N2640=08-0150
2008-05-16
Jason Merrill (jason@redhat.com), Daveed Vandevoorde (daveed@edg.com)

Initializer Lists — Alternative Mechanism and Rationale (v. 2)

Introduction

N2532 ("Uniform initialization design choices" by Bjarne Stroustrup) and N2531 ("Initializer lists WP wording" by J. Stephen Adamczyk, Gabriel Dos Reis, and Bjarne Stroustrup) propose to extend the use of "brace-enclosed initializer lists" ("{}-lists") to more contexts. See N2532 for a rationale.

The general idea of "initializer lists" (as discussed for many years in EWG) is to allow the use of a brace-enclosed list of expressions in all contexts that allow initializers. The following list is lifted from N2532:

- Variable initialization; e.g., X x {v};
- Initialization of a temporary; e.g, X{v}
- Explicit type conversion; e.g. x = X{v};
- Free store allocation; e.g. p = new X{v}
- Return value; e.g., X f() { /* … */ return {v}; }
- Argument passing; e.g., void f(X); /* … */ f({v});
- Base initialization; e.g., Y::Y(v) : X{v} { /* … */ };
- Member initialization; e.g., Y::Y(v) : mx{v} { X mx; /* … */ };

As soon as EWG contemplated this uniform syntactic vehicle, the desire to do away with a "copy-initialization" vs. "direct-initialization" distinction for brace-enclosed initializers arose.

The design evolved mostly uncontroversially over a few years, and culminated in the formal proposal referenced above (N2531+N2532). Parts of that proposal turned out to be controversial when discussed at the Bellevue meeting in February 2008, and the net result is that EWG voted for a significantly reduced proposal (see below for a summary).

This paper argues for a different approach to initializer lists to (a) recover desirable use cases, and (b) better address the concerns that have been raised.

(Note that this is not a proposal for alternative syntax, but for alternative mechanisms that make the already proposed design work without triggering behavior that raised concerns.)

Concerns about N2532+N2531

At the February 2008 meeting of WG21 and J16 in Bellevue, some elements of N2531+N2532 gave rise to significant debate, with some of the major concerns voiced being (in no particular order):

(a) "Explicit constructors" may be called for implicit conversions, e.g.:

```
struct Array {
  explicit Array(int);
  // ...
};
void print(Array);     // (1)
struct Number {
  Number(long long);
};
void print(Number);    // (2)
print({100});  // Prefers (1); i.e., constructs a temporary Array
```

This particular example would not remain a problem if **`Array`** evolved to include an initializer_list constructor, which is both natural and likely. However, initializer_lists will not make sense for other types that run into the same issue.

(b) User-defined conversions may silently be preferred over more common standard (but narrowing) conversions during overload resolution

```
struct BCD {
  BCD(int val);
};
void f(char);  // (1)  (assume 8-bit char)
void f(BCD);   // (2)
f({1000});     // Would call (2)
```

Selecting such a user-defined conversion would be surprising when its argument is not a representation of a value to be copied or converted, but an argument to an algorithm for constructing an object (in this example a size).

The net effect of the EWG discussions of Bellevue is that the rules for narrowing conversions were reluctantly agreed upon, but the ability to pass (untyped) initializer lists as function call arguments was voted against. Our sense is that no-one liked the implied compromise (at best).

We also find this an unfortunate compromise because (among other things):

- it diminishes the occurrence of situations that involve the concerns listed above, but it doesn't eliminate them altogether, and
- it takes away a highly convenient notational device.

It also results in a strange inconsistency that

```
A x = <expr>;
```

can be written as

```
A x(<expr>);
```

except if **`<expr>`** is of the form **`{`** ... **`}`**.

To illustrate the convenience of the notational device, we offer the following example:

```
struct Str {
  Str(char const*);
};

template<class T, class U>
struct HashEntry {
  HashEntry(T, U);
};

template<class T, class U>
struct HashTable {
  insert_entries(std::initializer_list<HashEntry<T, U>>);
};
HashTable h;
```

It would be really nice to be able to write:

```
h.insert_entries({ { "U.S.A.", "Washington D.C." },
                   { "Belgium", "Brussels" },
                   { "Guatemala", "Guatemala City" } });
```

If we must annotate the "type" on the list (or lists) in this last call, the code quickly becomes cluttered beyond recognition.

