PFA: A Generic, Extendable and Efficient Solution for Polymorphic Programming

Document number:	P0957R3
Date:	2019-10-06
Project:	Programming Language C++
Audience:	LEWG, LWG, SG7

Authors:	Mingxin Wang
Reply-to:	Mingxin Wang <mingxwa@microsoft.com>

Table of Contents

1 History

1.1 Change 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`**.

1.2 Change 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`**.

1.3 Change 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 for polymorphism, the "PFA" is proposed as a generic, extendable and efficient solution for polymorphic programming, named by 3 of its major concepts: "Proxy", "Facade", and "Addresser". The "PFA" combines the idea of OOP (Object-oriented Programming) and FP (Functional Programming). Meanwhile, eliminating their defects to some extent. Comparing to traditional OOP, PFA can largely replace the existing "virtual mechanism" and have no intrusion on existing code or runtime memory layout, without reducing performance. Comparing to FP, PFA 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 "PFA" is well-compatible with the C++ programming language and makes C++ easier to use. PFA 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 PFA without performance loss, for example, **`std::function`** and **`std::any`**.

The rest of the paper is organized as follows: Section 3 illustrates the motivation and scope of PFA; Section 4 illustrates PFA's impact on the C++ standard; Section 5 includes the pivotal decisions in the design; Section 6 describes a typical and meaningful use cases indicating the basic usage; Section 7 illustrates the technical specification of PFA in C++; Section 8 lists some of the future works to be done and summarizes the paper.

3 Motivation

Currently, there are two types of mechanisms for polymorphism in the standard: inheritance with virtual functions and polymorphic wrappers. This section illustrates architecting and performance limitations in inheritance-based polymorphism and other polymorphic facilities in the standard, and how the proposed library could help.

3.1 Limitations in Existing Mechanisms for Polymorphism

3.1.1 Architecting

Polymorphism is widely required in large-scale programming to decouple components and increase extendibility at a cost of reducing runtime performance. Because 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.

Prior research for future polymorphic usage is usually required when designing polymorphic types with inheritance. However, if the research is inadequate due to limited experience, 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 a system, 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, satisfies 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.

All in all, inheritance-based polymorphism has limitations in architecting because it is intrusive. As Sean Parent concluded on NDC 2017: *The requirements of a polymorphic type, by definition, comes from its use, and there are no polymorphic types, only polymorphic use of similar types. Inheritance is the base class of evil.*

3.1.2 Performance

In addition to being difficult to architect, existing mechanisms for polymorphism also have certain limitations in performance, especially in memory management and addressing.

3.1.2.1 Memory Management

There are technical limitations in memory management for existing mechanisms for polymorphism, especially in customizing Allocation algorithms and SOO (small object optimization, aka SBO, small buffer optimization).

3.1.2.1.1 Customizing Allocation Algorithms

In C++98, the concept of "Allocator" was introduced, and was widely adopted in STL. However, since the "Allocator" has too many customization points and is type-specific, it becomes difficult to reuse this concept in type-erased components. For example, the class template **`std::function`** used to have a constructor that allow customized allocator types, just like other STL containers does, but the constructor was eventually removed in C++17 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" [P0302R1]. For inheritance-based polymorphism, there is even less facilities in the standard to customize allocation algorithms.

3.1.2.1.2 SOO

SOO is a common technique to avoid unnecessary memory allocation. However, for inheritance-based polymorphism, there is little facilities in the standard that support SOO; for other standard polymorphic wrappers, implementations may support SOO, 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`**.

