Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0883R2**
Date: 2019-11-08
Reply to: Nicolai Josuttis (nico@josuttis.de)
Audience: SG1, **LEWG**, LWG
Prev. Version: P0883R1

Fixing Atomic Initialization, Rev2

Currently, std::atomic<> is the only type in C++ where list initialization does not work as expected. In addition, default initialization does not do what people expect, causing nasty unexpected program behavior. This paper proposes to fix that.

Rev2:

Updated due to feedback from LWG in Batavia 2018.

Rev1:

Driven by Feedback from http://wiki.edg.com/bin/view/Wg21jacksonville2018/P0883R0 and http://wiki.edg.com/bin/view/Wg21jacksonville2018/Atomic_init:

- Fix that the constructor "initializes the atomic object with the value of `T{}`".
- Deprecate `atomic_init()` and `ATOMIC_VAR_INIT` at all
- Apply corresponding modifications to `atomic_flag`

Motivation

Currently std::atomic<> is standardized to behave as follows:

```
std::atomic<int> x{};        // does NOT zero initialize

struct counter {
    int external_counters = 0;
    int count = 1;
};
std::atomic<counter> x;      // does not initialize with 0 and 1
std::atomic<counter> y{};    // does not initialize with 0 and 1
```

The reason is that the spec currently requires not to initialize any value inside a default-initialized atomic:

> **32.6.1 Operations on atomic types [atomics.types.operations]:**
>
> `atomic() noexcept = default;`
> 2 Effects: Leaves the atomic object in an uninitialized state. [Note: These semantics ensure compatibility with C. —end note]

This is even worse than undefined behavior; we ***require*** here not to do the right and expected behavior. As Herb Sutter comment with a lot of agreement:

> I have never seen the current behavior of deliberately failing to initialize an atomic<T> (to the obvious default value of T{}) to be anything but a source of user surprise and a bug farm.

And the fact that list initialization in this case doesn't work as everywhere else means that we cannot simply teach C++ the way that using curly braces *always* default/zero initializes (the correct technical term is "value initializes"). This unnecessarily creates confusion and blames C++ as a whole.

Why do we have the existing interface?

The reason for this unexpected behavior of `std::atomic<>` was to get "some planned future compatibility to C" for a common subset of `std::atomic<>`. You should be able to use the aliases e.g.

`atomic_int` to write programs that are valid and have the same semantics in both C and C++.
See also:

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html#DiscussInterop
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3057.html

This is also important for pure C++ programs. As Olivier Giroux wrote in http://lists.isocpp.org/lib/2017/12/4908.php:

> Likely, C++ programmers rely on C compatibility more than they know because C++ code tends to see the OS/platform headers directly. This may be my own brand of idealism, but I would like to see more "atomic_int" and less "volatile int" in those headers in the future. I would not want to turn away from that future.
>
> This being said, it's already the case that "atomic_int" is said to be equivalent to "atomic<int>" in a C++ translation and so it already has more functionality. This is fine because there is still that subset which can be used from both C and C++. I have no problem with initialization being in that set of extra C++ functionality, even if the effect is that the compatible subset diverges further from common C++ usage.

Nevertheless, the topic proposed here was already discussed as a design mistake:

- As part of the proposal of
  www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4130.pdf
  Here are the details:
  http://wiki.edg.com/bin/view/Wg21urbana-champaign/PadThyAtomicsNotes
- There is also a library issue:
  https://cplusplus.github.io/LWG/issue2334
- And this thread:
  http://lists.isocpp.org/lib/2017/12/4854.php

But so far all these discussion never resulted in a useful fix.

Was the intention of this interface successful?

The interesting thing is that at least two compilers don't follow the spec and implement "The Right Thing" (see http://lists.isocpp.org/lib/2017/12/4864.php).

