Practical Reflection

BARRY REVZIN

Cppcon
The C++ Conference

2025
September 13 - 19

About Me

- C++ Software Developer at Jump Trading since 2014

About Me

- C++ Software Developer at Jump Trading since 2014
- WG21 participant since 2016
  - C++20: `<=>`, `[...args=args]{}`, `explicit(bool)`, conditionally trivial
  - C++23: Deducing `this`, `if consteval`, bunch of `constexpr` and ranges papers
  - C++26: Reflection, packs in structured bindings, more `constexpr`

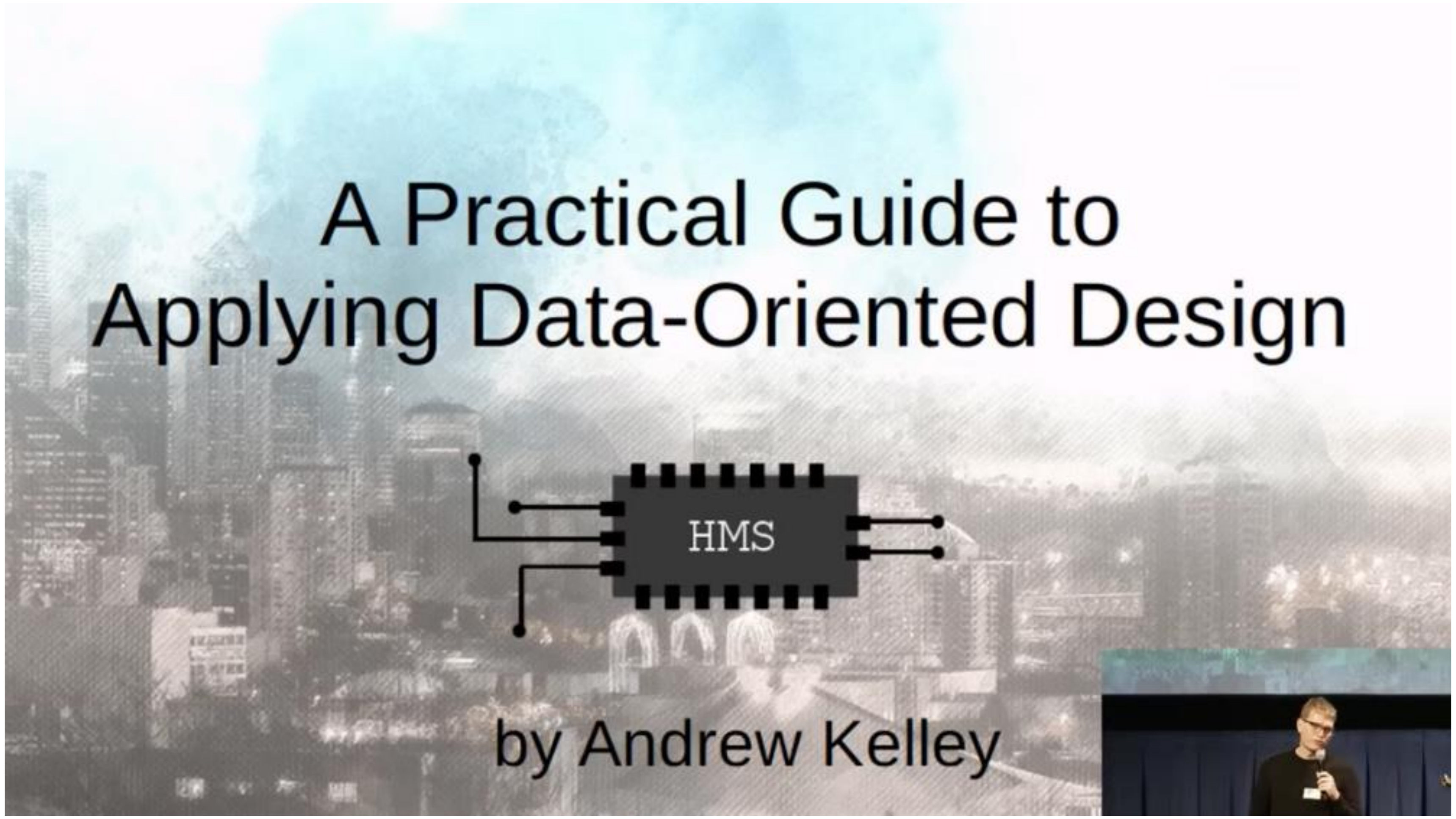
About Me

- C++ Software Developer at Jump Trading since 2014
- WG21 participant since 2016
  - C++20: `<=>`, `[...args=args]{}`, `explicit(bool)`, conditionally trivial
  - C++23: Deducing `this`, `if consteval`, bunch of `constexpr` and ranges papers
  - C++26: Reflection, packs in structured bindings, more `constexpr`