3.1.2.2 Addressing

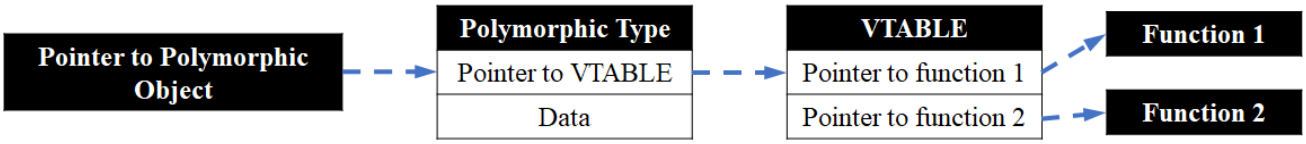

Figure 1

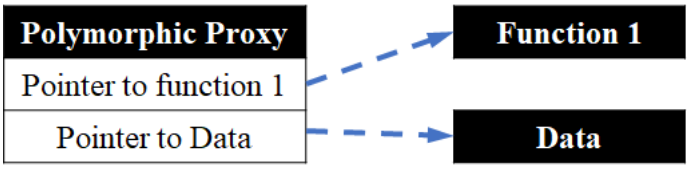

Figure 2

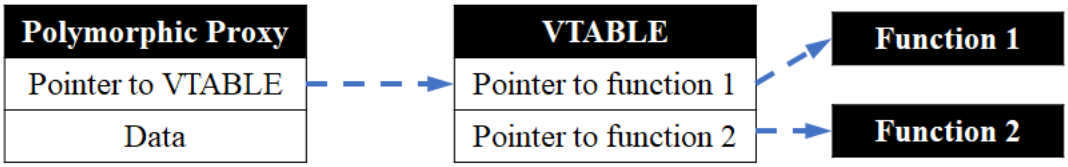

Figure 3

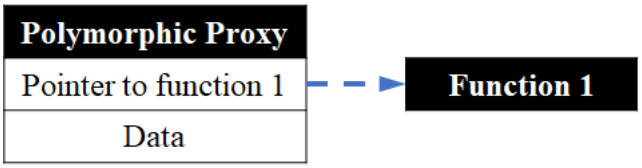

Figure 4

Because inheritance-based polymorphism shall support upward casting and multiple inheritance, it has limited design space in memory layout which is usually like the one shown in Figure 1. However, it is not always the optimal for addressing when the size of the VTABLE or data is small. For example, Figure 2, Figure 3 and Figure 4 shows a more efficient memory layout for addressing when the VTABLE is small, when the data is small, and when the VTABLE and data are both small, respectively.

3.2 The Solution

This section includes a brief introduction about how the proposed library, the PFA, mitigates the limitations illustrated in the last section.

3.2.1 Architecting Guidelines

As illustrated before, there are limitations in architecting polymorphic program with existing mechanisms. Concretely, inheritance-based polymorphism is intrusive and has limited maintainability, while the standard polymorphic wrappers has fixed semantics and has limited extendibility. Based on these considerations, the PFA is designed to be non-intrusive and able to carry rich semantics.

For example, if we want to use a series of "drawable" types polymorphically with "dot" expression like **`foo.draw()`**, we could define a **`facade Drawable`**.

```
facade Drawable {
  void draw() const;
};
```

Note that the **`facade`** is a newly proposed keyword describing the expressions for polymorphic use. For any future "Drawable" implementations, they are only required to satisfy the constraints that the expression **`foo.draw()`** shall be well-formed and have specific semantics without inheriting from any base class. For example, if there are two implementations for "Drawable":

```
struct square {
  explicit square(float width, float height);
  void draw() const;
  void set_position(float x, float y);
  void set_width(float width);
  void set_height(float height);
  void set_alpha(float alpha = 0.5);
};
struct circle {
  explicit circle(float radius);
  void draw() const;
  void set_position(float x, float y);
  void set_radius(float radius);
  void set_alpha(float alpha = 0.3);
};
```

We will be able to manage them with the "proxy". For example, the following code is well-formed with the PFA:

```
value_proxy<Drawable> p1 = square{2.0, 2.0};  // Construct a 'square' and manage with
the proxy
value_proxy<Drawable> p2 = make_sinking_construction<circle>(1.5);  // Construct a
'circle' in-place and manage with the proxy
p1.draw();  // Perform 'square::draw()'
p2.draw();  // Perform 'circle::draw()'
p1.reset();  // Destroy the 'square' stored in 'p1' and leave 'p1' not representing
any value
```

```
p1 = std::move(p2);  // Move owership for the 'circle' from 'p2' to 'p1' and leave 'p2' not representing any value
```

Note that **value_proxy** supports "Sink Argument" which is a concept proposed in the "Sink Argument" library [P1648R2], and **make_sinking_construction** is also proposed in that paper.

From the code above we can see that polymorphism usage is decoupled from the implementation and is non-intrusive. Meanwhile, the **facade** supports any expression and is open for extendibility.

