Proxy: A Pointer-Semantics-Based Polymorphism Library

Document number:	P3086R3
Date:	2025-01-13
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 P3086R2

- Revised the Motivation section.
- Added 9 named requirements (*ProOverload*, *ProDispatch*, *ProBasicConvention*, *ProConvention*, *ProBasicReflection*, *ProReflection*, *ProBasicFacade*, *ProFacade*, *ProAccessible*).
- Added class template **`proxy_indirect_accessor`**.
- Changed the definition of **`proxy::invoke()`** and **`proxy::reflect()`** into free functions **`proxy_invoke()`** and **`proxy_reflect()`**.
- Added accessibility support to **`proxy`** (the *ProAccessible* requirements and function template **`access_proxy()`**).
- Added **`proxy::operator bool()`**, **`proxy::operator->()`** and **`proxy::operator*()`**.
- Changed the definition of **`std::swap(proxy, proxy)`** into a friend function.
- Removed the Appendix section.

1.2 Changes from P3086R1

As per review comments from LEWGI in Tokyo,

- Removed function template **`make_proxy`** from the proposed wording.
- Updated the wording of **`concept facade`**, allowing tuple-like types in the definition of a facade or dispatch.
- Revised the semantics of **`concept facade`** to allow fallbacks in the invocation of a dispatch.
- Moved the proposed location of the library from a new header to <memory>.
- Added a section for freestanding specifications in section 6.
- Added discussion comparing with P3019R6 in section 5.
- Added discussion of ordering and hash support in section 5.
- Added another section for open questions.
- In the appendix, added specification of another two helper macros **`PRO_DEF_MEMBER_DISPATCH_WITH_DEFAULT`** and **`PRO_DEF_FREE_DISPATCH_WITH_DEFAULT`**.

1.3 Changes from P3086R0

- Added support for **`noexcept`** in the abstraction model and updated the **`noexcept`** clause of **`proxy::invoke`** and **`proxy::operator()`**.
- Removed **`concept basic_facade`** and the constraints on the class template **`proxy`** to allow more potential optimizations in code generation.

2 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.
- Concepts **`facade`** and **`proxiable`**.

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 mandating runtime 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, some of the major changes are listed as follows:

1. The facilities to help defining **`dispatch`**es and **`facad`**es 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. The requirements of **`facade`** are also revised.
3. Per user feedback, overloads are split from the **`dispatch`** definition.
4. Per user feedback, **`proxy::invoke()`** was redesigned as **`proxy_invoke`** with better support for accessibility.
5. Added concept **`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.

3 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 **`function`**, **`move_only_function`**, **`function_ref`**, **`any`**, **`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.

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 Proxy is to eliminate the usability and performance limitations in traditional OOP and functional programming.

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.

3.1 Implementation status

As proof of concept, we have implemented technical specifications as a single-header template library. The implementation, including unit tests and benchmarks, 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. We also maintain a copy of the technical specifications online to facilitate navigation. Note that this paper is the first one of the series and scopped to the foundation of the library. Extensions like class template `basic_facade_builder` or class template `operator_dispatch` in the library are not included in this paper.

3.2 Non-intrusive

To take advantage of virtual functions to implement runtime polymorphism, a C++ type needs to inherit a base type. This is intrusive to the derived type, not only semantically, but also affects the memory layout, even if runtime polymorphism or RTTI is not used in a certain context. On the other hand, since virtual functions can only be member functions, only a part of C++ expressions can be made polymorphic by using virtual functions.

Here is an example that makes any "formattable" type polymorphic by using the Proxy library, demonstrating its capability to make an arbiturary **`print()`** call apply to an abstract binding without using virtual functions (live demo). This is not implementable with the inheritance-based approach because a "formattable" type, like an **`int`** or **`tuple`**, may not inherit from any base type. Note that **`facade_builder`** is not included in this paper, but **`FFormattable`** below meets the **`ProFacade`** requirements which is introduced in this paper. **`make_proxy`** is introduced in a separate paper P3401R0.

```
// Define a "facade" that supports "format"
struct FFormattable : facade_builder
    ::support_format
    ::build {};

proxy<FFormattable> p = make_proxy<FFormattable>(1024);  // Make an
int polymorphic, even though int does not inherit anything
print("{:#06x}", *p);  // Prints "0x0400"
```

3.3 Well-managed

The library provides a GC-like capability that manages the lifetimes of different objects efficiently without the need for an actual garbage collector.

Here is an example of "simple factory". Suppose there are 3 "drawable" entities in a system: rectangle, circle, and point. Specifically.

- Rectangles have width, height, transparency, and
- Circles have radius, transparency, and
- Points do not have any property.

3.3.1 Inheritance-based approach

With the **`virtual`** keyword, 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);
```

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

The factory function could be designed as follows:

```
IDrawable* MakeDrawableFromCommand(const 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 **`unique_ptr<IDrawable>`**:

```
unique_ptr<IDrawable> MakeDrawableFromCommand(const 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 ~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 **`unique_ptr<IDrawable>`** is not copyable, this may require further API change, for example, using **`shared_ptr`** instead:

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

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

3.3.2 The Proxy library

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

Here is a sample definition:

```
PRO_DEF_MEM_DISPATCH(MemDraw, Draw);
struct FDrawable : facade_builder
    ::add_convention<MemDraw, void() const>
    ::build {};
```

Again, [`facade_builder`] and [`PRO_DEF_MEM_DISPATCH`] are not in the scope of this paper, but **`FDrawable`** meets the **`ProFacade`** requirements which is introduced in this paper.

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);
};
class Circle {
 public:
  void Draw() const;
  void SetRadius(double radius);
  void SetTransparency(double transparency);
};
class Point {
 public:
  void Draw() const;
};
```

We can define the factory function directly without further concern in lifetime management:

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

In the implementation, **`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, **`Circle`** may be small enough to be embedded into the proxy (aka. SBO, small buffer optimization), the value of **`Point`** could be cached throughout the lifetime of the program (live demo):

```
proxy<FDrawable> MakeDrawableFromCommand(const string& s) {
```

```
  vector<string> parsed = ParseCommand(s);
  if (!parsed.empty()) {
    if (parsed[0] == "Rectangle") {
      if (parsed.size() == 3u) {
        static pmr::unsynchronized_pool_resource
rectangle_memory_pool;
        pmr::polymorphic_allocator<> alloc{&rectangle_memory_pool};
        return allocate_proxy<FDrawable, Rectangle>(
            alloc, stod(parsed[1]), stod(parsed[2]));
      }
    } else if (parsed[0] == "Circle") {
      if (parsed.size() == 2u) {
        Circle circle{stod(parsed[1])};
        return make_proxy<FDrawable>(circle);  // SBO may apply
      }
    } else if (parsed[0] == "Point") {
      if (parsed.size() == 1u) {
        static Point instance;  // global singleton
        return &instance;
      }
    }
  }
  throw runtime_error{"Invalid command"};
}
```

Note that **`make_proxy`** is introduced in a separate paper P3401R0 that can effectively avoid heap allocation when the underlying object is small.

