Paper Number	P0329R4
Date	2017-07-12
Authors	Tim Shen <timshen@google.com> Richard Smith <richard@metafoo.co.uk>
Audience	CWG

P0329R4: Designated Initialization Wording

This is a formal wording for the designated initialization proposal P0329R0.

Changes compared to P0329R3:

- Extend example in [over.ics.list]
- Add Annex C section describing differences from ISO C.

Wording

Change 11.6 [dcl.init]p1 as follows

braced-init-list:
{ initializer-list ,$_{opt}$ *}*
{ designated-initializer-list ,$_{opt}$ *}*
{ }

designated-initializer-list:
designated-initializer-clause
designated-initializer-list , designated-initializer-clause

designated-initializer-clause:
designator brace-or-equal-initializer

designator:
. identifier

Add a new paragraph as 11.6 [dcl.init]p20:

The same *identifier* shall not appear in multiple *designator*s of a *designated-initializer-list*.

Change in 11.6.4 [dcl.init.list]p1:

List-initialization is initialization of an object or reference from a *braced-init-list*. Such an initializer is called an *initializer list*, and the comma-separated *initializer-clause*s of the *initializer-list* ~~list~~ or *designated-initializer-clause*s of the *designated-initializer-list* are called the *elements* of the initializer list. [...]

Add a new bullet at the start of 11.6.4 [dcl.init.list]p3:

If the *braced-init-list* contains a *designated-initializer-list*, `T` shall be an aggregate class. The ordered *identifier*s in the *designator*s of the *designated-initializer-list* shall form a subsequence of the ordered *identifiers* in the direct non-static data members of T. Aggregate initialization is performed ([dcl.init.aggr]). [Example:

```
  struct A { int x; int y; int z; };
  A a{.y = 2, .x = 1};   // error; designator order does not match declaration order
  A b{.x = 1, .z = 2};   // ok, b.y initialized to 0
```

— end example]

Add a new paragraph to 11.6.1 [dcl.init.aggr]:

The initializations of the elements of the aggregate are evaluated in the element order. That is, all value computations and side effects associated with a given element are sequenced before those of any element that follows it in order.

Drafting note: unlike 11.6.4/4, this also covers the initialization of elements for which no initializer is explicitly provided.

Change in 11.6.1 [dcl.init.aggr]p3 and split it into two paragraphs:

When an aggregate is initialized by an initializer list as specified in 11.6.4, the elements of the initializer list are taken as initializers for the elements of the aggregate~~, in order~~. The *explicitly initialized elements* of the aggregate are determined as follows:

- If the initializer list is a *designated-initializer-list*, the aggregate shall be of class type, the *identifier* in each *designator* shall name a direct non-static data member of the class, and the explicitly initialized elements of the aggregate are the elements that are, or contain, those members.
- If the initializer list is an *initializer-list*, the explicitly initialized elements of the aggregate are the first *n* elements of the aggregate, where *n* is the number of elements in the initializer list.
- Otherwise, the initializer list must be `{}`, and there are no explicitly initialized elements.

~~Each~~ For each explicitly initialized element:

- If the element is an anonymous union object and the initializer list is a designated-initializer-list, the anonymous union object is initialized by the *designated-initializer-list* `{ D }`, where *D* is the *designated-initializer-clause* naming a member of the anonymous union object. There shall be only one such *designated-initializer-clause*.
- Otherwise, the element is copy-initialized from the corresponding *initializer-clause* or the *brace-or-equal-initializer* of the corresponding *designated-initializer-clause*. If ~~the initializer-clause is an expression~~ that initializer is of the form *assignment-expression* or `=` *assignment-expression* and a narrowing conversion (11.6.4) is required to convert the expression, the program is ill-formed. [Note: If an initializer ~~clause~~ is itself an initializer list, the element is list-initialized, which will result in a recursive application of the rules in this section if the element is an aggregate. — end note]

[Example: …]

Change 11.6.1 [dcl.init.aggr]p8 as follows, and move it to immediately after the above paragraphs

For ~~If there are fewer initializer-clauses in the list than there are elements in~~ a non-union aggregate, ~~then~~ each element that is not an explicitly initialized element is initialized as follows:
...
[Example: ...

```
  struct A {
    string a;
    int b = 42;
    int c = -1;
  };
```

`A{.c=21}` has the following steps:

1. Initialize a with {}
2. Initialize b with = 42
3. Initialize c with = 21

]

Change 11.6.1 [dcl.init.aggr]p6 as follows

[Note: Static data members, non-static data members of anonymous union members, and anonymous bit-fields are not considered elements of the aggregate. — end note]

Change 11.6.1 [dcl.init.aggr]p7 as follows

An *initializer-list* is ill-formed if the number of *initializer-clauses* exceeds the number of elements ~~to initialize~~ of the aggregate.

Change 11.6.1 [dcl.init.aggr]p16 as follows

When a union is initialized with an ~~brace-enclosed~~ initializer list, there shall not be more than one explicitly initialized element. ~~the braces shall only contain an initializer-clause for the first non-static data member of the union~~. [Example:

```
union u { int a; const char* b; };
   u a = { 1 };
   u b = a;
   u c = 1; // error
   u d = { 0, "asdf" }; // error
   u e = { "asdf" }; // error
   u f = { .b = "asdf" };
   u g = { .a = 1, .b = "asdf" }; // error
```

]

Add new paragraph after 16.3.3.1.5 [over.ics.list]p1 as follows

If the initializer list is a *designated-initializer-list*, a conversion is only possible if the parameter has an aggregate type that can be initialized from the initializer list according to the rules for aggregate initialization ([dcl.init.aggr]), in which case the implicit conversion sequence is a user-defined conversion sequence whose second standard conversion sequence is an identity conversion. [Note: Aggregate initialization does not require that the members are declared in designation order. If, after overload resolution, the order does not match for the selected overload, the initialization of the parameter will be ill-formed ([dcl.init.list]). [Example:

```
  struct A { int x, y; };
  struct B { int y, x; };
  void f(A a, int); //#1
  void f(B b, ...); //#2
  void g(A a); //#3
  void g(B b); //#4
  void h() {
    f({.x = 1, .y = 2}, 0); // OK; calls #1
    f({.y = 2, .x = 1}, 0); // error: selects #1, initialization of a fails
                            // due to non-matching member order ([dcl.init.list])
    g({.x = 1, .y = 2});    // error: ambiguous between #3 and #4
  }
```

— end example] — end note]

Change 16.3.3.1.5 [over.ics.list]p2 as follows

Otherwise, if the parameter type is an aggregate [...]

Add new section to C.1.7 [diff.decl] as follows

[dcl.init.aggr]
Change: In C++, designated initialization support is restricted compared to the corresponding functionality in C. In C++, designators for non-static data members must be specified in declaration order, designators for array elements and nested designators are not supported, and designated and non-designated initializers cannot be mixed in the same initializer list.
Example:

```
struct A { int x, y; };
struct B { struct A a; };
struct A a = {.y = 1, .x = 2}; // valid C, invalid C++
int arr[3] = {[1] = 5};        // valid C, invalid C++
struct B b = {.a.x = 0};       // valid C, invalid C++
struct A a = {.x = 1, 2};      // valid C, invalid C++
```

Rationale: In C++, members are destroyed in reverse construction order and the elements of an initializer list are evaluated in lexical order, so field initializers must be specified in order. Array designators conflict with *lambda-expression* syntax. Nested designators are seldom used.
Effect on original feature: Deletion of feature that is incompatible with C++.
Difficulty of converting: Syntactic transformation.
How widely used: Out-of-order initializers are common. The other features are seldom used.