Data Members, Variables and Alias Declarations Can Introduce A Pack

Document #:	P3115R0
Date:	2024-02-15
Programming Language C++	
Audience:	EWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

In P1858R2 [7], Barry Revzin presented a design for alias and member packs. A similar set of features has been available for several years in Circle.

In this paper, we refine that set of features with the goal of standardization in C++26, report on implementation experience in another C++ compiler (clang) and explore synergies with reflection.

This paper is a work-in-progress that we hope to update ahead of the Tokyo meeting.

Motivation

Here are some examples showing the use of the features presented in this paper. They are accompanied by working compiler explorer links, showcasing an in-progress Clang implementation.

Tuple-like types

```
template <typename ...Ts>
struct tuple {
    [[no_unique_address]]  Ts ...elems;

    tuple(Ts... ts) : elems(ts)... {};
};

template <typename ...Ts>
tuple(Ts...) -> tuple<Ts...>;
```

[Compiler Explorer]

Making `tuple` easier to implement would help with the quality of diagnostic and compile-time performance of the standard libraries. But more importantly, there are reasons for users to want to write their own tuple-like types.

They might want a tuple that behaves like an aggregate, or they might want to use a lighter data structure, have data member specific to their use cases, etc.

bind-like utilities

```
template <typename Functor, typename... Args>
struct bind {
    [[no_unique_address]] Functor f;
    [[no_unique_address]] Args... args;
    template <typename Self, typename... Xs>
    auto operator()(this Self&& self, Xs&&... xs) -> decltype(auto) {
        return std::invoke(
        std::forward<Self>(self).f,
        (std::forward<Self>(self)....args)...,
        std::forward<Xs>(xs)...);
    }
};
template <typename Functor, typename... Args>
bind(Functor&& f, Args&&... args) -> bind<Functor, Args...>;


template <typename = void>
auto test() {
    return bind{[](int a, int b, int c) {
            return a + b + c;
        }, 1, 2}(3);
}
```

[Compiler Explorer]

Monadic Transform

```
template <typename Functor, typename... Args>
struct transform {
    [[no_unique_address]] Functor f;
    Args... args;
    template <some_optional Optional, typename Self>
    requires std::same_as<std::decay_t<Self>, transform>
    friend auto operator|(Optional&& opt, Self&& self) {
        using f_result = decltype(std::invoke(std::forward<Self>(self).f,
                                  std::forward<Optional>(opt).value(),
                                  (std::forward<Self>(self)....args)...));
        using return_type = std::optional<std::remove_cvref_t<f_result>>;
        if (opt)
            return return_type(std::invoke(std::forward<Self>(self).f,
                                           std::forward<Optional>(opt).value(),
                                           (std::forward<Self>(self)....args)...));
        return return_type(std::nullopt);
    }
};
template <typename Functor, typename... Args>
transform(Functor&& f, Args&&... args) -> transform<Functor, Args...>;
```

```
template <typename = void>
auto test() {
    return (std::optional{2} | transform{[](int a, int b, int c) {
            return a + b + c;
        }, 2, 3}).value();
}
```

[Compiler Explorer]

Similar to the previous example, this shows how easy it is to construct functors that can behave as aggregate.

std::execution (P2300)

In P2300R7 [5], senders provide a list of completion signatures, ie a list of list of parameters they may pass to the `set_value` and `set_error` functions of a receiver. This helps check whether a given sender could be connected to a given sender.

Variant types

Pack members can be member of an anonymous union. This allows implementing variant as a union (rather than an aligned buffer).

```
template <typename ...Ts>
struct variant {
    union {
        Ts ...data;
    };
};
```

The implementation of the constructors is not trivial (and left as an exercise for the reader!). While it is true that there is already a variant class in the standard, there are valid cases for rolling out variant implementation for specific use cases, for example in some cases it is possible to use unused bytes of the types to store the variant discriminant.

