# A plan for better template meta programming facilities in C++26

Document #: P2632R0
Date: 2022-10-16
Programming Language C++
Audience: EWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>
Pablo Halpern <phalpern@halpernwightsoftware.com>
John Lakos <jlakos@bloomberg.net>
Alisdair Meredith <ameredith1@bloomberg.net>
Joshua Berne <jberne4@bloomberg.net>
Gašper Ažman <gasper.azman@gmail.com>

*Incremental changes can substantially improve expressiveness - Sean Baxter*

## Abstract

Features such as concepts, `if constexpr`, and template parameters packs have made Template Meta Programming more powerful and accessible than it was 15 years ago, or even 5 years ago.

However, some programs are still very difficult, if not impossible, to express. Using template meta programming often requires a high level of expertise. Shortcomings of the core language have been addressed through library features with high compile-time costs and poor user experience.

These problems can be solved by extending the core language or lifting restrictions therein. The overarching goal is to present a set of features to make generic programming and template metaprogramming easier, faster to compile, and with better compile time diagnostics.

Individual features described in this document are or will be the object of separate proposals from different authors.

## Revisions

### R0

Initial revision

## Priorities and status

The following table attempts to classify the features discussed in this paper by some order of priority. This is based on both the usefulness of individual features and their maturity/risk.

It also serves as a table of content for features presented later in this paper.

Tier 1 features are extremely well understood, can be standardized independently of one another (as long as we maintain consistency), They have a great cost/benefits ratio. Pack alias in particular can be used by sender/receiver. Other features in tier 1 can simplify drastically, and improve the compile times of standard types.

Tier 2 features have more complexity or more risks associated with them, or slightly less usefulness. Both Tier 1 and Tier 2 features should target C++26.

Tier 3 features may be useful but carry more risk and need more research. They could be considered for a future C++ version.

Tier 4 features either seem redundant with other features (existing or in that list) or have a unreasonable level of complexity/low return on investment.

| Feature | Paper | In Circle? | Other |
|---|---|---|---|
| **Tier 1** | | | |
| Pack Indexing | P1858R2 [11] [P2662R0] | ✓ | |
| Member Pack | P1858R2 [11] | ✓ | |
| Pack Aliases | P1858R2 [11] | ✓ | |
| Unpacking structured types | P1858R2 [11] | ✓ | |
| `constexpr` structured bindings | P1481R0 [8] | ✓ | |
| `static_assert(false)` | P2593R0 [16] | ✓ | Implemented in GCC |
| `constexpr`/pack ternary | | ✓(...??) | |
| Expansion statements | P1306R1 [23] | ✓ | |
| **Tier 2** | | | |
| Reflection | P1240R2 [24] | ✓ | |
| Packs in structured bindings | P1061R2 [17] | ✓ | Implemented in clang |
| Generalized structural types | P2484R0 [19] | | |
| Integer sequences in the core language | | ✓ | |
| Deducing forwarding reference | P2481R1 [14] | ✓ | |

| | | | |
|---|---|---|---|
| Pack Slicing | P1858R2 [11] | ✓ | |
| Universal template parameters | P1985R1 [2] | ✓ | |
| Variable template template params | | ✓ | |
| Concept template template params | | ✓ | |
| Deleting variable templates | P2041R1 [21] | | |
| Packs outside of templates | P1858R2 [11] P2277R0 [13] | ✓ | |
| Non-trailing parameter packs deduction | P2347R2 [7] | | Implemented in Clang |
| `std::forward` in the language | P0644R1 [10] | | |
| Simplified structured bindings protocol | P2120R0 [12] | | |
| `std::is_structured_type` | | ✓(`__is_structured_type`) | |
| `static_assert` with expression as message | | | |
| `static_assert` packs | | ✓ | |
| **Tier 3** | | | |
| Packs as return value | | | |
| Destructuring aggregates/C-arrays | P2141R0 [9] P2580R0 [6] | | |
| Pack literals | | | |
| Meta algorithms on packs (`unique`, etc) | | ✓ | |
| **Not proposed (Tier 4)** | | | |
| Multiple levels of pack expansions | | | |
| Language variants/tuples | | | |
| Pack of variables declaration | | | |
| Step for pack slicing | | ✓ | |
| Member traits(Dot syntax on types/enums) | | ✓ | |

## Scope and Goals

### Not a paradigm shift

The C++ committee likes big, bold complex, and paradigm-shifting proposals, like concepts and reflection. This is no such proposal. Most of the proposed features and highlighted proposals are logical extensions or generalizations of existing features. Most are patterns that many people have tried to express only to find they can't. As such the combined goal of the presented features is not as much to make C++ more exciting as it is to make it less surprising and less frustrating.

### Iterative Improvements

This paper touches on pack handling, template template parameters, type deduction, diagnostics, `constexpr` features, etc. The features presented have various degrees of maturity and implementation experience. We need to find a balance between having a good long-term vision of the design space, and incremental improvements, such that well-understood features, are not held back by more controversial or less mature proposals. It is not our intention to present a large proposal, but several small to medium-sized proposals with compounded benefits. We should strive to standardize many of these individual features for C++26.

### Interaction with reflection

There is some overlap between what is considered Template Meta Programming, `constexpr` facilities, and reflection. The features presented in this paper are orthogonal to reflection (P1240R2 [24]) and do not intend to compete with reflection features. As reflection can be used as a mechanism to manipulate types, there will inevitably be some problems that can be solved by both reflection and template metaprogramming techniques, or a combination of the two. The best tool to use for a given task will be context-dependent, and this paper does not try to hold any particular tool as the best. Waiting for reflection should not stop us from improving the rest of the language.

### Circle

Many of the features presented in this paper were designed and implemented in Circle by Sean Baxter. There is a lot of commonality between this paper and the work presented by Sean Baxter at CppNow in May 2022. This is very much intentional: We were really impressed by this body of work and the underlying philosophy that the language should be incrementally improved to fit the need of library and user types, rather than relying on arcane, slow to compile, hard to diagnose template meta-programming heroics.

## Template parameters

### Universal template parameters

Universal template parameters and non-type template template parameters are covered in P1985R3 [?]. The goal of this feature is to be able to handle generically templates entity regardless of the kind of entity they are specialized with.

```
template <typename T, std::size_t BufferSize = default_size>
class small_vector;
view | ranges::to<small_vector>();
```

Without universal template parameter, the above code cannot be made to work, as `small_vector` has a non-type template parameter. Universal template parameters cannot be emulated by library solution, and their absence simply prevents generic code to support many generic types.

P1985R3 offers a holistic approach to the use of universal template parameter (template parameters of any kind) by considering them dependent until instantiation.

The approach chosen by Circle is to only allow universal template parameters as template arguments and template parameters. They can notably be used in primary template declarations which are then specialized by kind. While less encompassing than what P1985R3 proposes, it has fewer moving pieces and implementation complexity, and could prove sufficient as a first iteration. Either way, P1985R3 [?] is a good candidate for further experimental implementation.

