Doc No: N2477=07-0347
Date: 2007-11-28
Project: JTC1.22.32
Reply to: Bjarne Stroustrup
bs@cs.tamu.edu

Uniform initialization design choices

Bjarne Stroustrup
bs@cs.tamu.edu

Abstract

Here, I outline the main potentially difficult and potentially controversial design choices in the initializer list and uniform initialization design. The claim is that the decisions are strongly guided by the ideal or syntactic and semantic uniformity and consider the advantages and disadvantages of departing from that ideal in particular cases. My main concern is whether the compatibility with classical aggregate initialization is sufficiently good. A secondary concern (which will be major to some) is whether explicit constructors will be called in ways that would become too surprising.

The aim of this note is to help build consensus and understanding using concrete examples before going back to revise the WP wording. I hope that the discussion of problems and alternatives will not distract from the main benefits from the proposal: uniformity, conciseness, and general use of initializer lists.

The complete design can be found in N2215 and draft WP wording (found not to completely match the design) in N2385 and D2441.

Differences from N2215

The design described here differs in a few ways from the one described in N2215:

- The structure of an initializer list is more precisely described
  - Application to aggregates
  - The initializer of a class with an initializer list constructor must be an initializer list
- Narrowing is defined as agreed by EWG and CWG (all conversions from floating-point to integer types are considered narrowing)
- The overload and deduction rules are more precisely defined
  - **initializer_list<T>** is never deduced, but the **T** in **initializer_list<T>** may be

These differences are primarily to address comments at Kona and results of a more precise statement of some parts of the design.

This document assumes the non-narrowing rule (because of the EWG votes), whereas the "main proposal" part of N2215 didn't (because those votes had yet to happen).

1 Ideal

A fundamental aim of "uniform initialization" can be expressed as: "We want a single syntax that can be used everywhere and wherever it's used, the same initialized value results for the same initializer value". It may be that the ideal can be only approximated for C++0x, but I argue that it is the ideal.

I will assume that the uniform syntax is the use of {…} throughout. Exactly as for aggregate initializers, the … in {…} is a (possibly empty) comma-separated list of values to be used in initialization. If the object to be initialized requires a hierarchical organization of initializers, then that organization must be reflected in the initializer list through nested initializer lists.

The simplest version of the rules can be summarized as:

- {…} can be used for every initialization
- {…} implies direct initialization
- {…} implies no narrowing

By "a single syntax that can be used everywhere" I mean a syntax for initializers to be uses in all contexts where an object is initialized with an explicit initializer:

- Variable initialization; e.g., **X x {v};**
- Initialization of temporary; e.g, **X{v}**
- Casting; 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; /*** … ***/ };**

In every case, the value of the constructed/initialized/passed object is the same.

The types for which objects can be initialized in that way (**X** in the examples above) are

- Individual variables
- Aggregates (both **struct**s and arrays)
- Classes with constructors

- Classes with the new **std::initializer_list** constructors taking homogeneous variable length lists of elements

That is, all types.

An example of a violation of the ideal of uniform initialization would be a type **X** and a value **v** such that two initializations gave different values (assuming that copying and equality have their proper meanings):

```
X f(X a) { return {a}; }
X x{v};
If (x!=f({v})) cout << "violation\n";
```

Another example would be a type **T** and a value **v** such that two initializations would have to be done using different syntaxes:

```
void g(X);
X x{v};        // ok
g({v});        // error (use an alternative notation to pass v)
```

Note that this last case will naturally arise when we detect ambiguities, but we want to avoid it where there is only one possible function to call.

Similarly, we are forced to depart from the ideal that every initialization can be done using {…} if a class needs to have both an initializer-list constructor and an ordinary constructor that can take an argument list where all arguments are of the same type as the list elements. For example:

```
class X {
public:
        X(std::initializer_list<V>);
        X(V,V);
        // ...
};

X a{V{1}, V{2}};      // initializer list constructor: V{1} and V{2} are elements
X b(V{1},V{2});       // ordinary constructor: V{1} and V{2} are arguments
```

The standard **vector<int>** is an example. See Appendix B of N2215 for an explanation.

I explore the issues under four headings:

- Depth
- Directness
- Narrowing
- Overloading

Please note that the resolutions of individual issues must all match for the uniformity ideal to be successfully approximated. This is at least a four-dimensional puzzle and it may not be possible to adopt everyone's ideals for individual examples within a complete framework.

Please also note that even though there are many tricky examples in this paper, they are all resolved by just a few simple rules, such as "initializer list constructors take priority over all other constructors".

2 Depth

How does nesting affect initialization? Within an initializer, we use {…} to express grouping; for example **{{"dog", 10}, {"cat", 77}}**. This is (not accidentally) rather similar to aggregate initializer lists. For aggregates, we have to also accept **{"dog", 10, "cat", 77}** even where there is a grouping, such as for an array of **struct**s. This defeats ideas of completely enforcing expression of grouping in initializers.

2.1 The basic scheme

Here are the basic rules:

1. A {…} initializer is a possibly empty list enclosed in curly braces, such as **{}**, **{1}**, and **{1,2,3}**.
2. Every object can be initialized by a {…} initializer.
3. The elements of the {…} initializer must match the structure of the type of the object being initialized; for example, and **int** can be initialized by **{ 1 }** but not by **{1,2}**, a **struct** with two **int**s can be initialized by **{1,2}**, but not by **{1,2,3}**, and an object with a constructor requiring two **int**s can be initialized by **{1,2}**, but not by **{1}**.
4. An element that is to be used to initialize an aggregate, a class with a constructor taking more than one argument, a class taking an object of type **initializer_list** (or a **const** reference to such) must be a {…} list, an **initializer_list**, or an object of the element's type; for example a **vector**<**vector<int**>> can be initialized by **{{0}, {1}, {1,2,3}}** or **{v1,{2},ilst}**, where **v1** is a **vector<int>** and **ilist** is an **initializer_list<int>**, but not **{ 0, 1, 1, 2, 3 }**.
5. No redundant **{}**s are allowed in {…} initializers; for example an **int** cannot be inititalized by **{{1}}**, a **struct** with two **int**s cannot be initialized by **{{1,2}}** or **{{1},{2}}**, and a **vector**<**int>** cannot be initialized by **{{1,2,3}}**.

