Proxy: A Polymorphic Programming Library

Document number: P0957R7
Date: 2022-04-19
Project: Programming Language C++
Audience: LEWGI, LEWG
Authors: Mingxin Wang
Reply-to: Mingxin Wang <mingxwa@microsoft.com>

Table of Contents

1 History

1.1 Changes from P0957R6

- Replaced **`proxy::type()`** and **`proxy::cast()`** with **`proxy::reflect()`** for general-purpose static reflection.
- Removed the specifications of **`bad_proxy_cast`**.
- Revised the exception specifications of **`proxy::operator=`** overloads.
- Revised the specifications of named requirements **`BasicFacade`**.

1.2 Changes from P0957R5

- Redesigned the syntax of dispatch declaration to prevent users from ODR violation.
- Revised the specifications of **`dispatch`** and the overload of **`swap`**.
- Revised the specification of named requirements **`BasicDispatch`** and **`Dispatch`**.

1.3 Changes from P0957R4

- Renamed the proposed design from "PFA" into "proxy".

- Redesigned the "facade" from a language feature to a library feature per feedback from SG7;
- Replaced the abstraction of "Addresser" with C++ pointers.
- Revise the specifications of **`proxy`**.
- Added specifications of **`constraint_level`**, **`dispatch`**, **`facade`**, **`bad_proxy_cast`**, **`proxiable`**, **`make_proxy`**, **`swap`**.
- Removed the specifications of **`qualification_type`**, **`reference_type`**, **`add_qualification_t`**, **`qualification_of_v`**, **`add_reference_t`**, **`erased_reference`**, **`erased_value_selector`**, **`value_addresser`**, **`reference_addresser`**, **`proxy_meta`**, **`value_proxy`**, **`reference_proxy`**.

1.4 Changes from P0957R3

- Remove dependency from the concept of "Sink Argument" proposal [P1648R2];
- Remove support for allocators due to ambiguous semantics;
- Replace the member function **`assign`** with emplace to stay in the naming style with **`std::any`**;
- Remove the class template **`null_value_addresser_error`** per feedback from EWGI.

1.5 Changes from P0957R2

- Remove dependency from the concept of "Memory Allocator" proposal [P1172R1];
- Remove inheritance hierarchy among different instantiation of the class template **`proxy_meta`**;
- Remove convertibility among different instantiation of the class template **`reference_addresser`** and **`proxy`**;
- Remove the disambiguation tag **`delegated_tag_t`** and its value **`delegated_tag`**;
- Remove the class template **`allocated_value`**;
- Change the semantics of the delegated assignment reflecting to the assignment of the addresser, rather than the **`assign`** expression;
- Remove the **`assign`** member function from the class template **`reference_addresser`**.

1.6 Changes from P0957R1

- Reorganize the motivation part;
- Split **`static_addresser`** into **`value_addresser`** and **`reference_addresser`**;
- Add support for the "Extending Argument" [P1648R0], "Memory Allocator" [P1172R1] and configurable SOO for value semantics polymorphism;
- Add support for reference semantics in facade definitions;
- Add qualification type and reference type enumerations and corresponding type traits;
- Add the concept for "Erased Handles";
- Add exception support for value addresser;
- Change the semantics for the **`Addresser`** requirements;
- Revise the semantics for the **`proxy`**.

1.7 Changes from P0957R0

- Replace the "class template" in the declaration of the proxy with a "class";
- Remove the class template **`shared_addresser`** temporarily;
- Replace the class template **`direct_addresser`** and the class template **`unique_addresser`** with a uniform class template **`static_addresser`**;
- Replace the type aliases **`direct_proxy`**, **`unique_proxy`** and **`shared_proxy`** with a uniform alias **`static_proxy`**;
- Add support for "volatile" semantics.

2 Introduction

Since there are architecting and performance limitations in existing mechanisms of polymorphism, the "proxy" is proposed as a generic, extendable, and efficient template library solution of polymorphic programming. The "proxy" combines the idea of OOP (Object-oriented Programming) and FP (Functional Programming). Meanwhile, eliminating some of their known defects. Compared with traditional OOP, the "proxy" can largely replace the existing "virtual mechanism" and have no intrusion on existing code or runtime memory layout, without reducing performance. Compared with FP, the "proxy" is not only applicable to single dimensional requirements, but also can be applied to multi-dimensional ones, and could carry richer semantics.