### Variable and concept template template parameters

The impossibility to pass concepts and variable templates as template parameters makes imposssible to express ideas such as ”range of integrals” or ”tuple of floating points”. Circle supports concepts and variable templates as template parameters.

The syntax for these are

- `template <template </*...*/> typename V>` : Type template template parameter
- `template <template </*...*/> auto V>` : Variable template template parameter
- `template <template </*...*/> concept C>` : Concept template template parameter
- `template <template </*...*/> template auto U>` : Universal template template parameter

It is both the syntax chosen by Circle, but also the only logical syntax that falls out of the existing grammar.

Universal template parameters and concept template template parameters are complementary features. Consider for example

```
template <typename V, template auto T> // Primary universal template
constexpr bool __is_range_of /*= delete*/;
```

```
template <typename V, template <typename> concept C> // Specialization for concepts
constexpr bool __is_range_of<V, C> = C<V>;

template <typename V, typename T> // Specialization for concrete types
constexpr bool __is_range_of<V,T> = std::is_same_v<V, T>;

template <typename R, template auto T>
concept range_of = __is_range_of<std::remove_cvref_t<std::ranges::range_reference_t<R>>, T>;

// We can now constrain a range to a specific type
static_assert(range_of<std::string, char>);

// Or a concept
static_assert(range_of<std::string, std::integral>);
```

There is no syntax ambiguity between universal template parameters (`template auto`) and variable template template parameters (`auto`). I do not think there needs to be new syntax for disambiguating in dependent contexts, ie before specialization the only thing the compiler needs to know is that a given entity is a template disambiguated by `template`, which can be used for newly introduced entities.

Note that, by large, concepts behave very much like a variable template of type bool except they provide the compiler opportunity for better diagnostics and can be used to constrain parameters, variable declaration, etc. It is therefore important to support both.

```
template <template <typename T> concept C>
auto f(C auto x) -> C auto {
    return x;
}
```

However, concepts and variables are still distinct entities and a concept cannot be passed to a template expecting a variable. ie, the following is not supported by Circle.

```
template <template <typename T> auto V>
struct S {};

template <typename T>
concept C = true;

S<C>;
```

To keep implementations and partial specialization rules manageable, this should not be supported by the standard.

### Partial concepts application

We are **not** proposing to support the following

```
template <range_of<convertible_to<int>> R> void f(R&&);
```

While concepts usage with the first parameter exists as syntax in selected concepts - namely for parameter declaration ie (`template <convertible_to<int> foo>`), they do not exist as entity in the compiler and would be ambiguous in the presence of variadic parameters.

This use case can be emulated with an interface forwarding the concepts parameters as extra arguments, for example

```
template <class R, template <typename> concept C, template auto...Args>
concept range_of = input_range<R> and C<range_reference_t<R>, Args...>;

template <range_of<convertible_to, int> R> void f(R&&);
```

This does not require new facilities - beyond the ability to pass a concept as parameter, but it is less practical to use. Another solution would be to use a usage side syntax market to differenciate partially applied concepts from boolean variables:

```
template <class R, template <typename> concept >
concept range_of = input_range<R> and C<range_reference_t<R>>;

template <range_of<concept convertible_to<int> R>
void f(R&&);
```

This requires more analysis, but the support for partially applied concepts could be added afterward.

## Parameter Packs

Packs are fundamentally sequences of expressions or types. There exist, however no standard facilities to iterate, index, search, or modify these sequences. Most techniques to manipulate these sequences rely on recursion and rely on the instantiation of a number of types that is linear - sometimes quadratic - with the size of the packs, leading to high development costs and long compile times. By improving language support for packs we can, all at once improve compile times, diagnostic messages, and ease of development.

### Indexing

One of the most fundamental operations that can be done with a sequence is to index it: Given an index `n` and a pack of types, we want to know the `n`th type and given a pack of expressions, we want the nth expression.

Here is a meta-programming facility extracting the Nth type:

```
template <std::size_t I, typename T, typename ...Ts>
struct nth_element_impl {
    using type = typename nth_element_impl<I-1, Ts...>::type;
};

template <typename T, typename ...Ts>
struct nth_element_impl<0, T, Ts...> {
    using type = T;
};

template <std::size_t I, typename ...Ts>
using nth_element = typename nth_element_impl<I, Ts...>::type;
```

This code was taken from a blog post written by Louis Dione in 2015. This is the most straightforward approach presented.

It's a recursive solution, which is already non-trivial to write for a lot of C++ developers and it is the simple solution presented. Most importantly, Louis's benchmark shows that this is very slow to compile.

The conclusion of the article led to Clang implementing a built-in `__type_pack_element` builtin.

```
template <std::size_t N, typename... T>
using nth_element = __type_pack_element<N, T...>;
```

GCC is considering a similar facility.

An internet search reveals a large number of people searching for an answer to this exact problem.

Both P1858R2 [11] and Circle propose to solve this problem with a pack subscripting operator:

```
template<std::size_t N, typename... T>
auto get(T&&... args) -> std::remove_cvref_t<T...[N]> // Indexes a pack of type {
    return args...[N]; // Indexes a pack of expression
}
```

This can then be used to implement `std::get` like facilities, simplify standard and library times interfaces, etc. The following code, for example, implement `std::variant::variant()`.

```
template<class... Types>
class variant {
public:
    constexpr variant()
    noexcept(std::is_nothrow_default_constructible_v<Types...[0]>)
    requires(std::is_default_constructible_v<Types...[0]>);

};
```

### Negative Indexing

Both Circle and P1858R2 [11] propose negative indexing

```
template <typename... T>
using Last = T...[-1];
static_assert(std::same_as<Last<int, double, long>, long>);
```

`T...[-1]` is equivalent to `T...[sizeof...(T)-1]`.

This is fairly useful as getting the last element of a pack is a frequent operation.

Pack indexing is covered in more detail in P1858R2 [11] and [P2662R0] . P1858R2 [11] Demonstrate how pack indexing can be used to simplify `std::tuple` and tuple-adjacent facilities.

The proposed syntax is valid today (with a different meaning) however, as explained by Barry, the current meaning (declaring a pack of C arrays without a name) is neither super useful, nor used, nor implemented by all compilers ([Compiler Explorer]).

Pack indexing can be standardized and implemented independently of other features proposed in this paper.

Negative indexing, however, presents some challenges. Namely, other languages (C#, Rust to cite a couple) have observed that this capability prevents distinguishing intended negative index from underflowing index computations.

To solve this problem, we could do what C# does, ie introduce a different sigil for negative indexing - C# uses `^`.

## Pack slicing

Slicing, or subsetting a pack is also a fairly common operation. In particular, many constraints or requirements apply to all elements but the first, or all elements but the last.

