Doc. No.:	DXXXXR0
Date:	2023-4-9
Reply To:	Zhige Chen zhigec_cpp@outlook.com
Title:	`implement C++ : interface`
Audience:	

`implement C++ : interface`

Introduction

Customization support is one of the pillars on which C++ rests. Many of the language and library components of C++ are built upon customizations. But now we are heading towards C++26, and the current customization solutions are largely designed for *experts*. Consider the examples shown in the P2279R0, none of what we have today is better enough, and, more importantly, *simple* enough for those non-expert C++ programmers, particularly those who are new to C++, to learn and use. The lack of writing *proper customization* (see P2279R0) makes the generic code more write-only, more error-prone, and more difficult to promote to non-expert programmers. For instance, consider the following code from P2561R1:

```
auto strcat(int i) -> std::expected<std::string, E>
{
    int f = foo(i)??;
    //the code will roughly desugar into:
    using _Return = std::error_propagation_traits<
        std::expected<std::string, E>>;

    auto&& __f = foo(i);
    using _TraitsF = std::error_propagation_traits<
        std::remove_cvref_t<decltype(__f)>>;
    if (not _TraitsF::has_value(__f)) {
        return _Return::from_error(
            _TraitsF::extract_error(FWD(__f)));
    }
    int f = _TraitsF::extract_value(FWD(__f));
}
```

Yeah, the example is relatively easy, *especially* for those familiar with C++ generic and metaprogramming. But it's hard for non-experts to play with.

Could we write the customizations differently than template specialization? The P2547R1 provides us with a more easy-to-use customization mechanism:

```
//define the customization point...
template<typename T>
bool eq(const T& x, const T& y) customisable;

template<typename T>
requires requires(const T& x, const T& y) {
    eq(x, y);
}
bool ne(const T& x, const T& y) customisable;

bool ne(const T& x, const T& y) noexcept(eq(x, y)) default {
    return !eq(x, y);
};

template<typename T>
concept PartialEq =
requires(const T& x, const T& y) {
    eq(x, y);
    ne(x, y);
};

//...and use it...
struct Point {
    double x;
    double y;

    friend bool eq(const Point& lhs,
                   const Point& rhs) override {
        return eq(lhs.x, rhs.x) && eq(lhs.y, rhs.y);
    }
};

static_assert(PartialEq<Point>);

//...and call it
Point a{1.0, 2.0};
Point b{1.0, 3.0};

if (eq(a, b)) {
    std::puts("equal");
}
```

Nice! We are definitely making huge progress. Compared to the status quo, the code presented above is significantly cleaner and easier to understand for those who are not experts. But we still lack of proper customization, such as the atomic group of functionality or the associated types, and conciseness is missing too. So, can we have something *even* simpler than the P2547R1? Both P2279R0 and the outcome of related polls suggested that a language facility based on the C++0x Concepts may be the solution.

So here comes the interfaces:

```
//defining the interface...
interface PartialEq {
    requires auto eq(const self_t& x, const self_t& y) -> bool;
    requires auto ne(const self_t& x, const self_t& y) -> bool noexcept(noexcept(eq(x, y))) default {
        return not eq(x, y);
    }
};

//...and use it...
struct Point {
    double x;
    double y;
};

implement Point : PartialEq {
    override auto eq(const Point& lhs, const Point& rhs) -> bool {
        return (lhs.x == rhs.x) and (lhs.y == rhs.y);
    }
};

static_assert(PartialEq<Point>);

//...and call it
Point a{1.0, 2.0};
Point b{1.0, 3.0};

if (a.eq(b)) {
    std::puts("equal");
}
//or
if (PartialEq::eq(a, b)) {
    std::puts("equal");
}
//or
if (a.PartialEq::eq(b)) {
    std::puts("equal");
}
```

What This Paper *IS NOT*

1. A proposal that proposes any carefully designed and well considered language mechanism of customizations.
2. Criticisms of other customization solutions, like P2547R1.
3. Criticisms of C++20 Concepts
4. Proposal that reproposes C++0x Concepts.

What This Paper *IS*

This paper was written by a 17-year-old C++ enthusiast. Given my limited expertise and experience in C++, this paper hardly contains any carefully designed language features. However, I hope this paper, which largely copies New Circle, can show what customization solutions non-experts really want (or at least, what I want).

