Prohibit aggregates with user-declared constructors

Timur Doumler (papers@timur.audio)
Arthur O'Dwyer (arthur.j.odwyer@gmail.com)
Richard Smith (richardsmith@google.com)
Howard E. Hinnant (howard.hinnant@gmail.com)
Nicolai Josuttis (nico@josuttis.de)

Document #:	P1008R1
Date:	2018-06-08
Project:	Programming Language C++
Audience:	Evolution Working Group, Core Working Group

Abstract

C++ currently allows some types with user-declared constructors to be initialized via aggregate initialization, bypassing those constructors. The result is code that is surprising, confusing, and buggy. This paper proposes a fix that makes initialization semantics in C++ safer, more uniform, and easier to teach. We also discuss the breaking changes that this fix introduces.

1 Motivation

Today, the rules of initialization in C++ are frequently cited as one of the most confusing and hard-to-learn aspects of the language. A particular problem occurs with aggregates with user-declared constructors. The current rule says that an aggregate cannot have user-provided, inherited, or `explicit` constructors, but explicitly deleted or defaulted constructors are allowed. This rule has many surprising and unfortunate consequences.

1.1 Unintended instantiation

Consider:

```
struct X {
  X() = delete;
};

int main() {
  X x1;     // ill-formed - default c'tor is deleted
  X x2{};   // compiles!
}
```

Clearly, the intent of the deleted constructor is to prevent the user from initializing the class. However, contrary to intuition, this does not work: the user can still initialize `X` via aggregate initialization because this completely bypasses the constructors. The author could even explicitly delete all of default, copy, and move constructor, and still fail to prevent the client code from instantiating `X` via aggregate initialization as above. Most C++ developers are surprised by the current behaviour when shown this code.

The author of class `X` could alternatively consider making the default constructor `private`. But if this constructor is given a defaulted definition, this again does not prevent aggregate initialization (and thus, instantiation) of the class:

```
struct X {
private:
  X() = default;
};

int main() {
  X x1;     // ill-formed - default c'tor is private
  X x2{};   // compiles!
}
```

Because of the current rules, aggregate initialization allows us to "default-construct" a class even if it is not, in fact, default-constructible:

```
static_assert(!std::is_default_constructible_v<X>);
```

would pass for both definitions of `X` above.

The programmer has to know way too many details about both the class and about arcane initialization rules to be able to understand this behaviour. `X x1;` doesn't compile because the user-declared constructor inhibits the compiler from supplying a default constructor. However, `X x2{};` compiles because despite the rule that says the compiler doesn't supply a default constructor, the class has no user-provided, `explicit`, or inherited constructors, no private or protected non-static data members, no virtual functions, and no virtual, private, or protected base classes, and so the compiler allows the aggregate initialization syntax. The majority of C++ developers are unlikely to ever master those rules. As a result, we end up with code that gets used in unexpected ways, even after passing good unit tests such as the `static_assert` above. This leads to bugs and unintended behaviour.

1.2 Unintended member initialization

Aggregate initialization creates even more surprises if the type in question has non-static data members. Consider:

```
struct X {
  int i{4};
  X() = default;
};

int main() {
  X x1(3);  // ill-formed - no matching c'tor
  X x2{3};  // compiles!
}
```

Here, `X` has a default constructor, and the member `int i` is given a default member initializer to ensure that it will always hold the value `4` after default-construction. But surprisingly, aggregate initialization allows the user to bypass this interface and initialize `X` with a different value for `i`, potentially breaking class invariants. This can easily cause bugs and unintended behaviour.

It gets worse. We could try to make the intent even more clear by explicitly deleting the constructor that takes an `int`. But, surprisingly, this still doesn't make the class non-aggregate and therefore accomplishes nothing:

```
struct X {
  int i{4};
  X(int) = delete;
};
```

```
int main() {
  X x1(3);   // ill-formed - using deleted c'tor
  X x2{3};   // compiles!
}
```

This behaviour is confusing, not useful, and potentially dangerous.

