The Concept of Extending Argument and a Support Library

Document number:	P1648R0
Date:	2019-06-16
Project:	Programming Language C++
Audience:	LEWG, LWG

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

Table of Contents

1 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].

2 Motivation and Scope

Let us take **`std::tuple`** as an example. When constructing a value of **`std::tuple`** with its constructors or the function template **`std::make_tuple`**, 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, no matter we use **`std::make_tuple`** or directly use the constructor of **`std::tuple`**. There 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`**/**`std::make_pair`**. 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.

3 Design Decisions

3.1 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.

The type or semantics of an extending value depends on concrete template library design. Typically, a value of any type has a chance to be extended, including references or **void**. As far as I am concerned, extending **void** is useful in concurrent programming when a shared context is optional.

3.1.1 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. 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.

3.1.2 Expressions

Referring to the standard, when an extending argument is intended to be passed by reference, **std::reference_wrapper** is generally acceptable, e.g., in the constructor of **std::thread** or function template **std::make_pair**, **std::make_tuple**. However, as an extending argument could potentially be any type including **void**, I suggest to use **in_place_type_t** to represent **void** and other types that are intended to be default constructed.

In this way, there should be four categories of expression for extending argument (suppose its type is **`T&&`**):

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

3.1.3 Storing Extended Arguments

There should be an extra type to store extending arguments, since "construction from tuple" and "void type" shall be supported. Therefore, the type template **`extended`** is proposed with "extended values" constructible from **`std::tuple`** and "decay copy", and "extended references" constructible from corresponding references. Because "extended value", "extended reference" and even "extended void" have different construction strategy, the type template **`extended`** shall be an alias. The implementation shall be selected referring to the category of the type of the value to extend, and may use the "Applicable Template" library [P1649R0].

3.2 Typical Usage

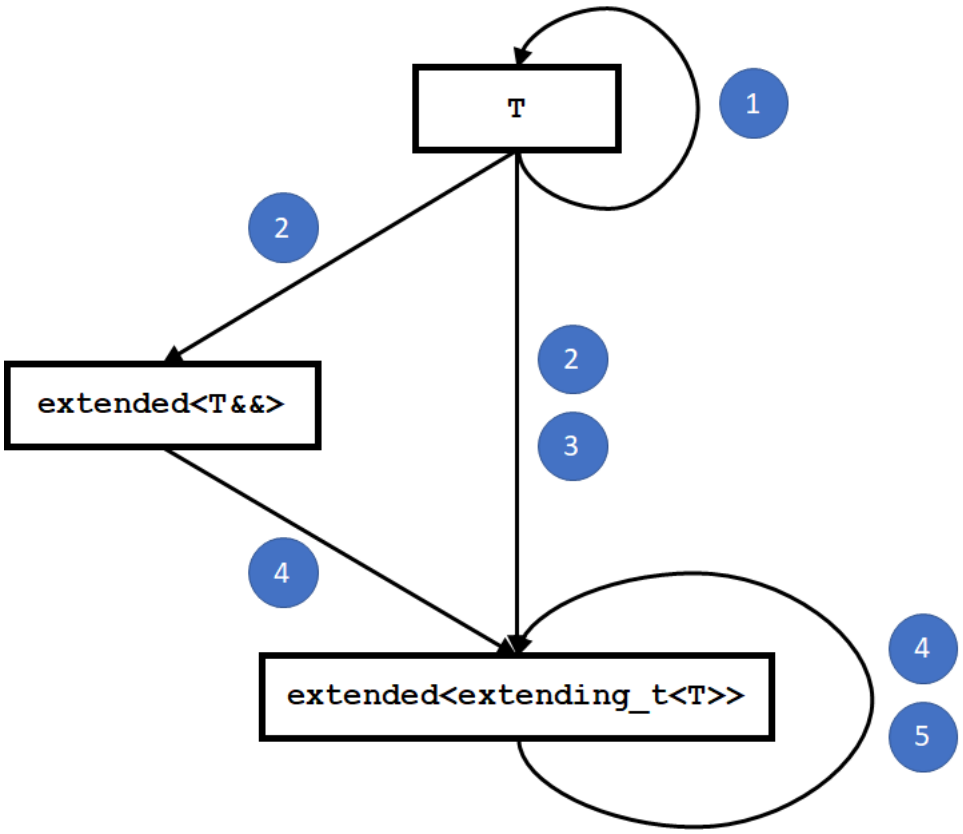

Figure 1

Figure 1 indicates the lifetime evolution paths for an extending argument `**arg**` of type `**T**`. Each of circled numbers indicates an evolution operation in specific contexts:

1. Decay copy of an extending argument is recommended when the extending argument is not used in the current context and will potentially be used in another context.
2. When an extending argument is initially used in a function scope, it is usually efficient to use it with the proposed function template `**make_extended_view(forward<T>(arg))**`.
3. When an extending argument is initially used as a member variable of a type, we may design the type of the variable as `**extended<extending_t<T>>**` and construct it with `**extending_arg(forward<T>(arg))**`.
4. When it is required to transfer the extended variable from function scope to type scope, we could construct a value of type `**extended<extending_t<T>>**` with the return value of prior call to `**make_extended_view**`.
5. When transferring the extended variable among type scope, copy/move constructors/assignments of instantiated `**extended**` type could also be used.