And even worse: The C compatibility issue was not successful, as Herb Sutter wrote in http://lists.isocpp.org/lib/2017/01/1611.php:

> However, as others mentioned, I think this was motivated by having an atomic int shared between C and C++ code. But I don't know that that even ended up being possible, or desirable. We also tried to make threads be the same type in WG14 and WG21. My impression was that those things didn't really work out, and if so perhaps we shouldn't be constrained by them.

and as Hans Boehm stated in http://lists.isocpp.org/lib/2017/01/1634.php:

> Initially <atomic> was designed before WG14 came up with _Atomic. Lawrence and I started out with directions from WG14 not to pursue something along the lines of _Atomic.
>
> A subsequent imperfect, but probably good enough, compatibility proposal was to
>
> ```
> #define _Atomic(T) std::atomic(T)
> ```
>
> in C++. That (and some promotion to the global namespace) is essentially what happens on Android if you include stdatomic.h from C++ code. (See e.g. the beginning of https://github.com/android/platform_bionic/blob/master/libc/include/stdatomic.h. This also requires that C _Atomic and C++ std::atomic share the same bit-level representation.)
>
> All of this is clearly outside the domain of the C++ standard, but is nonetheless often useful, as Jonathan points out.

and as Martin Sebor statet in http://lists.isocpp.org/lib/2017/12/4936.php:

> In response to C11 DR 485, WG14 has decided at the April 3 meeting,
> to acknowledge that the ATOMIC_VAR_INIT macro, besides being broken,
> serves no useful purpose and should be avoided. **The implication is**

> **that C allowing atomics to be initialized just like objects of the underlying non-atomic types, reflecting existing practice**.
>
> To that effect, the upcoming 2017 revision of C will be removing the requirement to use macro to initialize atomic variables. Going forward, WG14 intends to deprecate and ultimately remove the macro.

However Hans Bohm wrote in http://lists.isocpp.org/lib/2017/12/4943.php:

> I'm actually optimistic here that the C compatibility story still matters, and is not a lost cause. We actually do use stdatomic.h from C++ a fair amount, though the approach may be controversial. But this is clearly not currently standard conforming. I still need to write a paper.
>
> But in any case, I don't see a conflict here. Much of the status quo was predicated on the assumption that atomics might contain a lock requiring nontrivial initialization. I think we've generally given up on that, especially on the C side, and WG14 seems happy to require zero initialization for atomic_int (or _Atomic(int) or _Atomic int). See http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_422

The key point is here that we all seem to agree that a fix to the **C++-specific API** is fine.

How should we fix it?

In many C++ programs compatibility to C is not an issue at all. And at least in these cases C++ must do the right thing.

Of course, we could provide some wording that when sharing an atomic between C and C++ the initialization depends on which language calls the initialization, but I don't any need for it, because what we standardize with std::atomic<> is a feature that can only be used in C++ (templates are not available in C at all). We might need a note at some place, but I see even no need for that.

If we use pure C++ to initialize an atomic object, I see no reason not to give it a useful initial value via the constructor. And that's all what is proposed here.

One different suggestion when discussing N4130 was:

> C++'s atomic's ctor should call atomic_init, except for the T's where there exists a C-alias or if its constructed with the Macro ATOMIC(T*) in C++.

I don't think that this is enough, because C++ programmers will assume that any atomic type, even atomic<int> is a valid C++ type and follows the fundamental design principles of C++. I don't care about any macro or so. I am talking about an initialization via templates which is not valid C.

So, no "compromise" for the mixed C/C++ case proposed here. This is just a fix for the pure C++ type std::atomic<>. Users who want compatibility can follow the C guidance in either language.

noexcept?

Should the default constructor have a noexcept clause?

- It could always because it would be strange if trivially copyable types throw.
- It could conditionally, because his again is a wrapper case, which we already had a couple of times in the library. And as a result the type should guarantee not to throw if the underlying type does.
- It could follow N3279 to never throw conditional except in move or swap operations.

Usually I would follow the wrapper case argument, although it is not officially introduced yet (P0884 proposes a corresponding guideline).

An important argument for me is that we should have the same effect and guarantees for:

```
std::atomic<T> x;
```

and:

```
std::atomic<T> x{T()};
```

Note that the latter might throw if the default constructor throws, because the exception is thrown before the initialization starts.

So the further should only guarantee not to throw if the underlying default constructor gives this guarantee.

Proposed Wording

(All against N4835)

Note this overrules the proposed solution of https://cplusplus.github.io/LWG/issue2334 to require default initialization. It is the clear intention of this proposal to let the default constructor value initialize the underlying object.

