coding Guidelines

The behavior described in the following sentences is common developer knowledge. There is no obvious benefit in recommending against making use of it, on the basis that all behavior should be explicit.

1654 — if it has pointer type, it is initialized to a null pointer;

Commentary

The null pointer is also the only pointer value that is compatible with all pointer types.

null pointer
null pointer conversion yields null pointer

1655 — if it has arithmetic type, it is initialized to (positive or unsigned) zero;

Commentary

Zero is the constant most frequently assigned to an object and the most commonly occurring constant literal in source code. It is the initial value most likely to be chosen by a developer, if one had to be explicitly supplied.

integer constant usage

C90

The distinction between the signedness of zero is not mentioned in the C90 Standard.

Common Implementations

In most implementations this value is represented by all bits zero for all arithmetic types.

1656 — if it is an aggregate, every member is initialized (recursively) according to these rules;

member initialized recursively

Commentary

In the case of array types every element is assigned the same value.

Common Implementations

Most implementations treat an aggregate, that is being implicitly initialized, as a single entity. That is the most efficient way of assigning all bits zero is used.

1657 — if it is a union, the first named member is initialized (recursively) according to these rules.

Commentary

Having decided to support the initialization of objects having a union type, the specification either had to provide a mechanism for denoting a member to be initialized, or provide a rule to enable readers to deduce the member that will be initialized. The rule that the first named member is initialized fit in with the general English (and perhaps other cultures) convention of starting at the top and working down (rather than starting at the bottom and working up). It is also less likely to cause maintenance problems (since developers tend to add new members at the end of the list of current members).

If the first named member is an aggregate all the members of that aggregate are initialized.

C90

This case was not called out in the C90 Standard, but was added by the response in DR #016.

C++

The C++ Standard, 8.5p5, specifies the first data member, not the first named data member.

initializer scalar

The initializer for a scalar shall be a single expression, optionally enclosed in braces. 1658

Commentary

Allowing initializers for scalars to be enclosed in braces can simplify the automatic generation of C source (the generator does not need any knowledge of the type being initialized, it can always output braces). Braces are not optional when a value appears as the right operand of an assignment operator.

C++

8.5p13 *If T is a scalar type, then a declaration of the form*

```
T x = { a };
```

is equivalent to

```
T x = a;
```

This C++ specification is not the same the one in C, as can be seen in:

```
struct DR_155 {
                int i;
               } s = { { 1 } }; /* does not affect the conformance status of the program */
                                // ill-formed
```

8.5p14 *If the conversion cannot be done, the initialization is ill-formed.*

While a C++ translator is required to issue a diagnostic for a use of this ill-formed construct, such an occurrence causes undefined behavior in C (the behavior of many C translators is to issue a diagnostic).

Other Languages

Most languages that support initializers do not allow redundant braces to be used for scalars.

Coding Guidelines

Initializers for objects having scalar type are rarely enclosed in braces. For this reason they are not discussed further here.

The initial value of the object is that of the expression (after conversion); 1659

Commentary

The conversion is to the type of the object being initialized.

C90

The C90 wording did not include "(after conversion)". Although all known translators treated initializers just like assignment and performed the conversion.

initializer type constraints

the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type. 1660

Commentary

simple assignment constraints

simple assignment

The unqualified type needs to be specified because the constraints for simple assignment do not permit object having a const-qualified type to be assigned a value. A consequence of taking the unqualified type is that initialization does not have exactly the same semantics as simple assignment if the object has a volatile-qualified type.

C++

Initialization is not the same as simple assignment in C++ (5.17p5, 8.5p14). However, if only the constructs available in C are used the behavior is the same.

Coding Guidelines

The applicable guideline recommendation are those that apply to simple assignment.

simple assignment constraints

1661 The rest of this subclause deals with initializers for objects that have aggregate or union type.

Commentary

The standard defines additional syntax and semantics for initializers of objects having aggregate or union type.

1662 The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type.

Commentary

Just like simple assignment, it is possible to initialize a structure or union object with the value of another object having the same type. In the case of a union object the value assigned need not be that of the first named member.

C++

The C++ Standard permits an arbitrary expression to be used in all contexts (8.5p2).

This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation. C++ translators that have an *operate in C mode* switch do not always diagnose initializers that would not be acceptable to all conforming C translators.

initializing including unnamed members

1663 In the latter case, the initial value of the object, including unnamed members, is that of the expression.

Commentary

The former case is the subject of discussion of most of the rest of the C subclause. There is one case where unnamed members participate in initialization.

1650 unnamed members initialization of

```
#include <stdio.h>

struct T_1 {
            int mem_1;
            unsigned : 4;
            double mem_2;
           };
struct T_2 {
            int mem_1;
            unsigned int mem_name: 4;
            double mem_2;
           };
union T_3 {
           struct T_1 su;
           struct T_2 sn; /* Both structure types have a common initial sequence. */
          } gu;
struct T_1 s;

int main(void)
{
union T_3 lu_1 = {{0, 0.0}};
/*
 * The value of lu_1.sn.mem_name is unspecified here.
 */
gu.sn.mem_name = 1;
union T_3 lu_2 = gu;

if (lu_2.sn.mem_name != 1)
   print("This is not a conforming implementation\n");
}
```

C++

The C++ Standard does not contain this specification.