In general, this paper explores a new language static polymorphism mechanism following the following design principles:

1. No significant performance overhead than other solutions (especially static polymorphism).

2. Declarative and concise syntax, code expresses what they do, and it's easy to read, write and **learn** interfaces.
3. A **rich opt-in feature set** (syntax overhead) for experts.
4. Backward compatible.
5. Provide a seamless migration path to interfaces for existing customization point solutions.
6. Integrates naturally into the templates.

(As a non-native English speaker, I sincerely apologize for my poor writing skills)

Acknowledgment

- *Anyone who reviews this paper*. For spending sacred time on a newcomer's paper.
- *Bjarne Stroustrup*. Stroustrup's paper Thriving in a Crowded and Changing World: C++ 2006–20200 is the initial motivation for writing this paper.
- *Barry Revzin*. The paper P2279R0 vividly illustrated the pros and cons of different customization solutions and listed out the aspects a proper customization facility should have.
- *Arthur O'Dwyer*, for his inspiring advice on my very first ideas on the Std-Proposals forum.
- *Lewis Baker*, *Corentin Jabot*, and *Gašper Ažman*. They have put in great effort to solve the customization problem through their proposal P2547R1.
- *Sean Baxter*. See Sean Baxter's excellent project called Circle. Circle interfaces were copied shamelessly for the basic ideas and syntaxes of this paper.

interfaces

Syntax(Bikesheddable)

An `interface` is defined as follows:

```
//Syntax for interface
[template-head] [require-clauses]
[explicit] interface <interface-name> : [base-interface-list] {
    [associated-item-list]
};
```

- `explicit` : If a interface is marked explicit, then a types cannot *implicit* satisfy the interface. To satisfy the explicit interfaces, you need to write a `implement` for them.
- `base-interface-list` : If a interface has base interfaces, then the interface contains every associate items from its base interfaces.
- `associated-item-list` : See below
- `self_t` : Each interface has a *implicit* `self_t` declaration, which is a dependent type substituted when an `implement` is generated for an interface.

The `associated-item-list` can contain the following definitions:

associative function:

```
//Syntax for associative function
[template-head] [require-clauses]
[default-spec] [explicit-spec] requires [explicit] [static] [constexpr] [consteval] <return-type> <function-
    [function-default-implementation-body]
}

//or

[template-head] [require-clauses]
[default-spec] [explicit-spec] requires [explicit] [static] [constexpr] [consteval] auto <function-name>([pa
    [function-default-implementation-body]
}
```

- `explicit-spec` : If a associative function is marked with `explicit` , the implementation for the associative function in the `implement` s will not be implicitly mapped to a existing function in the type.
- `default-spec` : The default implementation availability of associative function can be controlled by `default-spec` . For example, `default(is_fundamental_v<self_t>)` will disable the default implementation of associative function when `self_t` is not a fundamental type.

Associated type:

```
//Syntax for associated type
[template-head] [require-clauses]
[default-spec] requires typename <associated-type-name> = [default-type];
```

Associated constants:

```
//Syntax for associated constants
[template-head] [require-clauses]
[default-spec] requires [const-or-constexpr] <associated-constant-type> <associated-constant-name> = [defau
```

An `implement` is defined as follows:

```
//Syntax for implement
[template-head] [require-clauses]
implement <type-name> : <interface-name-list> {
    [associative-item-implementation-list]
};
```

The `associative-item-implementation-list` can be consisted with following definitions:

`associative-function-implementation`

```
//Syntax for associative function implementation
[template-head] [require-clauses]
override [static] [constexpr] [consteval] <return-type> <function-name>([parameter-list]) [cv-qualifiers] [r
    [associative-function-implementation-body]
}

//or

[template-head] [require-clauses]
override [static] [constexpr] [consteval] auto <function-name>([parameter-list]) -> <trailing-return-type> |
    [associative-function-implementation-body]
}
```

associative-function-mapping

```
//Syntax for associative function mapping
[template-head] [require-clauses]
override using [static] [constexpr] [consteval] <return-type> <function-name>([parameter-list]) [cv-qualifie

//or

[template-head] [require-clauses]
override using [static] [constexpr] [consteval] auto <function-name>([parameter-list]) -> <trailing-return-t
```

associative-type-implementation

```
//Syntax for associative type implementation
[template-head] [require-clauses]
override using <associative-type-name> = <type-id>;
```

associative-constant-implementation

```
//Syntax for associative constant implementation
[template-head] [require-clauses]
override [const-or-constexpr] <associated-constant-type> <associated-constant-name> = <value>;
```

