Flat Dynamic Polymorphism library

Document number:	p???
Date:	2022-07-16
Project:	Programming language C++
Audience:	LEWG
Author:	Kelbon
Reply-to:	Kelbon kelbonage@gmail.com

Contents

1 Introduction

Since there are architecting limitations in existing mechanisms of dynamic polymorphism, the flat dynamic polymorphism(FDP) is proposed as a generic, extendable, and efficient bridge between static and dynamic polymorphism.
It can largely replace the existing "virtual mechanism", better express the intentions of the programmer in the code, improve perfomance (for example by reducing number of allocations) and less error prone(virutal destructors, slicing, memory management etc)
FDP goes well with existing mechanisms in C++, such as overloading and templates, so it makes it easier to use and learn C++. It can be used in almost any case where virtual functions are currently used, with improved performance and code readability.
The types such as std::function<R(Args...)>, std::any, std::move_only_function<R(Args...)> are special cases of FDP and therefore FDP is a development and generalization of the approaches existing in the C++ standard.
As proof of concept, i have implemented all proposed here as a single-header library, meeting the C++20 standard.
The implementation, including examples, could be found here

2 Design

2.1 Basic concepts

The library provides the following "type generators":

- any_with<Methods...>
- poly_ref<Methods...>
- poly_ptr<Methods...>
- const_poly_ref<Methods...>
- const_poly_ptr<Methods...>

typical usage:

```
using any_drawable = any_with<Draw, Method2, Method3>;
```

Where each Method in current implementation is a type template with one parameter(typename T) and addressable static member function named do_invoke, which first parameter represents erased type and must be exactly T, T&, const T&.
Example:

```
template <typename T>
struct Foo {
  static int do_invoke(const T& self, double value0, int value1) {
    return self.foo(value0, value1);
  }
};
```

Methods which 'self' type is a const T& or T(invoking by copy) are named "const methods".
Result type and all parameters types of do_invoke except the first must be independent from T, otherwise the program is ill-formed, no diagnostic required.
Below in article polymorphic value is an specialization of alias template any_with.

2.2 Polymorphic value, reference and pointer

Now we have nothing polymorphic with value semantic in standard library, except std::any, which is hard to use, because it can only destroy itself.
FDP provides abstraction of polymorphic value - type erased storage for one or zero values and std::decay_t<T> of last emplaced value is a dynamic stored type(DST).
Polymorphic value, reference and pointer are similar to non-polymorphic T, T&, T* as much as possible:
poly_ref may be created by dereferencing pointer.
poly_ref is converitble to const_poly_ref
poly_ptr is convertible to const_poly_ptr
operator& of poly_ref and polymorphic value returns poly_ptr.
Below in the article, polymorphic value, reference and pointer will be called the common name "polymorphic type".
Polymorphic types with more restrictions can be converted to polymorphic types with less restrictions
Example:

```
static_assert(
      std::is_convertible_v<
      poly_ref<M1, M2, M3, M4>,
      poly_ref<M2, M3>>);
```

3 Motivation

There are many cases, when you need dynamic polymorphism, but:

- don't need a hierarchy
- need both dynamic and static polymorphism without code duplication and performance loses
- want to separate polymorphic behavior and type
- just want to write simple and clear code that will reflect your intentions, rather than inefficient dealing with memory allocations and "throw "not implemented""

Let's take simple examples from the standard library:

3.1 Streams

You can see that standard streams use both virtual inheritance and virtual functions (stream destructor and a lot of virtual functions in the std::basic_streambuf), while all this is needed only to be able to .tie the stream, which is far from always needed.
If we could separate polymorphism and types, then we could write non-polymorphic stream types, which are essentially output/input iterators to the underlying buffer and some "any_iostream" type that would be used for situations where you need to be able to .tie the stream (for example, std::cout would be a poly_ref<Write>)

```
namespace std2 {
— Write and Read are Methods for FDP
template<typename T>
struct Write {...};

template<typename T>
struct Read{...};

template<typename OutBuf, typename Char,
typename Traits = std::char_traits<Char>>
struct basic_ostream;

template<typename InBuf, typename Char,
```

```
typename Traits = std::char_traits<Char>>
struct basic_istream;

using any_istreambuf = any_with<Read>;
using any_ostreambuf = any_with<Write>;
using any_iostreambuf = any_with<Read, Write>;

template<typename Char,
typename Traits = std::char_traits<Char>>
using legacy_basic_ostream =
basic_ostream<any_ostreambuf, Char, Traits>;
-- etc etc
}
-- and now spanstream/stringstream/etc are not use virtual functions!
```

We would gain in performance, remove virtual tables with destructors for streams(we really dont need it, because we never call std::cout destructor, isn't it?), and we would have much more flexibility in creating user defined streams.