The cases of missing elements in an aggregate initializer and of missing indication of nested structure in initializers of nested aggregates are dealt with as special cases for the sake of C and C++98 compatibility. My impression is that this degree of compatibility is essential (so that compatibility wart is part of the proposal).

The rest of this section is devoted to exploring the implication of these rules. The discussion of type issues is postponed to subsequent sections; this section is devoted to issues of syntactic structure in general and to present the reasons for the last two rules in particular.

2.2 Lists vs. scalars

So, we use {…} to indicate an initializer list. For example:

```
X a0{};        // default initialization
X a1{1};       // initialization with one value: 1
X a2{1,2};     // initialization with two values: 1, 2
```

If, for some reason, an initializer that's a list has to be part of an initializer list, we indicate that by a {…} sub-list. For example:

```
X a3{{1,2},{3,4}};     // initialization with two list values: {1,2} and {3,4}
X a4{{1},{2}};         // initialization with two list values: {1} and {2}
X a5{{ },{ }};         // initialization with two list values: {} and {}
```

Most confusion arise from the degenerate cases, such as **a2** versus **a4**. Note that a list initializes a single object, argument, etc., and that the type of that object, parameter, etc. determines whether a list or a single (non-list) value is acceptable, what type is expected, and how that initializer is used (as an initializer, as a constructor argument, to build an **initializer_list**). In this context, I'll call something that is initialized with a single (non-list) value a scalar. For example:

```
struct S1 { int x,y; };
struct S2 { S2(int,int); /* … */ };
struct S3 { S3(initializer_list<int>); /* … */ };
struct S4 { S1 z; };

int x0{1,2};     // error: you can't initialize a scalar with a list

S1 x1{1,2};      // x1.x=1 and x1.y=2
S2 x2{1,2};      // S1 x1(1,2);
S3 x3{1,2};      // x3 is constructed with an initializer_list with elements 1 and 2
S4 x4{1,2};      // error: cannot initialize an S1 with a 1

S1 x11{{1,2}};// error: can't initialize a scalar (x11.x) with a list
S2 x22{{1,2}};// error : S2(int,int) takes two arguments, not a single list
S3 x33{{1,2}};// error: S3(initializer_list) takes a list of ints,
              // not a list with its first element a list of ints.
S4 x44{{1,2}};// ok: x44.z is initializer by {1,2}
```

Let's try that example with just one **int** instead of two:

```
struct S1 { int x; };
struct S2 { S2(int); /* … */ };
struct S3 { S3(initializer_list<int>); /* … */ };
struct S4 { S1 z; };

int x0{1};      // ok: x0=1

S1 x1{1};       // x1.x=1
S2 x2{1};       // S1 x1(1);
S3 x3{1};       // x3 is constructed with an initializer_list with elements 1
S4 x4{1};       // ok (should have been an error, but accepted for compatibility)

S1 x11{{1}};    // error: can't initialize a scalar (x11.x) with a list
S2 x22{{1}};    // error : S2(int) takes a (scalar) arguments, not a list
S3 x33{{1}};    // error: S3(initializer_list) takes a list of ints,
                // not a list with its first element a list of ints.
S4 x44{{1}};    // ok: x44.z initialized by {1}
```

So, initializer lists with just one element ("degenerated to a single element") don't differ from longer lists except in the essential case where the number of elements itself is important.

The **x4** example is bothersome. My guess is that accepting this for compatibility is necessary, but will cause some confusion and errors. Here is one real-world case that argues for the compatibility hack:

```
struct S { int elem[6]; }
S s = { 1, 2, 3, 4 };
```

This is the essential initialization mechanism for **std::array**.

2.3 Uniform syntax and semantics

The basic idea for dealing with an aggregate is to deal with it as if it had a constructor initializing each element. For example:

```
struct Sc1 { char c; };
struct Sc2 { char c; Sc2(char); };
struct Sc4 { Sc4(list_initializer<char>); /* … */ }

Sc1 sc1{'a'};   // ok initializes sc1.c
Sc2 sc2{'a'};   // ok: invokes Sc2('a')
Sc4 sc4{'a'};   // ok: creates an initializer_list<char> with one char

char c1 {'a'};  // ok – as ever, an initializer list with one element
                // can be used to initializer a scalar
```

```
char a1[] {'a'}; // ok, initializes a1[0]
```

According to the uniformity principle every **char** should be initialized with the same value (or they should all fail) for a given initializer. To explore the semantic implications, assume that {…} initialization (list initialization) never narrows (see Section 4 for details) and consider:

```
struct Sc1 { char c; };
struct Sc2 { char c; Sc2(char); };
struct Sc4 { Sc4(list_initializer<char>); /* … */ }

int i;
Sc1 sc1{i};     // error: narrowing? (yes: error)
Sc2 sc2{i};     // error: narrowing? (yes: error)
Sc4 sc4{i};     // error: narrowing? (yes: error)

char c1 { i };  // error: narrowing? (yes: error)
char a1[] { i }; // error: narrowing? (yes: error)
```

Each of these initializes a **char** with an **int**. According to the ideal of uniformity, each should result in the same **char** value or they should all be errors. We assumed "no narrowing", so they must all be errors.

Let's look a bit closer at the initialization of **cl** and **a1[0]**. they were initialized directly with **i**. For **sc4**, **i** was passed to an initializer list constructor where the underlying array is an array of **char**, so this initialization is exactly equivalent to that of **a1[0]**. However, we might consider that initialization of **sc1** and **sc2** "different":

- The construction of the **char** from the **int** for **sc2** is argument passing (for the constructor's argument)
- The construction of the **char** inside **S1** for **sc1** is member initialization

Should these be different from other forms of initialization? If so, why? For **sc1** (the **struct**), the obvious answer is "compatibility", but let's consider what the ideal answer would be before blindly adopting the old rules.

