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

Document number:	P0957R0
Date:	2018-02-12
Project:	Programming Language C++
Audience:	CWG, EWG, LWG, LEWG, SG7, SG9

Authors:	Mingxin Wang
Reply-to:	Mingxin Wang <wmx16835vv@163.com>

Table of Contents

1 Introduction

PFA is a generic, extendible and efficient solution for polymorphic programming.

PFA is based on OOP (Object-oriented Programming), which consists of 3 concepts: Proxy, Facade, and Addresser. The "Proxy" is the component that performs polymorphism; the "Facade" defines the polymorphic expressions; the "Addresser" defines how to address the data and metadata. Providing well-formed Facades and Addressers, specific Proxies could be constructed to manage and use various types of objects with specific addressing strategy at runtime. For Addressers, not only are users able to use some typical built-in implementations, but also to define new addressing strategy referring to specific requirements if necessary, for example, varieties of GC (Garbage Collection) algorithms.

PFA combines the idea of OOP 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 requirements, and could carry richer semantics.

With the template meta programming mechanism, PFA is well compatible with the C++ programming language and makes C++ easier to use. Actually, PFA can be applied in almost every case that relates to virtual functions more elegantly. Naturally, 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 2 illustrates the motivation and scope of PFA; Section 3 illustrates PFA’s impact on the C++ standard; Section 4 includes the pivotal decisions in the design; Section 5 describes a typical and meaningful use cases indicating the basic usage; Section 6 illustrates the technical specification of PFA in C++; Section 7 lists some of the future works to be done and summarizes the paper.

2 Motivation

2.1 Limitations in Traditional OOP

```
class Base {
 public:
  virtual void fun_a() = 0;
  virtual void fun_b() { /* Function B for Base Class */ }

 protected:
  ~Base() { /* Destroy Base */ }

 private:
  /* Data */
};

class Derived : public Base {
 public:
  void fun_a() override { /* Function A for Derived Class */ }
  void fun_b() override { /* Function B for Derived Class */ }
  void fun_c() { /* Function C for Derived Class */ }

 private:
  /* Data */
};
```

Figure 1

"Inheritance" and "Virtual" are two fundamental keywords in traditional OOP. For polymorphism requirements in the C++ programming language, users are encouraged to declare a member function in a class to be virtual and inherit the class for various implementations, as is shown in Figure 1.

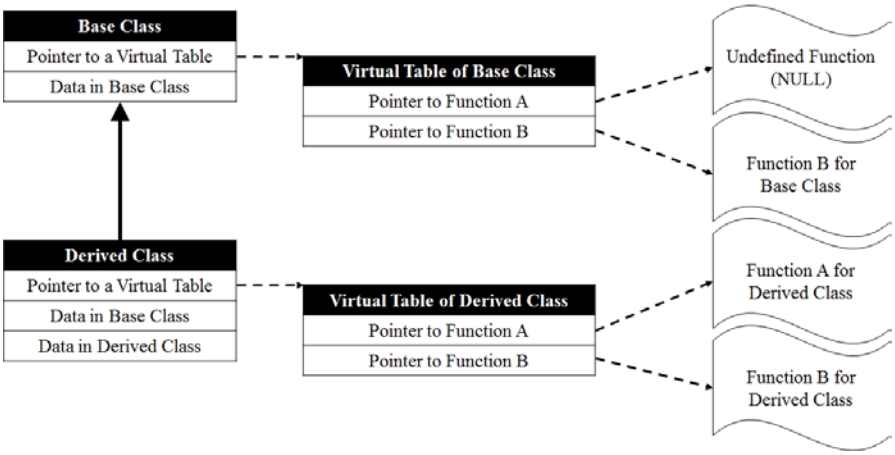

Figure 2

A widely adopted method to implement virtual functions is to introduce "virtual table", which is a data structure that supports random addressing for functions, whose memory layout at runtime is as shown in Figure 2Figure 2. A big limitation of such "inheritance-based polymorphism" in traditional OOP is "intrusion", which has adverse effects on both usability and performance.

Actually, PFA has no intrusion to existing implementation at all. Moreover, PFA allows a type to have different polymorphism support in different contexts, which could prevent users from misoperations and improve the performance to a certain extent.

