Doc No: N1919=05-0179
Date: December 11, 2005
Project: JTC1.22.32
Reply to: Bjarne Stroustrup
bs@cs.tamu.edu

Initializer lists

Bjarne Stroustrup and Gabriel Dos Reis
Texas A&M University

Abstract

This paper presents a synthesis of initialization based on consistent use of initializer lists. The basic idea is to allow the use of initializer lists wherever initialization occurs and to have identical semantics for all such initialization. For user-defined types, such initialization is defined by a sequence constructor.

The discussion is based on the earlier papers and on discussions in the evolution working group. Much of this paper summarizes the discussions of alternatives.

In addition to the main discussion and proposal, two subsidiary proposals (for the use of initializer lists as sub-expressions and for disallowing narrowing in initializations using initializer lists) are also presented.

If this proposal is accepted, we will propose that a sequence constructor be added to each standard library container.

Suggested working paper text is an appendix (yet to be completed).

1 Previous work

The direct ancestor to this paper is N1890=05-0150 “Initialization and initializers”. That paper provides an outline of solutions of a set of related problems. This paper refines the parts of that paper that deals with initializer lists and sequence constructors. Please note that the solution presented here differs slightly from the one presented in N1890 and is more general and more complete (though not yet ready for a vote). In particular, it provides for initializer lists as a general mechanism for variable length homogeneous argument lists.
The other parts of “the initialization puzzle” presented in N1890 are or will be presented in companion papers, such as Gabriel Dos Reis and Bjarne Stroustrup “Generalized constant expressions” (N1920=05-0180), dealing with constant expressions and constant

expression constructors). Here is a list of problems and suggested improvements that has led to the current design:

- General use of initializer lists (Dos Reis & Stroustrup N1509, Gutson N1493, Meredith N1806, Meridith N1824, Glassborow N1701)
- There are four different syntaxes for initializations (Glassborow N1584, Glassborow N1701)
- C99 aggregate initialization (C99 standard)
- Type safe variable length argument lists (C++/CLI)
- Overloading "new style" casts
- Making **T(v)** construction rather than conversion (casting)
- Variadic templates (N1603 and N1704)

In each case, the person and paper referred to is just one example of a discussion, suggestion, or proposal. In many cases, there are already several suggested solutions. This is not even a complete list: initialization is one of the most fertile sources of ideas for minor improvements to C++. Quite likely, the potential impact on the programmer of sum of those suggestions is not minor. In addition to the listed sources, we are influenced by years of suggestions in email, newsgroups, etc. Thanks and apologies to all of you who contributed, but are not explicitly mentioned here.

2 Summary

As the result of the detailed discussion presented in the following sections we propose:

- To allow an initializer list (e.g., **{1,2,3}** or **={1,2,3)**) wherever an initializer can appear (incl. as a return expression, an function argument, a base or member initializer, and an initializer for an object created using **new**). An initializer list appears to the programmer as an rvalue.
- To introduce type **initializer_list** for the programmer to use as an argument type when an initializer list is to be accepted as an argument. The name **initializer_list** is known to the compiler, but its use requires including a definition of **initializer_list** from namespace **std**.
- To distinguish sequence constructors (a single-argument constructor with a **initializer_list** argument type) in the overload resolution rules.
- To use a type name to indicate the intended type of an initializer list (e.g., **X{1,2,3}**). This construct is primarily for disambiguation.
- To allow an initializer list to be used as arguments to a constructor of a class when no sequence constructor can be used (e.g. **f({1,2})** can be interpreted as **f(X(1,2))** when **f()** unambiguously takes an **X** argument and **X** does not have a sequence constructor for **{1,2}**). This mirrors the traditional (back to K&R C) use of initializer lists for both arrays and **struct**s and is needed for initializer lists to be used for all initializers, thus providing a single notation with a single semantics for all initialization.

- Initialization using an initializer list, for example **X x = { y };** is direct initialization, not copy initialization.
- A separate/subsidiary proposal is to disallow narrowing conversions when using the initializer list notation. For example, **char c = { 1234 };** would become an error.
- A separate/subsidiary proposal is to allow an initializer list as a sub-expression, for example, **x=y+{1,2}**.

The aim is to make initialization with initializer lists a uniform notation for initialization with a single semantics for all cases. This proposal breaks no legal ISO C++ program except those that uses the proposed reserved name **initializer_list**.

3 Four ways of providing an initializer

Initialization of objects is an important aspect of C++ programming. Consequently, a variety of facilities for initialization are offered and the rules for initialization have become complex. Can we simplify them? Consider How to initialize an object of type **X** with a value **v**:

```
X t1 = v;       // "copy initialization" possibly copy construction
X t2(v);        // direct initialization
X t3 = { v };   // initialize using initializer list
X t4 = X(v);    // make an X from v and copy it to t4
```

We can define **X** so that for some **v**, 0, 1, 2, 3, or 4 of these definitions compile. For example:

```
int v = 7;
typedef vector<int> X;
X t1 = v;       // error: vector's constructor for int is explicit
X t2(v);        // ok
X t3 = { v };   // error: vector<int> is not an aggregate
X t4 = X(v);    // ok (make an X from v and copy it to t4; possibly optimized)
```

and

```
int v = 7;
typedef int X;
X t1 = v;       // ok
X t2(v);        // ok
X t3 = { v };   // ok; see standard 8.5; equivalent to "int t3 = v;"
X t4 = X(v);    // ok
```

and

```
int v = 7;
```

```
typedef  struct { int x; int y; } X;
X t1 = v;       // error
X t2(v);        // error
X t3 = { v };   // ok: X is an aggregate ("extra members" are default initialized)
X t4 = X(v);    // error: we can't cast an int to a struct
```

and

```
int v = 7;
typedef  int* X;
X t1 = v;       // error
X t2(v);        // error
X t3 = { v };   // error
X t4 = X(v);    // ok: unfortunately this converts an int to an int* (see §6.1)
```

Our aim is a design where a single notation where for every (**X**,**v**) pair:

- Either all examples are legal or none are
- Where initialization is legal, all resulting values are identical

3.1 Can we eliminate the different forms of initialization?

It would be nice if we didn't need four different ways of writing an initialization. Francis Glassborow explains this in greater detail in N1701. Unfortunately, we loose something if we eliminate the distinctions. Consider:

```
vector<int> v = 7;     // error: the constructor is explicit
vector<int> v(7);      // ok
```

If the two versions were given the same meaning, either

- both would be correct (and we would be back in "the bad old days" where all constructors were used as implicit conversions) or
- both would fail (and every program using a vector or similar type would fail).

