Structured naming for function object and CPO values

Draft Proposal

Document #:	P2486R1
Date:	2021-12-15
Project:	Programming Language C++
Audience:	LEWG Library Evolution
Reply-to:	Kirk Shoop <kirk.shoop@gmail.com>

Contents

1 Changes

1.1 R1

— apply feedback from email thread
— add the implementation experience provided by Dvir Yitzchaki

1.2 R0

— first revision

2 Introduction

There was a discussion of naming for the set of `fold` algorithms in [P2322R5]. Naming conversations are particularly fraught because names veer into the subjective and the technical aspects of naming get obscured by the subjective aspects of naming.

I submitted a novel way to name sets of function object values (like CPO sets and other function object sets such as `fold`). This is novel only for function naming, it is established practice for naming of other entities such as: namespace, class, struct, union, etc..

I submitted this very late. I agree that this proposal was not something that should block [P2322R5] from being forwarded. I am under the impression that this name change proposal can be applied after it arrives in LWG. I was asked to write a paper describing the motivation and reasoning behind this naming proposal.

The purpose of this paper is to attempt to explain why structure is a boon to function names. This is a challenge for me because it is so obvious in my head that I do not know where to begin to transfer that perspective into a head that has some unknown barrier to that perspective. For me, the paper should be one sentence:

> “We love structure in statements, classes, namespaces, lifetimes, etc.. - because structure is important for human reasoning about code, and now that we know how to structure sets of functions, we have the opportunity to create structured sets of functions”.

3 Motivation

Statement: Naming sets of things is made much easier by adding nested structure.

3.1 poll to forward ranges::fold

The seed for this paper was an extended process for deciding on names for the set of functions in [P2322R5] ‘ranges::fold’. IMO part of the naming challenge was comping up with a set of unstructured names that represented the clear structure in the set of functions added by the paper.

[P2322R5] was forwarded to electronic polling with this poll:

POLL: Forward P2322R5: ranges::fold with the name set in option D to LWG for C++23 (to be confirmed with electronic poll) priority 2

Strongly Favor	Weakly Favor	Neutral	Weakly Against	Strongly Against
8	4	2	1	0

The names in [P2322R5] option D:

name
`fold_left()`
`fold_left_first()`
`fold_right()`
`fold_right_last()`

These names are roots. The suffix `_with_iter` is applied to expand to the full set of names selected by option D in [P2322R5]

name
`fold_left()`
`fold_left_first()`
`fold_right()`
`fold_right_last()`
`fold_left_with_iter()`
`fold_left_first_with_iter()`

We are used to function names being unstructured. When these were plain functions, unstructured naming was very hard to avoid. [P2322R5] as I understand it, is proposing that the names in the above table, be implemented as function objects. There are also other proposals that are creating names for function objects and CPOs, that could be structured, but are not.

An excerpt from [P2322R5] indicates that more `fold` functions are forthcoming:

> The problem going past that is that we end up with this combinatorial explosion of algorithms based on a lot of orthogonal choices:
>
> iterator pair or range
> left or right fold
> initial value or no initial value
> short-circuit or not short-circuit
> return T or (iterator, T)
>
> Which would be… 32 distinct functions (under 16 different names) if we go all out. And these really are basically orthogonal choices. Indeed, a short-circuiting fold seems even more likely to want the iterator that the algorithm stopped at! Do we need to provide all of them? Maybe we do!

How would a user add missing functionality to the fold set in C++23 in a conditional way that could adopt official implementations in C++26 or beyond? *see: referring to functions*

3.2 unstructured names for things we expect to be structured

These are names for things that C++ users expect to be structured compared with an unstructured alternative.

structured	unstructured
`std::vector<T>`	`std_vector<T>`
`std::vector<T>::value_type`	`std_vector_value_type<T>`
`std::vector<T>::begin()`	`std_vector_begin<T>()`
`std::list<T>`	`std_list<T>`
`std::list<T>::const_iterator`	`std_list_const_iterator<T>`

structured	unstructured
`std::list<T>::begin()`	`std_list_begin<T>()`
`std::chrono`	`std_chrono`
`std::chrono::steady_clock`	`std_chrono_steady_clock`
`std::chrono::steady_clock::time_point`	`std_chrono_steady_clock_time_point`
`std::chrono::steady_clock::now()`	`std_chrono_steady_clock_now()`

The examples of unstructured naming in this table are probably perceived as an absurd straw-man. I would suggest that this reaction is due to an expectation of structure that we have unconsciously assumed was inviolate and obvious. I suggest that much of the committee agrees that structured naming, where it is possible, is necessary and any unstructured alternative is generally considered absurd.

