Doc No: WG21/N2215=J16/07-0075
Date: 2007-03-08
Reply to: Bjarne Stroustrup
bs@cs.tamu.edu

Initializer lists (Rev. 3)

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, initialization by an arbitrarily long list of values of a specified type can be 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, we present two subsidiary proposals:
1. to allow the use of initializer lists as sub-expressions
2. to disallow narrowing in initializations using initializer lists.

The latter would put an end to many narrowing problems.

Suggested working paper text is an appendix. The WP text is deliberately kept with the design discussions to allow crosscheck and understanding of rationale.

Changes in Revision 3:
- The overload resolution is simplified (see §4.1). In particular, overload resolution of aggregate initialization and constructor initialization are integrated (see §???)
- The **initializer_list** class has been simplified, denying users the ability to create one for an independently created array
- WP wording for handling initializer lists is added (§10)
- WP wording for adding **initializer_list** and sequence constructors for containers to the standard library is added (§10.4)
- There are many clarifications of the presentation of problems and solutions

The simplifications and clarifications are partly a result of implementing the proposal.

1 Previous work

The direct ancestor to this paper is “Initializer Lists (Rev 2)” N2100=J06-0170, based on N1919==05-179, which in turn was based on N1890=05-0150 “Initialization and initializers.” The other parts of “the initialization puzzle” presented in N1890 were presented in companion papers, such as Gabriel Dos Reis and Bjarne Stroustrup’s “Generalized constant expressions” (N1920=05-0180). 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; apologies to those 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 **std::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 the 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 allow the 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. See §7
- A separate subsidiary proposal is to allow an initializer list as a sub-expression, for example, **x=y+{1,2}**. See §6.

The main proposal offers initializer lists as a uniform notation for all initialization with a single semantics for all cases. The main proposal breaks no legal ISO C++ program except those that use the proposed standard library name **initializer_list** in a way that could clash. The first subsidiary proposal breaks code where an initialization depends on narrowing conversion (see §7).

3 Uniform syntax and semantics

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 §7)
```

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

This proposal meets these goals for every use of initialization in the language except that it is still possible to disallow pass-by-value by disabling a copy constructor. For example:

```
X a= {v};        // direct initialization
void f(X);
f({v});          // copy initialization
```

Here the **X** value constructed for **v** for **a** and for **f()** will be the same for all types **X** and values **v**, but the by-value passing of that **X** to **f()** may be disallowed or **X**'s copy constructor may (perversely) not copy correctly.

Note that there are further initialization contexts:

```
X(v);                          // create a temporary
X f(int v) { return v; }       // return a value
void g(X); g(v);               // pass an argument
new X(v);                      // create object on free store
Y::Y(int v) :X(v), m(v) { }    // base and member initializers
throw v;                       // throw an exception
```

We propose to allow initializer lists there also. For example

```
X{v};                              // create a temporary
X f(int v) { return { v }; }       // return a value
void g(X); g({v});                 // pass an argument
new X{v};                          // create object on free store
Y::Y(int v) :X(v), m(v) { }        // base and member initializers
throw {v};                         // syntax error
```

In these cases, the proposal simply offers the possibility to use lists in addition to single values. This is most significant for array types where no initializer syntax previously existed. For example

```
new int[] {1, 2, 3};               // create array on free store
```

We disallowed **throw{v}** because – alone of all initialization contexts – we have no idea which type will be used to catch **v**; there could even be several different ones. We could allow **throw{v}** in cases where **v** is an single value or where it is a completely homogenous list (without any conversions or promotions). However, until we see a solid use case, we do not propose that.

Appendix C discusses the idea of reaching a uniform initialization style and semantics without introducing new syntax. We conclude that we must live with different meanings for the existing different initialization syntaxes. It is possible that we have missed a satisfactory solution to this puzzle, but having looked repeatedly over several years we haven't found one and we don't propose to spend more time on that line of thought.

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 lose 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 (will be an error)

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 (will be an error)

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}) { }     // or just :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 general rule of the use of initializer lists is:

1. If a constructor is declared
    1. If there is a sequence constructor that can be called for the initializer list
        1. If there is a unique best sequence constructor, use it

        2. Otherwise, it’s an error
    2. Otherwise, if there is a constructor (excluding sequence constructors)
        1. If there is a unique best constructor, use it
        2. Otherwise, it’s an error
    3. Otherwise, it’s an error
2. Otherwise
    1. If we can do traditional aggregate or built-in type initialization, do it
    2. Otherwise, it’s an error

We could consider an even simpler rule: use initializer lists only for sequence constructors, but that would leave the problem of non-uniform initialization semantics unaddressed. Appendix B discusses (and rejects) alternatives to the rules 1.1 and 1.2 (that controls the choice between sequence constructors and “ordinary constructors”).

The “that can be called clause” ensures that “ordinary constructors” can be called if no sequence constructor can be used. The “unique best sequence constructor” clause ensures that we don’t ignore sequence constructors if more than one could have been used (that situation is a real ambiguity).

4.2 Sequence constructors

A sequence constructor is defined for a class like this:

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

The **initializer_list** 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** is a standard library class that offers three member functions to allow access to the sequence:

```
template<class E> class initializer_list {
        // representation (probably a pair of pointers or a pointer plus a length)
        // constructed by compiler

        // implementation defined constructor
public:
        // allow uses:  [first,last) and [first, first+length)

        // default copy constructor and copy assignment
        // no destructor (or the default destructor, which would mean the same)

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

The **constexpr** specifier (N1980=0050) indicates that if an **initializer_list** object happens to be **constexpr**, the operations on it will be usable as constant expressions.

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 …
    };
```

Consider:

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

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, the example works. The initializer list **{1, 2, 3.14}** is interpreted as a temporary constructed like this:

```
    const double temp[] = {double(1), double(2), 3.14 } ;
    initializer_list<double> tmp(temp,sizeof(temp)/sizeof(double));
    vector<double> v(tmp);
```

That is, the compiler constructs an array containing the initializers converted to the desired type (here, **double**). This array is passed to **vector**'s sequence constructor as an **initializer_list**. The sequence constructor then copies the values from the array into its own data structure for elements.

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()**.

An **initializer_list**s will be created by the compiler, but can be copied by users. Think of it as a pair of pointers. See also §5.7.

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

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

And rewrites the call to

```
f(initializer_list<int>(a,3));          // rewritten to use initializer_list
```

Assuming that **initializer_list** is in scope (§4.5.1), all is now well.

In general, given

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

the compiler looks at **X** and if it finds a sequence constructor taking a **initializer_list<Y>**, it lays down an array

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

and rewrites the definition to

```
X v(initializer_list<Y>(a,3));          // rewritten to use initializer_list
```

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 standard conversion (as opposed to a user-defined conversion), independently of what conversions were needed to generate the homogenous array. See also §5.7.

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
        // …
};
```

Based on extensive discussions, 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
C<E>::C(const E[N]);        // N "magically" becomes the number of elements
```

And more. All has been seriously suggested by someone. 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 can 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]**?
- 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; 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 = 7;
        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 would transform into an educational problem and 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, we might have a simple function:

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

We might want to generalize this to deal with N integers:

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

Unfortunately, each different argument list size generates its own 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' we use T[N]?

That is, why don't we just treat **{ 1,2,3 }** as an array or at least say that an array can be initialized by an initializer list?
This suggestion has the problems we saw with **T(&)[N]** plus a few stemming from arrays central role in C and its close connection to pointers. For example:

```
void f(int*);
void f(int[]);
void f(int[10]);
```

This declares just one (non-overloaded) function. Breaking this equivalence would be break a lot of code. On the other hand allowing

```
void g(int*);
g({ 1, 2, 3 });
```

would open a new hole in the type system (how come we lost the size of the list?) and consider:

```
void f2(int [n]);
void f3(int[N]);
f2( {1,2,3});
f3( {1,2,3});
```

Here the semantics would presumably be different depending on whether **n** and **N** were previously defined and the semantics of the two calls would differ if **n** was undefined and **N** was a constant expression. Also, what would be the meaning if **n** was an **int** variable?

The rules for array decay, multidimensional arrays, etc. adds many opportunities for confusion.

4.5.5 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 should not be sufficient clue that **f()** was willing to accept **f (1, {2,3,4,5,6},7)** as a call. To avoid chaos, we need something more explicit.

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 odd to accept the "iterator constructor" above as a sequence constructor for any sequence of **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));
```

Also, 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}**.