2.1.1 Usability

```
class Base {
 public:
  /**
   * fun_a() is deleted in Base class
   */
  // virtual void fun_a() = 0;
  virtual void fun_b() { /* Function B for Base class */ }

 protected:
  ~Base() { /* Destroy Base */ }

 private:
  /* Data */
};

class Derived : public Base {
 public:
  /**
   * The following line of code becomes ill-formed
   * because fun_a() is no longer virtual in Base class
   */
  void fun_a() override { /* Function A for Derived class */ }
  void fun_b() override { /* Function B for Derived class */ }
  void fun_c() { /* Function C for Derived class */ }

 private:
  /* Data */
};
```

Figure 3

Take the code snippet in Figure 1 as an example. Providing it turned out to be more reasonable to delete polymorphism support on Base::fun_a(), it is required to reconstruct the Derived class if virtual function fun_a() is simply deleted from the Base class, as is shown in Figure 3.

```
class AnotherBase {
 public:
  virtual void fun_c() { /* Function C for Base class */ }
};

class Derived : public Base, public AnotherBase {
 public:
  void fun_a() override { /* Function A for Derived class */ }
  void fun_b() override { /* Function B for Derived class */ }
  void fun_c() override { /* Function C for Derived class */ }

 private:
  /* Data */
};
```

Figure 4

When it is required to add polymorphism to an existing implementation with traditional OOP mechanism, it is usually unavoidable to reconstruct the implementation, as is shown in Figure 4. However, some implementations are not reconstructible at all, e.g., implementations for the standard library or some other 3rd-party libraries. In such scenarios, users have no alternative but to implement extra middleware themselves to add polymorphism support to existing implementations.

2.1.2 Performance

```
class Derived : public Base0, public Base1, public Base2 {
 public:
  void derived_fun() { /* Non-virtual Function */ }
};
```

Figure 5

When there are no polymorphic requirements within the lifecycle of an entity that has virtual functions, the space for polymorphism support is wasted, especially in the cases of multiple inheritance. For example, when a class inherits from 3 base classes, which have virtual functions only, as is shown in Figure 5, the size of the Derived class equals to three times of the space required for polymorphism support. If implementation uses the virtual table, the space required for polymorphism support is usually not less than a pointer. If the Derived class is used without polymorphism, e.g. only its non-virtual member functions are called, the space for polymorphism support is wasted.

```
class RunnableBase {
 public:
  virtual void operator()() = 0;
};
```

Figure 6

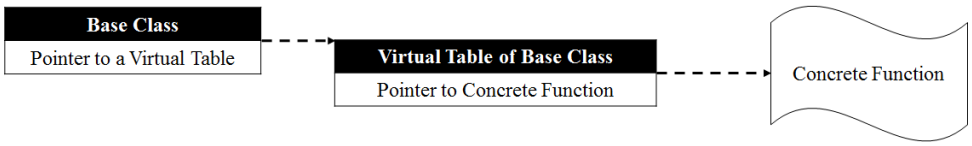

Figure 7

It is widely accepted that the virtual table is efficient in polymorphic programming. However, it is not always the efficient enough, especially when there is only one virtual function required to be polymorphic, as is shown in Figure 6, whose memory layout is shown in Figure 7.

Figure 8

When there are no requirements to extend the virtual table in the future, it is more reasonable to access the concrete function with a direct pointer instead of looking up in a virtual table, as is shown in Figure 8. This seems easy to implement, but unfortunately, since there is no mechanism for restricting the extension of the number of virtual functions in a non-final class in C++, the structure of the virtual tables must be consistent for potential requirements in upward transformation.

When it is required to manage the lifetime of a polymorphic entity in C++, especially in a concurrent program, operator `new` and `delete` are often used with virtual destructors (even if the "smart pointers" are used). However, the virtual destructors sometimes have bottleneck in performance, especially when the polymorphic entities are small in size, or

their types are trivial, which is a part of the motivation that some implementations of the class template `std::function` adopts the method of SOO (Small Object Optimization) to increase performance in managing the lifetime of those entities small in size. Unfortunately, there also are limitations in the class template `std::function` and traditional FP.