With the template meta programming mechanism, the "proxy" is well-compatible with the C++ programming language and makes C++ easier to use. The "proxy" can be applied in almost every case that relates to virtual functions more elegantly. Components defined in the standard that related to polymorphism can be easily implemented with the "proxy", e.g., **`std::function`** and **`std::any`**.

The rest of the paper is organized as follows: section 3 illustrates the motivation and scope of the "proxy"; section 4 includes the pivotal decisions in the design; section 5 illustrates the technical specification; the last sections summarize the paper.

3 Motivation and scope

Polymorphism is widely required in large-scale programming to decouple components and increase extendibility at a cost of reducing runtime performance. 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 implantation status of the proposed library, the limitations in inheritance-based polymorphism with concrete system design requirements and how the proposed library could help.

3.1 Implementation status

As proof of concept, we have **fully** implemented the technical specifications as a single-header template library, meeting the C++20 standard. The implementation could be found here. The sample code, including the implementation of the motivating examples in later sections, could be found here.

As we tested, the implementation compiles with the latest releases of gcc, clang and MSVC, as the language standard is set to C++20. We did not notice a bug when testing with gcc or MSVC, but clang will fail to compile if the **`minimum_destructibility`** is set to **`constraint_level::trivial`** in a facade definition. The root cause of this failure is that the implementation requires the language feature defined in P0848R3: Conditionally Trivial Special Member Functions, but it has not been implemented in clang, according to its documentation, by the time this paper was written.

3.2 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; its area is always zero.

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

3.2.1 Architecting with inheritance-based polymorphism

With the keywork **`virtual`**, a base class could be defined:

```
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);
```

3.2.2 Architecting with the "proxy"

To define an abstraction of "drawable", we need to define the dispatch "Draw" as a type with the following syntax:

```
struct Draw : std::dispatch<void()> {
  template <class T>
  void operator()(const T& self) { self.Draw(); }
};
```

`Draw` is defined as a "dispatch", which is a callable type tagged with the signature in absence of the operand. **`std::dispatch`** is one of the proposed class templates to help define polymorphic expressions. After defining **`Draw`**, the next step is to define the "facade" with the following syntax:

```
struct FDrawable : std::facade<Draw> {};
```

`FDrawable` is defined as a "facade", which is another empty type serves at compile-time. **`std::facade`** is another proposed class template to help specify the proxy.

The required 3 types could be implemented as normal types without any virtual function or inheritance:

```
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<FDrawable> p);
```

`std::proxy` is another proposed class template that implements runtime polymorphism. It could be specified by any well-formed facade type like `FDrawable`. 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()`.

3.3 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()`?

3.3.1 Inheritance-based polymorphism

For inheritance-based polymorphism, based on the design in 3.2.1, all the base and derived classes need to be updated:

1. Another new pure virtual function needs to be added in the base class:

```
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:

```
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; }
};
```

3.3.2 The "proxy"

For the "proxy", based on the design in 3.2.2, 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":

```
struct Area : std::dispatch<double()> {
  template <class T>
  double operator()(const T& self) { return self.Area(); }
};
struct FDrawable : std::facade<Draw, Area> {};
```

3.3.3 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:

```
struct at : std::dispatch<std::string(int)> {
  template <class T>
  auto operator()(T& self, int key) { return self.at(key); }
};
struct FResourceDictionary : std::facade<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.

3.4 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.

3.4.1 Inheritance-based polymorphism

For inheritance-based polymorphism, based on the design in 3.3.1, the new factory function could be designed as follows:

```
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>`**:

```
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:

```
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:

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

If we decided to change one API from **`std::unique_ptr`** into **`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.

3.4.2 The "proxy"

For the "proxy", based on the design in 3.3.2, we can define the factory function directly without further concern:

```
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:

```
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 rectangle_memory_pool;
        std::pmr::polymorphic_allocator<> alloc{&rectangle_memory_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 existing code is needed.

3.4.3 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.

3.5 Conclusion

Prior research for 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 much 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 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*.

4 Considerations and design decisions

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, an abstraction of constraints is 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 has certain limitations in lifetime management comparing 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 3.4.2.

4.1.2 Constraints

To allow implementation balance between extendibility and performance, a set of constraints to a pointer is introduced, including maximum size, maximum alignment, minimum copyability, minimum relocatability and minimum destructibility. The term "relocatability" was introduced in P1144R5, "equivalent to a move and a destroy". This paper uses the term "relocatability" but does not depend on the technical specifications of P1144R5.

Constraints	Defaults
Maximum size	No less than the size of two pointers
Maximum alignment	No less than the alignment of a pointer
Minimum copyability	None
Minimum relocatability	Nothrow
Minimum destructibility	Nothrow

Table 1 – Default constraints of 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 defaults are listed in Table 1 to try to meet the requirements of various implementations of (smart) pointers. It is encouraged 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 3.4 without extra considerations in the code design or performance decrement.

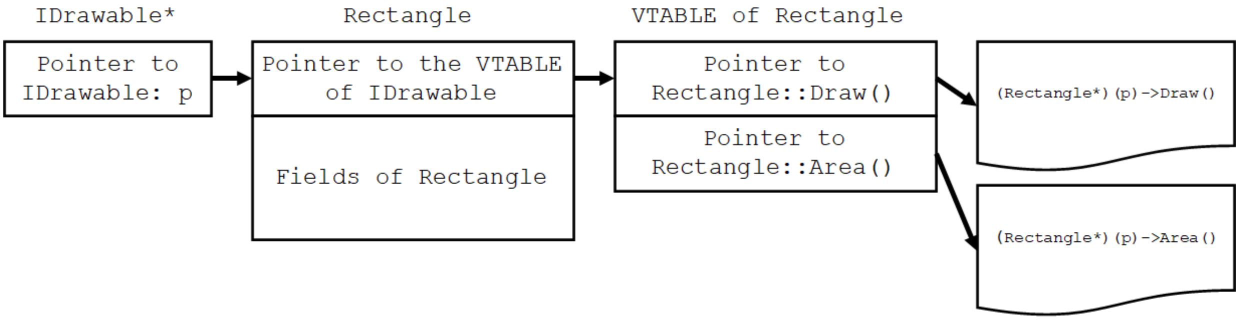

Figure 1 – Expected memory layout of inheritance-based polymorphism

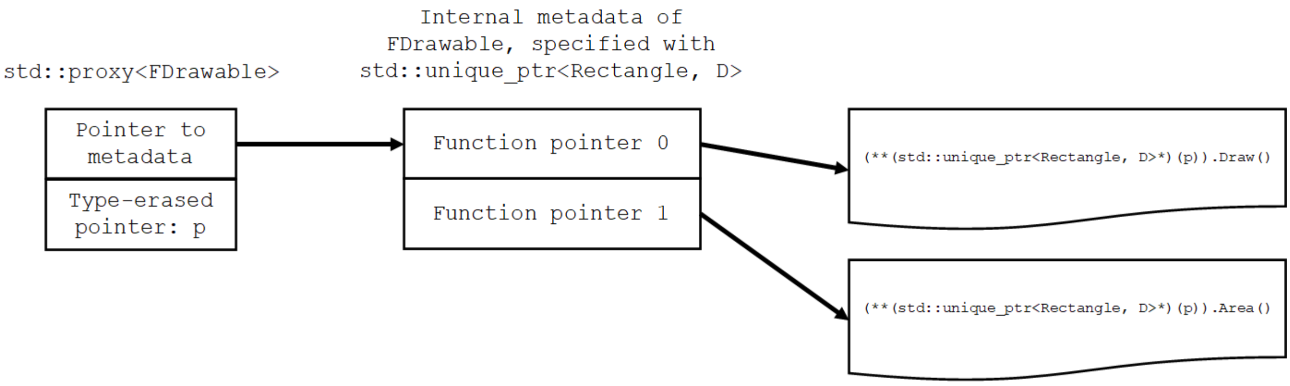

Figure 2 – Expected memory layout of std::proxy

Because of pointer semantics, the expected memory layout of **`std::proxy`** is also different from traditional inheritance. For instance, Figure 1 and Figure 2 shows their expected memory layout, respectively.

<table>
<tr><td></td><td>The "proxy"</td><td>Inheritance-based polymorphism</td></tr>
<tr><td>Abstraction</td><td><pre>struct Draw : std::dispatch<void()> {
  template <class T>
  void operator()(const T& self) { self.Draw(); }
};
struct Area : std::dispatch<double()> {
  template <class T>
  double operator()(const T& self) { return self.Area(); }
};
struct FDrawable : std::facade<Draw, Area> {};</pre></td><td><pre>struct IDrawable {
  virtual void Draw() const = 0;
  virtual double Area() const = 0;
  virtual ~IDrawable() {}
};</pre></td></tr>
<tr><td>Implementation</td><td><pre>class Rectangle {
 public:
  void Draw() const {
    printf("{Rectangle: width = %f, height = %f}", width_, height_);
  }
  double Area() const { return width_ * height_; }

 private:
  double width_;
  double height_;
};</pre></td><td><pre>class Rectangle : public IDrawable {
 public:
  void Draw() const override {
    printf("{Rectangle: width = %f, height = %f}", width_, height_);