Detailed Examples

- Hello World

```
interface can_hello_world {
    default requires auto hello_world() -> void noexcept {
        println("Hello World!");
    }
};

class explicit_implementing {};

implement explicit_implementing : can_hello_world {
    override auto hello_world() -> void noexcept {
        println(":)");
    }
    //but we can also do this
    override auto hello_world() -> void noexcept = default;
    //and we can also leave the implementation block empty
    //this is fine because the hello_world() function has an default implementation
};

class explicit_mapping {
    auto hi() -> void noexcept {
        println("hello");
    }
};

implement explicit_mapping : can_hello_world {
    //mapping the existing function to the interface
    override using auto hello_world() -> void noexcept = hi;
};

class implicit_mapping {
    //satisfy the can_hello_world interface implicitly
    //but we can avoid this by changing the interface definition to
    //"explicit interface can_hello_world concept"
    auto hello_world() -> void noexcept {
        println("Implicit Hello World! :)");
    }
};

auto say_hello_world(can_hello_world auto const& v) {
    v.hello_world();
    can_hello_world::hello_world(v);
    v.can_hello_world::hello_world();
    //or
    using implement decltype(v) : can_hello_world;
    hello_world(v);
}
```

- Better numeric_traits

```
interface numeric_traits {
    requires constexpr static self max() const noexcept;
    requires constexpr static self min() const noexcept;
};

//implement can be templatized...
template <typename T>
    requires is_arithmetic_v<T>
implement T : numeric_traits {
    override constexpr static T max() const noexcept {
        return numeric_limits<T>::max();
    }
    override constexpr static T min() const noexcept {
        return numeric_limits<T>::min();
    }
};

class A {};
class B {
    int val_;
};

template<>
struct std::is_arithmetic<B> : std::true_type {};

//... and be specialized (including full and partial)
template <>
implement<B> T : numeric_traits {
    override constexpr static T max() const noexcept {
        return numeric_limits<int>::max();
    }
    override constexpr static T min() const noexcept {
        return numeric_limits<int>::min();
    }
};

int main() {
    using implement int, double, B : numeric_traits;
    println("{}", int::max());  //good
    println("{}", double::max()); //good
    println("{}", A::max());  //error
    println("{}", B::max());  //good
}
```

- Sugaring P2561R1

```
//the library part
namespace xstd {
    interface error_propagation_traits {
        requires typename value_type;
        requires typename error_type;

        template<typename T>
        requires typename rebind;

        requires constexpr auto has_value(this self_t const& self) -> bool;

        requires constexpr auto extract_value(this self_t const& self) -> auto&&;
        requires constexpr auto extract_error(this self_t const& self) -> auto&&;

        requires constexpr static auto from_value(auto&& v) -> self_t;
        requires constexpr static auto from_error(auto&& e) -> self_t;
    };

    template<typename T, template E>
    implement std::expected<T, E> : error_propagation_traits {
        override using value_type = T;
        override using error_type = E;

        template<typename T>
        override using rebind = std::expected<remove_cvref_t<T>, E>;

        override constexpr auto has_value(this self_t const& self) -> bool {
            return self.has_value();
        }

        override constexpr auto extract_value(this self_t const& self) -> auto&& {
            //assuming P0644R1 get adopted
            return *(<<self);
        }
        override constexpr static auto extract_error(this self_t const& self) -> auto&& {
            return (<<self).error();
        }

        override constexpr static auto from_value(auto&& v) -> std::expected<T, E> {
            return std::expected<T, E>(in_place, <<v);
        }
        override constexpr static auto from_error(auto&& e) -> std::expected<T, E> {
            return std::expected<T, E>(unexpect, <<e);
        }
    };
}

//the user part
auto strcat(int i) -> std::expected<std::string, E>
{
    int f = foo(i)??;
    //now we can desugar the code into:
    using _Return = std::expected<std::string, E>;

    auto&& __f = foo(i);
    if (not __f.std::error_propagation_traits::has_value()) {
        return _Return::std::error_propagation_traits::from_error(__f.extract_error());
    }
    int f = __f.std::error_propagation_traits::extract_value();
}
```

The `interface`

C++ has two classic strategies for customization:

1. Inheritance-based OO Design. By using inheritance, we create a **strong** relationship between data and functions. The OO Design could have runtime overheads (virtual inheritance and RTTI).
2. Overload-based Free Function Design. The correct customization function is chosen by **overload resolution** and there are hardly any relations between data and functions. Also, the complicated overload resolution procedure often creates surprising results when not used correctly.

