Document Number:	P3978R0
Date:	2026-01-29
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	LEWG
Target:	DR against C++26, C++29

constant_wrapper should unwrap on call and subscript

ABSTRACT

This paper proposes unwrapping overloads of `operator()` and `operator[]` to `constant_wrapper`.

CONTENTS

1 CHANGELOG

(placeholder)

2 STRAW POLLS

(placeholder)

2.1 SUGGESTED POLLS

Poll: Adopt P3978R0 for C++26

SF	F	N	A	SA

Poll: Adopt P3978R0 for C++29

SF	F	N	A	SA

Poll: Adopt P3978R0 for C++29 and apply as a DR

SF	F	N	A	SA

3 MOTIVATION

As discussed in [P3948R0], because of language inconsistencies, `std::constant_wrapper` is inconsistently not unwrapping for call and subscript operators whereas all other operators can be found via ADL and the conversion operator implemented in `constant_wrapper`. Looking at `std::reference_wrapper` we see that this same issue has been resolved via an `operator()` overload that unwraps and applies `INVOKE`.

I always had these unwrapping overloads in my implementation of `constant_wrapper` shipping in the vir-simd library[1] (`vir::constexpr_wrapper`). While replacing my implementation with `std::constant_wrapper` I noticed the mismatch.

3.1 DESIGN PRINCIPLE

The following has always been my thinking on what `constant_wrapper` is / needs to be, which informed all my opinions on its API: `constant_wrapper` exists to be able to *use function arguments in place of template arguments*.

- As a template argument passing a `1` the function sees an `int`.
- As a function argument passing a `cw<1>` the function does not see an `int` but a `constant_wrapper<1, int>`; I actually wanted an `int`.

Therefore, `constant_wrapper<X>` should transparently unwrap into `X` whenever possible (similar to `reference_wrapper` unwrapping to the reference it holds), except if it *can* stay in the type space, in which case it unwraps all operands/arguments and wraps the result.

> it's the thing it holds, unless it can stay wrapped

This leads to the following expectations:

```
constexpr int iota[4] = {0, 1, 2, 3};
```

`cw<1> + 1`	→	`2`	the thing it holds
`cw<1> + cw<1>`	→	`cw<2>`	can stay wrapped
`cw<iota>[1]`	→	`1`	the thing it holds
`cw<iota>[cw<1>]`	→	`cw<1>`	can stay wrapped
`cw<fun>(1, 2)`	→	`fun(1, 2)`	the thing it holds
`cw<fun>(cw<1>, cw<2>)`	→	`cw<fun(1, 2)>`	can stay wrapped*
`cw<fun>(cw<1>, 2)`	→	`fun(cw<1>, 2)`	the thing it holds
`cw<unary>()`	→	`cw<unary()>`	can stay wrapped*

* `fun(1, 2)` and `unary()` need to be constant expressions, otherwise unwrap.

3.2 STATUS QUO EXAMPLES

1 https://github.com/mattkretz/vir-simd

```
auto test1()
{
  constexpr int iota[4] = {0, 1, 2, 3};
  auto x = std::cw<iota>;
  return x[1]; // #1 OK
}

auto test2()
{
  auto x = std::cw<std::array<int, 4> {0, 1, 2, 3}>;
  return x[1]; // #2 ill-formed
}
```

The subscript in #1 is fine because `x` is convertible to `int[4]` and because operator lookup is different for built-in types, the built-in subscript operator is found and `int(1)` is returned. The subscript in #2 does not work because, even though `array<int, 4>` is an associated namespace and `x` is convertible to `array<int, 4>`, the `array::operator[]` member function is not found. This is because only hidden friend operators are considered via ADL. Note how ADL makes the following work for `operator+`:

```
struct X { friend int operator+(X, int a) { return a + 1; } };

auto test3()
{
  auto x = std::cw<X{}>;
  return x + 1; // #3 OK
}
```

The expression `x + 1` in line #3 finds `X::operator+` via ADL and thus converts the left operand to `operator+` to `X`, returning `int(2)`.