  }
  double Area() const override
      { return width_ * height_; }

 private:
  double width_;
  double height_;
};</pre></td></tr>
<tr><td>Invocation</td><td><pre>void DoSomethingWithDrawable(std::proxy<FDrawable> p) {
  p.invoke<op::Draw>();
}</pre></td><td><pre>void DoSomethingWithDrawable(std::unique_ptr<IDrawable> p) {
  p->Draw();
}</pre></td></tr>
</table>

Table 2 – Sample code to compile

Processor architecture	Compiler family	Version	Compiler flags
x86-64 (AMD64)	clang	13.0.0	-std=c++20 -O3
ARM64	gcc	11.2	-std=c++20 -O3
RISC-V RV64	clang	13.0.0	-std=c++20 -O3

Table 3 – Sample compiler configurations

To evaluate the quality of code generation, we tried to compile the "Drawable" example from section 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 2, the sample compiler configurations for different processor architectures are listed in Table 3.

<table>
<tr><td></td><td>The "proxy"</td><td>Inheritance-based polymorphism</td></tr>
<tr><td>Library side</td><td><pre>mov   rax, qword ptr [rdi]
add   rdi, 8
jmp   qword ptr [rax + 24]</pre></td><td><pre>mov   rdi, qword ptr [rdi]
mov   rax, qword ptr [rdi]
jmp   qword ptr [rax]</pre></td></tr>
<tr><td>Client side</td><td><pre>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</pre></td><td><pre>movsd   xmm0, qword ptr [rdi + 8]
movsd   xmm1, qword ptr [rdi + 16]
mov   edi, offset .L.str
mov   al, 2
jmp   printf</pre></td></tr>
</table>

Table 4 – Generated code from clang 13.0.0 (x86-64)

	The "proxy"	Inheritance-based polymorphism

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

Client side	`mov x1, x0`
	`adrp x0, .LC3`
	`add x0, x0, :lo12:.LC3`
	`ldr d0, [x1]`
	`b printf`
		`mov x1, x0`
		`adrp x2, .LC0`
		`add x0, x2, :lo12:.LC0`
		`ldp d0, d1, [x1, 8]`
		`b printf`

Table 5 – Generated code from gcc 11.2 (ARM64)

	The "proxy"	Inheritance-based polymorphism

Library side	`ld a1, 0(a0)`
	`ld a5, 24(a1)`
	`addi a0, a0, 8`
	`jr a5`
		`ld a0, 0(a0)`
		`ld a1, 0(a0)`
		`ld a5, 0(a1)`
		`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`
		`ld a2, 16(a0)`
		`ld a1, 8(a0)`
		`lui a0, %hi(.L.str)`
		`addi a0, a0, %lo(.L.str)`
		`tail printf`

Table 6 – 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 4, Table 5 and Table 6. 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 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]". In the proposed library, the "facade" is a compile-time tag type that helps specify a proxy; the "proxy" could represent a pointer of different types and performs runtime polymorphism.

4.2.1 Abstraction of "facade"

Before revision 5 of this paper, the "facade" was proposed as a core language feature, but there was no consensus in the committee to "have a core language mechanism, such as "facade" for expressing a type-erased interface", per straw poll in Belfast. From revision 5, the "Facade" is proposed as named requirements of type to specify the proxy with required metadata.

The dispatches are also designed as named requirements. To support reuse of declaration of expression sets, like inheritance of virtual base classes, the "facade" allows combination of different dispatches with **std::tuple**, while duplication is allowed. For example,

