N3139=10-0129
Bjarne Stroustrup
9/4/2010

An Incomplete Language Feature

Abstract

As the draft standard stands, we cannot use {}-style initialization in default arguments. This paper argues that this is a serious problem, leaving the uniform initialization syntax and semantics as a technically incomplete feature. Fortunately, the problem is simple to remedy.

Introduction

When we voted in initializer lists and the uniform initialization syntax and semantics, default initializers were left untouched so that a {} list isn't allowed as a default initializer. Consider:

```
template<class T>class C {
        C(initializer_list<T>);
        // …
};

void f(C<int>);

void use1(C<int>& ci)
{
        C<int> x1 = {};
        C<int> x2 = {1};
        C<int> x3 = {1,2};
        C<int> x4 = C<int>();     // error: a list-constructor is not a default constructor
        C<int> x5 = C<int>{};     // verbose
        C<int> x6;                // error: a list-constructor is not a default constructor

        f({});
        f({1});
        f({1,2});
        f(C<int>());      // error
        f(C<int>{});      // verbose

}
```

However, when we want to specify a default argument, we don't have a free choice among the valid initializers. In particular, we can't pick the simplest default:

```
void f2(C<int> c = {});     // error: {} not allowed in default initializer
```

So we have to write

```
void f3(C<int> c = C<int>{});

void use3(C<int>& ci)
{
        f3({});
        f3({1});
        f3({1,2});
        f3();           // use default argument
}
```

This is at best "odd"/surprising. The problem is not just an oddity caused by a list constructor; it also emerges when you consider traditional aggregates:

```
struct S { int a,b,c; };

void f6(S);

void use1(S& s)
{
        S x1 = {};
         S x2 = {1};
        S x3 = {1,2};
        S x4 = S();     // OK: 5.2.3/2
        S x5 = S{};
        S x6;           // uninitialized

        f6({});
        f6({1});
        f6({1,2});
        f6(S());
        f6(S{});
}
```

Note that for S, the () notation works where it didn't for C<int>. We are back in the state where we have to remember which syntax to use for particular types.

Defining default arguments gets messy because we can't freely choose them from the expressions we can use as initializers and as arguments

```
void f7(S s = {});          // error: {} not allowed in default initializer
void f8(S s = S());         // OK: workaround (not general)
void f9(S a = S{});         // OK: but verbose workaround
void f10(S s = {1});        // error: {} not allowed in default initializer
void f11(S s = S(1));       // error: no appropriate constructor
void f12(S s = S{1});       // OK, but verbose workaround
```

Unfortunately, not every type name is as short as "S" so the verbosity implied in using S{1} rather than {1} can be significant. Also, it would be hard to explain why you have to write the

```
void f12(S s = S{1});        // OK, but verbose
```

when we usually would recommend writing

```
S x1 = {};
```

or

```
S x2{};
```

to get zero initialization.

The solution is simple: Allow the {}-initializer syntax for default arguments. The suggested wording is also simple: substitute *initializer-clause* for *assignment-expression* twice in the grammar and make the corresponding change to the accompanying text. An *initializer-clause* is already suitably defined:

```
initializer-clause:
        assignment-expression
        braced-init-list
```

The standard actually shrinks (by an insignificant amount). The suggested wording changes are presented below.

More examples

Ville Voutilainen comments: “Given

```
void f(vector<map<int, string>> x = {});
```

and

```
void f(vector<map<int, string>> x = vector<map<int, string>>{});
```

I know which I'd prefer to write” and gives the following (more realistic) example:

```
void f(const vector<map<int, string>>& x = {});
```

Daniel Krügler adds “the following key example

```
#include<initializer_list>

struct S {
        S(std::initializer_list<int>  = {});
};

S s1{};   // OK
S s2;     // OK
```

which provides an excellent bridge between list-initialization and default-initialization.”

How serious is this problem?