3.2.2 Exploring Performance

The PFA is designed to have the highest possible performance I could think of, providing proper inline and caching, and provide customization points for possible tradeoff between performance and extendibility.

3.2.2.1 SOO

The proposed **value_proxy** is configurable with SOO, and users could assign any possible size and alignment for SOO if necessary. For example, type **value_proxy<SomeFacade, 56u, 8u>** is configured that the size for SOO is **56** and align to **8** bytes.

3.2.2.2 Addressing

The PFA allows design space in memory layout and may have higher addressing performance than inheritance-based polymorphism. For example, if the size of VTABLE is small, implementations may directly store the VTABLE inside the proxy, as is shown in Figure 2; if the size of data is small and meeting some other specific preconditions, implementations may also store the data in the proxy, as is shown in Figure 3.

4 Impact on the Standard

PFA introduces a novel solution for polymorphic programming, including Proxy, Facade and Addresser. Because the components defined in the standard that related to polymorphism can be easily implemented with PFA without performance loss, I think the following features in the standard may gradually be deprecated in the future:

1. Virtual functions;
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. The "Polymorphic Memory Resources" library.

5 Design Decisions

5.1 Scope

As mentioned earlier, PFA is designed to help users build extendable and efficient polymorphic programs. In order to make implementations efficient in C++, it is helpful to collect as much requirements and generate high-quality code at compile-time as possible.

The basic goal of PFA is to eliminate the limitations in traditional OOP and FP, as was illustrated in the Motivation part.

5.2 OOP Based

Investigating C++ and other polymorphic programming languages, it is obvious that polymorphic requirements consist of:

(1) Procedure / Procedure set, and

(2) Data / Data set,

where (2) is optional.

If we want to save information of a procedure at runtime, it is usually easy to store it as a function pointer. When it comes to procedure sets, it is usually efficient to build some arrays of function pointers at compile time, each unit stores a specific polymorphic function; then we could use the addresses of the arrays to specify function sets, and this is exactly the mechanism of VTABLE.

For single data, it is OK to pass them by value (Direct addressing); for data sets, a widely accepted approach is to pass the data by a base pointer, and calculate each address of corresponding data with a unique offset (Indirect addressing). In order to unify the addressing mode, indirect addressing is usually adopted in existing polymorphism solutions.

Fortunately, the requirements above is exactly the basis of OOP. We could easily define a data structure corresponding to a set of procedures with the C++ type system. In a class definition in C++, a public member function defines the semantics of the type; the member variables are the data set representing the state of the entity.

5.3 Proxy

I think it is reasonable to allow users to use a uniform type to represent various concrete types to implement polymorphism, like the class template **`std::function`** that could represent any callable type whose input and output are convertible to specific types. PFA is designed as a generic solution for polymorphism, supporting not only callable types, but also types having various expressions and rich semantics. As there shall be "uniform types" to represent concrete implementations, these types are defined as "Proxy".

In order to configure a proxy type, on the one hand, information about

(1) "how should concrete implementations look like" and

(2) "how to manage the polymorphic information and the entity being represented"

shall be provided; on the other hand, a proxy type shall support the expressions specified by (1). As there is no rule defined in the standard for generating member functions, the code for the proxies shall be generated by the compiler, and this shall be a new language feature in C++.

In order to keep consistent with the type system defined in the standard, the "Proxy" is defined as a class template, which is a library feature but may have different implementations for different specializations duck typed by the compiler. Each of the two categories of information shall be specified by a type.

5.4 Facade

Although the Concepts TS is able to 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. I think this solution is not compatible with the type system in C++, 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.

I think it is necessary to add a new mechanism that could carry such information required by polymorphism, which is defined as "Facade". A facade shall be a descriptive placeholder type that carries information of user-defined expressions and semantics. Concretely, users are allowed to define expressions of a type with a facade. Like the type traits and disambiguation tags (**`std::in_place`**, **`std::in_place_type`**, and **`std::in_place_index`**) defined in the standard, facades shall be trivial types that works at compile time to specify templates only.

