Proxy: A Pointer-Semantics-Based Polymorphism Library

Document number: P3086R0
Date: 2024-01-16
Project: Programming Language C++
Audience: LEWGI, LEWG
Authors: Mingxin Wang
Reply-to: Mingxin Wang <mingxwa@microsoft.com>

Table of Contents
1 Introduction............................................................................................................. 2
2 Motivation and Scope .......................................................................................... 3
  2.1 Implementation status ...................................................................................... 4
  2.2 An example of system design .......................................................................... 4
    2.2.1 Architecting with inheritance-based polymorphism ..................................... 4
    2.2.2 Architecting with the "proxy" ..................................................................... 5
  2.3 Requirements change 1: More polymorphic expressions ............................. 6
    2.3.1 Inheritance-based polymorphism ............................................................... 7
    2.3.2 The "proxy" .................................................................................................. 7
    2.3.3 Comparison ................................................................................................. 7
  2.4 Requirements change 2: Simple factory ....................................................... 8
    2.4.1 Inheritance-based polymorphism ............................................................... 8
    2.4.2 The "proxy" .................................................................................................. 9
    2.4.3 Comparison ................................................................................................. 10
  2.5 Conclusion ....................................................................................................... 10
3 Impact on the Standard ....................................................................................... 11
4 Considerations and Design Decisions .................................................................. 12
  4.1 Pointer semantics ........................................................................................... 12
    4.1.1 Motivation .................................................................................................. 12
    4.1.2 Constraints ............................................................................................... 13
    4.1.3 Implementation ........................................................................................ 14
1 Introduction

This is a proposal for a reduced initial set of features to support general non-intrusive polymorphism in C++. Specifically, we are mostly proposing a subset of features suggested in P0957R9 with some significant improvements per user feedback:

- Class template `proxy`, representing type-erased pointers at runtime.
- Enum class `constraint_level` and struct `proxiable_ptr_constraints`, representing compile-time constraints of a pointer to model a proxy. 3 prototypes of `proxiable_ptr_constraints` are also proposed.
- Concepts `basic_facade`, `facade` and `proxiable`.
- Factory function template `make_proxy`.
For decades, object-based virtual table has been a de facto implementation of runtime polymorphism in many (compiled) programming languages including C++. There are many drawbacks in this mechanism, including life management (because each object may have different size and ownership), reflection (because it is hard to balance between usability and memory allocation) and intrusiveness. To workaround these drawbacks, some languages like Java or C# choose to sacrifice performance by introducing GC to facilitate lifetime management, and JIT-compile the source code at runtime to generate full metadata. We improved the theory and made it possible to implement generic non-intrusive polymorphism based on pointer semantics.

Comparing to P0957R9, the major changes are listed as follows:

1. The facilities to help defining dispatches and facades are removed. We are seeking easier ways to define these constructs by introducing new syntactic sugar, but this is not in the scope of this paper.
2. Per user feedback, struct proxiable_ptr_constraints is proposed as an abstraction of constraints to pointers, making it easier to learn and use. 3 prototypes are proposed, while only 1 is proposed in P0957R9 due to syntax limitation. The requirements of facade are also revised.
3. Per user feedback, multiple overloads are supported in one dispatch definition.
4. Per user feedback, proxy::invoke() has made const.
5. proxy::operator() is added when only one dispatch presents.
6. Added concept basic_facade and facade.

The rest of the paper is organized as follows: section 3 illustrates the motivation and scope of the proposed library; section 4 summarizes the impact on the standard; section 5 includes the pivotal decisions in the design; section 6 illustrates the technical specifications; the last sections summarize the paper.

2 Motivation and Scope

Polymorphism in OOP theory is an effective way to decouple components within a single programming language and allows deployment of stable ABI, therefore it is widely supported in modern programming languages including C++ and is vital in large-scale programming to decouple components and increase extendibility. Currently, there are two types of mechanisms for polymorphism in the standard: inheritance with virtual functions and polymorphic wrappers. Because the existing polymorphic wrappers in the standard, such as std::function, std::any, std::pmr::polymorphic_allocator, etc., have limited extendibility with regard to a variety of polymorphic requirements, inheritance-based polymorphism is usually inevitable in large systems nowadays.

The "proxy" is designed to help users build extendable and efficient polymorphic programs. To make implementations efficient in C++, it is helpful to collect requirements and generate high-quality code at
compile-time as possible. The basic goal of the "proxy" is to eliminate the usability and performance limitations in traditional OOP and FP.

This following section illustrates the implementation status of the proposed library, the limitations in inheritance-based polymorphism with concrete system design requirements and how the proposed library could help.

2.1 Implementation status

As a proof of concept, we have implemented the technical specifications as a single-header template library, meeting the latest standard. The implementation, including unit tests, could be found in our GitHub repo. As we tested, the implementation compiles with the latest releases of gcc, clang and MSVC, as the language standard is set to C++20 or later.

Because this paper does not aim to provide any syntactic sugar to define constructs for polymorphism, 5 macros are provided for exposition: PRO_DEF_MEMBER_DISPATCH, PRO_DEF_FREE_DISPATCH, PRO_DEF_COMBINED_DISPATCH, PRO_DEF_FACADE and PRO_MAKE_DISPATCH_PACK. The definition of the macros could be found in Appendix 8.1. There are some other experimental facilities in the codebase and will not be discussed in this paper.

2.2 An example of system design

Before discussing the limitations in inheritance-based polymorphism, it would be helpful to show the basic usage of the proposed library in concrete system design requirements compared to others. Here are the original requirements:

There are 3 "drawable" entities in a system: rectangle, circle, and point. Specifically.

- Rectangles have width, height, transparency, and area, and
- Circles have radius, transparency, and area, and
- Points do not have any property; their area is always zero.

A library function DoSomethingWithDrawable shall be defined with some algorithm. It should not be a function template to avoid code bloat and increase testability. It may "draw" any of the 3 "drawable" entities in its implementation.

2.2.1 Architecting with inheritance-based polymorphism

With the keyword virtual, a base class could be defined:

```cpp
class IDrawable {
    public:
        virtual void Draw() const = 0;
};
```

3 "drawable" entities could be defined as 3 derived classes:
class Rectangle : public IDrawable {
public:
    void Draw() const override;
    void SetWidth(double width);
    void SetHeight(double height);
    void SetTransparency(double);
    double Area() const;
};

class Circle : public IDrawable {
public:
    void Draw() const override;
    void SetRadius(double radius);
    void SetTransparency(double transparency);
    double Area() const;
};

class Point : public IDrawable {
public:
    void Draw() const override;
    constexpr double Area() const { return 0; }
};

The function could be defined as:

void DoSomethingWithDrawable(IDrawable* p);

2.2.2 Architecting with the "proxy"

To define an abstraction of "drawable", we need to define the dispatch Draw and facade FDrawable.

Here is a sample definition:

struct Draw {
    using overload_types = std::tuple<void()>
    template <class T>
        void operator()(T& self) requires(requires{ self.Draw(); }) {
            self.Draw();
        }
};

struct FDrawable {
    using dispatch_types = Draw
    static constexpr auto constraints =
        std::relocatable_ptr_constraints;
    using reflection_type = void;
};

Again, this paper does not aim to provide any syntactic sugar to define structures like above for polymorphism. 4 macros are provided for exposition: PRO_DEF_MEMBER_DISPATCH,
The definition of the macros could be found in our GitHub repo and 8.1 Appendix. With the macros, the definition above is equivalent to:

```
PRO_DEF_MEMBER_DISPATCH(Draw, void());
PRO_DEF_FACADE(FDrawable, Draw);
```

`Draw` and `FDDrawable` become two empty types with metadata required to instantiate a `proxy`. The required 3 types could be implemented as normal types without any virtual function or inheritance:

```cpp
class Rectangle {
    public:
        void Draw() const;
        void SetWidth(double width);
        void SetHeight(double height);
        void SetTransparency(double);
        double Area() const;
};
class Circle {
    public:
        void Draw() const;
        void SetRadius(double radius);
        void SetTransparency(double transparency);
        double Area() const;
};
class Point {
    public:
        void Draw() const;
        constexpr double Area() const { return 0; }
};
```

With the defined facade, the function could be defined as:

```
void DoSomethingWithDrawable(std::proxy<FDDrawable> p);
```

`std::proxy` is the major proposed class template that implements runtime polymorphism. It could be specified by any well-formed facade type like `FDDrawable`. It is implicitly convertible from pointer types of specific requirements. The syntax to invoke the `Draw` expression is: `p.invoke<Draw>()`. It is also allowed to omit the expression `Draw` since it is the only one defined in the facade, i.e., `p.invoke()` or simply `p()`.