2.2 Limitations in FP

Anonymous lambda expressions simplify the definition of single dimensional logic to a certain extent, and users are able to wrap any functor into a uniform wrapper, "std::function". However, FP can do nothing for multidimensional logic and cannot carry enough semantics. Moreover, since the standard does not provide users with the interface to configure std::function, the performance may have room for improvement on specific demand, e.g., the size of SOO or the memory management strategies.

PFA can not only be applied to single dimensional logic requirements, but also to multidimensional ones with rich semantics. Besides, users are free to choose addressing strategies on specific demand to optimize performance. Two different addressing strategies may be convertible to one another.

3 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 today 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.

4 Design Decisions

4.1 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.

4.2 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 "virtual tables".

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.

4.3 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.

4.4 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) (https://github.com/andyprowl/virtual-concepts/blob/master/draft/Dynamic%20Generic%20Programming%20with%20Virtual%20Concepts.pdf) 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.

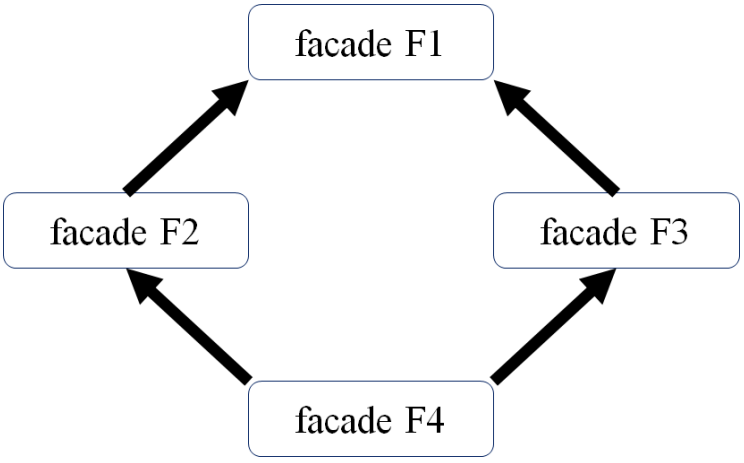

Figure 9

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; one facade A is convertible to another facade B if and only if the topological order of B is prior to A in the inheritance DAG. For example, in the case of "diamond inheritance", as is shown in Figure 9, facade F4 is directly convertible to F1.

4.5 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 smilar lifecycle as the entity being represented does, besides that std::any is CopyConstructible while std::unique_ptr is not.

4.5.1 Data and Metadata

Data is a straight forward 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.

4.5.2 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 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.

According to the UML (Unified Modeling Language), the 3 common relationships between entities and their users are "Dependency", "Aggregation" and "Composition", which corresponds to pointers, class template "std::shared_ptr" and "std::unique_ptr" in C++. However, pointers and "smart pointers" are type-specific, while the "Addressers" required shall be type-erased. In order to keep consistency with the standard, the addressers with the 3 lifetime management strategies are named as "direct addresser", "shared addresser" and "unique addresser".

The "direct addresser" has no responsibility in lifetime management issue, and saves the metadata and the row pointer of data. When the metadata is small in size, metadata could be saved directly without extra addressing operations (as is shown in Figure 8).

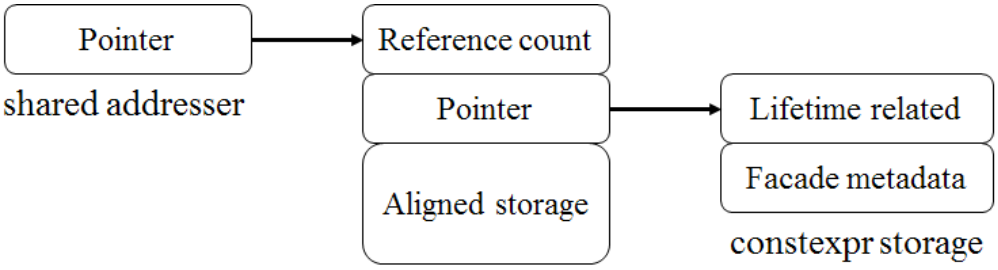

Figure 10

The "shared addresser" has similar properties as std::shared_ptr does. However, because the storage of the data is type-erased, implementation should pay attention to the alignment of various types. A possible memory layout of the "shared addresser" (in the sample implementation) is shown in Figure 10.

