Doc No: WG21/N2531=J16/08-0041
Date: 2008-02-02
Reply to: Bjarne Stroustrup
bs@cs.tamu.edu

Initializer lists WP wording (Revision 2)

J. Stephen Adamczyk, Gabriel Dos Reis, Bjarne Stroustrup

Abstract

This is the proposed WP wording for the initializer proposal as described in N2215 *Initializer lists* with the clarifications explained in N2477 *Uniform initialization design choices* and its revision N2532. There are four intentional differences between the wording here and the design presented in N2215:

- The term "sequence constructor" has been replaced by "initializer-list constructor" to minimize the confusion with other kinds of sequences and lists.
- The meaning of an initializer list as the subscript for an array has been specified.
- Narrowing is defined (as agreed by EWG and CWG in Kona) so that all floating-point to integer conversions are considered narrowing.
- A match on **f(std::initializer_list<T>)** is preferred to a match on **f(T)** where **T** is a template parameter.

This wording also reflects the clarifications from N2477 (as opposed to changes from N2215), as prompted by questions and discussions in Kona:

- Non-narrowing and direct-initialization now applies to aggregate element initialization and to the arguments passed to constructors for initializer-list initialization.
- The initialization and overload resolution rules have been reworked to eliminate any preference between different initializer-list constructors. Initializer-list constructors are still preferred over other kinds of matches, but an initializer-list constructor that exactly matches the inferred type is no longer any better than one that requires conversions on the list members.
- The initializer for an initializer-list constructor must be (syntactically) an initializer list or a **std::initializer_list** object.
- A type **std::initializer_list<Some_type>** can be deduced from a homogenous (without conversions) initializer list for a template parameter **T** or an **auto** variable. Similarly, the **T** in **std::initializer_list<T>** can be deduced.

This paper is a revision of N2385.

In 8.5 [dcl.init], change

initializer:

= initializer-clause
~~*(expression-list)*~~
direct-initializer

initializer-clause:
assignment-expression
~~*{ initializer-list ,opt }*~~
~~*{ }*~~
init-list

initializer-list:
initializer-clause ...opt
initializer-list , initializer-clause ...opt

direct-initializer:
(*expression-list*)
untyped-init-list

untyped-init-list:
bare-init-list

init-list:
untyped-init-list
typed-init-list

bare-init-list:
{ *initializer-list* ,*opt* }
{ }

typed-init-list:
simple-type-specifier bare-init-list
typename-specifier bare-init-list

In 5.2 [expr.post], change

postfix-expression:
…
postfix-expression [*expression*]
***postfix-expression* [*init-list*]**
…
expression-list:
~~*assignment-expression ...opt*~~
~~*expression-list , assignment-expression ...opt*~~
initializer-list

In 5.3.4 [expr.new], change

new-initializer:
~~*(expression-list$_{opt}$)*~~
direct-initializer

In 5.17 [expr.ass], change

assignment-expression:
conditional-expression
~~*logical-or-expression assignment-operator assignment-expression*~~
logical-or-expression assignment-operator initializer-clause
throw-expression

In 6.6 [stmt.jump], change

jump-statement:
…
return *expression$_{opt}$* ;
return *init-list* ;
…

In 12.6.2 [class.base.init], change

mem-initializer:
~~*mem-initializer-id (expression-list$_{opt}$)*~~
mem-initializer-id direct-initializer

In 8.5 [dcl.init], change paragraph 12:

The initialization that occurs in argument passing, function return, throwing an exception (15.1), **and** handling an exception (15.3~~), and brace-enclosed initializer lists (8.5.1)~~ is called copy-initialization and is equivalent to the form

In 8.5 [dcl.init], replace paragraph 14:

~~If T is a scalar type, then a declaration of the form~~
~~`T x = { a };`~~
~~is equivalent to~~
~~`T x = a;`~~

Initialization from a brace-enclosed initializer list is called list-initialization ([dcl.init.list]). The form using "=", where allowed, is equivalent to the form without "=". [***Example:***

```
T x = { a, b, c };
T y { a, b, c };
```

--- ***end example*** **]**

In 8.5 [dcl.init], change the beginning of paragraph 15:

The semantics of initializers are as follows. The *destination type* is the type of the object or reference being initialized and the *source type* is the type of the initializer expression. The source type is not defined when the initializer is **an initializer list** or when it is a parenthesized list of expressions.