3.2 polymorphic_allocator and memory resources

Here the situation is even more obvious. Do memory resources really need to be polymorphic? Why is a polymorphic allocator polymorphic and not a template from a memory resource?

```
template<typename T>
struct Allocate {...};

template<typename T>
struct Deallocate {...};

using any_memory_resource_ptr = poly_ptr<Allocate, Deallocate>;

template<typename T, typename ResourcePtr>
struct allocator {
  ResourcePtr ptr;
  ...
};
```

It gives us:

- both polymorphic and non polymorphic allocators with custom memory resources, which can be used in type erasing such as std::generator

(and not Allocator = void by default, which means "type erased", but any_memory_resource because generator dont needs typed allocator)(and not by default obviously)

- easy to create user defined resources(just type with allocate/deallocate)
- perfomance
- better semantics of the resulting code

3.3 shared_ptr

Another place where we pay for something. what we don't use: shared_ptr and its type erased deleter. Sometimes this is useful, but in the vast majority of cases it just wastes resources. With flat dynamic polymorphism we could do this:

```
— shared ptr in alternative universe
namespace std2 {

  template<typename U>
  struct delete_for {
    template <typename T>
    struct method {
      static constexpr void do_invoke(const T& self, U* value)
      noexcept {
          self(value);
      }
    };
  };
  template<typename T>
  using any_deleter_for =
  aa::any_with<aa::copy, aa::move, delete_for<T>::template method>;

  template <typename T,
  typename Deleter = ::std::default_delete<T>>
  struct shared_ptr;
  template<typename T>
  using legacy_shared_ptr =
  ::std2::shared_ptr<T, any_deleter_for<T>>;
}
```

4 Actions on polymorphic types

4.1 `invoke<Method>`

invoke<Method> is a functional object with operator(), which accepts polymorphic object and arguments of Method's do_invoke (except first, which represents erased type) and performs as if:

```
return Method<DST>::do_invoke(DynamicValue, static_cast<Args&&>(args)...);
```

4.2 `any_cast<T>`

accepts	returns	if DST is not `std::decay_t<T>`
poly_ptr	T*	nullptr
const_poly_ptr	`const std::remove_reference_t<T>*`	nullptr
poly_ref	std::remove_cv_t< T >	throws std::bad_cast
const_poly_ref	**see below**	throws std::bad_cast
any_x auto&&	`std::remove_cv_t<T>`	throws std::bad_cast
any_x auto*	`std::remove_cvref_t<T>*`	nullptr
const any_x auto*	`const std::remove_cvref_t<T>*`	nullptr

Where any_x is a concept of type created by basic_any_with<Alloc, SooS, Methods...> or inheritor of such type.
any_cast for const_poly_ref returns std:: conditional_t <std:: is_reference_v <T>, const std::remove_reference_t<T>&, std::remove_cv_t<T>>
If T is an array, function or (possibly cv) void, then program is ill-formed.

5 Interface

Library provides several from-box Methods:

- destroy - any_with has it by default, but references and pointers may not have it
- move - enables move ctor, move/copy assignment operators for polymorphic value
- copy_with<Alloc, SooS> which has inner template method<T> - enables copy ctor and copy assignment for polymorphic value with same Alloc and SooS
- type_id - enables enables any_cast for poly_ref/poly_ptr
- hash - enables specialization of std::hash for polymorphic types

5.1 `any_with<Methods...>`

any_with<Methods...> is an alias to basic_any_with<std::allocator<std::byte>, N, Methods...>, where N is an implementation defined number, which represents Small Object Optimization buffer size

```
template<typename Alloc, size_t SooS, template<typename> typename... Methods>
using basic_any_with = ...;
```

