Abstract

This paper presents a proposal for a generic factory $\text{make}<\text{TC}>(\text{args}...)$ that allows to make generic algorithms that need to create an instance of a wrapped class $\text{TC}$ from the underlying types.

$\text{P0091R4}$ extends template parameter deduction for functions to constructors of template classes. With this feature, it would seem clear that this proposal lost most of its added value but this is not completely the case.

## Table of Contents

- History
- Introduction
- Motivation and scope
- Proposal
- Design rationale
- Proposed wording
- Implementability
- Open points
- Acknowledgements
- References

### History

**R2**

- Added factory for $\text{std::array}$.
- Nest everything into $\text{types_constructible}$ namespace and introduce $\text{types_constructible::make}$ in namespace $\text{std}$.
- Remove from the wording the integer template parameters and replace them by a remark as the type $\text{T}$ must not be deduced.
Introduction

This paper presents a proposal for a family of generic factories `make<TC>(args...)` that create an instance of a wrapping class from a *type constructor* and his underlying types as well as emplace factories `make<T>(args...)` that creates an instance of a wrapping class by emplace constructing the underlying type from the provided arguments.

P0091R4 extends template parameter deduction for functions to constructors of template classes. With this feature, it would seem that this proposal has lost most of its added value but this is not the case.

**Motivation and scope**

**Possible valued types**

All these types, `shared_ptr<T>`, `unique_ptr<T,D>`, `optional<T>`, `expected<T,E>` (see P0323R2) and `future<T>` (see P0319R1), have in common that all of them have an underlying type `T`.

There are two kind of factories:

- type constructor with the underlying types as parameter
  - `back_inserter`
  - `makeOptional`
  - `make_ready_future`
  - `make_expected`

- emplace construction of the underlying type given the constructor parameters
  - `make_shared`
  - `make_unique`
  - `make_ready_future` P0319R1

When writing an application, the user knows if the function to write should return a specific type, as `shared_ptr<T>`, `unique_ptr<T,D>`, `optional<T>`, `expected<T,E>` or `future<T>`. E.g. when the user knows that the function must return an owned smart pointer it would use `unique_ptr<T>`.
```cpp
template <class T> 
unique_ptr<T> f() {
    T a,
    ...
    return make_unique<T>(a);
}
```

Note that

```cpp
return unique_ptr(a); // with [P0091R4]
```

will not compile as `unique_ptr<T>` constructor needs a `T*`.

If the user knows that the function must return a shared pointer

```cpp
template <class T> 
shared_ptr<T> f() {
    T a,
    ...
    return make_shared<T>(a);
}
```

Note that

```cpp
return shared_ptr(a); // with [P0091R4]
```

will not compile as `shared_ptr<T>` constructor needs a `T*`.

However, when writing a library, the author doesn't always know which type the user wants as a result. In these cases, the function library must take some kind of type constructor to let the user make the choice, such as a template.

```cpp
template <template <class> class TC, class T>
TC<T> f() {
    T a,
    ...
    return make<TC>(a);
}
```

Another generic example: Suppose that `N` is a `Nullable` type if

```cpp
nullable::none<type_constructor_t<N>>() , nullable::has_value(pv) and nullable::value(pv) are well formed. If in addition, we have that
make<type_constructor_t<N>>(c(nullable::value(pv))) is well formed, we can define for these classes the transform function as follows.
```
The Nullable types proposed in P0196R2 satisfy almost these requirement. What is yet missing is the `nullable::value(pv)` requirement.

**Product types**

In addition, we have factories for the product types such as `pair` and `tuple`.