A programmer's view of initialization kinds

Some of the concerns raised are rooted in the fact that the C++ standard distinguishes between two kinds of initializations: Direct-initialization and copy-initialization. Those terms are "standardese" that we hope programmers should not concern themselves with. Indeed, part of the rationale in N2531 is that the {}-list notation could serve as an initialization notation in all contexts where initialization makes sense and with a uniform meaning (and the chosen meaning corresponds to "direct-initialization" in C++03).

We have observed that expert programmers who are aware of a difference between copy-initialization and direct-initialization frequently erroneously think that the former is less efficient than the latter. (In practice, when both initializations make sense, they are equally efficient.)

We find, in contrast, that it is more useful to think about these things in different terms:
- constructing by calling a constructor (a "ctor-call")
- constructing by transferring a value (a "conversion")

(As it happens, the former corresponds to "direct-initialization", and the latter to "copy-initialization", but the standard's terms don't help the programmer.)

The "ctor-call" looks like a call: It involves a parenthesized list of arguments for that call. The "conversion" involves the transfer of a "value", and so must be characterized by a unique source value.

A few examples:

```
struct Cmplx { Cmplx(double re, double im); ... };

Cmplx f() {
  Cmplx z1(1, 2);              // "Ctor-call".
  auto *p = new Cmplx(3, 4);   // "Ctor-call".
  Cmplx z2 = z1+*p;            // "Conversion" from value z1+*p.
  delete p;
  z1 = Cmplx(1, 2);            // "Ctor-call".
  return z2*z1;                // "Conversion" from value z2*z1.
}
```

and

```
struct String { explicit String(int capacity); };

String x(10);   // "Ctor-call"; string of capacity 10.
String y = x;   // "Conversion" from string value x to y.
String z = 10;  // Error: 10 isn't the "value" of a string.
```

With this view of things, a constructor should be declared "explicit" if its argument specifies not the "value" being constructed (such as the characters making up a string) but other properties of the construction (such as the capacity of a string object).

Core idea

Where the current (i.e., C++03) situation falls short is that there is is no compact way to create ad-hoc value compositions for "conversions". Instead, we have to cast such composite values in terms of "ctor-calls". E.g., if we wanted to create a small table of complex values, we may have to write

```
Cmplx zz[] = { Cmplx(0, 0), Cmplx(0, 1), Cmplx(1, 1) };
```

and most of the time things are uglier still because aggregate initializers have limited application. E.g., it works with built-in arrays, but not with `std::vector<Cmplx>`.

Consider the following list of "initializations" enabled by the design described in N2532 (which we also want to support):

```
(a) T v{x, y};                  // Variable construction
(b) T v = {x, y};               // Value initialization
(c) T{x, y}                     // Temporary construction
(d) new T{x, y}                 // New-expression
(e) f({x, y})                   // Argument binding
(f) return {x, y};              // Return statement
```

```
(g) D::D(...): B{x, y}, m{x, y} ...  // Base or member initializer
(h) T[] = { { x, y } };              // Sub-element initializer
```

We find that the `{x, y}` is most useful as a single value in constructs (b), (e), (f), and (h). In the other cases, we think it's most useful to treat `{x, y}` as an argument list for construction (and there is a fairly compact alternative syntax — e.g., `T{{x, y}}` — to force the single-value view).

I.e., we propose to introduce {}-lists as a device that groups values for the specific purpose of "conversion" to a class type in contexts (b), (e), (f) and (h). When the conversion is to a class type, it is always ranked as a "user-defined conversion" and is implemented by a non-explicit constructor (as in C++03). Unlike C++03, constructors with multiple parameters can now participate in user-defined conversions when the "source value" has a corresponding number of values of the appropriate type. For example:

```
struct Cmplx { Cmplx(double re, double im); ... };

Cmplx z = { 1, 2 };  // Okay: User-defined conversion from
                     // { int, int } to Cmplx (via
                     // Cmplx(double, double) constructor).
```

Additionally, the "initializer-list constructors" of N2531 are also converting constructors (when the element types work out and when those constructors are not declared "explicit"). E.g.:

```
template<class T> struct List {
  List(initializer_list<T>);
};

List<int> seq = { 1, 1, 2, 3, 5 };
   // Okay, user-defined conversion from { ... } to List<int>.
```