---

initialize array of char

An array of character type may be initialized by a character string literal, optionally enclosed in braces. 1664

Commentary

A string literal is represented in storage as a contiguous sequence of characters, an array is a contiguous sequence of members. This specification recognizes parallels between them. The rationale for permitting optional braces is the same as that for scalars.

initializer scalar 1658

Coding Guidelines

What are the cost/benefit issues of expressing the initialization value using a string literal compared to expressing it as a comma separated list of values? The string literal form is likely to have a more worthwhile cost/benefit when:

- the value is likely to be familiar to readers as a character sequence (for instance, it is a word or sentence). There is a benefit in making use of this existing reader knowledge, or
- the majority of the individual characters in the string literal are represented in the visible source using characters, rather than escape sequences, the visually more compact form may require less effort, from a reader, to process.

One situation where use of a string literal may not be cost effective is when the individual character values are used independently of each other. For instance, they represent specific data values or properties. In this case it is possible that macro names have been defined to represent these values. Referencing these macro names in the initializer eliminates the possibility that a change to their value will not be reflected in the initializer.

---

initialize uses successive characters from string literal

Successive characters of the character string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array. 1665

Commentary

The declaration:

```
unsigned char uc[] = "\xFF";
```

is equivalent to:

```
unsigned char uc[2] = { (unsigned char)(char) 0xFF, 0 };
```

C++

The C++ Standard does not specify that the terminating null character is optional, as is shown by an explicit example (8.5.2p2).

An object initialized with a string literal whose terminating null character is not included in the value used to initialize the object, will cause a diagnostic to be issued by a C++ translator.

```
char hello[5] = "world"; /* strictly conforming */
                         // ill-formed
```

Common Implementations

Some implementations store the string literal as part of the program image and copy it during initialization. Others generate machine code to store constants (the individual character values, often concatenated to form larger constants, reducing the number of store instructions) into the object.

Coding Guidelines

Developers sometimes overlook a string literal's terminating null character in their calculation of the number of array elements it will occupy (either leaving insufficient space to hold the null character, when an array size is specified, or forgetting that storage will be allocated for one, in the case of an incomplete array type). There is no obvious guideline recommendation that might reduce the probability of a developer making these kind of mistakes.

Example

```
char a_1[3] = "abc", /* storage holds | a | b | c |         */
     a_2[4] = "abc", /* storage holds | a | b | c | 0 |     */
     a_3[5] = "abc", /* storage holds | a | b | c | 0 | 0 | */
     a_4[]  = "abc"; /* storage holds | a | b | c | 0 |     */
```

In (assuming any undefined behavior does not terminate execution of the program):

```
unsigned char s[] = "\x80\xff";
```

the first element of `s` is assigned the value `(unsigned char)(char)128` and the second element the value `(unsigned char)(char)255`.

1666 An array with element type compatible with **`wchar_t`** may be initialized by a wide string literal, optionally enclosed in braces.

initialize array of wchar_t

Commentary

The applicable issues are the same as for an array of character type.

1664 initialize array of char

Coding Guidelines

The cost/benefit of using a wide string literal in the visible source, rather than a comma separated list, will be affected by the probability that the wide characters will appear, to a reader, as a glyph rather than some multibyte sequence (or other form of encoding).

1664 initialize array of char

1667 Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.

initialize uses successive wchar_t from string literal

Commentary

The applicable issues are the same as for an array of character type

1665 initialize uses successive characters from string literal

---

1668 Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members.

Commentary

The syntax requires a list of initializers to be brace enclosed (the braces serve to disambiguate what would otherwise look like a declarator list). While the braces are not strictly necessary for union types (there is only one initializer), their visual appearance is consistent with braces being used in structure and union type definitions.

Other Languages

Many languages support this form of initializer (although the delimiters are not always braces).

current object brace enclosed initializer

Each brace-enclosed initializer list has an associated *current object*. 1669

Commentary

This introduces the term *current object*. This terminology is new in C99 and is not commonly used by developers. Current objects are associated only with brace-enclosed initializer lists.

footnote 127 1685

A lot of background knowledge of how "things are supposed to work" is needed to understand out how the *current object* maps to the object being initialized. Some of this knowledge is encoded in the extensive examples provided in the Standard.

C90

The concept of *current object* is new in C99.

C++

The concept of *current object* is new in C99 and is not specified in the C++ Standard. It is not needed because the ordering of the initializer is specified (and the complications of designation initializers don't exist, because they are not supported):

8.5.1p2 *. . . , written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate.*

Other Languages

Fortran does not restrict the initializer list to providing initial values for one object. Its **DATA** statement may contain a list of separate, unrelated, objects followed by a list of values (used to initialize the separate objects at program startup).

initialization no designator member ordering

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 named member of a union.[127)] 1670

Commentary

This selection of mapping between members and initializers is the one that is most consistent with what people are likely to expect to occur.

C90

Otherwise, the initializer for an object that has aggregate type shall be a brace-enclosed list of initializers for the members of the aggregate, written in increasing subscript or member order; and the initializer for an object that has union type shall be a brace-enclosed initializer for the first member of the union.

The wording specifying named member was added by the response to DR #016.