- `make_pair`
- `make_tuple`
- `make_array`

**Comparison with make_ factories**
<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>int v=0;</td>
<td>int v=0;</td>
</tr>
<tr>
<td>auto x1 = make_shared&lt;int&gt;(v);</td>
<td>auto x1 = make&lt;shared_ptr&gt;(v);</td>
</tr>
<tr>
<td>auto x2 = make_unique&lt;int&gt;(v);</td>
<td>auto x2 = make&lt;unique_ptr&gt;(v);</td>
</tr>
<tr>
<td>auto x3 = make_optional(v);</td>
<td>auto x3 = make&lt;optional&gt;(v);</td>
</tr>
<tr>
<td>auto x4v = make_ready_future();</td>
<td>auto x4v = make&lt;future&gt;();</td>
</tr>
<tr>
<td>auto x4 = make_ready_future(v);</td>
<td>auto x4 = make&lt;future&gt;(v);</td>
</tr>
<tr>
<td>auto x5v = make_ready_future().share();</td>
<td>auto x5v = make&lt;shared_future&gt;();</td>
</tr>
<tr>
<td>auto x5 = make_ready_future(v).share();</td>
<td>auto x5 = make&lt;shared_future&gt;(v);</td>
</tr>
<tr>
<td>auto x6v = make_expected();</td>
<td>auto x6v = make&lt;expected&gt;();</td>
</tr>
<tr>
<td>auto x6 = make_expected(v);</td>
<td>auto x6 = make&lt;expected&gt;(v);</td>
</tr>
<tr>
<td>auto x7 = make_pair(v, v);</td>
<td>auto x7 = make&lt;pair&gt;(v, v);</td>
</tr>
<tr>
<td>auto x8 = make_tuple(v, v, 1u);</td>
<td>auto x8 = make&lt;tuple&gt;(v, v, 1u);</td>
</tr>
<tr>
<td>auto x9 = make_array(v, v, 1u);</td>
<td>auto x9 = make&lt;array_tc&gt;(v, v, 1u);</td>
</tr>
<tr>
<td>future&lt;int&amp;&gt; x4r = make_ready_future(ref(v));</td>
<td>future&lt;int&amp;&gt; x4r = make&lt;future&gt;(ref(v));</td>
</tr>
<tr>
<td>auto x1 = make_shared&lt;A&gt;(v, v);</td>
<td>auto x1 = make&lt;shared_ptr&lt;A&gt;&gt;(v, v);</td>
</tr>
<tr>
<td>auto x2 = make_unique&lt;A&gt;(v, v);</td>
<td>auto x2 = make&lt;unique_ptr&lt;A&gt;&gt;(v, v);</td>
</tr>
<tr>
<td>auto x3 = make_optional&lt;A&gt;(v, v);</td>
<td>auto x3 = make&lt;optional&lt;A&gt;&gt;(v, v);</td>
</tr>
<tr>
<td>auto x4 = make_future&lt;A&gt;(v, v);</td>
<td>auto x4 = make&lt;future&lt;A&gt;&gt;(v, v);</td>
</tr>
<tr>
<td>auto x5 = make_shared_future&lt;A&gt;(v, v);</td>
<td>auto x5 = make&lt;shared_future&lt;A&gt;&gt;(v, v);</td>
</tr>
<tr>
<td>auto x6 = make_expected&lt;A&gt;(v, v);</td>
<td>auto x6 = make&lt;expected&lt;A&gt;&gt;(v, v);</td>
</tr>
</tbody>
</table>

We can use the class template name as a type constructor

```cpp
vector<int> vi1 = { 0, 1, 2, 3, 5, 8 };
vector<int> vi2;
copy_n(vi1, 3, maker<back_insert_iterator>(vi2));

int v=0;
auto x1 = make_shared_ptr(v);
auto x2 = make_unique_ptr(v);
auto x3 = make_optional(v);
auto x4v = make<future>();
auto x4 = make<future>(v);
auto x5v = make<shared_future>();
auto x5 = make<shared_future>(v);
auto x6v = make<expected>();
auto x6 = make<expected>(v);
auto x7 = make<pair>(v, v);
auto x8 = make<tuple>(v, v, 1u);
```

or making use of `reference_wrapper` type deduction
```cpp
int v=0;
future<int&> x4 = make_future(std::ref(v));
```

or use the class name to support in place construction

```cpp
auto x1 = make_shared_ptr<A>(v, v);
auto x2 = make_unique_ptr<A>(v, v);
auto x3 = make_optional<A>(v,v);
auto x4 = make_future<A>(v,v);
auto x5 = make_shared_future<A>(v, v);
auto x6 = make_expected<A>(v, v);
```