```
struct D1; struct D2; struct D3;
struct FA : std::facade<D1, D2, D3> {};
struct FB : std::facade<D1, std::tuple<D3, D2>> {};
struct FC : std::facade<std::tuple<D1, D2, D3>, D1, std::tuple<D2, D3>> {};
```

As demonstrated earlier, class template **std::facade** and **std::dispatch** are proposed facilities to simplify the syntax to define well-formed types meeting the proposed requirements. In the sample code above, given **D1**, **D2** and **D3** are well-formed dispatch types, **FA**, **FB** and **FC** are equivalent. This allows "diamond inheritance" of abstraction without any syntax ambiguity, coding techniques like "virtual inheritance", or runtime overhead.

Dispatches are not limited to member functions, but all valid expressions in C++. For example, we can define an **FIterable** to add polymorphism to the global function template **for_each** on any container:

```
template <class T> struct Call;
struct Call<R(Args...)> : std::dispatch<R(Args&&...)> {
  template <class T>
  auto operator()(T& self, Args&&... args)
      { return self(std::forward<Args>(args)...); }
};
template <class T>
struct FCallable : std::facade<Call<T>> {};

template <class T>
struct ForEach : std::dispatch<void(std::proxy<FCallable<void(T&)>>)> {
  template <class U>
  void operator()(U& self, std::proxy<FCallable<void(T&)>>&& func) {
    std::ranges::for_each(self, [&func](T& value) { func.invoke(value); });
  }
};
template <class T>
struct FIterable : std::facade<ForEach<T>> {};
```

With the definition of **FIterable**, the following library function implementation is well-formed:

```
void MyPrintLibrary(std::proxy<FIterable<int>> p) {
  auto f = [](double value) { printf("%f", value); };
```

```
  p.invoke(&f);
  puts("");
}
```

While the caller side only need to provide any pointer to a well-formed container type without considering any polymorphic use in the implementation of the library, e.g.:

```
std::forward_list<int> a{1, 2, 3, 4, 5};
std::deque<int> b{6, 7, 8, 9, 10};
MyPrintLibrary(&a);
MyPrintLibrary(&b);
```

To support recursive declaration of a facade, i.e., using a facade while declaring dispatch with the name of the facade, the "BasicFacade" requirements is proposed. It is weaker than the "Facade" requirements, allowing underlying dispatches to be incomplete type. For example,

```
struct Self;
struct Print : std::dispatch<void()> {
  template <class T>
  void operator()(T& self) { std::cout << self << std::endl; }
};

struct FPrintable : std::facade<Self, Print> {};

struct Self : std::dispatch<std::proxy<FPrintable>(std::proxy<FPrintable>)> {
  template <class T>
  auto operator()(T& self, std::proxy<FPrintable> p) { return std::move(p); }
};
```

The implementation of **`Self`** could be delayed, before the specified proxy, i.e., **`std::proxy<FPrint>`** is initialized with a concrete pointer.

4.2.2 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 3.3.2, is not copy constructible, because the default copyability constraints to a pointer is "None". However, user can specify different constraint level if needed, e.g.,

```
struct MyFacade : std::facade</* Omitted */> {
  static constexpr std::constraint_level minimum_copyability =
      std::constraint_level::nontrivial;
};
```

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.

4.2.3 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`**.

4.2.4 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`**.

Before revision 7 of this paper, **`std::proxy`** supports and only supports acquiring the corresponding **`std::type_info`** of a given type, similar to **`std::function::target_type`** of **`std::function`** and **`std::any_cast`** of **`std::any`**. However, **`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 TS 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. It's off by default, and theoretically won't impact runtime performance other than the target binary size if turned on. Here is an example to reflect the given types to **`MyReflectionInfo`**:

```
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_;
};

struct MyFacade : std::facade</* Omitted */> {
  using reflection_type = 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());
```

4.3 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.

4.3.1 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":

```
template <
  typename Concept,
  typename Storage = dyno::remote_storage,
  typename VTablePolicy = dyno::vtable<dyno::remote<dyno::everything>>
>
struct 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 limitation makes it difficult to have different lifetime management strategies at runtime without additional overhead. The "simple factory" mentioned in 3.4 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 limitation.

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:

```
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.

4.3.2 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:

```
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 **`some_class_template<SomeConcept>`**, 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. Comparing 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.

