Add make_proxy for the Pointer-Semantics-Based Polymorphism Library - Proxy

Project: ISO/IEC 14882 Programming Languages — C++, ISO/IEC JTC1/SC22/WG21

Authors: Tian Liao, Mingxin Wang

Reply-to: Tian Liao <tilia@microsoft.com>; Mingxin Wang <mingxwa@microsoft.com>

Audience: LEWG

Date: 2024-09-16

Abstract: *Proxy* is a new library feature that is being proposed to delegate general pointer types with the type erasure technique to support non-intrusive polymorphism programming in C++. The focus of paper is on utility functions – *make_proxy*, *allocate_proxy* and *make_proxy_inplace*, which were splitted from previous versions of the *proxy* proposal. We believe they are useful tools to help create *proxy* instances properly.

1 Introduction

Paper P3086 proposed a *Pointer-Semantics-Based Polymorphism Library*, which is designed to help people build extendable and efficient polymorphic programs with better abstractions and less intrusive code. This paper is proposing the utility part that is separated from early versions of paper P3086 and P0957.

More specifically, we are eager to add function templates *make_proxy*, *allocate_proxy*, *make_proxy_inplace*, and concept *inplace_proxiable_target* together with the *proxy* library into the standard as library features.

make_proxy's syntax is similar to the constuctors of `std::any`. It is designed to provide simple ways to construct *proxy* instances from values. With *make_proxy*, SBO (small buffer optimization) may implicitly apply to avoid the potential overhead that may come from unnecessary heap allocations.

allocate_proxy's syntax is similar to `std::allocate_shared`. It is intended for custom allocators.

inplace_proxiable_target is a concept that restricts its associated *facade* type and value type so that calling its correlated *make_proxy_inplace* is well-formed.

make_proxy_inplace, similar to `std::optional`, provides an SBO pointer that stores given values within its storage. More details about *make_proxy_inplace* can be found in Technical Specifications.

2 Motivation

Class template *proxy* is based on pointer semantics, which means it usually involves heap allocations to instantiate a *proxy* object from value, though sometimes those allocations could be evitable if the value's type is trivial enough.

For example, if the maximum pointer size defined by `F::constraints.max_size` is

$$2 \times \texttt{sizeof(void*)}.$$

When a user wants to have a *proxy* instance for an integer value 2024, the user may do

```
struct year {
  static constexpr proxiable_ptr_constraints constraints{
    .max_size = sizeof(void*[2]),
    .max_align = alignof(void*[2]),
    .copyability = constraint_level::none,
    .relocatability = constraint_level::nothrow,
    .destructibility = constraint_level::nothrow};
  // other members to meet the facade name-requirement.
};
std::proxy<year> CreateYear() {
  return std::make_unique<int>(2024);  // implicitly converts to std::proxy<year>
}
```

Apparently, `std::make_unique<int>(2024)` performs an allocation, which may be considered an expensive cost in certain scenarios. To improve the construction from integer values in this case, we shall introduce SBO here. Because the storage size (e.g. 16 bytes on 64-bit machines) of `std::proxy<year>` is mostly sufficient to place a value of integer type (e.g. 4 bytes). With the SBO capability provided by *make_proxy*, which could be an internal implementation of a *fancy pointer* that guarantees space efficiency, users can choose the implementation below for the `CreateYear()` function:

```
std::proxy<year> CreateYear() {
  return std::make_proxy<year>(2024);  // no heap allocation happens
}
```

In simple words, *make_proxy* shall initially try constructing a *proxy* object with given values stored inplace, and then fall back to storing the given values in an arbitrary memory range with heap allocations if the first trial failed. Both conditions, as well as their resolution, shall happen at compile-time.

3 Considerations

3.1 Construction of *proxy*

According to P3086, there are three different constructors of *proxy* that allow user to create a new *proxy* instance from an object of type `P` or `decay_t<P>`.

```
template <class P>
proxy(P&& ptr) noexcept(/*see P3086*/) requires(/*see P3086*/);

template <class P, class... Args>
explicit proxy(in_place_type_t<P>, Args&&... args)
  noexcept(/*see P3086*/) requires(/*see P3086*/);

template <class P, class U, class... Args>
explicit proxy(in_place_type_t<P>, initializer_list<U> il,
  Args&&... args)
  noexcept(/*see P3086*/) requires(/*see P3086*/);
```

The type name `P` used in template arguments stands for a raw pointer type or a *fancy pointer* type. These three constructors meet all the needs of *proxy* itself, but they are not always convenient to use when the type `P` is managing the ownership of the value that it is pointing to. Construction from a value and using a custom allocator are the very typical use cases related to the problem.