First, this is a practical problem. I had completely forgotten about it and was writing what I considered reasonable code in styles I would encourage using {} in default arguments (roughly equivalent to the two cases, I used as examples above) and found to my surprise and embarrassment that the FCD said I could not do so. The workarounds were ugly, verbose, (shock, horror) not always feasible, and forcing me to consider subtleties of initializer rules for individual types that I'm sure the average user don't know (for example, can a list constructor be used as a default constructor? Can I use () to value construct an aggregate?). Unless fixed, this will clearly become an embarrassment as others find that they cannot write default arguments just like they write arguments. Worse, the cases where {} seem "natural" is typically for the simplest cases which are the obvious candidates to become default arguments. Commentators and writers of style guides will have a field day. It appears that we have a clear example of "design by committee."

There is a further, and I think more significant, problem when we look at the bigger picture. As I write more code using {}-initialization and discuss initialization with people with bug records from large code bases, I find that a consistent use of {}-initialization can save people from errors, especially narrowing errors and unexpected conversion errors. I first tried to cater for "familiarity" by using {}-initialization only where obviously needed. However, "where obviously needed" is hard to define precisely, hard to explain, hard to practice, and led to unnecessary errors slipping by. The "universal" aspect of the {}-initialization is important: You can use it everywhere and it obeys the same semantic rules everywhere. That was our agreed upon aim and it delivers simplicity and ease of use (except currently for default initializers). Not allowing {} in default arguments

- leads to messy workarounds
- complicates the language, its learning, and its most effective use
- compromises claims of a uniform initialization mechanism.

This is an issue of completeness and consistency of the language design, a teaching issue, and a practical issue.

Why do we have this problem?

I honestly don't know why we have this problem. There was a discussion about this in EWG at some point in the evolution of the initializer proposal. There is a record of a split vote to prove that, but I don't recall any technical arguments or any presentation along the lines presented above. I suspect that before the final synthesis of the feature, default arguments may have been seen as a separate or distracting issue "to be considered later" or simply confounded in the heat of debates about different aspects of initialization. I have asked for opinions and recollections on the core reflector and elsewhere, but no one seems to have a clear idea and nobody offered a technical argument against.

One possible source of the problem is that in the grammar, an initializer list is not an expression, so that it was easy to forget a case or consider it "special." The reason that an initializer list isn't an expression is simply that we decided (correctly, IMO) not to allow initializer lists on the left hand side of assignments, as operands of ++, etc. and further decided (again correctly, IMO) to enforce that through the grammar.

I'm pretty sure that the discussion of default arguments was confused by becoming merged with a discussion of the use of the {} notation for enumerator initializers. I will discuss the enumerator initialization issue below, but I believe that the two issues can and should be separate. Enumerators are a rather odd part of C and C++ with a set of rules all of their own.

In the examples above, I use the ={} syntax (with the =) to emphasize that a default argument is never used as a direct initializer. I recall that someone worried about overly clever use of {} leading readability problems. For example, someone might have been worried by this:

```
void f2(C<int> c{});     // default: initialize collection to empty
void f5(S s{});          // default: initialize aggregate to all zeros
```

and this

```
void f2(C<int>{});       // default: initialize collection to empty
void f5(S{});            // default: initialize aggregate to all zeros
```

This does look a bit odd and lacks the = as a clue that an initializer is coming. Consequently proposed wording doesn't include this (i.e., it requires the =). However, I'm not in principle against it and the grammar change would be trivial.

In response to my questions, Doug Gregor mentioned that allowing {}-initializers as default arguments might complicate error recovery, but David Vandevoorde pointed out that we can already have {} there as part of a lambda.

Is it too late?

If you consider the use of {} in default arguments a separate and new feature, it is clearly too late for C++0x. However, I don't think that we can consider default arguments in isolation from other initializers or initialization without considering default arguments. The {} notation and semantics for initializers is (as repeatedly stated and agreed upon) one feature rather than N separate features for the N places in the syntax where an initializer can be specified. I see {} in default arguments as simply a case that was missed in the final design of a uniform and universal initialization syntax and semantics.