- If the destination type is a reference type, see 8.5.3.
- If the destination type is an array of characters, an array of `char16_t`, an array of `char32_t`, or an array of `wchar_t`, and the initializer is a string literal, see 8.5.2.
- **If the initializer is an initializer list (i.e., an *untyped-init-list* or *typed-init-list*), see [dcl.init.list].**
- **If the initializer is ( ), the object is value-initialized.**
- Otherwise, if the destination type is an array, ~~see 8.5.1~~ **the program is ill-formed**.
- If the destination type is a (possibly cv-qualified) class type:
  - ~~If the class is an aggregate (8.5.1), and the initializer is a brace-enclosed list, see 8.5.1.~~
  - If the initialization is direct-initialization, …

In 8.5.1 [decl.init.aggr] paragraph 2, change

~~When an aggregate is initialized the *initializer* can contain an *initializer-clause* consisting of a brace-enclosed , comma-separated list of *initializer-clause*~~ **When, as specified in 8.5.4 [dcl.init.list], an aggregate is initialized by an initializer list, the elements of the initializer list are taken as initializers** for the members of the aggregate, written in increasing subscript or member order. ~~If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate.~~ **Each member is direct-initialized from the corresponding *initializer-clause* according to the initialization rules in 8.5 [dcl.init]; if the *initializer-clause* is an expression, and a narrowing conversion ([dcl.init.list]) is required to convert it to the member type, the program is ill-formed. [** ***Note:*** **If an *initializer-clause* is itself an initializer list, the member is list-initialized, and if the member is an aggregate that will result in a recursive application of the rules in this section.]** [*Example:* …

In 8.5 [dcl.init], add a new section as 8.5.4 [dcl.init.list]:

8.5.4 **List-initialization** [dcl.init.list]

List-initialization is initialization of an object or reference from a brace-enclosed list having the form of an *untyped-init-list* or *typed-init-list*. Such an initializer is called an *initializer list*, and the comma-separated expressions or nested initializer lists of the list are called the *elements* of the initializer list. An initializer list may be empty. [*Example:*

```
int a = {1};
complex<double> z{1,2};
new vector<string>{"once", "upon", "a", "time"}; // 4 string elements
f( {"Nicholas", "Annemarie"} );     // pass list of two elements
return { "Norah" };  // return list of one element
int* e {};  // initialization to zero / null pointer
x=double{1};  // explicitly construct a double
map<string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };
```

--- *end example*]

[*Note*: List-initialization can be used

- as the initializer in a variable definition (8.5 [dcl.init])
- as the initializer in a new expression (5.3.4 [expr.new])
- in a return statement (6.6.3 [stmt.return])
- as a function argument (5.2.2 [expr.call])
- as a subscript (5.2.1 [expr.sub])
- as an argument to a constructor invocation (8.5 [dcl.init], 5.2.3 [expr.type.conv])
- as a base-or-member initializer (12.6.2 [class.base.init])

--- *end note*]

The type `std::initializer_list` ([support.initlist]) has a special relationship to initializer lists. In certain contexts (see below, [over.ics.list], and [temp.deduct.call]), an initializer list can be implicitly converted to an `initializer_list` object that points to an array containing the elements of the initializer list. Constructors and other functions can be written to accept an initializer list in this form. Of particular utility is an *initializer-list constructor*, a constructor taking a single argument of type `std::initializer_list<E>` for some type `E`. [*Note:* Initializer-list constructors are favored over other constructors in certain contexts.] The type `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `initializer_list` (even an implicit use in which the type is not named), the program is ill-formed.

The *simple-type-specifier* or *typename-specifier* of a *typed-init-list* specifies the type of such a list, which shall not be (possibly cv-qualified) `void`.