We view every set of braces as introducing a new initialization (hence the (h) view above). This means a chain of multiple user-defined conversion is possible, provided those conversions are "separated" by braces. E.g.:

```
struct Str {
  Str(char const*);
};
struct HashEntry {
  HashEntry(Str, Str);
};
struct HashTable {
  insert_entries(std::initializer_list<HashEntry>);
};
HashTable h;
h.insert_entries({ { "U.S.A.", "Washington D.C." },
                   { "Belgium", "Brussels" },
                   { "Guatemala", "Guatemala City" } });
```

In this last call, there are three levels of user-defined conversions:
(1) innermost level: string literal to **`Str`**
(2) middle level: **`{ Str, Str }`** to **`HashEntry`**
(3) outer level: **`{ HashEntry, ... }`** to **`HashTable`**

This is deemed acceptable because the programmer has clearly indicated — through braces — the expected extra number of user-defined implicit conversions.

The "implicit conversion" view means that explicit constructors are not considered when transferring the brace-composed values. E.g.:

```
struct Array {
  explicit Array(int size);
  // ...
};
void print(Array);     // (1)
struct Number {
  Number(long long);
};
void print(Number);    // (2)
print({100});          // Calls (2); (1) is not a candidate
                       // because it doesn't have a converting
                       // constructor.
```

(In standardese terms, it means that (b), (e), (f), and (h) are copy-initialization contexts... which is what they already were.)

{}-lists and narrowing

We propose that if — after deduction, overload resolution, etc. is done — a narrowing conversion (in the sense of N2531) occurs on a {}-enclosed value, the program is ill-formed. This is subtly different from N2531 in that it doesn't affect the existing overload resolution rules. For example:

```
struct Table {
  Table(int size);
};
void f(char);  // (1)  (assume 8-bit char)
void f(Table); // (2)
f({1000});     // Error under this proposal.
               // (Calls (2) with N2531.)
```

I.e., in situations like these, we do not try to guess what the programmer really meant: He or she must make it clearer (by specifying the intended type).

(Note: This example shows that the {}-list notation can be used for scalar values. The details of that aspect of the proposal — which matches N2531 — are discussed later in this paper.)

Typed lists and "the most vexing parse"

N2531 allows the following syntax:

```
return X{1, "Bye"};  // A "typed initializer list".
```

In discussions this has sometimes been called a "typed initializer list". It parallels the "function-style cast" notation (which itself is often thought of as a "ctor-call"), and is a form of direct-initialization.

We propose to retain that feature as proposed. Essentially, `X{...}` is equivalent to `X(...)`, except that

- initializer-list constructors (explicit or not) are also candidates for the "ctor-call", and
- the narrowing checks discussed above are performed.

Similarly, the following syntax is allowed:

```
X var{1, "Hello"};
```

Again, this is a form of direct-initialization and we propose to retain that aspect of it. It is particularly convenient to avoid running into what has been called "the most vexing parse":

```
struct A { A(); };
struct B { explicit B(A); };

B b(A());  // Vexing parse: b is a function declaration.
B c{A{}};  // No surprise: c is a variable constructed by
           // calling the B constructor with argument A{}.
B d(A{});  // A variable declaration: {} is never a declarator.
```

Note that since we treat the `{ ... }` in

```
X x = { ... };
```

as a single value, it is not equivalent to

```
X x{ ... };
```

where the `{ ... }` is an argument list for the constructor call (we emphasize it because it is unlike N2531). The former is a conversion (and therefore never calls an explicit constructor), while the latter is a constructor call (and therefore may call an explicit constructor). For example:

```
struct Array { explicit Array(int); };

Array z{10};     // Okay (presumably an array of 10 elements).
Array z = {10};  // Error in this proposal.  Same as
                 // "Array z(10);" with N2531.
```

{}-list conversion

Converting from a {}-list is always a user-defined conversion. Therefore, if overload resolution must choose between conversions to two distinct class types, the conversion is ambiguous. For example:

```
struct A { A(std::initializer_list<int>); };
void f(A);

struct B { B(int, int); };
void f(B);

f({1, 2});  // Error: Ambiguous conversion.
```

The fact that matching a {}-list always involves a user-defined conversion also addresses concerns raised in the past about the lack of visible types to guide a human reader of the code:

```
struct Duo {
  Duo(double, double);
};
struct HiLo {
  HiLo(Duo);
};
f(HiLo);
f({100, -20});  // Error: Cannot use two user-defined conversions
                //        to get {} -> Duo -> HiLo.
                // Use f(Duo{100, -20}) or f({{100, -20}}).
```

I.e., when a human reader tries to identify the target of the call `f({100, -20})`, she or he only need to examine the corresponding parameter types of every candidate `f` (this parallels C++03 for non-list arguments); there is no need to look for all types that could be reached with an additional conversion step.