Ideally, a user of **Sc1** and/or **Sc2** shouldn't have to know how the initialization is achieved: by constructor or by member initialization. The reason that's the ideal is that it hides an implementation detail and allows implementers to change implementations as needed, e.g. to add a constructor to a **struct** to provide argument checking without rewriting user code. Anything else would also clash with the original view of C++ that argument passing is initialization and add complication (as is the case with current = and (…) initialization notations). It would be odd if the value of **cl[0]** differed from **sc1.c** because these two initializations have always (since the dawn of C) been equivalent. I conclude that all the initializations above ideally are errors and further that all equivalent examples must be legal and yield the same value if any is legal:

```
struct Sx1 { X c; };
struct Sx2 { X c; Sc2(X); };
struct Sx4 { Sc4(list_initializer<X>); /* … */ }

T i;
Sx1 sx1{i};
Sx2 sx2{i};
Sx4 sx4{i};

X x1 { i };
X x1[] { i };
```

For the mitigation of narrowing problems see Section 4. Obviously, **char c{'a'};** must be ok.

What if we do want implicit narrowing? This works as ever:

```
char x1 = i;    // ok (old rules)
char x2(i);     // ok (old rules)
```

That is, to get "traditional non-uniform semantics", we must use "traditional non-uniform syntax". Using {…} initializer, there is no way of getting the "classical (narrowing) aggregate initialization behavior" without using an explicit conversion:

```
Sc1 sc1 = { static_cast<char>(i) };
char c1[] = { static_cast<char>(i) };
```

Note that we do not propose to allow initialization of an array with an **initializer_list** object:

```
// Warning; not proposed:
char c1[] = list_initializer<char>{i};              // error
vector<char> v1 = list_initializer<char>{i};        // ok (unavoidable)
```

Allowing initialization of an array (or a plain old struct) with an **initializer_list** would be an extra special case of no particular utility. On the other hand, the general rule ensures that we can initialize one array with another:

```
char cc[]{"asd"};
char ccc[]{cc};         // ok: ccc is a copy of cc
```

As noted in N2215, this solves a longstanding problem with member initialization and it would take a special rule to outlaw it.

2.4 Alternatives

The most obvious alternative (bar leaving things alone and completely giving up on uniformity) is to leave aggregate initialization "special." We could leave all aggregates alone:

```
Sc1 sc11{i};    // ok (not list initialization)
char a1[]{i};   // ok (not list initialization)
```

This would leave initialization with aggregate different from all other list initialization. I note that EWG repeatedly rejected that, but I don't recall using **struct** as the example and that the most recent WP draft went this way (with the inconsistency unnoticed). Note that defaulting to (old-style and backwards compatible) decision would imply that members of aggregates would follow the old rules for explicit constructors, etc.

Alternative, we could consider the = significant:

```
Sc1 sc11 = {i};         // ok (not list initialization)
char a1[] = {i};        // ok (not list initialization)

Scl scll2 { i };        // error: narrowing? (yes: error)
char a12[] { i };       // error: narrowing? (yes: error)
```

I don't propose that. The = should not be semantically significant:

- A significant = would be a violation of the uniformity principle
- A significant = would be very brittle (and the EWG argued and voted against it)
- A significant = would not work syntactically for **new**, **return**, function calls, etc.

Examples of a significant = in "other situations" are

```
// Warning; not proposed:
p = new char = { 200 };
f( ={ 200 });
return ={200};
```

This is discussed in N2215. Even if I could solve all syntax problems, I wouldn't propose that.

2.5 Missing {…} in nested aggregates

Like C, C++ allows nested aggregates to be initialized without matching their nesting in initializers. For example:

```
struct S { int a, b; };
S a1 = { {1,2}, {3,4} };        // ok
S a2 = { 1,2,3,4 };             // also ok
```

The scheme for {…} outlined here relies on nesting in initializers, so what do we do? First, accept the lack of nesting in traditional aggregate initializers (as in the example above); it does little harm and the compatibility problems involved with changes here would be immense. Second, let's examine such cases where **initializer_list**s are involved.

Consider a container of **int**s reduced to a minimum:

```
template<class T> struct T {
        T(int);
        T(Iterator<T>,Iterator<T>);
        T(initializer_list<T>);
};

vector<int> v = { 1,2,3,4,5,6 };

T<int> t1{1};                   // T(initializer_list<int>)
T<int> t2{v.begin(), v.end()};  // T(Iterator<T>,Iterator<T>)
T<int> t3 { 1,2,3,4,5 };        // T(initializer_list<int>)
T<int> t4(1);                   // ok: T(int)
```

This is all as it has to be to be consistent (and discussed above). Let's try a complication:

```
struct S {
        T<int> t;
};

S s1{v.begin(), v.end()};   // error: S has only one member
                            //        and it isn't an iterator
S s2 { 1,2,3,4,5 };         // error: S has only one member
S s3(1);                    // error: can't initialize aggregate using (…)
S s4{1};                    // ok: calls S(int); 1 is not a list
S s5{{1}};                  // ok: calls S(initializer_list<int>); {1} is a list
```

That is, the { } here indicates {…}-style initialization and passes it elements as initializers to S's elements; that is (just) **t**. The resolutions for **s4** and **s5** might seem surprising at first, but they follow directly from the usual rules for aggregate initialization. In fact, add a = and the initializations of **s4** works in C++98. For **s4**, the one value **1** isn't a list, so when used as an initializer for **s4.t** it invokes **S(int)**. On the other hand, for **s5**, the one value **{1}** is a list, so when used as an initializer for **s5.t** it invokes **S(initializer_list<int>)**.

We can try to see what adding {}s do in the other cases:

```
S s11{{v.begin(), v.end()}};  // ok: T(Iterator<T>,Iterator<T>)
S s22 {{1,2,3,4,5}};          // ok: T(initializer_list<int>)
```

```
S s33({1});                    // error: can't initialize aggregate using (…)
```

3 Directness

Can the explicit/implicit constructor distinction affect {…} initialization? For example, can **f({v})**, where **f()** is declared **void f(X)**, give a different **X** than **X{v}**? The proposed answer is "no, the **X** produced as an argument in **f({v})** and **X{v}** will always be the same value."

{…} provides an alternative solution to the problems that **explicit** was designed to solve.

