Safer `atomic_ref::address` (FR-030-310)

Document #:	P3936R1
Date:	2026-01-27
Programming Language C++	
Audience:	LWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Gonzalo Brito Gadeschi <gonzalob@nvidia.com>

Abstract

This paper proposes to resolve FR-030-310 by returning a `void*` from `atomic_ref::address`

Revisions

R1

- Wording fixes
- rename `address_return_t` and make it exposition only per LEWG guidance

Motivation

`atomic_ref<T>::address` currently returns a `T*`. This function was added by P2835R7 [1], which cover the use cases for this function. In short, we might want to know the address for hashing for contention-aware data structure, indexing an array atomically, etc.

However, it is generally unsafe to access the object at `address` directly, and so, the NB comment argues that `address` is maybe too convenient. IE, it is easy to use incorrectly, and hard to use correctly. A main concern is that any other `atomic_ref` over the same object would no longer guarantee atomic access.

These observation were made in the original paper.

> The name and return type should prevent accidental misuse that could result from accessing the object through the address while there are still live `atomic_ref` that reference it.

However, of the options explored at the time, none seemed compelling.

- `uintptr_t` is not mandated by the standard and would not work during constant evaluation.
- `void*` would not allow us to get a `T*` back during constant evaluation.

FR-030-310 proposes to mandate `uintptr_t`. However, EWG had no appetite to do it in the C++26 time frame (over concernns that there might exist architectures in which pointers can be larger than the largest integer type). And it would not resolve the concerns that it would not be usable during constant evaluation.

But... as it turns out, `void*` is now a suitable option. Indeed cast from `void*` to `T*` are now supported in constant evaluation. This feature was added by P2738R1 [2] in C++26.

The only downsize of using `void*` is then a slightly more complicated return type. However, that return type is really just copying the CV qualifiers, which is not exactly novel.

Therefore, we think using `void*` as the return type of `address` addresses the NB comment in a satisfactory manner.

- It addresses the concerns over missuse and safety: the `void*` pointer can be compared, hashed, etc without issue, but accessing the underlying object requires an explicit cast.
- Accessing the underlying object is still possible, even during constant evaluation.

What about functions?

Pointers to function are objects so this also works during constant evaluation [Compiler Explorer].

Wording

[*Editor's note:* Add the following paragraph at the start of [atomics.general]]

Let *COPYCV*(FROM, TO) be an alias for type TO with the addition of FROM's top-level cv-qualifiers.

[*Editor's note:* That wording is a verbatim copy from [meta], LWG might want to factorize it out] .

[diamond] General [atomics.ref.generic.general]

[*Editor's note:* Modify in the synopsys]

```
using value_type = remove_cv_t<T>;
using address-return-type = COPYCV(T, void)*; //expos
static constexpr size_t required_alignment = implementation-defined;

// ...

constexpr T* address-return-type address() const noexcept;
```

[diamond] Operations [atomics.ref.ops]

```
constexpr T* address-return-type address() const noexcept;
```

Returns: `ptr`.

🔲 Specializations for integral types [atomics.ref.int]

[*Editor's note:* Modify in the synopsys]

```
using value_type = integral-type;
using difference_type = value_type;
using address-return-type = COPYCV(integral-type, void)*; //exposition only
static constexpr size_t required_alignment = implementation-defined;

static constexpr bool is_always_lock_free = implementation-defined;

// ...

constexpr integral-type* address-return-type address() const noexcept;
```

🔲 Specializations for floating-point types [atomics.ref.float]

[*Editor's note:* Modify in the synopsys]

```
using value_type = floating-point-type;
using difference_type = value_type;
using address-return-type = COPYCV(floating-point-type, void)*; //exposition only
static constexpr size_t required_alignment = implementation-defined;


static constexpr bool is_always_lock_free = implementation-defined;

// ...

constexpr floating-point-type* address-return-type address() const noexcept;
```

🔲 Partial specialization for pointers [atomics.ref.pointer]

[*Editor's note:* Modify in the synopsys]

```
using value_type = remove_cv_t<pointer-type>;
using difference_type = ptrdiff_t;
using address-return-type = COPYCV(pointer-type, void)*; //exposition only

static constexpr size_t required_alignment = implementation-defined;

// ...

constexpr pointer-type* address-return-type address() const noexcept;
```

Acknowledgments

We would like to thank Hana Dusíková and Michael Hava for their help and feedback on this paper.

References

[1] Gonzalo Brito Gadeschi, Mark Hoemmen, Carter H. Edwards, and Bryce Adelstein Lelbach. P2835R7: Expose std::atomic_ref's object address. https://wg21.link/p2835r7, 11 2024.

[2] Corentin Jabot and David Ledger. P2738R1: constexpr cast from void*: towards constexpr type-erasure. https://wg21.link/p2738r1, 2 2023.

[N5008] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N5008