Document Number:	P3978R3
Date:	2026-03-25
Reply-to:	Matthias Kretz <m.kretz@gsi.de> Tomasz Kamiński <tomaszkam@gmail.com>
Audience:	LWG
Target:	C++26

constant_wrapper should unwrap on call and subscript

ABSTRACT

This paper proposes unwrapping overloads of `operator()` and `operator[]` to `constant_wrapper` while making them all `static`. We also change the call operator to always use *INVOKE*.

CONTENTS

1 Changelog 1
- 1.1 Changes from revision 0 . . . 1
- 1.2 Changes from revision 1 . . . 1

2 Straw Polls 1
- 2.1 Suggested Polls . . . 1

3 Motivation 2
- 3.1 Design Principle . . . 2
- 3.2 Status Quo Examples . . . 3

4 Discussion 4
- 4.1 Exploration of potential ambiguities . . . 4
- 4.2 Example: Interaction with bind_front / bind_back . . . 5
- 4.3 INVOKE should be used (or not used) consistently . . . 7

5 Header for constant_wrapper 8

6 Future Work? 8

7 Wording 9
- 7.1 Instructions to the editor . . . 9
- 7.2 Feature test macro . . . 9
- 7.3 Modify [const.wrap.class] . . . 9

A Acknowledgments 10

B Bibliography 11

1 CHANGELOG

1.1 CHANGES FROM REVISION 0

Previous revision: P3978R0

- Change from overloads to a single catch-all template for each of `operator()` and `operator[]`.
- Make call and subscript `static`.
- Discuss perceived ambiguities.
- Discuss interaction with `bind_front` / `bind_back`.
- Propose a move to `<utility>`.

1.2 CHANGES FROM REVISION 1

Previous revision: P3978R1

- Use *INVOKE* consistently (in both modes or not at all).

2 STRAW POLLS

(placeholder)

2.1 SUGGESTED POLLS

Poll: Do *not* use *INVOKE* for either mode of `operator()`.

SF	F	N	A	SA

Poll: Move `constant_wrapper` and `cw` from `<type_traits>` to `<utility>`

SF	F	N	A	SA

Poll: Adopt P3978R3 for C++26

SF	F	N	A	SA

Poll: Adopt P3978R3 for C++29

SF	F	N	A	SA

Poll: Adopt P3978R3 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*.

- Passing a `1` as a template argument, the function sees an `int`.
- Passing a `cw<1>` as a function argument, 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 re-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.

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

3.2 Status quo examples

```
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 the value of the `constant_wrapper` 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.

4.1 EXPLORATION OF POTENTIAL AMBIGUITIES

Consider the examples discussed in Kozicki [P3792R0], which overload on `constant_wrapper` and the type it is convertible to. We can construct a similar case with the existing `constant_wrapper` operator overloads:

```
struct V {
  int data;

  constexpr V(int x) : data(x) {}

  template <auto N>
  constexpr V(std::constant_wrapper<N, int> x) : data(x + 1) {}

  friend constexpr V operator+(const V& a, const V& b) {
    return a.data + b.data;
  }
};

auto a = cw<V(1)> + 2;             //    V(3)
auto b = cw<V(1)> + cw<2>;         // cw<V(3)>
auto c = cw<V(1)> + cw<V(2)>;      // cw<V(3)>
auto d = cw<V(1)> + cw<cw<2>>;     // cw<V(4)>
auto e = cw<V(1)> + cw<cw<V(2)>>; // cw<V(3)>
```

`V` is structural, convertible from `int` and from `cw<N>` (where `N` is of type `int`).

a. ADL finds `V::operator+` and `constant_wrapper::operator+`, but only the former is viable. The result is `V(3)` ("it's the thing it holds").

b. ADL again finds both operators, but now `constant_wrapper::operator+` is viable and requires no conversions. The `constant_wrapper` operator evaluates `V(1) + 2`, which is `V(3)`. The result is thus `cw<V(3)>` ("it can stay wrapped").

c. See b.

d. ADL still finds both operators, but now `V::operator+` isn't even viable anymore. `constant_wrapper::operator+` evaluates `V(1) + cw<2>`, which calls `V::operator+` and requires a conversion from `cw<2>` to `V`. The conversion constructor adds `1` and `V(1) + cw<2>` thus equals `V(4)`. It is then wrapped into `constant_wrapper` again, making the result `cw<V(4)>` ("it can stay wrapped" and the inner evaluation is "the thing it holds").

e. See d, except that we don't need a conversion from `cw<2>` to `V` and thus the result is `cw<V(3)>`.

We observe that the `constant_wrapper` "it can stay wrapped" operator takes precedence over the `V` "it's the thing it holds" operator. The same behavior is now proposed for call and subscript. Consider:

```
struct Callable
{
  static constexpr int operator()(int) {
    return 1;
  }

  template <auto N>
  static constexpr int operator()(std::constant_wrapper<N, int>) {
    return 2;
  }
};

auto a = Callable{}(0);              // 1
auto b = Callable{}(cw<0>);          // 2
auto c = cw<Callable{}>(0);          // 1
auto d = cw<Callable{}>(cw<0>);      // cw<1>
auto e = cw<Callable{}>(cw<cw<0>>); // cw<2>
```

`a` and `b` are obvious. `c` is new with this paper and also fairly obvious ("it's the thing it holds"). `d` is the case one might consider ambiguous. Do we expect `cw<Callable>` to unwrap and then call the `constant_wrapper` overload? Or do we expect both to unwrap, call the `int` overload and wrap it again? Above, we observed that "it can stay wrapped" takes precedence over "it's the thing it holds". `d` being deduced as `cw<1>` is consistent and follows this one simple rule. `e`, finally uses the `constant_wrapper` call operator to call `Callable(cw<0>)`, which is 2, wraps it again and thus is deduced as `cw<2>`.

If we construct a 2-argument call operator, then all operands need to be `constant_wrapper` objects for "it can stay wrapped" to take precedence. A call like `cw<Callable2arg>(cw<1>, 2)` cannot produce a `constant_wrapper` (because 2 is a function argument and therefore not a constant expression in the operator body) and thus the expression evaluates as `Callable2arg(cw<1>, 2)`.

4.2 Example: interaction with bind_front / bind_back

With this paper `std::bind_front(std::cw<&T::memfun>, obj)` becomes useful. Without it, `std::bind_front(std::cw<&T::memfun>, obj)` itself is well-formed, but calling `operator()` on it isn't.

Consider:

```
struct X {
  int fun(int x);
};

int blackhole = 0;

void eval(auto&& b) { blackhole = b(1); }

void test1(X& t) {
  eval(std::bind_front(&X::fun, t)); // indirect call in eval
}

void test2(X& t) {
  eval(std::bind_front<&X::fun>(t)); // direct call in eval
}

void test3(X& t) {
  eval(std::bind_front(std::cw<&X::fun>, t)); // direct call in eval
}
```

All three are equivalent. However, `test3` compiles to the same code as `test2`[2] while `test1` is less efficient since it uses an indirect call in `eval`.

Finally, let's consider a realistic example where `bind_back<fn>(...)` vs. `bind_back(cw<fn>, ...)` makes an observable difference. We use `std::bind_back` to create a unary transformation that returns a fifth for a given input. The `Scaler` implementation uses a `constant_wrapper` overload to optimize division for statically known divisors.

```
template<typename T>
  struct Scaler {
    static constexpr T operator()(T v, T d) // #1
    { return v / d; }

    // optimization of the above: division by known constant doesn't need
    // slow DIV instruction
    template<auto D>
    static constexpr T operator()(T v, std::constant_wrapper<D, T> d) // #2
    { return v / d; }
  };

template<template<typename ...> class Tuple, typename... Args, typename S>
constexpr auto transform_tuple(Tuple<Args...> t, S&& s) {
  auto [... elems] = std::move(t);
  return Tuple{ s(elems)... }; // CTAD
}