Struct of arrays container

A few months ago, a Reddit user posted a comparison of using circle vs C++26 reflection (as implemented in EDG):

We have reproduced the circle implementation in clang. Note that in the absence of aggregate unpacking, our implementation requires a tuple-like type:

```
template<class T>
class soa_vector {
    public:
    constexpr void push_back(const T & value) {
        // We could use structured binding here
        (v.push_back(UNPACK(value)), ...);
    }
```

```
    [[nodiscard]] constexpr auto operator[](const auto pos) const {
        return T(v[pos]...);
    }
    private:
    std::vector<...tuple_elements_ts<T>> ...v;
};

template <typename T>
void test() {
    using vec3_t = tuple<float, float, float>;
    soa_vector<vec3_t> v{};
    v.push_back({1, 2, 3});
    assert(1 == v[0].elems...[0] and 2 == v[0].elems...[1] and 3 == v[0].elems...[2]);
}
```

[Compiler Explorer].

Note that this was also presented by Sean Baxter at CppNow.

More ergonomic `std::integer_sequence`

A one-line change to `std::integer_sequence` could make it a lot easier to use, as we would not need to pass it to a lambda/function to extract the actual pack on integers.

```
namespace std {
    template<class T, T... I> struct integer_sequence {
        using value_type = T;
        static constexpr size_t size() noexcept { return sizeof...(I); }
        static constexpr decltype(I) ...values = I;
    };
}
```

The grand organic plan

Before C++11, if you wanted a tuple, it might have looked like that

```
// boost::tuple
template <
class T0 = null_type, class T1 = null_type, class T2 = null_type,
class T3 = null_type, class T4 = null_type, class T5 = null_type,
class T6 = null_type, class T7 = null_type, class T8 = null_type,
class T9 = null_type>
class tuple;
```

WG21 saw that this could be improved and N2242 [1] added variadic template parameters and function parameter packs in C++11. This was a huge success, and it is used in virtually every C++ library from tuples to JSON parsers, CTRE, fmt, Qt, testing frameworks, HPC applications, std::exec, TensorFlow, embed applications, and everything in between.

In C++17, we realized that if a base specifier could be a pack, then we needed to allow using declarations to be packs too (P0195R2 [2]). This enables efficient implementation of the least-terrible way to visit a variant.

In C++20, we fixed that lambdas could not capture a function parameter pack (P0780R2 [6]).

And more recently we made packs indexable (P2662R3 [4]), a feature that library vendors promptly requested to be made available to them in older language modes, as it has the potential to simplify standard library implementations as well as compile times.

We are about to allow befriending packs (P2893R1 [3]), so we clearly like them.

I think this success can be explained by making metaprogramming less arcane and allowing a surprisingly expressible functional style to build types and expressions in linear times.

Fold expressions in particular, even if their syntax takes some getting used to are a very powerful way to express some ideas and are less bug-prone than trying to write the same computation manually.

And then there is ”Structured Bindings can introduce a Pack” (P1061R6 [9]), currently making its way through the committee. This paper is of particular significance because

- Packs are no longer tied to templates in some way.
- Consequently, expansions can now appear in non-templated entities, which is novel.

Untangling packs from parameters in implementation is going to require some work, if only because some implementations might refer internally to parameter packs :). However, it does create a model and the groundwork to support tside of templates.

In summary, pack declarations can appear in:

- Template parameters
- Function parameter
- Init capture
- Structured bindings

That list feels rather incomplete, and in this paper, we explore extending it to data members, variables, and alias declarations.

It should be noted that this paper is entirely based on existing work, and the ideas presented here have been discussed since at least 2016. Notably, in the trailblazing P0341R0 [10] and excellent P1858R2 [7]. The Circle C++ Compiler extensively experimented with most of the same ideas.

But before talking about exciting new features, let’s build a common understanding of packs as they exist today.

A primer on pack syntax

A pack is a name that represents a sequence of entities (variables, types, template template parameters).

Packs can be part of a wider pattern, and when instantiated, they are replaced by some transformation of the pattern, ie an expansion. Any substitution of the pattern is an expansion (including `sizeof...` and pack indexing).

So there are really 2 fundamental categories of operations that can be done to a pack:

- Designating an entity as being a pack.
- Expanding a pack, potentially as part of a wider pattern.

That's really all there is to it. However

- Expansion can be arbitrarily complex (for example, they may involve lambdas).
- A number of proposals (this one, expansion statements (P1306R1 [11]), the generalized unpacking mechanism proposed in P1858R2 [7]) introduce constructs that, if not very carefully designed, could lead to a host of grammar ambiguities.