The *inferred type* of an initializer list is its specified type, for a *typed-init-list*. An *untyped-init-list* has an inferred type only if it has at least one element, and all of its elements --- after application of lvalue-to-rvalue (4.1 [conv.lvalue]), array-to-

pointer (4.2 [conv.array]), and function-to-pointer (4.3 [conv.func]) conversions, if applicable --- have the same type E [*Note:* This type is possibly an inferred type from an initializer list or an initializer-list expression]. In that case, the inferred type of the initializer list is `std::initializer_list<E>`. [*Note:* As described in 14.8.2.1 [temp.deduct.call], the inferred type of an initializer list is used for template argument deduction. [*Example:*

```
template<class T> void f(T);
f({});          // error: cannot deduce type of empty initializer list
f({1,2,3});     // ok: T is initializer_list<int>
f({1,2,3,4.0}); // error: the list is not homogenous without conversions
int a [10];
int *p = a;
f({p,a});       // ok: array decay accepted
f({p,0});       // error: the list is not homogenous without conversions
```

--- *end example*]]

List-initialization of an object or reference of type T is defined as follows. If the initializer list is a *typed-init-list*, its type shall match T.

1. If T is an aggregate, do aggregate initialization (8.5.1 [dcl.init.aggr]). [*Example:*
   ```
   double ad[] = { 1, 2.0 };       // ok
   int ai[] = { 1, 2.0 };          // error: narrowing
   ```
   --- *end example*]
2. Otherwise, if T is a class type, do a modified version of direct-initialization using the elements of the initializer list as arguments. The constructors of T are enumerated according to [over.match.list], and the best one is chosen through overload resolution ([over.match]). If an initializer-list constructor is selected, construct the `initializer_list` object (as described below) and call that initializer-list constructor. If another kind of constructor is selected, the constructor is called with the elements of the initializer list as arguments. [*Note:* as indicated in [over.match.list], the constructor parameters are initialized from the arguments by direct-initialization, and no narrowing conversions are allowed.]
   [*Example:*
   ```
   struct S {
       S(std::initializer_list<double>);     // #1
       S(std::initializer_list<int>);        // #2
       // …
   };
   S s1 = {1.0, 2.0, 3.0 };          // invoke #1
   S s2 = { 1, 2, 3 };               // ambiguous: #1 or #2
   ```
   --- *end example*]

[*Example:*

```
struct Map {
        Map(std::initializer_list<std::pair<std::string,int>>);
};
Map<std::string,int> ship = {{"Sophie",14}, {"Surprise",28}};
```

--- *end example*]

[*Example:*

```
struct S {
        // S(std::initializer_list<double>);
        // no initializer-list constructors
        S(int, double, double);         // #2
        S();    // #3
        // …
};
S s1 = {1, 2, 3.0 };    // ok: invoke #2
S s2 { 1.0, 2, 3 };     // error: narrowing
S s3 { };       // ok: invoke #3

struct S2 {
        int m1;
        double m1,m3;
};
S2 s21 = {1, 2, 3.0 };  // ok
S2 s22 { 1.0, 2, 3 };   // error: narrowing
S2 s23 {};              // ok: default to 0,0,0
```

--- *end example*]

3. Otherwise, if T is a reference type, do list-initialization of an rvalue temporary of the type referenced by T, and bind the reference to that temporary. [*Note:* The binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type.]

   [*Example:*

   ```
   struct S {
           S(std::initializer_list<double>);     // #1
           S(const std::string&);                        // #2
           // …
   };
   const S& r1 = {1, 2, 3.0 };     // ok: invoke #1
   const S& r2 { "Spinach" };  // ok: invoke #2
   S& r3 = { 1, 2, 3 };            // error: initializer is not an lvalue
   ```

   --- *end example*]

4. Otherwise (i.e., if T is not an aggregate, class type, or reference) if the initializer list has a single element, do direct-initialization from it; if a narrowing conversion (see below) is required to convert the element to T, the program is ill-formed; [*Example:*

   ```
   int x1 {2};     // ok
   int x2 {2.0};   // error: narrowing
   ```

```
        string s{"can call explicit constructor"};   // ok
```

--- *end example*]

- if the initializer list has no elements, do value-initialization of the object; [*Example*

```
        int** pp {};    // initialized to null pointer
```

  --- *end example*]
- otherwise, the program is ill-formed.