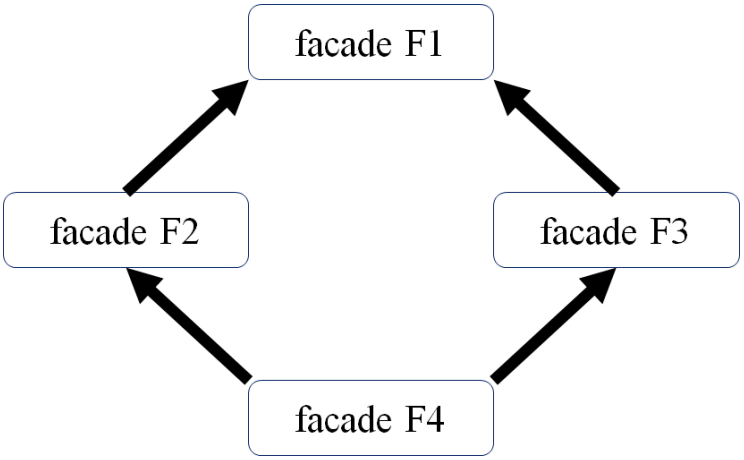

Figure 5

Facades shall support inheritance. As is defined in the C++ type system, a derived facade shall be convertible to any of its base. Like an ordinary type, with facades as the vertex and the inheritance relation between them as directed edges, all the facades in a program form a static DAG (Directed Acyclic Graph).

Repeated inheritance is also allowed in facades, but their corresponding proxies are usually not convertible based on performance considerations.

5.5 Addresser

In addition to the Facade, there is another category of information required to specify a proxy: "how to manage the polymorphic information and the entity being represented". Before C++17, the most widely used utilities in the standard are the pointers and "smart pointers" (specifically, class template **`std::shared_ptr`** and **`std::unique_ptr`**). In C++17, there is another utility, **`std::any`**. The fundamental difference between smart pointers and **`std::any`** is that smart pointers are type-specific, while **`std::any`** is type-erased. In the aspect of lifecycle control, **`std::unique_ptr`** and **`std::any`** have similar lifecycle as the entity being represented does, besides that **`std::any`** is **`CopyConstructible`** while **`std::unique_ptr`** is not.

5.5.1 Data and Metadata

Data is a straightforward concept and is the basis of computer science. However, the concept of "metadata" is sometimes ignored by software designers.

From a philosophical point of view, anything can deduce an infinite number of things, and any type can deduce infinite amount of metadata in programming. However, only a limited part of them is required at any moment. In PFA, the format of metadata is defined by the Facade, and the Addresser is responsible for managing the values of various types and various metadata.

5.5.2 Addressing Patterns

The "Addresser" is associated with the responsibility of addressing and may have different lifecycle management

strategies. It seems difficult to extend DGPVC with other lifetime management strategies as it only supports the basic "reference semantics" and "value semantics", e.g. reference-counting based algorithm and more complicated GC algorithms. In this solution, users are free to specify different types of addressers for any lifetime management requirements. Besides, I think it is rude to couple the "characteristics of construction and destruction" with other expressions required in DGPVC. When it is not required to manage the lifetime issue (e.g. with reference semantics), the constraints related to constructors and destructors are redundant; conversely, when we need value semantics, it is natural that the type being type-erased shall be at least **`MoveConstructible`** most of the time. This problem does not exist in this solution as constructors and destructors are not able to be declared pure virtual, and an addresser type may carry such constraints if necessary.

For metadata, it is usually efficient to build them at compile-time. In the cases of size-critical situations, generating the metadata at runtime may also acceptable.

6 Basic Usage

Suppose it is required to design a function that accepts a "map" entity (mapping from integers to **`std::string`**) and does "query" operations only, and compile it as a static library. One may define it as:

```
void do_something_with_map(const std::map<int, std::string>&);
```

or,

```
void do_something_with_map(const std::unordered_map<int, std::string>&);
```

Actually, the library does not need to care the implementation of the "map" at all and could be more extendable with PFA. For example, the following facade will clearer the semantics of the "map":

```
template <class K, class V>
facade ImmutableMap {
  const V& at(const K&) const;
};
```

Users are able to declare the function as:

```
void do_something_with_map(reference_proxy<const ImmutableMap<int, std::string>>);
```

The improved signature of the function accepts any type that has facade **`ImmutableMap<int, std::string>`**, so that all of the following expressions are well-formed:

```
std::map<int, std::string> var1{{1, "Hello"}};
std::unordered_map<int, std::string> var2{{2, "CPP"}};
std::vector<std::string> var3{"I", "love", "PFA", "!"};
std::map<int, std::string> var4{};
do_something_with_map(var1);
```

```
do_something_with_map(var2);
do_something_with_map(var3);
do_something_with_map(var4);
```