4.5.6 General (std::) class?

Why don't we use a general standard library class (e.g. **vector**, **Range,** or **Array**)?

Fundamentally, because **initializer_list** has a privileged position in the overload resolution rules, we don't really want it to be a general container. In particular, we want

```
void f(vector<int>);
void f(list<int>);
void f(my_stl_style_container<int>);
// …
```

We want the containers to be fundamentally similar as far as language rules are concerned and interchangeable (where reasonable). This is only possible provided that **initializer_list** is not general and available to implement initializer list initialization for every general container.

The compiler-generated array that is the in-memory representation of the initializer list must be immutable. If not, we could be back to "the good old days of Fortran 2 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 {      // very odd
        // …
        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.

4.5.7 Constant expression?

Can the **size()** be a constant expression? Yes, but only when a use of **size()** is in the same translation unit as the initializer list and after it. Consider:

```
template<class T> class initializer_list {
        // …
        constexpr 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) shows how this would work. We are not sure whether this is really important or just something people thought interesting. However, it follows directly from the definition for **constexpr**. Similarly, the "in the same translation unit" restriction follows from the **constexpr** definition, which in turn simply reflects the underlying realities of separate compilation. For example:

```
// file 1:
        void f(initializer_list<int>);
        // …
        f({1,2,3});

// file 2:
        void f(initializer_list<int> s)
        {
                char a[s.size()];       // error: size is not a constant expression
                // …
        }
```

There simply isn't sufficient information in file2 to evaluate **s.size()** at compile time.

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 §4.1 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). Appendix B discusses the decision to give priority to sequence constructors over "ordinary constructors" in quite some detail.

4.6.1 Disambiguation

Occasionally, we'd like to say "this really is an initializer list, don't try to use it as a set of constructor arguments." Two ways of saying that falls out of the general rules:

```
initializer_list<int>{ 1,2 };   // qualify by initializer_list
{ 1, 2, }                       // use the otherwise redundant trailing comma
```

Consequently we do not propose a separate mechanism (such as a list prefix or suffix).

4.7 Initializer lists, aggregates, and built-in types

This section considers the interaction of the proposal with traditional initialization using initializer lists. Basically, no change is needed except making the semantics of aggregate member initialization direct initialization rather than copy initialization (making a few more examples legal; see §4.7.4).

4.7.1 Aggregate initialization

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. This is compatibility reason for not requiring initializer lists to be homogenous and for using initialize lists to be used both as elements for a container and as initializers for a non-container.

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.

We do *not* propose to prohibit specifying fewer initializers than members:

```
struct S { int a, b;};
S s2 = { 1, 2 };           // ok
S s1 = { 1 };              // ok: b becomes 0
S s0 = {  };               // ok: a and be become 0
```

Disallowing such constructs would break too much code, and it’s useful.

Consider a class without a user-defined constructor but with members that require construction:

```
struct S { string a, b; };
S ss2 = { “one”, “two” };
S ss1 = { “one” };
S ss0 = { };
S ss00;
```

This works in C++98. A class with a generated default constructor can co-exist with initializer list initialization. However, this would be an error:

```
struct SS {
        SS() { }
        string a, b;
};
SS ss0 = { };   // error:
```

We cannot use an initializer list to initialize a struct with user-defined constructor. We propose no changes to this. However, the effect of the proposal is that a user does not need to know whether the initialization is done by a constructor or not. We could transparently replace S by:

```
struct S {
        S() :a(), b() { /* do something */ }
        S(const string& s) :a(s), b() { /* do something */ }
        S(const string& s1, const string& s2) :a(s1), b(s2) { /* do something */ }
        string a, b;
};
```

4.7.2 Overloading and aggregates

Consider:

```
struct S { int x double d; };

void f(int[]);   // takes an int*
voidf(S);
void f(const vector<int>&);

f( { 1, 2 } );     // which f()? f(const<int>&)!
f( { 1 } );        // which f()? f(const<int>&)!
f( { 1, 2.0 } );   // which f()? f(const<int>&)!
```

Why don't we choose **f(int[])**? The answer is relatively simple: **f(int[])** is really **f(int*)** and an initializer list does not "decay" to a pointer. Consider:

```
int* p = { 1, 2 };       // error: initialization of scalar requires a single element
int x;
int* q = { &x };         // ok: q is &x; we said nothing about *q
int a[] = { 1, 2 };      // ok, no decay, a refers to two elements
```

We don't want to change that: "Do not meddle in the affairs of arrays; they are subtle and quick to anger." We note that if we did allow **f({1,2})** as a call to **f(int[])** in "the obvious manner" (it is not obvious to non-C(++) experts that this **f()** takes a pointer rather than an array), we would have added a type hole to the language because there would be no way within the type system for **f()** to determine the size of the array pointed to by its argument.

We conclude, we can never pass an initializer list to an array as an argument. An **initializer_list<T>** argument does that job much better. So, in the example above reduces to

```
struct S { int x double d; };
```

```
void f(S);
void f(const vector<int>&);

f( { 1, 2 } );     // which f()? f(const<int>&)!
f( { 1 } );        // which f()? f(const<int>&)!
f( { 1, 2.0 } );   // which f()? f(const<int>&)!
```

As usual, sequence constructors take priority when the **{ … }** syntax is used, so all three calls invokes **f(const vector<int>&)**. If you prefer **f(S)**, say so:

```
f( S{ 1, 2.0 } );          // call f(S)
```

4.7.3 References to arrays

A reference to an array could have become a tricky special case (though we choose not to make it a special case in any of the initializer list rules). Consider:

```
int (&r)[] = { 1, 2 };   // error

void f(int (&r)[]);
f( {1,2} );        // ok?

void f(const vector<int>&);
f( {1,2} );        // ok? which f?
```

It is currently a special rule that allows initializer lists for arrays (and structs). This special rule does not extend to references, not even references to arrays. We don't propose to change that: a reference to an array is not an aggregate. It a (sort of) scalar.

Now consider templates:

```
template<int N> void f(int(&r)[N]);
int a[] = { 1, 2 };
f(a);     // ok
```

This currently works and **N** is deduced to **2**. So, should we be able to eliminate the mention of the array and write:

```
f( {1, 2} );       // ???
f( {1, 2.0 } );    // ???
```

The answers depend on the questions "what is the type of an initializer list?" and "what is the type deduced for an initializer list?" These questions are discussed in §5.4 and the conclusion is that **{1,2.0}** does not have a type independently of context and that **{1,2}** can be considered an **initializer_list<int>**. So, unless we make a special rule for

initializer_list<int> to decay to **int[N]** or make the type of **{1,2}** context dependent, we get

```
f( {1, 2} );        // error
f( {1, 2.0 } );     // error
```

We propose to leave it at that. Furthermore, we don't want to mess with the general rule that gives a function taking an **initializer_list** priority over functions taking other types.

4.7.4 Other uses of arrays

Where an array does not decay, we can use **{…}** initialization with exactly the usual semantics. For example

```
struct S {
        int a[ 45];
        S() : a{} { }     // initialize all 45 ints to 0
};
```

4.7.5 Aggregate initialization semantics

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 except that it will:

- make legal examples where the only problem was a private copy constructor
- include explicit constructors into overload resolution (see §12.3) allowing it to find a better match.

Such examples are rare in well-designed programs

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 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 then 10 copies must be present. So, we propose that all initializer lists be rvalues. That enables 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**.

- An initializer list that consists exclusively of constant expressions can be stored in read-only memory.

The second optimization would require a clever compiler or literal constructors (**constexpr**).

Initializer lists that are used for aggregates and argument lists can be heterogeneous and need rarely be stored in memory.

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.

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**.

5.2 Sequence constructors

Can a class have more than one sequence constructor? Yes. For example:

```
Class V {       // poor design that's asking for trouble
        V(initializer_list<int>);
        V(initializer_list<double>);
        V(int,double);
        // …
};

V v = { 1, 2.1 };        // error: ambiguous
```

In other words, we don't look further after finding an ambiguity among sequence constructors (only if no sequence constructor matches).

Can a sequence constructor be a template?

Can a sequence constructor be invoked for a sequence that isn't an initializer list? No. For example, there is no way that **f(1,2,3)** can invoke a sequence constructor for an argument type the way **f({1,2,3})** can.

Can we avoid considering ambiguities among sequence constructors by banning multiple sequence constructors for a single class? No. Consider:

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

f( {1,2.0} );      // f of list of ints or f of list of doubles?
```

So we need to formulate rules defining "ambiguity" for initializer lists.