1.3 `explicit` default constructor

Under the current rules, a workaround exists to prevent the user from intializing an aggregate: adding `explicit` to the deleted default constructor causes the type to become non-aggregate.

```
struct X {
  X() = delete;
};

struct Y {
  explicit Y() = delete;
};

int main() {
  X x{};   // compiles
  Y y{};   // ill-formed because of explicit!
}
```

Unfortunately, this particular use of the `explicit` keyword is a workaround that fails to address the core of the problem, and yet another rule that is not well-known and can cause confusion.

1.4 Out-of-line default

There are more bear traps that emerge from the current rules. Because a type with a *user-declared* defaulted constructor can be an aggregate, but a type with a *user-provided* defaulted constructor cannot, it depends on the position of the `= default` whether aggregate initialization is ill-formed or not:

```
struct X {
   int i;
   X() = default;
};

struct Y {
  int i;
  Y();
};
Y::Y() = default;

int main() {
  X x{4};   // compiles
  Y y{4};   // ill-formed - not an aggregate!
}
```

1.5 P0960: Initializing aggregates from a parenthesized list of values

The proposal [P0960] allows to perform aggregate initialization from a parenthesized list of values. This new feature, if adopted, would make the issues described in section 1 even more urgent: the user would now run into unintended aggregate initialization not only with the `{...}` syntax, but also with the `(...)` syntax. Specifically, `X x1(3);` from the examples in 1.2 would actually compile, and explicitly deleting constructors of aggregates would no longer have any effect at all. These semantics are broken and need to be fixed.

2 Proposed solution

The solution to all of the above is to modify the definition of *aggregate* as follows:

> An *aggregate* is an array or a class with no ~~user-provided, `explicit`,~~user-declared or inherited constructors, no private or protected non-static data members, no virtual functions, and no virtual, private, or protected base classes.

This makes aggregate initialization ill-formed for any class that has an explicitly defaulted or deleted constructor.

2.1 Simpler and more uniform model for initialization semantics

The proposed change removes all cases in which aggregate initialization could bypass the declared initialization interface of a class. Not only does this make all the code from section 1 behave according to most people's expectations, but it also simplifies the rules, making initialization semantics in C++ more uniform and easier to teach.

The underlying mental model is as follows. A class, by default, gets a set of constructors: a default constructor, and aggregate "constructors" that initialize the fields. If the user declares any constructors themselves, they are taking ownership of the initialization semantics for the class, and the compiler no longer provides these defaults.

This mental model matches most users' expectations. The proposed change would align C++ initialization semantics with this model and remove the currently existing harmful exceptions. We won't have to teach unintuitive rules like "a `=deleted` or not-out-of-line-`=defaulted` constructor, if not marked `explicit`, does not suppress aggregate initialization, but a constructor without these properties does." Instead, we will be able to teach a simple and intuitive rule: "if a class declares any constructors, it will always be initialized through one of those constructors".

Copy and move semantics are notionally distinct from initialization semantics as described above, and we do not propose any changes for the generation of default copy and move constructors.

2.2 Eliminating "accidental aggregates"

The proposed change addresses a particular family of bugs. It often happens that the author of a library type intends to control its instantiation by declaring a constructor, but is not aware of the fact that this does not actually prevent aggregate initialization. This, in turn, allows clients of this type to use it in unintended ways and invoke unintended behaviour. This proposal eliminates such "accidental aggregates".

We found that in real-world codebases, occurrences of "accidental aggregates" that our proposal would fix typically occur more often than cases of code that our proposal would break (see section 3). Aggregates are sometimes useful and have their place in C++. However we believe that they are useful much less often than they are an unwanted mechanism that provides "constructors" that the programmer did not intend.

We found one particularly insidious case of an "accidental aggregate" in the LLVM/Clang codebase. Consider:

```
struct X {
   X() {}
   // some data members...
};
```

This class was initially not an aggregate, because it had a user-provided constructor. Later, someone decided to "modernize" the code base by changing `X() {}` to `X() = default;`. Unbeknownst to them, this silently turned the class into an aggregate, changing its initialization semantics.

3 Breaking changes

We implemented our proposed change in Clang and ran it on the Google codebase. We observed code breakage whenever a type was aggregate-initialized that ceases to be an aggregate with the change proposed here. We found that there is about one such type per 3 MLoC. We believe that this amount of breakage is well justified by the achieved simplification and improvement over the existing language rules.