In the P2279R0, we can see that both two strategies have their downsides. The former lacks an opt-in mechanism and runtime performance, and the latter often produces sophisticated code. Is there a solution that can avoid those drawbacks and combine their advantages? The C++0x concepts have already shown us a way to do so: *external polymorphism*. Many modern languages provide some language constructs of external polymorphism. Rust has traits, Haskell has typeclasses, and Swift has protocols. C++ also has attempted to introduce a similar language feature in C++11 (C++0x concepts), but it was finally rejected, and concepts newly introduced in C++20 have nothing to do with external polymorphism. Lack of critical ability to write proper customization has brought us million lines messy code and made customizations **significantly more expert-only**. Fortunately, there is continuous progress in this direction. The interface of Sean Baxter's New Circle is maybe the most systematic and comprehensive one among them(Great thanks to Sean Baxter again!). The design of this paper is largely borrowed from his work.

In short, an `interface` declares a set of *associative items*. An `implement` statement overrides the interface associative items for a specific type by providing implementations for them. The whole process is done at *compile time*, and the `implement` s are *external* to the type definitions. So we get a loose relationship between data and functions and the runtime performance is the same as classical static polymorphism. Consider the Hello World example:

```
//defining interface
namespace hi{
    interface hi_interface {
        default requires auto hello() -> void const noexcept {
            std::cout << "default hello!" << std::endl;
        }
    };
}

struct A { int a; };
struct B { std::string b };

//implementing types
implement A : hi::hi_interface {
    override auto hello() -> void const noexcept {
        std::cout << this->a << std::endl;
    }
};

implement B : hi::hi_interface {
    override auto hello() -> void const noexcept {
        std::cout << this->b << std::endl;
    }
};

//calling interface functions
auto main() -> int {
    A a{42};
    B b{"Hi C++ Interface!"};
    //free function call
    hi::hi_interface::hello(a);
    hi::hi_interface::hello(b);

    using interface hi_interface;

    hello(a);
    hello(b);

    //unqualified member function call
    a.hello();  //requires implements in scope
    b.hello();

    //qualified member function call
    a.hi::hi_interface::hello();    //now we can use interface functions even the implement
    b.hi::hi_interface::hello();    //statements are not in scope
}
```

We can see that most aspects mentioned by P2279R0(including those suggested by the telecon outcome, see here) are easily achieved in the above example. Looking declarative and concise though, some questions exist(in no particular order):

1. How to fit interfaces into templates
2. Should we add definition checking to interfaces?
3. What will the interactions between concepts(C++20 Concepts) and interfaces be like?
4. How to forward customizations to higher-order functions?
5. Supporting language type-erasure by interfaces?

The list is quite incomplete for such a fundamental language feature (I will add a few questions to the list later), but I think it's a good list to start our discussion on.

Interfaces and Templates

In the previous parts of this paper, we have been focusing on the ability of interfaces to provide customization points. But the interfaces also show power when combined with templates. The late-checkness of templates gives interfaces *far* more flexibility and expressiveness when compared to what Rust and Swift have. We will soon see that in language type erasure with interfaces.