### 2.3 Requirements change 1: More polymorphic expressions

As the system evolves, we may need to update the code to meet new requirements. For example, what if `DoSomethingWithDrawable` needs to call `Area()`?
2.3.1 Inheritance-based polymorphism

For inheritance-based polymorphism, based on the design in 2.2.1, all the base and derived classes need to be updated:

1. Another new pure virtual function needs to be added in the base class:
   ```cpp
class IDrawable {
   public:
       virtual void Draw() const = 0;
       virtual double Area() const = 0;
   };
```

2. The "override" keyword shall be added in the 3 derived classes. Although it's optional, it should usually be recommended to avoid ambiguity:
   ```cpp
class Rectangle : IDrawable {
   public:
       ...
       double Area() const override;
   };

class Circle : IDrawable {
   public:
       ...
       double Area() const override;
   };

class Point : IDrawable {
   public:
       ...
       double Area() const override { return 0; }
   };
```

2.3.2 The "proxy"

For the "proxy", based on the design in 2.2.2, only the definition of the "facade" needs to be updated, while no change is required in the implementation of the 3 entities. Specifically, another "dispatch" should be defined and added to the definition of the "facade":

```cpp
PRO_DEF_MEMBER_DISPATCH(Area, double());
PRO_DEF_FACADE(FDrawable, PRO_MAKE_DISPATCH_PACK(Draw, Area));
```

2.3.3 Comparison

When more polymorphic expressions are required in a well-designed system, inheritance-based polymorphism always changes the semantics of all the base and derived classes, while the "proxy" has less impact on the existing code.
We can also use other types in the standard library polymorphically with the "proxy" if needed. For example, if we want to abstract a mapping data structure from indices to strings for localization, we may define the following facade:

```cpp
PRO_DEF_MEMBER_DISPATCH(at, std::string(int));
PRO_DEF_FACADE(FResourceDictionary, at);
```

It could proxy any potential mapping data structure, including but not limited to `std::map<int, std::string>`, `std::unordered_map<int, std::string>`, `std::vector<std::string>`, etc.

### 2.4 Requirements change 2: Simple factory

What if a simple factory function of "drawable" is needed? For instance, parsing the command line to create a "drawable" instance.

#### 2.4.1 Inheritance-based polymorphism

For inheritance-based polymorphism, based on the design in 2.3.1, the new factory function could be designed as follows:

```cpp
IDrawable* MakeDrawableFromCommand(const std::string& s);
```

However, the semantics of the return type is ambiguous because it is a raw pointer type and does not indicate the lifetime of the object. For instance, it could be allocated via `operator new`, from a memory pool or even a global object. To make it the semantics cleaner, an experienced engineer may use smart pointers and change the return type to `std::unique_ptr<IDrawable>`:

```cpp
std::unique_ptr<IDrawable> MakeDrawableFromCommand(const std::string& s);
```

Although the code compiles, unfortunately, it introduces a bug: the destructor of `std::unique_ptr<IDrawable>` will call the destructor of `IDrawable`, but won't call the destructor of its derived classes and may result in resource leak. It is necessary to add a virtual destructor with empty implementation to `IDrawable` to avoid such leak:

```cpp
class IDrawable {
public:
    virtual void Draw() const = 0;
    virtual double Area() const = 0;
    virtual ~IDrawable() {};
};
```
Some types like Point are stateless and theoretically don't need to be created every time when needed. Is it possible to optimize the performance in this case? Because std::unique_ptr<IDrawable> is not copyable, this may require further API change, for example, using std::shared_ptr instead:

```cpp
std::shared_ptr<IDrawable> MakeDrawableFromCommand(const std::string& s);
```

If we decided to change one API from std::unique_ptr to std::shared_ptr, other APIs needs to be changed to stay compatible as well, every polymorphic type needs to inherit std::enable_shared_from_this, which may be significantly expensive in a large system.

### 2.4.2 The "proxy"

For the "proxy", based on the design in 2.3.2, we can define the factory function directly without further concern:

```cpp
std::proxy<FDrawable> MakeDrawableFromCommand(const std::string& s);
```

In the implementation, std::proxy<FDrawable> could be instantiated from all kinds of pointers with potentially different lifetime management strategy. For example, Rectangle may be created every time when requested from a memory pool, while the value of Point could be cached throughout the lifetime of the program:

```cpp
std::proxy<FDrawable> MakeDrawableFromCommand(const std::string& s) {
    std::vector<std::string> parsed = ParseCommand(s);
    if (!parsed.empty()) {
        if (parsed[0u] == "Rectangle") {
            if (parsed.size() == 3u) {
                static std::pmr::unsynchronized_pool_resource mem_pool;
                std::pmr::polymorphic_allocator<> alloc{&mem_pool};
                auto deleter = [alloc](Rectangle* ptr) mutable {
                    alloc.delete_object<Rectangle>(ptr);
                };
                Rectangle* instance = alloc.new_object<Rectangle>();
                std::unique_ptr<Rectangle, decltype(deleter)> p{
                    instance, deleter};
                p->SetWidth(std::stod(parsed[1u]));
                p->SetHeight(std::stod(parsed[2u]));
                return p;  // Implicit conversion happens
            }
            else if (parsed[0u] == "Circle") {
                if (parsed.size() == 2u) {
                    Circle circle;
                    circle.SetRadius(std::stod(parsed[1u]));
                }
            }
        }
    }
}
return std::make_proxy<FDrawable>(circle); // SBO may apply
}
}
else if (parsed[0u] == "Point") {
    if (parsed.size() == 1u) {
        static Point instance; // Global singleton
        return &instance;
    }
}
throw std::runtime_error("Invalid command");
}

No change to the existing code is needed.

2.4.3 Comparison

Lifetime management with inheritance-based polymorphism is error-prone and inflexible, while the "proxy" allows easy customization of any lifetime management strategy, including but not limited to raw pointers and various smart pointers with potentially pooled memory management.

Specifically, SBO (Small Buffer Optimization, aka., SOO, Small Object Optimization) is a common technique to avoid unnecessary memory allocation. However, for inheritance-based polymorphism, there is little facilities in the standard that support SBO; for other standard polymorphic wrappers, implementations may support SBO, but there is no standard way to configure so far. For example, if the size of `std::any` is \( n \), it is theoretically impossible to store the concrete value whose size is larger than \( n \) without external storage.

2.5 Conclusion

Prior research into future polymorphic usage is usually required when designing polymorphic types with inheritance. However, if the design research is inadequate in earlier phase, the semantics of the components may become overly complex when there are too many virtual functions, or the extendibility of the system may be insufficient when polymorphic types are coupled too closely. Anyway, the engineering cost may dramatically increase due to imperfect architecting. On the other hand, along with the evolution of the requirements, polymorphic usage may change, additional effort is usually necessary to keep the definition of polymorphic types consistent with their usage, staying good maintainability of the system. Moreover, some libraries (including the standard library) may not have proper polymorphic semantics even if they, by definition, satisfy the same specific constraints. In such scenarios, users have no alternative but to design and maintain extra middleware themselves to add polymorphism support to existing implementations.

Overall, inheritance-based polymorphism has limitations both in architecting and performance. As Sean Parent commented on NDC 2017: The requirements of a polymorphic type, by definition, comes
from its use, and there are no polymorphic types, only polymorphic use of similar types. Inheritance is the base class of evil.

3 Impact on the Standard

For existing polymorphic wrappers in the standard, including std::function, std::move_only_function, std::polymorphic_allocator and std::any, proxy can facilitate implementation with high quality. For new libraries in the standard, inventing new polymorphic wrappers is no longer necessary since proxy is ready for general polymorphism requirements.

The following example utilizes function template std::invoke to implement similar function wrapper as std::function and std::move_only_function while supporting multiple overloads.

