N1509=03-0092
September 18, 2003

Generalized Initializer Lists

Gabriel Dos Reis (gdr@acm.org)

Bjarne Stroustrup (bs@research.att.com)

Abstract

This proposal resents two suggestions:

(1) to allow initializer lists to be used in expressions
(2) to allow initializer lists to be used for containers

Basically, the proposal treats an initializer list as an expression that gets its meaning by comparing the type of its element with the type of arguments for possible constructors. A constructor taking a pair of initializers is called a sequence constructor and is used for initializer lists. If no sequence constructor is defined or if the sequence constructor didn't match the initializer list other constructors are tried.

The purpose of this proposal is to provide significant notational advantages and to make the language rules more general and uniform. This proposal is completely compatible with C++98. It fits with C99 uses of initializer lists, with the related proposals for literal constructors, and concepts.

The Problems

An initializer list is a useful way to introduce data into a program. K&R C provided that facility for initializer lists for global arrays and structures only. C89 and C++98 added the ability to initialize local variables using initializer lists. Dialects of C, C99, and languages with a C-style syntax, such as Java and C#, allows variants of initializer lists to be used as function arguments, as the right hand of the assignment, etc. For example;

```
f({1,2,3,4});
s = {1, "asdf", 5};
```

That creates a significant pressure on C++ to provide facilities that has become familiar, and they are clearly useful. Basically, the lack of the generality in the current rules is becoming painfully obvious.

In addition, it is a serious problem that initializer lists favor low-level built in types (such as arrays, C-style strings, and **struct**s without constructors) over higher-level types relying on constructors (such as **std::vector** and **std::string**). For example:

```
int a[] = { 0, 1, 2, 3, 4, 5 };    // simple, direct
vector<int> v;                      // oops: we can't directly initialize v with a list
vector<int> v2(a,a+5);              // indirect, error prone
```

The **vector<int>** example isn't just longer and less direct than the array example, it also forces people to use arrays in addition to **vector**s and suffer the chance of getting the size wrong (as I did above☺).

If we simply allow aggregate initialization more widely, this will further increase the support of built-in types over user-defined types. For example:

```
struct S { int a, b };
void f(S);
f({1,2});

void g(int[]);
g({1,2,3,4});

class C { public: C(int a, int b); /* ... */ };
void h(C);
h(C(1,2));      // C must be explicitly mentioned

void k(vector<int>&);
k(???);         // we cannot directly pass an initializer list to k()
```

This violates the C++ design rule that user-defined types should be supported as well as built-in types. In particular, it handicaps users of standard containers compared with users of arrays.

As we solve this problem, we aim to

- provide more uniform rules for initialization
- increase the range of initializations that can be directly expressed
- increase the degree of C99 compatibility
- maintain strict compatibility with C++98

In total, this will increase the support for both system programming techniques and generic programming.

The solution

We propose to allow an initializer list in any place where we can easily deduce the type of the object required. To do this, we first introduce the concept of a "sequence constructor". A sequence constructor is a constructor taking a pair of pointers (only) as arguments. Consider:

```
template<class T> class vector {
        // ...
        vector(const T* first, const T* last);      // sequence constructor
        // ...
};

vector<int> vi = { 1, 2, 3, 4, 5 };
```

The idea is to implement the initialization of a vector by an initializer list with a call of the sequence constructor with the first and one-past-the-last element of the list as arguments. A naive implementation would be what a user would currently have to write by hand:

```
int ai[] = { 1, 2, 3, 4, 5 };
vector<int> vi(ai,ai+5);
```

The second part of the proposal is to consider an initializer list as a constructor argument list in all contexts where the desired type is known. For example:

```
class C { public: C(int, double); /* ... */ };
void f(C);

f(C(1,2.3));    // current/normal/explicit syntax
f({1,2.3});     // equivalent call using initializer syntax
```

The two mechanisms can be used in combination. For example:

```
vector<complex> vc = { 1, {2,2}, 3, 4, {5,5}};
```

This is roughly equivalent to

```
complex ac[] = { 1, complex(2,2), 3, 4, complex(5,5) };
vector<complex> vc(ac,ac+5);
```

For classes without constructors, initializer lists simply initialize members in order. For example:

```
struct S { int; double d; };
S s = { 1, 2.3 };           // as ever

void f(S);
f({1,2.3});
```

The basic rules for use of an initializer list are

1. if the target has no constructors, try aggregate initialization; otherwise
2. if the target has a sequence constructor, use that; otherwise
3. use the initializer list as an argument list for the target's constructors

We call that the general initialization rule.

What exactly is a sequence constructor?