With interfaces, we can now constrain the templates like what concepts do:

```
template <typename T : interface1, interface2, ... ,interfaceN>
void func() {
    //something
}

//equals to

template <typename T>
    requires interface1<T> and interface2<T> and ... and interfaceN<T>
void func() {
    //bring implements in scope
    using implement T : interface1, interface2, ... , interfaceN;
    //something
}
```

and we can extend the P1985R3 to have the interface template template parameter:

```
template <auto N> // Variable template parameter
template <template </*...*/> typename> // Type-template template-parameter
template <template </*...*/> auto> // Variable-template template-parameter
template <template </*...*/> concept> // Concept-template template-parameter
template <template </*...*/> interface> //NEW: Interface-template template-parameter

//so we can now do this(the example is taken from the P1985R3):
template <typename R, template auto T> // Primary universal template
constexpr bool is_range_of = delete;

template <typename R, template <typename> concept C> // Specialization for concepts
constexpr bool is_range_of<R, C> = C<R>;

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

template <typename R, template <typename> interface I>  //NEW: Specialization for interfaces
constexpr bool is_range_of<R, I> = I<R>;

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

// We can now constrain a range to a specific type
static_assert(range_of<std::string, char>);
// Or a concept
static_assert(range_of<std::string, std::integral>);
//NEW: Or a interface
static_assert(range_of<std::string, some_interface>);
```

and thanks to the New Circle, we can also have the interface packs!

```
//the example is taken from the New Circle:
interface IPrint {
    requires void print() const;
};

interface IScale {
    requires void scale(double x);
};

explicit interface IUnimplemented { };

// IFace is an interface pack.
// Expand IFace into the interface-list that constrains T.
template<interface <typename> ... IFace, typename T : IFace...>
void func(T& obj) {
    obj.print();
    obj.scale(1.1);
    obj.print();
}

impl double : IPrint {
    override void print() const { }
};

impl double : IScale {
    override void scale(double x) { }
};

int main() {
    double x = 100;
    func<IPrint, IScale>(x);

    // Error: double does not implement interface IUnimplemented
    func<IPrint, IScale, IUnimplemented>(x);
}

//and we can also do this:
template <interface <typename> ... IFace>
interface IGroup : IFace... {};

interface IPrint {
    requires void print() const;
};

interface IScale {
    requires void scale(double x);
};

explicit interface IUnimplemented { };

template<interface <typename> IFace, typename T : IFace>
void func(T& obj) {
    obj.print();
    obj.scale(1.1);
    obj.print();
}

impl double : IPrint {
    override void print() const { }
};
```

```
impl double : IScale {
    override void scale(double x) { }
};

int main() {
    double x = 100;
    func<IGroup<IPrint, IScale>>(x);

    // Error: double does not implement interface IUnimplemented
    func<IGroup<IPrint, IScale, IUnimplemented>>(x);
}
```

To Definition Checking orNot to Definition Checking

A major discussion of C++0x Concepts is should the 0x concepts have definition checking. The draft of 0x concepts said yes, but the consequences of bringing definition checking into a late-checked generic system led to confusion and complexity (like the introduction of `late_check{}`), or even type system violation. These problems form a major reason which led to the removal of 0x concepts.

The same discussion of definition checking comes up again in Concept Lite. But this time, the definition checking didn't enter the C++20 concepts.

So, should the interfaces have definition checking? My answer is no.

At first, it seems plausible to have the definition checking in the templates. The early-checkness of definition checking can produce diagnostics in a much more human-friendly way, and improve programmers' productivity by creating a safer generics system. But the drawbacks quickly emerge when we dive deeper into the real-world use of generics. Rust, which has early-checked generics, has an unsolved problem with providing the following template parameter types (related discussion could be found here and here):

- Variadics
- Forwarding parameters
- Non-type parameters
- Type template parameters
- Interface parameters
- Interface template parameters
- Concept parameters
- Variable template parameters
- Universal parameters

Providing great safety though, early-checking is preventing us from the most ordinary usage of C++ templates (although C++ doesn't support some of the features mentioned). To bring templates to safety, we are required to sacrifice the flexibility of C++ templates, which is one of the fundamental aspects of C++ templates. And the major drawback of late-checking, the diagnostic problem could be largely eliminated by the development of compilers and the introduction of C++ concepts (C++20 concepts).

So the answer is clear: the definition checking is too immature and unacceptable to enter C++. Future exploration may bring it back, but such exploration is outside the scope of this paper.

Interact with Concepts

The usage of interfaces and concepts is largely overlapped especially when constraining the templates. Concepts check **syntax validity** to express requirements and interfaces use **function signatures** to do that. Both approaches have their advantages and drawbacks. Good interface design should handle the interaction between interfaces and concepts to enable users to easily write an interface-concept composition. Such a composition, if designed carefully, could combine the advantages of interfaces and concepts to get great expressiveness.

