N4130: Pad Thy Atomics

2014-09-01
JF Bastien jfb@google.com
Olivier Giroux ogiroux@nvidia.com

1. Introduction

This paper discusses multiple issues with `atomic` and proposes a few approaches to fix all of the issues. No definite conclusion is reached: the authors hope to obtain guidance from SG1 at the upcoming Redmond meeting on alternatives marked with **[TBD]**.

1.1. Background

1.1.1. Uninitialized State

LWG 2334: `atomic`'s default constructor requires "uninitialized" state even for types with non-trivial default-constructor. According to 29.6.5 [atomics.types.operations.req] p4:

> `A ::A () noexcept = default;`
> *Effects*: leaves the atomic object in an uninitialized state. [*Note*: These semantics ensure compatibility with C. — *end note*]

The user-defined default constructor is currently never called. This requirement is for C compatibility, but C++ types also currently fall under this requirement.
This led to a further discussion in the Rapperswil SG1 meeting about `atomic<T>` either being POD even if `T` is not POD, or not POD when `T` is POD, both of which seem counter-intuitive. This behavior is probably not portable.

1.1.2. Structs Comparing Equal

WG14 DR 431: `atomic_compare_exchange`: What does it mean to say two `structs` compare equal? The main concern is on comparing padding bit, whose value is unspecified, yet are compared because comparison through `atomic_compare_exchange` is done by `memcpy`. The standard mentions that `atomic_compare_exchange` is used in a loop and will therefore converge, but usage in a loop isn't necessarily true for all valid uses. The same question applies for `unions` in `atomic`, class-type is therefore used in this paper.
A sample problematic example:

```
struct C { char c; /* padding bits may live here */ int i; };
atomic<C> ac;
```

1.1.3. 1.1.3.C Compatibility

We want to keep compatibility with C. Specifically, `ATOMIC_INIT` should fall into this solution.
Note: default-initialization leaves padding unspecified, whereas zero-initialization sets padding bits.

1.2. Objective

An ideal resolution makes `atomic` construction work in a non-surprising way while ensuring that padding bits of class-types used atomically are always well-defined, making compare-and-exchange operations with these types perform as best as the underlying ISA can guarantee. No full solution is advocated for at the moment since the C compatibility issue doesn't seem to be resolvable without a change that involves more than just `atomic`, and the value return issue doesn't currently have a proposed solution.

2. Default Construction

The standard currently specifies `atomic`'s default constructor as:

```
atomic() noexcept = default;
```

Implementations of `atomic` which aren't lock-free either use global locks partitioned on a hash of the object's address, or they add a lock to every `atomic` object which isn't lock-free. The lock is an implementation detail, but it isn't specified as trivially constructible (case in point: `mutex` isn't trivially constructible).

This seems wrong because an implementation which uses a per-object lock which isn't trivially constructible makes the default constructor magically non-trivial even though it is specified to be trivial. We could either **[TBD]**:

1. Specify that `atomic` always has a non-trivial default constructor.
2. Both:
    - Mandate that implementation-defined locks be trivially constructible to avoid the magical behavior.
    - Specialize `atomic` for class-types that aren't trivially constructible, making `is_pod<atomic<T>> == is_pod<T>`.

LWG 2334 proposes changing 29.6.5 [atomics.types.operations.req] p4 to default-initialize the `atomic` object. To guarantee that padding bits are of known value it could **[TBD]**:

1. First zero-initialize (setting padding), then default-initialize `atomic` objects of class-types.
2. Mandate that users `memset(this, init_val, sizeof(*this));` in the default constructor of class-types which they intend to use in `atomic`s. This default constructor will be invoked because of the proposed change. There should be notes explaining the following (either here or closer to compare-and-exchange):
    - How users should use `memset` in the default constructor to make padding bits predictable.
    - Demonstrate that padding bits have known values when using a compare-and-exchange loop which reloads the value.
3. Keep the behavior as it currently is, not call user-defined default constructors for `atomic` objects, and instead zero-initialize the entire object. In this case we should

add a non-normative note that a compiler should warn when their user-defined default constructor won't be called.

Note that the above proposal regarding `atomic` instances of types with user-defined default constructors has no impact on C compatibility.