Note, with P0091R4, the following is already possible

```cpp
int v=0;
auto x3 = optional(v);
auto x7 = pair(v, v);
auto x8 = tuple(v, v, 1u);
```

We can also make use of the class name to avoid the type deduction

```cpp
int i;
auto x1 = make_future<long>(i);
```

Sometimes the user wants that the underlying type be deduced from the parameter, but the type constructor needs more information. A type holder `_t` can be used to mean any type `T`.

```cpp
auto x2 = make_expected<_t, E>(v);
auto x2 = make_unique_ptr<_t, MyDeleter>(v);
```

Comparison with P0091
Proposal

Type constructor factory

```cpp
template <class TC>
meta::invoke<TC, int> safe_divide(int i, int j)
{
    if (j == 0)
        return {};
    else
        return make<TC>(i / j);
}
```

We can use this function with different type constructors as

```cpp
auto x = safe_divide<optional<_t>)(1, 0);
```

How to define a class that wouldn't need customization?
For the `make` default constructor function, the class needs at least to have a default constructor.

`C();`

For the `make` copy/move constructor function, the class needs at least to have a constructor from the underlying types.

`C(Xs&&...);`

### How to customize an existing class

When the existing class doesn't provide the needed constructor as e.g. `future<T>`, the user needs specialize the `std::experimental::type_constructor::traits<T>` class providing the needed overloads for `make`.

```cpp
namespace std::experimental::type_constructible {

    template <class T>
    struct traits<future<T>> {

        template <class ...Xs>
        static //constexpr
        future<T> make(Xs&& ...xs) {
            return make_ready_future<T>(forward<Xs>(xs)...);
        }
    }

    namespace future {
        struct traits<future<void>> {

            static //constexpr
            future<void> make() {
                return make_ready_future();
            }
        }
    }

};
```

### How to define a type constructor?

The make function is already useful with the class template parameter. However, we need in some cases the high-order interface, so that the user is able to have some context.

The simple case is when the class has a single template parameter as is the case for `future<T>`.
namespace boost
{
    struct future_tc {
        template <class T>
        using invoke = future<T>;
    };
}

When the class has two parameters and the underlying type is the first template parameter, as it is the case for `expected`,

namespace std::experimental
{
    template <class E>
    struct expected_tc<E> {
        template <class T>
        using invoke = expected<T, E>;
    };
}

If the second template depends on the first one as it is the case of `unique_ptr<T, D>`, the rebinding of the second parameter must be done explicitly.

namespace std
{
    template <class D, class T>
    struct rebind;
    template <template <class...> class TC, class ...Ts, class ...Us>
    struct rebind<TC<Ts...>, Us...> {
        using type = TC<Us...>;
    };
    template <class M, class ...Us>
    using rebind_t = typename rebind<M, Us...>::type;
}

struct default_delete_tc
{
    template <class T>
    using invoke = default_delete<T>;
};

template <class D>
struct unique_ptr_tc
{
    template <class T>
    using invoke = unique_ptr<T, detail::rebind_t<D, T>>;
};
}
**Helper classes**

Defining these type constructors is cumbersome. This task can be simplified with some helper classes. P0343R0 presents these helper classes.

The previous type constructors could be rewritten using these helper classes as follows:

```cpp
namespace std {
    template <>
    struct future<experimental::_t> : experimental::meta::quote<future> {};
}

namespace std {
    namespace experimental {
        template <class E>
        struct expected<_t, E> : meta::bind_back<expected, E> {};
    }
}

namespace std {
    template <>
    struct default_delete<experimental::_t> : experimental::meta::quote<default_delete> {};

    template <class D>
    struct unique_ptr<experimental::_t, D>
    {
        template<class T>
        using invoke = unique_ptr<T, experimental::meta::rebind_t<D, T>>;
    }
}
```