We consider both alternatives unacceptable.

> **Question:** but why would anyone expect the **v = 7** notation to work? And if they did why would they expect it to have a different effect from the **v(7)**? Some people expect the **v = 7** example to initialize **v** with the single element **7**. Scripting languages supply a steady stream of people with that expectation.

The equivalent problem for argument passing demonstrates that we cannot simplify by eliminating copy initializations or explicit constructors while defining argument passing and value return as initialization:

```
void f(const vector<int>& v);
f(7);     // error: the constructor is explicit
```

```
vector<int> v = { 1,2,3,4,5,6,7 };
f(v);     // copy
```

We want the **f(7)** example to fail as an example of a class of programming errors that occurred frequently before explicit constructors were introduced and continue to this day when people forget to make their single-argument constructors explicit. Thus, we need the current copy initialization semantics for argument passing, whereas people most often prefer direct initialization of variables.

3.2 A constructor problem: Disabled copy

Also, consider the common practice of "outlawing" copying by declaring a private copy constructor:

```
class X { /* … */ X(int); private: X(const X&); };  // no copy allowed
X x0 = X(1);   // error (copy)
X x1 = 1;      // error (copy)
X x2(1);       // ok (no copy)
```

To have a single rule here would require us to choose between

- breaking a lot of code (disallow all three cases) and
- requiring that copy not be considered (allow all three cases).

We suspect we could live with the latter choice, but it would be a change making the language more permissive and unless we guaranteed that no copy was done in any of the cases, the result (invoking a private constructor) would be surprising and violate a very reasonable assumption: A private function is not called from outside the class' members.

Consider finally the most explicit form of initialization:

```
vector<int> v = vector<int>(7);      // copy?
X e3 = X(1);                         // copy?
```

We cannot recommend that style for systematic use because it is unnecessarily verbose and implies serious inefficiency unless compilers are guaranteed to eliminate the copy. It would also break reasonable expectations unless the access to the copy constructor is checked (to make the initialization of **e3** fail). If we special-cased this form of initialization (to make the examples legal and efficient), we would end up with a semantics that differed from that of argument and return value initialization. For example:

```
template<class T> void f(T v);
f(vector<int>(7));     // copy? Yes, we must copy
f(X(1));               // copy? Yes, we must copy and copy of X is disallowed
```

We conclude that we must live with different meanings for different initialization syntaxes. That implies that we can try to make the syntax and semantics more general and

regular, but we cannot reach the ideal of a single simple rule without serious side effects on existing code. It is possible that some satisfactory solution exists to this puzzle, but having looked repeatedly we haven't found one and we don't propose to spend more time on this.

3.3 A constructor problem: explicit constructors

Explicit constructors can cause different behavior from different forms of initialization. Consider:

```
struct X {
        explicit X(int);
        X(double);      // not explicit
};

X a = 1;        // call f(double)
X b(1);         // call f(int)

void f(X);
f(1);           // call f(double)
```

The reason **f(double)** is called is that the explicit constructor is considered only in the case of direct initialization. We consider this backwards: what should happen is that the best matching constructor should be chosen, and the call then rejected if it is not legal. That would make the resolution of these cases identical to the cases where a constructor is rejected because it is private.

We don't make a proposal for that change here, but note this as a case where a difference in initialization behavior could be eliminated by a rule change. See also Section 6.1.1.

We furthermore conjecture that having both an explicit and a non-explicit constructor taking a single argument is poor class design.

4 Initializer lists

There is a widespread wish for more general use of initializer lists as a form of user-defined-type literal. The pressure for that comes not only from "native C++" wish for improvement but also from familiarity with similar facilities in languages such as C99, Java, C#, C++/CLI, and scripting languages. Our basic idea is to allow initializer lists for every initialization. What you loose by consistently using initializer lists are the possibilities of ambiguities inherent in = initialization (as opposed to the direct initialization using **( )** and proposed **{ }**).

Consider a few plausible examples:

```
X v = {1, 2, 3.14};                 // as initializer
const X& r1 = {1, 2, 3.14};         // as initializer
X& r2 = {1, 2, 3.14};               // as lvalue initializer

void f1(X);
f1({1, 2, 3.14});                   // as argument
void f2(const X&);
f2({1, 2, 3.14});                   // as argument
void f3(X&);
f3({1, 2, 3.14});                   // as lvalue argument

X g() { return {1, 2, 3.14}; }      // as return value

class D : public X {
        X m;
        D() : X({1, 2, 3.14}),      // base initializer
              m({1, 2, 3.14}) { }   // member initializer
};
X* p = new X({1, 2, 3.14});  // make an X on free store X
                             // initialize it with {1,2,3.14}

void g(X);
void g(Y);
g({1, 2, 3.14});        // (how) do we resolve overloading?

X&& r = { 1, 2, 3 };  // rvalue reference
```

We must consider the cases where **X** is a scalar type, a class, a class without a constructor, a union, and an array. As a first idea, let's assume that all of the cases should be valid and see what that would imply and what would be needed to make it so. Our design makes these examples legal, with the exceptions of the lvalue examples. We don't propose to make initializers lvalues.

Note that this provides a way of initializing member arrays. For example:

```
class X {
        int a[3];
public:
        X() :a({1,2,3}) { }
};
```

Some people consider this important. Over the years, there has been a slow, but steady, stream of requests for some way of initializing member arrays.