4.4 Impact on the standard

Because the components defined in the standard that related to polymorphism can be easily implemented with the "proxy" without performance loss, the following features in the standard may gradually be deprecated in the future:

1. Virtual functions (except in some cases requiring casting based on inheritance hierarchy, e.g., exception handling);
2. Class template **`std::function`** and related components, e.g. **`std::bad_function_call`**;
3. Class **`std::any`** and related components, e.g. **`std::make_any`**, **`std::any_cast`**;
4. Other related proposals, including P0288R9 (move_only_function), P0792R5 (function_ref: a non-owning reference to a Callable).

No language feature change is required.

5 Technical specifications

5.1 Header <proxy> synopsis

```
namespace std {
  enum class constraint_level { none, nontrivial, nothrow, trivial };

  template <class T> struct dispatch;  // not defined

  template <class R, class... Args>
    struct dispatch<R(Args...)>;

  template <class... Ds>
    struct facade;

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

  template <class F> requires(see below)
    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);
}
```

5.2 Facade

5.2.1 Requirements

5.2.1.1 `BasicDispatch` Requirements

A type **`D`** meets the **`BasicDispatch`** requirements if D is default-constructible and the following expressions are well-formed and have the specific semantics:

typename D::return_type

Note: This type indicates the return type of the dispatch.

typename D::argument_types

Note: This type shall be an instantiation of **std::tuple**, indicating the argument types of the dispatch in the given order.

5.2.1.2 **Dispatch** Requirements

A type **D** meets the **Dispatch** requirements of type **T** if it meets the **BasicDispatch** requirements and the following expressions are well-formed and have the specific semantics (**d** denotes a value of **D**, **v** denotes a value of type **T**, **args...** denotes values of argument types defined by **typename D::argument_types**):

std::forward<D>(d)(std::forward<T>(v), std::forward<Args>(args)...)

Note: The return value shall be convertible to **typename D::return_type**.

5.2.1.3 **BasicFacade** Requirements

A type **F** meets the **BasicFacade** requirements if the following expressions are well-formed and have the specific semantics:

typename F::dispatch_types

Note: This type indicates the dispatch types and is not required to be a complete type.

typename F::reflection_type

Note: This type indicates compile-time reflection.

F::maximum_size

Note: This expression shall be a constant expression, convertible to **size_t**, indicating the allowed maximum size of pointers to instantiate **std::proxy<F>**.

F::maximum_alignment

Note: This expression shall be a constant expression, convertible to **size_t**, indicating the allowed maximum alignment of pointers to instantiate **std::proxy<F>**.

F::minimum_copyability

Note: This expression shall be a constant expression, convertible to **constraint_level**, indicating the minimum copyability requirements of pointers to instantiate **std::proxy<F>**.

F::minimum_relocatability

Note: This expression shall be a constant expression, convertible to **constraint_level**, indicating the minimum relocatability requirements of pointers to instantiate **std::proxy<F>**.

`F::minimum_destructibility`

Note: This expression shall be a constant expression, convertible to **`constraint_level`**, indicating the minimum destructibility requirements of pointers to instantiate **`std::proxy<F>`**.

5.2.1.4 `Facade` Requirements

A type **`F`** meets the **`Facade`** requirements if it meets the **`BasicFacade`** requirements and the following expressions are well-formed and have the specific semantics:

`typename F::dispatch_types`

Note: This type indicates the dispatch set, organized by **`std::tuple`**. Specifically, a dispatch set could be

- a complete type meeting the **`BasicDispatch`** requirements, or
- an instantiation of **`std::tuple`** of any number of dispatch sets.

5.2.2 Help classes

5.2.2.1 Class template `dispatch`

```
namespace std {
  template <class T> struct dispatch;  // not defined

  template <class R, class... Args>
    struct dispatch<R(Args...)> {
      using return_type = R;
      using argument_types = tuple<Args...>;
  };
}
```

5.2.2.2 Class template `facade`

```
namespace std {
  template <class... Ds>
    struct facade {
      using dispatch_types = tuple<Ds...>;
      static constexpr size_t maximum_size = see below;
      static constexpr size_t maximum_alignment = see below;
      static constexpr constraint_level minimum_copyability =
          constraint_level::none;
      static constexpr constraint_level minimum_relocatability =
          constraint_level::nothrow;
```

```
    static constexpr constraint_level minimum_destructibility =
        constraint_level::nothrow;
    facade() = delete;
  };
}
```

The value of **maximum_size** shall be greater than or equal to **(2 * sizeof(void*))**. The value of **maximum_alignment** shall be greater than or equal to **alignof(void(*))**. It is encouraged to define their value large enough to satisfy the implementation of **std::unique_ptr** of any type with default deleters, **std::unique_ptr** of any type with any one-pointer-size of deleters and **std::shared_ptr** of any type.