Actually, std::any is an available solution for composition relationship. However, std::any could carry limited metadata (the type info of the type being represented), and thus has little extendibility. The same as aggregation relationship, composition relationship requires runtime polymorphism to control the lifecycle of the type-erased entity. Implementation may support SOO (Small Object Optimization) to improve performance.

4.6 Casting

Any proxy shall be implicitly convertible to another proxy with compatible type specifications. In other words, a proxy type P1 specified with a facade type F1 and an addresser type A1 shall be convertible to another proxy type P2 specified with a facade type F2 and an addresser type A2 iff and A1 is convertible to A2.

5 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(
   std::proxy<ImmutableMap<int, std::string>, std::direct_addresser>);
```

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);
```

As some feedback suggests that the type "std::proxy<ImmutableMap<int, std::string>, std::direct_addresser>" is too long, there are type aliases for the proxies with built-in addresser types. The type "std::proxy<ImmutableMap<int, std::string>, std::direct_addresser>" could be abbreviated as " std::direct_proxy<ImmutableMap<int, std::string>>".

Providing this function is asynchronous, and the "map" is shared by multiple threads, the signature of the function could be redefined as:

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

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, "in_place" tags are supported as std::any does. For example, the following expression is well-formed:

```
std::shared_proxy<ImmutableMap<int, std::string>>
   p{std::in_place_type<std::map<int, std::string>>};
```

Additionally, in the implementation of the async version, std::shared_proxy can implicitly be converted to std::direct_proxy to increase performance, like converting an std::shared_ptr to a pointer within its lifetime.

6 Technical Specifications

6.1 Header <proxy> synopsis

```
namespace std {
```

```
// class template proxy
template <class F, template <class> class AT> class proxy;

// class template facade_meta_t
template <class F>
struct facade_meta_t;

// proxy traits
template <class P> struct is_proxy;
template <class P> inline constexpr bool is_proxy_v = is_proxy<P>::value;

// class null_proxy_t
struct null_proxy_t { explicit null_proxy_t() = default; };
inline constexpr null_proxy_t null_proxy {};

// class template unique_addresser
template <class M> class unique_addresser;

// class template shared_addresser
template <class M> class shared_addresser;

// class template direct_addresser
template <class M> class direct_addresser;

// type aliases
template <class F> using unique_proxy = proxy<F, unique_addresser>;
template <class F> using shared_proxy = proxy<F, shared_addresser>;
template <class F> using direct_proxy = proxy<F, direct_addresser>;

// hash support
template <class T> struct hash;
template <class M> struct hash<unique_addresser<M>>;
template <class M> struct hash<shared_addresser<M>>;
template <class M> struct hash<direct_addresser<M>>;
template <class F, template <class> class AT> struct hash<proxy<F, AT>>;

}
```

6.2Facade

The "Facade" is a descriptive placeholder that abstracts the types that have specific expressions and semantics. A general declaration of a facade is as follows:

```
facade `name` [: `more_abstract_facade_1`, `more_abstract_facade_2`, ...] {
```

```
  [static] `return_type` `function_name`([`arg_1`, `arg_2`, ...]);
  [...]
};
```

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.

6.3 Addresser

6.3.1 Requirements for Addresser Types

6.3.1.1 `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.

6.3.1.2 `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.data()`

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

Return type: any type convertible to **`void*`**.

6.3.2 Addressers

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

6.3.2.1 Class template unique_addresser

A unique addresser is an object, satisfying the Addresser requirements, managing a user-supplied value of some type and corresponding metadata with exclusive strategy.

```
template <class M> class unique_addresser {
public:
  // construction and destruction
  constexpr unique_addresser() noexcept;
  unique_addresser(unique_addresser&& other) noexcept;
  template <class T, class... Args>
    explicit unique_addresser(in_place_type_t<T>, Args&&... args);
  ~unique_addresser();

  // assignment
  unique_addresser& operator=(unique_addresser&& rhs) noexcept;

  // addressing
  void* data() const noexcept;
  const M& meta() const noexcept;

  // modifiers
  template <class T, class... Args>
    T& emplace(Args&&... args);
  void reset();
  template <class T> void reset(T&& rhs);
  void swap(unique_addresser& rhs) noexcept;

  // observers
  bool has_value() const noexcept;
  const type_info& type() const noexcept;
};
```