Other Languages

This convention is common to most languages that support some form of initializer.

Coding Guidelines

An object declaration, whose derived type includes a structure type, that includes an initializer where no designations are present contains an ordering dependency between the structure members and the values given in the initializer. For instance, the following definition contains a member ordering assumption that is not guaranteed by the C Standard:

EXAMPLE div_t 1702

```
#include <stdlib.h>

div_t val = {3, 4};
```

Is there worthwhile cost/benefit in a guideline recommendation specifying that initialization values corresponding to structure members always include a designator? Designators are not supported by C++ or C90, so such a recommendation would be C99 specific. The following are some of the potential benefits of using designators in the initialization lists of objects having a structure type:

initialization using a designator

- Removing the dependency between members and their corresponding initialization value. This benefit is only realized if the ordering of existing members changes (adding new members after the existing members does not change the existing dependencies). The possibility of developers writing code that contains dependencies on member ordering is appreciated by developers (and third-party library vendors) and the cost of changes to existing code (customer complaints) are factored into any consideration of changing structure member order. Experience shows that new members are usually added to the end of existing structure types.
- As a memory aid for the reader of the initializer. While comprehending an initializer readers have to process its components, while remembering information about their position within the being initialized. Having a designator visible provides a mechanism for readers to for holding and refreshing information in the visuo-spatial sketch pad, freeing up other working memory resources for other tasks.

visuo-spatial memory
memory developer

Given the existing practice of adding new members to the end of an existing structure type (the C++/C90 compatibility issue could be solved via the use of a macro) there does not appear to be sufficient benefit for a guideline recommending that designators always be used.

Initialization of objects having an array type involves a (potentially) large number of values of the same type. Experience shows that the value of many elements is often zero. Designators provide a method of reducing the number of zeroes that appear in the source. However, specifying the conditions under which there is a worthwhile cost/benefit for their usage is likely to involve a complex calculation. At the time of this writing there is insufficient experience with the use of this construct to know whether simple rules can be specified. For this reason no guideline recommendation is given.

1671 In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator.

initialization using a declarator

Commentary

The designator explicitly specifies which member (of the current object) the initialization value refers to.

```
int a[5] = { [2] = 3 };
```

C90

Support for designations is new in C99.

C++

Support for designations is new in C99, and is not available in C++

Other Languages

Some languages (e.g., Ada) support an equivalent construct.

Coding Guidelines

Some of the costs and benefits of using designators are discussed elsewhere.

1670 initialization no designator member ordering

1672 Initialization then continues forward in order, beginning with the next subobject after that described by the designator.[128)]

Commentary

The designation can be thought of as a “goto” within the current object. Once a designator has specified a member at which initialization is to occur, subsequent initializations continue from that point until another designator or the end of the type of the current object is reached.

Other Languages

Ada requires that initializers either be written without using any designators or be written only using designators (i.e., no mixing).

Coding Guidelines

initialization 1670 no designator member ordering

Is the use of designators an all or nothing choice? This question is a special case of the member ordering dependency issue discussed elsewhere. At the time of this writing there is not sufficient experience available with the use of this construct to be able to reliably make any recommendations.

In the two definitions:

```
int a1[] = {
            [10] = 5, 9, 3,
            [23] = 1, 0, 16,
           };
int a2[] = {
            [10] = 5, [11] = 9, [12] = 3,
            [23] = 1, [24] = 0, [25] =16,
           };
```

it is possible to imagine the following two scenarios— where modifications to a program:

- require the three initial values starting at [10] need to be offset by four elements, to [14]. In this case there is greater potential for mistakes being introduced by incorrectly modifying (or not modifying) the two following designators, in the initializer for a2, or
- require that the initial value at [10] be moved by four elements to [14]. In this case there is greater potential for mistakes being introduced by failing to create a designator for [11] (or incorrectly specifying it), in the initializer for a1.

---

designator list current object

Each designator list begins its description with the current object associated with the closest surrounding brace pair. 1673

Commentary

Here usage of the term *designator list* refers to the syntactic terminal of that name. The member specified by a designator is context dependent. While it is not necessary to specify the path from the outermost object to the current object, it is necessary to specify a path to any nested subobject being initialized (or use nested braces). An initializer enclosed in a brace pair is used to initialize the members of a contained subaggregate or union.

initializer 1679 brace pair

C90

Support for designators is new in C99.

C++

Support for designators is new in C99, and is not available in C++

Coding Guidelines

It is possible to write initializers without using any nested brace pair. However, the additional cost of inserting pairs of braces into an initializer is minimal and the potential benefit is large. These benefits include the following:

- Providing a visual aid that can be used, in conjunction with white space and new line characters, to highlight individual sequences of initializers. In particular the appearance of an opening brace helps disambiguate whether initializers at the start of a line are a continuation of the initializers from the previous line, or denote the start of the initializers for a different object. A closing brace provides explicit visual information that any following initializers belong to a different subobject.

- An increased likelihood of developer mistakes (e.g., a missing initializer value or unintended additional initializer value) resulting in diagnostics being generated by translators. It is possible to specify fewer initializers than there are objects to initialize, and many initializers and objects have an integer type (which reduces the probability of a mismatched initializer value and object having an incompatible type, i.e., no diagnostic will be generated). An opening brace may only appear when the current object has an aggregate or union type, and translators are required to perform this check. There is a limit to the number of initializers that can occur between matching braces (the number of subobjects being initialized) and translators are required to perform this check.

