Document Number:	P2772R0
Date:	2023-01-17
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	LEWG-I, LEWG
Target:	C++26

`std::integral_constant` LITERALS DO NOT SUFFICE — `constexpr_t`?

ABSTRACT

Laine [P2725R0] proposes user-defined literals for simpler use of `std::integral_constant`, simplifying basically a notion of passing `constexpr` function arguments. I fully support the idea, but I believe it does not cover the complete problem & design space. In this paper I show the use cases and solutions that I believe need to be considered at the same time.

CONTENTS

1 INTRODUCTION

I am convinced we need simpler and shorter syntax for passing constant expressions to functions. Especially if the function cannot easily resort to an NTTP (operator overload or member function that often would require the `template` keyword when called).

My solution idea started from Listing 1, which uses the variable template `Const` for constructing

```
template <auto N>
inline constexpr std::integral_constant<decltype(N), N> Const = {};

template <typename T>
struct my_complex
{
  T re, im;
};

template <typename T>
struct X
{
  void f(auto c) {
    // c can be used in constant expressions here
  }
};

inline constexpr short foo = 2;

template <typename T>
void g(X<T> x) {
  x.f(Const<1>);
  x.f(Const<2uz>);
  x.f(Const<3.0>);
  x.f(Const<4.f>);
  x.f(Const<foo>); // P2725R0 doesn't solve this
  x.f(Const<my_complex(1.f, 1.f)>); // nor this
}
```

Listing 1: `integral_constant` from variable template

objects of type `integral_constant`. When passed as deduced function parameter, the value can be used in constant expressions in the function body. In Listing 1 the alternative is an NTTP to the function `f`, making all calls in `g` look like `x.template f<1>()`.

The first four calls to `f` in Listing 1 are possible with P2725R0, but the last two are not. P2725R0 can only turn integer literals into `integral_constant`s. The problem space is larger than what P2725R0 solves. Nevertheless, integer literals are a common case and therefore the solution of P2725R0 seems what we want, just incomplete. I think a viable outcome could be to add both

'1ic' and 'std::cnst<1>' at the same time.[1] I believe there is no good rationale for adding *only* `integral_constant` literals. Simple tasks such as, how do I write an `integral_constant` for `INT_MAX`, `std::numeric_limits<int>::max()`, or any other constexpr variable? Should `std::integral_constant<decltype(foo), foo>{}` really be our only answer?

2 WAIT, WHAT? `INTEGRAL_CONSTANT<DOUBLE>`?

Oh, not to forget. I instantiated `integral_constant<double>` (and `float` and `my_complex`) in Listing 1. `integral_constant` is misnamed nowadays. Should it be constrained to integers (for no good reason other than the name)? Or should we consider a new type so that our type names can still be used to carry intent?

A type for passing any possible NTTP could e.g. be named `constexpr_t`:

```
template <auto Value>
struct constexpr_t {
  using value_type = decltype(Value);
  using type = constexpr_t;
  static inline constexpr value_type value = Value;
  constexpr operator value_type() const noexcept { return Value; }
  static constexpr value_type operator()() noexcept { return Value; }
};
```

3 A COMPILE-TIME NUMERIC TYPE

Laine [P2725R0] proposes the addition of unary minus to `integral_constant`. That's a breaking change, as shown by Listing 2.

```
void f(std::same_as<int> auto);

void g(auto x) {
  f(-x); //valid now, ill-formed with P2725R0:
}

void h() {
  g(std::integral_constant<int, 1>());
}
```

Listing 2: Adding unary minus to `integral_constant` is a breaking change

In addition, the return type of unary minus is controversial. `-short(1)` is of type `int`. Whether you dislike integral promotions it or not, that would be inconsistent with `integral_constant::operator-()` returning `integral_constant<short, ... >`.

1 I'd like to write `std::const<1>`, but arrgh. 'std::constant<1>' is a bit too long for my taste.

While the proposed return type seems to be an improvement, what about a user-defined structural type that returns a different type on unary minus? `bounded::integer` [1] is an example of such a type (though not structural). Example:

```
bounded::integer<1, 10> a;
auto b = -a; // b is bounded::integer< -10, -1>
```

For such a case we need `integral_constant::operator-` to return `decltype(-std::declval<T>())`.

