

# Atomic minimum/maximum

Document #: P0493R3  
Date: 2021-12-02  
Project: Programming Language C++  
Audience: WG21 SG21 (Contracts)  
Reply-to: Al Grant  
<[al.grant@arm.com](mailto:al.grant@arm.com)>  
Bronek Kozicki  
<[brok@spamcop.net](mailto:brok@spamcop.net)>  
Tim Northover  
<[tnorthover@apple.com](mailto:tnorthover@apple.com)>

## Contents

1	Abstract	1
2	Changelog	1
3	Introduction	2
4	Background and motivation	2
5	The problem of conditional write	2
6	Infix operators in <code>&lt;atomic&gt;</code> and <code>min/max</code>	3
7	Motivating example	4
8	Implementation experience	5
9	Acknowledgments	5
10	Changes to the C++ standard	5
11	References	8

## 1 Abstract

Add integer *max* and *min* operations to the set of operations supported in `<atomic>`. There are minor adjustments to function naming necessitated by the fact that `max` and `min` do not exist as infix operators.

## 2 Changelog

- Revision R3, published 2021-12-??
  - Change formatting
  - Revert to require *read-modify-write* semantics, based on SG1 feedback
  - Remove `replace_key` functions, based on SG1 feedback
  - Remove one (exceedingly long) motivating example
  - Rewrite other motivating example in modern C++

- Rebase on draft [N4901]
- Add floating numbers support to wording
- Add example implementation based on CAS loop
- Add benchmark comparing intrinsic vs CAS-loop implementation **TODO**
- Revision R2, published 2021-05-11
  - Change proposal to make the store unspecified if the value does not change
  - Align with C++20
- Revision R1, published 2020-05-08
  - Add motivation for defining new atomics as read-modify-write
  - Clarify status of proposal for new-value-returning operations.
  - Align with C++17.
- Revision R0 pulished 2016-11-08
  - Original proposal

## 3 Introduction

This proposal extends the atomic operations library to add atomic maximum/minimum operations. These were originally proposed for C++ in [N3696] as particular cases of a general “priority update” mechanism, which atomically combined reading an object’s value, computing a new value and conditionally writing this value if it differs from the old value.

In revision R2 of this paper we have proposed atomic maximum/minimum operations where it is unspecified whether or not the store takes place if the new value happens to be the same as the old value. This has caused contention in LEWG, but upon further discussion in SG1, turned out to be not needed - as discussed in section 5.

## 4 Background and motivation

Atomic addition (*fetch-and-add*) was introduced in the NYU Ultracomputer [Gottlieb 1982], has been implemented in a variety of hardware architectures, and has been standardized in C and C++. Atomic maximum/minimum operations (*fetch-and-max*, *fetch-and-min*) have a history almost as long as atomic addition, e.g. see [Lipovski 1988], and have also been implemented in various hardware architectures but are not currently standard in C and C++. This proposal fills the gap.

Atomic maximum/minimum operations are useful in a variety of situations in multithreaded applications:

- optimal implementation of lock-free shared data structures - as in the motivating example later in this paper
- reductions in data-parallel applications: for example, **OpenMP** supports maximum as a reduction operation
- recording the maximum so far reached in an optimization process, to allow unproductive threads to terminate
- collecting statistics, such as the largest item of input encountered by any worker thread.

Atomic maximum/minimum operations already exist in several other programming environments, including **OpenCL**, and in some hardware implementations. Application need, and availability, motivate providing these operations in C++.

The proposed language changes add atomic max/min to `<atomic>` for builtin types, including integral, pointer and floating point.

## 5 The problem of conditional write

The existing atomic operations (e.g. `fetch_and`) have the effect of a *read-modify-write*, irrespective of whether the value changes. This is how atomic max/min are defined in several APIs (OpenCL, CUDA, C++AMP, HCC)

and in several hardware architectures (ARM, RISC-V). However, some hardware (POWER) implements atomic max/min as an atomic *read-and-conditional-store*.

If we look at an example CAS-loop implementation of this proposal, it is easy to see why such *read-and-conditional-store* can be more efficient.