Template non-type Default Arguments

The issue of {} in the context of default arguments also appears for non-type template arguments:

```
template<class T, T v = 0> T f(T a)
{
        return a+v;
}

void x = f(7);           // x becomes 7
void y = f<int,2>(7);    // y becomes 9
```

This cannot currently be written as

```
template<class T, T v = {}> T f(T a)
```

```
{
        return a+v;
}

void x = f(7);              // x becomes 7 (because the default int value is 0)
void y = f<int,2>(7);       // y becomes 9
```

The latter formulation is more generic in that it does not use the integer literal 0. For uniformity and because it might be useful, this ought to be accepted. This is really a variant of the default function argument problem because the same grammar rule is used for both. This implies that the change allowing a function default argument to use {} also allows the syntax of this example. To allow the example, we need to add {} initializers to the list of allowed forms of default template arguments in [temp.arg.nontype].

Enumerator initializers

Should we be able to use {} in the initialization of enumerators? Consider

```
enum E1 { a=1, b=2 };
enum E2 : char { x=12, b=1234 };
```

We cannot write this as

```
enum E1 { a={1}, b={2} };
enum E2 : char { x={12}, b={1234} };
```

nor like this

```
enum E1 { a{1}, b{2} };
enum E2 : char { x{12}, b{1234} };
```

Note that the {} examples would – if allowed – catch the narrowing error.

The technical arguments for and against allowing {} for enumerators are simple

> ***For***: we can use {} everywhere else, so why not? If we can't, people will try anyway out of habit and be annoyed. It catches narrowing errors.
> ***Against***: enumerators are different in nature and syntax from objects. There isn't much gain to be had since enumerators must be of integral type. There is nothing than we can say with this feature than we cannot say without it and nothing that we can say terser with the feature than without it.

In principle, I'm in favor of allowing {} in enumerator initializers because I don't see enumerators as fundamentally different from other integral constants. The grammar change would be change

> *enumerator-definition:*
>   *enumerator*
>   *enumerator = constant-expression*

to

> *enumerator-definition:*
> *enumerator*
> *enumerator* = *constant-expression*
> *enumerator* =$_{opt}$ { *constant-expression* }

However, I remember vocal opposition in EWG (though I have trouble recalling specific arguments) and think that enumerators are sufficiently different from variables and objects that someone could reasonably considered this a separate issue. Consequently, I'm not proposing a change to enumerators.

Acknowledgements

Thanks to Gabriel Dos Reis, Jason Merrill, Ville Voutilainen, Daniel Krügler,and Clark Nelson for constructive comments.

Suggested wording

Change the grammar in [dcl.fct] and [gram.decl] from

> *parameter-declaration:*
> *attribute-specifier*$_{opt}$ *decl-specifier-seq declarator*
> *attribute-specifier*$_{opt}$ *decl-specifier-seq declarator* = *assignment-expression*
> *attribute-specifier*$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$
> *attribute-specifier*$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$ = *assignment-expression*

to

> *parameter-declaration:*
> *attribute-specifier*$_{opt}$ *decl-specifier-seq declarator*
> *attribute-specifier*$_{opt}$ *decl-specifier-seq declarator* = *initializer-clause*
> *attribute-specifier*$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$
> *attribute-specifier*$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$ = *initializer-clause*

To reflect that change, we have to change the word "expression" in [dcl.fct.default] paragraph 1 to "initializer-clause":

> If an *initializer-clause* is specified in a parameter declaration this *initializer-clause* is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

To avoid confusion and increase consistency, I suggest we shorten "default argument expression" to "default argument" in the rest of [dcl.fct.default]. The phrase "default argument expression" appears in about each of the following sections: 3.4.1, 7.1.2, 8.3.6, 11, 12.2, 14.7.1. Ultimately, the modified text needed here may have to reference all of them -- unless by some chance it's considered to be clear enough as is.

Change the first item of the list in paragraph 1 of [temp.arg.nontype] from

> an integral constant expression (including a constant expression of literal class type that can be used as an integral constant expression as described in 5.19); or

to

> an integral constant expression (including a constant expression of literal class type that can be used as an integral constant expression as described in 5.19) or a *braced-init-list* that can be used as an integral constant expression); or