6.3.2.1.1 Construction and Destruction

`constexpr unique_addresser() noexcept;`

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

`unique_addresser(unique_addresser&& other) noexcept;`

Effects: If **`other.has_value()`** is **`false`**, constructs an object that has no value. Otherwise, constructs an object of type **`unique_addresser`** that contains either the contained value of **`other`**, or contains an object of the same type constructed from the contained value of **`other`** considering that contained value as an rvalue.

Postconditions: **`other`** is left in a valid but otherwise unspecified state.

`template <class T, class... Args>`
`explicit unique_addresser(in_place_type_t<T>, Args&&... args);`

Requires: **`T`** shall be consistent with **`decay_t<T>`**.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type **`T`** with the arguments **`std::forward<Args>(args)...`**.

Postconditions: **`*this`** contains a value of type **`T`**.

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

Remarks: This constructor shall not participate in overload resolution unless **`is_same_v<T, decay_t<T>>`** is **`true`** and **`is_constructible_v<T, Args...>`** is **`true`**.

`~unique_addresser();`

Effects: As if by reset().

6.3.2.1.2 Assignment

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

Effects: As if by **`unique_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.

6.3.2.1.3 Addressing

`void* data() const noexcept;`

Requires: **`*this`** shall be initialized with a value.

Returns: The pointer of the stored value.

`const M& meta() const noexcept;`

Requires: **`*this`** shall be initialized with a value.

Returns: The constant reference of the related metadata.

6.3.2.1.4 Modifiers

`template <class T, class... Args>`
`T& emplace(Args&&... args);`

Requires: **T** shall be consistent with **decay_t<T>**.

Effects: Calls **reset()**. Then initializes the contained value as if direct-non-list-initializing an object of type **T** with the arguments **std::forward<Args>(args)...**.

Postconditions: ***this** contains a value.

Returns: A reference to the new contained value.

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

Remarks: If an exception is thrown during the call to **T**'s constructor, ***this** does not contain a value, and any previously contained value has been destroyed. This function shall not participate in overload resolution unless **is_same_v<T, decay_t<T>>** is true and **is_constructible_v<T, Args...>** is true.

```
void reset() noexcept;
```

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

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

```
template<class T>
  void reset(T&& rhs);
```

Let **VT** be **decay_t<T>**.

Effects: Constructs an object **tmp** of type unique_addresser that contains an object of type **VT** direct-initialized with **std::forward<T>(rhs)**, and **tmp.swap(*this)**. No effects if an exception is thrown.

Returns: ***this**.

Remarks: This operator shall not participate in overload resolution unless **VT** is not the same type as **unique_addresser**.

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

```
void swap(unique_addresser& rhs) noexcept;
```

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

6.3.2.1.5 Observers

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

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

```
const type_info& type() const noexcept;
```

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.

6.3.2.1.6 Non-member Functions

```
void swap(unique_addresser& x, unique_addresser& y) noexcept;
```

Effects: As if by **x.swap(y)**.

6.3.2.2 Class template shared_addresser

The "shared addresser" is similar with "unique addresser". The only difference between the two is that the "shared addresser" is **`CopyConstructible`** and **`CopyAssignable`**, while the "unique addresser" is not.

```
template <class M> class shared_addresser {
public:
  // construction and destruction
  constexpr shared_addresser() noexcept;
  shared_addresser(shared_addresser&& other) noexcept;
  shared_addresser(const shared_addresser& other) noexcept;
  template <class T, class... Args>
    explicit shared_addresser(in_place_type_t<T>, Args&&... args);
  ~shared_addresser();

  // assignment
  shared_addresser& operator=(shared_addresser&& rhs) noexcept;
  shared_addresser& operator=(const shared_addresser& rhs) noexcept;

  // addressing
  void* data() const noexcept;
  const M& meta() const noexcept;

  // modifiers
  template <class T, class... Args>
    T& emplace(Args&&... args);
  void reset();
  template <class T> void reset(T&& rhs);
  void swap(shared_addresser& rhs) noexcept;

  // observers
  bool has_value() const noexcept;
  const type_info& type() const noexcept;
};
```

In order to save space, some technical specifications for the class template shared_addresser are omitted.