The syntactic model that we have is, for some pack `T`:

- `...T` introduced a pack
- `T...` expands (consumes) the pack.

We have been pretty good at following that model, even if `sizeof...(T)` is somewhat of an outlier, and even if we had to break some arcane/unused/unimplemented syntax in order to ensure pack indexing is consistent with that model.

Consistency is important for teaching, implementability, and making sure we don't close evolutionary doors or introduce hard-to-fix ambiguities.

Multiple packs can be expanded at once, in lockstep, as long as they have the same size.

Keeping that in mind, what is this paper actually proposing? We are going to present Alias Declaration Packs, Variable Packs, and Member Packs in that order because the notions, design, and implementation challenges roughly build up in that order.

Alias Declaration Packs

If an alias declaration introduces a type, an alias pack declaration introduces a pack of type, and the defining-type-id is the pattern that will be substituted when the type is used.

```
template <typename T>
struct S{
    using value_type = std::remove_cvref_t<T>;
};
```

```
template <typename ...Ts>
struct S{
    using ...value_types = std::remove_cvref_t<
    Ts>;
};
```

This brings the bulk of the design and implementation questions of this proposal, the rest are incremental additions.

Packs outside of templates

Because alias declarations can appear at namespace and class scope, they can be named outside of template entities.

```
template <typename ...Ts>
void g();
void f() {
    g<S<int, int>::value_types...>();
}
```

Packs outside of templates are not really a new feature, assuming P1061R6 [9] gets adopted, the specification and implementation will be able to support them (including at namespace scope).

But compilers that are not AST-based might rely on the fact that P1061R6 [9] can only declare packs at function scope in a statement that appears before any expression might expand the pack. Depending on that implementation choice, dealing with a pack that appears out of thin air in an expansion might be quite a bit harder to support for such implementations, than the limited scenario where a pack is previously introduced. This problem has been described in P2277R0 [8].

Note that this argument might be bit moot as P1061R6 [9] is going in the direction of allowing structure bindings at global scope. So all implementation will have to be prepared to see a pack in any function, if a structured binding is declared at namespace scope.

And so, extending to member packs would not have any additional complexity. To be clear, we are aware that for one implementation the complexity will be great… but the present paper is not materially affecting that one way or another.

We have roughly two options here:

- Not allowing some packs to be expanded outside of templates (users would have to whip out a function / generic lambdas), creating inconsistencies for users.

- Bite the bullet understanding it will take significant implementation efforts.

I did briefly consider a third alternative: Adding some syntax to advertise the presence of packs in the function. However this would be a transient solution pushing the burden of implementation toward users, and neither transient solutions, not putting undue burden on users seems compelling.

Plus, if we wanted a transient syntax to advertise the possibility of pack expansion in a function then making the function a template ought to be good enough.

```
template <std::same_as<void> T = void> // not great, not terrible
auto get_deduplicated_variant()  {
    auto variant = get_variant();
    return std::variant <
        unique_types_ts<decltype(variant)...
    >(std::visit(std::identity{}, variant));
}
```

But ultimately, I think we should bite the bullet. Knowing this is going to require significant effort for some implementations, knowing our resources are scarce and our time is limited, we should bite the bullet.

We have been having that discussion for many years now, making incremental progress in that direction with P1306R1 [11] for example, and there is a lot of interest in the features presented here, so we have an understanding the effort would justify itself...

But mostly, I think we have a shared understanding getting there is unavoidable. Indeed, a range of meta info is a pack, and so reflection needs this feature too.

Citing P1306R1 [11]:

> However, range splicing of dependent arguments is at least an order of magnitude harder to implement than ordinary splicing. We think that not including range splicing gives us a better chance of having reflection in C++26. Especially since, as this paper's examples demonstrate, a lot can be done without them.

It would be a missed opportunity to design reflection without the ability to expand ranges of reflections, and we risk introducing mismatched facilities and inconsistencies. Concepts, Modules, constexpr cmaths, parallel algorithms, etc. were all standardized with the understanding of being long-term efforts.