Providing this function is asynchronous, and the "map" should be passed to other threads, the signature of the function could be redefined as:

```
void do_something_with_map_async(value_proxy<ImmutableMap<int, std::string>>);
```

Just change **reference_proxy** into **value_proxy**, and the object is stored in value! The expressions above are still well-formed but with slightly different semantics: the values are copied to the proxy. In order to avoid unnecessary copy operation, the "Sink Argument" [P1648R2] is supported. For example, the following expression is well-formed:

```
value_proxy<ImmutableMap<int, std::string>> p
    = std::in_place_type<std::map<int, std::string>>;
```

7 Technical Specifications

7.1 Header <proxy> synopsis

```
namespace std {

enum class qualification_type
    { none, const_qualified, volatile_qualified, cv_qualified };

enum class reference_type { lvalue, rvalue };

template <class T, qualification_type Q> using add_qualification_t = see below;
template <class T> inline constexpr qualification_type qualification_of_v = see below;
template <class T, reference_type R> using add_reference_t = see below;

class null_value_addresser_error;

template <qualification_type Q, reference_type R> using erased_reference = see below;

template <size_t SIZE, size_t ALIGN>
struct erased_value_selector {
  template <qualification_type Q, reference_type R>
  using type = see below;
};

template <class M, size_t SIZE, size_t ALIGN> class value_addresser;
```

```
template <class M, qualification_type Q> class reference_addresser;

template <class F, template <qualification_type, reference_type> class E>
    struct proxy_meta;
template <class F, class A> class proxy;

template <class F, size_t SIZE = sizeof(void*), size_t ALIGN = alignof(void*)>
using value_proxy = proxy<F, value_addresser<proxy_meta<
    F, erased_value_selector<SIZE, ALIGN>::template type>, SIZE, ALIGN>>;

template <class F>
using reference_proxy = proxy<decay_t<F>, reference_addresser<
    proxy_meta<decay_t<F>, erased_reference>, qualification_of_v<F>>>;

}
```

7.2 Qualification and Reference Computation

`template <class T, qualification_type Q> using add_qualification_t =` `see below;`

Definition: The type **`U`** obtained by adding qualification defined by **`Q`** to **`T`**,

- if **`Q`** is **`qualification_type::none`**, **`U`** is **`T`**, or
- if **`Q`** is **`qualification_type::const_qualified`**, **`U`** is const **`T`**, or
- if **`Q`** is **`qualification::type::volatile_qualified`**, **`U`** is **`volatile T`**, or
- otherwise, if **`Q`** is **`qualification_type::cv_qualified`**, **`U`** is **`const volatile T`**.

`template <class T> inline constexpr qualification_type qualification_of_v =` `see below;`

Definition: The qualification **`Q`** of **`T`**,

- if **`T`** is not qualified, **`Q`** is **`qualification_type::none`**, or
- if **`T`** is const-qualified, **`Q`** is **`qualification_type::const_qualified`**, or
- if **`T`** is volatile-qualified, **`Q`** is **`qualification_type::volatile_qualified`**, or
- otherwise, if **`T`** is cv-qualified, **`Q`** is **`qualification_type::cv_qualified`**.

`template <class T, reference_type R> using add_reference_t =` `see below;`

Definition: The type **`U`** obtained by adding reference defined by **`R`** to **`T`**,

- if **`R`** is **`reference_type::lvalue`**, **`U`** is **`T&`**, or
- otherwise, if **`R`** is **`reference_type::rvalue`**, **`U`** is **`T&&`**.

The three type traits above are proposed for qualification and reference computation required by other facilities.