### 5.1 Example CAS-loop implementation with *read-modify-write*

In this version we are performing an unconditional store, which means all writers need exclusive cache line access. This may result in excessive writer contention.

```
template <typename T>
T atomic_fetch_max_explicit(atomic<T*> pv,
                           typename atomic<T*>::value_type v,
                           memory_order m) noexcept {
    auto t = pv->load(memory_order_relaxed);
    while (!pv->compare_exchange_weak(t, max(v, t), m, memory_order_relaxed))
        ;
    return t;
}
```

### 5.2 Example CAS-loop implementation with *read-and-conditional-store*

Note the condition of the `while` loop below. It skips skip write entirely if `pv` is already equal to `max(v, t)`. This significantly reduces writer contention.

```
template <typename T>
T atomic_fetch_max_explicit(atomic<T*> pv,
                           typename atomic<T*>::value_type v,
                           memory_order m) noexcept {
    auto t = pv->load(m);
    while (max(v, t) != t) {
        if (pv->compare_exchange_weak(t, v, m, memory_order_relaxed))
            break;
    }
    return t;
}
```

If we require *read-modify-write*, this *could* be a non-conforming implementation. However if we add a dummy write such as `pv->fetch_add(0, m)` in any point of the above algorithm (presumably only if `m` contains release) then it would be conforming again, since it is unobservable where *exactly* such a store happened. Indeed, on some architectures, it is also unobservable whether or not a store took place, *if the value does not change* as a result of store.

Similarly, given an architecture which implements this instruction in hardware, but with *read-and-conditional-store* semantics, a conforming *read-modify-write* `fetch_max()` can be implemented on top of such an instruction, e.g. with an addition of a dummy store.

For this reason **and** for consistency with all other atomic instructions, we have decided to use *read-modify-write* semantics for the new atomic minimum/maximum.

## 6 Infix operators in `<atomic>` and `min/max`

The current `<atomic>` provides atomic operations in several ways:

- as a named non-member function template e.g. `atomic_fetch_add` returning the old value
- as a named member function template e.g. `atomic<T*>::fetch_add()` returning the old value

— as an overloaded compound operator e.g. `atomic<T>::operator+=( )` returning the **new** value

Adding ‘max’ and ‘min’ versions of the named functions is straightforward. Unlike the existing atomics, max/min operations exist in signed and unsigned flavors. The atomic type determines the operation. There is precedent for this in C, where all compound assignments on atomic variables are defined to be atomic, including sign-sensitive operations such as divide and right-shift.

The overloaded operator `atomic<T>::operator key=(n)` is defined to return the new value of the atomic object. This does not correspond directly to a named function. For `max` and `min`, we have no infix operators to overload. So if we want a function that returns the new value we would need to provide it as a named function. However, for all operators the new value can be obtained as `fetch_key(n) key n`, (the standard defines the compound operator overloads this way) while the reverse is not true for non-invertible operators like ‘and’ or ‘max’.

Thus new functions returning the new result would add no significant functionality other than providing one-to-one equivalents to `<atomic>` existing compound operator overloads. Revision R2 of this paper tentatively suggested such functions, named `replace_key` (following some of the early literature on atomic operations - [Kruskal 1986] citing [Draughon 1967]). Having discussed this in SG1, the authors have decided *not* to propose addition of extra functions and correspondingly they have been *removed* in revision R3. This same result can be obtained by the user with a simple expression such as `max(v.fetch_max(x), x)` or `min(v.fetch_min(x), x)`.

During discussion in SG1, it was suggested that a new paper could be written proposing `key_fetch` functions returning new values. This is not such paper.