For example, `std::ranges::cartesian_product_view`, and almost all supporting functions is defined as

```
template<input_range First, forward_range...Vs>
class cartesian_product_view : public view_interface<cartesian_product_view<First, Vs...>> {};
```

Because of the different requirements on `First` and the necessity to be able to extract `First` is many places.

Having the ability to take a subset of the pack can simplify that code:

```
template<input_range...Vs>
requires (forward_range<Vs...[1:]> && ...)
class cartesian_product_view : public view_interface<cartesian_product_view<Vs...>> {};
```

Circle supports a third step parameter, which can be used to stride the pack, or reverse it, and that seems less useful. It is not proposed by P1858R2 [11] (ie, there is a wide variety of algorithms that are useful on packs, and it is not clear that reversing a pack or striding it are more common than other operations).

It is important to note that slicing yields a pack rather than its elements, so there might be dots on both sides

```
auto f(int a, int b);
int g(auto... pack) {
    return f(pack...[1:3]...); // pack...[1:3] slices the pack and ... expands it
}
```

Note that the indexes of pack slices cannot themselves be pack

```
template<int... Is>
void f(auto... xs) {
    g(xs...[Is:]...); // ill-formed because Is is a pack
}
```

This is to avoid having multiple level of packness in expressions, which would bring tons of complexity for little value. In particular, if `Is` could be a pack, would the expansion apply to the index or the pack, how would we distinguish the two?

## Member Packs

The ability to declare a pack as a member is covered in P1858R2 [11]. This is a critical ability for the implementation of `mdspan`, `tuple`, `variant`, etc. Current implementation strategies for these classes rely on inheritance as it is possible to inherit from a pack. But, in addition to creating a large number of templates and therefore slow compile times, such a structure make any further access to the members difficult to implement.

```
template <typename...T>
class tuple {
    [[no_unique_address]] T ...elem;
};
```

This of course needs to work for unions too

```
template <typename...T>
class variant {
    union {
        T ...member;
    };
    std::size_t active_index;
};
```

Supporting packs as members requires:

- The ability to declare a pack of members - the syntax here is consistent with function parameter pack declarations
- Tweaking the CTAD rules for aggregate so that they can support deducing the size of the pack
- Supporting initializing a pack in the member initializer list. However, this requires special consideration. We might want to not initialize the whole pack in the same way.

  ```
  template <typename...T>
  struct s {
       T ...member;
       s() : member()... {}; // default construct a pack
       s(auto... args) : member(args)... {}; // with arguments
       s() : member...[0](42), member()... {}; // handle one of the member differently
  };
  ```

- Default member initializer: We probably want to support initializing a member pack with a pack ( such that each member is initialized with a different value). This is of lesser importance.

C++ already supports member packs. But only in a limited capacity: they can appear in the closure object of a lambda expression. As demonstrated by Eric Niebler, this can be leveraged to implement a `std::tuple`-like type. Despite the challenge of implementation (due, once again to the lack of pack indexing, etc), leveraging the ability to declare member packs in lambdas results in 1.6x faster compile times [Benchmark].

This demonstrates the need for making this ability a first-class feature.

## Generalized Unpacking

Circle supports expanding a tuple, *tuple-like*, aggregates as a pack.

```
template <typename Tuple>
auto apply(std::invocable auto&& f, const Tuple & t) {
    return f(t.[:]...);
}
```

`.[:]` returns a pack of all the elements in the tuple, which is then expanded by `...`.

The same syntax can be used to slice a tuple to a pack by specifying indexes

```
static_assert(std::tuple{std::tuple{1, 2, 3, 4}.[1:3]...} == std::tuple{2, 3});
```

Like for structured bindings, the tuple protocol (`std::tuple_size`, `std::tuple_element`, `get`) is used to construct a pack.

Indexing is also possible:

```
static_assert(std::tuple{1, 2, 3, 4}.[1] == 2);
```

In circle the `.[]` syntax can be applied to both types and values

```
static_assert(std::tuple<int, double>.[0] == int);
```

The same features were proposed in P1858R2 [11], except that it uses the syntax `Type::[]` to index, produce and slice a pack of elements' types.

For tuple-like types, this is just a light dusting of syntactic sugar over `get` and `tuple_element` (both of which are otherwise made more ergonomic my other features presented in this paper, notably the ability to get the current expansion and pack aliases).

However, it allows to unpack aggregates of arbitrary size to a pack, which is a new capability

```
static_assert(S.[0] == int);
static_assert(S{42, 3.1415}.[0] == 42);
```

This can be very useful for serialization for example. Reflection should allow to get the types of members as a pack fairly easily - although in a more cumbersome way. The following code (or siomething similar to it) could be used to extract the nth data member of an aggregate using reflection (P1240R2 [24])

```
S s{42, 3.1415};
static_assert(s.[:std::meta::members_of(^S, std::meta::is_nonstatic_data_member)[0]:] == 42);
static_assert(std::tuple{s....[:std::meta::members_of(^S, std::meta::is_nonstatic_data_member):]...}
    == std::tuple{{42, 3.1415}});
```

Given the difference in usability, the generalized unpacking syntax should be considered independently of reflection.

## Packs in structured bindings

P1061R2 [17] proposes to allow declaring a pack in structured bindings, to convert a tuple into a pack. This is a sensible feature that is a bit redundant as the following functions would be equivalent.

```
template <typename... types>
auto apply1(auto f, std::tuple<types...> t) {
    return f(t:]...);
}

template <typename... types>
auto apply2(auto f, std::tuple<types...> t) {
    auto && [...pack] = t;
    return f(pack...);
}
```

In general, packs and tuples should offer the same ergonomy, so it is not clear that unpacking a tuple-like into a pack structured binding has a lot of benefits over simply unpacking it in place.

However, the paper offers to give names to part of the pack (ie `auto [first, ...tail]`), which could improve readability in some cases.

It also allows to reuse an unpack tuple, or to materialize a temporary tuple as an l-value

```
auto [...pack] = get_tuple();  // Store the tuple in an lvalue and unpack it
f(pack...);          // Use it once (possibly modifying it)
g(pack...);          // Use it again
```

But then again, this is similar to

```
std::tuple copy = get_tuple();  // Store the tuple in an l-value
f(my_tuple.[:]...); // unpack it (generate a sequence of get<> calls)
g(my_tuple.[:]...); // unpack it (generate the same sequence of get<> calls)
```

General unpacking and packs in structured bindings are overlapping features, and we should focus on standardizing one of them.

## Pack in non-dependent contexts

By allowing packs as public members variables, aliases, and supporting expansions of tuple-like types, and packs in structured bindings, we find ourselves in the presence of packs in non-template contexts. This is a departure from current implementation techniques whereby packs are expanded as part of template specialization and substitution. This is a significant change that requires implementation experience. The main concern expressed by EWG is that the compiler is not aware that an expression will be a pack expansion while parsing it, and make wrong assumptions. It was suggested that additional syntax at the start of the expression would help the compiler. Presumably, something like

```
...pack....
```

To quote Barry Revzin, that's a lot of dots.