3.3 module naming polls

These polls [moduleNamePolls], for [P2465R0], indicate that there is a preference for '.' separated names (used elsewhere for structured names) over '_' separated names (used elsewhere for unstructured names) even when there is no difference in semantic meaning (modules naming does not have any structure, '.' and '_' separated names are all unstructured names):

POLL: The std:: + global module should be of the form std.X (where X is a placeholder).

Strongly Favor	Weakly Favor	Neutral	Weakly Against	Strongly Against
6	8	6	3	0

POLL: The std:: + global module should be of the form std_X (where X is a placeholder).

Strongly Favor	Weakly Favor	Neutral	Weakly Against	Strongly Against
0	0	10	9	2

For me this is easily generalizable to other names in C++. Even with no semantic difference the notation used for structured access elsewhere in C++ was chosen over naming used for unstructured names elsewhere in C++.

It was a surprise to me to receive this response

> That vote tells me dots are preferred by the people who voted, in the context of module names. It doesn't tell me that the same group think that dots are preferred in non-module contexts. And it tells me nothing about what different groups would say on the subject.

My whole approach to life is to generalize from specific actions, behaviours, choices, beliefs, patterns, etc.. and apply them to other contexts to test them for consistency and applicability and generality. The idea that the vote on module names, in the context of a long history of C and C++ names, only applies to those people, on that day, for that name, is so jarring as to feel absurd - in my head. It is easy for me to accept that perspective is true for others, once stated, but impossible for me to adopt personally. Humans IMO, when not applying patterns across contexts intentionally, still apply patterns from one context into other contexts unconsciously (thus implicit bias). Only by intentionally inspecting patterns and intentionally applying them to different contexts can I track what patterns are being applied. In this case it is obvious, in my head, that the pattern of structure in C++ generally, is being applied here in module naming. I expect that this was conscious for some voters and unconscious for others.

4 Proposals

Given that many new functions are being specified as namespaced function objects and CPOs, there is a structured option available.

Observation: function objects and CPOs can have nested function objects and CPOs as members.

This observation provides the option for using structure for sets of nested functions that would historically use unstructured naming.

4.1 ranges::fold

An example of structured functions as an alternative for the unstructured list of functions in [P2322R5] option D:

structured	unstructured
`fold.left()`	`fold_left()`
`fold.left.with_iter()`	`fold_left_with_iter()`
`fold.left.first()`	`fold_left_first()`
`fold.left.first.with_iter()`	`fold_left_first_with_iter()`
`fold.right()`	`fold_right()`
`fold.right.last()`	`fold_right_last()`

The concrete proposal for this paper is to adopt this structured set of functions in `ranges::fold` for C++23.

Additionally, I propose that any other sets of functions and sets of CPOs that are using unstructured function definitions and that are being added in C++23 should adopt structured definitions for C++23.

4.2 Concepts and CPOs

The core observation of this paper can also be applied to structure other function sets and CPOs being proposed for C++23:

structured	unstructured
`std::execution::receiver.set_value()`	`std::execution::set_value()`
`std::execution::receiver.set_error()`	`std::execution::set_error()`
`std::execution::receiver.set_done()`	`std::execution::set_done()`
`std::execution::sender.connect()`	`std::execution::connect()`

NOTE: `receiver` and `sender` are already defined as concepts today which cannot have members.

The challenge to this naming approach for concepts and structured CPOs today is that without some structured naming solution for concept definitions, we will waste a huge amount of time in LEWG trying to name concepts and their structured CPOs in a way that does not conflict.

