Doc No: WG21 N3452 = .16 12-0142
Date: 2012-09-23
Reply to: Herb Sutter (hsutter@microsoft.com)
Subgroup: EWG – Evolution

Let {x,y,z} ⇒ explicit

Herb Sutter

This paper proposes allowing an initializer-list expression to be used in "explicit" situations, notably to call an *explicit* constructor.

Rationale: By definition, an initializer-list expression (and/or an *initializer_list* object) exists to initialize something – a conversion is not only expected, but tantamount required. Therefore the *{x,y,z}* syntax is already "explicit" and should be acknowledged in the language as *explicit* without requiring the user to still-more-explicitly redundantly type out the name of an already-known type.

Examples

For convenience, I'll use *unique_ptr* and *tuple* as strawman types that have explicit constructors, and have already raised concerns about usability with { init } expressions (see References).

Note that using a helper like *make_tuple* is a partial workaround, but is more verbose and not always available on other types with explicit constructors (e.g., there is currently no *make_unique*).

Example 1: Return Values

Consider:

```
// Example 1(a)
unique_ptr<widget> void f() {
   return new widget();          // error: right, this should be an error
   return { new widget() };      // error: why shouldn't this work?
}
```

This doesn't work because the constructor is explicit. Instead, the user is forced to write the type redundantly:

```
   return unique_ptr<widget>{ new widget() };
```

What value is there in forcing the author of the function – the same person who just wrote the return type – to explicitly write out this longer version with the explicit type (which now has to be repeated twice in the same expression, on top of being repeated from where he just wrote it as the return type), when it can't be anything else and there's nothing else he could possibly be doing but intending to construct a *unique_ptr* with those parameters?

The original example should be allowed, and call the explicit constructor.

Similarly, now that we have standard tuples, we should be encouraging people to use them as return types when a function has multiple return values ('yes, C++11 really supports multiple return values!'). But:

```
// Example 1(b)
tuple<int,float> f() {
   tuple<int,float> this_is_okay{ 1, 2. };   // ok (demonstration for ironic value)
   return { 1, 2. };                         // error: (wha?!) (oh, not that again…)
}
```

Same deal: The last line doesn't work, because tuple's ctor is explicit. You have to write:

```
   return tuple<int,float>{ 1, 2. };
```

or

```
   return make_tuple( 1, 2. );
```

What value is there in forcing people to explicitly write out the type or call a helper that merely deduces the types(!), when it can't be anything else than the return type – never mind that I myself just wrote that return because I'm the author of the function?

This should be allowed, because there's nothing else the programmer could possibly be intending.

Writing:

```
widget func() { return {init}; }     // not ok if explicit, but should be
```

should be considered just as "explicit" a syntax as the already-allowed

```
widget var{ init-list };             // ok if explicit
```

The only difference is that I'm defining a function vs. a variable, and in both cases: (a) I can't possibly be doing anything else so I can't see any room for implicit-conversion errors here; and (b) requiring the type to be written again explicitly is redundant because the author of the type and the expression are necessarily the same.

Example 2: Arguments

Consider:

```
// Example 2(a)
void f( tuple<int,float> );
f( { 1, 2. } );                      // error: only because ctor is explicit
```

This doesn't work because the constructor is explicit. Instead, the user is forced to write the type redundantly:

```
f( tuple<int,float >{ 1, 2. } );
```

or use *make_tuple* which happens to be available for *tuple* but isn't actually much better here:

```
f( make_tuple( 1, 2. ) );          // arguably an attempt to make it "implicit"
```

What value is there in making him write out the type (or a factory that deduces the type) by hand? If there is only one function *f* that takes a single argument, what else could be possibly mean – remembering that the user *knows* he's getting a conversion because he's explicitly asking for it with *{ }*, unlike the cases where we don't want him to accidentally get an implicit conversion when he's not aware of it?

The original example should be allowed, and call the explicit constructor.

Consider also:

```
// Example 2(b)

void f( unique_ptr<widget> );

auto spw = make_shared<widget>();
f( spw.get() );                    // error: right, this should be an error

f( new widget );                   // error: right, this should be an error

f( { new widget } );               // error: only because ctor is explicit
```

This doesn't work because the constructor is explicit. Instead, the user is forced to write the type redundantly:

```
f( unique_ptr<widget>{ new widget } );
```

What value is there in making him write out the type by hand – remembering that the user *knows* he's getting a conversion because he's explicitly asking for it with *{ }*, unlike the cases where we don't want him to accidentally get an implicit conversion when he's not aware of it?

The original example should be allowed, and call the explicit constructor.

Example 3: "=" Initialization

Consider:

```
// Example 3(a)

tuple<int,float> f() {

   tuple<int,float> this_is_okay{ 1, 2. };   // ok (demonstration for ironic value)

   tuple<int,float> this_is_not = { 1, 2. }; // error: explicit

}
```

Here again, the last line doesn't work, because *tuple*'s ctor is explicit. The simplest workaround is to omit the '=', otherwise you have to write this:

```
tuple<int,float> this_is_okay = tuple<int,float>{ 1, 2. };
```

or this:

```
tuple<int,float> this_is_okay = make_tuple( 1, 2. );
```

What value is there in forcing people to explicitly write out the type, or explicitly ask for it *to be deduced*,[1] when it can't be anything else than the variable's type – which I just wrote?

This should be allowed, because there's nothing else the programmer could possibly be intending.

In this particular case, I would make the argument that the correct fix is to say that "= { init-list }" should not be viewed as copy-initialization at all, but as a synonym for "{ init-list }" alone (i.e., the '=' is redundant and optional and is guaranteed not to involve a temporary object).

Aside: { } Copy Construction

Since we're speaking of copying and {}, we should allow {} to be used for copy constructors:

```
complex<float> z1 {1.};

complex<float> z2{z1};
```

References

-ext thread starting with [c++std-ext-12178]

-core thread starting with [c++std-core-21937]

LWG Issue 2051 on tuple: http://lwg.github.com/issues/lwg-closed.html#2051

[1] "Isn't it ironic? Don't you think?" – A. Morissette