We should note that the implementation concerns for packs outside of templates discussed here would overlap with module implementation concerns. Implementations retaining an internal representation of template entities during translations are not impacted by that specific concern.

Nested Packs? Dependent Pack Names

Given

```
template <typename ...Ts>
struct S {
    using ...types = Ts;
};

using Type = S<int, int>;

template <typename T>
int f() {
    do_something<Type::types...>(); // # 1
    return do_something<typename T::types...>(); // # 2
}
int a = f<Type>();
```

In #1, `Type::types` is a pack of types, which the compiler knows, so that does not present any particular issue. In #2, `T::types` is a dependent name. The compiler doesn't know it is a type, which it needs to in order to resolve parsing ambiguities, hence the use of `typename` as disambiguator in `typename T::types`.

But we don't know that `T::types` names a pack either. This is something we need to communicate so that we can diagnose invalid expansion in non-instantiated templates… but more crucially, we need to be able to identify where the packs are so that we can extract their size prior to extraction.

We use the `...` **prefix** to designate a dependent name as a pack. This is very much purposefully consistent with the declaration of packs; In both places, we are telling the compiler a name is that of a pack. and the syntax ought to be the same.

The `...` prefix should be allowed everywhere a nested pack is named, but required only for dependent names. Unlike `typename`, (and more like `template`), the `...` applies to a name, rather than a fully qualified type. This is because a qualified name could contain more than one pack.

Nested packs? No

Consider:

```
template <typename ...Ts>
struct A {
    using ...inner = Ts;
};

template <typename ...Ts>
struct B {
    using ...outer = Ts;
};

using Matryoshka = B<A<int>>;
```

Both `Matryoshka::outer` and `Matryoshka::outer::inner` are packs. And this is perfectly fine.

However, we do not have a good model for expanding multiple levels of packs at once, although this is something that was explored by Barry Revzin.

But it is something that would require a lot more thought and implementation experience. In particular, the generalization of that idea is a cartesian product in the language, which seems unwieldy.

Ideally, simple ideas should be easy to express, and complex ideas should be possible. If multiple levels of pack expansion are necessary, then other techniques might be more legible.

Maybe there is an argument to be made for the specific case of turning a pack of tuple-like into a pack of their elements.

So for this paper, In a pack expansion, if the pattern would expand multiple component names, the program is ill-formed.

```
DoSomething<Matryoshka>(); // Ok
DoSomething<Matryoshka::...outer...>(); // Ok
DoSomething<Matryoshka::...outer...[0]::...inner...>(); // Ok
DoSomething<Matryoshka::...outer::...inner...>(); // Error
```

Like any other packs, packs presented in this paper can also be expanded in step with other (non-nested) packs of the same size.

Dependent packs need to be substituted before they can be expanded

```
template <typename ...Ts>
struct A {
    using ...types = Ts;
};
template <typename T>
void f() {
  std::variant<T::...types...> v;  // #1
}

f<A<int, int>>();
```

Currently, the size of packs is inferred from template arguments or (in the case of init captures, structured bindings), from entities already instantiated. The presence of packs in dependent names introduces the need to substitute dependent packs before they can be expanded (#1). This represents extra work for the compiler compared to expanding a pack whose size is known. However, this is still a lot less work than expressing the same idea without access to nested packs.

Packs in type requirements

```
template <typename T>
concept tuple_like = requires {
    typename T::...elements...;
};
```

Variable Packs

The case for variable packs may be less strong than the other features presented in this paper. However, It would be very inconsistent to support structured bindings (really, a form of variable declaration, data members, and using declarations), but not variable pack declarations. So, we propose them.

And they do have some use cases of their own. For example, we can improve the ergonomics of `std::integer_sequence`.

```
template <typename T, T... V>
struct __ints {
    static constexpr decltype(V) ...values = V;
};
template <auto N>
constexpr auto ...ints = __make_integer_seq<__ints, decltype(N), N>::...values;


template<typename... Ts>
void print_tuple(const std::tuple<Ts...>& t) {
    constexpr std::size_t size = sizeof...(Ts);
    (print(std::get<...ints<size>>(t)), ...);
}
```

That same technique can be used, with ranges, to produce a pack out of arbitrary sequences of numbers, for example.