Consider first some current uses of explicit constructors:

```
struct String1 {
        explicit String1(const char*) { cout << "String1\n"; }
};

struct String2 {
        String2(const char*) { cout << "String2\n"; }
};

//String1 s11 = "asd";          // error
String1 s12("asd");
String2 s21 = "asd";
String1 s22("asd");

void f1(String1) { cout << "f1(String1)\n";}
void f2(String2) { cout << "f2(String2)\n";}

void f(String1) { cout << "f(String1)\n";}
void f(String2) { cout << "f(String2)\n";}

int main()
{
        //f1("asdf");   // error
        f2("asdf");
        f("asdf");      // call f(String2)
}
```

The real advantage of **explicit** is that it renders **f1("asdf")** an error. A problem is that overload resolution "prefers" non-**explicit** constructors, so that **f("asdf")** calls **f(String2)**. I consider the resolution of **f("asdf")** less than ideal because the writer of **String2** probably didn't mean to resolve ambiguities in favor of **String2** (at least not in every case where explicit and non-explicit constructors occur like this) and the writer of **String1** certainly didn't. The rule favors "sloppy programmers" who don't use **explicit**.

I don't think that the distinction between = and ( ) initialization is particularly useful.

Now consider that example, modified to use {…} as proposed:

```
String1 s11 {"asd"};        // now ok
String1 s12 {"asd"};
String2 s21 {"asd"};
String1 s22 {"asd"};

int main()
{
        f1( {"asdf"} );     // now ok
        f2( {"asdf"} );
        f( {"asdf"} );      // now: error, ambiguous
}
```

I don't think anyone consider accepting the definition of **s11** problematic. Nor would I expect anyone to have problems with deeming **f({"asdf"})** an error; personally, I consider it an improvement. The potential problem is accepting **f1({"asdf"})** because "there is no **String1** in sight at the point of call". I have heard the claim that "The whole point of **explicit** is to prevent that!" That latter claim is a bit overstated because **explicit** will still prevent **f("asdf")**, whereas **f1({"asdf"})** is accepted only with more stringent ambiguity control rules (see Section 5). I also expect the latter to be far less common.

Consider the other classical example of the use of **explicit**:

```
vector<double> v{7};        // v is a vector with one element with the value 7.0
vector<double> v(7);        // v is a vector with 7 elements with the value 0.0

void v(const vector<double>&);
f(7);                       // error
f({7});                     // pass a vector with one element with the value 7
f(vector<double>(7));       // ok: seven elements
f(vector<double>{7});       // ok: one element with the value 7
```

I claim that this is exactly what people should and would expect. In particular, **f({7})** passes an initializer list to **f()**. More generally, I think that the problems with **explicit** don't happen for classes with **initializer_list** constructors.

3.1 Composite objects

So, the surprises that **explicit** was introduced to prevent (still) do not occur for ordinary function calls nor for calls of functions that take classes with **initializer_list** constructors. Where could/will it occur? I suspect the answer is "when people use {…} to compose an

argument list for a constructor in a function call, rather than using **T(…)** or **T{…}**." Consider

```
struct Composite {
        Composite(String1, const vector<double>&, int);
}

Composite c1 {"asdf",7,1};          // error: the second argument should be a list
Composite c2 {"asdf",{7},1};        // ok (and 7 is a vector element)

void f(const Composite&);
f("asdf",7,1);          // error: f() takes just one argument

f(Composite("asdf",7,1));   // error: no implicit conversion of "asdf" to String1
                            // and no conversion of 7 to vector<double>

f(Composite(String1("asdf"),vector<double>(7),1)); // ok
                                                   // (but different meaning)

f(Composite({"asdf"},{7},1));       // ok (7 is a vector element)

f( {"asdf",7,1} );      // error: the second argument of Composite should be a list
f( {"asdf",{7},1} );    // ok (and 7 is a vector element)
```

Why does **Composite c1 {"asdf",7,1};** fail? It fails because according to rule 4 in Section 2.1, the initializer of an object with a list initializer constructor must be initialized by an initializer list. Now, why doesn't it call **vector**'s ordinary constructor taking an **int** instead (that is, pick **vector<int>(7)** since it didn't get an initializer)? Well, the rule says it doesn't and if it did we would get some seriously surprising resolutions.

3.2 Discussion

There are two uses of {…} that some has expressed concern about:

- The use of {…} to get direct initialization for an individual argument; e.g., in **f(Composite({"asdf"},{7},1))**. In this case, there is only one possible type for **"asdf"** to convert into and the {…} explicitly says that we want direct initialization (allowing use of **explicit** constructors).
- The use of {…} to group arguments into a single argument (to be passed to a constructor) without naming the intended class; e.g., in **f({"asdf",{7},1})**. In this case, there might be another **f()** with a constructor for which **{"asdf",{7},1}** is a better match than **Composite(String1, const vector<double>&, int)**.

In both cases, {…} is a loud warning that something new and potentially different is going on. In both cases, you can use the old syntax if you so prefer. The worry would be that because **Composite**, **String1**, and/or **vector<double>** were not explicitly mentioned

in the call, the creation of the object would be seen as a surprising conversion – the kind of surprise that **explicit** was introduced to avoid.

The overload resolution issues that could arise from **f({"asdf",{7},1})** are discussed in Section 5.4 (after the overload resolution rules have been presented).

For **f(Composite({"asdf"},{7},1))**, there is only one possible type for **"asdf"** to convert into (**String1**) and the {…} explicitly says that we want direct initialization (allowing use of **explicit** constructors). Similarly for **{7}**.

How serious is this problem compared with what we get in exchange? Basically, I don't consider the treatment of **explicit** in the {…} proposal a serious problem. It doesn't break old code, it catches a few bugs, and if you don't like it you don't have to use {…} in function calls unless you actually want an **initializer_list**. What you get in exchange is uniformity, a terse way of expressing a few argument lists, and a way of catching some rare overloading errors involving **explicit**.

For me, uniformity is a big deal and decides this tradeoff in favor of the {…} proposal.

In addition to the arguments I can add that several people have commented that the consider the terseness of

```
f( {"asdf",{7},1} );     // ok (and 7 is a vector element)
```

compared to

```
f(Composite(String1("asdf"),vector<double>(7),1)); // ok
                                                   // (but different meaning)
```

and

```
initializer_list<int> a = { 1 };
f(Composite(String1("asdf"),vector<double>(a),1)); // ok
```

most significant and desirable, especially for application programmers.

