The Concept of Extending Argument and a Support Library

Document number:	P1648R1
Date:	2019-07-18
Project:	Programming Language C++
Audience:	LEWGI, LEWG, LWG

Authors:	Mingxin Wang (Microsoft (China) Co., Ltd.), Agustín Bergé
Reply-to:	Mingxin Wang <mingxwa@microsoft.com>, Agustín Bergé <agustinberge@gmail.com>

Table of Contents

1 History

1.1 Change from P1648R0

- Remove support for **`void`** and **`reference_wrapper`**;
- Replace function templates **`extending_arg`** and **`make_extended_view`** with **`make_extended`**;
- Remove class template alias **`extended`**.

2 Introduction

When designing template libraries, I found it difficult to extend the lifetime of an argument without copy/move construction or implicit type conversion, especially when a function template accepts multiple arguments with different semantics. The proposed library is a solution for template library API design, enabling them to have elegant APIs while making it easy to extend the lifetime of arguments with potentially lower overhead even if they are not convertible from any other type or not move constructible themselves.

I think this library has the potential for simplifying the API of several facilities in the standard. Meanwhile, it was already used in the API of PFA [P0957R2] and the concurrent invocation library [P0642R2].

3 Motivation and Scope

Let us take **`std::tuple`** as an example. When constructing a value of **`std::tuple`** with its constructors, a copy/move operation or probably a type conversion (with constructors only) is inevitable for each input argument.

Consider the following scenario:

```
struct X {
  X(int, double);
  X(X&&);
  X& operator=(X&&);
};
std::tuple<X, int>{ X{1, 0.37}, 123 };
```

In the sample code above, the move constructor of **`X`** is always invoked, and here is currently no standard way to construct **`X`** in-place and therefore the move construction is inevitable.

Actually, there are many other facilities in the standard other than **`std::tuple`** that have the same issue, e.g., the constructor of **`std::function`**/**`std::thread`**/**`std::packaged_task`** and the function template **`std::async`**. It will be even worse if the input type is not move constructible at all. For example, when the input value is a concurrent data structure that holds mutexes or atomic variables, these mentioned facilities will not work unless extra runtime overhead is introduced (e.g., the data is managed with extra pointers to additional allocated memory).

The issue was mitigated when it comes to **`std::any`**/**`std::variant`**/**`std::optional`**, as they all have two variable parameter constructors with the type of the first parameter being **`std::in_place_type_t`**. However, this approach is not applicable to **`std::tuple`** and some other facilities as they need to accept multiple arguments with different semantics.

Among all the mentioned facilities, **`std::pair`** is the only one that support in-place construction without variable parameter constructor.

```
template <class T1, class T2>
template <class... Args1, class... Args2>
constexpr pair<T1, T2>::pair(piecewise_construct_t,
    tuple<Args1...> first_args, tuple<Args2...> second_args);
```

Inspired by the constructor, I think it could be a good idea to use **`std::tuple`** as an intermedium to hold the arguments for in-place construction, and this is a part of the design direction of the proposed library.

However, since there should be more considerations in compatibility, this proposal only aims to provide a reusable solution rather than updating existing APIs in the standard.

4 Design Decisions

4.1 Extending Argument

When designing a template library, it is usually easy to tell if the lifetime of an argument shall (potentially) be extended. For example, when the input value shall be processed in another thread of execution, the lifetime shall always be extended; when the input value is only used within the scope of a function template, it is usually not necessary to extend the lifetime. Therefore, it is the responsibility for library designers to determine whether/where/when shall the lifetime of an argument be extended. To simplify illustration, an argument whose lifetime shall (potentially) be extended is described as "extending argument" in the rest of the paper.

4.1.1 Representing In-place Construction

One of the most important things is to find a way to represent in-place construction without variable parameters other than copy/move construction or type conversion. As inspired from the piecewise constructor of **`std::pair`**, I think **`std::tuple`** is a good choice to carry variable constructor arguments as one value. However, there should be extra "type" information to carry by the in-place construction expression. To make the semantics clear, I think it could be a good idea to design a facility named **`extending_construction`**, not only does it carry the type information, but also stores the argument for in-place construction. Meanwhile, to increase usability, another helper function template **`make_extending_construction`** is proposed.

4.1.2 Expressions

I think there should be three categories of expression for extending argument (suppose its type is **`T&&`**):

1. if **`std::decay_t<T>`** is **`std::in_place_type_t`** of **`U`**, it shall be regarded as an in-place default construction;
2. if **`std::decay_t<T>`** is **`extending_construction`** mentioned earlier, it shall be regarded as an in-place construction;
3. otherwise, a "decay copy" is expected.

4.1.3 Storing Extended Arguments

Since we are able to construct a non-copyable and non-moveable value with a creation function, thanks to copy elision, it is not necessary to introduce any extra container type to store the extended values. Therefore, the container for "extended value" is removed from this revision.