NOTE: If I might speculate wildly here. It would be exceptionally useful for the `receiver` and `sender` objects to be able to define `template<class... TN> operator concept<TN...>() = ..;` that would operate as a concept when `receiver<T>` and `sender<T>` were used. This could be an avenue for resolving the naming discussions in LEWG between concepts and their structured CPO names. I expect that some disambiguator would be needed - like `o::template m<>` is today - for objects that have template arguments and an `operator concept<>` definition. Perhaps by attempting to follow existing practice `o<>::concept <>` would be an option. Obviously, I do not expect that a change like this could be applied in C++23.

```
struct receiver {
  template<class T, class Error = std::exception_ptr>
  operator concept<T, Error>() =
    requires (T&& t, Error e) {
      set_done(t);
      set_error(t, e);
    } .. ;

    static inline constexpr set_value = ..;
    static inline constexpr set_error = ..;
    static inline constexpr set_done = ..;
};
template <typename T, typename... Vs>
concept receiver_of = requires (T&& t, Vs... vs) {
    receiver.set_value(t, vs...);
  } && receiver<T>;
```

5 problems this solves

Please don't use an objection over an example given here to reject or ignore the problem and the solution. My examples can be flawed **and** the underlying problem and solution be valid.

5.1 passing function sets as values

A problem this solves is creating sets of related functions that can be passed around as values. This might not be the primary use case for `ranges::fold` but it is an advantage over unstructured functions.

5.2 simplifies 'importing' names into a different naming scope

A problem this solves is importing sets of functions without importing every name in a namespace and without listing out each function name in an unstructured set explicitly and maintaining that explicit list forever. Usually this burden prevents any attempt to constrain the set of names to include from a namespace once it exceeds some human limit for the number of names that the humans responsible are comfortable maintaining.

5.3 fold.left() vs fold::left()

A problem this solves is to improve over namespace use. `fold.left()` allows `fold()` to be made available if that is desired at any point. `fold::left()` does not allow the name `fold` to be a function ever. (This has no impact on `fold_left()` please don't try to use that as a counter to this point about **namespace** usage).

5.4 committee time

A problem this solves is reducing the amount of work to name sets of related functions. Once the structure of the function set is agreed on, the discussions for names will be more easily compartmentalized. Rather than a proposal that has a long list of unstructured names, a name for each object in the structure can be considered individually with only limited examination of the context that the names will be placed.

`fold` easy. `left` & `right` easy. `first` & `last` took more debate over consistency that was resolved by the suggestion that left and right were the differentiators and that this made first and last consistent with left and right. `with_iter` takes more discussion - it could be a separate function set `fold_with_iter` or it could be members added to `left` and `first`.

These become much more clearly separable discussions when we use structured functions. Instead, I never had a clear understanding of the whole set (particularly WRT `_with_iter`) until after the meeting was over, because only some of the set was listed during the meeting.

6 referring to functions

6.1 function sets in a namespace

We use `using` to pull in names from a namespace or base class into a different name scope. How do we do this with structured functions?

On one level, structured functions are just values of trivially-constructible and trivially-copyable types. `using <name>;` works when name is in a namespace, also `auto name = name;` works for members of namespaces and function objects. What would be the expected pattern for using of structured functions in a different name scope?

Table 9: using an entire function set

structured	unstructured

`using std::ranges::fold;`	`using std::ranges::fold_left;`
	`using std::ranges::fold_left_first;`
	`using std::ranges::fold_right;`
	`using std::ranges::fold_right_last;`
	`using std::ranges::fold_left_with_iter;`
	`using std::ranges::fold_left_first_with_iter;`

Table 10: aliasing an entire function set

structured	unstructured

`using reduce = std::ranges::fold;`	`using reduce_left = std::ranges::fold_left;`
	`using reduce_left_first =`
	`  std::ranges::fold_left_first;`
	`using reduce_right = std::ranges::fold_right;`
	`using reduce_right_last =`
	`  std::ranges::fold_right_last;`
	`using reduce_left_with_iter =`
	`  std::ranges::fold_left_with_iter;`
	`using reduce_left_first_with_iter =`
	`  std::ranges::fold_left_first_with_iter;`