```cpp
// Abstraction (poly is short for polymorphism)
namespace poly {

    template <class... Overloads>
    PRO_DEF_FREE_DISPATCH(Call, std::invoke, Overloads...);

    template <class... Overloads>
    PRO_DEF_FACADE(MovableCallable, Call<Overloads...>);

    template <class... Overloads>
    PRO_DEF_FACADE(CopyableCallable, Call<Overloads...>,
                    std::copyable_ptr_constraints);

} // namespace poly

// MyFunction has similar functionality as std::function,
// but supports multiple overloads
// MyMoveOnlyFunction has similar functionality as
// std::move_only_function but supports multiple overloads

template <class... Overloads>
using MyFunction = std::proxy<poly::MovableCallable<Overloads...>>;

template <class... Overloads>
using MyMoveOnlyFunction = 
                        std::proxy<poly::CopyableCallable<Overloads...>>;

int main() {
    auto f = [](auto&&... v) {
    printf("f() called. Args: ");
    ((std::cout << v << ":" << typeid(decltype(v)).name() << ",", ",", ...));
    puts(""");
    };
    MyFunction<void(int)> p0{&f};
    p0(123); // Prints "f() called. Args: 123:i," (assuming GCC)
```
```cpp
MyMoveOnlyFunction<void(), void(int), void(double)> p1{&f};
p1();  // Prints "f() called. Args:"
p1(456);  // Prints "f() called. Args: 456:i,"
p1(1.2);  // Prints "f() called. Args: 1.2:d,"
return 0;
```

4 Considerations and Design Decisions

Comparing to P0957R9, the major changes in the decisions are:

1. It is no longer recommended to define dispatches and facades with direct inheritance.
2. Simplified semantics of dispatches and facades.
3. Supported multiple overloads of a dispatch.

Specific considerations and design decisions have been made in the following aspects.

4.1 Pointer semantics

We decided to design the "proxy" based on pointer semantics for both usability and performance considerations. To allow balancing between extensibility and performance in specific cases, 3 abstractions of constraints are proposed with preferred defaults.

4.1.1 Motivation

Currently, the standard polymorphic wrapper types, including `std::function` and `std::any`, are based-on value semantics. Polymorphic wrappers based on value semantics have certain limitations in lifetime management compared to pointer semantics. Designing the "proxy" library based on pointer semantics decouples the responsibility of lifetime management from the "proxy", which provides more flexibility and helps consistency in API design without reducing runtime performance.

For example, in cases where allocator customization is required for performance considerations, `std::function` and `std::any` are not supported. Back to C++14, `std::function` used to have several constructors that take an allocator argument, but these constructors were removed per discussion in P0302R1 (Removing Allocator Support in std::function), because "the semantics are unclear, and there are technical issues with storing an allocator in a type-erased context and then recovering that allocator later for any allocations needed during copy assignment". Similarly, `std::any`, introduced in C++17, does not allows customization in allocator at all. With the proposed "proxy" library, it becomes easy to implement such requirements with customized pointers, even in hybrid lifetime management scenarios, as demonstrated earlier in 2.4.2.
4.1.2 Constraints

The first constraint to all pointer types to be eligible for proxy is the capability to be dereferenced from a const lvalue reference. Specifically, if a pointer \( p \) is a fancy pointer, \texttt{std::to_address}(p) shall be well-formed.

To allow implementation balance between extendibility and performance, a set of constraints to a pointer is introduced, including maximum size, maximum alignment, copyability, relocatability and destructibility. The term "relocatability" was introduced in \texttt{P1144R9}, "equivalent to a move and a destroy". This paper uses the term "relocatability" but does not depend on the technical specifications of \texttt{P1144R9}.

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum size</td>
<td>No less than the size of two pointers</td>
</tr>
<tr>
<td>Maximum alignment</td>
<td>No less than the alignment of a pointer</td>
</tr>
<tr>
<td>Copyability</td>
<td>None</td>
</tr>
<tr>
<td>Relocatability</td>
<td>Nothrow</td>
</tr>
<tr>
<td>Destructibility</td>
<td>Nothrow</td>
</tr>
</tbody>
</table>

Table 1 – Default constraints of relocatable pointer types

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum size</td>
<td>No less than the size of two pointers</td>
</tr>
<tr>
<td>Maximum alignment</td>
<td>No less than the alignment of a pointer</td>
</tr>
<tr>
<td>Copyability</td>
<td>Nontrivial</td>
</tr>
<tr>
<td>Relocatability</td>
<td>Nothrow</td>
</tr>
<tr>
<td>Destructibility</td>
<td>Nothrow</td>
</tr>
</tbody>
</table>

Table 2 – Default constraints of copyable pointer types
<table>
<thead>
<tr>
<th>Copyability</th>
<th>Trivial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relocatability</td>
<td>Trivial</td>
</tr>
<tr>
<td>Destructibility</td>
<td>Trivial</td>
</tr>
</tbody>
</table>

Table 3 – Default constraints of trivial pointer types

While the size and alignment could be described with `std::size_t`, there is no direct primitive in the standard to describe the constraint level of copyability, relocatability or destructibility. Thus, 4 levels of constraints, matching the standard wording, are defined in this paper: none, nontrivial, nothrow and trivial. The proposed 3 sets of defaults are listed in Table 1, Table 2 and Table 3 to try to meet the requirements of various implementations of (smart) pointers. For relocatable- and copyable-pointer-constraints It is encouraged for implementation to set the default maximum size and maximum alignment greater than or equal to the implementation of raw pointers, `std::unique_ptr` with default deleters, `std::unique_ptr` with any one-pointer-size of deleters (for pooling) and `std::shared_ptr` of any type.

### 4.1.3 Implementation

Inheritance-based polymorphism or standard polymorphic wrappers are all based on value semantics. For inheritance, although polymorphism is expressed with pointer or reference of a base type, the VTABLE is bound to the value itself. For other standard polymorphic wrappers, like `std::function` or `std::any`, the lifetime of the stored values are bound to these polymorphic wrappers without allocator customization. These limitations make it difficult to implement requirements like 2.4 without extra considerations in the code design or performance decrement.

![Figure 1 – Expected memory layout of inheritance-based polymorphism](image-url)
Because of pointer semantics, the expected memory layout of \texttt{std::proxy} is also different from traditional inheritance. For instance, Figure 1 and Figure 2 shows their expected memory layout, respectively. The expected memory layout is similar with the implementation of \texttt{std::move_only_function} in \texttt{libstdc++}, where the pointer of the actual object is dereferenced inside the virtual dispatch via \_S\_access.

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>The &quot;proxy&quot;</th>
<th>Inheritance-based polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_DEF_MEMBER_DISPATCH(Draw, void()); PRO_DEF_MEMBER_DISPATCH(Area, double()); PRO_DEF_FACADE(FDrawable, PRO_MAKE_DISPATCH_PACK(Draw, Area));</td>
<td>struct IDrawable { virtual void Draw() const = 0; virtual double Area() const = 0; virtual ~IDrawable() {} };</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>class Rectangle { public: void Draw() const { printf(&quot;{Rectangle: width = %f, height = %f}&quot;, width_, height_); } double Area() const { return width_ * height_; } private: double width_; double height_; };</td>
<td>class Rectangle : public IDrawable { public: void Draw() const override { printf(&quot;{Rectangle: width = %f, height = %f}&quot;, width_, height_); } double Area() const override { return width_ * height_; } private: double width_; double height_; };</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invocation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>void DoSomethingWithDrawable(std::proxy&lt;FDrawable&gt; p) { p.invoke<a href="">op::Draw</a>(); }</td>
<td>void DoSomethingWithDrawable(std::unique_ptr&lt;IDrawable&gt; p) { p-&gt;Draw(); }</td>
</tr>
</tbody>
</table>

Table 4 – Sample code to compile

<table>
<thead>
<tr>
<th>Processor architecture</th>
<th>Compiler family</th>
<th>Version</th>
<th>Compiler flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-64 (AMD64)</td>
<td>clang</td>
<td>13.0.0</td>
<td>-std=c++20 -O3</td>
</tr>
<tr>
<td>ARM64</td>
<td>gcc</td>
<td>11.2</td>
<td>-std=c++20 -O3</td>
</tr>
<tr>
<td>RISC-V RV64</td>
<td>clang</td>
<td>13.0.0</td>
<td>-std=c++20 -O3</td>
</tr>
</tbody>
</table>

Table 5 – Sample compiler configurations

To evaluate the quality of code generation, we tried to compile the "Drawable" example from section 2.3 with various compilers and compare the generated assembly between the sample implementation of the "proxy" and traditional inheritance-based polymorphism. Specifically, the sample code to compile is listed in Table 4, the sample compiler configurations for different processor architectures are listed in Table 5.
The "proxy" | Inheritance-based polymorphism
---|---
**Library side**
- mov rax, qword ptr [rdi]
- add rdi, 8
- jmp qword ptr [rax + 24]
- mov rdi, qword ptr [rdi]
- mov rax, qword ptr [rdi]
- jmp qword ptr [rax]

**Client side**
- mov rax, qword ptr [rdi + 8]
- movsd xmm0, qword ptr [rax]
- movsd xmm1, qword ptr [rax + 8]
- mov edi, offset .L.str.18
- mov al, 2
- jmp printf
- movsd xmm0, qword ptr [rdi + 8]
- movsd xmm1, qword ptr [rdi + 16]
- mov edi, offset .L.str
- mov al, 2
- jmp printf

<table>
<thead>
<tr>
<th>mov</th>
<th>rax, qword ptr [rdi + 8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>movsd</td>
<td>xmm0, qword ptr [rdi + 8]</td>
</tr>
<tr>
<td>movsd</td>
<td>xmm1, qword ptr [rdi + 16]</td>
</tr>
<tr>
<td>mov</td>
<td>edi, offset .L.str</td>
</tr>
<tr>
<td>mov</td>
<td>al, 2</td>
</tr>
<tr>
<td>jmp</td>
<td>printf</td>
</tr>
</tbody>
</table>

Table 6 – Generated code from clang 13.0.0 (x86-64)

**Library side**
- ldr x1, [x0], 8
- ldr x1, [x1, 24]
- mov x16, x1
- br x16

**Client side**
- mov x1, x0
- adrp x0, .LC3
- add x0, x0, :lo12:.LC3
- ldr d0, [x1]
- b printf

<table>
<thead>
<tr>
<th>mov</th>
<th>x1, x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>adrp</td>
<td>x0, .LC3</td>
</tr>
<tr>
<td>add</td>
<td>x0, x2, :lo12:.LC0</td>
</tr>
<tr>
<td>ldp</td>
<td>d0, d1, [x1, 8]</td>
</tr>
<tr>
<td>b</td>
<td>printf</td>
</tr>
</tbody>
</table>

Table 7 – Generated code from gcc 11.2 (ARM64)

**Library side**
- ld a1, 0(a0)
- ld a5, 24(a1)
- addi a0, a0, 8
- jr a5

**Client side**
- ld a0, 8(a0)
- ld a2, 8(a0)
- ld a1, 0(a0)
- lui a0, %hi(.L.str.18)
- addi a0, a0, %lo(.L.str.18)
- tail printf

<table>
<thead>
<tr>
<th>ld</th>
<th>a0, 0(a0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>a1, 0(a0)</td>
</tr>
<tr>
<td>ld</td>
<td>a5, 0(a1)</td>
</tr>
<tr>
<td>jr</td>
<td>a5</td>
</tr>
<tr>
<td>ld</td>
<td>a2, 16(a0)</td>
</tr>
<tr>
<td>ld</td>
<td>a1, 8(a0)</td>
</tr>
<tr>
<td>lui</td>
<td>a0, %hi(.L.str)</td>
</tr>
<tr>
<td>addi</td>
<td>a0, a0, %lo(.L.str)</td>
</tr>
<tr>
<td>tail</td>
<td>printf</td>
</tr>
</tbody>
</table>

Table 8 – Generated code from clang 13.0.0 (RISC-V RV64)

Trying to compile the two pieces of sample code with 3 different compilers, the generated assembly are shown in Table 6, Table 7 and Table 8. From the instructions we can see:

1. Invocations from `std::proxy` could be properly inlined, except for the virtual dispatch on the client side, similar to inheritance-based polymorphism.
2. Because `std::proxy` is based on pointer semantics, the "dereference" operation may happen inside the virtual dispatch, which generates different instructions.

3. With "clang 13.0.0 (x86-64)" and "clang 13.0.0 (RISC-V RV64)", `std::proxy` generates one more instruction than inheritance-based polymorphism, while the situation is reversed with "gcc 11.2 (ARM64)". This may infer that `std::proxy` could have similar runtime performance in invocation with inheritance-based polymorphism on the 3 processor architectures.

### 4.2 Language vs. Library

During review of P0957 series, one of the most asked questions is that why `proxy` is not a language feature, like Java or Rust. Our answer is divided into two parts:

1. We believe a programming language needs more than an abstraction of "interface" (like Java) or "trait" (like Rust) for general runtime polymorphism while allowing best-in-class code generation for modern processors. It has become clear about what is required to model a good abstraction of runtime polymorphism (proposed in this paper), but the syntax is not finalized (not in the scope of this paper). As a short-term solution in our PoC implementation, some macros are defined to facilitate definition of abstractions (see Appendix 8.1).

2. When it comes to the runtime binding to be manipulated in an application, we believe the class template in C++ is good enough to standardize the behavior, and therefore no language feature should be expected for this part.

### 4.3 The "proxy"

To provide a unified API to improve ease of use and reduce learning costs, the design of the "proxy" consults the "proxy" and "facade" design pattern from "Design Patterns: Abstraction and Reuse of Object-Oriented Design".

#### 4.3.1 Facade: Abstraction of Runtime Polymorphism

Although we are not proposing a syntax to define something like "interface", corresponding concepts are proposed. To describe the requirements of runtime polymorphism based on pointer semantics, the term "facade" is introduced. The runtime polymorphic requirements defined by facade are divided into three parts:

1. Dispatches: How to dispatch function calls to concrete objects. Each dispatch should specify the function signature and the body template.

2. Constraints: Specific constraints of applicable pointer types, as a compile-time value.

3. Reflection: Optionally, any compile-time metadata carried to runtime.

These requirements can be easily expressed with the type system of C++. A facade type models a compile-time tag to specify a proxy.
4.3.2 Copy/move constructions and assignments

To ensure the quality of code generation, the semantics of copy/move constructions and assignments are aligned with the constraints of pointers illustrated in 4.1.2. For example, `std::proxy<FDrawable>`, demonstrated in 2.3.2, is not copy-constructible, because the default copyability constraint to a pointer is "None". However, users can specify different constraint level if needed, e.g.,