7.3 Facade

The "Facade" is a descriptive placeholder that abstracts the types that have specific expressions and semantics. If the reference type of a non-static expression is not specified, it implicitly implies lvalue reference semantics. A general

declaration of a facade is as follows:

```
facade 'name' [: 'more_abstract_facade_1', 'more_abstract_facade_2', ...] {
  [static] 'return_type' 'name'(['arg0', 'arg1', ...]) [const] [volatile] [&|&&];
  [...]
};
```

The same as class templates, facades could be template and accept partial specialization. For example, the following facade template is well-formed:

```
template <class T>
facade Callable; // undefined

template <class R, class... Args>
facade Callable<R(Args...)> {
  R operator()(Args...);
};
```

Like the type traits and disambiguation tags (**std::in_place**, **std::in_place_type**, and **std::in_place_index**) defined in the standard, facades are trivial types that works at compile time to specify templates only.

7.4 Addresser

7.4.1 Erased Handles

```
template <qualification_type Q, reference_type R>
using erased_reference = see below;

template <size_t SIZE, size_t ALIGN>
struct erased_value_selector {
  template <qualification_type Q, reference_type R>
  using type = see below;
};
```

The following expressions shall be well-formed and have specific semantics (**er** denotes a value of type **erased_reference<Q, R>**, **ev** denotes a value of type **erased_value_selector<SIZE, ALIGN>::template type<Q, R>**):

er.template cast<T>()

Requires: **R** is **reference_type::lvalue**, **add_qualification_t<T, Q>** shall be the original type before type erasure.

Returns: An lvalue reference to the original value before type erasure.

Return Type: **`add_qualification_t<T, Q>&`**.

`ev.template cast<T>()`

Requires: **`T`** shall be the original type before type erasure.

Returns: A reference to the original value before type erasure.

Return Type: **`add_reference_t<add_qualification_t<T, Q>, R>`**.

7.4.2 Addressers

This section provides mechanisms for addressing with frequently used lifecycle management and addressing strategies. These mechanisms ease the production of PFA based programs.

7.4.2.1 Requirements for Addresser Types

7.4.2.1.1 `BasicAddresser` Requirements

A type **`A`** meets the **`BasicAddresser`** requirements if the following expressions are well-formed and have the specific semantics (**`a`** denotes a value of type **`A`**).

`a.meta()`

Effects: acquires the metadata of a specific object if there is one, otherwise, this behavior is undefined.

7.4.2.1.2 `Addresser` Requirements

A type **`A`** meets the **`Addresser`** requirements if it meets the **`BasicAddresser`** requirements and the following expressions are well-formed and have the specific semantics (**`a`** denotes a value of type **`A`**).

`a.erased()`

Effects: acquires the erased handle of a specific object if there is one, otherwise, this behavior is undefined.

7.4.2.2 Class `null_value_addresser_error`

```
class null_value_addresser_error : public logic_error {
 public:
  explicit null_value_addresser_error();
};
```

This exception is thrown when acquiring metadata via a value of type **`value_addresser`** that is not associating with a concrete value.

7.4.2.3 Class template `value_addresser`

```
template <class M, size_t SIZE, size_t ALIGN>
class value_addresser {
 public:
  bool has_value() const noexcept;
  const type_info& type() const noexcept;
  void reset() noexcept;
  template <class S_T, class A> void assign(S_T&& val, A&& alloc);
  void swap(value_addresser& rhs) noexcept;

 protected:
  value_addresser(value_addresser&& rhs) noexcept;
  value_addresser() noexcept;
  template <class S_T>
  value_addresser(S_T&& value);
  template <class S_T, class A>
  value_addresser(S_T&& value, A&& alloc);
  value_addresser& operator=(value_addresser&& rhs);
  template <class S_T>
  value_addresser& operator=(S_T&& value);
  ~value_addresser();

  const M& meta() const;
  storage_t& erased() &;
  storage_t&& erased() &&;
  const storage_t& erased() const&;
  const storage_t&& erased() const&&;
};
```

`value_addresser` is the build-in addresser for value semantics and is trivially-relocatable [P1144R4].

7.4.2.3.1 Construction and Destruction

`value_addresser(value_addresser&& rhs) noexcept;`

Effects: If **`rhs.has_value()`** is **`false`**, construct an object associating with no concrete object, or otherwise, move the ownership of the concrete object from **`rhs`** to **`*this`**.

`value_addresser() noexcept;`

Effects: Construct an object associating with no concrete object.

```
template <class S_T >
  explicit value_addresser(S_T&& value);
```

Requires: Equivalent to **`value_addresser(std::forward<S_T>(value), allocator<char>{})`**.

