Document Number: P3948R0
Date: 2025-12-15
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG
Target: C++26

# CONSTANT_WRAPPER IS THE ONLY TOOL NEEDED FOR PASSING CONSTANT EXPRESSIONS VIA FUNCTION ARGUMENTS

## ABSTRACT

We added `std::constant_wrapper<value>` to C++26 to enable “passing constant expressions as function arguments”. This is especially important for constructors where we cannot give explicit template arguments. However, `constant_wrapper` also overloads all operators in such a way that values stay in the “type space”. This was initially motivated by user-defined literals, which require a unary minus operator. For consistency, LEWG voted to overload all operators. The existence of `constant_wrapper::operator()` lead to fear, uncertainty, and doubt[1], whether wrapping a callable with `constant_wrapper` leads to inconsistent semantics. This paper shows that there is no real problem, and provides a simple solution to resolve the contrived problems that can be constructed.

## CONTENTS

1 not intentional FUD, though

## 1 CHANGELOG

(placeholder)

## 2 STRAW POLLS

(placeholder)

## 3 MOTIVATION

The primary design intent of `std::constant_wrapper<x>` is the ability to pass `x` as a template argument via a function (esp. constructor) argument. The presence of a second type `std::constant_arg` that has the same design intent but with extra semantics and a narrow focus on `std::function_ref` is just inconsistent and hard to teach. After reviewing the reasons why LEWG decided to not use `std::constant_wrapper` I got only more convinced that we should use `std::constant_wrapper` for `std::function_ref`.

The status quo of C++26 can be understood with this example:[2]

```
#include <functional>
#include <type_traits>

int nullary();
int unary(int);
int binary(int, int);

int f0(std::function_ref<int()>);
int f1(std::function_ref<int(int)>);
int f2(std::function_ref<int(int, int)>);
int g0(std::move_only_function<int()>);
int g1(std::move_only_function<int(int)>);
int g2(std::move_only_function<int(int, int)>);

void test()
{
  f0(nullary);          // 1
  f0(std::cw<nullary>); // 2
  f1(std::cw<unary>);   // 3
  f2(std::cw<binary>);  // 3
  f0(std::cw<[]() { return 1; }>); // 4
  g0(nullary);          // 5
  g0(std::cw<nullary>); // 5
  g1(std::cw<unary>);   // 5
  g2(std::cw<binary>);  // 5
  g0(std::cw<[]() { return 1; }>); // 5

  // these can't work because of a language limitation:
  //f1(std::cw<[](int) { return 1; }>);
  //f2(std::cw<[](int, int) { return 1; }>);
  //g1(std::cw<[](int) { return 1; }>);
  //g2(std::cw<[](int, int) { return 1; }>);
```

2 https://compiler-explorer.com/z/8z7M6q7PM

```
}
```

The commented calls are ill-formed because of a language inconsistency: A call expression on an object that is implicitly convertible to a function pointer is valid (calling through the function pointer). However, if such an object is implicitly convertible to a type implementing a call operator, that call is not considered. This is similar to operator lookup for fundamental types vs. user-defined types: the former requires only a conversion operator, the latter is required to be in an associated namespace (which `constant_wrapper` does) and the operator needs to be defined as a hidden friend (which the language doesn't allow for call operators).

This is unfortunate, but not really important for this discussion. This discussion wants to focus on what `cw<fun>` already means *right now* without adopting any other paper.

1. `f0(nullary)` is a simple type-erased call through a function pointer into `nullary`.

2. `f0(cw<nullary>)` is translated into a *direct* jump to `nullary`; i.e., this doesn't need an indirect call.[3] This is possible because the function pointer is passed as a template argument into the place where the `function_ref` implementation constructs the call. (Note that `constant_wrapper::operator()` is not invoked, because this operator is not viable in a constant expression.) Also, note that `f0(constant_arg<nullary>)` is equivalent. The only difference is that with `cw` a non-null value is stored for the function pointer, even though that value is never needed.

3. `f1(cw<unary>)` and `f2(cw<binary>)` are the same as `f0(cw<nullary>)` with one or two function arguments. Both also result in a direct jump when invoking the `function_ref`.

4. `f0(cw<[]() { return 1; })` shows that a nullary lambda also works (using `-fno-inline`: `https://compiler-explorer.com/z/vP5YMea8v`). In this case the expression `cw<[]() { return 1; }>()` returns `cw<1>` because it is a valid constant expression invoking `constant_wrapper::operator()`. Contrary to `nullary`, the lambda is `constexpr` and can thus be used to initialize the `cw` template argument. `cw<1>` is subsequently converted to `int(1)` when the `function_ref` is invoked. This is functionally equivalent to `f0(constant_arg<[]() { return 1; }>)`, which also returns constant `int(1)`, except that the implementation doesn't need to go through *INVOKE*.