[*Example:*

```
struct A { int i; int j; };
A a1 { 1, 2 };   // aggregate initialization
A a2 { 1.2 };    // error: narrowing
struct B {
        B(std::initializer_list<int>);
};
B b1 { 1, 2 };    // creates initializer_list<int> and calls constructor
B b2 { 1, 2.0  }; // error: narrowing
struct C {
        C(int i, double j);
};
C c1 = { 1, 2.2 };  // calls constructor with arguments (1, 2.2)
C c2 = { 1.1, 2 };  // error: narrowing

int j { 1 };  // initialize to 1
int k {};     // initialize to 0
```

--- *end example*]

When an initializer list is implicitly converted to a `std::initializer_list<E>`, the object passed is constructed as if the implementation allocated an array of **N** elements of type **E,** where **N** is the number of elements in the initializer list and **E** is the element type deduced or specified for the elements. Each element of that array is initialized with the corresponding element of the initializer list converted to E, and the `initializer_list<E>` object is constructed to refer to that array. [*Example:*

```
void f(std::initializer_list<double> v);
f({ 1,2,3 });
```

The call will be implemented in a way equivalent to this:

```
double __a[3] = {double{1}, double{2}, double{3}};
f(std::initializer_list<double>(__a, __a+3);
```

assuming that the implementation can construct an `initializer_list` with a pair of pointers. --- *end example*]

The lifetime of the `initializer_list` object and the array (and its elements) is identical to that of a temporary created in the same place as the initializer list. [*Example:*

```
typedef std::complex<double> cmplx;
vector<cmplx> v1 = { 1, 2, 3 };
void g(const vector<cmplx>&);

void f()
{
        vector<cmplx> v2 = { 1, 2, 3 };
        g({ 1, 2, 3 });
}
```

In each case, the object and array created for **{ 1, 2, 3 }** have the same lifetime, that is, full-expression lifetime. --- *end example*]

A *narrowing conversion* is an implicit conversion

- from a floating-point type to an integer type, or
- from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or
- from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or
- from an integer type or unscoped enumeration type to an integer type with lesser integer conversion rank (4.13 [conv.rank]) except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type.

[*Note:* As indicated above, such conversions are not allowed at the top level in list-initializations. [*Example:*

```
int x = 999;                    //  x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x;        // ok, might narrow (in this case, it does narrow)
char c2{x};         // error, might narrow
char c3{y};         // error: narrows
char c4{z};         // ok, no narrowing needed
unsigned char uc1= {5}; // ok: no narrowing needed
unsigned char uc2 = {-1}// error: narrows
```

```
unsigned int ui1 = {-1}    // error: narrows
signed int si1 = { (unsigned int)-1 };      // error: narrows
int ii = {2.0};       // error: narrows
float f1 { x };       // error: narrowing
float f2 { 7 };       // ok: 7 can be exactly represented as a float
int f(int);
int a[] = { 2, f(2), f(2.0) };          // ok: the double-to-int conversion
                                        //       is not at the top level
```

--- *end example*]]

In 5.2.1 [expr.sub], add as a new paragraph 2:

An *init-list* may appear as a subscript for a user-defined `operator[]`. In that case, the initializer list is treated as the initializer for the subscript argument of the `operator[]`. An initializer list shall not be used with the built-in subscript operator. [*Example:*

```
struct X {
        Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7;          // ok: meaning x.operator[]({1,2,3})
int a[10];
a[{1,2,3}] = 7;          // error: built-in subscripting
```

--- *end example*]

In 5.3.4 [expr.new] paragraph 16, change part of the bullet list:

- …
- ~~If the *new-initializer* is of the form `()`, the item is value-initialized (8.5);~~
- ~~If the *new-initializer* is of the form `(`*expression-list*`)` and T is a class type, the appropriate constructor is called, using *expression-list* as the arguments (8.5);~~
- If the *new-initializer* is of the form `(`*expression-list*`)` and T is ~~an arithmetic, enumeration, pointer, or pointer-to-member~~ **a scalar** type and *expression-list* comprises exactly one expression, then the object is initialized to the (possibly converted) value of the expression (8.5);
- ~~Otherwise the new-expression is ill-formed.~~
- **Otherwise, the *new-initializer* is interpreted according to the initialization rules of 8.5 [dcl.init] for direct-initialization.**

In 5.17 [expr.ass], add as a new final paragraph:

An initializer list may appear on the right-hand side of

- an assignment to a scalar, in which case the initializer list must have at most a single element. The meaning of **x={v}**, where **T** is the scalar type of the expression **x**, is that of **x=T(v)** except that no narrowing conversion is allowed. The meaning of **x={}** is **x=T()**. If the initializer list is a *typed-init-list*, its type shall be **T**, ignoring cv-qualifiers.
- an assignment defined by a user-defined assignment operator, in which case the meaning is defined by the initialization rules for that operator function's argument.