`template <class S_T, class A>`
`explicit value_addresser(S_T&& value, A&& alloc);`

Effects: Construct an object associating with an object of type **`sunk_t<S_T>`** constructed with **`sink(forward<E_T>(value))`**. If the size and the alignment of **`sunk_t<S_T>`** is less than or equal to **`SIZE`** and **`ALIGN`**, respectively, and is trivially-relocatable, the concrete object will be constructed without allocating new memory. Otherwise, a block of memory will be allocated with **`alloc`**.

`~value_addresser();`

Effects: As if by **`reset()`**.

7.4.2.3.2 Assignment

`value_addresser& operator=(value_addresser&& rhs) noexcept;`

Effects: As if by **`value_addresser(std::move(rhs)).swap(*this)`**.

Returns: **`*this`**.

Postconditions: The state of **`*this`** is equivalent to the original state of **`rhs`** and **`rhs`** is left in a valid but otherwise unspecified state.

`template<class S_T>`
`value_addresser& operator=(S_T&& value);`

Effects: Constructs an object **`tmp`** of type **`value_addresser`** with **`std::forward<T>(value)`**, and **`tmp.swap(*this)`**. No effects if an exception is thrown.

Throws: Any exception thrown during the construction of **`tmp`**.

7.4.2.3.3 Addressing

`const M& meta() const;`

Requires: **`*this`** shall be associated with a concrete object.

Returns: A const reference of the metadata.

Throws: **`null_value_addresser_error`** if **`*this`** is not associated with a concrete object.

`see below` **`erased() &;`**

Returns: A value that is implicitly convertible to types **`erased_value_selector<SIZE, ALIGN>::template type<qualification_type::none, reference_type::lvalue>`** and **`erased_value_selector<SIZE, ALIGN>::template type<qualification_type::const_qualified, reference_type::lvalue>`**.

`see below` **`erased() &&;`**

Returns: A value that is implicitly convertible to types **`erased_value_selector<SIZE, ALIGN>::template type<qualification_type::none, reference_type::rvalue>`** and **`erased_value_selector<SIZE, ALIGN>::template`**

type<qualification_type::const_qualified, reference_type::rvalue>.

`see below` **erased() const&;**

Returns: A value that is implicitly convertible to type **erased_value_selector<SIZE, ALIGN>::template type<qualification_type::const_qualified, reference_type::lvalue>**.

`see below` **erased() const&&;**

Returns: A value that is implicitly convertible to type **erased_value_selector<SIZE, ALIGN>::template type<qualification_type::const_qualified, reference_type::rvalue>**.

7.4.2.3.4 Modifiers

void reset() noexcept;

Effects: If **has_value()** is true, destroys the contained value.

Postconditions: **has_value()** is false.

template<class S_T, class A>
void assign(S_T&& value, A&& alloc);

Effects: Constructs an object **tmp** of type **value_addresser** with **std::forward<T>(value)** and **std::forward<A>(alloc)** and **tmp.swap(*this)**. No effects if an exception is thrown.

Throws: Any exception thrown during the construction of **tmp**.

void swap(static_addresser& rhs) noexcept;

Effects: Exchanges the states of ***this** and **rhs**.

7.4.2.3.5 Observers

bool has_value() const noexcept;

Returns: **true** if ***this** contains an object, otherwise **false**.

const type_info& type() const noexcept;

Requires: **R** shall be false.

Returns: **typeid(T)** if ***this** has a contained value of type **T**, otherwise **typeid(void)**.

Note: Useful for querying against types known either at compile time or only at runtime.

7.4.2.4 Class template `reference_addresser`

```
template <class M, qualification_type Q>
class reference_addresser {
 protected:
```

```
  reference_addresser(const reference_addresser&) noexcept;
  template <class T>
  reference_addresser(T& value) noexcept;
  reference_addresser& operator=(const reference_addresser& rhs);
  template <class T>
  reference_addresser& operator=(T& value) noexcept;

  const M& meta() const noexcept;
  auto erased() const noexcept;
};
```

reference_addresser is the build-in addresser for reference semantics and is trivially-copyable.

7.4.2.4.1 Construction

```
template <class T>
  reference_addresser(T& value) noexcept;
```

Effects: Construct an object associating to **value**.