If the destination type is unambiguous, but the class contains both an ordinary converting constructor and an initializer_list constructor, then the latter is preferred (assuming both are applicable):

```
struct A {
  A(initializer_list<int>);  // (1)
  A(int, int);               // (2)
};
void f(A);

f({1, 2});  // Uses (1).
```

We did look at reversing the preference so that a few multi-parameter constructors could efficiently handle short lists while the initializer-list constructor would handle arbitrarily long lists. However, that is an unfortunate choice for cases like the following:

```
struct Vec {
  Vec(std::initializer_list<int>);
  explicit Vec(unsigned size);
};
Vec v1{1, 2};  // Would construct a 2-element vector.
Vec v{10};     // ?? Would construct a 10-element vector
               // (not proposed).
```

So instead we stick with the simple rule that an initializer_list constructor is always preferred when given a {}-list.

For completeness' sake, the following example is worth mentioning:

```
struct A {
  A(initializer_list<double>);  // (1)
  A(int, double);               // (2)
};
void f(A);
f({ 1, 2.0 });  // Converts the list via constructor (1), even
                // though (2) is a "better match" in many ways.
```

When we first looked at this example, it seemed like the outcome that the initializer_list constructor is preferred was the lesser choice. However, after trying to find use cases for which the other constructor would be preferable, the classes we end up with were invariably designs we'd not be satisfied with. Indeed, the proposed ordering turns out to work as desired for all realistic designs we could come up with.

A few more examples illustrating the "user-defined conversion" nature of {}-list conversions:

```
struct A {
  A(int, int);  // Can convert {1, 2} -> A(1, 2)
};
struct B {
  B(A);  // Can convert A -> B
};
void f(B);
f({1, 2});  // Error: Cannot use two user-defined conversions
            //        {1, 2} -> A -> B
            // Use f({{1, 2}}) or f(A{1, 2}) or f(B({ 1, 2}))
            // instead.

struct C {
  C(int);
};
struct D {
  D(C, C);
};
g(D);
g({1, 2});
```

```
// Okay: int -> C for the initialization in the braces, and
//       {C, C} -> D for the initialization of the function
//       argument.
```

Finally, we note that the wording changes that implement the rules above are not only simpler than the alternative, but they naturally result in the following desirable behavior:

```
struct Z_List {
  Z_list(initializer_list<double>);   // (1)
  Z_list(initializer_list<Cmplex>);   // (2)
};
Z_list zs{ 1.0, 2.0, 3.0 };  // Now prefers (1).
```

Surprisingly, with N2531 this example turns out to be ambiguous.

Template argument deduction

First, it's worth mentioning that a converting constructor can be a constructor template. It might even be a variadic constructor template:

```
struct CodeList {
  template<class ... Ts> CodeList(Ts ... values);  // (1)
};

void f(CodeList); // (2)

f({1, 2.0});  // Matches (2) and converts {1, 2.0} to a CodeList
              // with constructor (1) and Ts deduced to
              // <int, double>.
```

(This is not fundamentally different from C++03 converting constructors.)