[*Example:*

```
complex<double> z;
z = { 1,2 };      // meaning z.operator=({1,2})
z += { 1, 2};     // meaning z.operator+=({1,2})
a = b = { 1 };    // meaning a=b=1;
a = { 1 } = b;    // syntax error
```

--- *end example*]

In 6.6.3 [stmt.return] paragraph 2, change

A return statement without an expression can be used only in functions that do not return a value, that is, a function with the return type void, a constructor (12.1), or a destructor (12.4). A return statement with an expression of non-void type can be used only in functions returning a value; the value of the expression is returned to the caller of the function. The expression is implicitly converted to the return type of the function in which it appears. A return statement can involve the construction and copy of a temporary object (12.2). [*Note:* A copy operation associated with a return statement may be elided or considered as an rvalue for the purpose of overload resolution in selecting a constructor (12.8). — *end note*] **A return statement with an *init-list* initializes the object or reference to be returned from the function by list-initialization (8.5.4 [dcl.init.list]) from the specified initializer list.** [*Example:*

```
std::pair<string,int> f(const char* p, int x)
{
        return {p,x};
}
```

--- *end example*]

Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function.

In 7.1.5.4 [dcl.spec.auto], paragraph 6, change

Once the type of a *declarator-id* has been determined according to 8.3, the type of the declared variable using the *declarator-id* is determined from the type of its initializer using the rules for template argument deduction. Let T be the type that has been determined for a variable identifier d. Obtain P from T by replacing the

occurrences of auto with a new invented type template parameter U. Let A be the type of the initializer expression for d. **If the initializer is an initializer list, let A be its inferred type (8.5.4 [dcl.init.list]; as described in 14.8.2.1 [temp.deduct.call], if the initializer list does not have an inferred type the deduction will fail).** [*Example:*

```
auto x1 = { 1,2 };      // x1 is an initializer_list<int>
auto x2 = { 1, 2.0 };   // error: no inferred type
```

--- *end example*]

The type deduced for the variable d is then the deduced type determined using the rules of template argument deduction from a function call (14.8.2.1), where P is a function template parameter type and A is the corresponding argument type. If the deduction fails, the declaration is ill-formed.

In 12.2 [class.temporary], paragraph 3, change

The second context is when a reference is bound to a temporary. The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except as specified below. A temporary bound to a reference member in a constructor's ctor-initializer (12.6.2) persists until the constructor exits. A temporary bound to a reference parameter in a function call (5.2.2) persists until the completion of the full expression containing the call. A temporary bound to the returned value in a function return statement (6.6.3) persists until the function exits. **A temporary bound to a reference in a *new-initializer* (5.3.4 [expr.new]) persists until the completion of the full expression containing the *new-initializer***[*Example:*

```
struct S { int mi; const std::pair<int,int>& mp; };
S a { 1, {2,3} };
S* p = new S{1, {2,3} };        // Creates dangling reference
```

--- *end example*] **[*Note:* This may introduce a dangling reference, and implementations are encouraged to issue a warning in such a case.]** The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary which is constructed earlier in the same full-expression. …

In 12.6.1 [class.expl.init] paragraph 2, change

~~When an aggregate (whether class or array) contains members of class type and is initialized by a brace-enclosed *initializer-list* (8.5.1), each such member is copy-initialized (see 8.5) by the corresponding *assignment-expression*. If there are fewer *initializer* s in the *initializer-list* than members of the aggregate, each member not explicitly initialized shall be value-initialized (8.5). [Note: 8.5.1 describes how *assignment-expression* s in an *initializer-list* are paired with the aggregate members they initialize. — end note]~~ **An object of class type can also be initialized by a brace-enclosed initializer list. List-initialization semantics apply; see 8.5 [dcl.init] and 8.5.4 [dcl.init.list].** [Example: …