```cpp
PRO_DEF_FACADE(MyFacade, /* Any dispatch */,
    std::copyable_ptr_constraints);
```

This requires the pointer at least to be copyable, regardless of whether it is nothrow or trivial. In the meantime, `std::proxy<MyFacade>` becomes copyable with both copy constructor and copy assignment.

4.3.3 Construction from a value

To simplify construction from a value, like other standard polymorphic wrapper types, the function template overloads `std::make_proxy` are proposed. With `std::make_proxy`, SBO may implicitly apply, depending on the implementation. The proposed syntax of `std::make_proxy` is similar to the constructor of `std::any`.

4.3.4 Reflection

Reflection is an essential requirement in type erasure, and the proposed class template `std::proxy` welcomes general-purpose static (compile-time) reflection other than `std::type_info`.

As, `std::type_info` is usually not adequate to carry enough useful information of a type to inspect at runtime. In other languages like C# or Java, users are allowed to acquire detailed metadata of a type-erased type at runtime with simple APIs, but this is not true for `std::function, std::any` or inheritance-based polymorphism in C++. Although these reflection facilities add certain runtime overhead to these languages, they do help users write simple code in certain scenarios. In C++, as the reflection specifications keeps evolving, there will be more static reflection facilities in the standard with more specific type information deduced at compile-time than `std::type_info`. It becomes possible for general-purpose reflection to become zero-overhead in C++ polymorphism.

As a result, we decided to make `std::proxy` support general-purpose static reflection. Here is an example to reflect the given types to `MyReflectionInfo`:

```cpp
class MyReflectionInfo {
    public:
        template <class P>
```
constexpr explicit MyReflectionInfo(std::in_place_type_t<P>) :
    type_(typeid(P)) {}

const char* GetName() const noexcept { return type_.name(); }

private:
    const std::type_info& type_; 
};
PRO_DEF_FACADE(MyFacade, /* Any dispatch */,
    std::relocatable_ptr_constraints, MyReflectionInfo);

Users may call MyReflectionInfo::GetName() to get the implementation-defined name of a
type at runtime:

std::proxy<MyFacade> p;
puts(p.reflect().GetName());

4.4 Compared to other solutions

This section summarizes the design of several other C++ libraries and typical programming languages
in polymorphism. They all have certain limitations in usability or performance, which are resolved in the
proposed "proxy" library.

4.4.1 The "dyno" library

The "dyno" is an open-source C++ library that also aims to "solve the problem of runtime
polymorphism better than vanilla C++ does". Here is a sample usage copied from its documentation:

using namespace dyno::literals;

// Define the interface of something that can be drawn
struct Drawable : decltype(dyno::requires_(
    "draw"_s = dyno::method<void (std::ostream&) const>
)) { };

// Define how concrete types can fulfill that interface
template <typename T>
auto const dyno::default_concept_map<Drawable, T> =
dyno::make_concept_map(
    "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); } );

// Define an object that can hold anything that can be drawn.
struct drawable {
    template <typename T>
    drawable(T x) : poly_{x} { }
}
void draw(std::ostream& out) const
{ poly_.virtual_("draw"_s)(out); }

private:
    dyno::poly<Drawable> poly_;
};

The "dyno" library also provides some macros to simplify the definition above, which will not be discussed in this paper. As illustrated in its documentation, the "goodies" we get from the "dyno" library are:

**Non-intrusive**
An interface can be fulfilled by a type without requiring any modification to that type. Heck, a type can even fulfill the same interface in different ways! With Dyno, you can kiss ridiculous class hierarchies goodbye.

**100% based on value semantics**
Polymorphic objects can be passed as-is, with their natural value semantics. You need to copy your polymorphic objects? Sure, just make sure they have a copy constructor. You want to make sure they don't get copied? Sure, mark it as deleted. With Dyno, silly clone() methods and the proliferation of pointers in APIs are things of the past.

**Not coupled with any specific storage strategy**
The way a polymorphic object is stored is really an implementation detail, and it should not interfere with the way you use that object. Dyno gives you complete control over the way your objects are stored. You have a lot of small polymorphic objects? Sure, let's store them in a local buffer and avoid any allocation. Or maybe it makes sense for you to store things on the heap? Sure, go ahead.

**Flexible dispatch mechanism to achieve best possible performance**
Storing a pointer to a vtable is just one of many different implementation strategies for performing dynamic dispatch. Dyno gives you complete control over how dynamic dispatch happens, and can in fact beat vtables in some cases. If you have a function that's called in a hot loop, you can for example store it directly in the object and skip the vtable indirection. You can also use application-specific knowledge the compiler could never have to optimize some dynamic calls — library-level devirtualization.

For "non-intrusive", the design direction also applies to the proposed "proxy" library.

For "100% based on value semantics", the design direction is different from the proposed "proxy" library, while the "proxy" is based on pointer semantics, as discussed in 4.1.1, value semantics has certain limitations in lifetime management.

For "Not coupled with any specific storage strategy", I don't think the statement is accurate for the "dyno" library. Looking at the definition of the class template "dyno::poly":

```cpp
 void draw(std::ostream& out) const
 { poly_.virtual_("draw"_s)(out); }

private:
    dyno::poly<Drawable> poly_;
};
```
Since the `Storage` is defined on the template, even we can specify different storage strategies at compile-time, one instantiation of `poly` is always bound to a specific storage strategy. Such limitations make it difficult to have different lifetime management strategies at runtime without additional overhead. The "simple factory" mentioned in 2.4 is a good example of such requirements. As mentioned earlier, the proposed "proxy" library allows different lifetime management strategies of one instantiation of proxy and thus does not have such limitations.

Taking a closer look at the implementation of "`dyno::sbo_storage`", which is designed to eliminate heap allocation, we can see a runtime conditional logic when getting the pointer of the underlying object, which is a "hot" expression each time a polymorphic expression is performed:

```cpp
return static_cast<T*>(uses_heap() ? ptr_ : &sb_);
```

Such overhead could be eliminated in the proposed "proxy" library, as discussed in 4.1.3.

For "Flexible dispatch mechanism to achieve best possible performance", I don't think de-virtualization is a major requirement of runtime polymorphism.

### 4.4.2 The "DGPVC" library

Although the Concepts can define "how should concrete implementations look like", not all the information that could be represented by a concept is suitable for polymorphism. For example, we could declare an inner type of a type in a concept definition, like:

```cpp
template <class T>
concept bool Foo() { 
    return requires {
        typename T::bar;
    };
}
```

But it is unnecessary to make this piece of information polymorphic because this expression makes no sense at runtime. Some feedback suggests that it is acceptable to restrict the definition of a concept from anything not suitable for polymorphism, including but not limited to inner types, friend functions, constructors, etc. This solution does not seem to be compatible with the C++ type system because:

1. There is no such mechanism to verify whether a definition of a concept is suitable for polymorphism, and
2. There is no such mechanism to specify a type by a concept, like
\texttt{som e\_class\_tem plate\langle SomeConcept \rangle}, because a concept is not a type.

The "Dynamic Generic Programming with Virtual Concepts" (DGPVC) is a solution that adopts this. However, on the one hand, it introduces some syntax, mixing the "concepts" with the "virtual qualifier", which makes the types ambiguous. From the code snippets included in the paper, we can tell that "virtual concept" is an "auto-generated" type. Compared to introducing new syntax, I prefer to make it a "magic class template", which at least "looks like a type" and much easier to understand. On the other hand, there seems not to be enough description about how to implement the entire solution introduced in the paper, and it remains hard for us to imagine how are we supposed to implement for the expressions that cannot be declared virtual, e.g., friend functions that take values of the concrete type as parameters.

5 Technical Specifications

5.1 Feature test macro

In \texttt{/version.syn}, add:

\begin{quote}
\#define \_\_cpp\_lib\_proxy YYYYMM // also in <proxy>
\end{quote}

The placeholder value shall be adjusted to denote this proposal's date of adoption.

5.2 Header <proxy> synopsis

namespace std {
    enum class constraint\_level { none, nontrivial, nothrow, trivial };

    struct proxiable\_ptr\_constraints {
        std::size\_t max\_size;
        std::size\_t max\_align;
        constraint\_level copyability;
        constraint\_level relocatability;
        constraint\_level destructibility;
    };

    constexpr proxiable\_ptr\_constraints relocatable\_ptr\_constraints{
        .max\_size = at least sizeof(void\*) * 2u,
        .max\_align = at least alignof(void\*),
        .copyability = constraint\_level::none,
        .relocatability = constraint\_level::nothrow,
        .destructibility = constraint\_level::nothrow,
    };

    constexpr proxiable\_ptr\_constraints copyable\_ptr\_constraints{
        .max\_size = at least sizeof(void\*) * 2u,
        .max\_align = at least alignof(void\*),
        .copyability = constraint\_level::nontrivial,
        .relocatability = constraint\_level::nothrow,

22
.destructibility = constraint_level::nothrow,
};
constexpr proxiable_ptr_constraints trivial_ptr_constraints{
    .max_size = at least sizeof(void*),
    .max_align = at least alignof(void*),
    .copyability = constraint_level::trivial,
    .relocatability = constraint_level::trivial,
    .destructibility = constraint_level::trivial,
};

template <class F>
concept basic_facade = see below;

template <class F>
concept facade = see below;

template <class P, class F>
concept proxiable = see below;

template <basic_facade F>
    class proxy;

template <class F, class T, class... Args>
    proxy<F> make_proxy(Args&&... args);

template <class F, class T, class U, class... Args>
    proxy<F> make_proxy(initializer_list<U> il, Args&&... args);


template <class F, class T>
    proxy<F> make_proxy(T&& value);

template <class F>
    void swap(proxy<F>& a, proxy<F>& b) noexcept(see below);

5.3 Constraints

template <class F>
concept basic_facade = see below;
A type F satisfies concept basic_facade when:

- typename F::dispatch_types is an instantiation of std::tuple, and
- F::constraints is a compile-time constant of type proxiable_ptr_constraints, and
- F::constraints.max_align is a power of 2, and
- F::constraints.max_size is a multiple of F::constraints.max_align, and
- typename F::reflection_type is either void or a trivially copyable type.
template <class F>
concept facade = see below;
A type F satisfies concept basic_facade when:

- F satisfies concept basic_facade, and
- For each tuple element D of typename F::dispatch_types,
  o D is trivially default constructible, and
  o typename D::overload_types is an instantiation of std::tuple of at least 1
    function type with distinct argument type combinations.

template <class P, class F>
concept proxiable = see below;
Types P and F satisfies concept proxiable when (p denotes a value of const P):

- F satisfies concept facade, and
- P is a pointer type, or std::to_address(p) is well-formed,
- sizeof(P) <= F::constraints.max_size, and
- alignof(P) <= F::constraints.max_align, and
- The copyability of P satisfies F::constraints.copyability, and
- The relocatability of P satisfies F::constraints.relocatability, and
- The destructibility of P satisfies F::constraints.destructibility, and
- For each tuple element D of tuple element of typename F::dispatch_types, for each
  tuple element O of typename D::overload_types (Args... denotes the argument
  types of O, args... denotes values of Args...), D{}(*DEDUCE_ADDRESS(p),
  std::forward<Args>(args)... ) is well-formed, where DEDUCE_ADDRESS is defined
  as:
  template <class P>
  auto DEDUCE_ADDRESS(const P& p) {
    if constexpr (std::is_pointer_v<P>) {
      return p;
    } else {
      return std::to_address(p);
    }
  }
- typename F::reflection_type is either void or constructible from
  std::in_place_type_t<P> at compile-time.
5.4 Proxy

5.4.1 Class template proxy

5.4.1.1 General

namespace std {

    template <basic_facade F>
    class proxy {
    public:
        proxy() noexcept;
        proxy(const proxy& rhs) noexcept;
        proxy(proxy&& rhs) noexcept;
        template <class P>
            proxy(P&& ptr) noexcept;
        template <class P, class... Args>
            explicit proxy(in_place_type_t<P>, Args&&... args);
        template <class P, class U, class... Args>
            explicit proxy(in_place_type_t<P>, initializer_list<U> il, Args&&... args);
        proxy& operator=(nullptr_t) noexcept;
        proxy& operator=(const proxy& rhs) noexcept;
        proxy& operator=(proxy&& rhs) noexcept;
        template <class P>
            proxy& operator=(P&& ptr) noexcept;
        ~proxy() noexcept;

        bool has_value() const noexcept;
        see below reflect() const noexcept requires;
        void reset() noexcept requires;
        void swap(proxy& rhs) noexcept requires;
        template <class P, class... Args>
            P& emplace(Args&&... args) noexcept requires;
        template <class P, class U, class... Args>
            P& emplace(initializer_list<U> il, Args&&... args) noexcept requires;
        template <class D = see below, class... Args>
            see below invoke(Args&&... args) const requires;
        template <class... Args>
            see below operator()(Args&&... args) const requires;
    };
}

Any instance of proxy<F> at any given time either proxies a pointer or does not proxy a pointer. When an instance of proxy<F> proxies a pointer, it means that an object of some pointer type P, referred to as the proxy's contained value, where proxiable<P, F> is true, is allocated within the storage of the proxy object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value. The contained value shall be allocated in a region of the proxy<F> storage suitably aligned for the type P.
The following constants are defined for exposition only:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class P, class... Args&gt;</td>
<td>conditional_t&lt;proxiable&lt;P, F&gt;, is_nothrow_constructible&lt;P, Args...&gt;, false_type&gt;::value</td>
</tr>
<tr>
<td>HasNothrowPolyConstructor&lt;P, Args...&gt;</td>
<td>conditional_t&lt;proxiable&lt;P, F&gt;, is_constructible&lt;P, Args...&gt;, false_type&gt;::value</td>
</tr>
<tr>
<td>HasTrivialCopyConstructor</td>
<td>F::constraints.copyability == constraint_level::trivial</td>
</tr>
<tr>
<td>HasNothrowCopyConstructor</td>
<td>F::constraints.copyability &gt;= constraint_level::nothrow</td>
</tr>
<tr>
<td>HasCopyConstructor</td>
<td>F::constraints.copyability &gt;= constraint_level::nontrivial</td>
</tr>
<tr>
<td>HasNothrowMoveConstructor</td>
<td>F::constraints.relocatability &gt;= constraint_level::nothrow</td>
</tr>
<tr>
<td>HasMoveConstructor</td>
<td>F::constraints.relocatability &gt;= constraint_level::nontrivial</td>
</tr>
<tr>
<td>HasTrivialDestructor</td>
<td>F::constraints.destructibility == constraint_level::trivial</td>
</tr>
<tr>
<td>HasNothrowDestructor</td>
<td>F::constraints.destructibility &gt;= constraint_level::nothrow</td>
</tr>
<tr>
<td>HasDestructor</td>
<td>F::constraints.destructibility &gt;= constraint_level::nontrivial</td>
</tr>
<tr>
<td>template &lt;class P, class... Args&gt;</td>
<td>HasNothrowPolyAssignment</td>
</tr>
<tr>
<td>HasNothrowPolyAssignment</td>
<td>HasNothrowPolyConstructor&lt;P, Args...&gt; &amp;&amp; HasNothrowDestructor</td>
</tr>
<tr>
<td>template &lt;class P, class... Args&gt;</td>
<td>HasPolyAssignment</td>
</tr>
<tr>
<td>HasPolyAssignment</td>
<td>HasPolyConstructor&lt;P, Args...&gt; &amp;&amp; HasDestructor</td>
</tr>
<tr>
<td>HasTrivialCopyAssignment</td>
<td>HasTrivialCopyConstructor &amp;&amp; HasTrivialDestructor</td>
</tr>
<tr>
<td>HasNothrowCopyAssignment</td>
<td>HasNothrowCopyConstructor &amp;&amp; HasNothrowDestructor</td>
</tr>
<tr>
<td>HasCopyAssignment</td>
<td>HasNothrowCopyAssignment</td>
</tr>
<tr>
<td>HasNothrowMoveAssignment</td>
<td>HasNothrowMoveConstructor &amp;&amp; HasNothrowDestructor</td>
</tr>
<tr>
<td>HasMoveAssignment</td>
<td>HasMoveConstructor &amp;&amp; HasDestructor</td>
</tr>
</tbody>
</table>

5.4.1.2 Construction and destruction

proxy() noexcept;
proxy(nullptr_t) noexcept;

Postconditions: *this does not contain a value.

Remarks: No contained value is initialized.

proxy(const proxy& rhs) noexcept(see below) requires(see below);

Constraints: The expression inside requires is equivalent to HasCopyConstructor.
Effects: If \( \text{rhs.\ has\ value()} \) is \texttt{false}, constructs an object that has no value. Otherwise, equivalent to \( \text{proxy(in\ place\ type<P>, rhs.cast<P>());} \) where \( P \) is the type of the contained value of \( \text{rhs} \).
Postconditions: \( \text{has\ value()} == \text{rhs.\ has\ value()} \).
Throws: Any exception thrown by the selected constructor of \( P \).
Remarks: The expression inside \texttt{noexcept} is equivalent to \texttt{HasNothrowCopyConstructor}. Specifically,
- if the constraints are not satisfied, the constructor is deleted, or
- if \texttt{HasTrivialCopyConstructor} is \texttt{true}, the constructor is trivial.

\[
\text{proxy(proxy&& rhs) noexcept(see\ below) requires(see\ below);}\
\]
Constraints: The expression inside \texttt{requires} is equivalent to \texttt{HasMoveConstructor}.
Effects: If \( \text{rhs.\ has\ value()} \) is \texttt{false}, constructs an object that has no value. Otherwise, equivalent to
\[
(\text{proxy(in\ place\ type<P>, std\::move(rhs.cast<P>());)}, \text{rhs.reset();})\]
where \( P \) is the type of the contained value of \( \text{rhs} \).
Postconditions: \( \text{rhs} \) does not contain a value.
Throws: Any exception thrown by the selected constructor of \( P \).
Remarks: The expression inside \texttt{noexcept} is equivalent to \texttt{HasNothrowMoveConstructor}. If the constraints are not satisfied, the constructor is deleted.

\[
\text{template <class P>}
\]
\[
\text{proxy(P&& ptr) noexcept(see\ below) requires(see\ below);}\
\]
Let \( VP \) be \texttt{decay_t<P>}. 
Constraints: The expression inside \texttt{requires} is equivalent to \texttt{HasPolyConstructor<VP, P>}. 
Effects: Initializes the contained value as if direct-initializing an object of type \( VP \) with \( \text{std\::forward<P>(ptr)} \). 
Postconditions: \( \ast\text{this} \) contains a value of type \( VP \). 
Throws: Any exception thrown by the selected constructor of \( VP \). 
Remarks: The expression inside \texttt{noexcept} is equivalent to \texttt{HasNothrowPolyConstructor<VP, P>}. 

\[
\text{template <class P, class... Args>}
\]
\[
\text{explicit proxy(in\ place\ type_t<P>, Args&&... args)}\
\text{noexcept(see\ below) requires(see\ below);}\
\]
Constraints: The expression inside \texttt{requires} is equivalent to \texttt{HasPolyConstructor<P, Args...>}. 
Effects: Initializes the contained value as if direct-non-list-initializing an object of type \( P \) with the arguments \( \text{std\::forward<Args>(args)}\ldots \). 
Postconditions: \( \ast\text{this} \) contains a value of type \( P \). 
Throws: Any exception thrown by the selected constructor of \( P \). 
Remarks: The expression inside \texttt{noexcept} is equivalent to \texttt{HasNothrowPolyConstructor<P, Args...>}. 

27
template <class P, class U, class... Args>
explicit proxy(in_place_type_t<P>, initializer_list<U> il,
            Args&&... args)
    noexcept(see below) requires(see below);

Constraints: The expression inside requires is equivalent to HasPolyConstructor<P,
initializer_list<U>&, Args...>.
Effects: Initializes the contained value as if direct-non-list-initializing an object of type P with the
arguments il, std::forward<Args>(args)....
Postconditions: *this contains a value of type P.
Throws: Any exception thrown by the selected constructor of P.
Remarks: The expression inside noexcept is equivalent to
HasNothrowPolyConstructor<P, initializer_list<U>&, Args...>.

~proxy() noexcept(see below) requires(see below);
Constraints: The expression inside requires is equivalent to HasDestructor.
Effects: As if by reset().
Throws: Any exception thrown by the destructor of the contained value.
Remarks: The expression inside noexcept is equivalent to HasNothrowDestructor.
Specifically,
- if the constraints are not satisfied, the destructor is deleted, or
- if HasTrivialDestructor is true, the destructor is trivial.

5.4.1.3 Assignment
proxy& operator=(nullptr_t) noexcept(see below) requires(see below);
Constraints: The expression inside requires is equivalent to HasDestructor.
Effects: If has_value() is true, destroys the contained value.
Postconditions: *this does not contain a value.
Remarks: The expression inside noexcept is equivalent to HasNothrowDestructor.

proxy& operator=(const proxy& rhs) noexcept(see below) requires(see
below);
Constraints: The expression inside requires is equivalent to HasCopyAssignment.
Effects: As if by proxy(rhs).swap(*this). No effects if an exception is thrown.
Returns: *this.
Throws: Any exception thrown during copy construction, relocation, or destruction of the contained
value.
Remarks: The expression inside noexcept is equivalent to HasNothrowCopyAssignment.
Specifically,
- if the constraints are not satisfied, the assignment operator is deleted, or
- if HasTrivialCopyAssignment is true, the assignment operator is trivial.

proxy& operator=(proxy&& rhs) noexcept(see below) requires(see
below);
Constraints: The expression inside requires is equivalent to HasMoveAssignment.
Effects: As if by proxy(std::move(rhs)).swap(*this).

Returns: *this.
Throws: Any exception thrown during move construction, relocation, or destruction of the contained
value.
Remarks: The expression inside noexcept is equivalent to HasNothrowMoveAssignment.
Specifically,
- if the constraints are not satisfied, the move assignment operator is deleted, or
- if HasTrivialMoveAssignment is true, the move assignment operator is trivial.
Returns: \*this.
Throws: Any exception thrown during relocation, destruction, or swap of the contained value.
Remarks: The expression inside noexcept is equivalent to HasNothrowMoveAssignment. If the constraints are not satisfied, the assignment operator is deleted.