For example, if an extending argument is only for repeatedly usage and does not cross context, the value does not need to be copy/move constructible. In this case, library designers will need to deduce the concrete type for construction and corresponding argument. For example, if the type of the extending value is **const extending_construction<X>**, the concrete type for construction shall be **X**; if the type of the extending value is **reference_wrapper<Y>**, the concrete type for construction shall be **Y&**.

To help figuring out the type and argument for construction, a type template **extending_t** and a function template **extended_arg** 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_(std::forward<_T>(v1)), v2_(std::forward<_U>(v2)) {}

  extended<T> v1_;
  extended<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>>{
      extended_arg(std::forward<E_T>(v1)),
      extended_arg(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}));
```

4 Technical Specification

4.1 Header <utility> synopsis

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

```
namespace std {

template <class T>
using extended = see below;

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>
see below extended_arg(T&& value);

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

}
```

4.2 Class template alias `extended`

```
template <class T>
using extended = see below;
```

4.2.1 For Value Types

If **T** is not a reference type or **void**, **extended<T>** shall be a type that meets the **CopyConstructible**, **MoveConstructible**, **CopyAssignable** or **MoveAssignable** requirements if **T** meets any of the requirements, respectively. The following expressions shall be well-formed and have the specified semantics. In the expressions,

- **E** denotes the type referred by **extended<T>**;
- **le** denotes an expression of type **E&**;
- **cle** denotes an expression of type **const E&**;
- **re** denotes an expression of type **E&&**;
- **cre** denotes an expression of type **const E&&**;
- **E_Args** denotes a parameter pack;
- **t** denotes an expression of type **std::tuple<E_Args...>**;
- **U** denotes a type;
- **u** denotes an expression of type **U**;
- `**v**` denotes an expression of type `**extended<T&&>&&**`.

E(t)

Requires: **std::is_constructible_v<T, extending_t<E_Args>...>** is **true**.

Effects: Initializes the extended value of type **T** with arguments of type **extending_t<E_Args>&&...** obtained with forwarding the elements of **t**, and for each element **arg** in **t** performing **make_extended_view(arg).get()**.

E(u)

Requires: **std::is_constructible_v<T, U>** is **true**.

Effects: Initializes the extended value of type **T** with **u**.

E(v)

Effects: Initializes the extended value of type **T** with `**v.get()**`.

le.get()

Return type: **T&**.

Returns: An lvalue reference to the extended value.

cle.get()

Return type: **const T&**.

Returns: A const lvalue reference to the extended value.

re.get()

Return type: **`T&&`**.

Returns: An rvalue reference to the extended value.

`cre.get()`

Return type: **`const T&&`**.

Returns: A const rvalue reference to the extended value.

4.2.2 For Reference Types

If **`T`** is a reference type, regardless it is an lvalue reference or an rvalue reference, **`extended<T>`** shall be a type that meet the **`CopyConstructible`** and **`CopyAssignable`** requirements. The following expressions shall be well-formed and have the specified semantics. In the expressions,

- **`E`** denotes the type referred by **`extended<T>`**;
- **`cle`** denotes an expression of type **`const E&`**;
- **`t`** denotes an expression of type **`T`**;

`E(t)`

Effects: Construct the extended reference with **`t`**.

`cle.get()`

Return type: **`T`**.

Returns: The extended reference.

4.2.3 For Void Types

If **`T`** is a **`void`** type, regardless if it has any cv-qualifiers, **`extended<T>`** shall be a type that meet the **`CopyConstructible`** and **`CopyAssignable`** requirements. The following expressions shall be well-formed and have the specified semantics. In the expressions,

- **`E`** denotes the type referred by **`extended<T>`**;
- **`cle`** denotes an expression of type **`const E&`**;
- **`t`** denotes an expression of type **`std::tuple<>`**;

`E(t)`

Effects: Construct the extended void.

`cle.get()`

Return type: **`void`**.

4.3 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.

4.4 `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)...`.

4.5 Extending argument resolution utilities

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

Definition:

- `U&` if `std::decay_t<T>` is an instantiation of `std::reference_wrapper<U>` of some type `U`, or
- `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 extended_arg(T&& value);
```

Returns:

- **value.get()** if **std::decay_t<T>** is an instantiation of **std::reference_wrapper<U>** of some type **U**, or
- **std::tuple<>{}** if **std::decay_t<T>** is an instantiation of **std::in_place_type_t<U>** of some type **U**, or
- **std::forward<T>(value).get_args()** if **std::decay_t<T>** is an instantiation of **extending_construction<U, E_Args...>** of some type **U** and **E_Args...**, or
- otherwise, **std::forward<T>(value)**.

4.6 extended creation function

```
template <class T>
auto make_extended_view(T&& value);
```

Returns: A value of **extended<T&&>** constructed with **std::move(value)** if **std::is_same_v<extending_t<T>, T>** is **true** and **std::is_reference_v<T>** is **false**, or a value of **extended<extending_t<T>>** constructed with **extended_arg(std::forward<T>(value))** otherwise.

5 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]. Please find the tested implementation of this library with this link, which compiles with latest GCC, LLVM and MSVC.