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>

Audience: LEWG

Abstract: *Proxy* is a new library feature that is being proposed to delegate general pointer types with the type-erased technique to support non-intrusive polymorphism programming in C++. The focus of paper is on utility functions – *make_proxy* and *allocate_proxy*, which used to be included in the *proxy* proposal as a sub-feature. We believe they are useful tools to help allocate *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 *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 void the potential overhead that may come from extra heap allocations.

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

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, they may do

```
struct Any : some_facade_builder::build {};
std::proxy<Any> CreateYear() {
  return std::make_unique<int>(2024);  // implicitly converts to std::proxy<Any>
}
```

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<Any>` is mostly sufficient to place a value of integer type (e.g. 4 bytes). With the SBO capability provided by *make_proxy*, users can choose the implementation below for the `CreateYear()` function:

```
std::proxy<Any> CreateYear() {
  return std::make_proxy<Any>(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 Technical specification

3.1 Additional synopsis for header <memory>

```
namespace std {
  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);

  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 F>
    class proxy; // see p3086r3
}
```

3.2 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 with `std::optional`, `allocated-ptr<T, Alloc>` provides `operator*` for accessing the contained object with the same qualifiers, but does not necessarily support the state where the contained object is absent.

allocate_proxy returns a construct *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 an `allocated-ptr<T, Alloc>` direct-non-list-initialized with `std::forward<Args>(args)...`, where the contained value of type `T` 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 an `allocated-ptr<T, Alloc>` 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 an `allocated-ptr<std::decay_t<T>, Alloc>` direct-non-list-initialized with `std::forward<T>(value)`.

3.3 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));
   ```

4 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.

5 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