```cpp
template <class P>
proxy& operator=(P&& ptr) noexcept(see below) requires(see below);
Let VP be decay_t<
Constraints: The expression inside requires is equivalent to HasPolyAssignment<VP, P>.
Effects: As if by proxy(std::forward<P>(p)).swap(*this).
Returns: \*this.
Throws: Any exception thrown during construction, destruction, or swap of the contained value.
Remarks: The expression inside noexcept is equivalent to HasNothrowPolyAssignment<VP, P>.
```

```cpp
template <class P, class... Args>
P& emplace(Args&&... args) noexcept(see below) requires(see below);
Constraints: The expression inside requires is equivalent to HasPolyAssignment<P, Args...>.
Effects: Calls \*this = nullptr. Then initializes the contained value as if direct-non-list-initializing an object of type P with the arguments std::forward<Args>(args)....
Postconditions: \*this contains a value of type P.
Returns: A reference to the new contained value.
Throws: Any exception thrown during the destruction of the previous contained value or by the selected constructor of P.
Remarks: The expression inside noexcept is equivalent to HasNothrowPolyAssignment<P, Args...>. If an exception is thrown during the call to P's constructor, \*this does not contain a value, and the previous contained value (if any) has been destroyed.
```

```cpp
template <class P, class U, class... Args>
P& emplace(initializer_list<U> il, Args&&... args) noexcept(see below) requires(see below);
Constraints: The expression inside requires is equivalent to HasPolyAssignment<P, initializer_list<U>&, Args...>.
Effects: Calls \*this = nullptr. Then initializes the contained value as if direct-non-list-initializing an object of type P with the arguments il, std::forward<Args>(args)....
Postconditions: \*this contains a value of type P.
Returns: A reference to the new contained value.
Throws: Any exception thrown during the destruction of the previous contained value or by the selected constructor of P.
Remarks: The expression inside noexcept is equivalent to HasNothrowPolyAssignment<P, initializer_list<U>&, Args...>. If an
```
exception is thrown during the call to P's constructor. *this does not contain a value, and the previous contained value (if any) has been destroyed.

5.4.1.4 Swap

void swap(proxy& rhs) noexcept(see below) requires(see below);

Constraints: The expression inside requires is equivalent to HasMoveConstructor.

Effects: See the table below:

<table>
<thead>
<tr>
<th>rhs contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swap the contained values of *this and rhs with a temporary storage. If an exception is thrown, each of *this and rhs is in a valid state with unspecified value.</td>
<td>Equivalent to (*this = std::move(rhs)); post condition is that *this contains a value and rhs does not contain a value.</td>
</tr>
</tbody>
</table>

| rhs does not contain a value | Equivalent to (rhs = std::move(*this)); post condition is that *this does not contain a value and rhs contains a value. |
| no effect |

Remarks: The expression inside noexcept is equivalent to HasNothrowMoveConstructor.

5.4.1.5 Observers

bool has_value() const noexcept;

Returns: true if and only if *this contains a value.

see below reflect() const noexcept requires(see below);

Constraints: The expression inside requires is equivalent to !is_void_v<typename F::reflection_type>.

Return type: const typename F::reflection_type&.

Returns: A const reference of typename F::reflection_type constructed from in_place_type_t<P> and has static storage duration, where P is the type of the contained value.

Remarks: If *this does not contain a value, the behavior is undefined.

5.4.1.6 Modifiers

void reset() noexcept(see below) requires(see below);

Constraints: The expression inside requires is equivalent to HasDestructor.

Effects: If *this contains a value, destroys the contained value; otherwise, no effect.

Postconditions: *this does not contain a value.

Remarks: The expression inside noexcept is equivalent to HasNothrowDestructor. If an exception is thrown during the call to P's destructor, *this is in a valid state with unspecified value.

5.4.1.7 Invocation

template <class D = see below, class... Args>
see below invoke(Args&&... args) const requires(see below);
**Constraints:** The expression inside requires is equivalent to that \( F \) meets the Facade requirements, and \( D \) is a valid dispatch defined by \( F \), and \( \text{Args...} \) matches one overload of \( D \).

**Preconditions:** \(*\text{this}\) contains a value.

**Effects:** Equivalent to return \( D\{}(*\text{DEDUCE_ADDRESS}(p)\), 
\text{static\_cast<_\text{Args}>(args)\ldots}) \), where \( p \) is the contained value, _\text{Args}... are the argument types defined by the matched overload of \( D \).

**Throws:** Any exception thrown from the equivalent expression.

**Remarks:** The default type of \( D \) applies if and only if \( F \) defines exactly one dispatch. If \(*\text{this}\) does not contain a value, the behavior is undefined.

**template <class... Args>**

*see below* operator() (Args&&... args) const requires (see below);

**Constraints:** The expression inside requires is equivalent to that \( F \) meets the Facade requirements, and only one dispatch \( D \) defined by \text{typename} \( F\:\text{::dispatch\_types} \), and \( \text{Args...} \) matches one overload of \( D \).

**Preconditions:** \(*\text{this}\) contains a value.

**Effects:** Equivalent to return \text{invoke}(std\::forward<Args>(args)\ldots).

**Throws:** Any exception thrown from the equivalent expression.

**Remarks:** If \(*\text{this}\) does not contain a value, the behavior is undefined.

### 5.4.2 Creation

**template <class F, class T, class... Args>**

proxy<F> make_proxy (Args&&... args);

**Effects:** Creates an instance of \text{proxy<F>} with an unspecified pointer type of \( T \), where the value of \( T \) is direct-non-list-initialized with the arguments \text{std\::forward<Args>(args)\ldots}.

**Remarks:** Implementations are not permitted to use additional storage, such as dynamic memory, to allocate the value of \( T \) if the following conditions apply:

- \( \text{sizeof}(T) \leq \text{F\::\text{constraints.max\_size}} \) is true, and
- \( \text{alignof}(T) \leq \text{F\::\text{constraints.max\_alignment}} \) is true, and
- \( T \) meets the copyability requirements defined by \text{F\::\text{constraints.copyability}}, and
- \( T \) meets the relocatability requirements defined by \text{F\::\text{constraints.relocatability}}, and
- \( T \) meets the destructibility requirements defined by \text{F\::\text{constraints.destructibility}}.

**template <class F, class T, class U, class... Args>**

proxy<F> make_proxy (initializer_list<U> il, Args&&... args);

**Effects:** Equivalent to return make_proxy<F, T>(il, 
\text{std\::forward<Args>(args)\ldots}).

**template <class F, class T>**

proxy<F> make_proxy (T&& value);

**Effects:** Equivalent to return make_proxy<F, 
\text{decay\_t<T>>(std\::forward<T>(value))}.
5.4.3 Specialized algorithms