## 7 Motivating example

Atomic fetch-and-max can be used to implement a lockfree bounded multi-consumer, multi-producer queue. Below is an example based on [Gong 1990]. Note, the original paper assumed existence of `EXCHANGE` operation which in practice does not exist on many platforms. Here this was replaced by a two-step read and write, in addition to translation from C to C++. For this reason the correctness proof from [Gong 1990] does not apply.

```
template <typename T, size_t Size>
struct queue_t {
    static_assert(std::is_nothrow_default_constructible_v<T>);
    static_assert(std::is_nothrow_copy_constructible_v<T>);
    static_assert(std::is_nothrow_swappable_v<T>);

    using elt = T;
    static constexpr int size = Size;

    struct entry {
        elt item {}; // a queue element
        std::atomic<int> tag {-1}; // its generation number
    };

    entry elts[size] = {}; // a bounded array
    std::atomic<int> back {-1};

    friend void enqueue(queue_t& queue, elt x) noexcept {
        int i = queue.back.load() + 1; // get a slot in the array for the new element
        while (true) {
            // exchange the new element with slots value if that slot has not been used
            int empty = -1; // expected tag for an empty slot
            auto& e = queue.elts[i % size];
            // use two-step write: first store an odd value while we are writing the new element
            if (std::atomic_compare_exchange_strong(&e.tag, &empty, (i / size) * 2 + 1)) {
                using std::swap;
```

```

        swap(x, e.item);
        e.tag.store((i / size) * 2);           // done writing, switch tag to even (ie. ready)
        break;
    }
    ++i;
}
std::atomic_fetch_max(&queue.back, i); // reset the value of back
}

friend auto dequeue(queue_t& queue) noexcept -> elt {
    while (true) {
        int range = queue.back.load();           // keep trying until an element is found
        for (int i = 0; i <= range; i++) {       // search up to back slots
            int ready = (i / size) * 2;          // expected even tag for ready slot
            auto& e = queue.elts[i % size];
            // use two-step read: first store -2 while we are reading the element
            if (std::atomic_compare_exchange_strong(&e.tag, &ready, -2)) {
                using std::swap;
                elt ret{};
                swap(ret, e.item);
                e.tag.store(-1);                 // done reading, switch tag to -1 (ie. empty)
                return ret;
            }
        }
    }
}
};

```

## 8 Implementation experience

The required intrinsics have been added to Clang.

## 9 Acknowledgments

This paper benefited from discussion with Mario Torrecillas Rodriguez, Nigel Stephens, Nick Maclaren, Olivier Giroux and Gašper Ažman.

## 10 Changes to the C++ standard

The following text outlines the proposed changes, based on [N4901].

### 31: Atomic operations library [atomics]

#### 31.2: Header synopsis [atomics.syn]

— Add following free functions, immediately below *atomic\_fetch\_xor\_explicit*:

```

namespace std {
    // [atomic.nonmembers], non-member functions
    ...
    template<class T>
        T atomic_fetch_max(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
    template<class T>
        T atomic_fetch_max(atomic<T>*, typename atomic<T>::value_type) noexcept;

```

```

template<class T>
    T atomic_fetch_max_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
                                memory_order) noexcept;
template<class T>
    T atomic_fetch_max_explicit(atomic<T>*, typename atomic<T>::value_type,
                                memory_order) noexcept;
template<class T>
    T atomic_fetch_min(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
    T atomic_fetch_min(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
    T atomic_fetch_min_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
                                memory_order) noexcept;
template<class T>
    T atomic_fetch_min_explicit(atomic<T>*, typename atomic<T>::value_type,
                                memory_order) noexcept;
...
}

```

### 31.7.3: Specializations for integral types [atomics.ref.int]

— Add following public functions, immediately below *fetch\_xor*:

```

namespace std {
    template <> struct atomic_ref<integral> {
        ...
        integral fetch_max(integral, memory_order = memory_order_seq_cst) const noexcept;
        integral fetch_min(integral, memory_order = memory_order_seq_cst) const noexcept;
        ...
    };
}

```

— Change:

- 6 *Remarks:* ~~For~~ Except for *fetch\_max* and *fetch\_min*, for signed integer types, the result is as if the object value and parameters were converted to their corresponding unsigned types, the computation performed on those types, and the result converted back to the signed type.