Consider:

```
class X {
public:
        X(int, int);                  // not a sequence constructor
        X(int*, int*);                // not a sequence constructor
        X(const int*,const int*);     // a sequence constructor
        X(const int[],const int[]);   // (unfortunately) a sequence constructor
                                      // * and [] are equivalent in argument declarations
        X(Iter<int>, Iter<int>);      // not a sequence constructor
        X(const Iter<int>, const Iter<int>);          // not a sequence constructor either
        // …
};
```

A sequence constructor takes exactly two pointers to **const**. Its purpose is to "internalize" an initializer list and will typically be invoked by compiler-generated code, so we see no point in accepting a more general notion of iterator. A sequence constructor is declared with exactly two arguments, both arguments are of exactly the same type, and the argument type is a pointer to **const** type and not:

- a void*
- a pointer to function
- a pointer to member

Can a sequence constructor be a member template? Following the current rule that a template cannot be used to generate a copy constructor, we suggest not:

```
class X {
        // …
        template<class T> X(const T&);              // not a copy constructor for T == X
        template<class T> X(const T*, const T*);  // not a sequence constructor
};
```

This might be constraining, and we don't consider this restriction fundamental.

One could imagine a constructor taking two pointers that was not a sequence constructor in the way defined here (that is, using its pair of pointers as iterators). For example, a piece of old code might simply take pointers to two unrelated objects:

```
class Precious {
public:
        Precious(const Obj*);                               // one Obj
        Precious(const Obj*, const Obj*);                   // two Objs
        Precious(const Obj*, const Obj*, const Obj*);       // three Objs
        // …
};
```

The general initialization rule ensures that such code works unless you happen to try to initialize a **Precious** using an initializer list:

```
Precious a = new Obj(1);
Precious b(new Obj(1),new Obj(2));
Precious c(new Obj(1),new Obj(2),new Obj(3));
Precious d = { 1, 2, 3 };               // oops! very unlikely to have the right semantics
```

In other words, if a constructor isn't a sequence constructor you must avoid invoking it for an initializer list. This condition is trivially met by old code. The requirement that a sequence constructor takes **const T***s rather than plain **T***s also minimize the chance of confusion. If a programmer accidentally leaves out the const for an intended sequence constructor, the compiler catches the problem:

```
class Minor_confusion {
public:
        Minor_confusion(int*,int*);          // supposed to be sequence constructor
        // …
```

```
    };

    Minor_confusion a = { 1,2,3 };      // error: Minor_confusion has no sequence constructor
```

It does not seem necessary to introduce new syntax simply to protect against confusion between "ordinary" constructors and sequence constructors. In particular, a keyword-based syntax seems excessive. For example:

```
    class Overkill {
    public:
            sequence Overkill(const T*, const T*);     // sequence constructor
            Overkill(const T*, const T*);              // not sequence constructor – no keyword used
            // …
    };
```

So we don't propose that.

Type matching

In a call, an initializer list is a single argument. For example:

```
    void f(int,char);
    f(1,'a');          // ok
    f({1,'a'});        // error: two arguments expected

    class X {
    public:
            X(int,char);
            // ...
    };

    void f(X);
    f({1,'a'});        // ok: f(X(1,'a')); (3) of the general initialization rule
```

This has an important implication. Consider assignment:

```
    X a;
    a = {1,'a'};
```

by definition this assignment means

```
    a.operator=({1,'a'});
```

which resolves to

```
    a.operator=(X(1,'a'));
```

In other words the rules for assignment of initializer lists follow from the general rules for assignment and for initializer lists.

Element conversions

What happens if the elements of an initializer list don't have the same type as is specified in a sequence constructor? Consider;

```
class X {
        X(const int*, const int*);
        // …
};

X a = { 'a', 'b' };     // equivalent to  { int('a'), int('b') }
X b = { 1.2, 3 };       // equivalent to  { int(1.2), 3 }
X c = { &a, &b };       // error: no conversion from X* to int
X d = { a, b };         // error: no conversion from X to int
```

The usual conversion rules apply for the elements. In particular, an element in an initializer list is considered a direct initializer for an element, so that even an `explicit` constructor will be used. For example:

```
class E {
public:
        explicit E(int);
        // …
};

class Y {
public:
        Y(const E*,const E*);
        // …
};

Y y = { 1,3 };  // ok: equivalent to { E(1), E(2) };
```

This is consistent with the view that the use of an initializer list is equivalent with (one or more) invocations of a constructor.