```cpp
template <class F>
void swap(proxy<F>& a, proxy<F>& b) noexcept(see below);
```

**Effects:** Equivalent to `a.swap(b)`.

**Remarks:** The expression inside `noexcept` is equivalent to `(noexcept(a.swap(b)))`.

6 Acknowledgements

Mingxin would like to thank: Tian Liao (Microsoft) for the insights into the library design and open source. Roger Orr for pointing out the potential ODR violation in the proposed syntax in revision 5. Wei Chen (Jilin University), Herb Sutter, Chandler Carruth, Daveed Vandevoorde, Bjarne Stroustrup, JF Bastien, Bengt Gustafsson and Chuang Li (Microsoft) for their valuable feedback on earlier revisions of this paper.

7 Summary

The "proxy" library is an extendable and efficient solution for polymorphism. We believe this feature will largely improve the usability of the C++ programming language, especially in large-scale programming.

8 Appendix

8.1 Helper macros

```cpp
#define PRO_DEF_MEMBER_DISPATCH(NAME, ...) see below
#define PRO_DEF_FREE_DISPATCH(NAME, FUNC, ...) see below
#define PRO_DEF_COMBINED_DISPATCH(NAME, ...) see below
#define PRO_MAKE_DISPATCH_PACK(...) see below
#define PRO_DEF_FACADE(NAME, ...) see below
```

The helper macros are intended to facilitate definition of dispatch and facade types. Considering the standard keeps evolving, these macros not proposed to merge to the standard for now. The implementation of the helper macros could be found in our GitHub repo.

```cpp
#define PRO_DEF_MEMBER_DISPATCH(NAME, ...) see below
Syntax: PRO_DEF_MEMBER_DISPATCH(NAME, OVERLOADS...)

Constraints: `NAME` shall not be defined in the context, and is a valid name of member function. `OVERLOADS` shall not be empty. Each type in `OVERLOADS` shall be a valid function type.
```
**Effect:** Define a type named \texttt{NAME}, which contains:
- an inner type alias \texttt{overload\_types} defined as \texttt{std::tuple<OVERLOADS...>}, and
- various overloads of \texttt{operator()} matching each function type in \texttt{OVERLOADS}; for each \texttt{O} in \texttt{OVERLOADS}, where the return type is \texttt{R} and argument types are \texttt{Args...}, a SFINAE-friendly member function template \texttt{operator()} is defined to forward invocation to the member function \texttt{NAME} with a given value, equivalent to:

```cpp
template <class T>
decltype(auto) operator()(T& self, Args&&... args)
    requires(requires{self.NAME(std::forward<Args>(args)...);})
    { return self.NAME(std::forward<Args>(args)...); } 
```

\#define \texttt{PRO\_DEF\_FREE\_DISPATCH(NAME, FUNC, \ldots)} see below

**Syntax:** \texttt{PRO\_DEF\_FREE\_DISPATCH(NAME, FUNC, OVERLOADS\ldots)}

**Constraints:** \texttt{NAME} shall not be defined in the context, and is \texttt{FUNC} a valid name in the context. 
\texttt{OVERLOADS} shall not be empty. Each type in \texttt{OVERLOADS} shall be a valid function type.

**Effect:** Define a type named \texttt{NAME}, which contains:
- an inner type alias \texttt{overload\_types} defined as \texttt{std::tuple<OVERLOADS...>}, and
- various overloads of \texttt{operator()} matching each function type in \texttt{OVERLOADS}; for each \texttt{O} in \texttt{OVERLOADS}, where the return type is \texttt{R} and argument types are \texttt{Args...}, a SFINAE-friendly member function template \texttt{operator()} is defined to forward invocation to \texttt{FUNC} with a given value, equivalent to:

```cpp
template <class T>
decltype(auto) operator()(T& self, Args&&... args)
    requires(requires{FUNC(self, std::forward<Args>(args)...);})
    { return FUNC(self, std::forward<Args>(args)...); } 
```

\#define \texttt{PRO\_DEF\_COMBINED\_DISPATCH(NAME, \ldots)} see below

**Syntax:** \texttt{PRO\_DEF\_COMBINED\_DISPATCH(NAME, DISPATCHES\ldots)}

**Constraints:** \texttt{NAME} shall not be defined in the context. \texttt{DISPATCHES} shall not be empty. Each type in \texttt{DISPATCHES} shall be a valid dispatch type.

**Effect:** Define a type named \texttt{NAME}, which contains:
- an inner type alias \texttt{overload\_types} defined as the aggregate of every \texttt{overload\_types} of each type \texttt{D} that has this type alias defined in \texttt{DISPATCHES}, and
- various overloads of \texttt{operator()} that forward the calls to each dispatch type \texttt{D} in \texttt{DISPATCHES}. 

33


```c
#define PRO_MAKE_DISPATCH_PACK(...) see below
   Syntax: PRO_MAKE_DISPATCH_PACK(DISPATCHES...)
   Effect: Equivalent to std::tuple<DISPATCHES...>.

#define PRO_DEF_FACADE(NAME, ...) see below
   Syntax:
   PRO_DEF_FACADE(
       NAME,
       [optional] DISPATCH_PACK = std::tuple<>,
       [optional] CONSTRAINTS = std::relocatable_ptr_constraints,
       [optional] REFLECTION_TYPE = void)
   Constraints: NAME shall not be defined in the context. DISPATCH_PACK shall be a dispatch pack, which could be a dispatch type or a tuple of various dispatch packs. CONSTRAINTS shall be a value of std::proxiable_ptr_constraints. REFLECTION_TYPE shall be void or a trivially copyable type that constructible from std::in_place_type_t<P> at compile-time.
   Effect: Define a type named NAME, which contains:
   - an inner type alias dispatch_types defined as the flattered tuple of DISPATCH_PACK, and
   - a compile-time constant constraints defined as the value of CONSTRAINTS, and
   - an inner type alias reflection_type defined as REFLECTION_TYPE.
```