Alternatively `...` could be used either before or after the expansion and always before in non-dependent contexts. That's a lot of inconsistencies to teach, especially as the placement of `...` is something students have reported having trouble with.

Circle demonstrates that this additional syntax is not necessary.

Packs in structured bindings (P1061R2 [17]) have been implemented in Clang by Jason Rice(Compiler Explorer). In effect this implementation treats a pack as dependent and performs pack exansion and substitution of the pack once the complete pack expanding expression is parsed.

The implementation technique used by Jason can be used to support other kinds of non-dependant packs.

We recommend more implementation experience with this feature before burdening users with more inconsistent syntax that would only exist for the not-yet-established benefits of some implementations. Because pack expansions can only appear in specific contexts, it should be possible to parse an expression assuming it's not a pack expansion and then transform it when it is established than it is without affecting the compile times of existig code. Either way, an implementation is necessary before making premature syntax decisions. If determined to be possible, any added implementation complexity is preferable to added complexity in the language.

More importantly, the other features presented in this paper are not contingent on a resolution to this question being found, and the issue of packs in non-dependent contexts can be shelved until more implementation experience is available without holding progress back. Pack expansions in non-dependent contexts can just be made ill-formed for now. Not ideal but better than not progressing everything else Circle, this paper and related proposals offer.

## Non-trailing packs

P2347R2 [7] proposes the deduction of non-trailing parameter packs. Some of the use cases proposed by this paper can be supported by pack indexing and slicing. However, one of the main use cases for P2347R2 [7] is to allow function interfaces with a semantically meaningful parameter order, rather than one forced by the language. Another use case is better support of source location. Neither use cases are served well by pack indexing/slicing, even if they can somewhat be emulated. as such **P2347R2 [7] should still be pursued, at least for functions**.

In other contexts where overload resolution does not take place, non-trailing pack deduction can be replaced by pack indexing without loss in expressivity.

## Integer sequences

Current implementation techniques rely heavily on index sequences to index, splice, and manipulate packs. In the standard, it is used in the implementation of `tuple`, `zip`, `variant`, `bind_back`, `array` and other facilities. A search on GitHub reveals over 15000 uses in open source projects.

It is important to observe that many uses of `integer_sequence` could be replaced by other facilities presented in this paper. For example, `std::tuple` uses `std::integer_sequence` to create indexed elements holder that are inherited from. That can be replaced by member packs. `std::integer_sequence` is also used to splice and index elements, or to switch over elements of a pack, all of which can be replaced by pack indexing, slicing, and expansion statements.

Nevertheless, generating a sequence of integers is still a useful operation.

Observing that, Circle offers the following facilities to generate a list of integers as a pack of variables:

- `int...`: Generate a pack of consecutive integers between 0 and the size of the current expansion (so it can only be used in the context in the context of an expansion)
- `int...(N)` : Generate a pack of `N` consecutive integers starting at 0
- `int...(begin:end:step)` : Generate a pack of integers in the range `[begin, end)` with a step of `step`.

```
template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>& t,
const tuple<UTypes...>& u) {
   constexpr size_t N = sizeof...(TTypes);
   static_assert(N == sizeof...(UTypes));
   return (... && (get<int...(N)>(t) == get<int...>(u)));
}
```

Like other Circle features, `int...(N)` can be used outside of a dependent context.

```
int main() {
    constexpr std::array v{1, 2, 3};
    return (0 + ... + v[int...(v.size())]);
}
```

As such, `int...(N)` is just a facility to stamp integer literals.

### What is wrong with `integer_sequence`

`integer_sequence` relies on type deduction, and as such cannot be used directly, it requires the introduction of an additional class or function template.

For example, tuple equality can be written in standard C++ as follow:

```
template<class... TTypes, class... UTypes>
constexpr bool operator==(const std::tuple<TTypes...>& t,  const std::tuple<UTypes...>& u) {
    constexpr std::size_t N = sizeof...(TTypes);
    static_assert(N == sizeof...(UTypes));
    return [&]<auto...I>(std::index_sequence<I...>) {
        return (... && (get<I>(t) == get<I>(u)));
    }(std::make_index_sequence<N>{});
}
```

Similarly, using an integer sequence as part of a base class specification requires some contortions

```
template <auto N, typename T>
struct Holder { T elem; };

template <typename I, typename...T>
struct tuple_impl;

template <auto... I, typename...T>
struct tuple_impl<std::index_sequence<I...>, T...> : Holder<I, T>... {};

template <typename...T>
struct tuple : tuple_impl<std::make_index_sequence<sizeof...(T)>, T...> {};
```

Contrast that with Circle, which is much better at expressing the intent:

```
template <auto N, typename T>
struct Holder { T elem; };

template <typename...T>
struct tuple : Holder<int..., T>... {}; // int... is a shorthand for int...(sizeof..(T))
                                        // as that's the size of the pack under expansion
```

Maybe more critically, vendors have found `make_index_sequence` to have a major negative impact on compile times.

GCC added a builtin `__integer_pack(N)` which is similar to `int...(N)` in Circle, except it only works in templates. Clang provides `__make_integer_seq` that instanciates an `integer_sequence`.

These approaches are very different but solve the same problem: implementing integer sequences in the language has a huge impact on performance. Quoting the commit that introduced the built-in in clang:

> `__make_integer_seq<std::integer_sequence, int, 90000>` takes 0.25 seconds.
>
> `std::make_integer_sequence<int, 90000>` takes unbound time, it is still running. Clang is consuming gigabytes of memory.

Note that `std::make_integer_seq` can only be implemented recursively, and so needs to instantiate $\mathcal{O}(n)$ types.

### Improved library solution 1: Unpacking integer sequences

We can subscribe `integer_sequence` to the tuple protocol.

```
namespace std {
    template <class T, T... N>
    struct tuple_size<integer_sequence<T, N...>>
    : integral_constant<size_t, sizeof...(N)>
    { };

    template <size_t I,  class T, T... N>
    struct tuple_element<I, integer_sequence<T, N...>> {
        using type = T;
    };

    template <std::size_t I,  class T, T... N>
    constexpr auto get(std::integer_sequence<T, N...>) {
        return N...[I];
    }
}
```

We can then unpack the sequence without relying on deduction, either with a structured binding or with the generalized `.[:]` syntax:

```
template<class... TTypes, class... UTypes>
constexpr bool operator==(const std::tuple<TTypes...>& t,  const std::tuple<UTypes...>& u) {
    constexpr std::size_t N = sizeof...(TTypes);
    static_assert(N == sizeof...(UTypes));
    constexpr auto seq = std::make_index_sequence<N>{};
    return (... && (get<seq.[:]>(t) == get<seq.[:]>(u)));
}
```

This lets us declare the sequence as a local variable and gets rid of the nested lambda. You can find an example of that technique on Compiler Explorer.