5.3 Ambiguities and deduction

An initializer list is simply a sequence of values. If we considered it to have a type, it is would 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. We can create ambiguities among sequence constructors of a single class:

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

X x1 = { 1, 2.0 };        // error: ambiguous
X x2 = { 1, 2 };          // X(initializer_list<int>);
X x3 = { 1.0, 2.0 };      // X(initializer_list<double>);
```

The resolution rule for sequence constructors is

- if an initializer list has at least one element and every element is of the same type (before any conversions) **X** then the list has type **initializer_list<X>**
- Otherwise, the initializer list is not considered to have a type and can only be used as an initializer to an object with a single sequence constructor (i.e. cannot take part in overload resolution among **initializer_list**s). In that case every element must be convertible to the element type of that sequence constructor.

For example:

```
class X2 {
        X2(initializer_list<double>);           // sequence constructor
        // ... no other sequence constructor …
};

X2 a{ 1.0, 1, 'a'};        // ok: can convert 1 and 'a' to double
X2 b{ 1.0, "seven" }; // error: no conversion from string to double
X2 c{ };                   // ok: only one possible meaning for { }
```

and

```
class X3  {
        X3(initializer_list<double>);          // sequence constructor
        X3(initializer_list<int>);      // sequence constructor
        // ... no other sequence constructor …
};

X3 a{ 1.0, 1, 'a'};        // error: ambiguous
X3 b{ 1.0, "seven" }; // error: no conversion from string to double
X3 c{ };                   // error: ambiguous
```

In particular, there is no rule saying that some element conversions are considered better than others for **initializer_list**s elements. For example:

```
X3 a{ 1.0, 1 }; // error: ambiguous
```

We do *not* resolve that by preferring the promotion to 1.0 over the conversion to 1. Doing so would lead to extremely brittle and non-obvious resolutions in real use. For example:

```
X3 b{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1.1, 1, 1, 1, 1,  };      // huh?
```

Once we have found the best match (if any) for a single class, we can also have ambiguities among sequence constructors for different classes:

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

class Y {
        Y(initializer_list<double>); // 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 match if there is one. Note that

- all sequence initializer matches are equal so if there is more than one such match the call is ambiguous
- a sequence constructor match in one class still takes priority over a match to an ordinary constructor in another class.

Basically, the rule is (still) that we prefer sequence constructors over “other constructors.”

When an ambiguity is found, we need a way to resolve it. How do we resolve ambiguity errors from an initializer list? 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 a general 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.

What we do propose is the simpler rule that an initializer list has the type initializer_list<T> when every element if of the same type T before any conversions. For example:

```
auto xx = { 1,2,3};             // ok: xx is an initializer_list<int>
auto yy = { '1', '2', '3' };    //ok: yy is an initializer_list<char>
int* p;
int a[10];
auto zz = { p, a };             // ok: zz is an initializer_list<int*>
```

We accept array to pointer decay as "not a conversion".

```
auto pp = { p, 0 };             // error: 0 is an int
auto qq = { p, nullptr };       // error: nullptr to int* is a conversion
```

5.4 Initializer lists and templates

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

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

f({ });                           // error
f({1.0, 2.0});                 // ok: E is double
f({1,2,3,4,5,6});                // ok: E is int
f({1,2.0});                       // error
f(initializer_list<X>{1,2.0});   // ok: E is X
```

This follows from existing rules. Now consider:

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

f({ });                           // error
f({1.0, 2.0});           // T is initializer_list<double>
f({1,2,3,4,5,6});          // T is initializer_list<int>
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 calls f**({1}) and f({1,2,3,4,5,6})**. Initializer lists that are homogenous without any conversions have a type, so we deduce that type.

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) in C99
g((Z){1,2});    // ok: g(Z) in C99
```

This is not something we propose. 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)** did. Also, there would be subtle incompatibilities between the C99 definition of such as construct and any consistent C++ view (see N1509).

Using the **X{x,y}** syntax rather than the **(X){x,y}** syntax will require people to introduce a **typedef** if they want to resolve to a built-in type such as **char***. We think that is preferable to introducing an additional syntax with a semantics that subtly differs from C’s.

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 propose to allow the programmer to leave 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 **std::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 special syntax for saying "this is a sequence: don't treat is as a constructor argument list". The general resolution mechanism for resolving initializer list ambiguities will do. For example:

```
vector<double> v3 (initializer_list<int>{ 7 }); // redundant qualification
```

Also, a redundant terminating comma is allowed, so this is also explicitly an initializer list:

```
vector<double> v4 ( { 7, });        // redundant qualification
```

However, such qualification is redundant and should never be needed; see Appendix B.

> **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 very few (0, 1, or 2) arguments. 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> v0 { };        // a vector with no elements
> vector<double> v1 { 7 };      // a vector with one element
>                               // (not a vector with 7 elements initialized to 0)
> vector<double> v2 { 7, 8 };   // a vector with two elements
>                               // (not a vector seven elements initialized to 8)
> vector<double> v3 { 7, 8, 9 };        // a vector with three elements
> ```
>
> We feel that this must work as stated. See Appendix B for a detailed discussion of design choices in this area.

So, do we need a way of saying “these arguments are not an initializer list?” Well, yes, and we have several. For example:

```
vector<double> vv0;           // a vector with no elements
vector<double> vv1(7);        // a vector with 7 elements initialized to 0
vector<double> vv2(7, 8);     // a vector seven elements initialized to 8
vector<double> vv3(7, 8, 9); // error: vector has not 3-argument constructor
```

Falling back on “old syntax” is not ideal because (as shown above) it is not a single uniform disambiguation mechanism. However, there doesn’t seem to be sufficient reason to introduce a new construct nor do we know of an obviously elegant such construct.

5.7 Lifetime of initializer lists

In §4.4, we explained the initialization of a container **X** like this:

> In general, given
>
> ```
> X v = {1, 2.0, '3'};
> ```
>
> the compiler looks at **X** and if it finds a sequence constructor taking a **initializer_list<Y>**, it lays down an array
>
> ```
> const Y a[] = { Y(1), Y(2.0), Y('3') };
> ```

and rewrites the definition to

```
        X v(initializer_list<Y>(a,3));          // rewritten to use initializer_list
```

Now, where it that array "**a**" allocated? and why? What is the lifetime of the array and the elements in it? Consider

```
struct Y {              // give Y a destructor to emphasize lifetime issues
        Y(double);
        ~Y();
};

struct X {
        X(initializer_list<Y>);
        ~X();
};

X v1 = { 1, 2.0, '3' };

void f(int a)
{
        Y y = {2.0};
        X  v2 = { 1, y, a };
}
```

For **v1**, we can obviously place the array for its **initializer_list** in static storage, though we have defined **Y** so that dynamic initialization (and destruction) is required. For many – hopefully the vast majority of lists – dynamic initialization and destruction are not necessary (and the array can be placed in some unwritable memory, such as with the code or in ROM).

For **v2**, the lifetime of the array for its **initializer_list** is the function (specifically, the block in which **v2** is defined). We must construct the elements of that array at the point of the initialization and destroy the elements at the end of **v2**'s scope. Obviously, for simpler cases where dynamic initialization and destruction are not needed the compiler might prefer to allocate the array in static storage.

This follows the usual rules for the lifetime of objects and the point of initialization. It is also consistent with the current rules for initializer lists for aggregates.

One implication is that an initializer_list is "pointer like" in that it behaves like a pointer in respect to the underlying array. For example:

```
int * f(int a)
{
        int* p = &a;
        return p;       // bug waiting to happen
}

initializer_list<int> g(int a, int b, int c)
{
        initializer_list<int> v = { a, b, c };
        return v;       // bug waiting to happen
}
```

It actually takes a minor amount of ingenuity to misuse an **initializer_list** this way. In particular, variables of type **initializer_list** are going to be rare.

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. We don't see a problem with the syntax, so this example should 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. There are many uses where initializer lists combined with operators is a “natural” notation. However, it is not trivial to write a LR(1) grammar that allows arbitrary use of initializer lists. A block also starts with a **{** so allowing an initializer list as the first (leftmost) entity of an expression would lead to chaos in the grammar.

It is trivial to allow initilalizer lists as the right-hand operand of binary operators, in subscripts, and similar isolated parts of the grammar. The real problem is to allow **;a={1,2}+b;** as an assignment-statement without also allowing **;{1,2}+b;**. We suspect that allowing initializer lists as right-hand, but nor as left-hand arguments to most operators is too much of a kludge, so we try to express a more general rule. The current grammar is

assignment-expression:
conditional-expression
logical-or-expression assignment-operator assignment-expression
throw-expression

We can get what we want by essentially cloning the expression grammar

assignment-expression:
conditional-expression
logical-or-expression assignment-operator assignment-expression
logical-or-expression assignment-operator list-expression
throw-expression

Where a *list-expression* is just like an *assignment-expression* except that a *list-expression*’s version of *multiplicative-expression* allows an initializer list.

We would like to postpone a more concrete proposal until after a discussion of this solution and any alternatives.

6.3 Function calls

Consider:

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

Why two pairs of parentheses? Why not just use one:

```
f(1,2,3);       // usual parentheses with new semantics
f{1,2,3};       // new parentheses with new semantics
```

Without major changes this wouldn’t make sense. The **{…}** syntax specifies arguments for creating a single object (here a, **vector**) whereas in general, **(…)** specifies initializers for a sequence of objects. The first alternative (old **f(1,2,3)** syntax for new list semantics) would lead to a confusing mess, and we don't see how we could craft rules that don't change the meaning of existing programs. The second alternative (**f{1,2}** as a call of **f** with a single object initialized by **{1,2}**) is more tempting. Obviously, there would not be compatibility problems related to the semantics. However, we don't see sufficient gain so we don't propose this.

6.4 Subscripts

Subscripts are another restricted context where initializer lists can be used without technical problems. For example:

```
m[{1,2}] = 8;
```

We are not fully convinced that this is useful, but we have no reason to prohibit it and others have suggested that it might be useful to subscript using tuples.

6.5 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 completely separate question. Obviously, if we allowed that, "initializer lists" on the left-hand side would have to contain lvalues.

Sneaky attempts to modify an initializer list without an assignment operator are caught by the rule that makes initializer lists non-modifiable (§???). For example:

```
operator=({a,b},x);    // ok, but
```

You can write the call, but there is no way to define that operator=() to modify the elements of the **initializer_list** created for **{a,b}**.

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 (incl. **reinterpret_cast**) 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 even though we like the idea, we consider banning narrowing by default would too radical (Bjarne should have done that in 1983).

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 unfortunate 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)
  {
          T x;
          // …
          x = T(v);        // construct (but for some types it is a cast)
          // …
  }
  ```

We consider that a serious problem. The **{ }** syntax can be used as a remedy: **T{v}** means "construct a **T** given the initializer **v**". 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**".

So, we get: **T{v}** means

- for built-in types (including structs without constructors and arrays) we get a (temporary) object with the same value as **x** in **T x = { v };**
- for classes with constructors we get a (temporary) object with the same value as **x** in **T x(v);**

Note that we can affect/define the meaning of **T{x}** by defining an explicit conversion operator from **v**'s type to **T** (Goldthwaite N1592).

7.1 Can we ban narrowing for T{v}?

It is extremely tempting to outlaw narrowing in **T{v}**. However, we can't do that by itself. We must maintain the uniformity of **{ }** initialization. After all, one of the two 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 or we have lost uniformity.

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 can 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 these (non-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.

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 considered 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 for initialization using **{}** (unfortunately) does not touch these very common examples. It relies on explicit use of **{}**.

8 Variadic templates

N1704 proposes a general and type safe method of passing both homogenous and heterogeneous lists. Why don't we just use that proposal? Because, these proposals address different problems.

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.

The heavy use of templates might make the proposal unsuitable for long initializer lists. For example, consider adding a variadic constructor to **vector**:

```
vector<int> v(1,2,3, …. 1001, 1002, 1003);
```

Also, the problem of disambiguation between lists and "ordinary constructor argument" would re-emerge without any syntax to help:

```
vector<int> v(1,2);    // list or one element with the value 2?
```

These proposals address different problems.

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 A: Suggested working paper changes

Here are draft 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.

Our general strategy is to specify what's new and then fall back on existing rules whenever the syntax or semantics is unchanged. Only programs that define something called **initializer_list** can be affected by the main proposal (§10.1) and the general syntax proposal (§10.3); for a truly obscure exception to this rule see §12.3. Only programs that rely on narrowing conversions of values presented in { … } initializer lists are affected by the narrowing proposal (§10.2).

(We use paragraphs in simple parentheses (like this paragraph) to embed comments to help the reader of the proposal text. These comments are not part of the suggested WP text.)

10.1 Main proposal

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

10.1.1 Grammar changes

We propose to allow **{ … }** initializer lists for every initialization context (incl. the right-hand side of user-defined assignments and the subscript of user-defined subscripts).

(Note that the grammar changes are less than for the general syntax proposal (§10.3); if that proposal is accepted, this section will not apply).

In 8.5 [dcl.init] and A.7 [gram.decl], change

> *initializer*:
>> = *initializer-clause*
>> (*expression-list*)
>
> *initializer-clause*:
>> *assignment-expression*
>> { *initializer-list* ,$_{opt}$ }
>> { }

to

> *initializer*:
>> = *initializer-clause*
>> *ilist*
>> (*expression-list*)
>
> *initializer-clause*:
>> *assignment-expression*
>> *ilist*
>> *qilist*
>
> *ilist*:
>> { *initializer-list* ,$_{opt}$ }
>> { }
>
> *qilist*:
>> *tname* { *initializer-list* ,$_{opt}$ }
>> *tname* { }
>
> *tname*:
>> ::$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
>> ::*opt nested-name-specifier* template *simple-template-id*
>> char
>> wchar_t
>> bool
>> short
>> int
>> long
>> signed
>> unsigned
>> float
>> double

(This grammar works because before these changes { is never allowed after an identifier in the grammar, but see Appendix D. Note that *tname* is just *simple-type-specifier* with auto and void removed; a slight refactoring of the grammar would remove this redundancy.)

In 6.6 [stmt.jump] and A.5 Statements [gram.stmt], change

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

to

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

In 5.3.4 [expr.new] and A.4 Expressions [gram.expr], change

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

to

> *new-initializer*:
>> (*expression-list*$_{opt}$)
>> *ilist*

In 5.2 [expr.post] and A.4 Expressions [gram.expr], change

> *postfix-expression*:
>> …
>> *postfix-expression* [*expression*]
>> *postfix-expression* (*expression-list*$_{opt}$)
>> *simple-type-specifier* (*expression-list*$_{opt}$)
>> *typename-specifier* (*expression-list*$_{opt}$)
>> …

to

> *postfix-expression*:
> …
>> *postfix-expression* [*expression*]

postfix-expression [*init-list*]
postfix-expression (*expression-list*$_{opt}$)
postfix-expression (*init-list*)
simple-type-specifier (*expression-list*$_{opt}$)
simple-type-specifier (*init-list*)
typename-specifier (*expression-list*$_{opt}$)
typename-specifier (*init-list*)
…

In 12.6.2 [class.base.init] and A.10 Special member functions [gram.special], change

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

to

mem-initializer:
mem-initializer-id (*expression-list*$_{opt}$)
mem-initializer-id (*init-list*)
mem-initializer-id ilist

In 5.17 [expr.ass] and A.4 Expressions [gram.expr], change

assignment-expression:
conditional-expression
logical-or-expression assignment-operator assignment-expression
throw-expression

to

assignment-expression:
conditional-expression
logical-or-expression assignment-operator assignment-expression
logical-or-expression assignment-operator init-list
throw-expression

10.1.2 Initializers

In 8.5 [dcl.init], remove the mentions initialization of a scalar with a **{ v }** initializer list:

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

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

8.5.4 List-initialization [dcl.list.list]

This section defines the rules for selecting the meaning of an initialization using the *init-list* notation.

list-initialization is initialization of an object with an *init-list* (consisting of zero or more comma-separated expressions in curly braces; 8.5 [dcl.init]).