6.2 member of a function

Table 11: aliasing a subset of the functions in a set

structured	unstructured

`inline constexpr auto foldLeft =`
`  std::ranges::fold.left;`
	`using foldLeft = std::ranges::fold_left;`
	`using foldLeftWithIter =`
	`  std::ranges::fold_left_with_iter;`
	`using foldLeftFirst =`
	`  std::ranges::fold_left_first;`
	`using foldLeftFirstWithIter =`
	`  std::ranges::fold_left_first_with_iter;`

Table 12: aliasing one function in a set

structured	unstructured

`inline constexpr auto foldLeft =`
`  std::ranges::fold.left;`
`// includes members. like: .first,`
`// .with_iter etc..`
	`using foldLeft = std::ranges::fold_left;`

Table 13: aliasing one constrained function from a set

structured	unstructured

`using base_t = decltype(`
`  std::ranges::fold.left);`
`struct foldLeft_t : private base_t {`
`  using base_t::operator();`
`};`
`inline constexpr auto foldLeft = foldLeft_t{};`
	`using foldLeft = std::ranges::fold_left;`

note: The `foldLeft_t` above is easily generalizable since it has no named members. Something like: `constrained_function<std::ranges::fold.left>` with the template parameter `auto& UnconstrainedFunction`

Table 14: creation of a constrained with_iter subset

structured

```cpp
using basel_t =
  decltype(std::ranges::fold.left.with_iter);
using basef_t =
  decltype(
    std::ranges::fold.left.first.with_iter);
struct with_iter_t {
  struct left_t : basel_t {
    static inline constexpr first = basef_t{};
  };
  static inline constexpr left = left_t{};
};
inline constexpr auto with_iter = with_iter_t{};
```

unstructured

```cpp
using std::ranges::fold_left_with_iter;
using std::ranges::fold_left_first_with_iter;
```

6.3 polyfills

Table 15: creation of a polyfill for fold right if it did not exist

structured

```cpp
using base_t = decltype(std::ranges::fold);
struct fold_t : protected base_t {
  using base_t::left;

  template<typename Base>
  requires requires (Base& b) {b.right;}
  // use the real implementation when it
  // exists
  using right_t = decltype(
    std::ranges::fold.right);

  template<typename Base>
  requires (!requires (Base& b) {b.right;})
  struct right_t {
      // fallback implementation of right
      // when it is missing
  };
  static inline constexpr right =
    right_t<base_t>{};
};
inline constexpr fold = fold_t{};
```

unstructured

```cpp
using std::ranges::fold_left_with_iter;
using std::ranges::fold_left_first_with_iter;
using std::ranges::fold_left;
using std::ranges::fold_left_first;
using std::ranges::fold_left_with_iter;
using std::ranges::fold_left_first_with_iter;

struct fold_right_t {
  template<typename... AN>
  requires
    std::invocable<
      std::ranges::fold_right, (AN&&)an...>
  auto operator()(AN&& an) const {
    return std::ranges::fold_right((AN&&)an...);
  }

  template<typename... AN>
  requires
    (!std::invocable<
      std::ranges::fold_right, (AN&&)an...>)
  auto operator()(AN&& an) const {
    // fallback implementation when it is
    // missing
  }
};
inline constexpr auto fold_right =
  fold_right_t{};

struct fold_right_last_t {
  template<typename... AN>
  requires std::invocable<
    std::ranges::fold_right_last, (AN&&)an...>
  auto operator()(AN&& an) const {
    return
      std::ranges::fold_right_last((AN&&)an...);
  }

  template<typename... AN>
  requires (!std::invocable<
    std::ranges::fold_right_last, (AN&&)an...>)
  auto operator()(AN&& an) const {
    // fallback implementation when it is
    // missing
  }
};
inline constexpr auto fold_right_last =
  fold_right_last_t{};
```

7 Implementation Experience

Dvir Yitzchaki sent me email with both unstructured and structured [implementation]s of the fold set.