4.1 The basic rule for initializer lists

The most general rule of the use of initializer lists is:

- Look for a sequence constructor and use it if we find a best one; if not
- Look for a constructor (excluding sequence constructors) and use it if we find a best one; if not
- Look to see if we can do traditional aggregate or built-in type initialization; if not
- It's an error

We propose to retain the slightly more restrictive rule "never use aggregate initialization if a constructor is declared". Without this restriction, we would not be able to enforce invariants by defining constructors. Consequently, we consider a restriction necessary and get this modified basic rule:

- If a constructor is declared
  - Look for a sequence constructor and use it if we find a best one; if not
  - Look for a constructor (excluding sequence constructors) and use it if we find a best one; if not
  - It's an error
- If no constructor is declared
  - look to see if we can do traditional aggregate or built-in type initialization; if not
  - It's an error

This can (and should) be integrated into the overload resolution rules.

4.2 Sequence constructors

A sequence constructor is defined like this:

```
class C {
        C(initializer_list<int>); // construct from a sequence of ints
        // …
};
```

The **initializer_list** ("sequence initialize" or "sequence initializer") argument type indicates that the constructor is a sequence constructor. The type in <…> indicates the type of elements accepted. A sequence constructor is invoked for an array of values that can be accessed through the **initializer_list** argument. The **initializer_list** type offers three member functions to allow access to the sequence (for details see ???):

```
template<class E> class initializer_list {
        // representation (a pair of pointers or a pointer plus a length)
public:
        initializer_list(const E*, const E*); // from [first,last)
```

```
        initializer_list(const E*, int);  // from [first, first+length)

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

The three member functions provide STL-style (**begin(),end()**) access or "Fortran-style" (**first(),size()**) access. It is essential that the sequence is immutable: A sequence constructor cannot modify its input sequence. A sequence constructor might look like this:

```
template<class E> class vector {
        E* elem;
public:
        vector (initializer_list<E> s) // construct from a sequence of Es
        {
                reserve(s.size());
                uninitialized_fill(s.begin(),s.end(),elem);
        }
        // … as before …
};
```

Let's consider the examples above (Section 4) when **X** is **std::vector<double>**. For example:

```
std::vector<double> v = {1, 2, 3.14};                    // as initializer
```

That's easily done: **std::vector** has no sequence constructor (until we add the one above), so we try **{1, 2, 3.14}** as a set of arguments to other constructors, that is, we try **vector(1,2,3.14)**. That fails, so all of the examples fail to compile when **X** is **std::vector**.

Now add **vector(initializer_list<E>)** to **vector<E>** as shown above. Now, some (but not all) of the examples work when **X** is **vector<double>**. In each case, **{1, 2, 3.14}** is interpreted as a temporary constructed like this:

```
double temp[] = {double(1), double(2), 3.14 } ;
vector<double> tempv(temp,temp+sizeof(temp)/sizeof(double));
```

That is, the compiler constructs an array containing the initializers converted to the desired type. This array is read by **vector**'s sequence constructor, which copies the values from the array into its own data structure for elements. This implies that every use of **{1, 2, 3.14}** in a place that accepts an rvalue succeeds. Uses that require an lvalue fails.

Note that an **initializer_list** is a small object (probably two words), so passing it by value makes sense. Passing by value also simplifies inlining of **begin()** and **end()** and constant expression evaluation of **size()**.

4.3 The initializer list rewrite rule

A simple way of understanding initializer list is in terms of a rewrite rule. Given

```
void f(initializer_list<int>);
f({1,2.0,'3'});
```

The compiler lays down an array

```
int a[] = {int(1), int(2.0), int('3') };
```

And rewrites the call to

```
f(initializer_list<int>(a,3));
```

Assuming that **initializer_list** has been suitably declared and is in scope (§4.5.1), all is now well.

Similarly, given

```
X v = {1,2.0,'3'};
```

The compiler looks at **X** and assuming it finds a sequence constructor taking a **initializer_list<int>**, it lays down an array

```
int a[] = { int(1), int(2.0), int('3') };
```

And rewrites the definition to

```
X v(initializer_list<int>(a,3));
```

Thus from the point of view of the rest of the language an initializer list that is accepted by a sequence constructor is simply an invocation of the suitable constructor.

For the purpose of overloading, going from an initializer list to its **initializer_list** object counts as a built-in conversion (as opposed to a user-defined conversion), independently of what conversions were needed to generate the homogenous array.

4.4 Syntax

In the EWG there were strong support for the idea of the sequence constructor, but initially no consensus about the syntax needed to express it. There was a strong

preference for syntax to make the "special" nature of a sequence constructor explicit. This could be done by a special syntax

```
class X {
        // …
        X{}(const int*, const int* );  // construct from
                                       // a initializer list of ints
        // …
};
```

or a special (compiler recognized) argument type. For example:

```
class X {
        // …
        X(initializer_list <int>);      // construct from a initializer list of ints
        // …
};
```

We prefer the **X(initializer_list<int>)** design, because this "special compiler-recognized class name" approach

- Hides the representation of the object generated by the constructor and used by the sequence constructor. In particular, it does not expose pointers in a way that force teachers to introduce pointers before initializer lists.
- Is composable: We can use **initializer_list<initializer_list<int>>** to read a nested structure, such as **{ {1,2,3}, {3,4,5}, {6,7,8} }** without introducing a name for the inner element type.
- The **initializer_list** type can be used for any argument that can accept an initializer list. For example **int f(int, initializer_list<int>, int)** can accept calls such as **f(1, {2,3}, 4)**. This eliminates the need for variable argument lists (**…** arguments) in many (most?) places.

Finding a syntax for sequence constructors was harder – much harder – than finding its semantics. Here are some alternatives. Consider these possible ways of expressing a sequence constructor for a class **C<E>**:

```
template<Forward_iterator For> C<E>::C(For first, For last);
template<int N> C<E>::C(E(&)[N]);
C<E>::C(const E*, const E*);
C<E>::C{}(const E* first, const E* last);
C<E>::C(E … seq);
C<E>::C(... E seq);
C<E>::C(... initializer_list<T> seq);
C<E>::C(... E* seq);
C<E>::C ({}<E> seq);
C<E>::C(E{} seq);
```