3.3.3 Conclusion

Lifetime management with inheritance-based polymorphism is error-prone and inflexible, while 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.

3.4 Fast

To better understand the performance of the library, we designed 15 benchmarks against our implementation, tested in four different environments, and automated them in our GitHub pipeline to generate benchmarking reports for every code change. Everyone can download the reports and raw benchmarking data attached to each build. The numbers shown below were generated from a recent CI build.

3.4.1 Indirect Invocation

Both **`proxy`** objects and virtual functions can perform indirect invocations. However, since they have different semantics and memory layout, it should be interesting to see how they compare to each other.

Because **`make_proxy`** can effectively place a small object alongside metadata, the benchmarks are divided into two categories: invocation on small objects (4 bytes) and on large objects (48 bytes). By invoking 1,000,000 object of 100 different types, we got the first two rows of the report:

	MSVC on Windows Server 2022 (x64)	**GCC on Ubuntu 24.04 (x64)**	**Clang on Ubuntu 24.04 (x64)**	**Apple Clang on macOS 15 (ARM64)**
Indirect invocation on small objects via `proxy` **vs. virtual functions**	🟢 **`proxy`** is about **261.7% faster**	🟢 **`proxy`** is about **44.6% faster**	🟢 **`proxy`** is about **71.6% faster**	🟡 **`proxy`** is about **4.0% faster**
Indirect invocation on large objects via `proxy` **vs. virtual functions**	🟢 **`proxy`** is about **186.1% faster**	🟢 **`proxy`** is about **15.5% faster**	🟢 **`proxy`** is about **17.0% faster**	🟢 **`proxy`** is about **10.5% faster**

Table 1 – Indirect invocation benchmarking report

From the report, **`proxy`** is faster in all four environments, especially on Windows Server. This result is expected because the implementation of **`proxy`** directly stores the metadata of the underlying object, making it more cache friendly.

3.4.2 Lifetime Management

In many applications, lifetime management of various objects can become a performance hotspot compared to indirect invocations. We benchmarked this scenario by creating 600,000 small or large objects within a single **`vector`** (with reserved space).

Besides **`proxy`**, there are three typical standard options for storing arbitrary types: **`unique_ptr`**, **`shared_ptr`**, and **`any`**. **`variant`** is not included because it is essentially a tagged union and can only provide storage for a known set of types (though useful in data context management).

For small objects, **`proxy`** and **`any`** usually won't allocate additional storage. For large objects, **`proxy`** and **`shared_ptr`** offer allocator support (via **`allocate_proxy`** (P3401R0) and

allocate_shared) to improve performance, while there is no direct API to customize **unique_ptr** or **any**.

Here are the types we used in the benchmarks:

Small types	**Large types**
int	**array<char, 100>**
shared_ptr<int>	**array<string, 3>**
unique_lock<mutex>	**unique_lock<mutex> + void*[15]**

By comparing **proxy** with other solutions, we got the following numbers:

	MSVC on Windows Server 2022 (x64)	**GCC on Ubuntu 24.04 (x64)**	**Clang on Ubuntu 24.04 (x64)**	**Apple Clang on macOS 15 (ARM64)**
Basic lifetime management for small objects with proxy **vs.** unique_ptr	🟢 **proxy** is about **467.0% faster**	🟢 **proxy** is about **413.0% faster**	🟢 **proxy** is about **430.1% faster**	🟢 **proxy** is about **341.1% faster**
Basic lifetime management for small objects with proxy **vs.** shared_ptr **(without memory pool)**	🟢 **proxy** is about **639.2% faster**	🟢 **proxy** is about **509.3% faster**	🟢 **proxy** is about **492.5% faster**	🟢 **proxy** is about **484.2% faster**
Basic lifetime management for small objects with proxy **vs.** shared_ptr **(with memory pool)**	🟢 **proxy** is about **198.4% faster**	🟢 **proxy** is about **696.1% faster**	🟢 **proxy** is about **660.0% faster**	🟢 **proxy** is about **188.5% faster**
Basic lifetime management for small objects with proxy **vs.** any	🟢 **proxy** is about **55.3% faster**	🟢 **proxy** is about **311.0% faster**	🟢 **proxy** is about **323.0% faster**	🟢 **proxy** is about **18.3% faster**
Basic lifetime management for large objects with proxy **(without memory pool) vs.** unique_ptr	🟢 **proxy** is about **17.4% faster**	🟢 **proxy** is about **14.8% faster**	🟢 **proxy** is about **29.7% faster**	🔴 **proxy** is about **6.3% slower**
Basic lifetime management for large objects with proxy **(with memory pool) vs.** unique_ptr	🟢 **proxy** is about **283.6% faster**	🟢 **proxy** is about **109.6% faster**	🟢 **proxy** is about **204.6% faster**	🟢 **proxy** is about **88.6% faster**
Basic lifetime management for large objects with proxy **vs.** shared_ptr **(both without memory pool)**	🟢 **proxy** is about **29.2% faster**	🟢 **proxy** is about **6.4% faster**	🟢 **proxy** is about **6.5% faster**	🟡 **proxy** is about **4.8% faster**

Basic lifetime management for large objects with `proxy` vs. `shared_ptr` (both with memory pool)	🟢 **`proxy`** is about **10.8%** **faster**	🟢 **`proxy`** is about **9.9%** **faster**	🟢 **`proxy`** is about **8.3%** **faster**	🟢 **`proxy`** is about **53.2%** **faster**
Basic lifetime management for large objects with `proxy` (without memory pool) vs. `any`	🟢 **`proxy`** is about **13.4%** **faster**	🟡 **`proxy`** is about **1.3%** **slower**	🟡 **`proxy`** is about **0.9%** **faster**	🟢 **`proxy`** is about **9.5%** **faster**
Basic lifetime management for large objects with `proxy` (with memory pool) vs. `any`	🟢 **`proxy`** is about **270.7%** **faster**	🟢 **`proxy`** is about **80.1%** **faster**	🟢 **`proxy`** is about **136.9%** **faster**	🟢 **`proxy`** is about **120.4%** **faster**

Table 2 – Lifetime management benchmarking report

From the report:

- **`proxy`** is much faster than any other 3 when the underlying object is small or managed with memory pools.