6.3.2.3 Class template direct_addresser

The "direct addresser" is much simpler than the previous two addresser class templates, but could be constructed from any other valid addresser types that has convertible metadata type.

```
template <class M> class direct_addresser {
```

```
public:
  // construction and destruction
  constexpr direct_addresser() noexcept;
  direct_addresser(direct_addresser&& other) noexcept;
  direct_addresser(const direct_addresser& other) noexcept;
  template <class A>
    direct_addresser(A&& rhs);
  template <class T, class U>
    explicit direct_addresser(in_place_type_t<T>, U& rhs);
  ~direct_addresser();

  // assignment
  direct_addresser& operator=(direct_addresser&& rhs) noexcept;
  direct_addresser& operator=(const direct_addresser& rhs) noexcept;
  template <class A>
    direct_addresser& operator=(A&& rhs);

  // addressing
  void* data() const noexcept;
  const M& meta() const noexcept;

  // modifiers
  void reset();
  template <class T> void reset(T& rhs);
  void swap(direct_addresser& rhs) noexcept;

  // observers
  bool has_value() const noexcept;
};
```

6.3.2.3.1 Constructor from Other Addresser Types

`template <class A>`
`direct_addresser(A&& rhs);`

Requires: **`A`** meets the **`Addresser`** requirements, and **`rhs`** shall be initialized with a value.

Effects: Initialize the contained data and metadata from **`rhs`** by calling **`rhs.data()`** and **`rhs.meta()`**.

In order to save space, some technical specifications for the class template direct_addresser are omitted.

6.3.2.4 Addresser hash support

```
template <class M> struct hash<unique_addresser<M>>;
template <class M> struct hash<shared_addresser<M>>;
```

```
template <class M> struct hash<direct_addresser<M>>;
```

These specializations are enabled.

6.4 Proxy

6.4.1 Compiler-dependent Type Templates

The implementations for the class templates facade_meta_t and proxy rely on the compiler. For a well-formed facade F, facade_meta_t<F> is the type of metadata readable by corresponding proxies.

```
template <class F, template <class> class AT>
class proxy : public AT<facade_meta_t<F>> {
 public:
  // Type aliases
  using facade_type = F;
  using meta_type = facade_meta_t<facade_type>;
  template <class M> using addresser_template = AT<M>;
  using addresser_type = AT<meta_type>;

  // Construction
  proxy() : addresser_type() {}
  proxy(null_proxy_t) : addresser_type() {}
  proxy(const proxy&) = default;
  template <class _F, template <class> class _AT>
  proxy(const proxy<_F, _AT>& rhs)
      : addresser_type(static_cast<const _AT<facade_meta_t<_F>>&>(rhs)) {}
  proxy(proxy&&) = default;
  template <class _F, template <class> class _AT>
  proxy(proxy<_F, _AT>&& rhs)
      : addresser_type(static_cast<_AT<facade_meta_t<_F>>&&>(rhs)) {}
  template <class T, class = enable_if_t<!is_proxy_v<decay_t<T>>>>
  proxy(T&& value) : proxy(in_place_type<decay_t<T>>, forward<T>(value)) {}
  template <class T, class U, class... _Args,
          class = enable_if_t<is_same_v<T, decay_t<T>>>>
  explicit proxy(in_place_type_t<T>, initializer_list<U> il, _Args&&... args)
      : addresser_type(in_place_type<T>, il, forward<_Args>(args)...) {}
  template <class T, class... _Args,
          class = enable_if_t<is_same_v<T, decay_t<T>>>>
  explicit proxy(in_place_type_t<T>, _Args&&... args)
      : addresser_type(in_place_type<T>, forward<_Args>(args)...) {}
  template <class... _Args, class = enable_if_t<(sizeof...(_Args) > 1u)>>
  explicit proxy(_Args&&... args) : addresser_type(forward<_Args>(args)...) {}
```

```
  // Assignment
  proxy& operator=(null_proxy_t) {
    addresser_type::reset();
    return *this;
  }

  template <class T, class = enable_if_t<!is_proxy_v<decay_t<T>>>>
  proxy& operator=(T&& value) {
    addresser_type::reset(forward<T>(value));
    return *this;
  }

  proxy& operator=(const proxy& rhs) = default;

  template <class _F, template <class> class _AT>
  proxy& operator=(const proxy<_F, _AT>& rhs) {
    static_cast<addresser_type&>(*this) =
        static_cast<const _AT<facade_meta_t<_F>>&>(rhs);
    return *this;
  }

  proxy& operator=(proxy&& rhs) = default;

  template <class _F, template <class> class _AT>
  proxy& operator=(proxy<_F, _AT>&& rhs) {
    static_cast<addresser_type&>(*this) =
        static_cast<_AT<facade_meta_t<_F>>&&>(rhs);
    return *this;
  }

  /* Other facade related functions */
};
```