Finally, adding only unary minus is inconsistent. We should then also add unary plus and unary tilde (bit flip).

And why stop with unary operators? Binary operators are also missing.

If we want an `integral_constant` type that implements unary minus, I believe we need to have a new type. E.g. `std::numeric_constant<auto value>` that requires the NTTP to have the properties of a numeric type. Such a type would then overload all operators accordingly, similar to `boost::hana::integral_constant`. See `https://godbolt.org/z/vdzzdKdKz`.

4 CONTEXT & UNBAKED EXPLORATIONS

My original angle was the exploration of possible APIs for simple integration of `std::simd` into the ranges and container world.

1. `std::simd::size` shouldn't be a function, but an `integral_constant`. (You can still call it like a function.) `std::array::size` should also be changed to be an `integral_constant` (same for spans of static extent).

   Changing `array::size` should be a non-breaking change. The user code that could get broken is not allowed AFAIU. ("Moreover, the behavior of a C++ program is unspecified (possibly ill-formed) [...] if it attempts to form a pointer-to-member designating [...] a standard library non-static member function [...]." `https://eel.is/c++draft/constraints#namespace.std-6`)

2. I've been playing with (needs my GCC branch for non-member `operator[]`):

   ```
   constexpr std::span<...>
   operator[](std::ranges::contiguous_range auto&&,
              index_like auto first, index_like auto size)
   ```

   I.e. similar to `submdrange`'s `strided_index_range`, I was looking at index ranges. If you pass an `integral_constant` size, you get a `span` of static extent which can CTAD into a `std::simd`. Result:

   ```
   std::vector<float> x = ...;
   std::simd v = x[0, 8ic];
   std::simd w = x[0, std::simd<float>::size];
   ```

Literals as proposed by Laine [P2725R0] make this code much simpler to read and write!

For the second point, my prototype code is at:
`https://github.com/mattkretz/index-range-subscripting/blob/main/subscript.h#L71`

Given:

```
std::vector<float> data;
const auto& cdata = data;
```

I can write:

1. `data[1u, Const<4>]` returning a `span<float, 4>`
2. `cdata[1u, Const<4>]` returning a `span<const float, 4>`
3. `data[1, Const<4>]` returning a `span<float, dynamic_extent>`
4. `data[1u, Const<-4>]` returning a `span<float, dynamic_extent>`
5. `data[-1, Const<-4>]` returning a `span<float, dynamic_extent>`
6. `data[Const<-1>, Const<-4>]` ERROR: index range results in negative size
7. `data[Const<-4>, Const<-1>]` returning a `span<float, 4>`
8. `data[Const<-1>, Const<4>]` ERROR: index range exceeds bounds
9. `(data | transform( … ))[1, 5]` returning a `subrange< … >`

And, can I have more crazy...? After I can write

```
std::simd v = data[1u, std::simd<float>::size];
```

I want to write

```
data[1u, v.size] = v;
```

This requires a new `span::operator=`. Or a non-member `operator=` so that I can implement it as a hidden friend in `simd`?

The more I think of it, the more I like the direction. Originally, I wanted to only allow non-member `operator[]` and non-member `operator?:` overloads. But by now I'm ready to propose that we should simply make all operators the same, i.e. allow non-member overloads for `[]`, `()`, `=`, and `->` in addition to allowing member and non-member operator?: (even if I see little use for member `operator?:` — but consistency wins over my imagination).

A BIBLIOGRAPHY

[P2725R0] Zach Laine. *P2725R0: `std::integral_constant` Literals*. ISO/IEC C++ Standards Committee Paper. 2022. URL: `https://wg21.link/p2725r0`.

[1] David Stone. *davidstone / bounded_integer — Bitbucket*. URL: `https://bitbucket.org/davidstone/bounded_integer` (visited on 02/26/2018).