Member Packs

Member packs is the most interesting part of this proposal and builds on what has been presented so far.

```
template <typename T>
struct monotype {
    [[no_unique_address]] T elem;
};
```

```
template <typename ...Ts>
struct tuple{
    [[no_unique_address]] Ts ... elems;
};
```

Simply, a data member can be declared to be a pack, and that packs expand N members (which then behave like regular data members). This is supported in classes, union and struct, for data members and anonymous members, with the exception of bit-fields.

[TODO: Is there actually a reason to disallow bit-fields]?

Accessing dependent member packs

As previously discussed, dependent pack names are disambiguated with a `...` prefix. From that, a dependent member expression whose right-hand side is a pack looks like this:

```
template <typename... T>
void f(tuple<T...> t) {
   do_something(t. ...elems...);
   do_something_else(t....elems...); // not sensitive to spacing
   do_something_else((&t)->...elems...); // not sensitive to spacing
};
```

While four consecutive dots is a bit odd, it is consistent (and consistency is important). Previous works explored different syntaxes, like the introduction of a disambiguator keyword (`t.packname elems...`), but that does not look better.

To avoid making C++ sensitive to spaces, we propose a tweak to the maximal munch behavior such that a sequence of exactly 4 period is recognized as Period, Ellipsis (rather than Ellipsis, Period), without affecting sequences of 6 periods (`f(auto.....)` being valid syntax!).

`.`	Period
`..`	Period, Period
`...`	Ellipsis
`....`	Period, Ellipsis
`.....`	Period, Ellipsis, Period
`......`	Ellipsis, Ellipsis

Member initializers

Member initializers can be pack expansions, in the same way that base initializers already can be

```
template <typename... Ts>
struct S : Ts... {
    S(Ts... ts) :
        Ts(ts)...,   // can initialize a pack of bases
        elems(ts)... {}   // NEW! can initialize a pack of members
         {}
    Ts... elems;
};
```

CTAD

It is possible to write deduction guides for a class or aggregate that has a member pack.

```
template <typename... Ts>
struct S {
    S(Ts... ts) : elems(ts)... {}
    Ts... elems;
};

template <typename... Ts>
S(Ts...) -> S<Ts...>;
```

However, we should not add deduction guides automatically for

- Aggregates with multiple member packs
- Aggregates with non-trailing packs
- Aggregates with bases that themselves contain member packs.

Designated initializers

TBD

Pointer to members

TBD

Member Expressions, Prvalues, and Packs

If member expressions can name a pack and the LHS of a member expression can be a pr-value, then a member expression can name a pack produced by a pr-value.

This introduces a new design question, perhaps the biggest design question in this paper as everything else is a relatively straightforward application of existing features. Indeed, every other pack expression so far is an l-value.

To illustrate the question, how many times should `f` be evaluated? 1 or 3.

```
template <typename... Ts>
struct S {
    S(Ts... ts) : elems(ts)... {}
    Ts... elems;
};

template <typename... Ts>
S(Ts...) -> S<Ts...>;

auto f (auto... args) {
    std::puts("f");
    return S{args...};
}

int main() {
    return (f(1, 2, 3).elems + ... );
}
```

The question is that of the nature of `f(1, 2, 3).elems`.

Either we consider `f(1, 2, 3).elems` a pattern, which is trivial and does not require additional specification or implementation work (in which case `f`) is called 3 times.). or we realize that `f(1, 2, 3).elems` **produces** a pack which is then expanded (in which case `f`) is called once).

The second model is less surprising for users (after talking with a lot of people with various levels of experience). if the function producing the pack either has side effects or takes time to execute, etc, calling it multiple times does not match the intent.

Worse, if `f()` is evaluated more than once, each expansion of the pack could produce a different result.

```
template <typename... Ts>
struct S { Ts... elems; };

auto f (auto... args) {
    static int i = 0;
    i++;
    return S{(args * i)... };
}

int main() {
    return (f(1, 2, 3).elems + ... );
}
```

This program returns... 14 (`1*1 + 2*2 + 3*3`), which is confusing.