In order to achieve such goals, we need to provide the following abilities:

1. The ability to treat interfaces and concepts more uniformly
2. The ability to build interfaces on existing concepts

Treating Interfaces and Concepts as Requirements

Consider the previous interface template template parameter example:

```
template <typename R, template auto T> // Primary universal template
constexpr bool is_range_of = delete;

template <typename R, template <typename> concept C> // Specialization for concepts
constexpr bool is_range_of<R, C> = C<R>;

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

template <typename R, template <typename> interface I>  //NEW: Specialization for interfaces
constexpr bool is_range_of<R, I> = I<R>;

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

We can observe that we are required to write two separate structures to handle concepts and interfaces:

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

template <typename R, template <typename> interface I>  //NEW: Specialization for interfaces
constexpr bool is_range_of<R, I> = I<R>;
```

But there is hardly any difference between them! The lack of ability to treat interfaces and concepts inevitably leads to more syntax burden. Fortunately, the universal template parameters can be easily extended to eliminate such unnecessary repetition. So we have **requirement** template template parameters(Bikesheddable):

```
template <typename R, template auto T> // Primary universal template
constexpr bool is_range_of = delete;

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

template <typename R, template <typename> requirement I>  //NEW: Specialization for requirements
constexpr bool is_range_of<R, I> = I<R>;

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

//and now we can use it with both interfaces and concepts!
static_assert(range_of<std::string, std::integral>);
static_assert(range_of<std::string, some_interface>);
```

And some new traits:

```
template <template auto>
constexpr bool is_requirement_v = false;
template <typename <template auto ...> requirement C>
constexpr bool is_requirement_v = true;

template <template auto>
constexpr bool is_concept_v = false;
template <typename <template auto ...> concept C>
constexpr bool is_concept_v = true;

template <template auto>
constexpr bool is_interface_v = false;
template <typename <template auto ...> interface I>
constexpr bool is_interface_v = true;

template <template auto X>
struct is_requirement : std::bool_constant<is_requirement_v<X>> {};
template <template auto X>
struct is_concept : std::bool_constant<is_concept_v<X>> {};
template <template auto X>
struct is_interface : std::bool_constant<is_interface_v<X>> {};
```

And a the syntax of constraining templates by interfaces should be extended:

```
template <typename T : requirement1, requirement2, ... ,requirementn>
void func() {
    //something
}
```

Build Interfaces on Existing Concepts

As the concepts are low-level than the interfaces, the necessity of building interfaces on existing concepts is obvious. The interfaces can now be more declarative and less error-prone.

We can just use require clauses to constrain implements, consider the better numeric traits example:

```
interface numeric_traits {
    requires constexpr static self max() const noexcept;
    requires constexpr static self min() const noexcept;
};

template <typename T>
concept arithmetic = is_arithmetic_v<T>;

template <typename T>
    requires arithmetic<T>
implement T : numeric_traits {
    override constexpr static T max() const noexcept {
        return numeric_limits<T>::max();
    }
    override constexpr static T min() const noexcept {
        return numeric_limits<T>::min();
    }
};
```

With interfaces on concepts, we can avoid accidentally performing an implicit implementation of the interface `numeric_traits` for some non-arithmetic types.

We can also use `require`s in the definition of interfaces:

```
interface numeric_traits {
    requires arithmetic<self_t>;
    requires constexpr static self max() const noexcept;
    requires constexpr static self min() const noexcept;
};
```

Now every type that tries to implement the interface `numeric_traits` must meet the requirements of the concept `arithmetic`.

Forwarding Customizations

Similar to what P2547R1 does (thanks to the authors of P2547R1 again!), requiring an associative function in an interface creates an *associative function object (AFO)*. An AFO is a `constexpr` object which has an unspecified, implementation-generated, default constructible trivial type with no data members. A call expression on an AFO will perform lookup and overload resolutions specific to associative functions.

