Doc No: WG21 N4014
Date: 2014-05-25
Reply to: Nicolai Josuttis (nico@josuttis.de)
Subgroup: EWG
Prev. Version: none

# Uniform Copy Initialization

C++ distinguishes between copy initialization and direct initialization (see 8.5 §17).
This has the effect that an `explicit` in a constructor disables copy initialization, while direct initialization is still supported.
This proposal suggests to deal with copy initializations as with direct initializations.

Note that this proposal has nothing to do with the solution of the following issue (LWG issue 2051):

```
tuple<string,int,double> f()
{
   ...
   return {s,i,d};      // error: explicit constructor
}
```

## Motivation

Consider:

```
class C
{
  public:
    explicit C (const std::string&, const std::string&) {
    }
};

C c1 { "Steve", "Brown" };     // OK
C c2 = { "Steve", "Brown" };   // Error
```

Because copy initialization is disabled by `explicit`, the initialization of `c2` fails. This impact of explicit for copy initialization for years again and again raises a couple of confusions:

- Programmers using an equal sign for copy initialization assume that it works and **waste a lot of time looking the reason of the error**.
- Programmers are surprised that the former is possible but the latter is not. Note that the statement the only difference is an equal sign. **You can't argue that the second initialization is in any sense more dangerous than the first one.**
- Programmers are used from C that whenever `{` and `}` are used to initialize values, a `=` has to be placed between the object and the `{`.
  In that sense, **current uniform initialization is not as uniform as it could be**.

- This difference also leads to the **wrong impression** that this is not a direct initialization and **that an assignment operator is used** after the expression on the right side of the = is created.

This problem also applies for use cases when using the C++ standard library.

For example:

```
tuple<int> t1(1);      // OK
tuple<int> t2{1};      // OK
tuple<int> t3 = 1;     // Error !
tuple<int> t4 = {1};   // Error !
tuple<int> t5 = {};    // OK !! (default constructor has higher priority)
```

In principle, this problem is nothing new and was an issue even before initializer lists were introduced. Thus, before C++11, copy initialization with one argument also is influenced by explicit constructors.

For example:

```
std::shared_ptr<int> sp1(new int(42));    // OK
std::shared_ptr<int> sp2 = new int(42);   // Error
```

and:

```
std::vector<int> v1(2);      // OK, init with size
std::vector<int> v2 = 2;     // ERROR
std::vector<int> v3{2};      // OK, init with value 2
std::vector<int> v4 = {2};   // OK, init with value 2
```

However, when discussing this paper, it turned out that the experience is that copy assignments of plain integral values to vectors is a possible error, we still suggest to be able to detect. That is, this proposal does not suggest to change the behavior of initialization without braces.

**That is:**

> **This paper proposes to change C++ so that copy initialization using brace-init-lists shall follows the same rules as direct initialization (that is, that in practice the = in initializations with braces has no effect).**

Of course, `explicit` still shall play its important role when converting values on argument passing:

```
class Collection
{
  public:
    explicit Collection (int); // initial size
    ...
};

void foo (const Collection&);
```

```
fp(42);                 // still error
```

And even in initializations, `explicit` shall still play the role it currently pays in direct initialization. For example:

```
class C
{
  public:
    explicit C (const std::string&, const std::string&) {
    }
};

std::vector<C> v1 { {"Joe","Smi"}, {"Jim","Last"} };     // still error
std::vector<C> v2 = { {"Joe","Smi"}, {"Jim","Last"} };   // still error
```

## Resulting Effects of the Proposed Change

The proposal would mean for vector no change:

```
std::vector<int> v1(2);      // OK, init with size
std::vector<int> v2 = 2;     // still ERROR
std::vector<int> v3{2};      // OK, init with value 2
std::vector<int> v4 = {2};   // OK, init with value 2
```

But the proposal would mean for shared_ptr:

```
std::shared_ptr<int> sp1(new int(42));      // OK
std::shared_ptr<int> sp2 = new int(42);     // still ERROR
std::shared_ptr<int> sp3{new int(42)};      // OK
std::shared_ptr<int> sp4 = {new int(42)};   // now OK (formerly Error)
```

And the proposal would mean for tuple:

```
tuple<int> t1(1);      // OK
tuple<int> t2{1};      // OK
tuple<int> t3 = 1;     // still ERROR
tuple<int> t4 = {1};   // now OK (formerly Error)
tuple<int> t5 = {};    // OK  (default constructor has higher priority)
```

And as my motivating common case:

```
class C
{
  public:
    explicit C (const std::string&, const std::string&) {
    }
};
C c1 { "Steve", "Brown" };    // OK
C c2 = { "Steve", "Brown" }; // OK now (formerly Error)
std::vector<C> v1 { {"Joe","Smi"}, {"Jim","Last"} };     // still error
std::vector<C> v2 = { {"Joe","Smi"}, {"Jim","Last"} };   // still error
```

## Backward Compatibility

In principle, this proposal allows something that hasn't been supported before. Thus, all existing code should still be valid and behave the same.

The only exception could be that due to overload resolution now other functions might be considered than before. I don't suggest that. Instead, I suggest to use the same overload resolution, but to ignore the explicit in a constructor if the result of the overload resolution selects an explicit constructor.

## Is it worth it?

Of course you can argue that this proposal solves a small issue and that people have to learn not to use the = when initializing with one argument or using a braced init list.

However, this proposal helps to make C++ more convenient in a place where programmers even 17 years after C++98 struggle.

The current proposed solution thus avoids a common small problem that exists without any known reason. It would therefore count as one of the multiple small steps (as `auto`, range-based `for` loop, and initializer lists) to make C++ a more convenient programming language for the ordinary application programmer.

## Proposed Changed Wording

open

## Acknowledgements

Thanks to all people in the C++ community I discussed this topic with. Special thanks to Bjarne Stroustrup, and Jonathan Wakely, for some feedback.