In other words, if the left-hand side of a member expression is evaluated for each expansion of the pack, the identity of the pack itself changes, which seems extremely confusing.

Note that any expression that does not produce the pack being expanded (and that can only be the LHS of a member expression) would be evaluated for each expansion. This itself can arguably be confusing: In the example below, `f` is evaluated 4 times.

```
template <typename... Ts>
struct S {
    Ts... elems;
    operator int();
};

auto f (auto... args) {
    return S{1, 2, 3 };
}

int main() {
    return ((f(1, 2, 3).elems + f()) + ... );
}
```

The other issue with the ”Produce the pack once” model, is that we ought to preserve value categories of expansions. So if a pack is produced by a pr value, the elements of the packs should be x-values.

This could potentially be an issue if pack elements have a reference to the object they are part of.

We haven’t implemented the ”LHS is called once” model yet (we certainly plan to). The idea would be that each member expression would reuse the same materialized temporary.

The 3rd option is to make member pack expansions where the LHS is not an *id-expression* ill-formed, forcing users to declare a variable.

This seems reasonable in the context of this proposal, and it would allow us to make forward progress. The same question will however appear in other contexts notably for reflection range splicers and an an unpacking expression that would unpack an aggregate/constant range/tuple-like entity without introducing a structured binding (such things are described in "Future work" and in P1858R2 [7]).

Implementation experiences

The features presented in this paper, and previous work, have demonstrated implementability twice. First in Sean Baxter's Circle, and more recently in Clang.

Neither implementations cover everything implemented here: Circle does not support packs of aliases.

And Clang (an experimental branch thereof), only has limited support for alias packs and member packs.

Our implementations has no support for packs outside of templates (although I did some experience to validate viability). We intent to complete that work as part of P1061R6 [9].

Implementation of variable packs and variable packs initialization, implicit deduction guides, anonymous union pack members, missing diagnostics, will be worked on both ahead of Tokyo, and ahead of St Louis. What we have so far is an incomplete prototype.

Nevertheless, the implementation effort done so far is sufficient to assert the viability of this paper and to give some measure of the effort involved.

A lot of the architecture changes to support packs outside of templates will be brought about by P1061R6 [9], and what is presented here constitutes incremental changes over that (at least for AST-based implementations).

This is not to say that the implementation effort would not be substantial as we introduce additional ideas. Notably packs in dependent types and nested name specifiers, as well as new syntax to disambiguate packs. More generally we increase the number of entities that can be or contain a pack.

In all, for AST-based implementations, this is a medium-sized set of features, probably accounting for a few months of work.

And it would reduce the burden put on library implementers, which I am sure would be appreciated.

Relation with Reflection

Consider a facility to unique a list of types. This is not a theoretical problem, such things are pretty widely used for example in P2300 implementations. Writing that entirely as a

template meta program is certainly possible. Both `boost::hana` and `mp11` do provide such an algorithm, for example. But these things are widely inefficient in terms of the number of template instantiations, and they are hard to write.

A better alternative would be to use reflection, and leverage existing algorithms like `std::ranges::sort` / `std::ranges::unique` to produce the list of type.

```
std::vector<std::meta::info> unique(std::vector<std::meta::info> v) {
    std::ranges::stable_sort(v);
    const auto [first, last] = std::ranges::unique(v);
    v.erase(first, last);
    return v;
}
```

And then, every time you want a produce a uniqued list of types, you could write

```
typename [: ...unique({^Ts...}) :]...;
```

At the time of writing reflections are not less-than comparable (which is sensible), nor are they hashable (which is surprising), so this example does actually require some amount of suspension of disbelief. Ranges of reflection are also not expandables in the current proposal, so maybe you'd write something like that (assuming you are willing to call `unique` N times, which is certainly a bad idea).

```
typename [: unique({^Ts...})[ints<sizeof...(Ts)>] :]...
```

This is a lot of syntax, resulting from the fact that we are exposing users directly to these domain switches. Instead, we can expose the algorithm as a trait:

```
template <typename... Ts>
using ...unique_types = typename [: ...unique({^Ts...}) :];

static_assert(std::same_as<
    std::tuple<unique_types<int, double, int, int double, int>...>,
    std::tuple<int, double>
>);
```

This is still using reflection, and this is still efficient in the number of instantiations. But reflection is an implementation detail of a more expressive and concise interface.