Table 16: fold left with iter result

structured

```cpp
struct with_iter_fn {
  template <
    input_iterator I,
    sentinel_for<I> S,
    class T,
    indirectly_binary_left_foldable<T, I> F>
  constexpr fold_left_with_iter_result<
      I,
      decay_t<invoke_result_t<
        F&,
        T,
        iter_reference_t<I>>>>
  operator()(
    I first,
    S last,
    T init,
    F f) const {
      using U = decay_t<invoke_result_t<
        F&,
        T,
        iter_reference_t<I>>>;
      if (first == last) {
          return {
            std::move(first),
            U(std::move(init))};
      }

      U accum = invoke(
        f,
        std::move(init),
        *first);
      for (
        ++first; first != last; ++first) {
        accum = invoke(
          f,
          std::move(accum),
          *first);
      }
      return {
        std::move(first),
        std::move(accum)};
  }
};
```

unstructured

```cpp
template <
  input_iterator I,
  sentinel_for<I> S,
  class T,
  indirectly_binary_left_foldable<T, I> F>
constexpr fold_left_with_iter_result<
    I,
    decay_t<invoke_result_t<
      F&,
      T,
      iter_reference_t<I>>>>
fold_left_with_iter(
  I first,
  S last,
  T init,
  F f) {
    using U = decay_t<invoke_result_t<
      F&,
      T,
      iter_reference_t<I>>>;
    if (first == last) {
        return {
          std::move(first),
          U(std::move(init))};
    }

    U accum = invoke(
      f,
      std::move(init),
      *first);
    for (
      ++first; first != last; ++first) {
      accum = invoke(
        f,
        std::move(accum),
        *first);
    }
    return {
      std::move(first),
      std::move(accum)};
}
```

Table 17: fold left with iter result usage

structured	unstructured
`int main() {`	`int main(){`
`  std::array<double, 2> v = {0.25, 0.75};`	`  std::array<double, 2> v = {0.25, 0.75};`
`  const auto [_, value] =`	`  const auto [_, value] =`
`    std::ranges::fold.left.with_iter(`	`    std::ranges::fold_left_with_iter(`
`      v, 1, std::plus{});`	`      v, 1, std::plus{});`
`  return value;`	`  return value;`
`}`	`}`

8 Q&A

8.1 Q: why should I care?

I struggle to answer this. Not because it is hard to describe why we should care, but because it is such an obvious answer I struggle to understand why the question is asked.

A story tailored to explain my thought process:

In the beginning there was assembly. it was unstructured.

C was designed to bring structure to assembly. Experience with assembly led to an understanding of human limitations that limited code quality and code sharing. The structure added was not motivated by a need to improve the computer's understanding of code. functions and structs and unions and statements (if, while, for, switch) - these were some of the elements that were added to accommodate human limitations.

C++ was designed to bring structure to C. Experience with C led to an understanding of more areas where additional structure would improve human ability to reason about code. constructors/destructors for lifetime, classes, public/private/protected, base classes, nested classes, namespaces for names, templates for stamping out replicas reliably - these were some of the elements that were added to accommodate human limitations.

We should care about structure because we are human and have limitations. We should care about structured functions because they match the structure of member values, member functions, static member functions, nested classes, namespaces and every other good thing that makes C++ easier to reason about than C or assembly.

see problems this solves

8.2 Q: Is this inconsistent with existing practice in the standard library?

It depends. Consistency is relative to what is being associated. Being consistent in one aspect could create an inconsistency in another.

It is consistent with how namespaces, classes, structs, and all other structured entities in the library are defined:

— `std`, `vector`, `list`, `string`, `unique_ptr`, `enable_shared_from_this`, `thread`, `initializer_list`, `numeric_limits`, etc..

It is consistent with existing functions that are not part of a structured set:

— `begin`, `end`, `size`, `advance`, `distance`, `equal`, `mismatch`, `all_of`, `any_of`, `none_of`, `for_each`, `transform`, etc..

It is inconsistent with existing sets of functions that were defined in an unstructured way:

— `copy`, `copy_n`, `copy_if`, `copy_backward`
— `find`, `find_if`, `find_if_not`, `find_end`, `find_first_of`

— etc..