4.2 Typical Usage

To help figuring out the result for construction, a type template **`extending_t`** and a function template **`make_extended`** are proposed. For example, when it is required to construct a value with a function template library, the library could be designed like:

```
template <class T, class U>
struct foo {
  template <class _T, class _U>
  foo(_T&& v1, _U&& v2)
      : v1_(make_extended(std::forward<_T>(v1))),
        v2_(make_extended(std::forward<_U>(v2))) {}

  T v1_;
  U v2_;
};

template <class E_T, class E_U>
auto make_foo(E_T&& v1, E_U&& v2) {
  return foo<extending_t<E_T>, extending_t<E_U>>{
      std::forward<E_T>(v1), std::forward<E_U>(v2)};
}
```

With the sample library above, users are able to pass any extending arguments to the library if the concrete constructed type is supported by the library:

```
// construct foo<int, double> with decay copy
make_foo(1, 3.4);

int bar = 2;
// construct foo<int, int&> with decay copy and std::reference_wrapper
```

```
make_foo(bar, std::ref(bar));

// construct foo<any, vector<int>> with in_place_type_t and extending_construction
make_foo(
    std::in_place_type<std::any>,
    make_extending_construction<std::vector<int>>({1, 2, 3}));
```

5 Technical Specification

5.1 Header <utility> synopsis

The following content is intended to be merged into [utility.syn].

```
namespace std {

template <class T, class... E_Args>
class extending_construction;

template <class T, class... E_Args>
auto make_extending_construction(E_Args&&... args);

template <class T, class U, class... Args>
auto make_extending_construction(std::initializer_list<U> il, Args&&... args);

template <class T>
using extending_t = see below;

template <class T>
auto make_extended(T&& value);

}
```

5.2 Class template `extending_construction`

```
template <class T, class... E_Args>
class extending_construction {
 public:
  template <class... _E_Args>
  constexpr explicit extending_construction(_E_Args&&... args);

  constexpr extending_construction(extending_construction&&) = default;
  constexpr extending_construction(const extending_construction&) = default;
  constexpr extending_construction& operator=(extending_construction&&) = default;
  constexpr extending_construction& operator=(const extending_construction&)
     = default;

  constexpr std::tuple<E_Args...> get_args() const&;
  constexpr std::tuple<E_Args...>&& get_args() && noexcept;
};
```

`template <class... _E_Args>`
`constexpr explicit extending_construction(_E_Args&&... args);`

Effects: Initializes the arguments with the corresponding value in **`std::forward<_E_Args>(args)`**.

`constexpr std::tuple<E_Args...> get_args() const&;`

Returns: A copy of the stored arguments tuple.

`constexpr std::tuple<E_Args...>&& get_args() && noexcept;`

Returns: An rvalue reference of the stored arguments tuple.

5.3 `extending_construction` creation functions

`template <class T, class... E_Args>`
`auto make_extending_construction(E_Args&&... args);`

Returns: A value of **`extending_construction<T, std::decay_t<E_Args>...>`** constructed with **`std::forward<E_Args>(args)...`**.

`template <class T, class U, class... Args>`
`auto make_extending_construction(std::initializer_list<U> il, Args&&... args);`

Returns: A value of **`extending_construction<T, std::initializer_list<U>, std::decay_t<E_Args>...>`** constructed with **`il, std::forward<E_Args>(args)...`**.

5.4 Extending argument resolution utilities

`template <class T>`
`using extending_t =` *`see below;`*

Definition:

- **`U`** if **`std::decay_t<T>`** is an instantiation of **`std::in_place_type_t<U>`** of some type **`U`**, or
- **`U`** if **`std::decay_t<T>`** is an instantiation of **`extending_construction<U, E_Args...>`** of some type **`U`** and **`E_Args...`**, or
- otherwise, **`std::decay_t<T>`**.

`template <class T>`
`see below` **`make_extended(T&& value);`**

Returns:

- `U{}` if `std::decay_t<T>` is an instantiation of `std::in_place_type_t<U>` of some type `U`, or
- `make_from_extending_tuple<U>(forward<T>(value).get_args(), make_index_sequence<sizeof...(E_Args)>{})` if `std::decay_t<T>` is an instantiation of `extending_construction<U, E_Args...>` of some type `U` and `E_Args...`, where `make_from_extending_tuple` is defined as follows:

  ```
   template <class T, class E_ArgsTuple, size_t... I>
   T make_from_extending_tuple(E_ArgsTuple&& args, index_sequence<I...>)
      { return T{make_extended(get<I>(move(args)))...}; }
  ```
- otherwise, `std::forward<T>(value)`.

6 Summary

I think this library could be useful in template library design with nice maintainability and potentially higher performance than "decay copy", since it provides a standard way to pass arguments to a function template in 4 manners, and users could determine which to use depending on concrete requirements.

I have also used it in another two proposals of mine to simplify the API design, the "PFA" [P0957R2] and the "Concurrent Invocation" library [P0642R2].