```
C<E>::C(E seq{});
C<E>::C(E[*] seq);  // use sizeof to get number of elements
C<E>::C(E seq[*]);
C<E>::C(const E (&)[N]);   // N "magically" becomes the number of elements
```

And more. None provided the three advantages of the **initializer_list<E>** approach without other problems.

The hardest part of the design was probably to pick a name for the "special compiler recognized class name". Had we been designing C++ from scratch, we would probably have chosen **C::C(Sequence<int>)**. However, all the short good names have been taken (e.g., **Sequence**, **Range**, and **Seq**). Alternatives considered included **seqinit**, **seqref**, **seqaccess**, **seq_access seq_init**, and **Seq_init**. Our choice, **initializer_list,** seems the most descriptive and the least obnoxious name that has not already been widely used; we hope that the extravagant length is a protection. A quick check using google found only one occurrence with that capitalization, and that was in a Java program. We suggest **initializer_list** rather than **Initializer_list** because initial lower case is the norm in the standard library.

The name **initializer_list** is not a keyword. Rather it is assumed to be in namespace **std**, so you use it for something unrelated. For example:

```
int initializer_list = 7;
```

Doing so is would probably not be a good idea, though, once people get used to the standard (library) meaning.

4.5 The initializer_list class

Some obvious questions:

- Is **initializer_list** a keyword? No, but.
- Must I **#include** a header to use **initializer_list**? Yes, **#include<initializer_list>**
- Why don't we use a constructor that takes a general STL sequence?
- Why don't we use a general standard library class (e.g. **Range** or **Array**)?
- Why don't we use **T(&)[N]**?
- Can the **size()** be a constant expression? Yes.

More detailed answers and reasoning follows.

4.5.1 Keyword?

Is **initializer_list** a keyword? No, but it is a name in the standard library namespace and the compiler will use it. In particular, if you declare an argument of type **initializer_list<int>** and pass an initializer list to it, the compile will generate a call **std::initializer_list<int>(p,s)**, where **p** is the pointer to the start of the initializer list array and **s** is its number of elements. For example:

```
// won't compile unless std::initializer_list is in scope:

void f(std::initializer_list<int> s);

void g()
{
        int initializer_list;
        f({1,2,3});      // ok: use std::initializer_list
}
```

If you don't declare **initializer_list** (e.g., by including **<initializer_list>**), you get compile-time errors.

4.5.2 Include header?

Must I **#include** a header to use it? Yes, you must include **<initializer_list>**.

4.5.3 Why don't we use T(&)[N]?

Using "a notation" would save us a keyword (or the moral equivalent of a keyword: a frequently used name in **std**, such as **initializer_list**) and make it clear that a core language facility was used. Using **T(&)[N]** in particular would make it clear that we were dealing with a fixed length homogenous list (that is, an array).

We have an aesthetic problem with **T(&)[N]**, which will transform into an educational problem an myths about its rationale. However, the critical problem is that relying on this would turn every function that takes an initializer list as an argument into a template. For example, say we have a pair of functions that handles the case of one and two integer arguments:

```
void f(int);
void f(int,int);
```

If I find myself wanting to have a version that takes three integers and I suspect that I might need more such versions, I would write:

```
template<int N> void f(int (&)[N]);
```

Now each different argument list size generates a separate specialization. For example:

```
f({1});
f({1,2});
f({1,2,3});
```

Each calls a different function. This implies code replication, inability to use **f()** in a dynamically linked library, and problems with overloading: no use of **f()** as a virtual

function, for callbacks etc. That's too high a price to pay for solving a naming problem. This is especially so, as the fact that the length of the list is a constant is rarely particularly useful.

4.5.4 Why don't we use a constructor that takes a general STL sequence?

For example, for **vector**, why don't we just deem

```
template<class For> vector(For first, For last);
```

to be the sequence constructor for vector? First of all, it doesn't support the use of initializer lists for arbitrary arguments. For example

```
void f(int, int*, int*,int);
```

This would/should not be a good/sufficient clue that **f()** was willing to accept **f (1, {2,3,4,5,6},7)** as a call.

Secondly, the overload resolution rules can't work as described unless a sequence constructor is distinguishable from other constructors (and we can't eliminate current uses of these "iterator constructors"). It would also be hard to accept the "iterator constructor" above as a sequence constructor for ant **T** while rejecting a constructor taking two **int*** arguments as a sequence constructor. However,

```
X::X(int*,int*);
```

Just might be taking two unrelated integers, rather than a sequence. For example:

```
X a(new int(7), new int(9));
```

Finally, pairs of iterators are not trivially composable; for example, handling **{{1},{2,3},{3,4,5}}** would require an intermediate named type with a sequence constructor to handle the sub-sequences **{1}**, **{2,3}**, and **{4,5,6}**.

An earlier discussion of the initializer list problem used a distinguished form of the "iterator constructor" as the sequence constructor:

```
template<class For> vector{}(For first, For last);
```

However, there was unanimous agreement that the **initializer_list** approach was preferable.

4.5.5 General (std::) class?

Why don't we use a general standard library class (e.g. **vector**, **Range,** or **Array**)? The compiler-generated array that is the in-memory representation of the initializer list must

be immutable. If we could write to it, we could be back to "the good old days of Fortran where you could change the value of the literal **1** to **2**". For example, imagine that **initializer_list** allowed modification of the array:

```
int f()
{
        Odd_vector<int> v = { 1, 2, 3 };
        return v[0];
}
```

We would certainly expect **f()** always to return **1**. But consider

```
template<class T> class Odd_vector {
        // …
        Odd_vector(initializer_list<T> s)
        {
                // copy from the array into the vector
                *s.begin() += 1;        // illegal, but imagine what if
        }
}
```

Assuming (reasonably, according to the simple memory model presented in §4.3) that **{1,2,3}** defines a single array with initial value **{1,2,3}** repeatedly accessed by the sequence constructor, we can get