The member functions above are the basic ones for the proxy. Proxies with different facades usually has different additional member functions. For example, if a proxy is specified by a Callable facade:

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

should have a facade-related member function:

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

For example, if the implementation of the specialized class template facade_meta_t for facade template Callable is as follows:

```
template <class R, class... Args>
struct facade_meta_t<Callable<R(Args...)>> {
  template <class, template <class> class>
  friend class proxy;

 public:
  template <class T>
  constexpr explicit facade_meta_t(in_place_type_t<T>)
      : callable_op_0_(callable_op_0<T>) {}

  constexpr facade_meta_t(facade_meta_t&) = default;

 private:
  template <class T>
  static R callable_op_0(void* data, Args... args) {
    if constexpr(is_void_v<R>) {
      (*static_cast<T*>(data))(forward<Args>(args)...);
    } else {
      return (*static_cast<T*>(data))(forward<Args>(args)...);
    }
  }

  R (*callable_op_0_)(void*, Args...);
};
```

The implementation of the member function template should be as follows:

```
template <class R, class... Args, class AT, class... _Args>
R proxy<Callable<R(Args...)>, AT>::operator()(_Args... args) {
  return addresser_type::meta().callable_op_0_(
      addresser_type::data(), forward<Args>(args)...);
}
```

6.4.2 Type traits

```
template <class P>
```

```
  struct is_proxy : false_type {};

template <class F, template <class> class AT>
  struct is_proxy<proxy<F, AT>> : true_type {};

template <class P>
  inline constexpr bool is_proxy_v = is_proxy<P>::value;
```

The class template is_proxy is a type traits for the proxy.

6.4.3 Class null_proxy_t

```
struct null_proxy_t {
  explicit null_proxy_t() = default;
};
inline constexpr null_proxy_t null_proxy {};
```

The class null_proxy_t is a tag for empty proxies.

6.4.4 Type Aliases

```
template <class F>
using unique_proxy = proxy<F, unique_addresser>;

template <class F>
using shared_proxy = proxy<F, shared_addresser>;

template <class F>
using direct_proxy = proxy<F, direct_addresser>;
```

In order to shorter the name when using the proxy, the type aliases are introduced.

6.4.5 Proxy hash support

```
template <class F, template <class> class AT> struct hash<proxy<F, AT>>;
```

The specialization is enabled, and shall be equivalent to hash<AT<facade_meta_t<F>>>.

6.5 Illustrative Example

The illustrative example shows how to program with PFA.

6.5.1 Define Facades

Suppose there are three facade declarations:

```
facade FA {
  void fun_a_0();
  int fun_a_1(double);
};

facade FB {
  static void fun_b(unique_proxy<FA>);  // fun_b accepts any type that has the facade
FA, and manages it lifecycle
};

facade FC : FA, FB {
  void fun_c();
};
```