When performing template argument deduction of `std::initializer_list<T>` against a {}-list, deduction succeeds if each element of the list deduced against `T` produces the same deduction.

```
template<class T> size_t count(std::initializer_list<T>);
size_t n = count({1, 2, 3});    // Okay.
size_t m = count({1, 2, 3.0});  // Error.
```

If some elements don't produce a deduction at all, then deduction still succeeds if the other elements produce consistent deductions. For example:

```
template<class T> struct A {
  A(std::initializer_list<T>);
};
template<class T> void f(std::initializer_list<A<T>>);
A<int> a, b;
f({a, b, {1, 2, 3}});  // Okay: The first two elements deduce a
```

```
                              // consistent T, and the third element
                              // ({1, 2, 3} which doesn't produce a
                              // deduction) is compatible after
                              // substitution.
```

A {}-list cannot deduce against a plain type parameter `T`. For example:

```
template<class T> void count(T);  // (1).

struct Dimensions { Dimensions(int, int); };
size_t count(Dimensions);  // (2).

size_t n = count({1, 2});  // Calls (2); deduction doesn't
                           // succeed for (1).
```

Another example:

```
template<class T>
void inc(T, int);  // (1)

template<class T>
void inc(std::initializer_list<T>, long);  // (2)

inc({1, 2, 3}, 3);  // Calls (2).  (If deduction had succeeded
                    // for (1), (1) would have been called — a
                    // surprise.)
```

On the other hand, being able to deduce an `initializer_list<X>` for `T` is attractive to allow:

```
auto x = { 1, 1, 2, 3, 5 };
f(x);
g(x);
```

which was deemed desirable behavior since the very beginning of the EWG discussions about initializer lists.

Rather than coming up with a clever deduction rule for a parameter type `T` matched with a {}-list (an option we pursued in earlier sketches and drafts of this paper), we now prefer to handle this with a special case for "auto" variable deduction when the initializer is a {}-list. I.e., for the specific case of a variable declared with an "auto" type specifier and a {}-list initializer, the "auto" is deduced as for a function `f(initializer_list<T>)` instead of as for a function `f(T)`.

Aggregate initialization quirks

Aggregate initialization is the mechanism that allows initialization with brace-enclosed lists in C++03, and dates back from C. It applies only to types that don't have user-defined constructors (i.e., cases not covered by the {}-list conversions described above) and has some

properties that we do not try to carry over to the {}-lists used for conversion constructors. For example, consider:

```
struct I { I(); I(int); }; // Non-aggregate.
struct N { I n; };         // Aggregate.
struct A { N n[3]; };      // Aggregate.
```

Aggregate initialization can leave out braces and trailing elements:

```
A a1 = { 1, 2 };  // Okay.
```

The latter is equivalent to the following more fully-specified construct:

```
A a1 = { { 1 }, { 2 }, { I() } };
```

The following is invalid C++03, but would become valid under our proposal:

```
A a2 = { { { 1 },  { 2 }, {} } }
       ^ ^ ^-- New-style initializer
       | `---- Aggregate initializer for struct N
       `------ Aggregate initializer for struct A
```

This proposal allows {}-lists for non-aggregates but for those cases one cannot leave out brace levels. Trailing elements can only be omitted if default arguments are provided for the corresponding conversion constructor parameter. For example:

```
struct L {
  L(int, int, int = 0);
};
L v1 = { 1, 2, 3 };  // Okay.
L v2 = { 1, 2 };     // Okay, converts by calling
                     // L::L(1, 2, 0).
L v3 = { 1 };        // Error (this is not aggregate
                     // initialization).
```

N2531 allows leaving out the "=" in aggregate initialization, with no change in constraints and/or semantics when compared to C++03 aggregate initialization. This is consistent with the "uniformity" design principle that guides N2531+N2532.

Our alternative proposal, however, does not maintain semantics between "X x = { .... };" (value transfer) and "X x{ ... };" (ctor-call) for non-aggregates. We therefore propose to require all braces to be specified when leaving out the "=" in aggregate initialization (so the structure is clear), but we do allow trailing elements to be omitted if they can be default-initialized (in the value-initialization sense). I.e., with the definition of `A` as above, the following is invalid:

```
A a3{ 1, 2 };  // Error: Missing braces.
```

and (more importantly, in our opinion):

```
f(A{ 1, 2 });  // Error: Missing braces.
```

Another example:

```
struct S { int a, int b[3]; int c};
S s1 = { 1, 2, 3, 4, 5 };  // Sloppy old style (missing braces).
S s2 = { 1, 2 };           // Sloppy old style (missing braces).
S s4{ 1, { 2, 3, 4 }, 5 }; // Proper new style.
S s5{ 1, {2} };            // Proper new style, leaving out
                           // trailing elements; means
                           // { 1, { 2, 0, 0}, 0 }.
```

Note that for aggregates there is no "value transfer" vs. "ctor-call" distinction, since there is no constructor to call. Instead, whether or not the "=" token is present, construction occurs by transferring the values of the individual members. (Those element transfers correspond to copy-initialization in standardese terms, but are akin to the copy-initialization of the arguments of the constructor call that occur when performing direct-initialization on a type with a constructor.)