Enclosing initializers in matching braces isolates them from other initializers for other members. Such usage also enables translators to perform a finer grained level of checking on the expected number of initializers for each member.

Cg 1673.1 Any optional braces shall not be omitted in an initializer for an aggregate or union type.

Example

In the following, the initializers all assign the same value to the same members:

```
struct S {
          char m_1_1;
          struct {
                  char m_2_1;
                  struct {
                          char m_3_1;
                          char m_3_2;
                         } m_2_2;
                  char m_2_3;
                 } m_1_2;
          char m_1_3;
         };

struct S x_1 = {'a',           'b', 0,             'c', 0, 'd'};
struct S x_2 = {'a', .m_1_2 = {'b',      {.m_3_2 = 'c'} }, 'd'};
struct S x_3 = {'a',          {'b',            {0, 'c'} }, 'd'};
struct S x_4 = {'a', {.m_2_1 = 'b', .m_2_2.m_3_2 = 'c'},   'd'};
```

1674 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.[129)]

Commentary

Each designator list is independent of any other designator list and any change of current object only applies during the evaluation of a given designator list.

1689 designator list independent

Coding Guidelines

The cost/benefit issues associated with using designators are discussed elsewhere. This coding guideline subsection discusses whether any guideline recommendations on the forms of usage, if it was decided to use them in initializers, might be cost effective.

1670 initialization no designator member ordering
1670 initialization using a designator

There may be a reason for automatically generated source to initialize members in what appears to be a random order. Having a designator list specify members in the same order as they occur in the declaration of the aggregate type is a consist mapping in many cultures and is also consistent with the order used when no designators are present. However, one of the benefits of using designators is that the relative order of values within an initializer is independent of the order used in the declaration of an aggregate.

1670 initialization no designator member ordering

There is no evidence to suggest that (designators are a new construct and at the time of this writing little experience has been gained in how they are used), in practice, developers will commonly order designators in any way other than an order that matches the order of the member in the declaration of the aggregate type (or at least the one visible at the time the code was first written, which may subsequently change). Until more experience has been gained in how developers use this construct there is no point in idle speculation over what guideline recommendations might be appropriate.

Example

In the following the initializers all assign the same values to the same members:

```
struct S {
         char m_1_1;
         struct {
                char m_2_1;
                struct {
                          char m_3_1;
                          char m_3_2;
                         } m_2_2;
                char m_2_3;
                } m_1_2;
         char m_1_3;
         };

struct S x_1 = {'a',                'b', 0,             'c'      , 0, 'd'};
struct S x_2 = {'a',       {.m_2_1 = 'b', .m_2_2.m_3_2 = 'c'},        'd'};
struct S x_3 = {'a', .m_1_2.m_2_1 = 'b', .m_1_2.m_2_2.m_3_2 = 'c', 0, 'd'};
```

The current object that results at the end of the designator list is the subobject to be initialized by the following initializer. 1675

Commentary

The following initializer can be a brace enclosed initializer.

```
struct S {
         char m_1_1;
         struct {
                struct {
                        char m_3_1;
                        char m_3_2;
                        } m_2_1;
                char m_2_2;
                } m_1_2;
         };
struct S x_1 = {.m_1_2 =       {1, 2,  0}};
struct S x_2 = {.m_1_2 =     { {1, 2}, 0}};
struct S x_3 = {.m_1_2.m_2_1 = {1, 2}};
```

initialization in list order

The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;[130)] 1676

Commentary

Objects with automatic storage duration are not required to have initializers that are constant expressions. Expressions can cause side effects. While this requirement specifies the order in which initialization occurs it does not specify the order in which the initializers are evaluated. The following declaration:

footnote 130 1690

```
union { char x[5]; } u = { .x[1] = 7 };
```

is a shorthand for:

```
union { char x[5]; } u = { .x = { [1] = 7 } };
```

Expanding out shorthand forms can help make it easier to deduce the initial values of some members. For instance, in:

```
union {
       char a[5];
       char b[5];
      } u = {
              .a[1] = 7,
              .b[2] = 8
            };
```

it might be thought that the value of `u.b[1]` is `7`. However, writing it out in non-shorthand form we get:

```
union {
       char a[5];
       char b[5];
      } u = {
             .a = { [1] = 7 },
             .b = { [2] = 8 }
            };
```

where it is easily seen that the value of `u.b[1]` is `0`.

C++

The C++ Standard does not support designators and so it is not possible to specify more than one initializer for an object.

Other Languages

Ada does not permit more than one initializer to be specified for the same subobject.

Common Implementations

It is too early to know whether it is worthwhile for optimizers to invest in looking for and removing overridden initializers. Initializers that are overridden and do not generate side effects or are depended on by other initializers may be removed general dead code elimination optimizations.