It is also possible to use the same technique in a base specifier,

```
template <typename... T>
struct tuple : Holder<std::make_index_sequence<sizeof...(T)>{}.[:], T>... {};
```

But here, the index sequence is created `sizeof..(T)` times and a single element is kept each time. Not ideal. Not necessarily evident, and the syntax may also surprise non-experts.

We also did not solve the performance aspect, and still rely on compiler magic to save us.

### Improved library solution 2: Using ranges?

Taking this idea further, we can observe that any constexpr random access range can be turned into a destructurable type, and therefore, a pack.

```
template<auto Rng>
requires std::ranges::random_access_range<decltype(Rng)>
struct destructurable_range {
    using value_type = std::ranges::range_value_t<decltype(Rng)>;
    using reference  = std::ranges::range_reference_t<decltype(Rng)>;
    static constexpr auto & range = Rng;
    template <std::size_t I>
    constexpr decltype(auto) get() {
        return range[I];
    }
};
namespace std {
    template <auto Rng>
    struct tuple_size<destructurable_range<Rng>>
    : integral_constant<size_t, ranges::size(destructurable_range<Rng>::range)>
    { };
    template <size_t I,  auto Rng>
    struct tuple_element<I, destructurable_range<Rng>> {
        using type = typename destructurable_range<Rng>::value_type;
    };
}
int main() {
    auto [... elems] = destructurable_range<std::views::iota(3, 10)>{};
    return (0 + ... + elems);
}
```

This however relies on ranges being structural types (see P2484R0 [19]). There is at least the intent that `std::vector` should be a structural type so, at least the following should be supported:

```
    auto [... elems] = destructurable_range<std::views::iota(3, 10) | std::ranges::to<vector>>{};
    return (0 + ... + elems);
```

This has the nice property of being to yield complex sequences of arbitrary types and can leverage the suit of standard algorithms and views to produce interesting non-sequential... sequences.

It does however suffer the same issues as the previous solution: Performance pitfalls (in terms of compile times) without the introduction of a variable, and harder to use than it probably should be for a relatively common operation in generic code.

### Introducing a language feature

Given that, we ought to consider a language solution, similar to Circle's `int...(N)`. The syntax chosen by Circle is entirely workable but also a bit surprising. Ideally, we would want a pack size type to be `std::size_t`, and so the token `int...` may lead to confusion.

It is, however, surprisingly difficult to find a better alternative. We probably want some kind of keyword here. but good keywords are hard to find.

Gašper Ažman observed that ”an integer sequence of the size of the current expansion” is not sufficient, as illustrated in this example in which two differently sized tuples are expanded:

```
template <typename T>
```

```
int expand(auto&&...);

template <typename... Ts, typename... Us>
auto f(auto&& g, std::tuple<Ts...> x, std::tuple<Us...> y) {
    (..., expand<Us>(
        (..., expand<Ts>(
            g(int...(sizeof...(Ts)), std::get<int...(sizeof...(Ts))>(x),
              int...(sizeof...(Us)), std::get<int...(sizeof...(Us))>(y))))));
};
```

More work is needed to fylly understand the requirements of a good syntax here but we could adopt Circle `int...` syntax, or find an identifier that no one uses (such as `integer_pack`) and support 3 use cases:

- Create a sequence of the size of the current (innermost) expansion
- Create a sequence of user-provided size `N`
- Create a sequence between Start and End
- Create a sequence of the size of a user-provided pack id (as an alias of `int...(sizeof...(T))`)

## Pack aliases

One interesting feature introduced in P1858R2 [11] is the ability to declare a pack alias. A compelling use case is presented in P2120R0 [12]

```
template <typename... Ts>
class tuple {
    [[no_unique_addres]] Ts... elems;
public:
    using ...tuple_elements = Ts;
    template <size_t I>
    constexpr const auto& get() const& {
        return elems...[I];
    }
};
```

This could be used by the language and the library as a simplification of the tuple protocol. It could also be used to implement meta-algorithms.

```
template<typename...T>
using ...unique_types = __builtin_unique_types<T...>;
```

We could assume `__builtin_unique_types` is a built-in that deduplicates the types in `T...` and produces a pack. Given that this is a common operation in the sender/receiver world, such a builtin would be useful.

In effect, currently, all meta algorithms operating on packs can only be manipulated through type lists, and using `apply`/`quote` meta functions, as for example in MP11 or Eric Niebler's `meta` library.

Pack aliases allow meta-algorithm whose results are types. This has the potential to simplify meta-programming heavy libraries, including much of the sender/receiver machinery.

For completeness, let us see if we can implement `unique_types` using reflection. We are going to make some assumptions:

- `std::set` is constexpr (it currently isn't).
- `std::vector<std::meta::info>` is a structural types.

```
template <std::meta::info i>
using reify =  typename [:i:];

template<auto range_of_meta_info>
using ...unique_types_impl = reify<range_of_meta_info.[:]>;

template<typename...T>
using ...unique_types =
    unique_types_impl<destructurable_range<[] {
                std::vector<std::meta::info> v{^T...};
                auto [last, end] = std::ranges::unique(v);
                v.erase(last, end);
                return v;
            }()>{}>;
```

In the above code, we first reflect on types - `meta::info` then contains a `meta::info` object for each type. We can use regular algorithms to remove duplicates. This deduplicated vector can then be lifted back to the template/type world by using an NTTP. We wrap it in `destructurable_range` so that the vector - which is a constexpr random access sized range - acts as a destructurable type. Which we can then expand and reify, giving us, a pack of types back.

### tuple protocol

P2120R0 [12] demonstrates how pack aliases can be used to simplify greatly the tuple protocol.

Copying the code from this paper:

```
template <typename... Ts>
class tuple {
    Ts... elems;
public:
    using ...tuple_elements = Ts;

    template <size_t I>
    constexpr auto get() const& -> Ts...[I] const& {
        return elems...[I];
    }
};
```

Here, the `...tuple_elements` pack alias is sufficient for the tuple protocol to deduce the size of the tuple (ie `sizeof...(tuple<...>::tuple_elements)`), and the type of each element (tuple<...>::tuple_elements...[N] is the type of the Nth element).

In addition, we should consider make std::get a customization point object so that the tuple-like concept and structured binding behave in a similar fashion, and allowing non-standard types to be tuple-like.

### Pack Aliases in Sender/Receivers

Senders/Receivers P2300R5 [5] need a mechanism to check whether a given sender can be connected to a receiver. In effect, a sender may produce different value types, and each such value is itself multiple arguments.

This is explained in libunifex's documentation

```
struct some_typed_sender {
    template<template<typename...> class Variant, template<typename...> class Tuple>
    using value_types = Variant<Tuple<int>,
    Tuple<std::string, int>,
    Tuple<>>;
};
```

This sender supports the following methods:

- set_value(R&&, int)
- set_value(R&&, std::string, int)
- set_value(R&&)

To query these set of value types the documentation advises using the following pattern

```
typename unifex::sender_traits<some_typed_sender>::template value_types<std::variant, std::tuple>
```

Every subsequent manipulation of these types - such as concatenation, deduplication, etc which are common operations, needs to use a mix of apply and quote meta functions.