Initialization of scalar types

Although so far we've almost exclusively addressed the use of {}-lists to specify values for classes (with constructors, or aggregates) and arrays (a different kind of aggregate), the notational devices for those cases must be extended to other types to make generic programming practical. For example:

```
template<class T> void f(T x) {
  T y{x};  // T might end up being "int" or "double*" and this
}          // should remain valid.
```

I.e., a single-element {}-list is a valid initializer for a non-class, non-aggregate type. The brace-enclosed value behaves as if it had been enclosed by parentheses instead, except that the check for narrowing is done.

A remaining concern

Our proposed approach to enabling wider use of brace-enclosed lists of expressions turns an existing class of constructors into vehicles for "implicit conversion". Although this proposal does allow disabling such conversions using the keyword "explicit" (and that's different from N2531+N2532), existing code is unlikely to have annotated multi-parameter constructors with "explicit". This could lead to surprises in code that uses the new {}-list syntax. For example:

```
struct Matrix {
  Matrix(int rows, int cols);
};
Matrix diagonal(Matrix);
Matrix test = diagonal({1, 3});  // Probably not the expected
                                 // behavior.
```

We do not attempt to address this concern because the solutions that we can think of (annotating constructors to indicate that they can be used for {}-list conversion) will burden the language with a device that will likely quickly become obsolete as C++0x is adopted. We'd be inclined to impose that burden if existing code were affected, but only new {}-list constructs can trigger the issue.

As proposed, every C++0x programmer will have to balance the desire for convenient syntax vs. the desire to avoid silent surprises. E.g., one could imagine transitional coding guides that only allow passing {}-lists to calls of member functions (where the overload set cannot be influenced by #include'd headers, etc.).

The Meaning of Explicit

In C++03, the specifier "explicit" on a constructor means that that constructor is not considered at all when performing a conversion ("value transfer", aka. "copy-initialization). For example:

```
struct X {
  explicit X(int);  // (1)
  X(double);        // (2)
};
X x1(10);      // Calls (1).
X x2 = 10;     // Calls (2).
```

With brace notation, we can add:

```
X x3{10};      // Calls (1).
X x4 = {10};   // Calls (2).
```

In N2531+N2532, the latter two initializations are equivalent: Both call the first constructor. Depending on expectations, either one of the resolutions may seem surprising. That can be an argument to deem examples like these ill-formed in some way.

A first approach to make the example ill-formed is to require that all constructors (explicit and non-explicit) are considered for implicit conversions, but if an explicit constructor ends up being selected, that program is ill-formed. This rule may introduce its own surprises; for example:

```
struct Matrix {
  explicit Matrix(int n, int n);
};
Matrix transpose(Matrix);

struct Pixel {
  Pixel(int row, int col);
};
Pixel transpose(Pixel);
```

```
Pixel p = transpose({x, y}); // Error.
```

A second approach is to ignore the explicit constructors when looking for the viability of an implicit conversion, but to include them when actually selecting the converting constructor: If an explicit constructor ends up being selected, the program is ill-formed. This alternative approach allows the last (Pixel-vs-Matrix) example to work as expected (`transpose(Pixel)` is selected), while making the original example ("`X x4 = { 10 };`") ill-formed.

We propose to follow that second approach. However, out of concern for backward compatibility, the associated rule only applies to conversions of {}-list initializers.

Conclusions

Our proposal attempts to match N2531+N2532 in allowing an elegant notation for "composite values" to be used with user-defined types. Our primary goals here are the same: We want to diminish the advantage of "built-in types" over "user-defined" types.