In this breakage study, we found some relevant use cases that our proposal breaks and that are not already covered by section 1. We discuss those use cases below. They are rare and either contain redundant declarations or exploit the existing language rules to achieve goals that the rules were never meant to support in the first place.

The proposed changes are fully compatible with C++98/03, which states that aggregates cannot have *user-declared* constructors. All breakage only affects post-C++11 code, which changed this rule to *user-provided* (see Appendix). It is this change that we propose to revert.

3.1 Forcing aggregate initialization by deleting the default constructor

Consider an aggregate with data members of built-in type. A programmer might choose to delete the default constructor of such a type to force aggregate initialization, which ensures that data members will always be initialized to prevent indeterminate values:

```
struct X {
  int i, j;
  X() = delete;
};

int main() {
  X x1;        // ill-formed
  X x2{};      // zero-initializes i and j. OK in C++17, ill-formed with this proposal
  X x3{0, 0}; // value-initializes i and j. OK in C++17, ill-formed with this proposal
}
```

This proposal would make both `x2` and `x3` ill-formed, because `X` would no longer be an aggregate. There are better ways to express the intent of forcing initialization of members. One such way are default member initializers:

```
struct X {
  int i = 0, j = 0;
};

int main() {
  X x1;        // zero-initializes i and j. OK in C++17 and with this proposal
  X x2{} ;     // zero-initializes i and j. OK in C++17 and with this proposal
  X x3{0, 0}; // value-initializes i and j. OK in C++17 and with this proposal
}
```

3.2 Redundant constructor declarations

Consider an aggregate with an implicitly deleted default constructor (for example, because of a reference member). The programmer may choose to explicitly declare this property for ”code documentation purposes”, although this is redundant.

Consider:

```
struct X {
  int& i;
  X() = delete;  // redundant declaration; default c'tor would be implicitly deleted
};
```

With the changes proposed here, `X` would cease to be an aggregate, because it has a user-declared constructor:

```
int main() {
  int val = 4;
  X x{val};  // OK in C++17, ill-formed with this proposal
}
```

It is possible to get back the old behaviour by simply removing the redundant constructor declaration:

```
struct X {
  int& i;
};

int main() {
  int val = 4;
  X x{val};  // OK in C++17 and with this proposal
}
```

It is important to note that, conceptually, an implicitly deleted constructor is actually not the same as an explicitly deleted one. By using the latter, the programmer *takes ownership* of the type's initialization semantics. Consider:

```
struct Foo; // not default-constructible

struct Bar {
  Foo foo;
};
```

Here, the compiler generates an implicitly deleted constructor for `Bar` because `Foo` is not default-constructible. If `Foo` is later changed to be default-constructible, the compiler will make `Bar` default-constructible as well — the initialization semantics are *implicit*. On the other hand, if you declare `Bar() = delete;`, then changing `Foo` to be default-constructible will *not* make `Bar` default-constructible as well — the initialization semantics are *user-defined*. Due to this conceptual difference we believe that adding redundant constructor declarations for "code documentation purposes" is not a very strong use case.

3.3 Non-copyable aggregates

A rare, but interesting use of the current rules are types like this:

```
struct X {
  std::string s;
  std::vector<Foo> v;

  // make noncopyable:
  X() = default;
  X(const X&) = delete;
  X(X&&) = default;
};
```

This class is an aggregate in C++17. It contains data members that are potentially expensive to copy. The author explicitly deleted the copy constructor and defaulted the move constructor to prevent such copies.

Our proposal makes aggregate initialization of this class ill-formed. In the mental model that our proposal follows, conceptually an aggregate should only be a "bag of members" with simple, implicit initialization semantics. Therefore, it should only be non-copyable if it has an implicitly deleted copy constructor, such as through a non-copyable data member. By overriding this default, the user is also opting out of the other default (the member-wise aggregate initialization).