[example
```
        int a = {1};
        complex<double> z{1,2};
        new vector<string>{"once", "upon", "a", "time"}; // 4 string elements
        f( {"Nicholas", "Annemarie"} );     // pass list of two elements
        return { "Norah" };  // return list of one element
```
example]

list-initialization can be used as

- the initializer in a variable definition (8.5 [dcl.init])
- the initializer in a new expression (5.3.4 [expr.new])
- in a return expression (6.6 [stmt.jump])
- as a function argument (5.2 [expr.post])
- as an argument to an explicitly invoked constructor (5.2 [expr.post])
- as a base-or-member initializer (12.6.2 [class.base.init])

[comment

That is, an initializer-list can be used to specify the initial value in every initialization. In general, the state (value) of the resulting object involves not just the initializer-list, but also conversions and constructors. The state (value) of the resulting object is the same independently of how the object was created. For example, **T x{1,2};** gives the same value for **x** of type **T** as **T{1,2}** and **new T{1,2}** do.

comment]

An initializer list used as the initializer for an object of a scalar type (3.9 [basic.types]) must contain at most one element.

The type of a *qilist* **X{** … **}** is **X**.

The type of an unqualified *init-list* **{e1,…en}** can be considered a **std::initializer_list<E>** (for template argument deduction and any other use of a type) if the list of elements is non-empty and the type of every element of the initializer list (**e1** … **en**) is the same before any promotions or conversions. Array-to-pointer conversion (4.2 [conv.array]) is not counted as a conversion. If an unqualified *init-list* doesn't have such a completely homogenous element type it is considered not to have a type independently of its use.

[example

```
template<class T> void f(T);
f({});            // error: cannot deduce type of empty initializer list
f({1,2,3});       // ok: T is initializer_list<int>
f({1,2,3,4.0});   // error: the list is not homogenous without conversions
int a [10];
int p = a;
f({p,a});         // ok: array decay accepted
f({p,0});         // error: the list is not homogenous without conversions
initializer_list<int> v1 = { 1, 2, 3 };        // ok
initializer_list<int> v2 = { 1, 2, 3, 4.0 };   // error
```

example]

For many initializations we uniquely know the type of the object to be initialized; in other cases, overload resolution is required to choose among alternatives.

[example

```
int a = {1};      // initialize an int
complex<double> z{1,2};     // initialize a complex<double>
new vector<string>{"the", "Hobbit" }; // initialize a vector<string>
double{1}         // initialize a double

void f(int);
void f(complex<double>);
f({1});           // initialize an int (and call f(int))
                  // or a complex<double> (and call f(complex<double>))?
```

example]

A sequence constructor is a constructor taking a single argument of type **std::initializer_list<E>** for some type **E** (ref <<definition of initializer_list in the library>>).

It there is only one possible type **T** for the object to be initialized by an *init-list* the resolution is as follows:

1. If **T** has a constructor
   a. If there is a sequence constructor that can be called for the initializer list
      i. If there is a unique best sequence constructor, use it
      ii. Otherwise, it’s an error
   b. Otherwise, if there is a constructor (excluding sequence constructors)
      i. If there is a unique best constructor, use it
      ii. Otherwise, it’s an error
   c. Otherwise, it’s an error
2. Otherwise

   a. If we can do traditional aggregate or built-in type initialization, do it
   b. Otherwise, it's an error

[comment
This is the initialization used in cases, such as **T a{v};** and **return {v};**
comment]

If there are two or more possible types **T1** … **Tn** for the object to be initialized by an *init-list* the resolution is as follows:

For each type **Ti** find the best match (if any) according to the algorithm for initializing an object of known type with an initializer list (above).

1. If exactly one **Ti** has a best match, use it
2. Otherwise, if exactly one **Ti** has a best mach for a sequence constructor, use it
3. Otherwise, if two or more **Ti** have best matches for a sequence constructor, its an error
4. Otherwise, choose a best match according to the usual overload resolution rules (13.3 [over.match]) as modified to deal with overloading involving aggregates.

[comment
This is the initialization used in cases, such as in a call **f({v});** where **f** is overloaded and in a declaration **T x{v};** where **T** has overloaded constructors.
comment]

8.5.4.1 List initialization semantics [dcl.init.list.sem]

When an *init-list* **{ e1, … en }** is used as an initializer and passed as an argument to a **std::initializer_list<E>**, the **std::initializer_list<E>** object passed is constructed as if the compiler allocated an array **E[N]** where **N** is the number of elements in the *init-list* (and where **E** is the element type deduced or specified for the elements). Each element of that array will be the corresponding element of the *init-list* converted to E. The **begin()** and **end()** for that **initializer_list<E>** will refer to the first and one-past-the-last of the array, and the **size()** will be the number of elements.

[example
Consider:
```
void f(initializer_list<double> v);
f({ 1,2,3 });
```
The call will be implemented in a way equivalent to this:
```
double __a[double{1}, double{2}, double{3})];
f(std::initializer_list<double>(__a, __a+3);
```

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

The lifetime of that array (and its elements) is identical to that of a temporary created in the same place as the initializer list. [example

```
typedef std::complex<double> cmplx;
vector<cplx> v1 = { 1, 2, 3 };
vector<cplx> v2(3,cmplx(1));

void g(const vector<cmplx>&);

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

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

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

When an *init-list* **{ e1, … en }** is used as an initializer and passed as arguments to a constructor that is not a sequence constructor, the semantics is exactly as if the constructor had been called directly. [example

```
T a = { x, y };  // means T a(x,y);
void f(T);
f( { x, y} );       // means f(T(x,y))
```

example]

10.1.3 Assignments

We propose to accept initializer lists as the right-hand operand of an assignment (see grammar in §10.1.1):

In 5.17 [expr.ass], add

An *init-list* may appear on the right-hand side of

- an assignment to a scalar, in which can the initializer list must have at most a single element. The meaning of **x={v}** is that of **x=decltype(x)(v)** The meaning of **x={}** is **x=decltype(x)()**.
- an assignment defined by a user-defined assignment operator, in which case the meaning is define by the initialization rules for that operator function's argument.

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

[comment
The reason for converting the value to the target type before assigning is to ensure that the resulting value is identical to the one you would have gotten by initialization
comment]

10.1.4 Subscripting

We propose to accept initializer lists as subscripts for types where the subscript operator is user defined.

In 5.2.1 [expr.sub], add:

An *init-list* may appear as a subscript for a user-defined operator[]. In that case, the init-list is treated as the initializer for the subscript argument of the operator[].
[example
```
        struct X {
                Z operator[](initializer_list<int>);
        };
        X x;
        x[{1,2,3}] = 7;            // ok: meaning x,operator[]({1,2,3})
        int a[10];
        a[{1,2,3}] = 7;            // error: built in subscripting
```
example]

(We have provided no specific text for the base-and-initializer, return, and throw uses of initializer lists because we think that the rules follow from the existing text and the text provided for initialization; maybe a few examples would be an idea?).

10.2 Narrowing proposal

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

Add to the end of 8.5.4.1 [dcl.init.list.sem]

A program that attempts to narrow in an initialization using an *init-list* is ill formed. This means that such initialization does not convert

- from a floating-point-type to an type that is not a floating point type

- from **long double** to **double** or from **double** to **float**
- from an **long long** to **long**, from **long** to **int**, from **int** to **short**, from **short** to **char**, from **char** to **bool**, or the equivalent conversions for unsigned types
- from an unsigned to a signed type
- from a signed to and unsigned type

or any combination of these conversions, except in cases where the initializer is a constant expression and the actual value will fit into the target object; that is, where **decltype(x)(T{x})==decltype(x)(x)**. [example