https://brevzin.github.io/

Barry

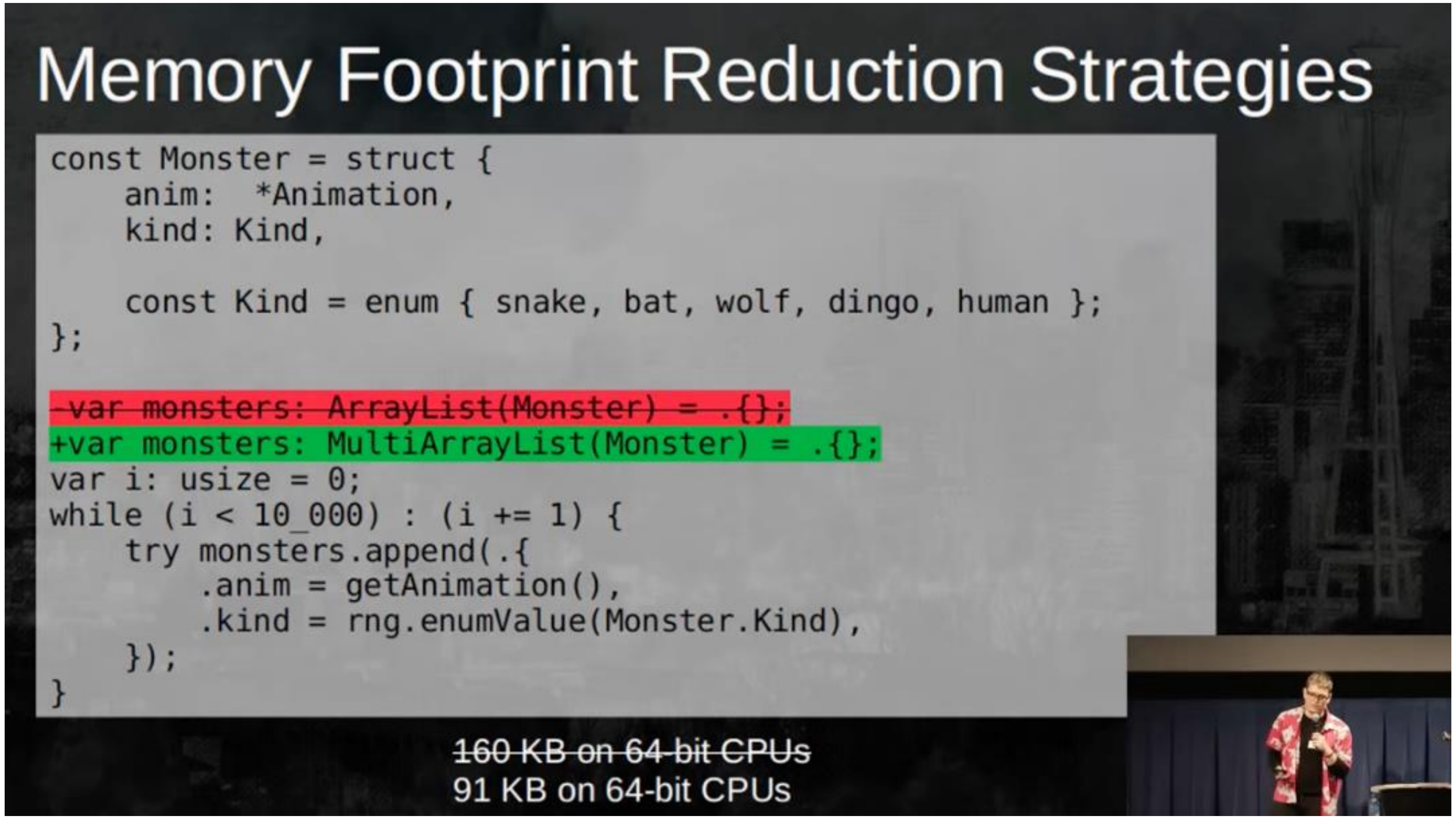
Origin Story



A Practical Guide to Applying Data-Oriented Design

by Andrew Kelley

Origin Story



Memory Footprint Reduction Strategies

```
const Monster = struct {
    anim:  *Animation,
    kind: Kind,

    const Kind = enum { snake, bat, wolf, dingo, human };
};

~~-var monsters: ArrayList(Monster) = .{};~~
+var monsters: MultiArrayList(Monster) = .{};
var i: usize = 0;
while (i < 10_000) : (i += 1) {
    try monsters.append(.{
        .anim = getAnimation(),
        .kind = rng.enumValue(Monster.Kind),
    });
}
```

~~160 KB on 64-bit CPUs~~
91 KB on 64-bit CPUs

Goals for this Talk

Implement `SoaVector<T>`

Socialize reflection techniques

Storage

Storage

```
struct Square {
    char x;
    long y;
};
```

Storage

```
struct std::vector<Square> {
    Square* data_;
    size_t size_;
    size_t capacity_;
};
```

Storage

```
struct Square {
    char x;
    long y;
};
```

Storage

```
struct Square {
    char x;
    long y;
};
```

```
struct SoaVector<Square> {
    std::vector<char> x;
    std::vector<long> y;
};
```

Storage

```
struct Square {
    char x;
    long y;
};
```

```
struct SoaVector<Square> {
    std::vector<char> x;
    std::vector<long> y;

    auto push_back(Square const& s)
        -> void
    {


    }
};
```

Storage

```
struct Square {
    char x;
    long y;
};
```

```
struct SoaVector<Square> {
    std::vector<char> x;
    std::vector<long> y;

    auto push_back(Square const& s)
        -> void
    {
        x.push_back(s.x);

    }
};
```

Storage

```
struct Square {
    char x;
    long y;
};
```

Same size()
Same capacity()

```
struct SoaVector<Square> {
    std::vector<char> x;
    std::vector<long> y;

    auto push_back(Square const& s)
        -> void
    {
        x.push_back(s.x);
        y.push_back(s.y);
    }
};
```

Storage

```
struct Square {
    char x;
    long y;
};
```

$\Longrightarrow$

```
struct SoaVector<Square> {
    std::vector<char> x;
    std::vector<long> y;
};
```

Storage

```
struct Square {
    char x;
    long y;
};
```

```
struct SoaVector<Square> {
    char* x;
    long* y;
    size_t size_;
    size_t capacity_;
};
```

Storage

```
template <class T>
class SoaVector {

};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        // ...
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        // ...
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        // ...
        std::meta::define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        // ...
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```cpp
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(
                           ^^T,
                           std::meta::access_context::unchecked()))
        {
            // ...
        }
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(
                          ^^T,
                          ctx))
        {
            // ...
        }
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            // ...
        }
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                type_of(m), {.name=identifier_of(m)});
        }
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                add_pointer(type_of(m)), {.name=identifier_of(m)});
        }
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                add_pointer(type_of(m)), {.name=identifier_of(m)});
        }
        specs.push_back(data_member_spec(^^size_t, {.name="size_"}));
        specs.push_back(data_member_spec(^^size_t, {.name="capacity_"}));
        define_aggregate(^^Storage, specs);
    }
};
```

Storage

```
template <class T>
class SoaVector {
  ① struct Storage;
  ② consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                add_pointer(type_of(m)), {.name=identifier_of(m)});
        }
        specs.push_back(data_member_spec(^^size_t, {.name="size_"}));
        specs.push_back(data_member_spec(^^size_t, {.name="capacity_"}));
  ③     define_aggregate(^^Storage, specs);
    }
  ④ Storage storage_ = {};
};
```

Storage

```
struct C {
    int size_;
};
```

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                add_pointer(type_of(m)), {.name=identifier_of(m)}));
        }
        specs.push_back(data_member_spec(^^size_t, {.name="size_"}));
        specs.push_back(data_member_spec(^^size_t, {.name="capacity_"}));
        define_aggregate(^^Storage, specs);
    }
    Storage storage_ = {};
};
```

Storage

```cpp
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                add_pointer(type_of(m)), {.name=identifier_of(m)});
        }
        define_aggregate(^^Storage, specs);
    }
    Storage storage_ = {};
};
```

Storage

```
template <class T>
class SoaVector {
    struct Storage;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                add_pointer(type_of(m)), {.name=identifier_of(m)});
        }
        define_aggregate(^^Storage, specs);
    }
    Storage storage_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Storage

```
template <class T>
class SoaVector {
    struct Pointers;
    consteval {
        std::vector<std::meta::info> specs;
        for (auto m : nonstatic_data_members_of(^^T, ctx)) {
            specs.push_back(data_member_spec(
                add_pointer(type_of(m)), {.name=identifier_of(m)});
        }
        define_aggregate(^^Pointers, specs);
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Storage

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) {
        return nonstatic_data_members_of(^^T, ctx)
            | std::views::transform([=](auto m){
                return data_member_spec(f(type_of(m)),
                                        {.name=identifier_of(m)});
            });
    }

    struct Pointers;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Storage

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) {
        auto v = nonstatic_data_members_of(^^T, ctx)
        for (std::meta::info& m : v) {
            m = data_member_spec(f(type_of(m)), {.name=identifier_of(m)});
        }
        return v;
    }

    struct Pointers;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Constraint

Constraint

1. De-structure values
2. Re-structure values
3. Put names to storage

$$\left.\begin{array}{l}\texttt{T\{m}_1\texttt{, m}_2\texttt{, …, m}_N\texttt{\}}\end{array}\right\} \text{T is an aggregate}$$

$\}$ T has no base classes

Constraint