- **`proxy`** is slightly slower than **`unique_ptr`** when the underlying object is large and not managed with a memory pool.

- The performance of **`any`** varies in different environments but is generally slower than **`proxy`**.

3.4.3 Conclusion

Although the test environments (GitHub-hosted runners) may differ from actual production environments, the test results show significant performance advantages of Proxy in both indirect invocations and lifetime management.

4 Impact on the Standard

For existing polymorphic wrappers in the standard, including **`function`**, **`move_only_function`**, **`polymorphic_allocator`** and **`any`**, **`proxy`** can facilitate implelemtation 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 **`operator()`** to implement similar function wrapper as **`std::function`** and **`std::move_only_function`** while supporting multiple overloads (live demo).

```
template <class... Overloads>
struct FMovableCallable : facade_builder
    ::add_convention<operator_dispatch<"()">, Overloads...>
    ::build {};
```

```
template <class... Overloads>
struct FCopyableCallable : facade_builder
    ::support_copy<constraint_level::nontrivial>
    ::add_facade<FMovableCallable<Overloads...>>
    ::build {};

// MyFunction has similar functionality as function,
// but supports multiple overloads
// MyMoveOnlyFunction has similar functionality as
// move_only_function but supports multiple overloads
template <class... Overloads>
using MyFunction = proxy<FMovableCallable<Overloads...>>;
template <class... Overloads>
using MyMoveOnlyFunction = proxy<FCopyableCallable<Overloads...>>;

int main() {
  auto f = [](auto&&... v) {
    printf("f() called. Args: ");
    ((cout << v << ":" << typeid(decltype(v)).name() << ", "), ...);
    puts("");
  };
  MyFunction<void(int)> p0{&f};
  (*p0)(123);  // Prints "f() called. Args: 123:i," (assuming GCC)
  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,"
}
```

5 Considerations and Design Decisions

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

1. Added some named requirements.
2. Simplified semantics of dispatches and facades.
3. Supported multiple overloads of a convention.
4. Added support for custom accessibility.

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

5.1 Pointer semantics

We decided to design 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.

5.1.1 Motivation

Currently, the standard polymorphic wrapper types, including **`function`** and **`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 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, **`function`** and **`any`** are not supported. Back to C++14, **`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, **`any`**, introduced in C++17, does not allow 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 3.3.2.

5.1.2 Constraints

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

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.

5.1.3 Implementation

Inheritance-based polymorphism or most of the standard polymorphic wrappers are 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 **`function`** or **`any`**, the lifetime of the stored values is bound to these polymorphic wrappers without allocator customization. These limitations make it difficult to implement requirements like 3.3 without extra considerations in the code design or performance decrement.

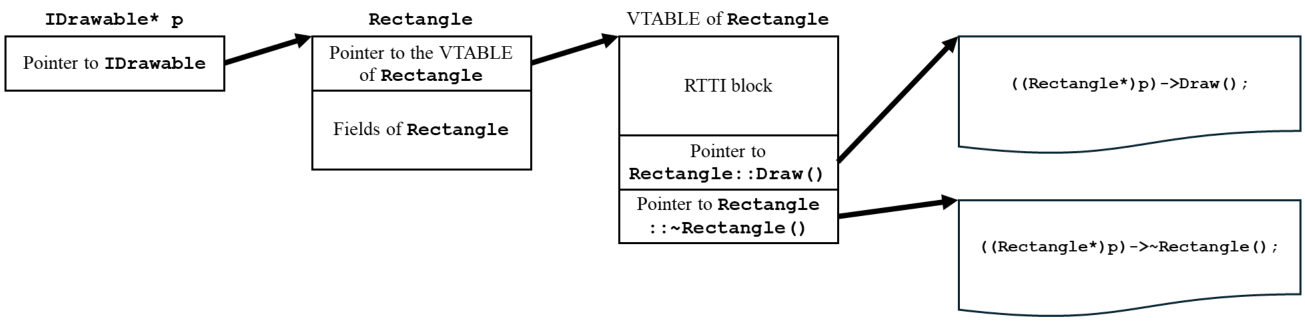

Figure 1 – Expected memory layout of **IDrawable*** (pointing to **Rectangle**)

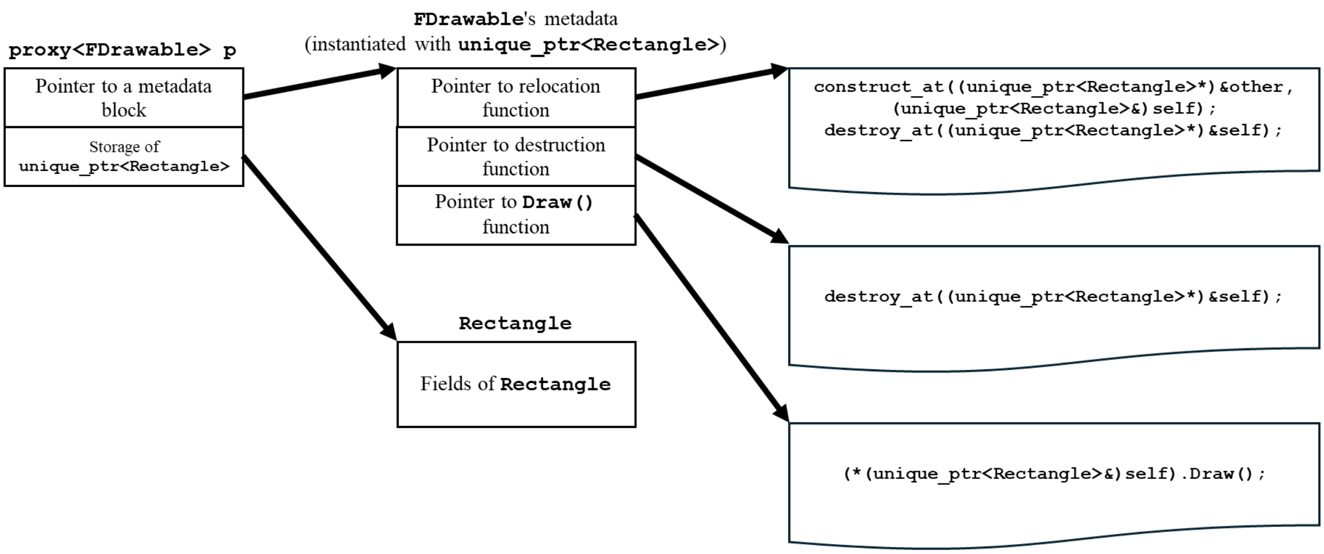

Figure 2 – Expected memory layout of **proxy<FDrawable>** (containing **unique_ptr<Rectangle>**)