```
cout << f();    // write 1
cout << f();    // write 2
cout << f();    // write 3
…
```

As each invocation of the sequence constructor modifies that array's first element. It follows that we cannot accept anything as our accessor to the underlying array unless it can keep the array immutable.

4.5.6 Constant expression?

Can the **size()** be a constant expression? Maybe. At least, **size()** could be a constant expression where the use of **size()** is in the same translation unit as the initializer list and after it. Consider:

```
template<class T> class initializer_list {
        // …
        int size() const { /* … */ }
};

// …
```

```
initializer_list<int> s = {1,2,3};

char a[s.size()];          // ok: size is a constant expression
```

Clearly, there is enough information to deduce that **s.size()** is **3**. Equally clearly, making **s.size()** a constant expression requires a special rule. The proposal for generalizing constant expressions (N1920=05-0180) suggests that this can be made to work. We are not sure whether this is really important or just something people thought interesting. We need a solid use case.

4.6 Initializer lists and ordinary constructors

When a class has both a sequence constructor and an "ordinary" constructor, a question can arise about which to choose. The resolution outlined in §3.2 is that the sequence constructor is chosen if the initializer list can be considered as an array of elements of the type required by the sequence constructor (possibly after conversions of elements). If not, we try the elements of the list as arguments to the "ordinary" constructors. The former ("use the sequence constructor") matches the traditional use of initializer lists for arrays. The latter ("use an ordinary constructor") mirrors the traditional use of initializer lists for **struct**s (initializing constructor arguments rather than **struct** members). Consider a few examples:

```
vector<double> v1({1,2});  // v1 has two elements (values: double(1),double(2))
                           // use sequence constructor
vector<double> v2({1});    // v2 has one element (value: double(1))
                           // use sequence constructor
vector<double> v3({});     // v3 has no elements
                           // use sequence constructor
```

Since we can convert 1 and 2 to the **double**s required by **vector<double>**'s sequence constructor, the sequence constructor is used for v1 and v2. If we don't want that, we must use another form of initialization:

```
vector<double> v11(1,2);   // v11 has one element (value: double(1))
                           // use ordinary constructor
vector<double> v22(1);     // v22 has one element (value: double(), i.e. 0.0)
                           // use ordinary constructor
vector<double> v33();      // oops: v33 is a function!
```

If the type of the elements in the initializer list doesn't match what the sequence constructor requires, we use an ordinary constructor:

```
vector<double> v111({1,2,My_alloc}); // use ordinary constructor
                                     // can't convert My_alloc to double
```

```
vector<double> v222({v2.begin(),v2.end()});  // use ordinary constructor
                                             // to copy v2 into v4
                                    // (can't convert vector<double>::iterator to double)
```

Discussion: Should the initialization of **v111** and **v222** simply be errors? That is, should we reject the use of initializer list when there is no sequence constructor? For aggregates, initializer lists serve two purposes:

- Initialize homogeneous sequences (i.e. arrays)
- Initialize heterogeneous sequences (i.e. structs)

To provide a general initializer mechanism, we must preserve this dual use. To support user-defined types as well as built-in types and to provide a uniform syntax for initialization, we must somehow ensure that initializer lists can be used for both sequences (to initialize containers) and "ordinary objects". We can have that support disjoint: "if you have a sequence constructor, you can't use initializer lists for other constructors, but if you don't have a sequence constructor you can" or we can have it with "sequence constructor has priority" as suggested above. If we choose the "either/or" rule, we will not be able to use **{}** initialization uniformly; another initialization syntax will have to be used for classes with sequence constructors; importantly, we would not be able to use **{}** initialization for standard library containers. This would seriously weaken any effort to teach people to uniformly use a single initialization syntax (the **{}** notation). Furthermore, we would not be able to add a sequence constructor to a class that is already in use because that would break any **{}** initialization already used.

4.7 Initializer lists, aggregates, and built-in types

So what happens if a type has no constructors? We have three cases to consider: an array, a class without constructors, and non-composite built-in type (such as an **int**). First consider a type without constructors:

```
struct S { int a; double v; };
S s = { 1, 2.7 };
```

This has of course always worked and it still does. Its meaning is unchanged: initialize the members of **s** in declaration order with the elements from the initializer list in order, etc.

Arrays can also be initialized as ever. For example:

```
int d[] = { 1, 2, 3, 5, 8 };
```

What happens if we use an initializer list for a non-aggregate? Consider:

```
int a = { 2 };          // ok: a==2
                        // (as currently: there is a single value in the initializer list)
int b = { 2, 3 };       // error: two values in the initializer list
```

```
int c = {};          // ok: default initialization: c==int()
```

In line with our ideal of allowing initializer lists just about everywhere – and following existing rules – we can initialize a non-aggregate with an initializer list with 0 or 1 element. The empty initializer list gives value initialization. The reason to extend the use of initializer lists in this direction is to get a uniform mechanism for initialization. In particular, we don't have to worry about whether a type is implemented as a built-in or a user-defined type and we don't have to depart from the direct initialization to avoid the unfortunate syntax clash between **()** initialization and function declaration. For example:

```
X a = { v };
X b = { };
```

This works for every type **X** that can be initialized by a **v** and has a default constructor. The alternatives have well known problems:

```
X a = v;      // not direct initialization (e.g. consider a private copy constructor)
X b;          // different syntax needed (with context sensitive semantics!)
X c = X();    // different syntax, repeating the type name

X a2(v);      // use direct initialization
X b2();       // oops!
```

It appears that **{}** initialization is not just more general than the previous forms, but also less error prone.

We do not propose that surplus initializers be allowed:

```
int a = { 1, 2 };       // error no second element
struct S { int a; };
S s = { 1,2 };          // error no second element
```

Allowing such constructs would simply open the way for unnecessary errors.

> **Discussion:** Discussion: The standard currently says (12.6.1/2) that when an object is initialized with a brace-enclosed initializer list, elements are initialized through "copy-initialization" semantics. For uniformity and consistency of the initialization rules this should be changed to "direct-initialization" semantics. That will not change the semantics of current well-formed programs; it will make legal examples where the only problem was a private copy constructors.

5 Initializer list technicalities

As the saying goes "the devil is in the details", so let's consider a few technical details to try to make sure that we are not blindsided.

5.1 Sequence constructors

Can a class have more than one sequence constructor? Yes. An initializer list that would be a valid for two (or more) sequence constructors is ambiguous.

Can a sequence constructor be a template? Yes. Note that a "yes" here implies that more than one sequence constructor is possible.

Can a sequence constructor be invoked for a sequence that isn't an initializer list? No.

5.2 What really is an initializer list?

The simplest model is an array of values placed in memory by the compiler. That would make an initializer list a modifiable lvalue. It would also require that every initializer list be placed in memory and that if an initializer list appears 10 times than 10 copies must be present. We therefore propose that all initializer lists be rvalues. That will enable two optimizations:

- Identical initializer lists need at most be store once (though of course that optimization isn't required).
- An initializer list need not be stored at all. For example, **z=complex{1,2}** may simply generate two assignments to **z**.

The second optimization would require a clever compiler or literal constructors (§5).

Note that an initializer list that is to be read by a sequence constructor must be placed in an array. The element type is determined by the sequence constructor. Sometimes, it will be necessary to apply constructors to construct that array.

Initializer lists that are used for aggregates and argument lists can be heterogeneous and need rarely be stored in memory (separately from the copy stored as an array).

Must initializer lists contain only constants? No, variables are allowed (as in current initializer lists); we just use a lot of literals because that's the easiest in small examples.

Can we nest initializer lists? Yes (as in current initializer lists). For example:

```
vector<vector<int>> v = { {1,2,3}, {4,5,6}, {7,8,9} };        // a 3 by 3 matrix
```

A more interesting example might be

```
Map<string,int> m = { {"ardwark",91}, {"bison", 43} };
```

Assuming that map has a sequence constructor for from a **pair<string,int>**, this will work, correctly converting the literal strings to **strings**.

5.3 Ambiguities and deduction

An initializer list is simply a sequence of values. If it is considered to have a type, it is the list of its element types. For example, the type of {**1,2.0}** would be **{int,double}**. This implies that we can easily create examples that are – or at least appears to be – ambiguous. Consider:

```
class X  {
        X(initializer_list<int>);       // sequence constructor
        // …
};

class Y {
        Y(initializer_list<int>);       // sequence constructor
        // …
};

class Z {
        Z(int,int);      // not a sequence constructor
        // …
};

void f(X);
void f(Y);

void g(Y);
void g(Z);

f({1,2,3});      // error: ambiguous (f(X) and f(Y)?)
g({1,2,3});      // ok: g(Y)
g({1,2});        // ok: g(Y)  (note: not g(Z));
g({1});          // ok
```

The overload resolution rules are basically unchanged: try to mach all functions in scope and pick the best mach if there is a best match. What is new is a need to specify conversions used for a legal call using an initializer list so that it can be compared with other successful matches.

> **Discussion:** We resolve the **g({1,2})** call by preferring the sequence constructor in one class over an ordinary constructor in another class. The alternative would be to have the resolution depend on the number of elements in the initializer list.

How do we resolve ambiguity errors? By saying what we mean; in other words by stating our intended type of the initializer list:

```
f(X{1,2,3});     // ok: f(X)
g(Z{1,2});       // ok: g(Z)
```

Apart from using **{ }** rather than **( )**, it's the same idea as the current techniques of using explicit constructor calls.

> **Discussion:** We do not propose to allow an "unqualified initializer list" to be used as an initializer for a variable declared auto or a template argument. For example:
>
> ```
> auto x = {1, 2, 3.14};                  // error
> template<class T> void ff(T);
> ff({1, 2, 3.14});                       // error
> ```
>
> There is no strong reason not to allow this, but we don't want to propose a feature until we have a practical use in mind. If we wanted to allow this, we could simply "remember" the type of the initializer list and use it when the **auto** variable or template argument is used. In this case, the type of **x** would be **{int,int,double}** which can be converted into a named type when necessary. For example:
>
> ```
> auto x = {1, 2, 3.14};          // remember x' is a {int,int,double}
> vector<int> v = x;              // initialize v {1, 2, 3.14};
> g(x);                           // as above
> ```
>
> It's comforting to know that the concepts extend nicely even if we have no use for the extension.

5.4 Initializer lists and templates

Can an initializer list be used as a template argument? Consider:

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

f({});                          // error
f({1});
f({1,2,3,4,5,6});
f({1,2.0});                     // error
f(X{1,2.0});                    // ok: T is X
```

There is obviously no problem with the last call (provided **X{1,2.0}** itself is valid) because the template argument is an **X**. Since we are not introducing arbitrary lists of types (product types), we cannot deduce **T** to be **{int,double}** for **f({1,2.0})**, so that call is an error. Plain **{}** does not have a type, so **f({})** is also an error.

This leaves the homogeneous lists. Should **f({1}) and f({1,2,3,4,5,6})** be accepted? If so, with what meaning? If so, the answer must be that the deduced type, **T**, is **initializer_list<int>**. Unless someone comes up with at least one good use of this simple feature (a homogeneous list of elements of type **E** is deduced to be an

initializer_list<E>), we won't propose it and all the examples will be errors: No template argument can be deduced from an (unqualified) initializer list. One reason to be cautious here is that we can imagine someone getting confused about the possible interpretations of single-element lists. For example, could **f({1})** invoke **f<int>(1)**? No, that would be quite inconsistent.

5.5 C99 style initializers with casts

If we wanted to increase C99 compatibility, we could additionally accept the more verbose version:

```
f((X){1,2,3});  // ok: f(X)
g((Z){1,2});    // ok: g(Z)
```

This is not something we recommend. The C semantics require the initializer list to be an lvalue with weird results. Here is an example from the C99 standard [6.5.2.5 Compound literals]:

EXAMPLE 8 Each compound literal creates only a single object in a given scope:

```
struct s { int i; };
int f (void)
{
      struct s *p = 0, *q;
      int j = 0;
again:
      q = p, p = &((struct s){ j++ });
      if (j < 2) goto again;
      return p == q && q->i == 1;
}
```

The function `f()` always returns the value 1.