3. Value-Initialization, atomic_init and Assignment

LWG 2334 proposes changing value-initialization (`constexpr A::A(C desired) noexcept;`) to direct-initialize the value referred to by the `atomic` object with the value desired. This is flawed for class-types, because `T` has to be trivially-copyable and the defaulted copy constructor only does member-wise copying. The same applies for `atomic_init` as well as assignment (`operator=`). These three operations should instead be defined in terms of either **[TBD]**:

1. `memcpy`.
2. `obj2.store(obj1.load())`.

Side-effect of note in this change:

This affects the active type of an `atomic union` as implemented in some compilers, and makes `atomic<union T>` behave differently from `union T` in these implementations: type-punning of `atomic<union T>` works through value-initialization, whereas non-`atomic union T` can't use anything but the active type of the `union` because most implementations don't propagate type information across `memcpy`. This behavior isn't guaranteed by the standard.

```
union U { float f; uint32_t i; };
U u = { 1. };
atomic<U> au;
au.store(u);
cout << u.i << endl; // Undefined.
cout << au.load().i << endl; // Defined on most implementations.
```

4. Load and Store

Load and store operations on class-types should be specified to copy all padding bits (through `memcpy`), this currently isn't the case. The above proposal for value-initialization, `atomic_init` and assignment relies on this change to load and store.

Is this acceptable? **[TBD]**

5. Compare and Exchange

5.1. Fixing Compare and Exchange

Compare-and-exchange is already defined in terms of `memcmp` and `memcpy` in p23. Should this be made clearer since it's merely an example in a note **[TBD]**:

> [*Note*: For example, the effect of `atomic_compare_exchange_strong` is
>
> ```
> if (memcmp(object, expected, sizeof(*object)) == 0)
> memcpy(object, &desired, sizeof(*object));
> else
> memcpy(expected, object, sizeof(*object));
> ```
>
> — *end note*] [Example: the expected use of the compare-and-exchange operations is as follows. The compare-and-exchange operations will update expected when another iteration of the loop is needed.
>
> ```
> expected = current.load();
> do {
>   desired = function(expected);
> } while (!current.compare_exchange_weak(expected, desired));
> ```
>
> — end example]

29.6.5 [atomics.types.operations.req] p26 has note:

> [*Note*: The `memcpy` and `memcmp` semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with `operator==` if the underlying type has padding bits, trap bits, or alternate representations of the same value. Thus, `compare_exchange_strong` should be used with extreme care. On the other hand, `compare_exchange_weak` should converge rapidly. — *end note*]

Should we remove it, and replace it with a note explaining how compare-and-exchange works with padding bits (see the suggestion above for default construction)? **[TBD]**

The use of "trap bits" and "alternate representations of the same value" needs to be clarified, or removed **[TBD]**. It could be referring to:

- Floating-point values in a struct with signaling NaN causing a trap.
- Load/store normalizing NaNs.
- Load/store flushing denormals to zero.

The proposed change would affect this behavior: `atomic<T>` won't behave the same as a plain `T`.

5.2. Disallowing Compare and Exchange

Another option discussed on the mailing is to disallow `atomic<T>`, or simply disallow compare-and-exchange, if `has_padding_bits<T>` (this trait was propose in a recent paper about hashing). This has portability implications, Peter Dimov points out that this would leave `atomic</*unsigned*/ char>` as the only strictly portable construct. All types can have padding bits except the character types. From 3.9.1 p1:

> For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types.

Should this option be proposed, and can 3.9.1 p1 be modified to reduce the portability issue? **[TBD]**

6. Parameter Pass-by-Value, and Value Return

Atomic value-initialization, assignment, store, exchange as well as compare-and-exchange currently take their `expected` parameter by value, meaning that the padding bits aren't known and `memcpy` won't necessarily preserve the right padding bits. The `expected` parameter for these five functions could be changed to be a reference instead. **[TBD]**

Atomic load and exchange return by value, which will lose padding bits. Is there a way to avoid this while keeping the same function signature? **[TBD]**

7. C Compatibility

We want to keep compatibility with C. Specifically, `ATOMIC_INIT` says:

> 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.

We can't guarantee that this properly initializes padding: 8.5.1 [dcl.init.aggr] doesn't specify what the value of padding bits or anonymous bitfield members are, neither does C99's designated initializers specification.

[dcl.init.aggr] could be modified to specify the value of padding bits, and C would have to be modified in the same way. This change would affect more than just `atomic`s containing class-types. Should it be proposed? **[TBD]**

❧