```
template <class T>
concept CanSoa =
    std::is_aggregate_v<T>
    and bases_of(^^T, ctx).empty();
```

} T is an aggregate

} T has no base classes

Constraint

```
consteval auto can_soa(std::meta::info type) -> bool {
    return is_aggregate_type(type)
        and bases_of(type, ctx).empty();
}

template <class T>
concept CanSoa = can_soa(^^T);
```

Constraint

```
consteval auto can_soa(std::meta::info type) -> bool {
    return is_aggregate_type(type)
       and bases_of(type, ctx).empty();
}

template <class T>
concept CanSoa = can_soa(^^T);
```

Constraint

```cpp
namespace std {
    template <class T>
    inline constexpr bool is_aggregate_v = /* ... */;
}

consteval auto is_aggregate_type(std::meta::info type) -> bool {
    // ???
}

consteval auto can_soa(std::meta::info type) -> bool {
    return is_aggregate_type(type)
       and bases_of(type, ctx).empty();
}

template <class T>
concept CanSoa = can_soa(^^T);
```

Constraint

```
namespace std {
    template <class T>
    inline constexpr bool is_aggregate_v = /* ... */;
}

consteval auto is_aggregate_type(std::meta::info type) -> bool {
    return std::is_aggregate_v<[:type:]>;   ✗
}

consteval auto can_soa(std::meta::info type) -> bool {
    return is_aggregate_type(type)
        and bases_of(type, ctx).empty();
}

template <class T>
concept CanSoa = can_soa(^^T);
```

Constraint

```
^^std::is_aggregate_v<[:type:]>
```

```
namespace std {
    template <class T>
    inline constexpr bool is_aggregate_v = /* ... */;
}

consteval auto is_aggregate_type(std::meta::info type) -> bool {
    auto r = substitute(^^std::is_aggregate_v, {type});
}

consteval auto can_soa(std::meta::info type) -> bool {
    return is_aggregate_type(type)
      and bases_of(type, ctx).empty();
}

template <class T>
concept CanSoa = can_soa(^^T);
```

Constraint

```
namespace std {
    template <class T>
    inline constexpr bool is_aggregate_v = /* ... */;
}

consteval auto is_aggregate_type(std::meta::info type) -> bool {
  ① auto r = substitute(^^std::is_aggregate_v, {type});
  ② return extract<bool>(r);
}

consteval auto can_soa(std::meta::info type) -> bool {
    return is_aggregate_type(type)
       and bases_of(type, ctx).empty();
}

template <class T>
concept CanSoa = can_soa(^^T);
```

Constraint

```
namespace std {
    template <class T>
    inline constexpr bool is_aggregate_v = /* ... */;
}

inline constexpr auto pred = [](std::meta::info templ){
    return [=](auto... args){
        return extract<bool>(substitute(templ, {args...}));
    };         ②             ①
};

consteval auto is_aggregate_type(std::meta::info type) -> bool {
    auto r = substitute(^^std::is_aggregate_v, {type});
    return extract<bool>(r);
}
```

Constraint

```cpp
namespace std {
    template <class T>
    inline constexpr bool is_aggregate_v = /* ... */;
}

inline constexpr auto pred = [](std::meta::info templ){
    return [=](auto... args){
        return extract<bool>(substitute(templ, {args...}));
    };         (2)           (1)
};

inline constexpr auto is_aggregate_type = pred(^^std::is_aggregate_v);
```

Adding Elements

Adding Elements

```
struct SoaVector<Square> {
    struct Pointers { char* x; long* y; };
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;

    auto push_back(Square const& s) -> void {
        // ...
    }
};
```

Adding Elements

```
struct SoaVector<Square> {
    struct Pointers { char* x; long* y; };
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;

    auto push_back(Square const& s) -> void {
        if (size_ == capacity_) {
            grow(new_capacity());
        }
    }
};
```

Adding Elements

```
struct SoaVector<Square> {
    struct Pointers { char* x; long* y; };
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;

    auto push_back(Square const& s) -> void {
        if (size_ == capacity_) {
            grow(new_capacity());
        }

        ::new (pointers_.x + size_) char(s.x);
    }
};
```

Adding Elements

```
struct SoaVector<Square> {
    struct Pointers { char* x; long* y; };
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;

    auto push_back(Square const& s) -> void {
        if (size_ == capacity_) {
            grow(new_capacity());
        }

        ::new (pointers_.x + size_) char(s.x);
        ::new (pointers_.y + size_) long(s.y);
    }
};
```

Adding Elements

```
struct SoaVector<Square> {
    struct Pointers { char* x; long* y; };
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;

    auto push_back(Square const& s) -> void {
        if (size_ == capacity_) {
            grow(new_capacity());
        }

        ::new (pointers_.x + size_) char(s.x);
        ::new (pointers_.y + size_) long(s.y);
        ++size_;
    }
};
```

members of `Pointers`

members of `Square`

Adding Elements

```cpp
struct SoaVector<Square> {
    struct Pointers { char* x; long* y; };
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;

    auto push_back(Square const& s) -> void {
        if (size_ == capacity_) {
            grow(new_capacity());
        }

        std::construct_at(pointers_.x + size_, s.x);
        std::construct_at(pointers_.y + size_, s.y);
        ++size_;
    }
};
```

members of `Pointers`

members of `Square`

Adding Elements

```cpp
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    // copy each member into its own slot

    ++size_;
}
```

Adding Elements: Option #1

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    // copy each member into its own slot

    ++size_;
}
```

Adding Elements: Option #1

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    auto& [...ptrs] = pointers_;
    auto& [...mems] = value;

    ++size_;
}
```

Packs in structured bindings (P1061)

Adding Elements: Option #1

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    auto& [...ptrs] = pointers_;
    auto& [...mems] = value;
    (std::construct_at(ptrs + size_, mems), ...);

    ++size_;
}
```

Packs in structured bindings (P1061)

Adding Elements: Option #2

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    auto& [...ptrs] = pointers_;
    auto& [...mems] = value;
    template for (constexpr int I : /* ... */) {
        std::construct_at(ptrs...[I] + size_, mems...[I]);
    }