6.5.2 Implement Facades

Here are some corresponding implementations for the facades:

```
// Has facade FA
class FaImpl {
 public:
  // OK: fun_a_0() is a well-formed expression.
  int fun_a_0();

  // OK: fun_a_1(double) is a well-formed expression, because
  //    double is implicitly convertible to float;
  // The result type is convertible to int.
  int fun_a_1(float);
};

// Has facade FB
class FbImpl {
 public:
  // OK: fun_b(unique_proxy<FA>) is a well-formed expression, because
  //    direct_proxy<FA> is implicitly constructible from unique_proxy<FA>, because
  //       direct_addresser is implicitly constructible from composed_addresser.
  void fun_b(std::direct_proxy<FA>);
};
```

```
// Has facade FC
class FcImpl {
 public:
  // OK: fun_a_0() is a well-formed expression;
  int fun_a_0();

  // OK: fun_a_1(double) is a well-formed expression;
  // The result type is convertible to int.
  int fun_a_1(double);

  // OK: fun_b(unique_proxy<FA>) is a well-formed expression.
  void fun_b(std::unique_proxy<FA> p);

  // OK: fun_c() is a well-formed expression.
  void fun_c();
};
```

6.5.3 Polymorphic Programming with Facades

The following expressions are well-formed:

```
// p1 is a proxy with facade FC, because
//    FcDemo1 has facade FC, and
//    FcDemo1 is constructible with an integer.
std::shared_proxy<FC> p1(std::in_place_type<FcImpl>, 8);

// FcDemo1::fun_a_0() is called.
p1.fun_a_0();

// FcDemo1::fun_a_1(double) is called.
p1.fun_a_1(1.5);

// FcDemo1::fun_b(unique_proxy<FA>) is called, because
//    FaDemo1 is convertible to unique_proxy<FA>, because
//       FaDemo1 has facade FA, and
//       FaDemo1 is MoveConstructible.
p1.fun_b(FaImpl{123});

// FcDemo1::fun_c() is called.
p1.fun_c();

// p2 is a proxy with facade FA, because
//    direct_proxy<FA> is constructible from shared_proxy<FC>, because
//       direct_addresser is constructible from shared_addresser, and
```

```
//        facade FA is base of facade FC;
// The validity of p2 depend on the validity of p1, because
//    direct_proxy<FA> does not store the concrete data used by p1, because
//        direct_addresser only stores the address of the concrete data, and
//        direct_addresser is not responsible for managing the life-cycle issue.
std:: direct_proxy<FA> p2(p1);

// FcDemo1::fun_a_0() is called.
p2.fun_a_0();
```

6.6 Implementation Prototype

A sample implementation for the PFA can be found at: https://github.com/wmx16835/cpp_pfa.

7 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, there are still much work to do before standardization:

1. Due to limited time, some definitions of operator are omitted in the "Technical Specification" part. I hope the standard committee could help in this respect.
2. In this paper, users are not able to specify the algorithms in memory allocation for the class templates "unique_addresser" and "shared_addresser". Actually, I have already implemented a configurable version for the two class templates. However, it challenges the concept Allocator and the PMR library in C++, and I am still working on it. (see the discussion here: https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/iw3JRVF8EPk)
3. According to the Ranges TS, some general facades shall be defined in the standard, for example, the facade template Callable and facade ImmutableMap.
4. Besides the three addresser class templates, I designed another addresser template only for trivial types with satisfying performance. However, as it has a maximum size for trivial types, and I could not find an optimal configuration yet, it was not included in this paper.
5. Classes with virtual functions supports upwards transition (that is, a derived class with new virtual functions could convert to a base class with a bunch of virtual functions). Although the "direct addresser" could do that, the "unique addresser" and "shared addresser" does not support the operation. Adding such support to "unique addresser" and "shared addresser" is possible, but will restrict the memory layout of the implementation to some extent. Moreover, if this operation should support multiple inheritance, there may be performance loss in the implementation comparing to the prototype provided.

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.

8 Acknowledgement

Thanks to my parents for their wholehearted support.

Thanks to dear Linping Zhang, Bengt Gustafsson, Wei Chen, Nicol Bolas, Jakob Riedle, Christopher Di Bella, Thiago Macieira, Tony V E, Myriachan, Barry Revzin, Bryce Glover, Aarón Bueno Villares, Magnus Fromreide, Joël Lamotte, and Chuang Li for their support and valuable feedback.

Thanks to my colleagues, Xiangliang Meng, Xiang Fu, Shizhi Zhu, Junyu Lin, Yaya Zhang, Shuangxing Zhang, Shujun Xiong, Hao Ren, Niping Chen, Ruihao Zhang for their understanding and support.