In 31.2 Header <atomic> synopsis [atomics.syn]

Strike and move to Annex D as follows:

```
namespace std {
  …
 template<class T>
  void atomic_init(volatile atomic<T>*, typename atomic<T>::value_type)
                                                              noexcept;
 template<class T>
  void atomic_init(atomic<T>*, typename atomic<T>::value_type) noexcept;
  …
 // 32.6.1, initialization
 #define ATOMIC_VAR_INIT(value) see below
  …
 #define ATOMIC_FLAG_INIT see below
  …
};
```

31.8 Class template atomic [atomics.types.generic]

In the synopsis change:

```
namespace std {
 template <class T> struct atomic {
  …
  constexpr atomic() noexcept(is_nothrow_default_constructible_v<T>)
           = default;
  …
 };
}
```

Strike and move to Annex D:

~~#define ATOMIC_VAR_INIT(value) *see below*~~

~~2 The macro expands to a token sequence suitable for constant initialization of an atomic variable of static storage duration of a type that is initialization-compatible with value. [*Note*: This operation may need to initialize locks. —*end note*] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [*Example*:~~

```
atomic<int> v = ATOMIC_VAR_INIT(5);
```

~~—*end example*]~~

31.8.1 Operations on atomic types [atomics.types.operations]:

```
constexpr atomic() noexcept(is_nothrow_default_constructible_v<T>) = default;
```

Mandates: is_default_constructible_v<T> is true.

Effects: ~~Leaves the atomic object in an uninitialized state. [Note: These semantics ensure compatibility with C. — end note]~~
Initializes the atomic object with the value of T(). Initialization is not an atomic operation (4.7).

31.8.2 Specializations for integers [atomics.types.int]

In the synopsis change:

```
namespace std {
 template <> struct atomic<integral> {
  …
  constexpr atomic() noexcept = default;
  …
 };
}
…
```

And:

2 The atomic integral specializations are standard-layout structs. They each have ~~a trivial default constructor and~~ a trivial destructor.

31.8.3 Specializations for floating-point types [atomics.types.float]

In the synopsis change:

```
namespace std {
 template <> struct atomic<floating-point> {
  …
  constexpr atomic() noexcept = default;
  …
 };
}
…
```

And:

2 The atomic floating-point specializations are standard-layout structs. They each have ~~a trivial default constructor and~~ a trivial destructor.

31.8.4 Partial specialization for pointers [atomics.types.pointer]

In the synopsis change:

```
namespace std {
 template <class T> struct atomic<T*> {
  …
  constexpr atomic() noexcept = default;
  …
 };
}
…
```

And:

1 There is a partial specialization of the atomic class template for pointers. Specializations of this partial specialization are standard-layout structs. They each have ~~a trivial default constructor and~~ a trivial destructor.

20.11.8.1 Atomic specialization for shared_ptr [util.smartptr.atomic.shared]

In the synopsis change:

```
namespace std {
 template <class T> struct atomic<shared_ptr<T>> {
  …
  constexpr atomic() noexcept = default;
  …
 };
}

constexpr atomic() noexcept = default;
```

1 Effects: Initializes `p{}`.

20.11.8.2 Atomic specialization for weak_ptr [util.smartptr.atomic.weak]

In the synopsis change:

```
namespace std {
 template <class T> struct atomic<weak_ptr<T>> {
  …
  constexpr atomic() noexcept = default;
  …
 };
```

```
}

constexpr atomic() noexcept = default;
```
1 Effects: Initializes `p{}`.

In 31.9 Non-member functions [atomics.nonmembers]

remove (and move with modified Effects clause to Annex D):

~~`template<class T>`~~
~~`void atomic_init(volatile atomic<T>* object,`~~
~~`typename atomic<T>::value_type desired) noexcept;`~~
~~`template<class T>`~~
~~`void atomic_init(atomic<T>* object,`~~
~~`typename atomic<T>::value_type desired) noexcept;`~~

~~*Effects:* Non-atomically initializes *object with value desired. This function shall only be applied to objects that have been default constructed, and then only once. [*Note:* These semantics ensure compatibility with C. — *end note*] [*Note:* Concurrent access from another thread, even via an atomic operation, constitutes a data race. — *end note*]~~