Consider an attempt to make the **X** example work:

```
class X {
        X(const int*, const int*);
        X(const X*, const X*);
        // …
};

X b = { 2, 3 };         // ok to invoke X(const int*, const int*)?
X c = { a, b };         // ok to invoke X(const X*, const X*)?
```

In other words, can a class have more than one sequence constructor? We suggest that the answer is “no” because little generality is gained for the extra complexity. The complexity would arise from trying to do overload resolution for sequence constructors based on elements that themselves might be of several different types and require conversions.

Then, should a second candidate for a sequence constructor be handled? We could consider it an ordinary constructor:

```
class X {
        X(const int*, const int*);
        X(const X*, const X*);
        // …
};

X b = { 2, 3 };                 // ok to invoke X(const int*, const int*)?
X c = { &b, &c };               // ok? Equivalent to X c(&b,&c);?
```

However, this would imply a serious order dependency, so only a single sequence constructor is allowed.

There are two points where a double-sequence-constructor error can be detected: at the point of declaration or at the point of use. The former minimizes possible confusion, but the latter prevents breakage of existing code and is consistent with the view that a constructor taking a pair of pointers might not (semantically) be a sequence constructor. For example:

```
class X {
        X(const int*, const int*);          // sequence constructor?
        X(const X*, const X*);              // second sequence constructor?
        // …
};
```

We propose to allow the declaration, but ban the use of either for an initializer list. For example:

```
X a(new int(1),new int(2));  // ok
X b(&a,&b);                  // ok
X c = { 1,2,3,4 };           // error: two sequence constructors
```

It is the programmer’s job to know which constructors are semantically sequence constructors and use them accordingly.

Overload resolution

Clearly, ambiguous expressions can be constructed from initializer lists. How are those detected and how are they distinguished from initializer lists that can be disambiguated. For example:

```
class X {
public:
        X(const int*, const int*);          // sequence constructor
        // ...
};
```

```
class Y {
public:
        Y(const double*, const double*);    // sequence constructor
        // ...
};

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


f(1,2);           // error: no f takes two arguments
f({1,2});         // error: ambiguous  – candidates f(X(1,2)) and f(Y(1,2))
f({1.0,2.0});     // error: ambiguous – candidates f(X(1,2)) and f(Y(1,2))
f({1,2.0});       // error: ambiguous– candidates f(X(1,2.0)) and f(Y(1,2.0))
```

The resolution is done by first collecting the set of candidate functions (here **f(X)** and **f(Y)**) and applying the general initializer rule to each initializer list argument (and the usual argument rules for other arguments). This gives a set of candidate calls and the usual overload resolution rules are used to select among those and to reject ambiguities.

The reason that the calls **f({1,2})** and **f({1.0,2.0})** are considered ambiguous is that each is considered an exact match for both **f(X)** and **f(Y)**. We do not propose to add a new overload resolution rule that "remembers" that an initializer list match involved conversions and use that to introduce yet another level of match. That would be possible to do, but we'd like to see a clear need before adding such complications. We propose that an initializer list either mach or don't, with no third alternative.

Once a sequence constructor has failed to match, other constructor can be considered (using the usual rules). For example:

```
class X {
public:
        X(const int*, const int*);      // sequence constructor
        X(string, string);
        X(complex<int>,complex<int>);
        // ...
};

X a = {1,2 };                           // sequence constructor
X b = { "asdf", "zxcv" };               // string constructor
X c = { complex<int>(1), 2 };           // complex constructor
X c = { 1, complex<int>(2) };           // complex constructor

void f(int* p, int* q)
{
        X a(p,q);         // call X(const int*, const int*) as "ordinary constructor"
        X b = { p,q };  // call X(const int*, const int*) as "ordinary constructor"
}
```

Explicit resolution

Sometimes it is useful to explicitly specify the desired type of an initializer list. How can we do that? Consider:

```
class X {
public:
        X(const int*, const int*);      // sequence constructor
        // ...
};

class Y {
public:
        Y(const int*, const int*);      // sequence constructor
        // ...
};

f(X);
f(Y);

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

We see four more or less obvious solutions:

```
f(X{1,2,3});    // #1 call f(X)
f((X){1,2,3}); // #2 call f(X), C99 notation
f(X(1,2,3));    // #3 call f(X), constructor notation with implicit list
f(X({1,2,3})); // #4 call f(X), constructor notation with explicit list
```

Consider

- Alternative #1 introduces a new syntax that simply prefixes the class name in front of the initializer list, much in the same way that a class name prefixes an argument list in a constructor call.
- Alternative #2 is similar to #1 but with the class name in parentheses. It is the C99 cast notation.
- Alternative #3 uses the ordinary constructor syntax, but tries the sequence constructor if the argument list does not match any other constructor. The order of trying out the sequence constructor and other constructors would have to be reversed relative to the general initializer rule to avoid nasty surprises.
- Alternative #4 is similar to #3 but preserves the general initializer rule by explicitly using the initializer list notation.

Alternative #1 should be accepted because it is the clearest, shortest, and most intuitive.

Alternative #2 should be accepted for C99 compatibility even though it is ugly and overloads the cast notation.

Unfortunately, alternative #4 would introduce some nasty ambiguities because it really isn't direct resolution of the problem. It "overloads" the use of an initializer list as an argument. Unless given special treatment, **X({1,2,3)}** would mean "look for a constructor for **X** that takes a single argument of a type with a sequence constructor". If it doesn't mean that, we have confusion, and if it does potentially mean that, we have a possibility of ambiguities in a mechanism for resolving ambiguities. We don't propose to introduce that problem and therefore we don't propose any special meaning for initializer lists in constructor calls, so alternative #4 is not a way of resolving ambiguities.

Alternative #3 is the ideal because it preserves a uniform notation for all explicit constructor calls. It is really the inverse of the use of an initializer list: instead of transforming the explicit use of an initializer list into an implicit constructor call, it transforms an explicit constructor call into an implicit use of an initializer list. Consider:

```
class X {
public:
        X(int, int);
        X(int, int, int);
        X(const int*, const int*);      // sequence constructor
        // …
};

X x1 = { 1,2,3 };        // invoke sequence constructor
X x2 = { 1,2 };          // invoke sequence constructor
X(1,2,3);                // invoke "ordinary constructor"
X(1,2);                  // invoke "ordinary constructor"
```

In other words, when we see a constructor call, we first apply the usual rules for argument passing and only if there isn't a match, we try for a match for the sequence constructor (if any) treating the argument list as an initializer list.

Edge cases

The empty initializer list is allowed. For example:

```
X x = {};        // means construct x with the default initializer X()
```

This is equivalent to

```
X x;             // variable declaration
```

rather than

```
X x();           // function declaration.
```

An initializer list with a single element:

```
X x = { a };     // means X x(a);
```

Here, no copy constructor can be invoked (unless a is an X) so that this is different from plain

```
X x = a;
```

This would appear to make a slight difference from current rules (8.5[13]). However, that rule (that an **{ x }** initializer is equivalent to a **x** initializer) applies to scalar type where all initializer notations are equivalent anyway.

Consider also

```
class X {
public:
        X();
        X(int);
        // ...
};

void f();
void f(int);
void f(X);

f({});   // call f(int())
f();     // call f()
f({1}); // call f(int(1))
f(1)     // call f(int)
f(1.0); // call f(int)
```

There are no ambiguities because any use of X involves a user-defined conversion.

Note that the use of curly braces within an initializer list can be significant. For example:

```
{ 0, 1, 2, 3 }
```

and

```
{ {0,1}, {2,3} }
```

will have very different effects when you are trying to initialize a vector of complex numbers

Notational advantages

The initializer notation differs from the constructor notation in that the type is implicit and that an initializer list cannot be confused with a declarator. For example:

```
pair<int, string> p1 = { 1,"Tollan" };
pair<int, string> p2(1,"Tollan");
pair<int, string> p3 = pair<int, string>(1,"Tollan");        // repetitive

void f(pair<int,string>);
f(pair<int, string>(1,"Troldand"));            // verbose
f({1,"Troldand"});
```

The notational advantage increases with the complexity of the name of the type. In the case of **std::pair**, **make_pair()** was introduced to achieve the concise expression and type deduction provided by the initializer notation. For example:

```
f(make_pair(1,"Troldand"));
```

Consider:

```
vector<int> v(istream_iterator<int>(cin), istream_iterator<int>());
```

Contrary to the expectation of many (us included) who innocently tried, this does not declare a variable **v** of type **vector<int>** initialized with a sequence of **int**s read from the standard input. Rather, it declares a function named **v**, taking two **istream_iterator<int>**s and returning a **vector<int>**.

We do not propose to change this, but we can limit the embarrassment by suggesting an alternative:

```
vector<int> v = {
        istream_iterator<int>(cin),
        istream_iterator<int>()
};
```

or equivalently

```
vector<int> v({ istream_iterator<int>(cin), istream_iterator<int>() });
```

PODs

For classes without user-defined constructors, initialization with an initializer list degenerates to memberwise initialization:

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

S a = {1, 2.0 }
f(S);
f({1,2.0})
f(S(1,2.0));            // explicit construction; constructor syntax
f(S{1,2.0});            // explicit construction, new syntax
f((S){1,2.0});          // yet another alternative syntax
f((struct S){1,2.0});   // more verbose alternative syntax, C99 syntax
```

The constructor style and C99 style syntax are provided for generality and compatibility.