Coding Guidelines

The guideline recommendation dealing with the evaluation ordering between sequence points is applicable here. Providing an initializer for an object that is subsequently overridden might be treated as suspicious for several reasons, including the following:

?? sequence points all orderings give same value
1704 EXAMPLE overriding values

- A reader may only see the first initializer value and not any later ones, leading to an incorrect interpretation of program behavior.
- Evaluation of the initializer may be expected to cause side effects, or be used as an operand in the initializer for another subobject. Not evaluating an initializer, that is overridden, may have unexpected consequences.

```
extern int glob;
struct S {
         int m_1;
         int m_2;
         int m_3;
         };
```

```

void f(void)
{
struct S loc = {
                .m_1 = glob++,
                .m_2 = loc.m2+1,
                .m_1 = 1
                };
}
```

dead code

- The overridden initializer is essentially dead code. This issue is discussed elsewhere.
- The usage might be considered suspicious, especially if another member had not been explicitly initialized.

Designations are new in C99 and it is too early to know whether a specific guideline recommending against their use is worthwhile (there are many constructs guidelines could recommend against, but don't because they rarely appear in source code). The conclusions of the discussion on dead code may be sufficient.

dead code

---

object initialized but not explicitly

all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration. 1677

Commentary

The presence of an initializer in an object definition causes different behavior than the absence of an initializer. If the definition includes an initializer, even if it only initializes a single member, all members are given a known value. However, all members of an object defined without an initializer (and not having static storage duration) have an indeterminate value, at the point of definition.

object indeterminate each time declaration reached

static initialization default value 1653

The implicit value assigned to objects that have static storage duration is zero (or null).

C++

The C++ Standard specifies that objects having static storage duration are zero-initialized (8.5p6), while members that are not explicitly initialized are default-initialized (8.5.1p7). If constructs that are only available in C are used the behavior is the same as that specified in the C Standard.

Other Languages

Some languages (e.g., Ada and Extended Pascal) require that initializers be specified for all subobjects.

Common Implementations

Two implementation strategies are to zero all members of the initializer (using a loop) and then assigning specific values to specific members, or to assign specific values (which may include the implicit values) to all members. The better optimizers perform a cost/benefit analysis to select the strategy to use (which for large objects may involve applying different ones for different subobjects).

Coding Guidelines

coding guidelines other documents

A general principle promoted in many coding guideline documents is that all operations should be explicit. However, explicitly specifying zero values for members of an aggregate object has a number of costs that the implicit usage does not, including the following:

- A large number of zeros in the visible source increase the effort needed, by readers, to locate the nonzero values. The zeros have become visual noise, that do not provide information to readers.
- Increased maintenance costs. Having to update the initializer list every time the aggregate type changes (either because of a change in the number of elements, or the number of members). If designators are used member names may also need to be updated.

Given these costs and the fact that developers are generally aware of the default behavior, there does not appear to be a worthwhile benefit in a guideline recommending that the behavior be made explicit.

1678 If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions.

Commentary

There are no special cases that apply to nested aggregates or unions.

1679 If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union.

initializer brace pair

Commentary

The brace pair provide a mechanism for explicitly specifying which initializers belong to a subaggregate or contained union. Only the initializers enclosed by the braces are used to initialize its corresponding subaggregate or union. If there are fewer initializers than members the remaining members are implicitly initialized. Each brace-enclosed initializer list also has an associated current object.

1669 current object brace enclosed initializer

Other Languages

Fortran uses parenthesis and forward slash (e.g., `(...)` and `/.../`), Ada uses parenthesis (e.g., `(...)`) and Extended Pascal uses square brackets (e.g., `[...]`).

Coding Guidelines

The guideline recommendation that braces always be used is discussed elsewhere.

1673.1 initializer use braces

1680 Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union;

Commentary

If the initializer values are not enclosed by a matching pair of braces, values from the current list of initializers are used.

Other Languages

The Fortran `DATA` statement also specifies this behavior. However, most languages that support some form of initialization, associated with an objects declaration, require stricter correspondence between initializer value and subobject being initialized.

Coding Guidelines

The coding guideline issues associated with this form of initializer are discussed elsewhere.

1673.1 initializer use braces

1681 any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part.

Commentary

That is, any remaining initializers up until the matching right brace.

C90

This behavior was not pointed out in the C90 Standard.

1682 If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

initializer fewer in list than members

Commentary

This wording spells out particular cases, removing the possibility of other interpretations being applied to the general wording on implicit initialization given earlier.

1677 object initialized but not explicitly

C90

The string literal case was not explicitly specified in the C90 Standard, but was added by the response to DR #060.

C++

The C++ Standard specifies that objects having static storage duration are zero-initialized (8.5p6), while members that are not explicitly initialized are default-initialized (8.5.1p7). If only constructs available in C are used the behavior is the same as that specified in the C Standard.

Other Languages

Many other languages (e.g., Ada and Extended Pascal) do not explicitly specify a behavior for this case.

Coding Guidelines

object 1677 initialized but not explicitly

The coding guideline discussion given elsewhere is applicable here.

---

array of unknown size initialized

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. 1683

Commentary

An initializer provides another method of specifying the size (number of elements) of an array type.

C90

> *If an array of unknown size is initialized, its size is determined by the number of initializers provided for its elements.*

Support for designators is new in C99.

Other Languages

Ada specifies a similar rule for deducing both the lower and upper bounds of an array.