Pack aliases can let us simplify the interface.

```
template <typename... T>
struct sender_args {
    using... types = T;
};

template <typename... T>
using ...pack = T;

struct some_typed_sender {
    using ...value_types = pack<
        sender_args<int>,
        sender_args<std::string, int>,
        sender_args<>
    >;
};
```

## Single level of packness

`unifex::sender_traits<some_typed_sender>::value_types` is a pack of `sender_args`, and `sender_args` represents a pack of types. Pack of types? This is tempting, but here probably lies madness. The author spent some time trying to figure out how multiple levels of packness could possibly work in the general case. For example, could we write a `sender_of` concept as a one-liner?

```
template<class S, class... Ts>
concept sender_of = ((same_as<Ts,
typename unifex::sender_traits<some_typed_sender>::value_types::types && ...) || ...);
```

I don't think there is a good model to make that work, and it should probably not be attempted. It is almost always possible to construct a wrapper such that there is never more than one level of packness.

```
template<class S, class Args>
concept sender_of_impl =
(same_as<Args, unifex::sender_traits<some_typed_sender>::value_types::types> && ...);

template<class S, class... Ts>
concept sender_of = sender_of_impl<S, sender_args<Ts...>>;
```

Some operations would benefit from multi-level pack supports, namely `concat/join`-like operations (`tuple_cat` for example). We do not have a perfect solution to this.

Circle uses what it calls "argument-for".

> *argument-for* is a *generic-argument* construct. It loops over types, non-types, templates, a counter, or a multi-dimensional set of counters, from inside an argument list. Circle hasn't yet added syntax for nested pack expansion; it's possible, but complicated. argument-for is the tool for effecting this same thing.

This is an interesting feature which is a limited form of code injection (see P2237R0 [22]) to produce types and expressions imperatively.

With this, `tuple_cat` can be implemented as follow:

```
template<class... Tuples>
constexpr tuple<
for typename Ti : Tuples =>
    typename std::remove_cvref_t<Ti>::tuple_elements...
>
tuple_cat(Tuples&&... tpls) {
    return {
        for i, typename Ti : Tuples => std::forward<Ti>(tpls...[i])...
    };
}
```

It might be interesting to explore this in the future depending on how reflection and code injection evolves. It is the most novel feature presented here, and as powerful as it could be, it has a less compelling cost/benefit than other features presented in this paper. Reflection should be good enough as the imperative mechanism, and we should avoid having too many ways to do the same operations. We certainly should revisit that question if reflection does not progress or proves unsuitable.

### Packs as return values

Because tuple and packs are isomorphic, Supporting packs as return values does not have semantic or expressiveness benefits over returning a tuple.

```
tuple<int, int> f();
void g(auto...);

int main() {
    g(f().[:]...); // expands the return value of f() into g.
}
```

However, Lewis Baker noticed that this inhibits RVO and that there may be value in supporting packs as return values so that RVO can be supported.

A possible solution would be

```
void g(auto...);
auto f() -> ...(int, float) {
    return ...(42, 3.14f);
}

auto [i, f] = f();
g(f()...); // performs rvo
```

More research is necessary.

## Control flow statements/expressions

### Constexpr conditional: `constexpr?`

Circle has a constexpr ternary conditional operator. It is the ternary equivalent to `if constexpr`. Circle spells that operator `??`

```
constant-expression ?? a : b
```

If *constant-expression* is true, the expression is equivalent to `a`, otherwise `b`. Unlike the regular ternary operator, there are no common type requirements between the two branches. Like `if constexpr`, the branch that will not be evaluated is discarded - and need not be semantically valid.

This constexpr equivalent to `?:` is very useful where `if constexpr` might prove impractical to use, such as in a member initializer list - It can be emulated with an immediately invoked lambda.

The following example illustrates how this operator is used inside Circle's `mdspan` implementation:

```
template<size_t... Extents>
struct extents {
    [[no_unique_address]] storage_t<Extents> ...m;

    template<SizeType... IndexTypes>
    constexpr extents(IndexTypes... exts) :
        m( dynamic_extent == Extents ??  exts...[find_dynamic_index<int...>] : Extents)... { }
};
```

To avoid conflicts with P2561R0 [15], but also to improve readability, we propose `constexpr?` for the syntax of this new operator instead.

```
constant-expression constexpr? a : b
```

This is a nice to have feature, that simplifies code that can be written differently, but it is also an easy feature to specify and implement.

## multi-conditional: `...? :`

```
expanded-pack-of-convertible-to-bool-expressions ...? pack_of_expression : expression
```

This operator, present in Circle, works like a conditional operator but operates on an expanded pack of conditions, and a pack of expression. And return the corresponding expression for the first true condition. The else expression is returned when no condition is true:

```
int find_true(auto... b) {
    return b ...? int... : -1;
}

int main() {
    assert(find_true() == -1);
    assert(find_true(false) == -1);
    assert(find_true(true) == 0);
    assert(find_true(false, true) == 1);
    assert(find_true(false, true, true) == 1);
}
```

With a pack of boolean and b a pack of expression, all of size 3, the following expression:

```
a... ...? b : c
```

Is equivalent to

```
a...[0]? b...[0] : a...[1] ? b...[1] : a...[2] ? b...[2] : c;
```

There is a surprisingly high number of use cases for such an operator. It's a nice way to find a correspondence between a runtime condition and a compile-time table.

```
template <typename...Ts>
class variant {
    union {
        Ts ...member;
    };
    std::size_t active_index;
    ~variant() {
        _index == int... ...? member.~Ts() : std::unreachable();
    }
};
```

This is roughly semantically equivalent to:

```
template <typename...Ts>
class variant {
    ~variant() {
        template for(constexpr auto idx : pack_index<Ts...> )  {
            if(idx == active_index) {
                member...[idx].~Ts...[idx](); ?
                break;
            }
        }
    }
};
```

## Constexpr multi-conditional `... constexpr?:`

`... constexpr?:` is the logical combination of `...?:` and `constexpr?:`

Like `constexpr?:`, it evaluates conditions at compile time, does not require a common type and the branch not taken are discarded, and like `...?:`, the (constant) conditions, and the expressions are packs.

```
template<typename Arg, typename... Fn>
auto call_first(Arg&& arg, Fn&&... f) {
    return std::invocable<Fn, Arg> ...constexpr? f(std::forward<Arg>(arg)) : []<typename...> {
        static_assert(false, "No valid function");
    }();
}
```

`... constexpr?:` is less compelling than either `...?:` and `constexpr?:`, but it falls out naturally of these two things, and should be considered for completeness.

Note that, for better compatibility with packs and code patterns like the one above, Circle treats `static_assert` as an expression.