    ++size_;
}
```

Expansion Statement (P1306)

Pack indexing (P2662)

Adding Elements: Option #2

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    auto& [...ptrs] = pointers_;
    auto& [...mems] = value;
    template for (constexpr int I : std::views::iota(
                                        0zu, sizeof...(ptrs))) {
        std::construct_at(ptrs...[I] + size_, mems...[I]);
    }

    ++size_;
}
```

Expansion Statement (P1306)

Pack indexing (P2662)

Adding Elements: Option #2

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    auto& [...ptrs] = pointers_;
    auto& [...mems] = value;
    template for (constexpr int I : std::views::indices(
                                        sizeof...(ptrs))) {
        std::construct_at(ptrs...[I] + size_, mems...[I]);
    }

    ++size_;
}
```

(P3060)

Expansion Statement (P1306)

Pack indexing (P2662)

Adding Elements: Option #3

```cpp
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    template for (auto [ptr, mem] : tuple_zip(pointers_,
                                              value)) {
        std::construct_at(ptr + size_, mem);
    }

    ++size_;
}
```

```cpp
template <class T, class U>
constexpr auto tuple_zip(T&& t, U&& u) {
    auto& [...mt] = t;
    auto& [...mu] = u;
    return std::make_tuple(std::tie(mt, mu)...);
}
```

Adding Elements: Option #4

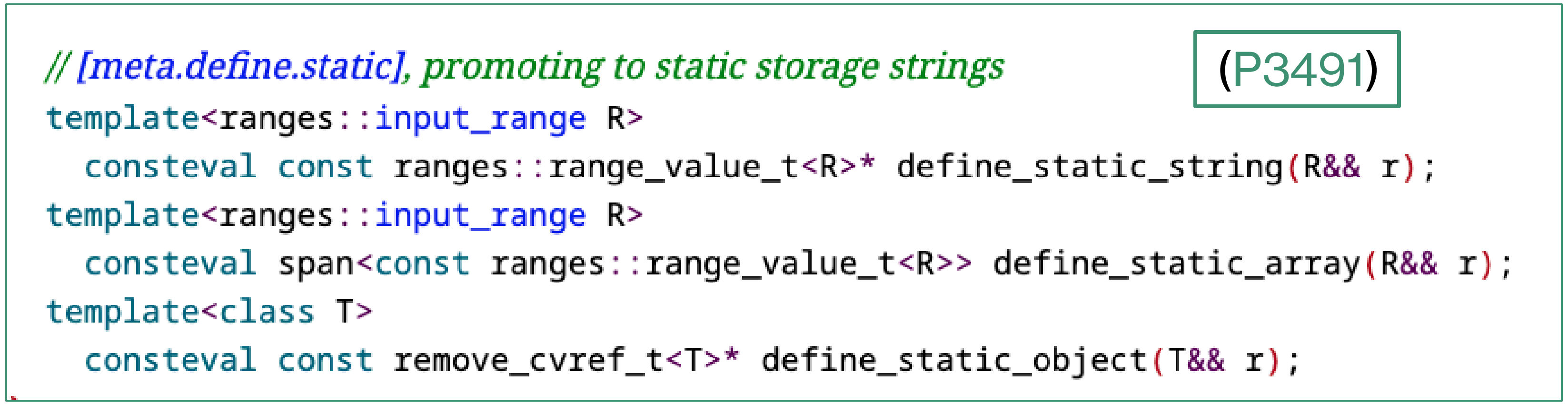
constexpr
structured bindings
(P2686)

Non-transient
constexpr
allocation
(PXXXX)

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    template for (constexpr auto [ptr, mem] : std::views::zip(
            nonstatic_data_members_of(^^Pointers, ctx),
            nonstatic_data_members_of(^^T, ctx))) {
        std::construct_at(pointers_.[:ptr:] + size_, value.[:mem:]);
    }

    ++size_;
}
```

Adding Elements: Option #4

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    template for (constexpr auto [ptr, mem] : std::views::zip(
            define_static_array(nonstatic_data_members_of(^^Pointers, ctx
            define_static_array(nonstatic_data_members_of(^^T, ctx)))) {
        std::construct_at(pointers_.[:ptr:] + size_, value.[:mem:]);
    }

    ++size_
}
```



(P3491)

```
// [meta.define.static], promoting to static storage strings
template<ranges::input_range R>
  consteval const ranges::range_value_t<R>* define_static_string(R&& r);
template<ranges::input_range R>
  consteval span<const ranges::range_value_t<R>> define_static_array(R&& r);
template<class T>
  consteval const remove_cvref_t<T>* define_static_object(T&& r);
```

Implementing `std::define_static_array`

```
template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {

}
```

Implementing `std::define_static_array`

```
template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
    // 1. produce the array (of type T[N])
    meta::info array = meta::reflect_constant_array(r);

}
```

Implementing `std::define_static_array`

```
template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
    // 1. produce the array (of type T[N])
    meta::info array = meta::reflect_constant_array(r);

    // 2. extract the contents
    return span(extract<T const*>(array), extent(type_of(array)));
}
```

Implementing `std::define_static_array`

```cpp
namespace meta {
  template <ranges::input_range R, class T = ranges::range_value_t<R>>
  consteval auto reflect_constant_array(R&& r) -> info {
  }
}

template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
    // 1. produce the array (of type T[N])
    meta::info array = meta::reflect_constant_array(r);

    // 2. extract the contents
    return span(extract<T const*>(array), extent(type_of(array)));
}
```

Implementing `std::define_static_array`

```
namespace meta {
  template <class T, T... Vs>
  inline constexpr T the_array[]{Vs...};

  template <ranges::input_range R, class T = ranges::range_value_t<R>>
  consteval auto reflect_constant_array(R&& r) -> info {
  }
}

template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
    // 1. produce the array (of type T[N])
    meta::info array = meta::reflect_constant_array(r);

    // 2. extract the contents
    return span(extract<T const*>(array), extent(type_of(array)));
}
```

Implementing `std::define_static_array`

```
namespace meta {
  template <class T, T... Vs>
  inline constexpr T the_array[]{Vs...};

  template <ranges::input_range R, class T = ranges::range_value_t<R>>
  consteval auto reflect_constant_array(R&& r) -> info {
    auto args = vector<info>{^^T};




  }
}

template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
     // 1. produce the array (of type T[N])
     meta::info array = meta::reflect_constant_array(r);
     // 2. extract the contents
     return span(extract<T const*>(array), extent(type_of(array)));
}
```

Implementing std::define_static_array

```
namespace meta {
  template <class T, T... Vs>
  inline constexpr T the_array[]{Vs...};

  template <ranges::input_range R, class T = ranges::range_value_t<R>>
  consteval auto reflect_constant_array(R&& r) -> info {
    auto args = vector<info>{^^T};
    for (auto&& elem : r) {
      args.push_back(reflect_constant(elem));
    }

  }
}

template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
     // 1. produce the array (of type T[N])
     meta::info array = meta::reflect_constant_array(r);
     // 2. extract the contents
     return span(extract<T const*>(array), extent(type_of(array)));
}
```

Implementing `std::define_static_array`