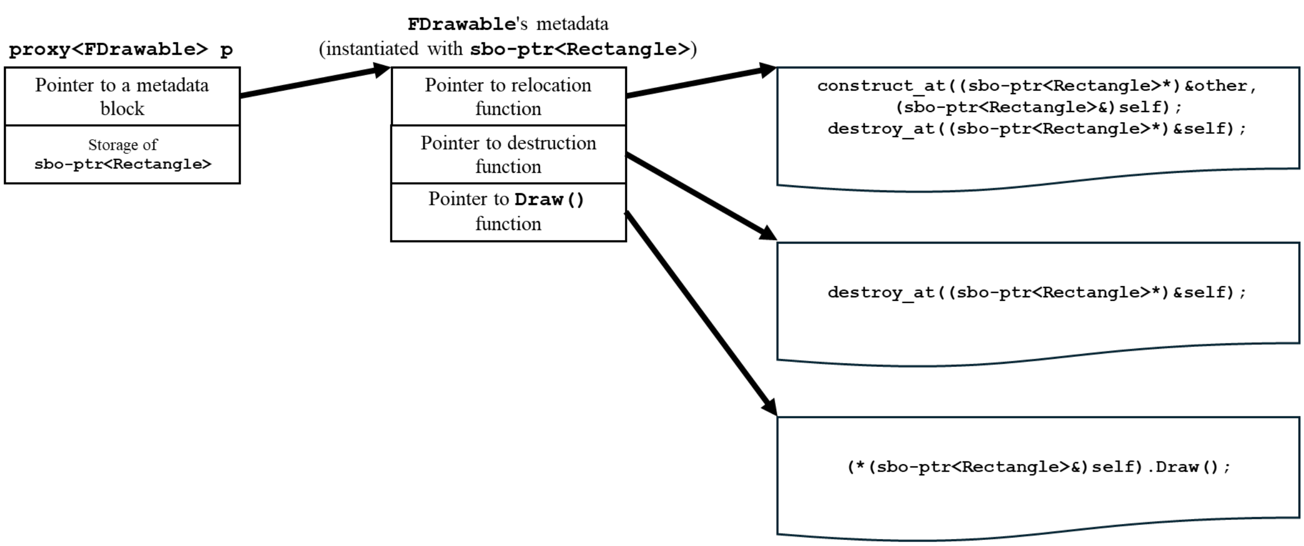

Figure 3 – Expected memory layout of **proxy<FDrawable>** (containing **sbo-ptr<Rectangle>**)

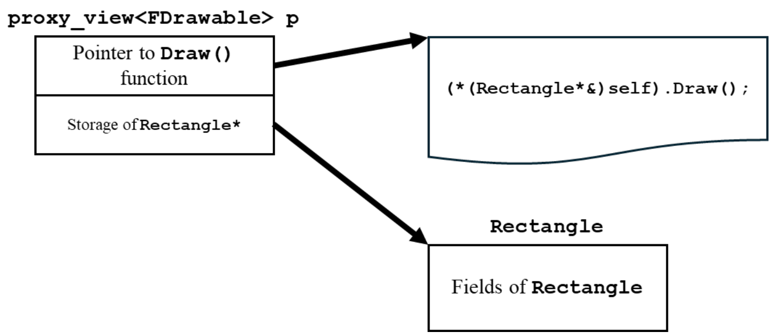

Figure 4 – Expected memory layout of **proxy_view<FDrawable>** (containing **Rectangle***)

Because of pointer semantics, the expected memory layout of **proxy** is also different from traditional inheritance. For instance, Figure 1 and Figure 2 shows their expected memory layout, respectively. The expected memory layout of **proxy<FDrawable>** is similar with [the implementation of move_only_function in libstdc++](), where the pointer of the actual object is dereferenced inside the virtual dispatch via **_S_access**.

In some cases where the object is small or the metadata is small, **proxy** is expected to embed the data within its footprint as shown in Figure 3 and Figure 4. These optimizations can further improve caching at runtime. (**sbo-ptr** was introduced in [P3401R0](), **proxy_view<F>** is an alias of **proxy<observer_facade<F>>**, both facilities are in the scope of this paper).

	Proxy	Inheritance-based polymorphism

Abstraction	PRO_DEF_MEM_DISPATCH(MemDraw, Draw);
	struct FDrawable : facade_builder
	::add_convention<MemDraw, void() const>
	::build {};
		struct IDrawable {
		virtual void Draw() const = 0;
		virtual ~IDrawable() {}
		};

Client	p->Draw(); // p is a value of proxy<FDrawable>	p->Draw(); // p is a value of IDrawable*

Table 3 – Sample code to compile

Processor architecture	Compiler family	Version	Compiler flags
x86-64 (AMD64)	Clang	19.1.0	-std=c++20 -O3 -DNDEBUG
ARM64	Clang	19.1.0	-std=c++20 -O3 -DNDEBUG
RISC-V RV64	Clang	19.1.0	-std=c++20 -O3 -DNDEBUG

Table 4 – Sample compiler configurations

To evaluate the quality of code generation, we tried to compile the "Drawable" example from section 3.3 with various compilers and compare the generated assembly between the sample implementation of Proxy and traditional inheritance-based polymorphism. Specifically, the sample code to compile is listed in Table 3, the sample compiler configurations for different processor architectures are listed in Table 4.

Proxy	Inheritance-based polymorphism

`mov rax, qword ptr [rdi]`
`add rdi, 8`
`jmp qword ptr [rax + 16]`
	`mov rdi, qword ptr [rdi]`
	`mov rax, qword ptr [rdi]`
	`jmp qword ptr [rax]`

Table 5 – Generated code from Clang 19.1.0 (x86-64)

Proxy	Inheritance-based polymorphism

`ldr x8, [x0], #8`
`ldr x1, [x8, #16]`
`br x1`
	`ldr x0, [x0]`
	`ldr x8, [x0]`
	`ldr x1, [x8]`
	`br x1`

Table 6 – Generated code from Clang 19.1.0 (ARMv8-A)

Proxy	Inheritance-based polymorphism

`ld a1, 0(a0)`
`ld a5, 16(a1)`
`addi a0, a0, 8`
`jr a5`
	`ld a0, 0(a0)`
	`ld a1, 0(a0)`
	`ld a5, 0(a1)`
	`jr a5`

Table 7 – Generated code from Clang 19.1.0 (RISC-V RV64)

Trying to compile the two pieces of sample code with 3 different compilers, the generated assembly are shown in Table 5, Table 6 and Table 7. 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 x86-64 and RISC-V RV64, **`proxy`** generates the same number of instructions from the client side, compared to inheritance-based polymorphism. With Clang ARMv8-A, **`proxy`** generates one instruction less. This may indicate **`proxy`** cannot be implemented with virtual functions, and won't surprising increase the binary size.