3.3 Alternatives

Can we eliminate those last implicit conversion problems without breaking uniformity? One idea that just doesn't work is to say that as function arguments, initializer lists have copy construction semantics. Consider first an old example that illustrates a (minor) problem with **explicit**:

```
struct Weird {
        Weird(double) { cout <<"Weird(double)\n"; }
        explicit Weird(int) { cout <<"Weird(int)\n"; }
};
```

```
void g(Weird) { }

void f()
{
        Weird w1(1);
        Weird w2 = 1;
        g(1);
}
```

The result is

```
Weird(int)
Weird(double)
Weird(double)
```

That is the non-direct initialization chooses the non-explicit, less good, match **Weird(double)**. Any system that distinguishes between "ordinary" and explicit constructors and allows the (weird) definition of **Weird**, will have a variant of this problem. That is a (technical) reason for preferring not to distinguish explicit constructors in the semantics of the uniform syntax. Consider:

```
Weird w{1};
g({1});
g(w);
```

Given the proposal, **w** gets the same value as is passed to **g()** (twice). Any proposal that distinguishes **explicit** constructors from "ordinary" constructors won't. That is, it violates the principle of uniformity as expressed in the first example in Section 1. Violating the second example from there does not require anything weird. Consider:

```
struct String1 {
        explicit String1(const char*) { cout << "String1\n"; }
};

void h(String1) { cout << "f1(String1)\n"; }

void ff()
{
        String1 ss("asd");      // ok
        h("asd");               // error
}
```

"Respecting **explicit**" for {…} initialization would give

```
// Warning; not proposed:
void fff()
{
```

```
        String1 ss{"asd"};    // ok
        h({"asd"});           // error
}
```

Given these fundamental examples, I don't see how to produce a proposal that both "respects **explicit**" and is uniform. Any solution would somehow involve respecting **explicit** for both single argument and multiple argument constructors. For example:

```
// Warning ("respects explicit"); not proposed:

struct Composite {
        explicit Composite(String1, const vector<double>&, int);
};

Composite c2 {"asd",{7},1};          // ok (and 7 is a vector element)

void f(const Composite&);

f( {"asd",{7},1} );                  // error: Composite() is explicit
f( Composite{"asd",{7},1} );         // error: String1() and vector<T>() are explicit

f( Composite{String1{"asd"},vector<double>{7},1} );     // ok
```

This would make {…} initialization "direct initialization, except in function calls". I don't see the significant benefit. Please again note that the ambiguity rules already catch all such problems where overloaded functions are involved. Also note that the notational burden imposed by "respecting **explicit**" in these unambiguous cases is significant. The enforcement of the ambiguity rule does not imply the same notational overhead as "respecting **explicit**" because it is only applied where a possible ambiguity exists. For example (continuing the example above):

```
// applying proposed rules:

struct Composite2 {
        Composite2(String1, const vector<double>&, int);
};

void f(const Composite2&);

f( {"asd",{7},1} );                  // error: ambiguous
f( Composite{"asd",{7},1} );         // ok
f( Composite2{"asd",{7},1} );        // ok
f( Composite{String1{"asd"},vector<double>{7},1} );     // ok, but verbose
```

Users can choose a degree of terseness/verbosity that suits their taste and application. Even the least verbose uses will be protected by the requirements on the structure of {…} initializer lists and the ambiguity control rules.

4 Narrowing

Are arrays and/or **struct**s without constructors exempt from the rule against narrowing? For example, can **char a[] = { 2000 };** be valid on a machine with 8-bit **char**s? The proposed answer is "no".

By narrowing, I mean implicit conversion that can throw away information such as **double** to **int**, **double** to **float**, and **int** to **char**. On the face of it, this implies a major and unsolvable compatibility problem; consider:

```
char a1[] = { 'a', 2000 };      /* int to char conversion */
struct S { char a, b; };

struct S s1 = { 'z', 0 };       /* note: 0 is an int, so int to char conversion */

void f(int x1, int x2, int x3)
{
        struct S s2 = { x1 };           /* int to char conversion */
        char a2[] = { x1, x2 };         / * int to char conversion */
        /* … */
}
```

This example is not just C++; it is also C89 (but not K&R C). But, there is hope! Looking at code, I found that actual narrowing was rare and even rarer if we used the rule that when the initializer was a known value (e.g., and integer literal or a null pointer) we only counted it as an error if the actual value couldn't be losslessly represented as the target type.

So, if you have access to a few million lines of code and a compiler, please try modifying the compiler to tell if narrowing (as described below) occurs. Current compilers often give warnings, but finding these particular narrowing warnings among other warnings and filtering out the non-problems, such as 0 for a **char**, is non-trivial. Only empirical studies can tell us if banning narrowing could do real hard and guide the detailed design. In the worst case, a simple backwards compatibility switch could serve people who have to accept narrowing conversions.

4.1 What is narrowing?

The definition of narrowing here is a representation of what the EWG came up with at the suggestion of CWG in Kona. A more formal description will be written later:

A conversion is accepted as non-narrowing if it converts

1. an integral type to an integral type with equal or greater range (e.g., **char** to **int** but **not int** to **char** for an implementation where **sizeof(char)<sizeof(int)**). Enumeration types are considered integral in this context.
2. a floating point type to a floating point type with equal or greater range.
3. a known integral value to an integral type so that the particular value isn't changed (i.e. can be converted back to its original type without changing its value, e.g., **char c1 = 20;** but not **char c2 = 2000;** for an 8-bit **char**)
4. a known floating point value into a floating point type for which it is in range (e.g., **float f1 = 2.1;** but not **float f2 = MAX_FLOAT*2;**)

It follows that **int x = 2.0;** is considered narrowing even though **2.0** can be losslesly converted to the integer **2**. However, floating to integer and integer to floating-point conversions were considered too subtle to handle in general.

Note that these definitions are implementation defined in that **2000** can be converted to a char provided a **char** is 16-bits or more. The intent is not to enforce portability (by checking against minimum ranges) but to catch actual errors of a specific target machine. This is obviously a design choice that can be debated, but the more restrictive choice (give an error if any legal implementation could give narrowing) would lead to many constants being considered narrowed even if they occurred in programs that were never intended for universal portability (e.g. a program on an embedded processor relying on large characters or a major application relying of 32-bit or more **int**s).