```
namespace meta {
  template <class T, T... Vs>
  inline constexpr T the_array[]{Vs...};

  template <ranges::input_range R, class T = ranges::range_value_t<R>>
  consteval auto reflect_constant_array(R&& r) -> info {
    auto args = vector<info>{^^T};
    for (auto&& elem : r) {
      args.push_back(reflect_constant(elem));
    }
①   return substitute(^^the_array, args);
  }
}

template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
     // 1. produce the array (of type T[N])
     meta::info array = meta::reflect_constant_array(r);
     // 2. extract the contents
②    return span(extract<T const*>(array), extent(type_of(array)));
}
```

Implementing `std::define_static_array`

```
namespace meta {
  template <class T, T... Vs>
  inline constexpr T the_array[]{Vs...};

  template <ranges::input_range R, class T = ranges::range_value_t<R>>
  consteval auto reflect_constant_array(R&& r) -> info {
    auto args = vector<info>{^^T};
    for (auto&& elem : r) {
      args.push_back(reflect_constant(elem));
    }
①   return substitute(^^the_array, args);
  }
}

template <ranges::input_range R, class T = ranges::range_value_t<R>>
consteval auto define_static_array(R&& r) -> span<T const> {
     // 1. produce the array (of type T[N])
     meta::info array = meta::reflect_constant_array(r);
     // 2. extract the contents
②    return span(extract<T const*>(array), extent(type_of(array)));
}
```

Adding Elements: Option #4

```
template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    template for (constexpr auto [ptr, mem] : std::views::zip(
            define_static_array(nonstatic_data_members_of(^^Pointers, ctx
            define_static_array(nonstatic_data_members_of(^^T, ctx)))) {
        std::construct_at(pointers_.[:ptr:] + size_, value.[:mem:]);
    }

    ++size_;
}
```

Adding Elements: Option #4

```cpp
static constexpr auto ptr_members =
    define_static_array(nonstatic_data_members_of(^^Pointers, ctx));
static constexpr auto members =
    define_static_array(nonstatic_data_members_of(^^T, ctx));

template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    template for (constexpr auto [ptr, mem] :
                  std::views::zip(ptr_members, members)) {
        std::construct_at(pointers_.[:ptr:] + size_, value.[:mem:]);
    }

    ++size_;
}
```

Adding Elements: Option #4

```
static constexpr auto ptr_members = nsdms_of(^^Pointers);
static constexpr auto members     = nsdms_of(^^T);

template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    template for (constexpr auto [ptr, mem] :
                  std::views::zip(ptr_members, members)) {
        std::construct_at(pointers_.[:ptr:] + size_, value.[:mem:]);
    }

    ++size_;
}
```

Adding Elements: Option #5

```
static constexpr auto ptr_members = nsdms_of(^^Pointers);
static constexpr auto members     = nsdms_of(^^T);

template <class T>
auto SoaVector<T>::push_back(T const& value) -> void {
    if (size_ == capacity_) { grow(new_capacity()); }

    template for (constexpr int I :
                  std::views::indices(members.size())) {
        std::construct_at(pointers_.[: ptr_members[I] :] + size_,
                          value.[: members[I] :]);
    }

    ++size_;
}
```

Adding Elements: growing

```
template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
}
```

Adding Elements: growing

```
template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_copy_n(src, size_, dst);
    std::destroy(src, src + size_);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_copy_n(src, size_, dst);
    std::destroy(src, src + size_);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_copy_n(src, size_, dst);
    std::destroy(src, src + size_);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_copy_n(src, size_, dst);
    std::destroy(src, src + size_);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_copy_n(src, size_, dst);
    std::destroy(src, src + size_);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_move_n(src, size_, dst);
    std::destroy(src, src + size_);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_move_n(src, size_, dst);
    std::destroy(src, src + size_);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Adding Elements: growing

```
template <class T>
template <class U>
auto SoaVector<T>::grow(U* src, size_t new_capacity) -> void {
    auto alloc = std::allocator<U>();
    U* dst = alloc.allocate(new_capacity);
    std::uninitialized_relocate_n(src, size_, dst);
    alloc.deallocate(src, capacity_);
    return dst;
}

template <class T>
auto SoaVector<T>::grow(size_t new_capacity) -> void {
    template for (auto*& p : pointers_) {
        p = grow(p, new_capacity);
    }
    capacity_ = new_capacity;
}
```

Printing

Printing

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Pointers;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Printing

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Pointers;
    struct Spans;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Printing

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Pointers;
    struct Spans;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
        define_aggregate(^^Spans, transformed([](std::meta::info ty){
            return substitute(^^std::span, {add_const(ty)});
        }));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

```
struct Spans {
    std::span<char const> x;
    std::span<long const> y;
};
```

Printing

```cpp
template <class T>
class SoaVector {
public:
    auto spans() const -> Spans {
        // ...
    }
};
```

Printing

```
template <class T>
class SoaVector {
public:
    auto spans() const -> Spans {
        // ...
    }
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    std::println("x={}", v.spans().x);
    std::println("y={}", v.spans().y);
}
```

Printing

```
template <class T>
class SoaVector {
public:
    auto spans() const -> Spans {
        // ...
    }
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    std::println("x={}", v.spans().x); // ['e', 'c']
    std::println("y={}", v.spans().y); // [4, 6]
}
```

Printing

```
template <class T>
class SoaVector {
public:
    auto spans() const -> Spans {
        auto& [...ptrs] = pointers_;
        return Spans{std::span(ptrs, size_)...};
    }
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    std::println("x={}", v.spans().x); // ['e', 'c']
    std::println("y={}", v.spans().y); // [4, 6]
}
```

Printing

```
template <class T>
class SoaVector {
public:
    auto spans() const -> Spans {
        auto& [...ptrs] = pointers_;
        return Spans{std::span(ptrs, size_)...};
    }
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    std::println("x={}", v.spans().x); // ['e', 'c']
    std::println("y={}", v.spans().y); // [4, 6]
    std::println("v[0]={}", v[0]);     // Square{.x='e', .y=4}
}
```

Printing

```
template <class T>
class SoaVector {
public:
    auto spans() const -> Spans {
        auto& [...ptrs] = pointers_;
        return Spans{std::span(ptrs, size_)...};
    }

    auto operator[](size_t idx) const -> T {
        auto& [...ptrs] = pointers_;
        return T{ptrs[idx]...};
    }
};
```

Formatting

```
struct Square {
    char x;
    long y;
};

template <>
struct std::formatter<Square> {
    // ...
};
```

Formatting

```cpp
struct Square {
    char x;
    long y;
};

template <>
struct std::formatter<Square> {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }
};
```