Consider the following example:

```
interface comparable {
    requires static bool operator<(const self_t& lhs, const self_t& rhs) const noexcept;
    requires static bool operator>(const self_t& lhs, const self_t& rhs) const noexcept;
};

template<typename It, typename F>
auto find_extreme(It begin, It end, F f) {
  assert(begin != end);

  auto x = *begin++;
  while(begin != end)
    x = f(x, *begin++);

  return x;
}

struct Int { int n; }

implement Int : comparable {
    override static bool operator<(const self_t& lhs, const self_t& rhs) const noexcept{
        return lhs.n < rhs.n;
    }
    override static bool operator>(const self_t& lhs, const self_t& rhs) const noexcept{
        return lhs.n > rhs.n;
    }
}

int main() {
    std::vector<Int> vec {{0}, {1}, {2}, {3}, {4}, {5}};
    auto max = find_extreme(vec.begin(), vec.end(), comparable::operator>);
    auto min = find_extreme(vec.begin(), vec.end(), comparable::operator<);
}
```

Additionally, the type of AFO cannot be used as a base class. All objects of a given AFO type are structurally identical and are usable as NTTPs. Members of the AFO type are implicitly no_unique_address.

Supporting Type-erasure and the dyns

> *There are many advanced libraries that try to reduce the boilerplate burden of implementing type erasure. But I think it's time that type erasure becomes a first-class language feature. Rust does it, everyone likes it, and with interfaces available in C++, it's an easy step to dynamic polymorphism. --New Circle Note*

Type erasure is famous for providing performant, flexible, and safe dynamic dispatch. There are a bunch of libraries that support type erasure (notably Boost.TypeErasure, Folly.Poly, Boost.Interfaces, and Adobe.Poly). Although they have made huge progress in reducing boilerplate, some issues still exist...

```cpp
//Boost.TypeErasure
namespace erasure = boost::type_erasure;

BOOST_TYPE_ERASURE_MEMBER((has_get_area), get_area)
BOOST_TYPE_ERASURE_MEMBER((has_get_perimeter), get_perimeter)

using ShapeRequirements = boost::mpl::vector<
 erasure::copy_constructible<>,
 has_get_area<double(), erasure::_self const>,
 has_get_perimeter<double(), erasure::_self const>,
 erasure::relaxed>;
using Shape = erasure::any<ShapeRequirements>;


//Folly.Poly
// This example is an adaptation of one found in Louis Dionne's dyno library.
#include <folly/Poly.h>
#include <iostream>

struct IDrawable {
    // Define the interface of something that can be drawn:
    template <class Base> struct Interface : Base {
        void draw(std::ostream& out) const { folly::poly_call<0>(*this, out);}
    };
    // Define how concrete types can fulfill that interface (in C++17):
    template <class T> using Members = folly::PolyMembers<&T::draw>;
};

// Define an object that can hold anything that can be drawn:
using drawable = folly::Poly<IDrawable>;

struct Square {
    void draw() const { std::println("Square"); }
};

struct Circle {
    void draw() const { std::println("Circle"); }
};

void f(drawable const& d) {
    d.draw();
}

int main() {
    f(Square{}); // prints Square
    f(Circle{}); // prints Circle
}
```

...and we can use the `dyn` s and interfaces to eliminate them...

```
interface IDrawable {
    requires void draw() const;
};

struct Square {
    void draw() const { std::println("Square"); }   //implicit mapping
};
struct Circle {
    void draw() const { std::println("Circle"); }   //implicit mapping
};

void f(std::unique_ptr<dyn<IDrawable>> d) {
    d->draw(std::cout);
}

int main() {
    //see below for more information of make_unique_dyn<>
    f(make_unique_dyn<Square, IDrawable>()); // prints Square
    f(make_unique_dyn<Circle, IDrawable>()); // prints Circle
}
```

...this is close to what Rust does!

```
trait IDrawable {
    fn draw(&self);
}

struct Square;
impl IDrawable for Square {
    fn draw(&self) { println!("Square"); }
}
struct Circle;
impl IDrawable for Circle {
    fn draw(&self) { println!("Circle"); }
}

fn f(d: Box<dyn IDrawable>) {
    d.draw();
}

fn main() {
    f(Box::new(Square{}));
    f(Box::new(Circle{}));
}
```

We can feel that the code use external polymorphism are quite similar to the code using inheritance polymorphism. The `dyn` s mirror the abstract base class pointers. So if you is familiar with virtual inheritance, you should also be familiar with the `dyn` s.