```
int x = 999;            //  x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x;    // ok, might narrow (in this case, it does narrow)
char c2{x};     // error, might narrow
char c3{y};     // error: narrows
char c3{z};     // ok, no narrowing needed
unsigned char uc1= {5};      // ok: no narrowing needed
unsigned char uc2 = {-1}     // error: narrows
unsigned int ui1 = {-1}      // error: narrows
signed int si1 = { (signed int)-1 };    // error: narrows
```

end example]

[comment
Our basic definition of narrowing (expressed using **decltype**) is that a conversion from **T** to **T2** is narrowing unless we can guarantee that

```
T a = x;
T2 b = a;
T c = b;
```

implies that **a==c** for every initial value **x**; that is, that the **T** to **T2** to **T** conversions are not value preserving. The "escape clause" for constant expressions simply say that we accept a conversion if it wasn't narrowing for a particular value.
comment]

10.3 General syntax proposal

We propose to accept initializer lists as (rvalue) expressions.

(The basic idea is to allow an initialize list in every where in an expression that doesn't cause an expression statement to start with **{**, which would cause chaos in the grammar).

In A.4 Expressions [gram.expr] change

<<insert the elaborated version of the grammar from §6.2 here>>

No built-in operator can be applied to an initializer list. The meaning of an initializer-list is determined by the function to which it is an argument. [example:

```
    int a = 0;
    a+{1};           // error: built-in + for initializer list
    complex<double> b;
    b = { 1,2 };     // ok, means b.operator=({1,2})
    b+{1};           // ok, means operator+(b,{1})
    c = {1,2}*b;
    c = b+{1,2};
    {1,2} + b;       // syntax error
    {1,2} = b;       // syntax error
```

end example]

10.4 sdt::initializer_list

We propose that the **initializer_list** class template is added to the standard library.

In ??? add

In **<initializer_list>**:

```
template<class E> class initializer_list {
        // representation implementation defined
        // (probably two pointers or a pointer and a size)

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

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

An **initializer_list** provides read-only access to an array of elements of type **E**. **E** must be a value type (ref ???). The representation of an **initializer_list** is implementation defined. [comment

A pair of pointers or a pointer plus a length would be obvious representations for **initializer_list**; **initializer_list** is used to implement initializer lists as specified in the core language (sec ???).
comment]

10.5 Containers

We propose that each standard library container (as well as **basic_string** and **basic_regexp**) is provided with a sequence constructor for its element type. N2220=07-0080 *Initializer Lists for Standard Containers* contains all the details. For example:

Section §23.2.2. Add the sequence construcor to the class template deque, along with new assignment operator, and overload of assign

```
template<class T, class Allocator = allocator<T>>
class deque {
//...
deque(initializer_list<T>, const Allocator& = Allocator());
deque& operator=(initializer_list<T>);
void assign(initializer_list<T>);
void insert(iterator, initializer_list<T>);
};
```

Section §23.2.2.1. Add the following paragraphs:

deque(initializer_list<T> s, const Allocator& a = Allocator());

Effects: Construct a deque equal to **deque(s.begin(), s.end(), a)**.
Complexity: Make **s.size()** calls to copy constructor of T.

void assign(initializer_list<T>);

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

deque& operator=(initializer_list<T> s);

Effects: **assign(s);**

void insert(iterator p, initializer_list<T> s);

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

(No we do not propose anything to **vector<bool>**; anyone who feels like improving that can easily propose it using the techniques demonstrated here)

11 Appendix B: Initializer lists and argument passing

This appendix is a discussion of the design alternatives to what we consider a key example: initializing a **vector**.

11.1 The problems

The design for initializer lists has two main aims:

- To provide a uniform initializer syntax and semantics
- To provide variable-length homogeneous lists of initializers

Any solution is constrained by the essential third aim

- Compatibility: Don't break old code

It is easy to meet any one of these three aims. Meting all three simultaneously is a difficult puzzle.

The central problem with uniform initializer syntax is that the underlying semantics cannot be completely uniform, leaving open the possibility of ambiguity or (potentially surprising) implicit resolution of what could have been considered an ambiguity. Consider our key example:

- Create a **vector<int>** with 1 element initialized to the value 2.
- Create a **vector<int>** with the values 1 and 2 as its initial elements

How would we express that? The obvious answer seems to be:

```
vector<int> v1(1,2);         // C++03
vector<int> v2 = { 1, 2 };   // extend the C aggregate initialize list syntax
```

Remember that the declaration of **v1** is surprising to most until they have it explained and get used to it; it also causes its own problems for the type system (a clash with the template constructor for sequences, which will only be satisfactorily be resolved by applying concepts). There is nothing fundamental about this solution. It relies on familiarity to let **(…)** hint at argument passing and **={…}** hint at assignment to elements. In reality, both cases involve passing of arguments to a constructor and the "assignment to element" is of course initialization (rather than assignment). Obviously, this solution does not provide a uniform syntax for initialization. Furthermore, the syntactic differences don't indicate semantic differences as clearly as we would like.

Consider a related example illustrating a compatibility concern:

```
struct S1 {
        int x,
        int y;
};

struct S2 {
        int x,
        int y;
        S2(int xx, int yy) : x(xx), y(yy { }
};

S1 s1 = { 1, 2 };
S2 s2(1,2);
```

The initialization of **s1** uses **{…}** and the initialization of **s2** uses **(…)** to achieve exactly the same end. The syntax emphasizes a difference in the way that initialization is achieved even though a compiler might very well generate the exact same code in both cases. This is out character for C++ where we don't usually introduce different syntax to distinguish between user-defined and built-in operations (semantics). For example, we use **+** for both built in add and a user-defined add and **=** for both built-in copy and a user-defined copy. We could (if we so chose) describe **S1** as having "the default member constructor" and allow

```
S1 s1(1,2);      // not C++03 and not proposed here
```

This would be perfectly consistent with our treatment of default and copy constructors. Using parentheses throughout was one line of exploration for a uniform syntax, but it doesn't extend cleanly and compatibly to arrays, to variable-length argument lists or return values. However, it falls naturally out of the use of **{ … }** lists.

Now return to the key problem: How do we distinguish between a **vector<int>** with 1 element initialized to the value 2 and a **vector<int>** with the values 1 and 2 as its initial elements? Assume first that we use a uniform syntax for initialization:

```
vector<int> v2 { 1, 2 };        // one or two elements?
```

For this example, we could have used **vector<int> v1(1,2)** but – as mentioned – we found that to be a dead end. What choices do we have for this example?

- Sequence constructors take priority: It's a variable-length initializer list that happens to have two elements
- "Ordinary constructors" take priority: It's an argument list that matches one of vector's constructors
- All constructors are equal: It's an ambiguity error.
- Outlaw it: a class can have only sequence constructors or ordinary constructors, but not both

Requiring a class designer to choose sequence constructors or ordinary constructors would outlaw many useful combinations. In particular, it would make it impossible to add sequence constructors to standard containers, such as **vector**. So that idea is a dead end.

Relying on ambiguity detection saves people from some unpleasant surprises, so that's the first alternative to explore. For "ambiguous" to be an acceptable answer, the ambiguities must be relatively rare and easily resolved by the user. In terms of numbers of classes, this ambiguity is going to be rare. Most classes won't have sequence constructors and many of those that do, won't have constructors that can clash with a sequence constructor. Note that ever **vector** doesn't suffer the problem for many (most?) element types. For example:

```
vector<int*> vp1 { 1, &obj };        // ordinary constructor
```

```
vector<int*> vp2 { &obj };            // sequence constructor
vector<int*> vp3 { 1 };               // ordinary constructor (initialize to 0)
```

Unfortunately, the examples that can/will cause problems are frequent and important: containers of elements of numeric types. Obviously, this importance is also why it would be unwise to accept a poor solution to the problem. So, assuming this is ambiguous:

```
vector<int> v2 { 1, 2 };       // one or two elements?
```

How do we resolve the ambiguity? We would need to have notations for resolving it both ways. Consider:

```
vector<int> v21 = initializer_list<int> { 1, 2 };     // two elements
vector<int> v22 { 1, 2, };                            // two elements
vector<int> v23 ( 1, 2 );                             // one element
```

These solutions follow from compatibility and general principles. Each suffers from a nasty problem:

- The resolution to initializer list using **initializer_list** is verbose
- The resolution to initializer list using **comma** is obscure
- The resolution to argument list brings us back to "the old world".