Formatting

```cpp
struct Square {
    char x;
    long y;
};

template <>
struct std::formatter<Square> {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(Square const& s, auto& ctx) const {
        return std::format_to(ctx.out(),
            "Square{{.x={:?}, .y={}}}",
            s.x,
            s.y
        );
    }
};
```

Formatting

```
#[derive(Debug)] struct Square {
    char x;
    long y;
};

template <>
struct std::formatter<Square> {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(Square const& s, auto& ctx) const {
        return std::format_to(ctx.out(),
            "Square{{.x={:?}, .y={}}}",
            s.x,
            s.y
        );
    }
};
```

Formatting

```cpp
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

template <>
struct std::formatter<Square> {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(Square const& s, auto& ctx) const {
        return std::format_to(ctx.out(),
            "Square{{.x={:?}, .y={}}}",
            s.x,
            s.y
        );
    }
};
```

Formatting

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>)) {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(Square const& s, auto& ctx) const {
        return std::format_to(ctx.out(),
            "Square{{.x={:?}, .y={}}}",
            s.x,
            s.y
        );
    }
};
```

Formatting

```cpp
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>)) {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(T const& value, auto& ctx) const {
        // do the right thing
    }
};
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {






}
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    for (std::meta::info a : annotations_of(r)) {



    }

}
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    for (std::meta::info a : annotations_of(r)) {
        if ($\boxed{\texttt{type\_of(a) == ^^T}}$ and extract<T>(a) == value) {
            return true;
        }
    }
    return false;
}
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    for (std::meta::info a : annotations_of_with_type(r, ^^T)) {
        if (extract<T>(a) == value) {
            return true;
        }
    }
    return false;
}
```

Formatting

```cpp
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    for (std::meta::info a : annotations_of_with_type(r, ^^T)) {
        if (constant_of(a) == std::meta::reflect_constant(value)) {
            return true;
        }
    }
    return false;
}
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    auto expected = std::meta::reflect_constant(value);
    for (std::meta::info a : annotations_of_with_type(r, ^^T)) {
        if (constant_of(a) == expected) {
            return true;
        }
    }
    return false;
}
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    auto expected = std::meta::reflect_constant(value);
    return std::ranges::any_of(
        annotations_of_with_type(r, ^^T),
        [=](std::meta::info a){
            return constant_of(a) == expected;
        }
    );
}
```

Formatting

```
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    auto expected = std::meta::reflect_constant(value);
    return std::ranges::contains(
        annotations_of_with_type(r, ^^T),
        expected,
        std::meta::constant_of
    );
}
```

Formatting

```cpp
template <typename T> struct Derive { };
template <typename T> inline constexpr Derive<T> derive;
struct Debug { };

template <typename T>
consteval auto has_annotation(std::meta::info r, T const& value) -> bool {
    return std::ranges::contains(
        annotations_of_with_type(r, ^^T),
        std::meta::reflect_constant(value),
        std::meta::constant_of
    );
}
```

Formatting

```cpp
template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>)) {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(T const& value, auto& ctx) const {
        // do the right thing
    }
};
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    // do the right thing
}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    *out++ = '}';
    return out;
}
```

```
// Square{}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));

    template for (constexpr auto base : define_static_array(
                                            bases_of(^^T, ctx))) {
        out = std::format_to(out, "{}", value.[:base:]);
    }


    *out++ = '}';
    return out;
}
```

```
// Square{}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));

    template for (constexpr auto base : bases_of2(^^T)) {
        out = std::format_to(out, "{}", value.[:base:]);
    }


    *out++ = '}';
    return out;
}
```

```
// Square{}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));

    auto delim = [first=true, &out]() mutable {
        if (not first) { *out++ = ','; *out++ = ' '; }
        first = false;
    };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }


    *out++ = '}';
    return out;
}
```

```
// Square{}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }


    *out++ = '}';
    return out;
}
```

```
// Square{}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : define_static_array(
                                        nonstatic_data_members_of(^^T, ctx))) {
        delim();
        out = std::format_to(out, "{}", value.[:nsdm:]);
    }

    *out++ = '}';
    return out;
}
```

```
// Square{}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : define_static_array(
                                        nonstatic_data_members_of(^^T, ctx))) {
        delim();
        out = std::format_to(out, "{}", value.[:nsdm:]);
    }

    *out++ = '}';
    return out;
}
```

```
// Square{e, 4}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : nsdms_of(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:nsdm:]);
    }

    *out++ = '}';
    return out;
}
```

```
// Square{e, 4}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : nsdms_of(^^T)) {
        delim();
        out = std::format_to(out, ".{}={}", identifier_of(nsdm), value.[:nsdm:]);
    }

    *out++ = '}';
    return out;
}
```

```
// Square{e, 4}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : nsdms_of(^^T)) {
        delim();
        out = std::format_to(out, ".{}={}", identifier_of(nsdm), value.[:nsdm:]);
    }

    *out++ = '}';
    return out;
}
```

```
// Square{.x=e, .y=4}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : nsdms_of(^^T)) {
        delim();
        out = std::format_to(out, ".{}={:?}", identifier_of(nsdm), value.[:nsdm:]);
    }

    *out++ = '}';
    return out;
}
```

```
// Square{.x='e', .y=4}
```

Formatting

```cpp
template <class T> struct FmtDebug { T& value; };

template <class T>
struct std::formatter<FmtDebug<T>> {
    std::formatter<std::remove_cvref_t<T>> underlying;

    constexpr auto parse(auto& ctx) {
        auto init = std::string_view(ctx.begin(), ctx.end()).substr(0, 2);
        if (init != "?}") { throw std::format_error("context must be ?"); }

        if constexpr (not requires { underlying.set_debug_format(); }) {
            ctx.advance_to(ctx.begin() + 1); // skip the ?
        }

        return underlying.parse(ctx);
    }

    auto format(FmtDebug<T> f, auto& ctx) const {
        return underlying.format(f.value, ctx);
    }
};
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : nsdms_of(^^T)) {
        delim();
        out = std::format_to(out, ".{}={:?}", identifier_of(nsdm),
                             FmtDebug{value.[:nsdm:]});
    }

    *out++ = '}';
    return out;
}
```

```
// Square{.x='e', .y=4}
```

Formatting

```
template <class T> requires (has_annotation(^^T, derive<Debug>))
auto std::formatter<T>::format(T const& value, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", display_string_of(^^T));
    auto delim = [first=true, &out]() mutable {/* ... */ };

    template for (constexpr auto base : bases_of2(^^T)) {
        delim();
        out = std::format_to(out, "{}", value.[:base:]);
    }

    template for (constexpr auto nsdm : nsdms_of(^^T)) {
        delim();
        out = std::format_to(out, ".{}={:?}", identifier_of(nsdm),
                                               FmtDebug{value.[:nsdm:]});
    }

    *out++ = '}';
    return out;
}
```

```
// Square{.x='e', .y=4}
```

Formatting

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    std::println("{}", v.spans().x); // ['e', 'c’]
    std::println("{}", v.spans().y); // [4, 6]
    std::println("{}", v[0]);        // Square{.x='e', .y=4}
}
```