   Important here is the observation that if `constant_wrapper::operator()` is well-formed, then its *semantics are equivalent* to unwrapping the `constant_wrapper` and then calling the function. The difference is equivalent to whether `function_ref::operator()` is implemented as

   ```
       int operator()() { constexpr int tmp = wrapped(); return tmp; }
   ```

   or

   ```
       int operator()() { return wrapped(); }
   ```

5. All of these conversions from `cw<…>` to `function_ref` also work for `move_only_function` (and also `copyable_function` and `function`). Consider the difference of `g0(nullary)` vs. `g0(cw<nullary>)` (`https://compiler-explorer.com/z/8nYhYP7G5`):

3 This assumes enabled optimizations; with `-O0` both `cw<nullary>` and `constant_arg<nullary>` are compiled into an indirect call.

```
"int std::__polyfunc::_Base_invoker<false, int>::_S_call_storage< …"
    jmp     [QWORD PTR [rdi]]
```

```
"int std::__polyfunc::_Base_invoker<false, int>::_S_call_storage< …"
    mov     eax, 2
    ret
```

Using `constant_wrapper`, the indirect call/jump is replaced by an inlined call to `nullary()`. `constant_wrapper` thus already enables an optimization similar to the `constant_arg` overload in `function_ref`. The same code-gen difference exists for `copyable_function` and `function`. (If `nullary()` cannot be inlined, then the `cw<nullary>` case turns into a direct call/jump to `nullary()`.)

In short: **`cw<func>` already works like `constant_arg` for all function wrappers.** Special casing `constant_arg` for the non-bind constructor is thus only a minor optimization for non-optimized builds.

There was a concern that the function pointer to `nullary` would be dangling when `cw<nullary>` is used. I do not know where this concern stems from. To the contrary, `constant_wrapper` is carefully designed in such a way that its `value` member is a "never dangling" object, since it is a const-ref to a template parameter object. Thus, indirect calls with `-O0` are also valid.

At this point we should be able to agree that using `std::cw` to wrap functions works correctly and can have a run-time performance benefit (at a compile-time cost). Except … what about the failure that was presented in [P3792R0]?

## 3.1 Constructing an inconsistency

Consider the following code[4] as introduced by [P3792R0], but with a minor adjustment to the assertion. The assertion is modified because [P3792R0] argues about broken consistency between the different function wrappers and the assertion used in this paper actually shows the issue:

```
#include <functional>
#include <type_traits>
#include <cassert>

static constexpr struct foo_t final
{
  constexpr auto operator()(auto &&...args) const -> int // #1
      requires(std::integral<std::remove_cvref_t<decltype(args)>> && ...)
  {
    return (0 + ... + args);
  }

  constexpr auto operator()(auto &&...args) const -> int // #2
    requires(std::integral<decltype(std::remove_cvref_t<decltype(args)>::value)> && ...)
  {
    return sizeof...(args);
  }

} foo = {};
```

4 https://compiler-explorer.com/z/59EMracKT

```
static constexpr struct baz_t final
{
  static constexpr int value = 42;
} baz = {};

auto main() -> int
{
  std::function_ref<int(baz_t)> f0(std::cw<foo>);
  std::move_only_function<int(baz_t)> f1(std::cw<foo>);
  assert(f0(baz) == f1(baz));
}
```

This is currently (C++26 status quo) consistent in that both `f0` and `f1` return `42`. Since it isn't obvious how the above code works, note that `#2` is an unnecessary overload at this point. What `cw<callable>` requires in order to work in this context is a `callable` that is either a function pointer/reference or an object with call operator whose arguments can be unwrapped (`value` member), passed to the call operator, and the result can be wrapped in `constant_wrapper` again. (`constant_wrapper<f>::operator()(Args...)` returns `constant_wrapper<f(Args::value...)>`) So it must be a constant expression; and currently the language requires all types to be structural. Therefore, `std::cw<foo>(baz)` is `std::cw<0 + 42>`. In other words, the code `function_ref<int(baz_t)>(cw<foo>)` is what I would call a useless obfuscation for saying `42`.

This is more apparent in a simpler example such as:

```
int f(std::function_ref<int(std::constant_wrapper<3.f>)>; // #1
int func(std::constant_wrapper<3.f>); // could also use 'auto' as function parameter type
                                      // if we were still able to name it
int test1() { return f(std::cw<func>); }
```

This is pointless. The point of passing a callable with function parameters is to call it with *unknown values* for those parameters. But here we had to define the value that it gets called with as *part of the type*. In line `#1` we could have just used a nullary `int()` signature.