## Other Circle features and improvements desirable to improve template meta-programming

### Member traits

Circle allows querying the value of a type trait by calling *`.trait`* on a type. Most of the traits available in `<type_traits>` and `<limits>` can be querying using this syntax.

```
T.remove_cv_ref  // same as std::remove_cvref_t<T>
int.max          // same as std::finite_max_v<int>;
```

Circle observes that member syntax on types is unused design space that can be used to provide syntactic sugar. There is also an argument to be made for compile times, given that some type traits are used very frequently, this syntax, implemented entirely in the compiler is therefore more efficient.

However, compilers can already provide builtins for these. Recent library work allows the `_v`/`_t` versions of these traits to be independent (P1715R0 [3]), and individual numerics type traits are being added - hopefully in C++26 (P1841R3 [4]).

Therefore, the usefulness of menber traits is debatable and as our time is limited, we do not consider this an important feature for WG21 to spend time on. But, the existence if this feature in Circle is illustrative of the usefulness of implementing frequently used type traits as builtins.

### Additional members on types

Circle also supports querying:

- `T.template` : The class template of a specialization `T`
- `T.type_args`, `T.nontype_args`, `T.template_args`, `T.universal_args` : The template arguments of a specialization `T`. (These four ways to get to the template parameters exist to disambiguate types and variables during parsing)
- `T.string`: The name of a type, template or enum as a `const char*`. This simplified form of reflection is notably used by Sean baxter to improve `static_assert` messages.

### Member traits on packs

If member traits on packs are sugar over existing features, Circle also provides a few member functions on packs. These functions are currently:

```
.filter
.sort
.unique
.contains_all
.find
```

The meta functions, called on packs, return either an element, an index, or another pack.

```
template <typename... T>
using filtered_tuple = std::tuple<T.filter(std::is_integral_v<_0>)...>;

static_assert(std::is_same_v<std::tuple<int, long>,
                             filtered_tuple<int, long, const char*>>);
```

There is currently no good way to achieve the same result. However, if reflection allows to convert a pack to a vector of `meta::info` and back, the same result can be achieved using standard algorithms, and so any consideration for this feature should be delayed, assuming reflection makes progress. We should however make sure reflection on a pack and back is a well supported, efficient use case with convenient syntax.

### Algorithms on packs

Regardless of the implementation prefered (pack member meta function, reflections, builtins, or type list handling), it would be useful for the standard to provide some algorithms to handle packs of types.

`P0949R0` ”Adding support for type-based metaprogramming to the standard library” propose many such algorithms to handle type lists. `contains`, `find`, `filter`, `unique`, `concat` are particularly useful algorithms when hamdling a list of types, and used fequently in `range-v3`, `libunifex`, and Circles demostrates that they are useful in the implementation of standard classes such as `tuple`, `variant`, `mdspan`.

In addition, even with better supoort for packs, `type_list` and related features could be useful to implement algorithms such as concat:

```
template<typename... Types>
struct type_list{
    using ...types = Types;
};

template<typename... Types>
struct concat;
template<typename... Types, typename... UTypes>
struct concat<type_list<Types...>, type_list<UTypes...>> {
    using ...types = type_list<Types..., UTypes...>...;
};
```

### `std::is_structured_type`

A type trait indicating whether a given type can be destructured can be useful, for example, to implement a generic formatter/serializer.

### `constexpr` structured bindings

Structured bindings are one of the rare features not supported in `constexpr` contexts.

This is discussed in P1481R0 [8](Nicolas Lesser).

### `std::forward` in the language

In generic library, `std::forward<decltype(x)>(x)` is rather tedious to write, and many prominent C++ experts suggest defining a macro.

```
#define FWD(x) std::forward<decltype(x)>(x)
```

Others resort to using static_cast (or C casts!) - in part for performance reasons. Note that both GCC and Clang now understand `std::forward` in the front end to guarantee it never generates code, improving debug performances.

P0644R1 [10] (Barry Revzin) proposes a very nice solution to this problem.

## Diagnostics improvements

Two common complaints about meta-programming are the compile times, and the diagnostics messages. The various features presented thus far improve both issues. By lifting common features in the compiler, we can reduce drastically compile times (a lot fewer types need to be created when instantiating `tuple`, for example), and improve diagnostic messages (again, because the code becomes much simpler).

However, more can be done to improve diagnostics.

### `static_assert` using expressions as messages

Being able to pass string-like expressions (ranges of `char`, or `char8_t`) to `static_assert`, would allow libraries to construct better diagnostic messages.

```
template <std::string str>
constexpr auto ctre_compile() {
   constexpr std::expected<regex, error> res = do_compile(str);
   if constexpr(!res) {
       static_assert(res, std::format(u8"Invalid regex '{}': {} at column {}",
                                      str, res.error().message(), res.error().column()));
   }
}
```

### `static_assert` pack

Circle treats `static_assert` a bit like an expression to support it in multi conditionals, and to have packs of `static_assert`.

```
template<class... Types>
constexpr bool operator<(const variant<Types...>& v,  const variant<Types...>& w) {
    static_assert(requires{ (bool)(get<int...>(v) < get<int...>(w)); },
        std::format("{} has no operator<", std::meta::name_of(^Types)))...;
}
```

There have also been efforts to support `static_assert` in non-dependent contexts P2593R0 [16], and P1936R0 [1] explores a mechanism that could be adapted to support user-crafted warnings.

## How do these things fit together?

The features presented in this paper do not compete with one another. They also do not compete with reflection, nor do they compete with one another. We should make the simplest things simple, and the more complicated things, slightly more involved.

Reflection, for example, can be used to implement type traits. It allows to express more complex or stateful algorithms but doesn't have the expressivity of a fold expression.

Yet, we have to admit that there are a lot of moving pieces.

### Structured Types, `std::tuple`, ranges, packs

`std::tuple` is a standard structured type. That is, a type that holds variables of heterogenous (non-type-erased) types, and which can be decomposed - using structured binding or the tuple-protocol, at compile time.

A pack then is an unstructured list of types or variables, which makes a tuple a way to wrap a pack. A tuple and a pack of variables are isomorphic, and a type list and a pack of types are also isomorphic (a tuple can also be used as a type list).

Whether a wrapper is useful (and whether one should use a pack or a tuple) is context-dependent. Functions and algorithms usually require a tuple or another kind of structured type.

Ranges are homogeneous and mutable types, so at first glance unrelated. However, a constexpr range can be turned into a tuple or a pack. and a homogenous pack can be turned into a range.

Further, using reflection, packs of types or heterogenous packs of values can be turned into a range of their reflection, and back again. This gives us a bridge between the template meta-programming world and the imperative world. Various reflection papers have proposed short-hand syntax to reify ranges of `meta::info` into a pack. But this can be emulated using NTTP. However, to make this work, we need something along the lines of P2484R0 [19]

### On native types

#### Language variants

P0095R2 [18] proposes a new kind of entity representing a variant, in the core language. However, the problems presented in the paper, are not as much as with `std::variant` as they are with `std::visit`. The paper also admits that the tradeoffs made by `std::variant` are not always ideal and proposes a different set of tradeoffs... which may also not be ideal.