Coding Guidelines

Specifying the size of the array via the contents of an initializer reduces the amount of effort needed to write the array definition. In this case the number of elements in the array has to be obtained using a constant expression of the form `(sizeof(arr) / sizeof(arr[0]))`. However, in some cases developers do not specify a size between the `[]` tokens, even although the value is readily available to them. Such usage becomes a potential cause of maintenance problems if two unrelated mechanisms are used to denote the number of array elements (e.g., some numeric literal, and an expression using **`sizeof`**). Using the known size value in the declaration of the array provides some degree of checking (i.e., translators are required to issue a diagnostic if too many initializers are specified, but not if too few are specified).

Example

```
#define A_SIZE 9

int a_1[]       = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int a_2[A_SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; /* Diagnostic issued. */
int a_3[A_SIZE] = {0, 1, 2, 3, 4, 5, 6, 7      }; /* Diagnostic not issued. */
```

---

initializer completes incomplete array type

At the end of its initializer list, the array no longer has incomplete type. 1684

Commentary

The initializer provides all of the information needed to complete the type. However, the single pass translation nature of C means that the completion of the type does not affect any constraint requirements that apply to constructs appearing earlier in the processing of the declaration. For instance, in: implementation single pass

```
int a[][] = {{1}, {2}, {3}};    /* Only one array size can be deduced from initializer. */
int b[sizeof(b)*2] = {1, 2, 3}; /* Operand of sizeof has incomplete type when it is processed. */
```

C++

The C++ Standard does not specify how an incomplete array type can be completed. But the example in 8.5.1p4 suggests that with an object definition, using an incomplete array type, the initializer creates a new array type. The C++ Standard seems to create a new type, rather than completing the existing incomplete one (which is defined, 8.3.4p1, as being a different type).

1685 127) 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. footnote 127

Commentary

```
struct S {
         int m_1_1;
         struct {
                int m_2_1;
                int m_2_2;
                } m_1_2;
         int m_1_3;
         };

struct S x_1 = {1,
                2, /* Current object does not become m_1_2 */
       .m_2_2 = 3, /* Constraint violation, designator should be .m_1_2.m_2_2 */
                4
               };
```

C90

The concept of current object is new in C99.

C++

The concept of current object is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

Developer misunderstandings, in this case, about what constitutes the current object could be harmless in that incorrect use of designators causes a diagnostic to be issued. However, the same member name appearing on its own (e.g., `.next`) in more than one designator is a possible source of reader visual confusion. One way of reducing the possibility of visual confusion is to use additional members in designator (e.g., `.left.next` and `.right.next`). But this usage is dependent on which subaggregate the current member denotes, which will be affected by the brace nesting. Until more experience has been gained in the use of designators it is not possible to estimate the cost/benefit trade-offs involved in using braces or designators containing more member selections.

1686 128) After a union member is initialized, the next object is not the next member of the union; footnote 128

Commentary

Only one member of a union is required to hold a value at any time. footnote 37

C90

The C90 Standard explicitly specifies, in all the relevant places in the text, that only the first member of a union is initialized.

---

instead, it is the next subobject of an object containing the union. 1687

Commentary

member initialized recursively 1656

Or, if that is a union or if the last member of an aggregate has just been processed, the next subobject of the containing type (and so on returning through any recursion) .

---

footnote 129

129) Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. 1688

Commentary

A designator cannot, for instance, denote a member of an object contained in the type of a member outside of the surrounding brace.

```
struct S {
          int m_1_1;
          struct {
                  int m_2_1;
                  int m_2_2;
                 } m_1_2;
          int m_1_3;
         };

struct S x_1 = {1,
                  {2,
                   .m_1_3 = 3 /* Constraint violation. */
                  }
               };
```

Coding Guidelines

Any attempt to specify, in a designator, a member that is not in the strict subobject will cause a diagnostic to be generated.

---

designator list independent

Note, too, that each separate designator list is independent. 1689

Commentary

current object brace enclosed initializer 1669

Each separate designator list is independent in the sense that a designator list does not change the current object in the way that a brace-enclosed initializer does.

```
struct S {
          int m_1_1;
          struct {
                  int m_2_1;
                  int m_2_2;
                 } m_1_2;
         };
struct S x_1 = {
                .m_1_2.m_2_1 = 1, /* Does not change the current object to be m_1_2. */
                      .m_2_2 = 3  /* Constraint violation. */
               };
struct S x_2 = {
                .m_1_2 = {.m_2_1 = 1,  /* { Changes the current object to be m_1_2. */
                          .m_2_2 = 3}  /* OK. */
               };
```

1690 130)Any initializer for the subobject which is overridden and so not used to initialize that subobject might not be evaluated at all.

footnote 130

Commentary

This sentence was added by the response to DR #208.

DR #208

Committee Response

The question asks about the expression

```
int a [2] = { f (0), f (1), [0] = f (2) };
```

and the meaning of the wording

each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;

It was the intention of WG14 that the call `f(0)` might, but need not, be made when a is initialized. If the call is made, the order in which `f(0)` and `f(2)` occur is unspecified (as is the order in which `f(1)` occurs relative to both of these). Whether or not the call is made, the result of `f(2)` is used to initialize `a[0]`.