constexpr std::strided_slice slice {15, std::cw<20>, std::cw<5>};
auto sl1 = transform_tuple(slice, std::bind_back<Scaler<int>{}>(std::cw<5>));
auto sl2 = transform_tuple(slice, std::bind_back(std::cw<Scaler<int>{}>, std::cw<5>));
```

`sl1`: `std::strided_slice {3, 4, 1}`

The calls to `s(elems)...` in `transform_tuple` all invoke overload #2. For `s(elems...[0])`

2 tested on GCC trunk with libstdc++

overload #2 is a perfect match. `s(elems...[1])` calls `Scaler<int>(cw<20>, cw<5>)` for which overload #2 is a better match (only one conversion). Same for `s(elems...[2])`.

sl2: `std::strided_slice {3, std::cw<4>, std::cw<1>}`
`s(elems...[0])` in `transform_tuple` calls `cw<Scaler<int>{}>(15, cw<5>)` which invokes overload #2. `s(elems...[1])` calls `cw<Scaler<int>{}>(cw<20>, cw<5>)`. Here all operands to the expression are `constant_wrappers` and `operator() #1` is declared `constexpr`, which allows the "it can stay wrapped" behavior. The result of `s(elems...[1])` therefore is `cw<4>`. Likewise, `s(elems...[2])` evaluates to `cw<1>`.

In this example, a change from `bind_back<Callable>(cw<5>)` to `bind_back(cw<Callable>, cw<5>)` leads to a *different* result. This is not necessarily wrong or bad. It can certainly be argued that the `sl2` result is better. It can also be argued that the `sl2` result is surprising. However, it still follows the simple rule of "it's the thing it holds, unless it can stay wrapped".

4.3 INVOKE should be used (or not used) consistently

[P3978R1] proposed to use *INVOKE* for the unwrapping call operator and otherwise apply `operator()` directly on `value`. This is inconsistent and not proposed anymore. Instead this paper proposes to use *INVOKE* in both cases. Relative to [P3978R1], this paper's wording has the following change:

---

5 Let *call-expr* be `constant_wrapper<~~value(~~INVOKE(value, remove_reference_t<Args>::value...)>{}` if all types in `remove_reference_t<Args>...` satisfy *constexpr-param* and `constant_wrapper<~~value(~~INVOKE(value, remove_reference_t<Args>::value...)>` is a valid type, otherwise let *call-expr* be *INVOKE*`(value, std::forward<Args>(args)...)`.

Example:

```
struct T {
  int v;
  constexpr int add(int x) { return v + x; }
};

auto a = cw<&T::add>(cw<T{1}>, cw<2>) // -> cw<3>
```

This seems to be the right choice for consistency. However, we did not use *INVOKE* initially for `constant_wrapper` because the intent was to model the exact interface of the underlying type. Using *INVOKE*, we are deviating from that choice, giving the call operator more functionality than the type it wraps. This is also true for the "it's the thing it holds" case. The other choice for resolving the inconsistency is the removal of *INVOKE* from the `constant_wrapper` wording altogether:

---

5 Let *call-expr* be `constant_wrapper<~~INVOKE(value,~~ value(remove_reference_t<Args>::value...)>{}` if all types in `remove_reference_t<Args>...` satisfy *constexpr-param* and `constant_wrapper<~~INVOKE(value,~~ value(remove_reference_t<Args>::value...)>` is a valid type, otherwise let *call-expr* be ~~*INVOKE*~~`(value, value(std::forward<Args>(args)...)`.

---

5 HEADER FOR CONSTANT_WRAPPER

We should reconsider the placement of `constant_wrapper` in `<type_traits>`.

1. With this change we use *INVOKE*, which typically isn't available already in `<type_traits>`.
2. `constant_wrapper` is not a trait. `integral_constant` is not a trait either, but it is the type of many traits. `constant_wrapper` is a utility (`<utility>`).
3. `<functional>`? Too heavy, in my opinion.

For reference in libstdc++ we have the following dependencies:

```
> grep include type_traits
# include <bits/c++0x_warning.h>
#include <bits/c++config.h>

> grep include utility
#include <bits/c++config.h>
#include <bits/stl_relops.h>
#include <bits/stl_pair.h>
#include <type_traits>
#include <bits/move.h>
#include <initializer_list>
#include <ext/numeric_traits.h>