The `dyn` is a new language entity. The `dyn` contains two fields:

1. The data object pointer points to the real object on the stack
2. A dyntable pointer points to the dyntable, which stores the real object's associative functions' pointers

We can create `dyn` s using the `make_dyn<interface-name>(pointer-expression)` . For example, `make_dyn<IFace>(&x)` works like the following (x is stored in the stack):

1. Create the data object pointer that points to the object `x`
2. Build the dyntable of `x`
3. Create the dyntable pointer that points to the dyntable of `x`
4. Assemble the two pointers into one `dyn`

Like raw pointers, `dyn` s also need manual memory management, which means we need to use `delete` to release the dyntables. To avoid explicit memory management, we can reuse smart pointers to manage them:

```
template<typename Type, interface IFace>
std::unique_ptr<dyn<IFace>> make_unique_dyn(auto&& ... args) {
    return std::unique_ptr<dyn<IFace>>(make_dyn<IFace>(new Type(std::forward<decltype(args)>(args)...)));
}   //thanks to the New Circle again
```

With the `dyn` s, we can now easily implement a type-erased value semantic container `box` like what Rust has!

```
//taken from the New Circle Note, slightly modified to support multiple interfaces

template <template <template auto ...> interface IFace>
using unique_dyn = std::unique_ptr<dyn<IFace>>;

// Implicitly generate a clone interface for copy-constructible types.
template<interface ... IFace>
interface IClone : IFace... {
    // The default-clause causes SFINAE failure to protect the program from
    // being ill-foremd if IClone is attempted to be implicitly instantiated
    // for a non-copy-constructible type.
    default(is_copy_constructible_v(self_t)) requires
        std::unique_dyn<IClone> clone() const
    {
        // Pass the const Self lvalue to make_unique_dyn, which causes copy
        // construction.
        return make_unique_dyn<self_t, IClone>(*this*);
    }
};

template<interface ... IFace>
class Box {
public:
    using Ptr = std::unique_dyn<IClone<IFace...>>;

    Box() = default;

    // Allow direct initialization from Ptr.
    explicit Box(Ptr p) : p(std::move(p)) { }

    Box(Box&&) = default;
    Box(const Box& rhs) {
        // Copy constructor. This is how we clone.
        p = rhs.p->clone();
    }

    Box& operator=(Box&& rhs) = default;
    Box& operator=(const Box& rhs) {
        // Clone here too. We can't call the type erased type's assignment,
        // because the lhs and rhs may have unrelated types that are only
        // common in their implementation of IFace...
        p = rhs.p->clone();
        return self;
    }

    // Return a dyn<IFace...>*. This is reached via upcast from dyn<IClone<IFace...>>*.
    // It's possible because IFace... is a base interface of IClone<IFace...>.
    // If the user wants to clone the object, it should do so through the Box.
    dyn<IFace...>* operator->() noexcept {
        return p.get();
    }

    void reset() {
        p.reset();
    }

private:
    Ptr p;
};

template<typename Type, interface ... IFace>
```

```
Box<IFace...> make_box(auto&& .. args) {
    return Box<IFace...>(make_unique_dyn<Type, IClone<IFace...>>(std::forward<decltype(args)>(args)...));
}
```

The users' code:

```
// This is the user-written part. Very little boilerplate.

interface IPrint {
    requires void print() const;
}

interface IText {
    requires void set(std::string s);
    requires void to_uppercase();
};

implement std::string : IText, IPrint {
    override void print() const {
        // Print the address of the string and its contents.
        std::cout<< "string.IText::print ("<< &self<< ") = "<< self<< "\n";
    }
    override void set(std::string s) {
        std::cout<< "string.IText::set called\n";
        self = std::move(s);
    }
    override void to_uppercase() {
        std::cout<< "string.IText::to_uppercast called\n";
        for(char& c : self)
        c = std::toupper(c);
    }
};

int main() {
    Box x = make_box<std::string, IText, IPrint>("Hello dyn");
    x->print();

    // Copy construct a clone of x into y.
    Box y = x;

    // Mutate x.
    x->to_uppercase();

    // Print both x and y. y still refers to the original text.
    x->print();
    y->print();

    // Copy-assign y back into x, which has the original text.
    x = y;

    // Set a new text for y.
    y->set("A new text for y");

    // Print both.
    x->print();
    y->print();
}
```