5.3Proxy

5.3.1 Concept `proxiable`

```
namespace std {
  template <class P, class F>
    concept proxiable = see below;
}
```

A pointer type **P** is proxiable with **F** (**p** denotes a value of **P**) if

- Expression ***p** is well-formed, and
- **F** meets the **Facade** requirements, and
- **sizeof(P) <= F::maximum_size** is **true**, and
- **alignof(P) <= F::maximum_alignment** is **true**, and
- **P** meets the minimum copyability requirements defined by **F::minimum_copyability**, and
- **P** meets the minimum relocatability requirements defined by **F::minimum_relocatability**, and
- **P** meets the minimum destructibility requirements defined by **F::minimum_destructibility**, and
- For each dispatch type **D** defined by **typename F::dispatch_types**, **D** meets the **Dispatch** requirements of type **decltype(*p)**, and
- If **typename F::reflection_type** is not **void**, it shall be constructible from **std::in_place_type_t<P>** in a constant expression.

5.3.2 Class template `proxy`

5.3.2.1 General

```
namespace std {
  template <class F> requires(see below)
  class proxy {
  public:
```

```
    proxy() noexcept;
    proxy(nullptr_t) noexcept;
    proxy(const proxy& rhs) noexcept(see below) requires(see below);
    proxy(proxy&& rhs) noexcept(see below) requires(see below);
    template <class P>
      proxy(P&& ptr) noexcept(see below) requires(see below);
    template <class P, class... Args>
      explicit proxy(in_place_type_t<P>, Args&&... args)
          noexcept(see below) requires(see below);
    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);
    proxy& operator=(nullptr_t) noexcept(see below) requires(see below);
    proxy& operator=(const proxy& rhs) noexcept(see below) requires(see below);
    proxy& operator=(proxy&& rhs) noexcept(see below) requires(see below);
    template <class P>
      proxy& operator=(P&& ptr) noexcept(see below) requires(see below);
    ~proxy() noexcept(see below) requires(see below);

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

    template <class D = see below, class... Args>
      see below invoke(Args&&... args) requires(see below);
  };
}
```

As the constraint of the class template, the expression inside **requires** is equivalent to that **F** meets the **BasicFacade** requirements.

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:

Name	Value
`template <class P, class... Args>`	`conditional_t<proxiable<P, F>,`

HasNothrowPolyConstructor<P, Args...>	is_nothrow_constructible<P, Args...>, false_type>::value
template <class P, class... Args> HasPolyConstructor<P, Args...>	conditional_t<proxiable<P, F>, is_constructible<P, Args...>, false_type>::value
HasTrivialCopyConstructor	F::minimum_copyability == constraint_level::trivial
HasNothrowCopyConstructor	F::minimum_copyability >= constraint_level::nothrow
HasCopyConstructor	F::minimum_copyability >= constraint_level::nontrivial
HasNothrowMoveConstructor	F::minimum_relocatability >= constraint_level::nothrow
HasMoveConstructor	F::minimum_relocatability >= constraint_level::nontrivial
HasTrivialDestructor	F::minimum_destructibility == constraint_level::trivial
HasNothrowDestructor	F::minimum_destructibility >= constraint_level::nothrow
HasDestructor	F::minimum_destructibility >= constraint_level::nontrivial
template <class P, class... Args> HasNothrowPolyAssignment	HasNothrowPolyConstructor<P, Args...> && HasNothrowDestructor
template <class P, class... Args> HasPolyAssignment	HasPolyConstructor<P, Args...> && HasDestructor
HasTrivialCopyAssignment	HasTrivialCopyConstructor && HasTrivialDestructor
HasNothrowCopyAssignment	HasNothrowCopyConstructor && HasNothrowDestructor
HasCopyAssignment	HasNothrowCopyAssignment || (HasCopyConstructor && HasMoveConstructor && HasDestructor)
HasNothrowMoveAssignment	HasNothrowMoveConstructor && HasNothrowDestructor
HasMoveAssignment	HasMoveConstructor && HasDestructor

5.3.2.2 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 **rhs.has_value()** is **false**, constructs an object that has no value. Otherwise, equivalent to **proxy(in_place_type<P>, rhs.cast<P>())** where **P** is the type of the contained value of **rhs**.