17 Note that if an iteration statement were used instead of an explicit `goto` and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have an indeterminate value, which would result in undefined behavior.

There is a danger that the "semi-compatible" syntax might become popular in C++ just as "the abomination" **f(void)**. Also, there would be subtle incompatibilities between the C99 definition of such as construct and any consistent C++ view (see N1509).

5.6 Refining the syntax

So far, we have used initializer lists after = in definitions (as always) and as function arguments. The aim is to allow an initializer list wherever an expression is allowed. In addition, we might consider leaving out the = in a declaration:

```
auto x1 = X{1,2};
X x2 = {1,2};
X x3{1,2};
```

```
X x4({1,2});
X x5(1,2);
```

These five declarations are equivalent (except for the name of the variables) and all variables get the same type (**X**) and value (**{1,2}**). Similarly, we can leave out the parentheses in an initializer after **new**:

```
X* p1 = new X({1,2});
X* p2 = new X{1,2};
```

It is never ideal to have several ways of saying something, but if we can't limit the syntactic diversity we can in this case at least reduce the semantics variation. We could eliminate these forms:

```
X x3{1,2};
X* p2 = new X{1,2};
```

However, since **X{1,2}** must exist as an expression, the absence of these two syntactic forms would cause confusion, and they are the least verbose forms. Note that **new X{1,2}** must be interpreted as "an **X** allocated on the free store initialized by **{1,2}**" rather than "**new** applied to the expression **X{1,2}**". This is equivalent to the current rule for **new X(1,2)**.

Note that if we add a sequence constructor to vector, each of these definitions will create a vector of one element with the value **7.0**:

```
vector<double> v1  = { 7 };
vector<double> v2 { 7 };
vector<double> v3 ({ 7 });

auto p1 = new vector<double>{ 7 };
auto p2 = new vector<double>({ 7 });
```

We don't propose a syntax for saying "this is a sequence: don't treat is as a constructor argument list". We don't see a need, because if you don't know anything about a type, you shouldn't try to tell it how to initialize itself. Similarly, we don't propose a syntax for saying "this is an aggregate initializer, don't use it for a class with constructors".

Discussion: we think that the most likely confusion and common error from the new syntax will (as with the old initialization syntax) be related to initializer lists with a single argument. Consider:

```
vector<double> v2 { 7 };
```

A naïve reader will have no way of knowing that this creates a **vector** of one **double** initialized to **7.0** and not a **vector** of seven **doubles**. Obviously, making the second interpretation the correct one would be even worse. Consider

```
vector<double> v1 { };      // a vector with no elements
vector<double> v2 { 7 };    // a vector with one element
                            // (not a vector with 7 elements)
vector<double> v3 { 7, 8 }; // a vector with two elements
```

We feel that this must work as stated. This also eliminated the possibility of making the initialization of **v2** ambiguous. Consequently, we consider the proposed design the best possible (at least of the ones we have seen so far).

6 Initializer lists in expressions

We have discussed initializer lists in the context of initialization. However, we could imagine them used elsewhere. Logically, an initializer list could appear in any place where and expression could. We would need a reason to prohibit that.

6.1 Assignments

Assignments and initializations are closely related. For example, there is no real implementation difference between them for built-in types. Consider:

```
X v = {1,2};
v = {3,4};
```

Having accepted the initialization, it would be hard to argue that the assignment was illegal. After all, we define **x=y** as (something like) **x.operator=(y)**. For some suitable type **X**, we could write the assignment as **v.operator={{3,4})** and have it work because now **{3,4}** is an initializer. Provided that there is no problem with the syntax, this example must be accepted.

6.2 General expressions

Consider more general uses of initializer lists. For example:

```
v = v+{3,4};
v = {6,7}+v;
```

When we consider operators as syntactic sugar for functions, we naturally consider the above equivalent to

```
v = operator+(v,{3,4});
v = operator+({6,7},v);
```

It is therefore natural to extend the use of initializer lists to expressions. We have not explored the grammar for this in detail and suggest that it should be explored. We see no obvious problems with this general use of initializer lists and suspect that people will expect it to work if the simpler uses work. In particular, the grammar will have to be explored.

6.3 Lists on the left-hand side

Whether we should allow lists on the right hand side of an assignment is a separate issue. For example:

```
{a,b} = x;
```

We make no proposal or recommendation about this. It is a separate question.

7 Casting

When a user-defined type is involved, we can define the meaning of C-style casting **(T)v** and functional style construction **T(v)** through constructors and conversion operators. However, we cannot change the meaning of a new-style cast and **T(v)** is by definition an old-style cast so its default meaning implies really nasty casts (reinterpret casts) for some built-in type combinations. For example, **int(p)** will convert a pointer **p** to an **int**. This leads to two common suggestions:

- Allow user-defined **static_cast**, etc.
- Default **T(v)** to mean **static_cast<T>(v)** rather than **(T)v**.

The two suggestions are related because often the reason for wishing **T(v)** to mean **static_cast<T>(v)** is to be able to define it as a range-checked operation for some built-in type **T**.

We have also heard the suggestion that **T(v)** should be “proper construction” and thus not allow narrowing conversions (e.g. **char(123456)**). However, the functional notation is used to be explicit about narrowing, so banning narrowing by default would be too radical.

We don’t propose to allow overloading of the new-style casts. If you want a different cast, you can define one using the same notational pattern, such as **lexical_cast<T>(v)**. The **T(v)** problem is worse: it basically defeats attempts to make casting safer and more visible. It also, takes the ideal syntax for the least desirable semantics. Unfortunately, it appears to be widely used for “nasty casts” (in correct code). For example:

```
typedef char* Pchar;
int i;
// …
```

Pchar p = Pchar(i); // would usually require an obviously nasty reinterpret_cast

Basically, this means that we cannot change the meaning of T(v). This is really nasty for several reasons:

- Consider:

  Pchar p = Pchar(i);

  This looks innocent, but hides nasty code.