5.2 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. Specifically, the capability to handle different lifetime models and various expression forms (calling a member function is not the only expression allowed in C++).
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 with acceptable accessibility, and therefore no language feature should be expected for this part.

5.3 Proxy

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

5.3.1 Facade: Abstraction of Runtime Polymorphism

Although we are not proposing a syntax to define something like "interface", corresponding named requirements and 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. Conventions: How are the indirect invocations defined.
2. Reflections: What compile-time metadata to be carried to runtime.
3. Constraints: Specific constraints of applicable pointer types, as a compile-time value.

These requirements can be easily expressed with the type system of C++. A facade type models a compile-time tag to specify a proxy. The figure below shows the basic schema of a facade:

```
facade: The runtime abstraction spec
 ├─ convention_types: A tuple-like type
 │    └─ <element>: Each element describes a convention
 │         ├─ is_direct: A bool constant
 │         ├─ dispatch_type: A callable type
 │         ├─ overload_types: A tuple-like type
 │         │    └─ <element>: Each element describes an overload
 │         └─ accessor: An optional class template for accessibility
 ├─ reflection_types: A tuple-like type
 │    └─ <element>: Each element describes a reflection
 │         ├─ is_direct: A bool constant
 │         ├─ reflector_type: A reflector type
 │         └─ accessor: An optional class template for accessibility
 └─ constraints: A constant of type proxiable_ptr_constraints
      ├─ max_size: size_t
```

```
├─ max_align: size_t
├─ copyability: constraint_level
├─ relocatability: constraint_level
└─ destructibility: constraint_level
```

5.3.2 Reflection

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

As **`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 **`function`**, **`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 **`type_info`**. It becomes possible for general-purpose reflection to become zero-overhead in C++ polymorphism.

As a result, we decided to make **`proxy`** support general-purpose static reflection. Here is an example to make proxy support RTTI with library extension **`basic_facade_builder::support_rtti`** (not in the scope of this paper, but live demo is available):

```
struct RttiAware : facade_builder
    ::support_rtti
    ::build {};
```

Users may call **`proxy_typeid()`** to get the implementation-defined name of a type at runtime:

```
proxy<RttiAware> p;
puts(proxy_typeid(*p).name());  // Prints "v" (assuming GCC)
p = make_proxy<RttiAware>(123);
puts(proxy_typeid(*p).name());  // Prints "i"
puts(p.reflect().GetName());
```

5.3.3 Invocation fallbacks

Since P3086R0, we have received feature requests to support invocation fallbacks, specifically,

1. There is a need for APIs to interact with the underlying pointer types. One example would be creating a **`weak_ptr`** from a **`shared_ptr`** stored in a value of proxy.
2. For types that do not support certain semantics, there is a need for fallback to a default implementation with guarantee not to generate duplicate code before linking.

The semantics of **`facade`** has been updated to support such fallback. If a dispatch cannot be invoked with the dereferenced type of the contained value in the **`proxy`**, we fall back to the pointer itself without dereferencing it, and eventually fall back to a default implementation without the context of the **`proxy`**.

5.3.4 Ordering and hash support

Since ordering and hash support is not trivial to implement for a polymorphic wrapper like **`proxy`**, similar with **`move_only_function`**, we decided not to propose them in this paper.

5.3.5 Freestanding

As per our implementation experience, there is no technical issue to implement the proposed library (not including facilities that are not proposed, yet in our codebase) as freestanding, therefore we propose the whole library to be standardized as freestanding.

5.4 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.

5.4.1 Compared with other active proposals

P3019R6: indirect and polymorphic: Vocabulary Types for Composite Class Design

This paper proposed two class templates to the standard library: **`indirect<T>`** and **`polymorphic<T>`**. Among them, **`polymorphic<T>`** confers value-like semantics on a dynamically allocated object that publicly derived from **`T`**. Although it facilitates lifetime management of an object that has virtual functions, it still requires a type to opt-in the existing virtual mechanism in the standard to have runtime polymorphism. In addition, pointer semantics of proxy allows more flexible storage and lifetime management, including SBO and shared semantics as mentioned earlier.

5.4.2 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 Proxy is based on pointer semantics, as discussed in 5.1.1, 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 limitations make it difficult to have different lifetime management strategies at runtime without additional overhead. The "simple factory" mentioned in **Error! Reference source not found.** 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:

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

Such overhead could be eliminated in the proposed "proxy" library, as discussed in 5.1.3.

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

5.4.3 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. 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.

6 Technical Specifications

6.1 Feature test macro

In ***[version.syn]***, add:

```
#define __cpp_lib_proxy YYYYMML // also in <memory>
```

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

6.2 Named requirements

6.2.1 The *ProOverload* requirements

A type **`O`** meets the *ProOverload* requirements if it matches one of the following definitions, where **`R`** is the *return type*, **`Args...`** are the *argument types*.

Definitions of `O`
`R(Args...)`
`R(Args...) noexcept`
`R(Args...) &`
`R(Args...) & noexcept`
`R(Args...) &&`

`R(Args...) && noexcept`	
`R(Args...) const`	
`R(Args...) const noexcept`	
`R(Args...) const&`	
`R(Args...) const& noexcept`	
`R(Args...) const&&`	
`R(Args...) const&& noexcept`	

6.2.2 The *ProDispatch* requirements

A type **`D`** meets the *ProDispatch* requirements of types **`T`** and **`O`** if **`D`** is a trivial type, **`O`** meets the *ProOverload* requirelemts, and the following expressions are well-formed and have the specified semantics (let **`R`** be return type of **`O`**, **`Args...`** be the argument types of **`O`**. **`args...`** denotes values of type **`Args...`**, **`v`** denotes a value of type **`T`**, **`cv`** denotes a value of type **`const T`**).