We do approach the problem with different "guiding principles":

N2531+N2532: "brace-notated initialization should use a uniform initialization mechanism"
Our proposal: "ctor-call and value-transfer benefit from distinct treatments"

We think the programming model resulting from this alternative is both teachable and manageable. Our programming advice:

- If the parameters of a constructor do not characterize the "value" of an instance of the type, the constructor should be "explicit". E.g., the length of a string does not characterize its value and so a string constructor taking a length should be "explicit". On the other hand, when constructing a string object from a string literal, the literal does characterize the string value, and hence a string constructor taking a "char const*" can be non-explicit.
- When a constructor looks like a call, interpret it as a constructor call: Any constructor of the given type is a candidate. Otherwise, interpret it as a value conversion: Only non-explicit constructors and conversion operators are candidates.
- For constructor calls, the parentheses can be replaced by braces. This has two advantages: (a) you'll avoid a surprising parsing rule, and (b) you'll catch accidental narrowing conversions.

WP Changes

(The following are the changes needed to implement our proposal. These changes are based on the Changes proposed in N2531.)

In 8.5 [dcl.init], change

initializer:
  = *initializer-clause*
  (*expression-list*)
  init-list

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

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

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

In 5.2 [expr.post], change

postfix-expression:
  ...
  postfix-expression [*expression*]
  postfix-expression **[** ***init-list*** **]**
  postfix-expression (*expression-list*$_{opt}$)
  simple-type-specifier (*expression-list*$_{opt}$)
  typename-specifier (*expression-list*$_{opt}$)
  simple-type-specifier init-list
  typename-specifier 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 6.4 [stmt.select], change

condition:
  expression
  type-specifier-seq declarator =
                          ~~*assignment-expression*~~*initializer-clause*
  type-specifier-seq declarator init-list ;

If the `auto` *type-specifier* appears in the *type-specifier-seq*, the *type-specifier-seq* shall contain no other *type-specifiers* except *cv-qualifiers*, and the type of the identifier being declared is deduced from the ~~*assignment-expression*~~**initializer** as described in 7.1.6.4.

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], 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]).

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 ~~brace-enclosed~~**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 *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.* If the *initializer-clause* is an expression, the corresponding member is copy-initialized; if a narrowing conversion ([dcl.init.list]) is required to convert the expression, the program is ill-formed. If an *initializer-clause* is itself an initializer list, the member is list-initialized. [*Note:* 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 *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. [*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 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])

[*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*] — *end note*]

An *initializer-list constructor* is a constructor taking a single argument of type `std::initializer_list<E>` or `const std::initializer_list<E>&` for some type `E`. [*Note:* Initializer-list constructors are favored over other constructors in list-initialization ([over.match.list]).] The template `std::initializer_list` is not predefined; if the header <`initializer_list`> is not included prior to a use of `std::initializer_list` (even an implicit use in which the type is not named), the program is ill-formed. In certain contexts (see below, [dcl.spec.auto]), an initializer list can be implicitly converted to an `std::initializer_list<E>` object that points to an array containing the elements of the initializer list.

An *init-list* is said to have an inferred type 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 can itself be an inferred initializer list type. --*end note*] In that case, the inferred type of the initializer list is `std::initializer_list<E>`. [*Note:* As described in 7.1.5.4 [dcl.spec.auto], the inferred type of an initializer list is used for `auto` deduction. [*Example:*

```
auto x = {1,2,3}; // decltype(x) is initializer_list<int>
```

— *end example*]]

List-initialization of an object or reference of type T is defined as follows.

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, call a constructor for T using the initializer list or its elements as arguments. The best constructor of T is chosen by overload resolution according to [over.match.list]. 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. If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.
[*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 }; // invoke #2
```

— end example]
[*Example:*

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

— end example]
[*Example:*

```
struct S {
  // 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:* As usual, 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, initialize the object from that element; 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*]

5. 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:*

```
struct X {
  X(std::initializer_list<double> v);
};
X x{ 1,2,3 };
```

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

```
double __a[3] = {double{1}, double{2}, double{3}};
X x(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 array is the same as that of the `initializer_list` object. [*Example:*