Why bother?
Implicit narrowing conversions have a long history as a source of bugs – stretching back to before C had explicit conversions (casts); see N2215 for a historical view. Many of these bugs are subtle and many manifest themselves only when code is ported.

Unfortunately, the strongest indicator of the size of the problem would be if we find a lot of narrowing in a survey of existing use of aggregate initializers. If we find a lot, we might have to back off for compatibility reasons, leaving

```
// Warning; not proposed:
vector<char> vc = { 1, 2, 2000 };          // error
char[] ac = { 1, 2, 2000 };                // not error (maybe, I hope not)
struct { char a, b, c; } s = { 1, 2, 2000 };   // not error (maybe, I hope not)
```

That would be a shame, both because of the non-uniformity, but also because of the lost opportunity to secure code against a class of errors.

Obviously, I expect a massive increase in the use of initializer lists. In particular, I expect a lot of constructor calls and calls of functions that insert values into something to become initializer lists in new code. People will still make the usual narrowing mistake and – unless we do something – find them the usual way: debug runs, code reviews, bugs found when porting and other "late events". What we have here is an opportunity because

the {…} initialization is new syntax so we can craft the rules to catch these bugs early. The question is whether existing code is sufficiently clean for those rules to apply to traditional aggregate initializers also.

Also, banning narrowing in initializer lists simplify and improve the overload resolution rules because with that ban in place conversions go only one way: towards types with larger ranges.

Please note that another key example:

```
typedef char* Pchar;
int i = 9;
Pchar p = "asd";
i = int(p);
p = Pchar(i);
i = int{p};      // error
p = Pchar{i};  // error
```

This has nothing to do with narrowing (though it has been discussed under that heading). The protection against pointer to **int** conversions comes directly from the uniformity principle, rather than from anti-narrowing. Now imagine that this was done in a template using a type parameter **T** and a value of another parameter type **v**:

```
x = T(v);       // construction or potentially dangerous cast
y = T{v};       // construction; not a cast
```

4.2 Alternatives

If the proposed rule proves to break too much code, I see several alternatives

- Don’t enforce anti-narrowing for aggregates
- Don’t enforce anti-narrowing for structs
- Don’t enforce anti-narrowing for arrays
- Enforce anti-narrowing for constant expressions and floating-point to integral only
- Enforce anti-narrowing for constant expressions only

Based on the EWG votes, I assume the simple “no narrowing conversions” rule for initialization with {…} and wait for studies of more code to see if we need to go to an alternative.

5 Overloading and type deduction

I don't think the rules for overloading can be derived directly from the ideals for initialization. However, I think the following rules are necessary and sufficient (once precisely stated):

- When using {…} initialization, i**nitializer_list<T>** constructors take priority over all other forms of initialization.
- If a {…} initializer list can match more that one **initializer_list<T>** from a set of alternatives, the result is an ambiguity; we don't prefer one **initializer_list** over another.
- The type of an **initializer_list<T>** cannot be deduced, but the **T** in **initializer_list<T>** can be deduced for completely homogenous lists
- A set of constructors resolves in the same way as a set of overloaded function with the same arguments.

I will address the conjectured problems in stages. Please note that the Kona "hall discussions" were terminally confused by errors in the WP formulation. Therefore, part of this section is tutorial.

5.1 Initializer list constructors and other constructors

Consider a class with both an initializer-list constructor and other constructors:

```
Struct S {
        S(initializer_list<int>);
        S(string);
        S(int);
};

S s1{ 1,2 };      // S(initializer_list<int>)
S s2{"asd"};      // S(string)
S s3{ 1 };        // S(initializer_list<int>)
S s4{ 1.8 };      // error: no narrowing
```

There is no way you can call **S::S(int)** using the {…} syntax. If you must use

```
S s5(1);          // S(int)
```

The simple rule that the initializer list constructor is preferred to other constructors if it can be called resolves all ambiguities involving only a single possible "target type", e.g. in return statements, variable initializations, and non-overloaded function calls.

The reason that the initializer list constructor must be preferred is that we can't have **{}**, **{1}**, **{1,2}**, and **{1,2,3}** select different constructors. See the Appendix B of N2215.

Calling a set of overloaded functions, leads to selection that's identical to that of constructors in variable initialization:

```
void f(initializer_list<int>);
void f(string);
void f(int);

f({ 1,2 });      // f(initializer_list<int>)
f({"asd"});      // f(string) – this is direct initialization
f({1});          // S(initializer_list<int>)
f({ 1.8 });      // error: no narrowing
```

In that, it differs from calls that do not use the {…} syntax:

```
f(1,2);          // error: no f() taking two arguments
f("asd");        // error: the char* -> string constructor is explicit
f(1);            // f(int)
f(1.8);          // ok: narrows (Ouch!)
```

If there are no initializer list constructors, {…} initialization works exactly as direct initialization:

```
struct S2 {
        // S2(initializer_list<int>);
        S2(string);
        S2(int);
};

S2 s21{"asd"};       // S2(string)
S2 s22("asd");       // S2(string)
S2 s31{1};           // S2(int)
S2 s41(1);           // S2(int)
S2 s51{1.8};         // error: no narrowing
S2 s61(1.8);         // S(int), narrowing, ouch!

void f2(string);
void f2(int);

f2({"asd"});         // f(string) – this is direct initialization
f2({1});             // f(int)
f2({1.8});           // error: no narrowing
```

In that, it differs from calls that do not use the {…} syntax:

```
f2("asd");       // error: the char* -> string constructor is explicit
f2(1);           // f(int)
```

```
f2(1.8);        //  ok, narrows (Ouch! )
```

Here, we have arrived at a point that caused some people to worry:

```
f2({"asd"});   // f(string) – this is direct initialization
f2("asd");     // error: the char* -> string constructor is explicit
```

There are "no **string** in sight" yet **"asd"** is converted to **string**. People also mentioned **vector** as an example of this (potential) problem:

```
fv(vector<int>&);
fv({1});       // no vector in sight; what happens?
fv(1);         // no vector in sight; what happens?
```

There was a worry that **fv({1})** would mean **fv(vector<int>(1))**, bypassing the protection offered by explicit conversions. That doesn't happen: **vector**'s initializer list constructor takes priority (See also Section 3):

```
fv({1});       // creates a vector with one element from the initializer list
fv(1);         // error: vector<int>(int) is explicit
```

I think that the case where **{1}** binds to an initializer list, just as **{}**, **{1,2}**, and **{1,2,3}** does is no problem – it is what most people would expect if they didn't know the rules. The real worry is **f2({"asdf"})**. That problem is addressed in Section 3.2.