Definitions of `O`	**Expressions**	**Semantics**
`R(Args...)`	**`INVOKE<R>(D{}, v, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with an lvalue reference of type **`T`** and **`args...`**, may throw.
`R(Args...) noexcept`	**`INVOKE<R>(D{}, v, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with an lvalue reference of type **`T`** and **`args...`**, shall not throw.
`R(Args...) &`	**`INVOKE<R>(D{}, v, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with an lvalue reference of type **`T`** and **`args...`**, may throw.
`R(Args...) & noexcept`	**`INVOKE<R>(D{}, v, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with an lvalue reference of type **`T`** and **`args...`**, shall not throw.
`R(Args...) &&`	**`INVOKE<R>(D{}, std::move(v), std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a rvalue reference of type **`T`** and **`args...`**, may throw.
`R(Args...) && noexcept`	**`INVOKE<R>(D{}, std::move(v), std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a rvalue reference of type **`T`** and **`args...`**, shall not throw.
`R(Args...) const`	**`INVOKE<R>(D{}, cv, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a **`const`** reference of type **`T`** and **`args...`**, may throw.
`R(Args...) const noexcept`	**`INVOKE<R>(D{}, cv, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a **`const`** reference of type **`T`** and **`args...`**, shall not throw.

`R(Args...) cosnt&`	**`INVOKE<R>(D{}, cv, std::forward<Args>(args)...)`**, **or** **`d(nullptr, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a **`const`** reference of type **`T`** and **`args...`**, may throw.
`R(Args...) const& noexcept`	**`INVOKE<R>(D{}, cv, std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a **`const`** reference of type **`T`** and **`args...`**, shall not throw.
`R(Args...) const&&`	**`INVOKE<R>(D{}, std::move(cv), std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a const rvalue reference of type **`T`** and **`args...`**, may throw.
`R(Args...) const&& noexcept`	**`INVOKE<R>(D{}, std::move(cv), std::forward<Args>(args)...)`**	Invokes dispatch type **`D`** with a **`const`** rvalue reference of type **`T`** and **`args...`**, shall not throw.

6.2.3 The *ProBasicConvention* requirements

A type **`C`** meets the *ProBasicConvention* requirements if the following expressions are well-formed and have the specified semantics.

Expressions	Semantics
`C::is_direct`	A core constant expression of type **`bool`**, specifying whether the convention applies to a pointer type itself (**`true`**), or the element type of a pointer type (**`false`**).
`typename C::dispatch_type`	A trivial type that defines how the calls are forwarded to the concrete types.
`typename C::overload_types`	A tuple-like type of one or more distinct types **`Os`**. Each type **`O`** in **`Os`** shall meet the *ProOverload* requirements.

6.2.4 The *ProConvention* requirements

A type **`C`** meets the *ProConvention* requirements of a type **`P`** if **`C`** meets the *ProBasicConvention* requirements, and the following expressions are well-formed and have the specified semantics.

Expressions	Semantics
`typename C::overload_types`	A tuple-like type that contains one or more distinct types **`Os`**. Each type **`O`** in **`Os`** shall meet the *ProOverload* requirements, and - when **`C::is_direct`** is **`true`**, **`typename C::dispatch_type`** shall meet the *ProDispatch* requirements of **`P`** and **`O`**, - or otherwise, when **`C::is_direct`** is **`false`**, let **`QP`** be a qualified reference type of **`P`** with the *cv ref* qualifiers defined by **`O`** (**`QP`** is an lvalue reference type if **`O`** does not define a *ref* qualifier), **`qp`** be a value of **`QP`**, **`*std::forward<QP>(qp)`** shall be well-formed, and **`typename C::dispatch_type`** shall meet the *ProDispatch* requirements of **`decltype(*std::forward<QP>(qp))`** and `` ` ``.

6.2.5 The *ProBasicReflection* requirements

A type **`R`** meets the *ProBasicReflection* requirements if the following expressions are well-formed and have the specified semantics.

Expressions	Semantics
`R::is_direct`	A core constant expression of type **`bool`**, specifying whether the reflection applies to a pointer type itself (**`true`**), or the element type of a pointer type (**`false`**).
`typename R::reflector_type`	A trivial type that defines the data structure reflected from the type.

6.2.6 The *ProReflection* requirements

A type **`R`** meets the *ProReflection* requirements of a type **`P`** if **`R`** meets the *ProBasicReflection* requirements, and the following expressions are well-formed and have the specified semantics (let **`T`** be **`P`** when **`R::is_direct`** is **`true`**, or otherwise **`typename std::pointer_traits<P>::element_type`**).

Expressions	Semantics
`typename R::reflector_type{std::in_place_type<T>}`	A core constant expression that constructs a value of type **`typename R::reflector_type`**, reflecting implementation-defined metadata of type **`T`**.

6.2.7 The *ProBasicFacade* requirements

A type **`F`** meets the *ProBasicFacade* requirements if the following expressions are well-formed and have the specified semantics.

Expressions	Semantics
`typename F::convention_types`	A tuple-like type that contains any number of distinct types **`Cs`**. Each type **`C`** in **`Cs`** shall meet the *ProBasicConvention* requirements.
`typename F::reflection_types`	A tuple-like type that contains any number of distinct types **`Rs`**. Each type **`R`** in **`Rs`** shall define reflection on pointer types.
`F::constraints`	A core constant expression of type **`proxiable_ptr_constraints`** that defines constraints to pointer types.

6.2.8 The *ProFacade* requirements

A type **`F`** meets the *ProFacade* requirements of a type **`P`** if **`F`** meets the *ProBasicFacade* requirements, and **`P`** meets the requirements defined by **`F::constraints`**, and the following expressions are well-formed and have the specified semantics.

Expressions	Semantics
`typename F::convention_types`	A tuple-like type that contains any number of distinct types **`Cs`**. Each type **`C`** in **`Cs`** shall meet the *ProConvention* requirements of **`P`**.
`typename F::reflection_types`	A tuple-like type that contains any number of distinct types **`Rs`**. Each type **`R`** in **`Rs`** shall meet the *ProReflection* requirements of **`P`**.

6.2.9 The *ProAccessible* requirements

Given that **`F`** is a type meeting the *ProBasicFacade* requirements, a type **`T`** meets the *ProAccessible* requirements of type **`F`**, if the following expressions are well-formed and have the specified semantics.

Expressions	Semantics
`typename T::template accessor<F>`	A type that provides accessibility to **`proxy`**. It shall be a *nothrow-default-constructible*, *trivially-copyable* type, and shall not be final.