In 12.6.2 [class.base.init] paragraph 3, change

The ~~*expression-list*~~ ***direct-initializer*** in a *mem-initializer* is used to initialize the base class or non-static data member subobject denoted by the *mem-initializer-id* **according to the initialization rules of 8.5 [dcl.init] for direct-initialization**. ~~The semantics of a *mem-initializer* are as follows:~~

- ~~if the expression-list of the *mem-initializer* is omitted, the base class or member subobject is value-initialized (see 8.5);~~
- ~~otherwise, the subobject indicated by *mem-initializer-id* is direct-initialized using *expression-list* as the initializer (see 8.5).~~

Add a new section 13.3.1.7:

13.3.1.7 **Initialization by list-initialization** [over.match.list]

When objects of class type are list-initialized ([dcl.init.list]), overload resolution selects the constructor. Assuming that "cv1 T" is the type of the object being initialized, with T a class type, the candidate functions are selected as follows:

- The constructors of T that are not initializer-list constructors are candidates. The argument list is the elements of the initializer list. However, the arguments are considered to direct-initialize the corresponding parameter, rather than copy-initializing it, and a constructor is not considered viable if calling it requires a narrowing conversion ([dcl.init.list]).
- The initializer-list constructors of T are considered. Those whose parameter type is std::initializer_list<X> or reference to const std::initializer_list<X>, where either std::initializer_list<X> is the inferred type of the initializer list, or the initializer list is an *untyped-init-list* and all its elements can be implicitly converted to X without use of a narrowing conversion, are candidate functions. The argument list is a single rvalue object of type std::initializer_list<X>.

Add a new section under 13.3.3.1 [over.best.ics]:

13.3.3.1.5 **List-initialization** [over.ics.list]

When an argument is an initializer list (8.5.4 [dcl.init.list]), it is not an expression and special rules apply for converting it to a parameter type.

If the parameter type is std::initializer_list<X> or reference to const std::initializer_list<X>,

- if the inferred type (8.5.4 [dcl.init.list]) of the initializer list is `std::initializer_list<X>`, the implicit conversion sequence is the identity conversion; [*Example:*

```
void f(std::initializer_list<int>);
f( {1,2,3} );     // ok: f(initializer_list<int>) identity conversion
```

--- end example]

- otherwise, if the initializer list is an *untyped-init-list*, and all the elements of the initializer list can be implicitly converted to X without a narrowing conversion (8.5.4 [dcl.init.list]), the implicit conversion sequence is the identity conversion. [*Example:*

```
struct A {
        A(std::initializer_list<int>);
};
void f(A);
f( {'a', 'b',} );  // ok: f(A(initializer<int>)) identity conversion
f( {1.0} );        // error: narrowing
```

--- end example]

Otherwise, if the parameter type is *cv* X or reference to `const` X where X is a class type, and the argument is a *typed-init-list* with type T or an *untyped-init-list*,

- if X is not an aggregate (8.5.1 [dcl.init.aggr]), if the rules for list-initialization given in 8.5.4 ([dcl.init.list]) choose a single best constructor of X to perform the initialization of an object of type X from the argument initializer list, the implicit conversion sequence is a user-defined conversion sequence; [*Example:*

```
struct A {
        A(int, double);
};
void f(A);
f( {'a', 'b',} );  // ok: f(A(int,double)) user-defined conversion
f( {1.0, 1,0} );  // error: narrowing
```

--- end example]

- if X is an aggregate, then if each element of the initializer list [*Footnote:* There might be zero elements, in which case the requirement is vacuously satisfied.] can be converted to the type of the corresponding initializable member of X according to the rules for aggregate initialization (8.5.1 [dcl.init.aggr]) and there are no more initializers than there are initializable members, the implicit conversion sequence is a user-defined conversion sequence. [*Example:*

```
struct A {
        int m1;
        double m2;
};

void f(A);
f( {'a', 'b',} );  // ok: f(A(int,double)) user-defined conversion
f( {1.0} );        // error: narrowing
```

--- end example]

Otherwise, if the parameter type is a reference to a non-array type, and the reference can be bound to an rvalue of the referenced type (e.g., if it is an lvalue reference to a const type), the implicit conversion sequence is the one required to convert the initializer list to the referenced type according to this section, if such a conversion is possible. [*Example:*

```
struct A {
        int m1;
        double m2;
};

void f(const A&);
f( {'a', 'b',} );  // ok: f(A(int,double)) user-defined conversion
f( {1.0} );        // error: narrowing

void g(const double &);
g({1});            // same conversion as int to double
```