What we don't have is a convenient way of disambiguating to ``ordinary construction'' without abandoning the **{ … }** notation. We could invent a notation; for example:

```
vector<int> v24 () { 1, 2, };                         // one elements
```

But no such disambiguation simply falls out of the design or follow from general principles, and all we have though of are ugly.

By defaulting resolution to either sequence constructor or ordinary construcor, we can trade one of these problems for surprises to someone who expected the opposite resolution.

11.2 Notation choices

At this point, most people will think, "why not simply have one notation for variable-length initializer lists and another for function arguments?" That is:

```
vector<int> v21 { 1, 2 };      // initializer list: elements 1 and 2
vector<int> v22 ( 1, 2 );      // argument list: 1 element initialized to 2
```

In other words, do we really need a uniform initialization syntax that includes variable-length initializer lists? First let us consider this question in the abstract; that is, independently of compatibility concerns:

- Do we really want to distinguish syntactically between initializing elements with values and initializing elements by passing values to a constructor? We think not.

The **S1** and **S2** example shows the feebleness of that distinction. Often, a constructor simply checks the values given to it (or puts them on a "normal form") before assigning them to members. That's not a fundamental logical distinction and in C++ we don't usually syntactically distinguish between user-defined and built-in operations. Fundamentally, it should be possible for the user to take the point of view that "I don't really care exactly how the object is initialized". Conversely, it should be possible for the write of a class to take control over initialization, however expressed (e.g. for checking).

- Any syntactic distinction that doesn't reflect a semantic distinction becomes a problem in the context of generic programming. For example, a template cannot without serious workarounds distinguish between argument types that require the one syntax for initialization from types that need the other. Uniformity of syntax is an important ideal here.
- The syntax above appears to distinguish between elements to be placed in the initialized object and arguments to a constructor. However,
  - sometimes, the arguments to a constructor are exactly values used to initialize members and
  - sometimes, the members initialized by an initializer list are just pointers to the "real elements" of the class – stored elsewhere and accessed indirectly through the members.

  Thus, the syntax doesn't necessarily reflect anything fundamental.
- Some homogeneous initializer lists (of values) are fixed-length, such as the list of coordinates for a point. Conversely, the list of arguments to a function can be variable length (we tend to simulate that with default arguments or overloading). Thus, the association of fixed-length with **(…)** and the association of variable-length with **{…}** is largely bogus.
- Some initializer lists are homogeneous, but many are not. A **pair** is a good example; so are most traditional initializer lists for **struct**s. Some argument lists are heterogeneous, but many are not. Thus the association of **{…}** with homogeneity and the association of **(…)** with heterogeneity is largely bogus.

We conclude that from a fundamental point of view, we would prefer a uniform initialization syntax that could be used to provide values for either a sequence constructor or other constructors. In particular, a uniform syntax will help generic programming and support the C++ design aim of equal support for user-defined and built-in types.

This is all very good, but couldn't we just cut through the subtle ties and say "initializer lists are really different; we'll use { } for those and ( ) for ordinary construction? Not really, consider compatibility. C introduced initializer lists for arrays and **struct**s long before C++ came onto the field. It also introduced separate syntax for initialization by non-aggregates (e.g., **=7**) and for providing arguments to functions (e.g., (**1,2)**). For compatibility, C++ adopted all that and added to possibility of using the function call method of specifying arguments to object initialization. The result is a mess. For C, the mess can be excused because uniformity of notation wasn't a C design goal, but for C++ the non-uniformity has become a serious problem (educationally and in writing more generic/reusable code). For compatibility, we must

- accept both **{…}** and **(…)** initialization in the language.

- accept **{…}** lists to be variable-length and (sometimes) heterogeneous.
- accept **(…)** lists to (sometimes) be variable length (think "**printf**") and (sometimes) heterogeneous
- not make any significant changes to the **(…)** semantics (but note the subsidiary/additional proposal to ban narrowing in initializations; §7).
- note that **(…)** often appears in contexts where it is not an initializer (e.g. as a list of (argument) types, a comma expression or (more generally) as a sub-expression.

Does this have implications on whether we should have a separate syntax for initializer lists and argument lists? Not really, but as ever compatibility constrains solutions and eliminates the possibility of simplifying by removing existing syntax.

We conclude that existing uses of **{…}** and **(…)** don't seem to significantly bias or guide the choice of notation for our **vector** example. In particular, both **{…}** and **(…)** are already used for both fixed-length and variable-length lists and for both homogeneous and heterogeneous lists. The wider use of **(…)** compared to **{…}** pushes us towards basing the unified initialization syntax on **{…}** rather than **(…)**, where a "green field" design might have had a free choice.

This example shows how a uniform notation helps:

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

11.3 Disambiguation

Back to our example: Assume that we use a uniform syntax for initialization:

```
vector<int> v2 { 1, 2 };        // one or two elements?
```

For this example, we have three choices:

- Sequence constructors take priority: It's a variable-length initializer list that happens to have two elements
- "Ordinary constructors" take priority: It's an argument list that happens to match one of vector's constructors
- All constructors are equal: It's an ambiguity error.

In §11.1, we saw that we need two ways of disambiguating. For example:

```
vector<int> v21 = initializer_list<int> { 1, 2 };     // two elements
vector<int> v22 ( 1, 2 );                             // one element
```

If we pick either of the first two alternatives, we need only one way to disambiguate. First alternative (prefer sequence constructors):

```
vector<int> v2 { 1, 2 };        // two elements
vector<int> v22 ( 1, 2 );       // one element
```

Alternative two (prefer "ordinary constructors"):

```
vector<int> v2 { 1, 2 };                              // one element
vector<int> v21 = initializer_list<int> { 1, 2 };     // two elements
vector<int> v21 { 1, 2, };                            // two elements
```

Preferring ordinary constructors for **{ … }** seems backwards, but it also has more fundamental problems:

```
vector<int> v2 {  };              // no elements (default constructor)
vector<int> v2 { 3 };             // three elements (initialized to 0)
vector<int> v2 { 1, 2 };          // one element (initialized to 2)
vector<int> v2 { 1, 2 , 3 };      // three elements with values 1, 2, 3
```

This is awful! Also:

```
vector<int*> vp1 { &i1 };               // one element
vector<int*> vp1 { &i1, &i2 };          // initialize using the sequence [&i1,&i2)
vector<int*> vp1 { &i1 , &i2, &i3 };    // three elements
```

Basically, giving priority to "ordinary constructors" implies a need for disambiguation that unpredictably appears depending on the number and types of elements. In other words, we can't use initializer lists uniformly even for a particular type. We will not pursue this alternative further. So we are left with two alternatives:

- Sequence constructors take priority: It's a variable-length initializer list that happens to have two elements.
- All constructors are equal: It's an ambiguity error.

The "ambiguity" resolution shares the non-uniformity problems we just mentioned with the "ordinary constructors take priority" approach. First we have:

```
vector<int> v2 {  };   // ambiguous: default constructor or empty initializer?
vector<int> v2 { 3 };          // ambiguous
vector<int> v2 { 1, 2 };       // ambiguous
vector<int> v2 { 1, 2 , 3 };   // three elements with values 1, 2, 3

vector<int*> vp1 { &i1 };               // ok (one element)
vector<int*> vp1 { &i1, &i2 };          // ambiguous
vector<int*> vp1 { &i1 , &i2, &i3 };    // ok (three elements)
```

We can resolve the ambiguities either way. First, let's resolve to use ordinary constructors:

```
vector<int> v2;                   // no elements (default constructor)
vector<int> v2(3);                // three elements (initialized to 0)
```

```
vector<int> v2(1, 2);                    // one element (initialized to 2)
vector<int> v2 { 1, 2 , 3 };             // three elements with values 1, 2, 3

vector<int*> vp1 { &i1 };                // ok (one element)
vector<int*> vp1 (&i1, &i2);             // initialize using the sequence [&i1,&i2)
vector<int*> vp1 { &i1 , &i2, &i3 };     // ok (three elements)
```

So much for uniform syntax! Alternatively, we can resolve the examples to use the sequence constructor:

```
vector<int> v2 = initializer_list<int>{  };
vector<int> v2  = initializer_list<int>{ 3 };
vector<int> v2  = initializer_list<int>{ 1, 2 };
vector<int> v2 { 1, 2 , 3 };

vector<int*> vp1 { &i1 };
vector<int*> vp1  = initializer_list<int*>{ &i1, &i2 };
vector<int*> vp1 { &i1 , &i2, &i3 };
```

It's verbose, but at least we didn't have to modify the initializer lists themselves. The alternative using the trailing comma disambiguation is:

```
vector<int> v2 = initializer_list<int>{  };
vector<int> v2  = { 3, };
vector<int> v2  = { 1, 2, };
vector<int> v2 { 1, 2 , 3 };

vector<int*> vp1 { &i1 };
vector<int*> vp1  = { &i1, &i2, };
vector<int*> vp1 { &i1 , &i2, &i3 };
```

That can be subtle.