6.3 Header <memory> synopsis

```
// all freestanding
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;
  };

  template <class F>
    concept facade = see below;

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

```
  template <class F>
    class proxy_indirect_accessor;

  template <class F>
    class proxy;

  template <class F, class A>
    proxy<F>& access_proxy(A& a) noexcept;

  template <class F, class A>
    const proxy<F>& access_proxy(const A& a) noexcept;

  template <class F, class A>
    proxy<F>&& access_proxy(A&& a) noexcept;

  template <class F, class A>
    const proxy<F>&& access_proxy(const A&& a) noexcept;

  template <bool IsDirect, class D, class O, class F, class... Args>
    see below proxy_invoke(proxy<F>& p, Args&&... args);

  template <bool IsDirect, class D, class O, class F, class... Args>
    see below proxy_invoke(const proxy<F>& p, Args&&... args);

  template <bool IsDirect, class D, class O, class F, class... Args>
    see below proxy_invoke(proxy<F>&& p, Args&&... args);

  template <bool IsDirect, class D, class O, class F, class... Args>
    see below proxy_invoke(const proxy<F>&& p, Args&&... args);

  template <bool IsDirect, class R, class F>
    const R& proxy_reflect(const proxy<F>& p) noexcept;
}
```

6.4 Constraints

```
template <class F>
  concept facade = see below;
```

The **`concept facade<F>`** specifies that a type **`F`** models a facade of **`proxy`**. If **`F`** depends on an incomplete type, and its evaluation could yield a different result if that type were hypothetically completed, the behavior is undefined. **`facade<F>`** is **`true`** when **`F`** meets the *ProBasicFacade* requirements; otherwise, it is **`false`**.

Note that **`concept facade`** does not impose strong constraints on the dependent convention and reflection types.

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

The concept **`proxiable<P, F>`** specifies that **`proxy<F>`** can potentially contain a value of type **`P`**. For a type **`P`**, if **`P`** is an incomplete type, the behavior of evaluating **`proxiable<P, F>`** is undefined. **`proxiable<P, F>`** is **`true`** when **`F`** meets the *ProFacade* of **`P`**; otherwise, it is **`false`**.

6.5 Proxy

6.5.1 Class template `proxy_indirect_accessor`

```
namespace std {
  template <class F>
  class proxy_indirect_accessor : see below {}
}
```

Class template **`proxy_indirect_accessor`** provides indirection accessibility for **`proxy`**. To instantiate **`proxy_indirect_accessor<F>`**, **`F`** shall model concept facade. As per **`facade<F>`**, **`typename F::convention_types`** shall be a tuple-like type containing any number of distinct types **`Cs`**, and **`typename F::reflection_types`** shall be a tuple-like type containing any number of distinct types **`Rs`**. For each type **`T`** in **`Cs`** or **`Rs`**, if **`T`** meets the *ProAccessible* requirements of **`F`** and **`T::is_direct`** is **`false`**, **`typename T::template accessor<F>`** is inherited by **`proxy_indirect_accessor<F>`** with **`public`** visibility.

6.5.2 Class template `proxy`

6.5.2.1 General

```
namespace std {
  template <class F>
  class proxy : see below {
  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;
    explicit operator bool() const noexcept;
    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);

    proxy_indirect_accessor<F>* operator->() noexcept;
    const proxy_indirect_accessor<F>* operator->() const noexcept;
    proxy_indirect_accessor<F>& operator*() & noexcept;
    const proxy_indirect_accessor<F>& operator*() const& noexcept;
    proxy_indirect_accessor<F>&& operator*() && noexcept;
    const proxy_indirect_accessor<F>&& operator*() const&& noexcept;

    friend void swap(proxy& lhs, proxy& rhs) noexcept(see below);
    friend bool operator==(const proxy& lhs, nullptr_t) noexcept;

  private:
    proxy_indirect_accessor<F> ia;    // exposition only
  };
}
```

Class template **`proxy`** is a general-purpose polymorphic wrapper for C++ pointers. It also supports flexible lifetime management without garbage collection at runtime.

To instantiate **`proxy<F>`**, **`F`** shall model concept **`facade`**. As per **`facade<F>`**, **`typename F::convention_types`** shall be a tuple-like type containing any number of distinct types **`Cs`**, and **`typename F::reflection_types`** shall be a tuple-like type containing any number of distinct types **`Rs`**. For each type **`T`** in **`Cs`** or **`Rs`**, if **`T`** meets the *ProAccessible* requirements of **`F`** and **`T::is_direct`** is **`true`**, **`typename T::template accessor<F>`** is inherited by **`proxy<F>`** with public visibility.

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>`
`HasNothrowPolyConstructor<P, Args...>`
	`conditional_t<proxiable<P, F>,`
	`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::constraints.copyability ==` `constraint_level::trivial`
`HasNothrowCopyConstructor`	`F::constraints.copyability >=` `constraint_level::nothrow`
`HasCopyConstructor`	`F::constraints.copyability >=` `constraint_level::nontrivial`
`HasNothrowMoveConstructor`	`F::constraints.relocatability >=` `constraint_level::nothrow`
`HasMoveConstructor`	`F::constraints.relocatability >=` `constraint_level::nontrivial`
`HasTrivialDestructor`	`F::constraints.destructibility ==` `constraint_level::trivial`
`HasNothrowDestructor`	`F::constraints.destructibility >=` `constraint_level::nothrow`
`HasDestructor`	`F::constraints.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`

6.5.2.2 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>, std::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>`** in conjunction with that **`decay_t<P>`** is not the same type as **`proxy`** nor a specialization of **`in_place_type_t`**.
Effects: Initializes the contained value as if direct-initializing an object of type **`VP`** with **`std::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 **`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, 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, 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.

6.5.2.3 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(std::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(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>**.

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::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::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.

6.5.2.4 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 exception is thrown, each of **`*this`** and **`rhs`** is in a valid state with unspecified value.	Equivalent to (**`*this = std::move(rhs)`**); post condition is that **`*this`** contains a value and **`rhs`** does not contain a value.
`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`**.

`friend void swap(proxy& lhs, proxy& rhs) noexcept(`*see below***`);`**

Effects: Equivalent to **`lhs.swap(rhs)`**.

Remarks: The expression inside **`noexcept`** is equivalent to **`HasNothrowMoveConstructor`**.

6.5.2.5 Observers

```
bool has_value() const noexcept;
explicit operator bool() const noexcept;
```

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

```
friend bool operator==(const proxy& lhs, nullptr_t) noexcept;
```

Returns: **`!lhs.has_value()`**.

```
proxy_indirect_accessor<F>* operator->() noexcept;
```

Returns: **`addressof(ia)`**.

```
const proxy_indirect_accessor<F>* operator->() const noexcept;
```

Returns: **`addressof(ia)`**.

```
proxy_indirect_accessor<F>& operator*() & noexcept;
```

Returns: **`ia`**.

```
const proxy_indirect_accessor<F>& operator*() const& noexcept;
```

Returns: **`ia`**.