It is still possible to manually make an aggregate non-copyable without declaring constructors, for example by adding a non-copyable empty member at the end:

```
struct NonCopyable {
  NonCopyable() = default;
  NonCopyable(const NonCopyable&) = delete;
  NonCopyable(NonCopyable&&) = default;
};

struct X {
  std::string s;
  std::vector<Foo> v;

  [[no_unique_address]] NonCopyable nc;
};
```

In our code breakage study, this particular case accounted for about half of the observed breakages.

3.4 Aggregate initialization becoming value-initialization

For classes with `=default`ed default constructors, this proposal may silently turn aggregate initialization from an empty *braced-init-list* into value initialization.

```
struct X {
  int i, j;
  X() = default;
};

int main() {
  X x{};  // aggregate initialization in C++17; value-initialization with this proposal
}
```

While this is notionally different, there is no change in behaviour: the data members `i` and `j` are zero-initialized in both cases.

3.5 Aggregate initialization becoming a copy constructor call

There is one corner case where the proposed change may silently turn aggregate initialization into a copy constructor call. This can happen if the type has a data member of another type convertible to its own type, and simultaneously a `=default`ed or `=delete`d copy constructor:

```
struct Y;

struct X {
  operator Y();
};

struct Y {
  X x;
  Y(const Y&) = default;
};

int main() {
    Y y{X{}};  // aggregate initialization in C++17; copy constructor call with this proposal
}
```

This introduces a change in behaviour. If the copy constructor is inaccessible or deleted, the initialization will now become ill-formed. Otherwise, it will now call the conversion operator and the copy constructor. We believe that this is actually a fix, since the presence of a user-declared copy constructor suggests that it should have been selected in the first place. The new behaviour is

consistent with the design goal of regularizing initialization semantics and removing all cases where aggregate initialization is applied notwithstanding declared constructors that match the provided arguments.

4 Proposed wording

The proposed changes are relative to the C++ working paper [Smith2018].

Modify **[dcl.init.aggr]** paragraph 1 as follows:

> An *aggregate* is an array or a class (Clause 12) with
>
> — no ~~user-provided, explicit,~~user-declared or inherited constructors (15.1),
>
> — no private or protected non-static data members (Clause 14),
>
> — no virtual functions (13.3), and
>
> — no virtual, private, or protected base classes (13.1).

Modify **[dcl.init.aggr]** paragraph 17 as follows:

> [*Note:* An aggregate array or an aggregate class may contain elements of a class type with a ~~user-provided~~user-declared constructor (15.1). Initialization of these aggregate objects is described in 15.6.1. — *end note*]

Add the following to **[diff.cpp17]** in Annex C, section C.5 C++ and ISO C++ 2017:

> **C.5.6 Clause 11: declarators** **[diff.cpp17.dcl.decl]**
>
> **Affected subclause:** [dcl.init.aggr]
> **Change:** A class that has user-declared constructors is never an aggregate.
> **Rationale:** Remove potentially error-prone aggregate initialization which may apply notwithstanding the declared constructors of a class.
> **Effect on original feature:** Valid C++ 2017 code that aggregate-initializes a type with a user-declared constructor may be ill-formed or have different semantics in this International Standard.

```
struct A {  // Not an aggregate; previously an aggregate
  A() = delete;
};

struct B {  // Not an aggregate; previously an aggregate
  B() = default;
  int i = 0;
};

struct C {  // Not an aggregate; previously an aggregate
  C(C&&) = default;
  int a, b;
};

A a{};                  // ill-formed; previously well-formed
B b = {1};              // ill-formed; previously well-formed
auto* c = new C{2, 3}; // ill-formed; previously well-formed
```

```
struct Y;

struct X {
    operator Y();
};

struct Y {  // Not an aggregate; previously an aggregate
    Y(const Y&) = default;
    X x;
};

Y y{X{}};  // copy constructor call; previously aggregate-initialization
```

Appendix: History of the definition of *aggregate* in C++

For reference, this appendix summarises the evolution of the definition of *aggregate* throughout the different versions of the C++ language.

Original C++98/03 wording:

> An *aggregate* is an array or a class with no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions.

Changes in C++11:

> An *aggregate* is an array or a class with no ~~user-declared~~user-provided constructors, no brace-or-equal-initializers for non-static data members, no private or protected non-static data members, no base classes, and no virtual functions.