Consider the case of overloading where the inilializer list constructor can be in one class and the ordinary constructor in another:

```
struct S1 {
        S1(initializer_list<int>);
};

struct S2 {
        S2(double);
        S2(int);
        S2(int,double);
};

f(S1);
f(S2);

f({1});        // f(S1): initializer_list constructors are preferred
f({1.0});      // f(S2): 1.0 cannot converted to an int   (no narrowing)
f({1, 1.0});   // f(S2); 1.0 cannot be converted to int
f({1, 1.0, 1}); // error: 1.0 cannot be converted to int
```

Will the resolution to call the "ordinary constructors" in **S2** be surprises? Undoubtedly: every form of overload resolution will be surprising to someone in some cases. The real questions are "will it be more surprising than alternatives?" and "will the surprises be frequent compared to other surprises?" I don't see why they should be. First of all, the rules for {…} initialization will encourage the use of completely homogenous initializer lists, so **f({1,1.0})** and **f({1,1.0,1})** will appear as likely arguments for non-**initializer_list** constructors. Secondly, people who worry will qualify; for example **f(S2{1,1.0})**.

5.2 Multiple initializer list constructors

We get to select among several initializer list constructors when two classes have them or in general if more than one overloaded function have **initializer_list** arguments. Consider

```
struct S1 {
        S1(initializer_list<int>);
};

struct S2 {
        S2(initializer_list<double>);
};

f(S1);
f(S2);

f({1});         // error: ambiguous,
                // we don't prefer one legal use of initializer_list over another
f({1.0});       // f(S2): 1.0 cannot converted to an int   (no narrowing)
f({1, 1.0});    // f(S2): 1.0 cannot be converted to int
f({1.0f});      // f(S2): 1.0f cannot be converted to an int
```

Here, we had to decide: Is one initializer list constructor ever better than another (when both could be called)? We decided that the answer should be "no".

Obviously, we could craft more subtle rules, but all we tried resulted in more traps for the unwary without significant benefits to people writing what we considered reasonable code. For example, we could have accepted **f({1})** as a call of **f(initializer_list<int>)** because **1** is an **int** but needs a conversion to become a **double**. However, there doesn't seem to be a need (a significant use case).

Note that if we didn't have the anti-narrowing rule **f({1.0})**, **f({1.0f})**, and **f({1,1.0})** would also have been ambiguous.

Note also that we did not say that an **initializer_list<float>** can be converted to an **initializer_list<double>**; there is no such rule. It is just that a **float** can be converted to a **double** so that we can construct an **initializer_list<double>** from a list of **float**s.

5.3 Type deduction

We can deduce **T** in **initializer_list<T>**, but not **initializer_list<T>**. This deduction takes place only for completely homogenous initializer lists (no conversions use). This formulation is new since N2215, but it is not a change in intent. Consider why we need any deduction:

```
template<class t> void f(const vector<T>& );

f({1,2,3});          // f(vector<int>&);
f({1.2, 3.4, 4.5 }); // f(vector<double>&);
f({"asd", "lkj"});   // f(vector<const char*>&);
f( {1, 2.3});        // error: cannot deduce from non-homogenous list
```

We consider this use case important enough to require some deduction, but consider Jaakoo's Kona example:

```
template<class t> void f(T);
template<class t> void f(const vector<T>&);

f({1,2,3});      // f(vector<int>&);
```

It has to resolve this way to make any sense; we couldn't have the general template "hijack" all initializer list. But how do we achieve that? We could consider **void f(const vector<T>&)** a specialization, but why should a conversion from an initializer list to a **vector** be better than a perfect match to the general template? That explanation doesn't fly. It is only the **int** in **vector<int>** that is deduced.

Consider also:

```
template<class t> void f(initializer_list<T>&);
template<class t> void f(const vector<T>&);

f({1,2,3});      // error: ambiguous
```

This is ambiguous, because both **initializer_list<int>** and **vector<int>** are considered matches and we don't consider one match of an **initializer_list** better than another. Yes, we could consider **initializer_list<int>** a match without conversions and **vector<int>** a match with conversions, but we decided (again) not to step on that slippery slope.

Note that this rule implies that we can never do both a **initializer_list<T>** deduction and a conversion of an element type.

Note that the homogenous initializer list rule does allow resolution for lists of unrelated types:

```
struct S1 {
        S1(initializer_list<int>);
};

struct S3 {
        S3(initializer_list<std::string>);
};

f(S1);
f(S3);

f({1});          // ok: f(S1)
f({"asd"});      // ok: f(S3)
```

What doesn't deal with is resolution of non-homogeneous lists of unrelated types:

```
string s = "xyz";
f({"asd", s }); // error: can't deduce element type
```

I'm not philosophically opposed to allowing such cases and I don't see any obvious technical problems with them, but I'd need some further reasoning to go in that direction.

5.4 Discussion

Let's consider some potentially problematic examples (based on what was presented in N2215 and its predecessors):

```
void f(Y);
void f(X);
f({1,2});
```

Assume that the programmer who wrote **f({1,2})** intends to invoke **f(X)** and see what the presence of a (potentially unsuspected) **f(Y)** can do to confound that. There are a host of alternatives:

(1) If **X** has a list-initializer constructor that accepts **{1,2}** the worst that can happen is that **Y** also has such a list-initializer constructor so **f({1,2})** is ambiguous. This is as I think it should be (someone might prefer to get into the tricky business of picking a best match among initializer-list constructors, but I don't propose to).

(2) If **X** does not have a list-initializer that accepts **{1,2}** but **Y** has, then **Y** has an opportunity to highjack the call by having a list-initializer that accepts **{1,2}**. This is exactly equivalent to the case where a call **g(1,2)** is a match with conversions and someone comes along with a **g()** that matches exactly. In either case, there shouldn't be a problem because neither **f()** nor **g()** should

have been overloaded unless the overloaded functions were semantically equivalent and the best match was the best to invoke.