The point is that there is no conflict between reflection and packs. In fact, they complement. Reflection is an objectively better way to apply non-trivial transformation to a sequence of types, both in terms of being more natural to write and, a lot more efficient in terms of compile time.

At the same time, an API that returns a `meta::info`, or a `range::meta::info` is always going to be clunky compared to producing a type, or a range of type.

Ultimately, reflection and what we propose here are not in conflict, and they complement each other.

The combination of this proposal and reflection enables efficient handling of lists of types. There is a lot of synergies to be found. And generally, "Reflection for C++ 26" should support splicing ranges.

Future work

Pack Literals

During review of the paper, several people have expressed the desire for a way to initialize packs of types / values with some sort of pack literal syntax.

That idea was also introduced in P0341R0 [10], there is really nothing new here. A pack literal syntax would be useful to give default values to packs of variables, for example.

This is not proposed because it can be fairly emulated with other features presented here

```
template <typename ...T>
struct Pack {
    T ...values;
};

constexpr Ts ...a = Pack{0, "Hello", 42}....values;
```

It does bring the question of whether a variable pack can be declared when its type is not a pack but its initializer is.

Unpacking

In the past, we discussed an expression syntax to expand a tuple-like type or an aggregate without the need to introduce a declaration. This would run into the same design question that are discussed for member expressions. Otherwise, it is an orthogonal feature, but one that would be fairly easy to add on top of what we are proposing here, as most of the implementation challenges would have been resolved. Please refer to P1858R2 [7] for more discussion of that feature.

Slicing

Taking a slice of a pack would be useful, especially considering a lot of variadic constructors may have different constraints for the first element. However it is otherwise a fairly orthogonal feature, one that can be pursued independently of this paper, or other pack-related papers.

Splicing

Splicing ranges of reflection, in particular ranges of members has a lot of overlap with the design questions presented there.

The inability to go from the reflection domain to a pack is going to force clunky syntaxes like the use of integer sequences, and is contributing to making these domain boundaries more verbose than they need to be.

Next Steps

Our implementation is not complete, and we evidently do not yet have wording. If EWG is interested in seeing more, we plan to produce both wording and implementation for St Louis.

The main questions we are seeking guidance on at this point are:

- Are we interested in seeing more?
- Do we agree with the full set of features (aliases, data members, static data members, namespace scope variable templates, and alias templates)?
- Do we think the initial version of this work needs some sort of pack literal feature to initialize variable packs?
- Do we want to solve the question of member expressions where the LHS is not an *id-expression*?

Wording

TBD

Acknowledgments

Many people contributed interesting ideas to this paper, including but not limited to Lewis Baker, Hana Dusíková, Sean Baxter, David Ledger, Gašper Ažman, Miro Knejp, Christopher Di Bella, Luke D'Alessandro.

I would also like to thank Barry Revzin for his work on P1858.

References

[1] D. Gregor, J. Järvi, J. Maurer, and J. Merrill. N2242: Proposed wording for variadic templates (revision 2). https://wg21.link/n2242, 4 2007.

[2] Robert Haberlach and Richard Smith. P0195R2: Pack expansions in using-declarations. https://wg21.link/p0195r2, 11 2016.

[3] Jody Hagins. P2893R1: Variadic friends. https://wg21.link/p2893r1, 10 2023.

[4] Corentin Jabot and Pablo Halpern. P2662R3: Pack indexing. https://wg21.link/p2662r3, 12 2023.

[5] Eric Niebler, Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, and Bryce Adelstein Lelbach. P2300R7: 'std::execution'. https://wg21.link/p2300r7, 4 2023.

[6] Barry Revzin. P0780R2: Allow pack expansion in lambda init-capture. https://wg21.link/p0780r2, 3 2018.

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

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

[9] Barry Revzin and Jonathan Wakely. P1061R6: Structured bindings can introduce a pack. https://wg21.link/p1061r6, 12 2023.

[10] Mike Spertus. P0341R0: parameter packs outside of templates. https://wg21.link/p0341r0, 5 2016.

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

[N4971] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4971