The situation for `operator()` is equivalent:

```
int fun(int x) { return x + 1; }

auto test1()
{
  auto x = std::cw<fun>;
  return x(1); // #4 OK
}

auto test2()
{
  auto x = std::cw<[](int x) { return x + 1; }>;
  return x(1); // #5 ill-formed
}
```

The call expression in #4 relies on special core wording that looks through the conversion operator to find the function pointer and then call the function. The expression in #5, however, cannot find the `operator()` member of the lambda, because the operator is a member, not a hidden friend.

4 DISCUSSION

The most glaring question on this issue is why would we do this for `operator()` and `operator[]` but for none of the other operators. This seems inconsistent. However, we need to realize that the inconsistency is in the language, forcing the inconsistent definition of operators in the library. This makes the *behavior* of the API, where operators unwrap transparently, more consistent. After all, the conversion operator and the additional type template argument in `constant_wrapper` exist exactly because `constant_wrapper` is supposed to transparently unwrap.

Wouldn't it then be better to fix the language?

For what it's worth, I think it would be a hugely helpful change to make the behavior of all operators as consistent as possible. Currently, every operator has its own set of restrictions and extras. If I could, I'd make *every* operator overloadable as non-member (and thus hidden friend) and implement all operators of standard library types as hidden friends. But the amount of code we would break … The only possibility that is not a breaking change is to add syntax that opts into new behavior, which has a high acceptance barrier.

In terms of consistency we also have `std::reference_wrapper` to consider. The similar naming is not accidental. Consequently, the unwrapping behavior should also be consistent.

5 WORDING

5.1 FEATURE TEST MACRO

In [version.syn] bump the `__cpp_lib_constant_wrapper` version.

5.2 MODIFY [CONST.WRAP.CLASS]

In [const.wrap.class], insert:

[const.wrap.class]

```
    // call and index
    template<constexpr-param T, constexpr-param... Args>
      constexpr auto operator()(this T, Args...) noexcept
        requires requires { constant_wrapper<T::value(Args::value...)>(); }
          { return constant_wrapper<T::value(Args::value...)>{}; }
    template<constexpr-param T, class... Args>
      constexpr see below operator()(this T, Args&&...) noexcept(see below);
    template<constexpr-param T, constexpr-param... Args>
      constexpr auto operator[](this T, Args...) noexcept
        -> constant_wrapper<(T::value[Args::value...])>
          { return {}; }
    template<constexpr-param T, class... Args>
      constexpr see below operator[](this T, Args&&...) noexcept(see below);

    // pseudo-mutators
    template<constexpr-param T>
      constexpr auto operator++(this T) noexcept
        -> constant_wrapper<++Y> { return {}; }
  [...]

constexpr cw-fixed-value(T (&arr)[Extent]) noexcept;
```

4 *Effects*: Initialize elements of *data* with corresponding elements of `arr`.

```
template<constexpr-param T, class... Args>
  constexpr invoke_result_t<decltype(T::value), Args...> operator()(this T, Args&&...)
    noexcept(is_nothrow_invocable_v<decltype(T::value), Args...>);
```

-?- *Constraints*: The type of `T::value` is not a fundamental type and no type in `remove_cvref_t<Args>...` models *constexpr-param*.

-?- *Returns:* *INVOKE*`(T::value, std::forward<Args>(args)...)` ([func.require]).

```
template<constexpr-param T, class... Args>
  constexpr see below operator[](this T, Args&&...) noexcept(see below);
```

-?- *Constraints*: The type of `T::value` is not a fundamental type and no type in `remove_cvref_t<Args>...` models *constexpr-param*.

-?- *Returns:* `T::value[std::forward<Args>(args)...]`.

A Acknowledgments

Thanks to Barry Revzin for helpful feedback on the first draft of this paper.

B Bibliography

[P3948R0] Matthias Kretz. *`constant_wrapper` is the only tool needed for passing constant expressions via function arguments*. ISO/IEC C++ Standards Committee Paper. 2025. URL: `https://wg21.link/p3948r0`.