### 31.7.4: Specializations for floating-point types [atomics.ref.float]

— Add following public functions, immediately below *fetch\_sub*:

```

namespace std {
    template <> struct atomic_ref<floating-point> {
        ...
        floating-point fetch_max(floating-point, memory_order = memory_order_seq_cst) const noexcept;
        floating-point fetch_min(floating-point, memory_order = memory_order_seq_cst) const noexcept;
        ...
    };
}

```

### 31.7.5: Partial specialization for pointers [atomics.ref.pointer]

— Add following public functions, immediately below *fetch\_sub*:

```

namespace std {
    template <class T> struct atomic_ref<T *> {
        ...
        T* fetch_max(T *, memory_order = memory_order::seq_cst) const noexcept;
    };
}

```

```

    T* fetch_min(T *, memory_order = memory_order::seq_cst) const noexcept;
};
}

```

### 31.8.3: Specializations for integers [atomics.types.int]

— Add following public functions, immediately below *fetch\_xor*:

```

namespace std {
    template <> struct atomic<integral> {
        ...
        integral fetch_max(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        integral fetch_max(integral, memory_order = memory_order_seq_cst) noexcept;
        integral fetch_min(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        integral fetch_min(integral, memory_order = memory_order_seq_cst) noexcept;
        ...
    };
}

```

— In table 148, [tab:atomic.types.int.comp], add the following entries (with empty *Op* column):

key	Op	Computation
max		maximum std::max
min		minimum std::min

— Change:

- 8 *Remarks:* ~~For~~ Except for *fetch\_max* and *fetch\_min*, for signed integer types, the result is as if the object value and parameters were converted to their corresponding unsigned types, the computation performed on those types, and the result converted back to the signed type.

### 31.8.4: Specializations for floating-point types [atomics.types.float]

— Add following public functions, immediately below *fetch\_sub*:

```

namespace std {
    template <> struct atomic<floating-point> {
        ...
        floating-point fetch_max(floating-point, memory_order = memory_order_seq_cst) volatile noexcept;
        floating-point fetch_max(floating-point, memory_order = memory_order_seq_cst) noexcept;
        floating-point fetch_min(floating-point, memory_order = memory_order_seq_cst) volatile noexcept;
        floating-point fetch_min(floating-point, memory_order = memory_order_seq_cst) noexcept;
        ...
    };
}

```

### 31.8.5: Partial specialization for pointers [atomics.types.pointer]

— Add following public functions, immediately below *fetch\_sub*:

```

namespace std {
    template <class T> struct atomic<T*> {
        ...
        T* fetch_max(T*, memory_order = memory_order_seq_cst) volatile noexcept;
        T* fetch_max(T*, memory_order = memory_order_seq_cst) noexcept;
        T* fetch_min(T*, memory_order = memory_order_seq_cst) volatile noexcept;
        T* fetch_min(T*, memory_order = memory_order_seq_cst) noexcept;
    };
}

```

```

    ...
};
}

```

— In table 149, [tab:atomic.types.pointer.comp], add the following entries (with empty Op column):

key	Op	Computation
max		maximum <code>std::max</code>
min		minimum <code>std::min</code>

## 11 References

- [Draughon 1967] E. Draughon, Ralph Grishman, J. Schwartz, and A. Stein. Programming Considerations for Parallel Computers.  
<https://nyuscholars.nyu.edu/en/publications/programming-considerations-for-parallel-computers>
- [Gong 1990] Chun Gong and Jeanette M. Wing. A Library of Concurrent Objects and Their Proofs of Correctness.  
<http://www.cs.cmu.edu/~wing/publications/CMU-CS-90-151.pdf>
- [Gottlieb 1982] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer.  
<https://ieeexplore.ieee.org/document/1676201>
- [Kruskal 1986] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient Synchronization on Multiprocessors with Shared Memory.  
<https://dl.acm.org/doi/10.1145/48022.48024>
- [Lipovski 1988] G. J. Lipovski and Paul Vaughan. A Fetch-And-Op Implementation for Parallel Computers.  
<https://ieeexplore.ieee.org/document/5249>
- [N3696] Bronek Kozicki. 2013-06-26. Proposal to extend atomic with priority update functions.  
<https://wg21.link/n3696>
- [N4901] Thomas Köppe. 2021-10-22. Working Draft, Standard for Programming Language C++.  
<https://wg21.link/n4901>