> grep include functional
#include <bits/c++config.h>
#include <bits/stl_function.h>
#include <new>
#include <tuple>
#include <type_traits>
#include <bits/functional_hash.h>
#include <bits/invoke.h>
#include <bits/refwrap.h>       // std::reference_wrapper and _Mem_fn_traits
#include <bits/std_function.h>  // std::function
# include <unordered_map>
# include <vector>
# include <array>
# include <utility>
# include <bits/stl_algo.h>
# include <bits/ranges_cmp.h>
# include <compare>
```

6 FUTURE WORK?

With this paper, `constant_wrapper` almost achieves parity with what was proposed for `fn_t` / `function_wrapper` by Schultke et al. [P3774R0] and Müller [P3843R2]. The one feature still missing is the conversion to a different function pointer type via implicit creation of a thunk. This could be added as a member function (`to_func()`), which returns a different type that implements the conversion operator as shown in Section 4.2 of [P3774R0].

7 WORDING

7.1 INSTRUCTIONS TO THE EDITOR

Move `constant_wrapper`, `cw`, *`cw-fixed-value`*, *`constexpr-param`*, *`cw-operators`* before `integer_sequence`. Move the declarations of those from [meta.type.synop] to [utility.syn].

7.2 FEATURE TEST MACRO

In [version.syn] bump the `__cpp_lib_constant_wrapper` version. Change the comment after `__cpp_lib_constant_wrapper` from `// freestanding, also in <type_traits>` to `// freestanding, also in <utility>`.

7.3 MODIFY [CONST.WRAP.CLASS]

In [const.wrap.class], change:

[const.wrap.class]

```
    template<constexpr-param L, constexpr-param R>
      friend constexpr auto operator->*(L, R) noexcept -> constant_wrapper<L::value->*(R::value)>
        { return {}; }

    // 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, constexpr-param... Args>
      constexpr auto operator[](this T, Args...) noexcept
        -> constant_wrapper<(T::value[Args::value...])>
          { return {}; }

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

  template<cw-fixed-value X, class>
  struct constant_wrapper : cw-operators {
    static constexpr const auto & value = X.data;
    using type = constant_wrapper;
    using value_type = decltype(X)::type;

    template<constexpr-param R>
      constexpr auto operator=(R) const noexcept
        -> constant_wrapper<X = R::value> { return {}; }

    constexpr operator decltype(auto)() const noexcept { return value; }

    template<class... Args>
      static constexpr decltype(auto) operator()(Args&&... args) noexcept(see below);
    template<class... Args>
```

```
      static constexpr decltype(auto) operator[](Args&&... args) noexcept(see below);
  };
}
 [...]

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

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

```
template<class... Args>
  static constexpr decltype(auto) operator()(Args&&... args) noexcept(see below);
```

-?- Let *call-expr* be `constant_wrapper<`*INVOKE*`(value, remove_cvref_t<Args>::value...)>{}` if all types in `remove_cvref_t<Args>...` satisfy *constexpr-param* and `constant_wrapper<`*INVOKE*`(value, remove_cvref_t<Args>::value...)>` is a valid type, otherwise let *call-expr* be *INVOKE*`(value, std::forward<Args>(args)...)`.

-?- *Constraints*: *call-expr* is a valid expression.

-?- *Effects*: Equivalent to: `return` *call-expr*`;`

-?- *Remarks:* The exception specification is equivalent to `noexcept(`*call-expr*`)`.

```
template<class... Args>
  static constexpr decltype(auto) operator[](Args&&... args) noexcept(see below);
```

-?- Let *subscr-expr* be `constant_wrapper<value[remove_cvref_t<Args>::value...]>{}` if all types in `remove_cvref_t<Args>...` satisfy *constexpr-param* and `constant_wrapper<value[remove_cvref_t<Args>::value...]>` is a valid type, otherwise let *subscr-expr* be `value[std::forward<Args>(args)...]`.

-?- *Constraints*: *subscr-expr* is a valid expression.

-?- *Effects*: Equivalent to: `return` *subscr-expr*`;`

-?- *Remarks:* The exception specification is equivalent to `noexcept(`*subscr-expr*`)`.

A Acknowledgments

Thanks to Barry Revzin for helpful feedback.

B BIBLIOGRAPHY

[P3792R0] Bronek Kozicki. *Why constant_wrapper is not a usable replacement for nontype*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3792r0.

[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.

[P3978R1] Matthias Kretz and Tomasz Kamiński. *`constant_wrapper` should unwrap on call and subscript*. ISO/IEC C++ Standards Committee Paper. 2026. URL: https://wg21.link/p3978r1.

[P3843R2] Jonathan Müller. *std::function_wrapper*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3843r2.

[P3774R0] Jan Schultke, Bronek Kozicki, and Tomasz Kamiński. *Rename `std::nontype` and make it broadly useful*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3774r0.