Modify 31.10 Flag type and operations [atomics.flag]
as follows:

```
namespace std {
 struct atomic_flag {
  bool test_and_set(memory_order = memory_order::seq_cst) volatile noexcept;
  bool test_and_set(memory_order = memory_order::seq_cst) noexcept;
  void clear(memory_order = memory_order::seq_cst) volatile noexcept;
  void clear(memory_order = memory_order::seq_cst) noexcept;

  constexpr atomic_flag() noexcept = default;
  atomic_flag(const atomic_flag&) = delete;
  atomic_flag& operator=(const atomic_flag&) = delete;
  atomic_flag& operator=(const atomic_flag&) volatile = delete;
 };
 …
}
```

§3: The atomic_flag type is a standard-layout struct. It has a ~~trivial default constructor and~~ a trivial destructor.

Add:

`constexpr atomic_flag::atomic_flag() noexcept;`
Effects: Initializes `*this` to the clear state.

Strike (and move without last sentence to Annex D (see below)):

~~`#define ATOMIC_FLAG_INIT see below`~~

~~§4: *Remarks:* The macro `ATOMIC_FLAG_INIT` shall be defined in such a way that it can be used to initialize an object of type `atomic_flag` to the clear state. The macro can be used in the form:~~
~~`atomic_flag guard = ATOMIC_FLAG_INIT;`~~
~~It is unspecified whether the macro can be used in other initialization contexts. For a complete static-duration object, that initialization shall be static. Unless initialized with `ATOMIC_FLAG_INIT`, it is unspecified whether an `atomic_flag` object has an initial state of set or clear.~~

Add in Annex D:

The header <atomic> (31.2) has the following additions:

```
namespace std {
  …
  template<class T>
   void atomic_init(volatile atomic<T>*, typename atomic<T>::value_type)
                                                                  noexcept;
  template<class T>
   void atomic_init(atomic<T>*, typename atomic<T>::value_type) noexcept;

  // ?.?.?, initialization
  #define ATOMIC_VAR_INIT(value) see below
  …
  #define ATOMIC_FLAG_INIT see below
}
```

```
template<class T>
void atomic_init(volatile atomic<T>* object,
                 typename atomic<T>::value_type desired) noexcept;
template<class T>
void atomic_init(atomic<T>* object,
                 typename atomic<T>::value_type desired) noexcept;
```

Effects: Equivalent to: `atomic_store_explicit(object, desired, memory_order_relaxed);`

`#define ATOMIC_VAR_INIT(value)` *see below*

The macro expands to a token sequence suitable for constant initialization of an atomic variable of static storage duration of a type that is initialization-compatible with value. [*Note*: This operation may need to initialize locks. —*end note*] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [*Example*:

```
atomic<int> v = ATOMIC_VAR_INIT(5);
```

—*end example*]

#define ATOMIC_FLAG_INIT *see below*

The macro `ATOMIC_FLAG_INIT` shall be defined in such a way that it can be used to initialize an object of type `atomic_flag` to the clear state. The macro can be used in the form:

```
atomic_flag guard = ATOMIC_FLAG_INIT;
```

It is unspecified whether the macro can be used in other initialization contexts. For a complete static-duration object, that initialization shall be static.

Feature Test Macro

The availability of this fix will make it a lot easier to program atomics safely. Thus, a feature macro is a must. It is:

```
__cpp_lib_atomic_value_initialization
```

Headers:

```
<atomic>, <memory>
```

Other Sources

https://cplusplus.github.io/LWG/issue2334

www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4130.pdf

https://developers.redhat.com/blog/2016/01/14/toward-a-better-use-of-c11-atomics-part-1/
https://developers.redhat.com/blog/2016/01/19/toward-a-better-use-of-c11-atomics-part-2/

Acknowledgements

Thanks to a lot of people who discussed the issue, proposed information and possible wording. Especially: JF Bastien, Hans Boehm, Casey Carter, Peter Dimov, Arthur O'Dwyer, Olivier Giroux, Billy O'Neal, Martin Sebor, Herb Sutter, Jeffrey Yasskin. Forgive me if I forgot anybody.