--- end example]

Otherwise, if the parameter type is *cv* X, with X not a class or reference type, and the argument is a *typed-init-list* with type (possibly cv-qualified) X or an *untyped-init-list*,

- if the initializer list has one element, the implicit conversion sequence is the one required to convert the element to the parameter type, if such a conversion is possible without using a narrowing conversion (8.5.4 [dcl.init.list]); [*Example:*

```
void f(int);
f( {'a'} );        // ok: same conversion as char to int
f( {1.0} );        // error: narrowing
```

  --- end example]
- if the initializer list has no elements, the implicit conversion sequence is the identity conversion. [*Example:*

```
void f(int);
f( { } );          // ok: identity conversion
```

  --- end example]

In all cases other than those enumerated above, no conversion is possible.

In 13.3.3.2 [over.ics.rank] paragraph 3, under the bullet beginning “Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if”, add a new final sub-bullet:

- The argument is an initializer list (8.5.4 [decl.init.list]), S1 and S2 are both identity conversions, S1 is a conversion to std::initializer_list<T> or reference to const std::initializer_list<T>, and S2 is not such a conversion.

In 14.5.3 [temp.variadic], paragraph 4, change the first bullet:

- In an expression-list (5.2); the pattern is an ~~*assignment-expression*~~ ***initializer-clause***.

In 14.8.2.1 [temp.deduct.call] paragraph 1, change

> Template argument deduction is done by comparing each function template parameter type (call it P) with the type of the corresponding argument of the call (call it A) as described below. **If the argument is an initializer list (8.5.4 [dcl.init.list]), its inferred type is used for A; if the initializer list does not have an inferred type, the associated parameter is considered a non-deduced context (14.8.2.5 [temp.deduct.type]).** [*Example:*
>
> ```
> template<class T> void f(std::initializer_list<T>);     // #1
> f({1,2,3});          // T deduced to int
> f({1,"asdf"});       // error: no inferred type; no deduction
>
> template<class T> void g(T);
> g({1,2,3});          // T deduced to initializer_list<int>
>
> template<class T> void f(T);     // #2
> f({1,2,3});          // invoke #1 (more specialized); T deduced to int
> ```
>
> --- end example] For a function parameter pack, …

In 14.8.2.5 [temp.deduct.type] paragraph 5, add as a final bullet at the top level (not the second bullet level)

- A function parameter for which the associated argument is an initializer list (8.5.4 [dcl.init.list]) that does not have an inferred type. [*Example:*

  ```
  template<class T> void f(std::initializer_list<T>);          // #1
  f({1,"asdf"}); // error: no inferred type; no deduction

  template<class T> void g(T);
  g({1,2.0});              // error: no inferred type; no deduction
  ```

  --- end example]

In 18 [language.support] paragraph 2, change

> The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, **support for initializer lists,** and other runtime support, as summarized in Table 16.

…and add 18.7 Initializer lists `<initializer_list>` to Table 16.

Add a new section after 18.7 [support.exception] and before 18.8 [support.runtime]:

18.8. **Initializer lists** [support.initlist]

The header `<initializer_list>` defines one type.

```
template<class E> class initializer_list {
public:
        initializer_list();

        size_t size() const;          // number of elements
        const E* begin() const;       // first element
        const E* end() const;         // one past the last element
};
```

An **initializer_list** provides access to an array of objects of type **const E**. [*Note:* A pair of pointers or a pointer plus a length would be obvious representations for **initializer_list**; **initializer_list** is used to implement initializer lists as specified in 8.5.4 [dcl.init.list]. Copying an initializer list does not copy the underlying elements. --- *end note*]

18.8.1 Initializer list **constructors** [support.initlist.cons]

initializer_list();

Effects: constructs an empty initializer list
Postconditions: **size() == 0**
Throws: nothing

18.8.2 Initializer list **access** [support.initlist.access]

const E* begin() const;

Returns: a pointer to the beginning of the array
Throws: nothing

const E* end() const;

Returns: **begin() + size()**
Throws: nothing

size_t size() const;

Returns: the number of elements in the array
Throws: nothing