```
typedef std::complex<double> cmplx;
vector<cmplx> v1 = { 1, 2, 3 };

void f()
{
  vector<cmplx> v2{ 1, 2, 3 };
  initializer_list<int> i3 = { 1, 2, 3 };
}
```

For `v1` and `v2`, the `initializer_list` object and array created for `{ 1, 2, 3 }` have full-expression lifetime. For `i3`, the `initializer_list` object and array have automatic 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 5.2.3 [expr.type.conv], change:

A *simple-type-specifier* (7.1.6.**2**) **or *typename-specifier*** **(14.6)** followed by a parenthesized *expression-list* constructs a value of the specified type given the expression list. If the

expression list is a single expression, the type conversion expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast expression (5.4). If the ~~*simple-type-specifier* specifies~~**type specified is** a class type, the class type shall be complete. If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (8.5, 12.1), and the expression `T(x1, x2, ...)` is equivalent in effect to the declaration `T t(x1, x2, ...);` for some invented temporary variable `t`, with the result being the value of `t` as an rvalue.

The expression `T()`, where `T` is a *simple-type-specifier* ~~(7.1.6.2)~~ **or typename-specifier** for a non-array complete object type or the (possibly cv-qualified) void type, creates an rvalue of the specified type, which is value-initialized (8.5; no initialization is done for the `void()` case). [*Note:* if `T` is a non-class type that is cv-qualified, the cv-qualifiers are ignored when determining the type of the resulting rvalue (3.10). --- *end note*]

Similarly, a *simple-type-specifier* or *typename-specifier* followed by an *init-list* creates a temporary object of the specified type list-initialized (8.5.4) with the specified *init-list*, and its value is that temporary object as an rvalue.

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] 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()**.

- 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 7.1.6.4 [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]); 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. --- *end note*] 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 12.3.1 [class.conv.ctor] paragraph 1, change

A constructor declared without the *function-specifier* `explicit` ~~that can be called with a single parameter~~ specifies a conversion from the ~~type of its first parameter~~**types of its parameters** to the type of its class. Such a constructor is called a *converting constructor*.

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 as follows:

- First, 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>`, for some X, are candidate functions. The argument list is a single rvalue object of type `std::initializer_list<X>`.

- If no viable initializer-list constructors are found, the constructors of T that are not initializer-list constructors are candidates. The argument list is the elements of the initializer list.

For copy-initialization, in both steps only converting constructors are candidates. However, if there is an `explicit` constructor in the same overload set which would have been a better match than the chosen constructor, the initialization is ill-formed.

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>`, and all the elements of the initializer list can be implicitly converted to X, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to X. [*Example:*

```
void f(std::initializer_list<int>);
f( {1,2,3} ); // ok: f(initializer_list<int>) identity conversion
f( {'a','b'} ); // ok: f(initializer_list<int>) integral promotion
f( {1.0} ); // error: narrowing

struct A {
  A(std::initializer_list<double>); // #1
  A(std::initializer_list<complex<double>>); // #2
};
A a{ 1.0,2.0 }; // ok, uses #1
```

— end example]

Otherwise, if the parameter type is *cv* X or reference to `const` X where X is a class type,

- if X is not an aggregate (8.5.1 [dcl.init.aggr]), if overload resolution per 13.3.1.7 [over.match.list] chooses 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(std::initializer_list<int>);
  };

  void f(A);
  f( {'a', 'b'} ); // ok: f(A(initializer<int>)) user-defined
conversion

  struct B {
    A(int, double);
  };

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

  void f(B);
  f( {'a', 'b'} ); // error: ambiguous f(A) or f(B)
```

— end example]

If multiple constructors are viable but none is better than the others, the implicit conversion sequence is the ambiguous conversion sequence.

- 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]) with no elided braces, 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,

- if the initializer list has one element, the implicit conversion sequence is the one required to convert the element to the parameter type; [*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 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], add a new paragraph:

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 P is `std::initializer_list<P'>` or reference to `const std::initializer_list<P'>` for some P' and the argument is an initializer list (8.5.4 [dcl.init.list]), the type P' is compared with each element of the initializer list in turn. Otherwise, an initializer list argument is considered a non-deduced context (14.8.2.5) [*Example:***

```
template<class T> void f(std::initializer_list<T>); // #1
f({1,2,3}); // T deduced to int
```

```
f({1,"asdf"}); // error: T deduced to both int and const char*

template<class T> void g(T);
g({1,2,3}); // error: non-deduced context
```

--- 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]) but the parameter does not have `std::initializer_list` type.
  [*Example:*
  ```
  template<class T> void g(T);

  g({1,2,3}); // error: non-deduced context
  ```
  — end example]

Acknowledgments

Thanks to Steve Adamczyk, Gabriel Dos Reis, and Bjarne Stroustrup, whose paper N2531 this proposal is based on.