A Library for Sink Argument Passing

Document number:	P1648R2
Date:	2019-10-06
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 P1648R1

- Remove the concept of "Extending Argument" as it is the same as a more well-known term "Sink Argument";
- Change the title from "The Concept of Extending Argument and a Support Library" into "A Library for Sink Argument Passing";
- Change the names of **`extending_construction`**, **`make_extending_construction`**, **`extending_t`** and **`make_extended`** into **`sinking_construction`**, **`make_sinking_construction`**, **`sunk_t`** and **`sink`**, respectively;
- Change the semantics of the function template **`sink`** not to be recursive;
- Add before/after tables in the motivation part;
- Add a series of alternative options for names.

1.2 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 sink template libraries, I found it difficult to extend the lifetime of a sink 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 sink arguments with potentially lower overhead even if they are not convertible from any other type or not move constructible themselves.

I think the proposed library has the potential for simplifying the API of several facilities in the standard. The full implementation for the library is available here. Meanwhile, it was already used in the API of PFA [P0957R3] and the concurrent invocation library [P0642R3].

3 Motivation and Scope

Let's take **`std::vector<T>::push_back`** as an example. Before C++11, there was only one version of **`push_back`** whose input type is the const reference of **`T`**:

```
void push_back(const T& x);
```

Because it always copies the input value, which introduce unnecessary overhead when the value is a temporary, two more function templates in **`std::vector<T>`** are introduced since C++11:

```
void push_back(T&& x);
template<class... Args> reference emplace_back(Args&&... args);
```

Their existence mitigated the potential performance and usability limitations in **`push_back`** before C++11, because rvalue references or in-place construction is supported, and copy-construction is no longer forced.

However, we found it awkward when designing other sink template libraries with such principle, because:

1. There always need to be multiple versions for every single sink function, which makes the API complicated to design, maintain and learn, and
2. In-place construction uses variable parameter function template, which could only serve one sink argument; but when there are multiple sink arguments, variable parameter will stop working.

The proposed library is designed to simplify the design of sink template libraries. The following table shows how it could help designing sink template libraries:

	Before	After
Number of overloads required	At least 2	1
Number of arguments supported for in-place construction	Only 1	Unlimited
Supporting initializer list in in-place construction	Manually adding overloads	Yes

Sample design for **`std::vector<T>::push_back`**	`void push_back(const T& x);`
	
	`void push_back(T&& x);`
	
	`template<class... Args>`
	`reference emplace_back(`
	`  Args&&... args);`
		`template <class U>`
		`void push_back(U&& x);`

Sample design for the ctor of **`std::any`** from semantic value	`template<class T>`
	`any::any(T&& value);`
	
	`template<class T,`
	`  class... Args>`
	`any::any(in_place_type_t<T>,`
	`  Args&&... args);`
	
	`template<class T, class U,`
	`  class... Args>`
	`any::any(in_place_type_t<T>,`
	`  initializer_list<U> il,`
	`  Args&&... args);`
		`template<class T>`
		`any::any(T&& value);`

4 Impact on the Standard

There are many sink functions in the standard, including **`std::vector<T>::push_back`** and **`std::any::any`** mentioned in the motivation part, that could benefit from the proposed library. These functions includes every "insert", "push" and "emplace" operation in the standard container library; constructors and modifiers of every "value holder" facility including the standard containers, **`std::pair`**, **`std::tuple`**, **`std::optional`**, **`std::any`**, **`std::variant`**, **`std::function`**; concurrency facilities including **`std::thread`**, **`std::packaged_task`**, **`std::promise`**, **`std::async`**;

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.

5 Design Decisions

5.1 Sink Argument

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

5.1.1 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 **`sinking_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_sinking_construction`** is proposed.

5.1.2 Expressions

I think there should be three categories of expression for sink arguments (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 **`sinking_construction`** mentioned earlier, it shall be regarded as an in-place construction;
3. otherwise, a "decay copy" is expected.

5.1.3 Storing Sink 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 sink values. Therefore, the container for "sink value" is removed from revision 1.

5.2 Typical Usage

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

  T v1_; U v2_;
};
template <class T, class U>
foo(T&&, U&&) -> foo<sunk_t<T>, sunk_t<U>>;
```

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

```
// construct foo<int, double> with decay copy
foo{1, 3.4};

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

6 Technical Specification

6.1 Header <utility> synopsis

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

```
namespace std {
```

```
template <class T, class... Args>
class sinking_construction;

template <class T, class... Args>
auto make_sinking_construction(Args&&... args);

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

template <class T>
using sunk_t = see below;

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

}
```

6.2 Class template `sinking_construction`

```
template <class T, class... Args>
class sinking_construction {
 public:
  template <class... _Args>
  constexpr explicit sinking_construction(_Args&&... args);

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

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

template <class... _Args>
constexpr explicit sinking_construction(_Args&&... args);
```

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

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

Returns: A copy of the stored arguments tuple.

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

Returns: An rvalue reference of the stored arguments tuple.

6.3 `sinking_construction` creation functions

```
template <class T, class... Args>
auto make_sinking_construction(Args&&... args);
```

Returns: A value of `sinking_construction<T, std::decay_t<Args>...>` constructed with `std::forward<Args>(args)...`.

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

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

6.4 Sink argument resolution utilities

```
template <class T>
using sunk_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 `sinking_construction<U, Args...>` of some type `U` and `Args...`, or
- otherwise, `std::decay_t<T>`.

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

Returns:

- `U{}` if `std::decay_t<T>` is an instantiation of `std::in_place_type_t<U>` of some type `U`, or
- `std::make_from_tuple<U>(forward<T>(value).get_args())` if `std::decay_t<T>` is an instantiation of `sinking_construction<U, Args...>` of some type `U` and `Args...`.
- otherwise, `std::forward<T>(value)`.