The one problematic change that we propose to revert here is the change from *user-declared* to *user-provided* introduced by [N2346]. The original motivation of this change was to allow aggregate initialization for `std::atomic` despite having a `=default`ed constructor for purposes of C compatibility. This motivation became obsolete after C11 introduced its atomic operations library. Thus, all the problems with aggregate initialization that we propose to fix here are essentially the result of a historic accident.

The second change was made through the resolution of core defect [CWG886], which was then reverted in C++14. The resolution of the following core defects also indirectly changed what types can be considered an *aggregate*: [CWG1355] and [CWG1578], which, in turn, was caused by [CWG1301].

Changes in C++14:

> An *aggregate* is an array or a class with no user-provided constructors, ~~no brace-or-equal-initializers for non-static data members,~~ no private or protected non-static data members, no base classes, and no virtual functions.

The restriction introduced in C++11 by the resolution of [CWG886] was found to be problematic and was reverted by [N3653]. The rationale is explained in [N3605].

Changes in C++17:

> An *aggregate* is an array or a class with no user-provided, explicit, or inherited constructors, no private or protected non-static data members, no virtual functions, and no virtual, private, or protected base classes~~, and no virtual functions~~.

The restriction of making classes with `explicit` constructors non-aggregate was introduced by [P0017]. It is a workaround for one of the problems introduced by the C++11 *user-provided* change. Tag types such as `std::piecewise_construct` should not be user-constructible from empty braces, and marking the deleted default constructor `explicit` was introduced as a way to make that possible. This workaround is no longer necessary with our proposed change.

The other changes come from [P0398] and implement an extension that is not impacted by our proposed change.

Document history

— **R0**, 2018-05-07: Initial version

— **R1**, 2018-06-08: Revised wording; added code breakage study; removed discussion of alternative proposal P1090; added appendix.

Acknowledgements

Many thanks to Igor Akhmetov, Dmitry Kozhevnikov, Ville Voutilainen, Nicole Mazzuca, Peter Dimov, Ben Craig, Kévin Alexandre Boissonneault, Michał Dominiak, Mark Zeren, Kate Gregory, Barry Revzin, Agustín Bergé, Tony Van Eerd, Jens Maurer, Gabriel Dos Reis, and Daveed Vandevoorde for their helpful comments and code examples. Many thanks to Alisdair Meredith, Thomas Köppe, Hubert Tong, and Tim Song for their help with the Annex C wording.

References

[CWG1301] Jason Merrill. Core Defect 1301: Value initialization of union. http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1301, 2011 (accessed 2018-06-08).

[CWG1355] Sean Hunt. Core Defect 1355: Aggregates and "user-provided" constructors. http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1355, 2011 (accessed 2018-06-08).

[CWG1578] Alisdair Meredith. Core Defect 1578: Value-initialization of aggregates. http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_closed.html#1578, 2012 (accessed 2018-06-08).

[CWG886] Daniel Krügler. Core Defect 886: Member initializers and aggregates. http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#886, 2009 (accessed 2018-06-08).

[N2346] Lawrence Crowl. Defaulted and Deleted Functions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2346.htm, 2007 (accessed 2018-06-08).

[N3605] Ville Voutilainen. Member initializers and aggregates. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3605.html, 2013 (accessed 2018-06-08).

[N3653] Ville Voutilainen and Richard Smith. Member initializers and aggregates. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3653.html, 2013 (accessed 2018-06-08).

[P0017] Oleg Smolsky. Extension to aggregate initialization. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0017r1.html, 2015 (accessed 2018-06-08).

[P0398] Jens Maurer. Core issue 1518: Explicit default constructors and copy-list-initialization. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0398r0.html, 2016 (accessed 2018-06-08).

[P0960] Ville Voutilainen. Allow initializing aggregates from a parenthesized list of values. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0960r0.html, 2018 (accessed 2018-06-08).

[Smith2018] Richard Smith. Working Draft, Standard for Programming Language C++. https://github.com/cplusplus/draft, 2018 (accessed 2018-06-08).