The wording of paragraph 23:

The order in which any side effects occur among the initialization list expressions is unspecified.

should be taken to only apply to those side effects which actually occur.

1691 The order in which any side effects occur among the initialization list expressions is unspecified.[131)]

Commentary

full expression initializer

An initializer was defined to be a full expression in C90. However, support for nonconstant initial values is new in C99 and this sentence points out the consequences. While operators (the comma in an initialization list is a punctuator, not an operator) that have sequence points may occur in the initialization list, just like in an expression, there is no requirement that these sequence points occur in a particular order.

C90

The C90 Standard requires that the expressions used to initialize an aggregate or union be constant expressions. Whatever the order of evaluation used the external behavior is likely to be the same (it is possible that one or more members of a structure type are volatile-qualified).

C++

The C++ Standard does not explicitly specify any behavior for the order of side effects among the initialization list expressions (which implies unspecified behavior in this case).

Other Languages

Ada explicitly states that the order of evaluation is unspecified.

Coding Guidelines

?? sequence points all orderings give same value

The guideline recommendation dealing with the evaluation order between sequence points is applicable here. The extent to which developers will incorrectly believe the comma punctuators between designators is a sequence point is not known. Correcting any such a belief is an educational issue and is outside the scope of these coding guidelines. Because of the declaration nature of an initializer it is more difficult to break it up into smaller, independent, components (as is possible for expressions in a statement context). The extent to which this will lead to complicated and difficult to comprehend initializers is not known.

Example

```
#include <stdio.h>

void f(void)
{
int loc_1 [] = {printf("Hello"), printf("world")};
}
```

there are two possible character sequences that may be output, either **Helloworld**, or **worldHello**.

EXAMPLE 1 Provided that **<complex.h>** has been **#included**, the declarations 1692

```
int i = 3.5;
double complex c = 5 + 3 * I;
```

define and initialize **i** with the value 3 and **c** with the value *5.0 + i3.0*.

Commentary

In this example the macro I expands to a constant expression of type **const float _Complex**. The initialization value is actually equivalent to 5.0f+i3.0f.

The wording was changed by the response to DR #293.

C90

Support for complex types is new in C99.

C++

The C++ Standard does not define an identifier named **I** in **<complex>**.

Coding Guidelines

integer conversion to floating

Some of the issues applicable to this example are discussed elsewhere.

EXAMPLE 2 The declaration 1693

```
int x[] = { 1, 3, 5 };
```

defines and initializes **x** as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

Commentary

array element not incomplete type

An arrays element type is required to be an object type, not an incomplete type. This means it is not possible to use an initializer to specify the size of more than one array dimension.

```
int y[][] = { {1, 2}, {3, 4}, {5, 6} }; /* Constraint violation. */
```

EXAMPLE 3 The declaration 1694

```
int y[4][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of **y** (the array object **y[0]**), namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise the next two lines initialize **y[1]** and **y[2]**. The initializer ends early, so **y[3]** is initialized with zeros. Precisely the same effect could have been achieved by

```
int y[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y[0]** does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for **y[1]** and **y[2]**.

Other Languages

Some languages (e.g., Ada and Extended Pascal) require the initializer to have a form similar to the first declaration and do not support a form equivalent to the second.

Coding Guidelines

The guideline recommendation that might be applicable to this example is the use braces to delimit initializers for members having an aggregate type.

1673.1 initializer use braces

1695 EXAMPLE 4 The declaration

```
int z[4][3] = {
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **z** as specified and initializes the rest with zeros.

Other Languages

Some languages (e.g., Ada and Extended Pascal) require that initializers be specified for all subobjects.

1696 EXAMPLE 5 The declaration

EXAMPLE inconsistently bracketed initialization

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

is a definition with an inconsistently bracketed initialization. It defines an array with two element structures: **w[0].a[0]** is 1 and **w[1].a[0]** is 2; all the other elements are zero.

Commentary

An additional pair of braces is needed for the declaration:

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

to define an array having a single element (i.e., b is initialized to 2). An example of an alternative form of initializer is given elsewhere.

1703 EXAMPLE designators with inconsistently brackets

1697 131) In particular, the evaluation order need not be the same as the order of subobject initialization.

footnote 131

Commentary

In the following:

```
#include <stdio.h>

void f(void)
{
int loc_1 [] = {2, loc_1[0]+1};
int loc_2 [] = {2, loc_2[0]++};
}
```

the element loc_1[1] is not guaranteed to be initialized with a value that is one greater than loc_1[0]. The initializer for loc_2 exhibits undefined behavior because the same object is modified more than once between sequence points.

full expression initializer

C++

The C++ Standard does explicitly specify the ordering of side effects among the expressions contained in an initialization list.

Common Implementations

While some implementations might evaluate each designator in an initializer in a first to last order, others might treats it as a sequence of independent simple assignments (relying on existing optimization routines within the translator to generate the best quality machine code).

sequence ?? points all orderings give same value

Coding Guidelines

The guideline recommendation dealing with expression evaluation is applicable here.

EXAMPLE 6 The declaration 1698