Postconditions: **has_value() == rhs.has_value()**.

Throws: Any exception thrown by the selected constructor of **P**.

Remarks: The expression inside **noexcept** is equivalent to **HasNothrowCopyConstructor**. Specifically,

- if the constraints are not satisfied, the constructor is deleted, or
- if **HasTrivialCopyConstructor** is **true**, the constructor is trivial.

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

Constraints: The expression inside **requires** is equivalent to **HasMoveConstructor**.

Effects: If **rhs.has_value()** is **false**, constructs an object that has no value. Otherwise, equivalent to (**proxy(in_place_type<P>, move(rhs.cast<P>())), rhs.reset()**), where **P** is the type of the contained value of **rhs**.

Postconditions: **rhs** does not contain a value.

Throws: Any exception thrown by the selected constructor of **P**.

Remarks: The expression inside **noexcept** is equivalent to **HasNothrowMoveConstructor**. If the constraints are not satisfied, the constructor is deleted.

```
template <class P>
  proxy(P&& ptr) noexcept(see below) requires(see below);
```

Let **VP** be **decay_t<P>**.

Constraints: The expression inside **requires** is equivalent to **HasPolyConstructor<VP, P>**.

Effects: Initializes the contained value as if direct-initializing an object of type **VP** with **forward<P>(ptr)**.

Postconditions: ***this** contains a value of type **VP**.

Throws: Any exception thrown by the selected constructor of **VP**.

Remarks: The expression inside **noexcept** is equivalent to **HasNothrowPolyConstructor<VP, P>**.

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

Constraints: The expression inside **requires** is equivalent to **HasPolyConstructor<P, Args...>**.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type **P** with the arguments **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, Args...>**.

```
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, 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.

5.3.2.3 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(move(rhs)).swap(*this)`**.

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.

`template <class P>`
`  proxy& operator=(P&& ptr) noexcept(`*`see below`*`) requires(`*`see below`*`);`

Let **`VP`** be **`decay_t<P>`**.

Constraints: The expression inside **`requires`** is equivalent to **`HasPolyAssignment<VP, P>`**.

Effects: As if by **`proxy(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>`**.

`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.

`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.

5.3.2.4 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:

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	Swap the contained values of **`*this`** and **`rhs`** with a temporary storage. If an	Equivalent to (**`*this = move(rhs)`**); post condition is that **`*this`** contains a

	exception is thrown, each of `*this` and `rhs` is in a valid state with unspecified value.	value and `rhs` does not contain a value.
`rhs` does not contain a value	Equivalent to (`rhs = 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`.

5.3.2.5 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.

5.3.2.6 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.

5.3.2.7 Invocation

`template <class D = `*`see below`*`, class... Args>`
  `see below` `invoke(Args&&... args) 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 each of the argument type in `Args...` is convertible to the argument types defined by `typename D::argument_types`, respectively.

Preconditions: `*this` contains a value.

Effects: Equivalent to `return D{}(*cast<P>(), static_cast<_Args>(args)...)`, where `P` is the type of the contained value, `_Args...` are the argument types defined by 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 **`*this`** does not contain a value, the behavior is undefined.

5.3.3 Creation

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

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

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

- **`sizeof(T) <= F::maximum_size`** is **`true`**, and
- **`alignof(T) <= F::maximum_alignment`** is **`true`**, and
- **`T`** meets the minimum copyability requirements defined by **`F::minimum_copyability`**, and
- **`T`** meets the minimum relocatability requirements defined by **`F::minimum_relocatability`**, and
- **`T`** meets the minimum destructibility requirements defined by **`F::minimum_destructibility`**, and

```
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, forward<Args>(args)...)`**.

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

Effects: Equivalent to **`return make_proxy<F, decay_t<T>>(forward<T>(value))`**.

5.3.4 Specialized algorithms

```
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 of the library design and testing. 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.