Nevertheless, let us consider the possibility of a type with a data member that also derives from a trait with a `static constexpr value` member (this is the closest I can imagine to such an issue turning up in real code). Now, if we change `function_ref` to unwrap `constant_wrapper`, thereby replacing `nontype`/`constant_arg`, then the assertion in the above example fails. That's because if `constant_wrapper` is unconditionally unwrapped by `function_ref` it now calls `foo(baz)` rather than `cw<foo>(baz)`. And since the overload set of `foo_t` is constructed such that this works and returns something different (`#2`), it returns `1`. However, since `move_only_function` does not unwrap `constant_wrapper` the assertion comes down to `1 == 42` and fails.

## 4 RESOLVING THE INCONSISTENCY

The actual problem in the [P3792R0] example is an ambiguity/dual semantics: Did the user mean to call `cw<foo>(baz)` or `foo(baz)`. It is fairly simply to detect this potential mismatch in `function_ref` and reject the program. Rejecting is consistent with overload resolution where ambiguities are rejected.

I implemented `function_ref` unwrapping `constant_wrapper` with such a detection in a libstdc++ fork.[5] The relevant code for rejecting the ambiguous case is straightforward:

```
template <auto __fn, typename... _Args>
  concept __constant_wrapped_invocable
    = sizeof...(_Args) >= 1 && requires(_Args... __args) {
      { cw<__fn>(__args...) } -> _ConstantWrapped;
    };

template<_ConstantWrapped __fn>
  requires __is_invocable_using<const typename __fn::value_type&>
  constexpr
  function_ref(__fn) noexcept
  {
    static_assert(!__constant_wrapped_invocable<__fn::value, _ArgTypes...>,
                  "The given cw<fn> object has ambiguous call semantics.");
    // …[]
```

The corresponding wording change for this constructor overload looks like this:

---

`template<~~auto f~~class F0> constexpr function_ref(~~nontype_t<f>~~F0 f0) noexcept;`

8 Let `f` be `f0.value`. Let `F` be `decltype(f)`.

9 *Constraints*:

- *`is-invocable-using`*`<F>` is `true`, and
- `F0` is a specialization of `constant_wrapper`.

10 *Mandates*:

- If `is_pointer_v<F> || is_member_pointer_v<F>` is `true`, then `f != nullptr` is `true`.
- `sizeof...(ArgTypes)` is 0 or the expression `f0(declval<ArgTypes>()...)` is either not well-formed or its type is not a specialization of `constant_wrapper`.

11 *Effects*: Initializes *`bound-entity`* with a pointer to an unspecified object or null pointer value, and *`thunk-ptr`* with the address of a function *`thunk`* such that *`thunk`*`(`*`bound-entity`*`, `*`call-args`*`...)` is expression-equivalent ([defns.expression.equivalent]) to `invoke_r<R>(f, `*`call-args`*`...)`.

---

With this change one cannot construct inconsistent *behavior* anymore. If it would be inconsistent it rather does not compile. This resolves the concern raised in §3.5 of [P3792R0]:

"

- if nontype was replaced by `constant_wrapper` as a construction parameter to `function_ref` and
- a user performed a (seemingly) simple refactoring by swapping a standard function wrapper with another standard function wrapper (e.g. to benefit from the low cost of `function_ref` or to use the data storage in other wrappers) where the constructor happens to rely on `constant_wrapper`

... any of the following might happen:

5 https://forge.sourceware.org/mkretz/gcc/commit/3277bbbc4e4aaad60294a6cda491ff5cf41c62d8

- the program will continue to work as designed
- the program will fail to compile
- ~~the program will continue to compile and "work", but with subtly changed behaviour~~

"

## 5 SUMMARY

Status quo inconsistencies:

- `constant_arg<fun>` works with `function_ref` but not with any other function wrapper.
- `cw<fun>` works with all function wrappers to an equivalent effect of `constant_arg<fun>` in `function_ref`, duplicating the `function_ref` constructor.
- `constant_wrapper` and `constant_arg` are tools (a.k.a. workarounds) for passing a "constexpr value" via a function/constructor argument. Why are there two? How do their names tell me when to use one or the other?

## 6 PROPOSAL

- Reword `function_ref` from `nontype`/`constant_arg` to `constant_wrapper` (disallowing ambiguous `constant_wrapper` use).
- Drop `nontype`/`constant_arg`.
- Optional: Mandate no `constant_wrapper` arguments to the other function wrappers for concern of future breaking changes. (not recommended: so far I am convinced potential changes can be non-breaking)

## 7 FURTHER/RELATED WORK

The current inconsistencies for "transparent" wrappers due to the language rules are not helpful. However, changes in this area can easily be breaking changes for existing code. Still, it should be possible to opt in to defining a truly transparent wrapper type (where operator lookup of the wrapped type is fully transparent). This needs more exploratory work. An initial patch to GCC to call `operator()` for wrappers like `constant_wrapper` already feels more consistent (to me).