Proxy References

Proxy References

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    std::println("{}", v.spans().x); // ['e', 'c’]
    std::println("{}", v.spans().y); // [4, 6]
    std::println("{}", v[0]);        // Square{.x='e', .y=4}
}
```

Proxy References

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};

    std::println("{}", v.spans().x); // ['f', 'c’]
    std::println("{}", v.spans().y); // [3, 6]
    std::println("{}", v[0]);        // Square{.x='f', .y=3}
}
```

Proxy References

```
template <class T>
class SoaVector {
public:
    auto operator[](size_t idx) -> T& {
    }

    auto operator[](size_t idx) const -> T;
};
```

Proxy References

```
template <class T>
class SoaVector {
public:
    auto operator[](size_t idx) -> T& {
        auto& [...ptrs] = pointers_;






    }

    auto operator[](size_t idx) const -> T;
};
```

Proxy References

```
template <class T>
class SoaVector {
public:
    auto operator[](size_t idx) -> T& {
        auto& [...ptrs] = pointers_;
        auto value = T{ptrs[idx]...};





    }

    auto operator[](size_t idx) const -> T;
};
```

Proxy References

```
template <class T>
class SoaVector {
public:
    auto operator[](size_t idx) -> T& {
        auto& [...ptrs] = pointers_;
        auto value = T{ptrs[idx]...};

        yield value; // give control back to caller



    }

    auto operator[](size_t idx) const -> T;
};
```

Proxy References

```
template <class T>
class SoaVector {
public:
    auto operator[](size_t idx) -> T& {
        auto& [...ptrs] = pointers_;
        auto value = T{ptrs[idx]...};

        yield value; // give control back to caller   ✗

        auto& [...elems] = value;
        (ptrs[idx] = elems, ...);
    }

    auto operator[](size_t idx) const -> T;
};
```

Proxy References

```cpp
template <class T>
class SoaVector {
public:
    auto operator[](size_t idx) -> Proxy {
        auto& [...ptrs] = pointers_;
        return Proxy{ptrs[idx]...};
    }

    auto operator[](size_t idx) const -> T;
};
```

Proxy References

```cpp
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Pointers;
    struct Spans;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
        define_aggregate(^^Spans, transformed([](std::meta::info ty){
            return substitute(^^std::span, {add_const(ty)});
        }));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Proxy References

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Pointers;
    struct Spans;
    struct Proxy;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
        define_aggregate(^^Spans, transformed([](std::meta::info ty){
            return substitute(^^std::span, {add_const(ty)});
        }));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Proxy References

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Pointers;
    struct Spans;
    struct Proxy;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
        define_aggregate(^^Spans, transformed([](std::meta::info ty){
            return substitute(^^std::span, {add_const(ty)});
        }));
        define_aggregate(^^Proxy, transformed(std::meta::add_lvalue_reference));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Proxy References

```
struct Square {
    char x;
    long y;
};

struct Proxy {
    char& x;
    long& y;
};
```

Proxy References

```cpp
struct Square {
    char x;
    long y;
};

struct Proxy {
    char& x;
    long& y;

    auto operator=(Square const& v) -> void {
        x = v.x;
        y = v.y;
    }
};
```

Proxy References

```cpp
struct Square {
    char x;
    long y;
};

struct Proxy {
    char& x;
    long& y;

    auto operator=(Square const& v) -> void {
        x = v.x;
        y = v.y;
    }

    operator Square() const {
        return Square{x, y};
    }
};
```

Proxy References

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Pointers;
    struct Spans;
    struct Proxy;
    consteval {
        define_aggregate(^^Pointers, transformed(std::meta::add_pointer));
        define_aggregate(^^Spans, transformed([](std::meta::info ty){
            return substitute(^^std::span, {add_const(ty)});
        }));
        define_aggregate(^^Proxy, transformed(std::meta::add_lvalue_reference));
    }
    Pointers pointers_ = {};
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Proxy References

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct Proxy;
    consteval {
        define_aggregate(^^Proxy, transformed(std::meta::add_lvalue_reference));
    }
};
```

Proxy References

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct ProxyBase;
    consteval {
        define_aggregate(^^ProxyBase,
                         transformed(std::meta::add_lvalue_reference));
    }
};
```

Proxy References

```
template <class T>
class SoaVector {
    static consteval auto transformed(auto f) { /* ... */ }

    struct ProxyBase;
    consteval {
        define_aggregate(^^ProxyBase,
                         transformed(std::meta::add_lvalue_reference));
    }

    struct Proxy : ProxyBase {
        auto operator=(T const& value) -> void;

        operator T() const;
    };

};
```

Proxy References

```
struct Proxy : ProxyBase {
    auto operator=(T const& value) -> void;

    operator T() const;
};
```

Proxy References

```
struct Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [...refs] = (ProxyBase&)*this;
        auto& [...mems] = value;
        ((refs=mems), ...);
    }

    operator T() const;
};
```

Proxy References

```
struct Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [[...refs]] = *this;
        auto& [...mems]   = value;
        ((refs=mems), ...);
    }

    operator T() const;
};
```

Proxy References

```
struct Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [...refs] = *this;
        auto& [...mems] = value;
        ((refs=mems), ...);
    }

    operator T() const;
};
```

Proxy References

```
struct Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [...refs] = *this;
        auto& [...mems] = value;
        ((refs=mems), ...);
    }

    operator T() const {
        auto& [...refs] = *this;
        return T{refs...};
    }
};
```

Proxy References

```
struct [[=derive<Debug>]] Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [...refs] = *this;
        auto& [...mems] = value;
        ((refs=mems), ...);
    }

    operator T() const {
        auto& [...refs] = *this;
        return T{refs...};
    }
};
```

Proxy References

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};

    std::println("{}", v.spans().x); // ['f', 'c’]
    std::println("{}", v.spans().y); // [3, 6]
    std::println("{}", v[0]);         // Square{.x='f', .y=3}
}
```

Proxy References

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};

    std::println("{}", v.spans().x); // ['f', 'c']
    std::println("{}", v.spans().y); // [3, 6]
    std::println("{}", v[0]);         // Proxy{ProxyBase{.x='f', .y=3}}
}
```

Formatting Proxy References

```cpp
struct [[=derive<Debug>]] Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [...refs] = *this;
        auto& [...mems] = value;
        ((refs=mems), ...);
    }

    operator T() const {
        auto& [...refs] = *this;
        return T{refs...};
    }
};
```

Formatting Proxy References

```
struct [[=derive<Debug>, =format_as(^^T)]] Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [...refs] = *this;
        auto& [...mems] = value;
        ((refs=mems), ...);
    }

    operator T() const {
        auto& [...refs] = *this;
        return T{refs...};
    }
};
```

Formatting Proxy References