7.4.2.4.2 Assignment

```
Template <class T>
  reference_addresser& operator=(T& value) noexcept;
```

Effects: Equivalent to ***this = reference_addresser{value}**.

Returns: ***this**.

7.4.2.4.3 Addressing

```
const M& meta() const noexcept;
```

Returns: A const reference of the metadata.

```
see below erased() const noexcept;
```

Returns: A value implicitly convertible to **erased_reference<_Q, reference_type::lvalue>** where **_Q** is equal to **Q** or is a more strict qualifier,

- if **Q** is **qualification_type::none**, **_Q** could be any value of type **qualification_type**, or
- if **Q** is **qualification_type::const_qualified**, **_Q** could be **qualification_type::const_qualified**, **qualification_type::cv_qualified**, or
- if **Q** is **qualification_type::volatile_qualified**, **_Q** could be **qualification_type::volatile_qualified** or **qualification_type::cv_qualified**, or
- otherwise, if **Q** is **qualification_type::cv_qualified**, **_Q** could be **qualification_type::cv_qualified**.

7.5 Proxy

The implementations for the class templates **proxy_meta** and **proxy** rely on compile-time programming techniques. For a well-formed facade **F**, **proxy_meta<F>** is the type of metadata readable by corresponding proxies.

7.5.1 Class template proxy_meta

```
template <class F, template <qualification_type, reference_type> class E>
struct proxy_meta {
 public:
  template <class T> constexpr explicit proxy_meta(in_place_type_t<T>) noexcept;
};
```

An instantiation of **proxy_meta** is the type of metadata required by the **proxy**. **F** shall be a facade type and **E** shall be a template for erased type.

```
template <class T>
  constexpr explicit proxy_meta(in_place_type_t<T>) noexcept;
```

Requires: For any expression defined in **F**, it shall be well-formed with operand **e.template cast<T>()**, where **e** is a value of an instantiation **IE** of **E** with specific qualification type **Q** and reference type **R**.

Q is defined as

- **qualification_type::none** if the expression is not qualified, or
- **qualification_type::const_qualified** if the expression is const-qualified, or
- **qualification_type::volatile_qualified** if the expression is volatile-qualified, or
- **qualification_type::cv_qualified** if the expression is cv-qualified.

R is defined as

- **reference_type::lvalue** if the expression has lvalue reference semantics, or
- **reference_type::rvalue** if the expression has rvalue reference semantics.

Effects: Construct an object with required metadata.

7.5.2 Class template proxy

```
template <class F, class A>
class proxy : public A {
 public:
  proxy(const proxy&) = default;
  proxy(proxy&&) = default;
  template <class... Args>
  proxy(Args&&... args) : A(forward<Args>(args)...) {}
  proxy& operator=(const proxy& rhs) = default;
  proxy& operator=(proxy&& rhs) = default;
  template <class T>
```

```
  proxy& operator=(T&& value) {
    A::operator=(forward<T>(value));
    return *this;
  }

  other facade-specific functions
};
```

7.5.2.1 Facade-specific Functions

The member functions above are the basic ones for the proxy. Proxies with different facades has different specific member functions. For static expressions, non-static member functions shall be defined in the **`proxy`** with no reference semantics or qualifications. For non-static expressions, non-static member functions shall be defined in the **`proxy`** with corresponding reference semantics and qualifications.

For example, if a proxy is specified by a facade **`Callable`**:

```
template <class T>
facade Callable;  // undefined

template <class R, class... Args>
facade Callable<R(Args...)> {
  R operator()(Args...);
};
```

The implementation of the partial specialized class template:

```
template <class R, class... Args, class A>
class proxy<Callable<R(Args...)>, A>
```

should have a facade-related member function:

```
R operator()(Args... args) &;
```

The implementation of the function should equivalent to:

```
using T = the type that A::meta() is instantiated with;
return A::erased().template cast<T>(forward<Args>(args)...);
```

8 Summary

The PFA is an extendable and efficient solution for polymorphism, and I am looking forward that it becomes a part of the fascinating C++ programming language. However, some definitions of operator are omitted in the "Technical Specification" part due to limited time. I hope the committee could help in this respect. Although there is a long way to go, I believe this feature will largely improve the usability of the C++ programming language, especially in large-scale programming.

Please find the tested implementation of this library with this link, which compiles with latest GCC, LLVM and MSVC.