8.3 Q: what are the downsides of using structured function definitions?

from the threads:

— "It is novel"
— "I don't see why I'd want this"
— "half baked idea"
— "I think you are underestimating the impact that a new, inconsistent naming convention would have on the community"
— "I think it's wasteful to spend committee time on it until there is some usage experience that can be cited"

I fully understand that structured function definitions are new and unfamiliar. So is every new feature, for a while. This isn't even a feature, just a realignment of function definitions that will structure sets of functions the same way other entities in C++ are structured.

8.4 Q: Has this idea been field tested anywhere?

No, it has not. Every language feature that this proposal depends on has been thoroughly field-tested. The only novelty here is using those features to add structure to sets of functions.

8.5 Q: Why should ranges::fold be the one to adopt this novel style?

Because `ranges::fold` is inherently a structured set of functions, it is unnecessary to hide the structure in an unstructured list of functions. Because `ranges::fold` is on track to being one of the largest sets of functions to be defined in the standard library. The paper itself says that a total of 16 functions is possible and perhaps desirable.

8.6 Q: does this propose replacing every '_' in every name of every new function object with a '.'?

Nope. I think this is subjective. I do not consider `with` in `with_iter` to represent a space that would contain `iter`. I see `with_iter` as a compound name. In fact, I think that this proposal allows those distinctions to be expressed by choosing to use '`.`' or '`_`' while unstructured naming does not allow those distinctions to be expressed in the name.

I do not think that structured functions should be blindly applied to every function that is not part of a set. I do think that there will be cases where a new function is known to be the first of a set that will be proposed over time and may preemptively be represented as a structured function even if it is initially the only nested function.

8.7 Q: if an object for a name (like fold) is claimed for one purpose, must an unrelated object with the prefix `fold_` become a member?

Nope. Even [P2322R5] option D can be an example of this question. Is `with_iter` a valid member of the `fold` set, or is it a different algorithm with its own distinct set of structured names? I think that the following expression of structured names is just as valid as the one derived from Option D in Proposals:

structured name	unstructured name
`fold.left()`	`fold_left()`
`fold.left.first()`	`fold_left_first()`
`fold.right()`	`fold_right()`
`fold.right.last()`	`fold_right_last()`

structured name	unstructured name
`fold_with_iter.left()`	`fold_left_with_iter()`
`fold_with_iter.left.first()`	`fold_left_first_with_iter()`

The difference between these two is somewhat subjective IMO and I would expect this to be part of the naming discussion.

8.8 Q: should all namespaces become objects (eg. should `views` have been an object instead of a namespace, given this proposal)?

Nope. I think that `views` intends to prevent naming conflicts by providing a common prefix name to a set of functions. The alternative was unstructured names for each function object and CPO that were prefixed with `view_` or suffixed with `_view`. I think that `views` would be valid as an object, with nested objects and CPOs, if its intent was to create a semantic nesting rather than it being intended to use nesting to prevent name conflicts.

The `views` namespace is an example of the common desire and accepted utility of structured names in C++.

Ultimately intent is subjective, so in practice `views` might have gone the other way under different circumstances, but I see the selection of the namespace for `views` as valid and compatible with this proposal.

9 Conclusion

We have an opportunity to apply the structure we love and expect for many entities in C++ to functions specified as objects and CPOs. We know that structure has concrete benefits. We prefer structure for other things. We should take this opportunity to adopt structure for function objects and CPOs going forward.

10 References

[implementation] Dvir Yitzchaki. Fold Implementation (godbolt).
https://gcc.godbolt.org/z/6Mz9rdGb7

[moduleNamePolls] Bryce Adelstein Lelbach. module name polls.
https://github.com/cplusplus/papers/issues/1082#issuecomment-941512716

[P2322R5] Barry Revzin. 2021-10-18. ranges::fold.
https://wg21.link/p2322r5

[P2465R0] Stephan T. Lavavej, Gabriel Dos Reis, Bjarne Stroustrup, Jonathan Wakely. 2021-10-13. Standard Library Modules std and std.all.
https://wg21.link/p2465r0