```
short q[4][3][2] = {
        { 1 },
        { 2, 3 },
        { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: **q[0][0][0]** is 1, **q[1][0][0]** is 2, **q[1][0][1]** is 3, and 4, 5, and 6 initialize **q[2][0][0]**, **q[2][0][1]**, and **q[2][1][0]**, respectively; all the rest are zero. The initializer for **q[0][0]** does not begin with a left brace, so up to six items from the current list may be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for **q[1][0]** and **q[2][0]** do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```
short q[4][3][2] = {
        1, 0, 0, 0, 0, 0,
        2, 3, 0, 0, 0, 0,
        4, 5, 6
};
```

or by:

```
short q[4][3][2] = {
        {
                { 1 },
        },
        {
                { 2, 3 },
        },
        {
                { 4, 5 },
                { 6 },
        }
};
```

in a fully bracketed form.

Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.

designa- 1673 tor list current object

Coding Guidelines

The use of braces to delimit designator lists reduces the probability that changes to the size of any array will cause a fault to be generated (because the change is not reflected in the number of initializer values needed).

Developers are familiar with non-initialized elements implicitly being assigned a value of zero. Explicitly specifying trailing zeros in a designator list requires effort from readers to visually process them.

1699 EXAMPLE 7
One form of initialization that completes array types involves typedef names. Given the declaration

```
typedef int A[];       // OK - declared with block scope
```

the declaration

```
A a = { 1, 2 }, b = { 3, 4, 5 };
```

is identical to

```
int a[] = { 1, 2 }, b[] = { 3, 4, 5 };
```

due to the rules for incomplete types.

Commentary

An example similar to this was submitted as DR #010 against the C90 Standard.

C++

The C++ Standard does not explicitly specify this behavior.

1700 EXAMPLE 8 The declaration

EXAMPLE
array initialization

```
char s[] = "abc", t[3] = "abc";
```

defines “plain” **char** array objects **s** and **t** whose elements are initialized with character string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
     t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines **p** with type “pointer to **char**” and initializes it to point to an object with type “array of **char**” with length 4 whose elements are initialized with a character string literal. If an attempt is made to use **p** to modify the contents of the array, the behavior is undefined.

Commentary

Initialization is a context where the implicit conversion of an array type to a pointer to its first element is dependent on the type of the object it initializes.

array
converted to pointer

C++

The initializer used in the declaration of t would cause a C++ translator to issue a diagnostic. It is not equivalent to the alternative, C, form given below it.

Other Languages

A few languages (e.g., Pascal, provided the, stronger, type compatibility rules are met) allow strings to be assigned to an object having an array of character type.

Coding Guidelines

The issues associated with using a string literal to represent a sequence of character constants are discussed elsewhere.

1664 initialize
array of char

1701 EXAMPLE 9 Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```
enum { member_one, member_two };
const char *nm[] = {
        [member_two] = "member two",
        [member_one] = "member one",
};
```

Commentary

The only type associations created by this usage exist in the readers head.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

Other Languages

Languages in the Pascal family allow the type of the indexing expression to be specified in the array type declaration. For instance:

```
TYPE
      member_enum = (member_one, member_two);
VAR
      nm[] : Array[member_enum] of char;
```

Coding Guidelines

initialization 1670 no designator member ordering

The issue of the order of designators used in member initialization is discussed elsewhere.

EXAMPLE div_t

EXAMPLE 10 Structure members can be initialized to nonzero values without depending on their order: 1702

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

Commentary

Use of designators in this way removes the dependency between the order of values in an initializer and the order of members in a structure (which in the case of `div_t` is not specified).

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

Coding Guidelines

initialization 1670 no designator member ordering

The issue of member initialization order is discussed elsewhere.

EXAMPLE designators with inconsistently brackets

EXAMPLE 11 Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood: 1703

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

Commentary

EXAMPLE 1696 inconsistently bracketed initialization

Initializers for an object having this type are discussed elsewhere.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

EXAMPLE overriding values

EXAMPLE 12 Space can be “allocated” from both ends of an array by using a single designator: 1704

```
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.

Commentary

A MAX value of less than five would be a constraint violation.

1642 initializer value not contained in object

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

Coding Guidelines

The discussion on initialization list order is applicable here.

1676 initialization in list order

1705 EXAMPLE 13 Any member of a union can be initialized:

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

Commentary

This form of initialization has the benefit that it is not dependent on the relative ordering of members within the union.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

1706 **Forward references:** common definitions **<stddef.h>** (7.17).

References

1. Diab Data. *D-CC & D-C++ Compiler Suites User's Guide*. Diab Data, Inc, `www.ddi.com`, 4.3 edition, June 1999.
2. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.
3. A. Pavese and C. Umiltà. Symbolic distance between numerosity and identity moulates stroop-like interference. *Journal of Experimental Psychology: Human Perception and Performance*, 24(5):1535–1545, 1998.
4. J. R. Stroop. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 28:643–662, 1935.
5. L. M. Trick and Z. W. Pylyshyn. What enumeration studies can show us about spatial attention: Evidence for limited capacity preattentive processing. *Journal of Experimental Psychology: Human Perception and Performance*, 19(2):331–351, 1993.
6. L. M. Trick and Z. W. Pylyshyn. Why are small and large numbers enumerated differently? A limited-capacity preattentive stage in vision. *Psychological Review*, 101(1):80–102, 1994.