

Initializer lists WP wording

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”. There are just two 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- The term “sequence constructor” has been replaced by “initializer-list constructor” to minimize the confusion with other kinds of sequences and lists.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.

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 expressionopt ;
return init-list ;
...

```

In 12.6.2 [class.base.init], change

```

mem-initializer:
mem-initializer-id ( expression-listopt )
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 initialized from the corresponding *initializer-clause* according to the initialization rules in 8.5 [dcl.init]. [Note: Therefore, 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 have no elements. [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
map<string,int> anim = { {"bear",4}, {"cassovary",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 an explicitly invoked constructor (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 specified in this standard, 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 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. `auto` shall not appear in the specifiers of such a type, and the type specified shall not be (possibly cv-qualified) `void`.

The *effective type* of an initializer list is its specified type, for a *typed-init-list*. An *untyped-init-list* has an effective type only if it contains no nested *untyped-init-lists*, 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`. In that case, the effective type of the initializer list is `std::initializer_list<E>`. [Note: As described in 14.8.2.1 [temp.deduct.call], the effective 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]).
2. Otherwise, if T is a class type,
 - if T has an initializer-list constructor taking the effective type of the initializer list, construct the `initializer_list` object (as described below) and call that initializer-list constructor;
 - otherwise, if T has a unique initializer-list constructor taking `initializer_list<E>`, and the initializer list is an *untyped-init-list*, and every element of the initializer list can be converted to E, [Footnote: This is vacuously satisfied if the initializer list has no elements.] construct the `initializer_list` object and call that initializer-list constructor;
 - otherwise, do direct-initialization using the elements of the initializer list as arguments.
3. Otherwise, if T is a reference type, do list-initialization of a temporary of the type referenced by T, and bind the reference to that temporary.
4. Otherwise (i.e., if T is not an aggregate, class type, or reference)
 - if the initializer list has a single element, do copy-initialization from it;
 - if the initializer list has no elements, do value-initialization of the object;
 - otherwise, the program is ill-formed.

[Example:

```

struct A { int i; int j; } a = { 1, 2 }; // aggregate initialization
struct B {
    B(std::initializer_list<int>);
};
B b { 1, 2 }; // creates initializer_list<int> and calls constructor
B c { 1, 2.0 }; // creates initializer_list<int> and calls constructor
struct C {
    C(int i, int j);
} c = { 1, 2 }; // calls constructor with arguments (1, 2)
int j { 1 }; // initialize to 1
int k {}; // initialize to 0

```

--- end example]

When an initializer list is implicit 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 will be initialized with corresponding element of the initializer list converted to `E`. The `begin()` and `end()` for that `initializer_list<E>` will refer to the first and one-past-the-last elements of the array, and the `size()` will be the number of elements.

[Example:

```
void f(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<cplx> v1 = { 1, 2, 3 };
vector<cplx> v2(3,cmplx(1));

void g(const vector<cmplx>&);

void f(int a)
{
    vector<cplx> v3= { 1, 2, 3 };
    vector<cplx> v4(3,cmplx(1));

    g({ 1, 2, 3 });
    g(vector<cplx>(3,cmplx(1)));
}
```

In each case, `{ 1, 2, 3 }` has the same lifetime as `vector<cplx>(3,cmplx(1))`. That is, static, local scope, and call, respectively. --- *end example*]

A program that, in a list-initialization, attempts to convert

- from a floating-point type to a type that is not a floating-point type
- from `long double` to `double` or from `double` to `float`
- from an integer type to another integer type with lesser integer conversion rank (4.13 [conv.rank])

or any combination of these conversions, except in cases where the initializer 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, is ill-formed. [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 c3{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};        // ok: value can be converted exactly
--- 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[](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 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].

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 $\mathbf{x}=\{\mathbf{v}\}$, where T is the scalar type of the expression $\mathbf{x}$, is that of $\mathbf{x}=\mathbf{T}(\mathbf{v})$. The meaning of $\mathbf{x}=\{\}$ is $\mathbf{x}=\mathbf{T}()$. If the initializer list is a typed-init-list, its type shall be T.
- 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.** 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 effective 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 effective type the deduction will fail).** 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 [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]***. ~~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 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<T>` or reference to `const std::initializer_list<T>`,

- if the effective type (8.5.4 [dcl.init.list]) of the initializer list is `std::initializer_list<T>`, the implicit conversion sequence is the identity conversion;
- otherwise, if the parameter under consideration is the parameter of a initializer-list constructor (8.5.4 [dcl.init.list]) that is the only initializer-list constructor in its class, and the initializer list is an *untyped-init-list*, and all the elements of the initializer list can be converted to T, the implicit conversion sequence is the identity conversion.

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

- if T is not an aggregate (8.5.1 [dcl.init.aggr]), if there is a single viable constructor of T when the elements of the initializer list are considered as arguments, the implicit conversion sequence is a user-defined conversion sequence;
- if T is an aggregate, then if each element of the initializer list [*Footnote:* There might be zero elements.] can be converted to the type of the corresponding initializable member of T and there are no more initializers than there are initializable members, the implicit conversion sequence is a user-defined conversion sequence.

Otherwise, if the parameter has reference to `const` type, the implicit conversion sequence is the one required to convert the initializer list to the underlying type of the reference according to this section. However, reference-to-array parameters are not considered to match an initializer list.

Otherwise, if the parameter type is `cv T` (with T not a class or reference type because of the previous rules), and the argument is a *typed-init-list* with type T or an *untyped-init-list*,

- if the initializer list has one element, the implicit conversion sequence is one the required to convert the element to the parameter type, if such a conversion is possible;
- if the initializer list has no elements, the implicit conversion sequence is the identity conversion.

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

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 effective type is used for A; if the initializer list does not have an effective type, the associated parameter is considered a non-deduced context (14.8.2.5 [temp.deduct.type]).** 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 effective type.

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 {
    // representation implementation defined
    // (probably two pointers or a pointer and a size)

    // implementation defined constructor
public:
    // default copy construction and copy assignment
    // no default constructor
    // default trivial destructor
```

```

    constexpr int size() const;    // number of elements
    const T* begin() const;        // first element
    const T* end() const;          // one-past-the-last element
};

```

An **initializer_list** provides read-only access to an array of elements of type **E**. **E** must be a value type (ref ???). The representation of an **initializer_list** is implementation defined. [*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 the core language (8.5.4 [dcl.init.list]). -- *- end note*]

Effects: **insert(p, s.begin(), s.end());**