(3) If **X** does not have a list-initializer that accepts **{1,2}** and neither does **Y**, we have a overload resolution problem (between **X**'s constructors and **Y**'s). This is exactly equivalent to the case where a call **g(1,2)** is a match with conversions and someone comes along with a **g()** that also has a match. The usual overload resolution rules decide.

In each of these cases, being explicit about passing an **X**: **f(X{1,2})** will resolve the issue. Why don't we require that in all cases?

(1) One of the reasons for liking {…} is terseness: **f({1,2})** really is simpler to read and write than **f(vector<int>{1,2})**.

(2) In many cases, there really is just one **f()** that is remotely relevant to **f({1,2})** so adding qualification is redundant (and many programmers object to such redundancy). This will be the case in many small test cases and specialized applications.

(3) The likely nasty ambiguity cases occur only for short initializer lists, such as **f({1,2})**. That is, **f({})** is unlikely to be pick an undesired match; **f({1})** is the most likely to, but the least likely to occur because it will only be used when there is an initializer-list constructor and not for grouping; **f({1,2})** needs an match on two constructor arguments to be a problem (so it's less likely to cause problems than an ordinary **g(1)** call; **f({1,2,3})** is less likely to cause an accidental match; etc.: **f({1,2,3,4,5,6,7,8,9})** could only be a real problem for someone with nine-argument constructors.

(4) The chance of accidental matches goes down dramatically when more specialized element types are used. For example, consider **associate({sp,db,cb})** where **sp** is a **Shape***, **db** is a **dialog_box&**, and is a pointer to a **callback** function. Here **{sp,db,cb}** is unlikely to accidentally match some unrelated constructor. That is, using a built-in type, such as **int**, is the worst case scenario.

(5) Why should the language rules force programmers who wants terseness and ambiguity control (for perfectly good reasons) to write more to please programmers who prefer (for perfectly good reasons) to be more explicit – and can be?

(6) Uniformity. {…} is an initializer in all contexts. Saying that {…} is an initializer in all cases except for argument passing where **X{…}** is needed (except possibly for non-overloaded functions) would be a blemish (IMO).

(7) Potential generic uses: Maybe the programmer really wanted **f({1,2})** to choose the best match among a set of functions? That is, "build an object of the type for which **{1,2}** is the best choice when you want to do **f()** on it." Remember, selection based on type is the basis of generic programming and was reviled as "seriously error prone" and "just a language design error" for decades.

For me, (5) is decisive. C++ is not and cannot become a seriously hand-holding language. If that's what we want, {…} initialization is not the worst problem by far. The C chaotic conversion rules, which {…} initialization helps to mitigate, are a much more serious problem. That doesn't mean that I disregard the other argument. For example, this *is* a proposal for uniform initialization and I'm very reluctant to introduce special cases.

But what about accidents and the prevention of accidents? Assume that a programmer expects **f(X)** to be called. How might a **f(Y)** hijack a call?

(1) Assume that both **X** and **Y** has an initializer-list constructor. For **Y** to "hijack" **f({1,2})**, **X**'s initializer-list constructor must not accept **int** elements. In that case, it seems unlikely that someone would use **{1,2}**. A more likely example would be **{1,2.0}**, which because of the anti-narrowing rule would not match **initializer_list<int>** so that **Y** could highjack using an **initilizer_list<double>**. I would think that is less likely than many existing overload resolution problems. I suspect that people will get used to writing homogeneous initializer lists when they want to invoke an initializer-list constructor or at least limit themselves to fairly obvious conversions, such as **int** to double and string literal to **std::string**.

(2) Assume that **X** has the only list-initializer constructor and that **f({1,2})** somehow failed to match it. That way, we can get competition among a set of non-initializer list constructors from **X** and **Y**. If that's a worry, we can write **f({1,2,})** that trailing comma will prevent a match on anything but an initializer-list constructor. It's an odd syntax (and postfix), but it's one we have from way back and have to accept for aggregate initializer compatibility anyway. Using a suffix list indicator is not ideal, but note that essentially all problems will happen for one, two, and three element lists (and of those I guess that one and three will not be all that common), so that the suffix comma will not be all that hidden. I suspect that (at least after a brief period of overuse) most programmers will prefer the shorter **f(1)** to **f({1})**, thus eliminating the one-element case.

(3) Assume that neither **X** nor **Y** has initializer-list constructors. In that case **f({1,2})** simply triggers an overload resolution "competition" equivalent to **g(1,2)**. That is sometimes what is desired and when it is not it is an existing problem that we know how to deal with.

(4) Assume that **X** has no initializer-list constructor, but **Y** does. In this case, the priority given to initializer-list constructors favor the hijacker (remember we assumed that the programmer somehow expected **f(X)** to be called). I think it would be fair to expect that someone who uses {…} will remember the possibility of initializer-lists constructors.

This discussion is only relevant if there is an **f(Y)** somewhere. Unless **f()** is overloaded, there is no problem. Furthermore, we don't propose overload resolution based on non-explicitly typed initializer lists except in a special case of **T** in **std::initializer_list<T>**. Hijacking by a perfectly generic **template<class T> void f(T)** is not possible.

How serious are these overloading problems compared with what we get in exchange? I don't see the problems of overload resolution among different sets of constructors and hijacking as especially likely or serious compared to existing problems. Programmers who feel **f({1,2})** too error-prone for their code (notably library writers):

(1) Can explicitly qualify: **f(X{1,2})**
(2) Can explicitly mark {…} as an initializer_list (and not arguments for a non-initializer constructor) by a suffix comma: : **f({1,2,})**
(3) Can combine (1) and (2): **f(X{1,2,})**
(4) Cannot (unfortunately) mark an {…} as arguments for a non-initializer constructor (and not an initializer_list); there, we have to fall back on the conventional **f(X(1,2))**

Again, there are people who prefer terseness.

6 Acknowledgements

Gabriel Dos Reis and Jaakko Järvi helped in formulating examples and resolutions. Lawrence Crowl sent examples and constructive comments.