The solution, in our mind, is not to move high-level opinionated constructs in the core language, but rather to make these types easier to write, and to support pattern matching, in the same way, the tuple protocol enables structured bindings for user types.

### Language tuples

P2163R0 [20] argues in favor of a core language feature to replace `std::tuple`. It might have been preferable to gain some sort of native tuple instead of `initializer_list` in the first place, given a tuple is a heterogeneous sequence and therefore useful in more contexts. However, that ship has sailed, and given that, the complexity of introducing a language tuple now would likely not be worth the reward. Indeed, we have presented features that would allow writing a tuple-like type in a few lines of code. `std::tuple`'s complexity can be removed by making the language more expressive, without a language-level tuple type, and that is more applicable as it enables users to write different *tuple-like* types, with different constraints, conversion supports etc.

As such, packs are the "language-level" tuples. some of the features presented here are already overlapping, and introducing new entities, with new deduction rules at the language-level without fully understanding the interactions could only lead to a repeat of `std::initalizer_-list`. Instead, the support for packs can be extended coherently such that there is no need for language tuples.

## Other Related works

### P2041R1 [21] Deleting variable and class templates

This paper proposes to allow deleting variable and class templates so only designated specialization are declared.

### P1240R2 [24] Scalable reflection

The main reflection proposal. We have showed that switching from template meta programming and back could allow users to manipulate type using their meta info where handling types would otherwise be complex or clunky. As such the two proposal complements each others.

### P2481R1 [14] Deducing forwarding reference of a specific types

P2481R1 [14] proposes a way to deduce qualifiers independently of the type of the deduced parameter, as to reduced the number of overload needed in many of the standard types. This problem was independently identified by Sean Baxter who added the ability to better deduce forwarding references in Circle. Note that the aformentioned mentionned only explore the design space and does not offer a specific solution. Circle's approach seems the most suitable, in particular it does not introduced yet another kind of template parameter/entity (unlike the `Qualifier` approach), and should be relatively easy to implement in other compilers.

### P2484R0 [19] Extending support for class types as non-type template parameters

The inability to use classes with private members as NTTP is limiting. This is especially true of `std::vector`, as some pattern presented in this paper rely on the ability to pass vectors as template parameters, as a bridge between ranges and packs, and reflection and packs.

### P1306R1 [23] Expansion statements

P1306R1 [23] is not mentioned in this paper because it has past the design stage. En expansion statement produces a different compount statement for each element of a range or a tuple. It does work on packs (as this was found ambiguous), however this is not an issue as a pack can trivially be expanged inside a tuple.

## Acknowledgments

We would like to thank Bloomberg for sponsoring this work.

Sean Baxter for his work on Circle.

Barry Revzin, Gašper Ažman and the many authors whose work we referenced in this paper.

Matt Godbolt and the maintainer of Compiler explorer for hosting experimental implementation of some of the features presented here.

Jason Rice for the implementation of P1061R2 [17].

Lewis Baker, for his valuable feedback on this paper.

Aaron Ballman for his guidance on the clang internals.

## References

This paper and associated work was inspired and based on Sean Baxter's excellent talk Circle Metaprogramming: Better Features Make Better Libraries.

The documentation for circle was used extensiuvely in the preparation of this doocument and can be consulted on circle-lang.org and Github.

[1] Ruslan Arutyunyan. P1936R0: Dependent static assertion. `https://wg21.link/p1936r0`, 10 2019.

[2] Gašper Ažman, Mateusz Pusz, Colin MacLean, and Bengt Gustafsonn. P1985R1: Universal template parameters. https://wg21.link/p1985r1, 5 2020.

[3] Jorg Brown. P1715R0: Loosen restrictions on "_t" typedefs and "_v" values. https://wg21.link/p1715r0, 6 2019.

[4] Walter E Brown. P1841R3: Wording for individually specializable numeric traits. https://wg21.link/p1841r3, 2 2022.

[5] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, and Bryce Adelstein Lelbach. P2300R5: 'std::execution'. https://wg21.link/p2300r5, 4 2022.

[6] Paolo Di Giglio. P2580R0: Tuple protocol for c-style arrays t[n]. https://wg21.link/p2580r0, 5 2022.

[7] Corentin Jabot and Bruno Manganelli. P2347R2: Argument type deduction for non-trailing parameter packs. https://wg21.link/p2347r2, 10 2021.

[8] Nicolas Lesser. P1481R0: constexpr structured bindings. https://wg21.link/p1481r0, 1 2019.

[9] Antony Polukhin. P2141R0: Aggregates are named tuples. https://wg21.link/p2141r0, 5 2020.

[10] Barry Revzin. P0644R1: Forward without forward. https://wg21.link/p0644r1, 10 2017.

[11] Barry Revzin. P1858R2: Generalized pack declaration and usage. https://wg21.link/p1858r2, 3 2020.

[12] Barry Revzin. P2120R0: Simplified structured bindings protocol with pack aliases. https://wg21.link/p2120r0, 2 2020.

[13] Barry Revzin. P2277R0: Packs outside of templates. https://wg21.link/p2277r0, 1 2021.

[14] Barry Revzin. P2481R1: Forwarding reference to specific type/template. https://wg21.link/p2481r1, 7 2022.

[15] Barry Revzin. P2561R0: operator?? https://wg21.link/p2561r0, 7 2022.

[16] Barry Revzin. P2593R0: Allowing static_assert(false). https://wg21.link/p2593r0, 5 2022.

[17] Barry Revzin and Jonathan Wakely. P1061R2: Structured bindings can introduce a pack. https://wg21.link/p1061r2, 4 2022.

[18] David Sankel, Dan Sarginson, and Sergei Murzin. P0095R2: Language variants. https://wg21.link/p0095r2, 10 2018.

[19] Richard Smith. P2484R0: Extending class types as non-type template parameters. https://wg21.link/p2484r0, 11 2021.

[20] Mike Spertus and Alex Damian. P2163R0: Native tuples in c++. https://wg21.link/p2163r0, 5 2020.

[21] David Stone. P2041R1: template = delete. https://wg21.link/p2041r1, 3 2021.

[22] Andrew Sutton. P2237R0: Metaprogramming. https://wg21.link/p2237r0, 10 2020.

[23] Andrew Sutton, Sam Goodrick, and Daveed Vandevoorde. P1306R1: Expansion statements. https://wg21.link/p1306r1, 1 2019.

[24] Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, and Faisal Vali. P1240R2: Scalable reflection. https://wg21.link/p1240r2, 1 2022.

[P2662R0] Corentin Jabot, Pablo Halpern, John Lakos, Alisdair Meredith, and Joshua Berne
Pack Indexing
https://wg21.link/P2662R0
October 2022

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4885