3.2 Construction from a value

Let's continue on the `CreateYear` example above. To eliminate unnecessary allocations, user may have to provide an adhoc *sbo-ptr* that wraps a trivial value and pretend to be a pointer type, and then use it to construct a *proxy* object. For example:

```
template <class T>
class sbo_ptr {
 public:
  template <class... Args>
  sbo_ptr(Args&&... args)
      noexcept(std::is_nothrow_constructible_v<T, Args...>)
      requires(std::is_constructible_v<T, Args...>)
      : value_(std::forward<Args>(args)...) {}
  sbo_ptr(const inplace_ptr&)
      noexcept(std::is_nothrow_copy_constructible_v<T>) = default;
  sbo_ptr(inplace_ptr&&)
      noexcept(std::is_nothrow_move_constructible_v<T>) = default;

  T* operator->() noexcept { return &value_; }
  const T* operator->() const noexcept { return &value_; }
  T& operator*() & noexcept { return value_; }
  const T& operator*() const& noexcept { return value_; }
  T&& operator*() && noexcept { return std::forward<T>(value_); }
  const T&& operator*() const&& noexcept
      { return std::forward<const T>(value_); }

 private:
  T value_;
};

auto CreateYear() {
  return std::proxy<year>{
    sbo_ptr<int>{2024}};
```

```
}
```

With *make_proxy_inplace* or *make_proxy*, the tools we are proposing in this paper, user can achieve the same goal with a simple line, like:

```
auto CreateYear() {
  return std::make_proxy<year, int>(2024);
}
```

Or, in this use case, it is equvialent to:

```
auto CreateYear() {
  return std::make_proxy_inplace<year, int>(2024);
}
```

3.3 Custom allocator

Similar to `std::allocate_shared`, *allocate_proxy* accepts a custom allocator and retains a copy of the allocator for releasing resources in future. To achieve this goal, user may have to provide an *allocated-ptr*, which could be adhoc again like the *sbo-ptr*, to allocates storage for its contained object of type `T` with an allocator of type `Alloc` and manages the lifetime of its contained object.

In addition, the implementation of *allocated-ptr* may vary depending on the definition of its associated *facade* type `F`. Specifically, when `F::constraints.max_size` and `F::constraints.max_align` are not large enough to hold both a pointer to the allocated memory and a copy of the allocator, *allocated-ptr* shall allocate additional storage for the allocator.

allocate_proxy shall provide a built-in implementation for any *facade* types. With textitallocate_proxy, user can easily construct a *proxy* object with a large data and a custom allocator. For example:

```
auto CreateHugeYear(){
  // sizeof(std::array<int, 1000>) is usually greater than the max_size defined in facade,
  // calling allocate_proxy has no limitation to the size and alignment of the target
  using HugeYearData = std::array<int, 1000>;
  return std::allocate_proxy<year, HugeYearData>(
    std::allocator<HugeYearData>{});
}
```

3.4 Freestanding

As per all *proxy* features proposed by P3086 are freestanding, the *inplace_proxiable_target* concept and the *make_proxy_inplace* function templates shall be freestanding as well. *allocate_proxy* and *make_proxy* are removed from freestanding because they involved dynamic allocations.

3.5 Feature test macro

All the features proposed by this paper honors the test macro defined in P3086. That is:

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

4 Technical specifications

4.1 Additional synopsis for header <memory>

```
namespace std {
  // concept inplace_proxiable_target
  template <class T, class F>
  concept inplace_proxiable_target = proxiable</* inplace_ptr<T> */, F>;

  // the allocate_proxy overloads, which are freestanding-deleted
  template <facade F, class T, class Alloc, class... Args>
  proxy<F> allocate_proxy(const Alloc& alloc, Args&&... args);

  template <facade F, class T, class Alloc, class U, class... Args>
  proxy<F> allocate_proxy(const Alloc& alloc, std::initializer_list<U> il, Args&&... args);
```

```
  template <facade F, class Alloc, class T>
  proxy<F> allocate_proxy(const Alloc& alloc, T&& value);

  // the make_proxy_inplace overloads
  template <facade F, inplace_proxiable_target<F> T, class... Args>
  proxy<F> make_proxy_inplace(Args&&... args)
      noexcept(std::is_nothrow_constructible_v<T, Args...>);

  template <facade F, inplace_proxiable_target<F> T, class U, class... Args>
  proxy<F> make_proxy_inplace(std::initializer_list<U> il, Args&&... args)
      noexcept(std::is_nothrow_constructible_v<
          T, std::initializer_list<U>&, Args...>);

  template <facade F, class T>
  proxy<F> make_proxy_inplace(T&& value)
      noexcept(std::is_nothrow_constructible_v<std::decay_t<T>, T>)
      requires(inplace_proxiable_target<std::decay_t<T>, F>);

  // the make_proxy overloads, which are freestanding-deleted
  template <facade F, class T, class... Args>
  proxy<F> make_proxy(Args&&... args);

  template <facade F, class T, class U, class... Args>
  proxy<F> make_proxy(std::initializer_list<U> il, Args&&... args);

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

The above synopsis is assumming the *memory* header has below synopsis defined in paper 3086:

```
namespace std {
  template <class F>
    concept facade = // see p3086r3;
  template <class P, class F>
    concept proxiable = // see p3086r3;
  template <class F>
    class proxy; // see p3086r3
}
```

4.2 Concept `std::inplace_proxiable_target`

The concept `inplace_proxiable_target<T, F>` specifies that a value type `T`, when wrapped by an implementation-defined non-allocating pointer type, models a contained value type of `proxy<F>`. The size and alignment of this implementation-defined pointer type are guaranteed to be equal to those of type `T`.

4.3 Function template `std::allocate_proxy`

The definition of *allocate_proxy* makes use of an exposition-only class template *allocated-ptr*. An object of type `allocated-ptr<T, Alloc>` allocates the storage for another object of type `T` with an allocator of type `Alloc` and manages the lifetime of this contained object. Similar to `std::optional`, `allocated-ptr<T, Alloc>` provides `operator*` for accessing the managed object of type `T` with the same qualifiers, but does not necessarily support the state where the contained object is absent.

allocate_proxy returns a constructed *proxy* object. It may throw any exception thrown by allocation or the constructor of `T`.

1. `template <facade F, class T, class Alloc, class... Args>`

   `proxy<F> allocate_proxy(const Alloc& alloc, Args&&... args);`

   Effects: Creates a `proxy<F>` object containing a value `p` of type `allocated-ptr<T, Alloc>`, where `*p` is direct-non-list-initialized with `std::forward<Args>(args)...`.

2. `template <facade F, class T, class Alloc, class U, class... Args>`

   `proxy<F> allocate_proxy(const Alloc& alloc, std::initializer_list<U> il, Args&&... args);`

   Effects: Creates a `proxy<F>` object containing a value `p` of type `allocated-ptr<T, Alloc>`, where `*p` is direct-non-list-initialized with il, `std::forward<Args>(args)...`.

3. `template <facade F, class Alloc, class T>`
   `proxy<F> allocate_proxy(const Alloc& alloc, T&& value);`

   Effects: Creates a `proxy<F>` object containing a value `p` of type `allocated-ptr<std::decay_t<T>, Alloc>`, where `*p` is direct-non-list-initialized with `std::forward<T>(value)`.

4.4 Function template `std::make_proxy_inplace`

The definition of `make_proxy_inplace` makes use of an exposition-only class template `sbo-ptr`. Similar to `std::optional`, `sbo-ptr<T>` contains the storage for an object of type `T`, manages its lifetime, and provides `operator*` for access with the same qualifiers. However, it does not necessarily support the state where the contained object is absent. `sbo-ptr<T>` has the same size and alignment as `T`.

1. `template <facade F, inplace_proxiable_target<F> T, class... Args>`
   `proxy<F> make_proxy_inplace(Args&&... args)`
   `noexcept(std::is_nothrow_constructible_v<T, Args...>);`

   Effects: Creates a `proxy<F>` object containing a value `p` of type `sbo-ptr<T>`, where `*p` is direct-non-list-initialized with `std::forward<Args>(args)...`.

2. `template <facade F, inplace_proxiable_target<F> T, class U, class... Args>`
   `proxy<F> make_proxy_inplace(std::initializer_list<U> il, Args&&... args)`
   `noexcept(std::is_nothrow_constructible_v<`
   `T, std::initializer_list<U>&, Args...>);`

   Effects: Creates a `proxy<F>` object containing a value `p` of type `sbo-ptr<T>`, where `*p` is direct-non-list-initialized with il, `std::forward<Args>(args)...`.

3. `template <facade F, class T>`
   `proxy<F> make_proxy_inplace(T&& value)`
   `noexcept(std::is_nothrow_constructible_v<std::decay_t<T>, T>)`
   `requires(inplace_proxiable_target<std::decay_t<T>, F>);`

   Effects: Creates a `proxy<F>` object containing a value `p` of type `sbo-ptr<std::decay_t<T>>`, where `*p` is direct-non-list-initialized with `std::forward<T>(value)`.

4.5 Function template `std::make_proxy`

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

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

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

   - `sizeof(T)` $\leq$ `F::constraints.max_size`, and
   - `alignof(T)` $\leq$ `F::constraints.max_align`, and
   - `T` meets the copyiability requirements defined by `F::constraints.copyability`, and
   - `T` meets the relocatability requirements defined by `F::constraints.relocatability`, and
   - `T` meets the destructibility requirements defined by `F::constraints.destructibility`, and
   - for any reflection type `R` defined by `F::reflection_types`,

     `R` shall be constructible from `std::in_place_type_t<sbo-ptr<T>>`.

2. `template <facade F, class T, class U, class... Args>`
   `proxy<F> make_proxy(std::initializer_list<U> il, Args&&... args);`
   Effects: Equivalent to

   ```
   return make_proxy<F, T>(il, std::forward<Args>(args)...);
   ```

3. `template <facade F, class T>`
   `proxy<F> make_proxy(T&& value);`
   Effects: Equivalent to

   ```
   return make_proxy<F, decay_t<T>>(std::forward<T>(value));
   ```

5 Acknowledgements

Thanks to Mingxin for his advanced work on P0957 and P3086, and for giving me a chance to join him together to complete the work of proxy.

6 References

References

[1] [P3086] Proxy: A Pointer-Semantics-Based Polymorphism Library

Mingxin Wang

[2] [P0957] Proxy: A Polymorphic Programming Library

Mingxin Wang

[3] Open-source: Microsoft Proxy at GitHub

URL: https://github.com/microsoft/proxy