```
proxy_indirect_accessor<F>&& operator*() && noexcept;
```

Returns: **`std::move(ia)`**.

```
const proxy_indirect_accessor<F>&& operator*() const&& noexcept;
```

Returns: **std::move(ia)**.

6.5.2.6 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.

6.5.3 Proxy manipulation functions

```
template <class F, class A>
proxy<F>& access_proxy(A& a) noexcept;
template <class F, class A>
const proxy<F>& access_proxy(const A& a) noexcept;
template <class F, class A>
proxy<F>&& access_proxy(A&& a) noexcept;
template <class F, class A>
const proxy<F>&& access_proxy(const A&& a) noexcept;
```

Preconditions: **F** shall model concept **facade**. As per **facade<F>**, **typename F::convention_types** shall be a tuple-like type containing distinct types **Cs**. There shall be a type **C** in **Cs** where **A** is the same type as **typename C::template accessor<F>**. **a** shall be instantiated from a **proxy** object.
Returns: A reference to the **proxy** that instantiated **a**.

```
template <bool IsDirect, class D, class O, class F, class... Args>
see below proxy_invoke(proxy<F>& p, Args&&... args);
template <bool IsDirect, class D, class O, class F, class... Args>
see below proxy_invoke(const proxy<F>& p, Args&&... args);
template <bool IsDirect, class D, class O, class F, class... Args>
see below proxy_invoke(proxy<F>&& p, Args&&... args);
template <bool IsDirect, class D, class O, class F, class... Args>
see below proxy_invoke(const proxy<F>&& p, Args&&... args);
```

Preconditions: There shall be a convention type **Conv** defined in **typename F::convention_types** where **Conv::is_direct == IsDirect && std::is_same_v<typename Conv::dispatch_type, D>** is **true**. **O** is required to be defined in **typename Conv::overload_types**. **p** shall contain a value.
Effects: Invokes a **proxy** with a specified dispatch type, an overload type, and arguments. Let **ptr** be the contained value of **p** with the same cv ref-qualifiers, **Args2...** be the argument types of **O**, **R** be the return type of **O**,

- if `IsDirect` is `true`, let `v` be `std::forward<decltype(ptr)>(ptr)`, or otherwise,
- if `IsDirect` is `false`, let `v` be `*std::forward<decltype(ptr)>(ptr)`,

equivalent to:

- `INVOKE<R>(D{}, std::forward<decltype(v)>(v), static_cast<Args2>(args)...)` if the expression is well-formed, or otherwise,
- `INVOKE<R>(D{}, nullptr, static_cast<Args2>(args)...)`.

Return type: The return type of `O`.

```
template <bool IsDirect, class R, class F>
const R& proxy_reflect(const proxy<F>& p) noexcept;
```

Preconditions: There shall be a reflection type `Refl` defined in `typename F::reflection_types` where `Refl::is_direct == IsDirect && std::is_same_v<typename Refl::reflection_type, R>` is `true`. `p` shall contain a value.

Effects: Let `P` be the type of the contained value of `p`. Retrieves a value of type `const R&` constructed from `in_place_type<T>`, where `T` is `P` when `IsDirect` is `true`, or otherwise `T` is `typename std::pointer_traits<P>::element_type` when `IsDirect` is `false`.

Remarks: The reference obtained from `proxy_reflect()` may be invalidated if `p` is subsequently modified.

7 Acknowledgements

Mingxin would like to thank: Tian Liao (Microsoft) for the insights into the library design and open source. Wei Chen (Jilin University), Herb Sutter, Roger Orr, Chandler Carruth, Daveed Vandevoorde, Bjarne Stroustrup, JF Bastien, Bengt Gustafsson, Chuang Li (Microsoft), Nevin Liber and Billy Baker for their valuable feedback on earlier revisions of this paper.

8 Open questions

As per review comments from LEWGI in Tokyo:

8.1 Naming of class template `proxy`

During the review there was some controversy over the name "proxy". Despite potential confusion with networking proxies, the use of distinct namespaces should mitigate ambiguity. Specifically,

- Clarity of Purpose: "proxy" accurately describes the functionality of the library, which is to serve as an intermediary that represents or stands in for another object. This clarity helps users immediately understand the role of the library without additional context.
- Consistency with Established Terminology: In programming, a "proxy" often refers to an object that controls access to another object, which is consistent with the behavior of the proposed library. This consistency with established patterns aids in learning and understanding for those already familiar with the concept.

- Domain Differentiation: While "proxy" is also a term used in networking, the concept of namespaces in C++ effectively separates concerns and prevents ambiguity. Just as **`std::copy`** and **`std::filesystem::copy`** have distinct functionalities within their respective domains, so too would a "proxy" within the **`std`** namespace be distinct from a networking proxy within a different namespace, such as **`std::net`**.
- Precedent for Overlapping Terms: There are numerous examples in C++ where the same term may have different meanings in different contexts, yet this does not typically lead to confusion due to the language's structure and namespace system.

In summary, we believe "proxy" is a term that conveys the intended functionality with precision, aligns with existing programming concepts, and can be clearly differentiated within the C++ namespace system. These factors make it a suitable choice for the library's name. In the meantime, we have come up with 3 alternatives to be considered:

Name	Pros	Cons
agent	Implies action on behalf of another, without direct stand-in implications.	May imply autonomous action, which is not the case with the proposed feature.
handle	Well-understood term in programming, especially for resource management.	Overused and may not convey the semantics of a pointer to another object.
poly	Reflects the capability of the proposed library to exhibit different behaviors at runtime.	May inadvertently suggest a connection to the traditional use of polymorphism in C++ through virtual functions, which is not the case here.
delegate	Implies that operations are passed on to another entity.	Already has a distinct meaning in other programming languages, which could lead to confusion.

8.2 Naming of `constraints` in concept `facade`

In the paper, the term "constraints" is utilized within concept **`facade`** to denote the restrictions applied to the pointer types that can be used to instantiate a **`proxy`**. This terminology was selected for its clear conveyance of the intended functionality and its familiarity within the C++ committee.

However, it has been brought to attention that the term "constraints" is also a key term within the domain of Concepts in C++, which may lead to ambiguity due to its general nature. While the term's broad recognition is beneficial for understanding, the potential for confusion with the established use in Concepts is acknowledged. To mitigate this concern, the proposal remains open to alternative nomenclature that would preserve the term's descriptive quality while distinguishing it from its broader usage in Concepts. Suggestions such as "pointer_constraints" or "proxy_constraints" may offer a more specific reference, thereby reducing ambiguity.

9 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.