```
struct format_as { std::meta::info type; };

struct [[=derive<Debug>, =format_as(^^T)]] Proxy : ProxyBase {
    auto operator=(T const& value) -> void {
        auto& [...refs] = *this;
        auto& [...mems] = value;
        ((refs=mems), ...);
    }

    operator T() const {
        auto& [...refs] = *this;
        return T{refs...};
    }
};
```

Formatting Proxy References

```
template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>)) {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(T const& value, auto& ctx) const {
        // do the right thing
    }
};
```

Formatting Proxy References

```cpp
template <class T>
struct derive_formatter {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }

    auto format(T const& value, auto& ctx) const {
        // do the right thing
    }
};

template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>))
    : derive_formatter<T>
{ };
```

Formatting Proxy References

```
template <class T>
struct derive_formatter {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }
    auto format(T const& value, auto& ctx) const;
};

consteval auto format_as_type(std::meta::info ty) -> std::meta::info {
    auto as = annotations_of_with_type(ty, ^^format_as);
    return as.empty() ? ty : extract<format_as>(as[0]).type;
}

template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>))
    : derive_formatter<T>
{ };
```

Formatting Proxy References

```
template <class T>
struct derive_formatter {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }
    auto format(T const& value, auto& ctx) const;
};

consteval auto format_as_type(std::meta::info ty) -> std::meta::info {
    auto as = annotations_of_with_type(ty, ^^format_as);
    return as.empty() ? ty : extract<format_as>(as[0]).type;
}

template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>))
    : derive_formatter<typename [: format_as_type(^^T) :]>
{ };
```

Formatting Proxy References

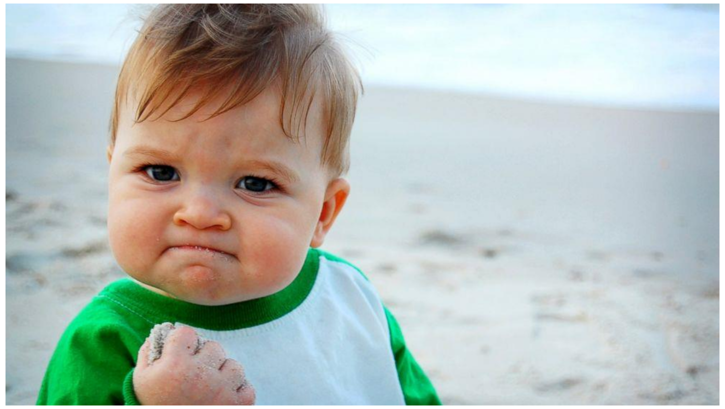
```
template <class T>
struct derive_formatter {
    constexpr auto parse(auto& ctx) { return ctx.begin(); }
    auto format(T const& value, auto& ctx) const;
};

consteval auto format_as_type(std::meta::info ty) -> std::meta::info {
    auto as = annotations_of_with_type(ty, ^^format_as);
    return as.empty() ? ty : extract<format_as>(as[0]).type;
}

template <class T>
struct std::formatter<T> requires (has_annotation(^^T, derive<Debug>))
    : derive_formatter<typename [: format_as_type(^^T) :]>
{ };

template <> struct std::formatter<Square> : derive_formatter<Square> { };
template <> struct std::formatter<Proxy> : derive_formatter<Square> { };
```

Proxy References

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};

    std::println("{}", v.spans().x); // ['f', 'c’]
    std::println("{}", v.spans().y); // [3, 6]
    std::println("{}", v[0]);        // Proxy{ProxyBase{.x='f', .y=3}}
}
```

Proxy References

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};

    std::println("{}", v.spans().x); // ['f', 'c’]
    std::println("{}", v.spans().y); // [3, 6]
    std::println("{}", v[0]);        // Square{.x='f', .y=3}
}
```



Practical Reflection

Practical Reflection

Implemented `SoaVector<T>`

`consteval` / `define_aggregate()`

Formatting an arbitrary type

`substitute() / extract<T>()`

Custom type traits and `std::define_static_array()`

Many new C++26 features

Possibilities are *endless*

Possibilities are *endless*

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};

    std::println("{}", v.spans().x); // ['f', 'c’]
    std::println("{}", v.spans().y); // [3, 6]
    std::println("{}", v[0]);        // Square{.x='f', .y=3}
}
```

Possibilities are *endless*

```
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};
    v.with([](char& x){ x += 1; });

    std::println("{}", v.spans().x); // ['g', 'd']
    std::println("{}", v.spans().y); // [3, 6]
    std::println("{}", v[0]);        // Square{.x='g', .y=3}
}
```

Possibilities are *endless*

```cpp
struct [[=derive<Debug>]] Square {
    char x;
    long y;
};

auto main() -> int {
    SoaVector<Square> v;
    v.push_back({.x='e', .y=4});
    v.push_back({.x='c', .y=6});

    v[0] = Square{.x='f', .y=3};
    v.with([](long& y){ y += 1; });

    std::println("{}", v.spans().x); // ['f', 'c']
    std::println("{}", v.spans().y); // [4, 7]
    std::println("{}", v[0]);        // Square{.x='f', .y=4}
}
```

Possibilities are *endless*

```
consteval auto selected(std::meta::info fn, std::meta::info ty)
    -> std::meta::info
{
    auto nsdms = nonstatic_data_members_of(ty, ctx);

    auto subset = parameters_of(fn)
        | std::views::transform([&](std::meta::info p){
            auto it = std::ranges::find_if(nsdms, [=](std::meta::info m){
                return identifier_of(m) == identifier_of(p);
            });
            if (it == nsdms.end()) {
                throw std::meta::exception(/* ... */);
            }
            return *it;
        });

    return std::meta::reflect_constant_array(subset);
}
```

Possibilities are *endless*

```
consteval auto selected(std::meta::info fn, std::meta::info ty)
    -> std::meta::info;

template <class T>
struct SoaVector {
    template <class F>
    auto with(F f) {
        constexpr auto [...nsdms] =
            [: selected(^^F::operator(), ^^Pointers) :];
    }
};
```

Possibilities are *endless*

```cpp
consteval auto selected(std::meta::info fn, std::meta::info ty)
    -> std::meta::info;

template <class T>
struct SoaVector {
    template <class F>
    auto with(F f) {
        constexpr auto [...nsdms] =
            [: selected(^^F::operator(), ^^Pointers) :];

        for (size_t i = 0; i < size_; ++i) {

        }
    }
};
```

Possibilities are *endless*

```
consteval auto selected(std::meta::info fn, std::meta::info ty)
    -> std::meta::info;

template <class T>
struct SoaVector {
    template <class F>
    auto with(F f) {
        constexpr auto [...nsdms] =
            [: selected(^^F::operator(), ^^Pointers) :];

        for (size_t i = 0; i < size_; ++i) {
            f(pointers_.[:nsdms:][i]...);
        }
    }
};
```