Related Suggestions

Initialization and constructors are central to much code. Consequently, this proposal for generalizing constructors touches upon issues concerning many other proposals and suggestions.

Generalized literals

This general use of initializer lists can be seen as relation to the issue of how to express literals of composite types; that is, initializers that can be handled at compile time. This question will be considered in a separate paper (Stroustrup ???). The two proposals are completely compatible.

Concept checking

It is not possible to do template argument deduction from an initializer list because an initializer list does not have a type and a template argument does not provide a context through which that type could be deduced. For example:

```
template<class T> void f(T);
f({1,2,3});             // can't be resolved - {...} has no type
f(X({1,2,3}));          // ok

template<class T> class Z { /* ... */ };
Z<{1,2,3}> z1;          // can't be resolved - {...} has no type
Z<X{1,2,3}> z2;         // ok
```

Here, "has no type" is a shorthand for "has no named type and its type is deduced/composed from its element types".

However, if a notion of concept is introduced into C++, initializer lists could potentially be resolved through concept matching rules. For example:

```
template<X T> void f(T);
template<Y T> void f(T);
f({1,2,3});             // can potentially be resolved based on X and Y

template<X T> class Z { /* ... */ };
template<Y T> class Z { /* ... */ };
Z<{1,2,3}> z1;          // can potentially be resolved based on X and Y
```

If **X** but not **Y** requires a sequence constructor for a sequence of **int**s or a three-argument constructor **X** will be chosen.

nullptr

The empty initializer list has been suggested as a notation for the null pointer. For example:

```
X* p = {};      // ok
int i = {};     // error: i is not a pointer
```

This ingenious idea would work because **{}** does not have a type and the rules for deducing its type could be designed to have the desired effect.

Such a proposal would be compatible with this proposal. However, we do not think that it would solve the problem with the lack of a named null pointer as seen by many programmers. People would simply rely on macros:

```
#define NULL {}
#define My_null {}
```

```
int* p = NULL;
int* q = My_null;
```

In other words, the cleverness of using **{}** to eliminate the need for a new keyword is wasted; it will simply encourage the use of de facto keywords and/or macros. In addition, in the context of template matching it is useful for the null pointer to have a distinct type. For a better approach to the null pointer problem, see the proposal for **nullptr** (Sutter and Stroustrup).

Type-safe variable length argument lists

Consider

```
f(const vector<int>& v);
g(int i, const vector<int>& v);
h(int i, const vector<int>& v, int j);

f({1,2,3});        // ok: v initialized to 1,2,3
g(1,{2,3,4});      // ok: v initialized to 2,3,4
h(1,{2,3,4},6);    // ok: v initialized to 2,3,4
```

It is conceivable to extend the notion of sequence constructors to allow

```
f(1,2,3);          // meaning f({1,2,3});
g(1,2,3,4);        // meaning g(1,{2,3,4});
```

but not (in any reasonable way) to

```
h(1,2,3,4,6);
```

Would this extension be a good idea? Our opinion is that this extension would be feasible. It is really the rule for turning a constructor argument list into a call of a sequence constructor applied arguments of a type with a sequence constructor. However, we don't see sufficient need or demand for this extension. The explicit use of initializer lists is sufficient, clearer, and already provides a form of type safe variable argument lists.

Tuples

Given a general tuple type, something very similar to sequence constructors can be achieved. For example:

```
class X {
public:
        X(const Tuple&);
        // …
};

X a = Tuple(1,2,1.3,4);

void f(const Tuple&);
f(Tuple(1,2,3.4,4));
```

However, each **Tuple** is its own type and require serious compile time effort and run time code to implement. Furthermore the **Tuple** approach doesn't quite achieve the notational convenience of initializer lists, doesn't quite handle all the cases, and doesn't address the issue of C99 compatibility.

Standard library containers

Naturally, the standard library containers should be modified to take advantage of sequence containers. For example:

```
template<class T, /* … */> class vector {
public:
        vector(const T* first, const T* last); // sequence constructor
        // …
};
```

Compatibility issues

We don't see any compatibility problems. The proposal has been specifically crafted to generalize existing notation and semantics.

Implementability

Because this proposal relies on existing rules and known techniques, we don't expect implementation difficulties. However, it should be noted that nothing that touches upon the issue of overload resolution is really trivial.

Acknowledgements

The notion of a sequence constructor, though not that name, was initially suggested to Bjarne by Alex Stepanov in the context of a discussion of how to provide better and more uniform support for generic programming along the lines of the standard library.