Disambiguation by prefixing an initializer list with its desired type is a genera and mechanism. The disambiguation by a trailing comma is needed for compatibility. However, neither is minimal or ideal. What would be minimal and ideal? Something that simply said "this initializer lists may not be used as arguments for an ordinary constructor". For example, we could make a = significant in declarations

```
vector<int> v21 { 1, 2 };       // potentially ambiguous
```

or have to disambiguate:

```
vector<int> v21 = { 1, 2 };     // definitively a set of elements
```

That is, we could have **{…}** mean “initializer list (either arguments of values)” and **={…}** mean “initializer list; do not use as constructor arguments”. This is a complete solution: It can be used in every initialization context. However, we consider it “too cute”. Note that is purely an aesthetic judgment. However, the minute we consider using initializer lists within general expressions that **={…}** notation starts to look seriously weird:

```
void f(const vector<double>&);
// …
f({x,y});         // potentially ambiguous
f(={ x ,y });     // ok (but weird)

v1 = { 1, 2 };                          // potentially ambiguous
v2 = vector<int>{1,2 };                 // also potentially ambiguous!
v3 = initializer_list<int>{ 1, 2 };     // ok
v4 = ={1,2};                            // ok (but weird)
```

For the assignments, we are looking at the initializer list as the argument to **vector<int>::operator=()**.

We do not propose a solution beyond the **initializer_list** prefix and the comma suffix that we can avoid anyway.

The simplest example of a false positive (if we use pure ambiguity resolution) is the default constructor:

```
vector<int> v;
vector<int> v { };      // potentially ambiguous
void f(vector<int>&);
// …
f({ });                 // potentially ambiguous
```

It is possible to think of classes where initialization with no members is semantically distinct from default initialization, but we wouldn’t complicate the language to provide better support for those cases than for the more common case where they are semantically the same.

Giving priority to sequence constructors breaks argument checking into more comprehensible chunks and gives better locality.

```
void f(const vector<double>&);
// …
struct X { X(int); /* … */ };
void f(X);
// …
f(1);           // call f(X); vector’s constructor is explicit
f({1});         // potentially ambiguous: X or vector? (vector with 1 element)
```

```
f({1,2});         // potentially ambiguous: 1 or 2 elements of vector (2 elements)
```

Here, giving priority to sequence constructors eliminates the interference from **X**. Picking **X** for **f(1)** is a variant of the problem with explicit shown in §3.3.

11.4 Conclusion

So, how do we decide between the remaining two alternatives ("ambiguity" and "sequence constructors take priority over ordinary constructors)? Our proposal gives sequence constructors priority because

- Looking for ambiguities among all the constructors leads to too many "false positives"; that is, clashes between apparently unrelated constructors. So, we give priority of sequence constructors.
- Using exactly the same syntax for every number of elements of a homogeneous list is important – disambiguation should be done for ordinary constructors (that do not have a regular pattern of arguments). See examples in §11.3.

These decisions lead to the rules in §4.1.

12 Appendix C: Is a new syntax needed?

In other words, can we – in a compatible manner – extend or change the definition of one of the existing notations to serve as a universal initialization syntax and semantics? The basic answer is "no".

12.1 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 many programs using a **vector** or similar type would fail).

We consider both alternatives unacceptable. It follows that we cannot eliminate the distinction between copy initialization and direct initialization without serious compatibility problems.

> **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.

12.2 Can we eliminate differences caused by copying?

In addition to the issue of implicit vs. explicit constructors, we have the issue of actual copying vs. construction "in place". Assume that **X** is a type that we can initialize with an **int**; consider

```
void f(const X& v);
void g(X);
X v(1);         // copy not allowed
X v2 = 1;       // copy undesirable and easily avoided
f(1);           // copy undesirable and easily avoided
g(1);           // copy (almost) unavoidable
```

For the results of initialization to be exactly the same in all cases, we must either copy in all cases or in none. Copying in all cases is clearly undesirable because of the significant overhead it would impose compared to current C++. On the other hand, avoiding copying in every case is essentially impossible. Either choice would be incompatible, so we will have to live with copying in some cases (e.g. for call-by-value arguments) and not in others (e.g. in direct initialization of local variables). As ever, copy operations that just copy correctly are only visible as a factor in performance, a private copy constructor can cause an initialization to be illegal, and a copy constructor may "become visible" if it throws an exception.

There is currently one way of getting uniform initialization: Always use the most explicit form of initialization:

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

template<class T> void f(T v);
f(vector<int>(7));     // copy
f(X(1));               // copy
```

We cannot recommend that style for systematic use because it is unnecessarily verbose and also implies serious inefficiency unless compilers are guaranteed to eliminate most copy operations.

12.3 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 since { … } initialization is defined as direct initialization the problems doesn't appear in that case (thus use of initializer list syntax will address this problem also). For example:

```
X c{1} ;         // invoke X(int)
```

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

13 Appendix D: A syntax problem

We have left = optional when initializing a variable with an initializer list. For example:

```
vector<int> v1 = {1, 2, 3, 4 };      // vector of 4 elements
vector<int> v2 {1, 2, 3, 4 };        // vector of 4 elements
```

Semantically, the initialization of **v1** and **v2** are equivalent.

Leaving out the = causes a problem for some parsers (notably GCC). Consider:

```
typedef void T();
T f1;            // same as void f1();
T f2 { }         // error
```

A traditional grammar that has grammar elements for the type, the declarator, and for what comes after (semi-colon, initialization, or body, depending on the type) has no problems with this example. However, you can look ahead for a ; = or { and then parse the declarator accordingly (in a smaller context). To such a parser **f2** and **v2** look the same.

This problem is easily solved by requiring a = before the {. What are the arguments for and against (requiring that =)?

Consider

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

X* p = new X{ v };
X* q = new X = {v};  // no

void f(X);
f(X{v});
f(X ={v});                  // no
```

We don't propose to allow the **=** in the initialization of ***q**. That would create serious grammatical chaos. Thus, we need **X{v}** in some contexts and can't have **={v}** in all. We would prefer not to have **=** required in some places and prohibited in other, with no semantic difference. (This is also the reason we cannot use = to distinguish between initializer lists of elements and initializer lists of constructor arguments).

The **=** in **={v}** strongly suggests assignment (copy). That was the reason **(v)** was invented for constructor arguments. Consider and imaginary task class that takes an operation, an input stream, an output stream, and a timeout):

```
Task* p = new Task(filter, cin, cout, 100);
Task t2(source, noop, cout, infinite);
```

Using **{** … **}** this would become

```
Task* p = new Task{filter, cin, cout, 100};
Task t2{source, noop, cout, infinite};
```

However, adding = would give this:

```
Task* p = new Task{filter, cin, cout, 100};
Task t2 = {source, noop, cout, infinite};
```

Basically, this becomes the case for every class that is not primarily a container of elements.

Also, consider:

```
X x = X(v);
X y = x;
```

Here, the = strongly suggests copying (that is, that an accessible copy constructor must exist (correct) and that inefficiency lurks (not really)). The same association/stigma will attract itself to

```
X x = {v};
```

but not

```
X y {v};
```

This is somewhat superficial of people to think this way, but we should not lightly offend the sensibilities of huge numbers of novices and would-be experts.