Lets call a type created by basic_any_with alias Any, then Any guaranteed to have following interface: (All constructors and copy/move assignment operators are provides strong exception guarantee, if emplace throws an exception, then Any is empty (has_value() == false))

```
struct *type created by basic_any_with* Any {
  template <template<typename> typename Method>
  static constexpr bool has_method;

  Any(); -- creates an empty Any
  -- from any type, if it satisfies requirements of Methods...
  Any(Alloc);
  Any(auto&& value);
  -- from any value, but with Alloc
  Any(const Any&) requires has_method<copy>;
```

```
  Any(std::allocator_arg_t, Alloc, auto&&);
  Any(Any&&) noexcept requires has_method<move>;

  Any& operator=(const Any&)
      requires has_method<move> && has_method<copy>;

  Any& operator=(Ant&&) noexcept requires has_method<move>;

  using ptr = poly_ptr<Methods...>;
  using ref = poly_ref<Methods...>;
  using const_ptr = const_poly_ptr<Methods...>;
  using const_ref = const_poly_ref<Methods...>;

  template <typename T, typename... Args>
  Any(std::in_place_type_t<T>, Args&&...);

  template <typename T, typename U, typename... Args>
  Any(std::in_place_type_t<T>, std::initializer_list<U>, Args&&...);

  bool has_value() const noexcept;

  -- returns reference to emplaced value
  template<typename T, typename... Args>
  std::decay_t<T>& emplace(Args&&...);

  template <typename T, typename U, typename... Args>
  std::decay_t<T>& emplace(std::initializer_list<U>, Args&&...);

  void reset() noexcept;

  poly_ptr<Methods...> operator&() noexcept;
  const_poly_ptr<Methods...> operator&() const noexcept;

};
```

5.2 poly_ref<Methods...>

Non owner, always not null, easy to create view to polymorphic value

```
template <template<typename> typename... Methods>
struct poly_ref {
  constexpr poly_ref(const poly_ref&) = default;
  constexpr poly_ref(poly_ref&&)      = default;
  -- cannot rebind reference
```

```
  void operator=(poly_ref&&)      = delete;
  void operator=(const poly_ref&) = delete;
  template<non_const T>
  constexpr poly_ref(T& value)
      noexcept requires(!std::same_as<poly_ref<Methods...>, T>);

  poly_ptr<Methods...> operator&() const noexcept;
};
```

5.3 poly_ptr<Methods...>

Non owner, nullable, easy to create view to polymorphic value. Trivially copyable.

```
template <template<typename> typename... Methods>
struct poly_ptr {
  poly_ptr() = default; -- creates null pointer
  poly_ptr(std::nullptr_t) noexcept;
  poly_ptr& operator=(std::nullptr_t) noexcept;

  poly_ptr(non_const auto* ptr) noexcept;
  template<any_x T>
  poly_ptr(T* ptr)
      noexcept requires(non_const<T> && *T has same Methods...*);

  void* raw() const noexcept; -- returns raw pointer to value

  bool has_value() const noexcept;
  bool operator==(std::nullptr_t) const noexcept;
  explicit operator bool() const noexcept;

  poly_ref<Methods...> operator*() const noexcept;
  const poly_ref<Methods...>* operator->() const noexcept;
};
```

5.4 const_poly_ref<Methods...>

Similar to poly_ref<Methods...>, but can be created from poly_ref<Methods...> or reference to const value. Not extends lifetime.

5.5 `const_poly_ptr<Methods...>`

Similar to poly_ptr<Methods...>, but can be created from poly_ptr<Methods...> or pointer to const value/polymorphic value.

6 Examples

6.1 Type erased print

Lets see simplest way to implement print for variable number of arguments:

```
template<typename... Ts>
void print(const Ts&... args) {
  (std::cout << ... << args);
}
int main() {
  print(5, 10, std::string{"abc"}, std::string_view{"hello world"});
}
```

But in this case we have different 'print' for each set of arguments, one for `<double, int>`, another for `<int, double>` etc.
Most effective way is to erase printing each type and then create only one print function for any count of them (similar to std::make_format_args)
How it looks with FDP:

```
template <typename T>
struct Print {
  static void do_invoke(const T& self) {
    std::cout << self;
  }
};
— we can remove initializer list here, but it is exposition only
void print(std::initializer_list<aa::const_poly_ref<Print>> list) {
  std::ranges::for_each(list, aa::invoke<Print>);
}

int main() {
  print({1, 2, std::string("hello"), std::string_view("world")});
}
```

6.1.1 About code generation

To check the compiler's ability to optimize such code, I now conducted several experiments, for example, here (exposition only) type erasure on std::visit. At the moment, with options to reduce the amount of generated code as much as possible, the clang can hardly improve the std::visit,

but with type erasure, it was able to reduce the amount of code by more than 10 times (and this value grows non-linearly with the number of variant types and the number of variants).
On the other hand, with maximum optimization for speed, the generated code is same. Line to line.
And here code generation for the main feature in polymorphic code - function calls. Again, identical code
And of course it would be difficult to avoid comparisons with one language, where such constructions are the only tool for polymorphism: link,

6.2 `fire_once`

For example, we are writing thread pool and we need a polymorphic object, which will store any function and we only need to call it once, we dont need RTTI, copy, move etc, how to express our intentions in code?

```
template<typename T>
struct invoke_once {
  static void do_invoke(T& self) {
    (void)self();
  }
};
using fire_once = any_with<invoke_once>;
```