- When we write generic code, there is no other general syntax for construction:

```
template<class T, class V> void f(T t, V v)
{
        // …
        X = T(v);       // construct (but for some types it casts)
        // …
}
```

We consider that a serious problem. The **{ }** syntax can be used as a remedy:

T{v}

Means "(direct) initialize **v** to type **T"**. That is, **T(v)** will have the same value as the variable **x** after **T x{v}**. Note that if **T** has a sequence constructor, **T{v}** means "make a **T** with a single element **v**".

7.1 Can we ban narrowing for T{v}?

It is extremely tempting to outlaw narrowing in a **T{v}** cast. However, we can't do that by itself. We must maintain the uniformity of **{ }** initialization. After all, one of the main aims of generalizing initializer lists and encouraging their use is to address the problems with the diversity of meanings of other initialization notations. In particular, consider:

```
T{v}
T x{v};
T y = {v};
T a[] = {v};
```

The values of **T{v}**, **x**, **y**, and **a[0]** must be identical.

That is, to get **T{v}** as a "safe" cast, we would have to disallow narrowing in all such initialization. That's still very tempting because the amount of code affected will be "relatively minor". However, remember that a "relatively minor" fraction of hundreds of million lines of C++ code could easily be far too much. Given the advantages of

addressing the problem with narrowing we will explore this possibility. Please note that this proposal the ban narrowing for **{ }** initialization (only) is separate for the main proposal for dealing with initializer lists.

First note that banning narrowing conversions for **{ }** initialization cannot lead to "silent" change of meaning; it will simply cause previously legal C++ programs to be rejected by the compiler. For example:

```
char x = { 1 };                    // error: 1 is an int
char a[] = { 'a', 'b', 'c', 0 };   // error: 0 is an int
```

This problem could be remedied by requiring the compiler to verify that no narrowing actually occurs:

```
char x = { 69 };        // ok
char y = { 1234 };      // error (assuming 8-bit chars)
```

For initializers that are literals, that's trivial and some current compilers already warn. That's the rule we propose. Note that whether narrowing would occur (if allowed) is often implementation defined.

That leaves initializer lists where the initializers are variables, such as:

```
void f(int a, int b, int c)
{
        char x = { a };                 // error: a is an int
        char a[] = { a, b, c, 0 };      // error: a, b, c are ints
        // …
}
```

The proposal to ban narrowing is based on the conjecture that such cases are rare and has a high enough incidence of errors, especially portability errors, that the community would be willing to accept (not silent) errors.

7.1.1 Narrowing of function argument values

Consider

```
struct X {
        X(int);
};

X a(2.1);       // ok
X b = 2.1;      // ok
X c{2.1};       // error: narrowing
```

```
void f(X);
f(2.1);          // ok
f({2.1});        // error: narrowing
```

This would follow from a ban of narrowing where ever we use **{** … **}**. This is backwards in the sense that the default (no use of **{ }** in ordinary calls) is less safe than the "odd" use with **{** … **}**. However, not doing it that way would break a lot of code. Note that this resolution is consistent with the behavior of **{** … **}** vis a vis **explicit** constructors (§3.2).

7.1.2 History: why do we have the narrowing problem?

Are there any inherent benefits of implicit narrowing? Yes, consider:

```
void f(int i, double d)
{
        char c = i;
        int i2 = d;
        // …
}
```

This is shorter than equivalent using casts (C-style):

```
void f(int i, double d)
{
        char c = (char)i;
        int i2 = (int)d;
        // …
}
```

Or (C++ style):

```
void f(int i, double d)
{
        char c = static_cast<char>(i);
        int i2 = static_cast<int>(d);
        // …
}
```

Some implicit casts, such as **double->int** and **int->char**, have traditionally been consider a significant – even invaluable – notational convenience. Others, such as **double->char** and **int*->bool**, are widely considered embarrassments. When Bjarne once asked around in the Unix room why implicit narrowing had actually been allowed. Nobody argued that there were a fundamental technical reason, someone pointed out the obvious potential for errors and all agreed that the reason was simply historical: Dennis Ritchie added floating point before Steve Johnson added casts. Thus, the use of implicit narrowing was well established before explicit casting became an option.

Bjarne tried to ban implicit narrowing in “C with Classes” but found that a combination of existing practice (especially relating to the use of chars) and existing code made that infeasible. Cfront, however, stamped out the **double->int** conversions for early generations of C++ programmers by providing long, ugly, and non-suppressible warnings.

Please note that the suggestion to ban narrowing does not actually touch these common examples. It relies on explicit use of **{ }**.

8 Variadic templates

N1704 proposes a general and type safe method of passing both homogenous and heterogenous lists. Why don’t we just use that proposal?

The major reason is that N1704 is a proposal for templates. We do not want to require that every variadic function should be a template. Doing so would imply the problems of code replication and the problems with defining virtual functions and (other) callbacks.

In addition, we worry that the heavy use of templates might make the proposal unsuitable for long initializer lists. For example,

```
vector<int> v = { 1,2,3, …. 1001, 1002, 1003 };
```

Consequently, we are of the opinion that the proposals address different problems and this is not the place for a details discussion of variadic templates.

9 Acknowledgements

Obviously, much of this initializer list and constructor design came from earlier papers and discussions. The main papers are listed in §1.

10 Appendix: Suggested working paper changes

<<Incomplete pending further discussion of the proposal and design alternatives>>

Here are working paper changes for the main proposal and two subsidiary proposals. The two subsidiary proposals make sense only if the main proposal is accepted, but the main proposal does not depend on the subsidiary proposals.

10.1 Main proposal

We propose to allow initializer lists wherever an initializer can appear.

10.2 Narrowing proposal

We propose to ban narrowing conversions of values in initializer lists.

10.3 Syntax proposal

We propose to accept initializer lists as expressions.

10.4 Containers

We propose that each standard library container is provided with a sequence constructor for its element type.