## 8 WORDING

---

```
template<class F> constexpr function_ref(F&& f) noexcept;
```

5 Let `T` be `remove_reference_t<F>`.

6 *Constraints*:

- `remove_cvref_t<F>` is not the same type as `function_ref`,

- `is_member_pointer_v<T>` is `false`~~, and~~
- *`is-invocable-using`*`<`*`cv`*` T&>` is `true`, and
- `remove_cvref_t<F>` is not a specialization of `constant_wrapper`.

7 *Effects*: Initializes *`bound-entity`* with `addressof(f)`, and *`thunk-ptr`* with the address of a function *`thunk`* such that *`thunk`*`(`*`bound-entity`*`, `*`call-args`*`...)` is expression-equivalent ([defns.expression.equivalent]) to `invoke_r<R>(static_cast<`*`cv`*` T&>(f), `*`call-args`*`...)`.

```
template<~~auto f~~class F0> constexpr function_ref(~~nontype_t<f>~~F0 f0) noexcept;
```

8 Let `f` be `f0.value` and ~~Let~~ `F` be `decltype(f)`.

9 *Constraints*:

- *`is-invocable-using`*`<F>` is `true`, and
- `F0` is a specialization of `constant_wrapper`.

10 *Mandates*:

- If `is_pointer_v<F> || is_member_pointer_v<F>` is `true`, then `f != nullptr` is `true`.
- `sizeof...(ArgTypes)` is 0 or the expression `f0(declval<ArgTypes>()...)` is either not well-formed or its type is not a specialization of `constant_wrapper`.

11 *Effects*: Initializes *`bound-entity`* with a pointer to an unspecified object or null pointer value, and *`thunk-ptr`* with the address of a function *`thunk`* such that *`thunk`*`(`*`bound-entity`*`, `*`call-args`*`...)` is expression-equivalent ([defns.expression.equivalent]) to `invoke_r<R>(f, `*`call-args`*`...)`.

```
template<~~auto f~~class F0, class U>
  constexpr function_ref(~~nontype_t<f>~~F0 f0, U&& obj) noexcept;
```

12 Let `T` be `remove_reference_t<U>`, `f` be `f0.value`, and `F` be `decltype(f)`.

13 *Constraints*:

- `is_rvalue_reference_v<U&&>` is `false`~~, and~~
- *`is-invocable-using`*`<F, `*`cv`*` T&>` is `true`, and
- `F0` is a specialization of `constant_wrapper`.

14 *Mandates*: If `is_pointer_v<F> || is_member_pointer_v<F>` is `true`, then `f != nullptr` is `true`.

15 *Effects*: Initializes *`bound-entity`* with `addressof(obj)`, and *`thunk-ptr`* with the address of a function *`thunk`* such that *`thunk`*`(`*`bound-entity`*`, `*`call-args`*`...)` is expression-equivalent ([defns.expression.equivalent]) to `invoke_r<R>(f, static_cast<`*`cv`*` T&>(obj), `*`call-args`*`...)`.

```
template<~~auto f~~class F0, class T>
  constexpr function_ref(~~nontype_t<f>~~F0 f0, cv T* obj) noexcept;
```

16 Let `f` be `f0.value` and ~~Let~~ `F` be `decltype(f)`.

17 *Constraints*:

- *`is-invocable-using`*`<F, `*`cv`*` T*>` is `true`, and
- `F0` is a specialization of `constant_wrapper`.

18 *Mandates*: If `is_pointer_v<F> || is_member_pointer_v<F>` is `true`, then `f != nullptr` is `true`.

19 *Preconditions*: If `is_member_pointer_v<F>` is `true`, `obj` is not a null pointer.

20 *Effects*: Initializes *`bound-entity`* with `obj`, and *`thunk-ptr`* with the address of a function *`thunk`* such that *`thunk`*`(`*`bound-entity`*`, `*`call-args`*`...)` is expression-equivalent ([defns.expression.equivalent]) to `invoke_r<R>(f, obj, `*`call-args`*`...)`.

```
template<class T> function_ref& operator=(T) = delete;
```

21 *Constraints*:

- `T` is not the same type as `function_ref`,
- `is_pointer_v<T>` is `false`, and
- `T` is not a specialization of ~~`nontype_t`~~`constant_wrapper`.

---

# A 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`.