**Design rationale**

**Customization point**

This proposal uses a trait to customize the behavior.
namespace std::experimental {
inline namespace fundamental_v3 {
namespace type_constructible {
    template <class T>
    struct traits_default {
        template <class ...Xs>
        constexpr auto make(Xs&& xs) {
            return T{forward<Xs>(xs)...};
        }
    };
}}

Alternatively, we could have used of overloading a `make_custom` function found by ADL having an additional `type<T>` parameter.

```cpp
    template <class T, class ...Xs>
    constexpr auto make(type<T>, Xs&& xs)
```

**Why the factory has 3 flavors?**

The first `make` factory uses default constructor to build a `C<void>`.

The second `make` factory uses conversion constructor from the underlying type(s).

The third `make` factory is used to be able to do emplace construction given the specific type.

`reference_wrapper<T>` overload to deduce `T&`

As it is the case for `make_pair` when the parameter is `reference_wrapper<T>`, the type deduced for the underlying type is `T&`.

**Why do we use defaulted integer parameters**

```
int = 0, int...>
```

If we had the following overload

```cpp
    template <class TC, class T>
    future<experimental::meta::decay_unwrap_t<T>> make_ready_future(T&& x); // (1)
```

the following call will be accepted by (1) resulting in a `future<int>`, as the type is decayed.
```cpp
int v=0;
std::future<int&> x = std::experimental::make_ready_future<int&>(v);
```

Adding at least a default int template parameter as follows:

```cpp
template <int=0, ...int, class T>
future<experimental::meta::decay_unwrap_t<T>> make_ready_future(T&& x); // (1)
template <class T, class ...Args>
future<T> make_ready_future(Args&&... args); // (2)
```

avoid the selection of overload (1) and selects (2).

## Product types factories

This proposal takes into account also *product type* factories (as `std::pair` or `std::tuple`).

```cpp
// make product factory overload: Deduce the resulting `Us`
template <template <class...> class TC, class ...Xs>
   TC<decay_unwrap_t<Xs>...> make(Xs&&... xs);
// make product factory overload: Deduce the resulting `Us`
template <class TC, class ...Xs>
   invoke<TC, decay_unwrap_t<Xs>...> make(Xs&&... xs);
```

```cpp
auto x = make<pair>(1, 2u);
auto x = make<tuple>(1, 2u, string("a"));
```

## High order factory

It is simple to define a high order `maker<TC>` factory of factories that can be used in standard algorithms.

For example:

```cpp
std::vector<X> xs;
std::vector<Something<X>> ys;
std::transform(xs.begin(), xs.end(), std::back_inserter(ys), maker<Something>{});
```

where
The main problem defining function objects is that we cannot have the same class with different template parameters. The previous `maker` class template has a template class parameter. We need an additional class that takes a type constructor or a type.

Now we can define a `maker` factory for high-order `make` functions as follows:
template <class T>
// requires not is_type_constructor<T>{}
maker_t<T> maker() { return maker_t<T>(); }

template <class TC>
// requires is_type_constructor<TC>()
maker_tc<TC> maker() { return maker_tc<TC>(); }

template <template <class ...> class TC>
maker_tmpl<TC> maker() { return maker_tmpl<TC>(); }

The previous example would be instead

```
std::vector<X> xs;
std::vector<Something<X>> ys;
std::transform(xs.begin(), xs.end(), std::back_inserter(ys), maker<Something>());
```

Note the use of () instead of {}.

**Impact on the standard**

These changes are entirely based on library extensions and do not require any language features beyond what is available in C++14. There are however some classes in the standard that needs to be customized.

**Proposed wording**

The proposed changes are expressed as edits to [N4564](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4564.html) the Working Draft - C++ Extensions for Library Fundamentals V2.

The current wording make use of `decay_unwrap_t` as proposed in [P0318R0](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n0450.html), but if this is not accepted the wording can be changed without too much troubles.

The current wording make use of some meta-programming utilities defined in [P0343R0](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n0450.html).

**General utilities library**

--- Insert a new section. ---

X.Y Factories [functional.factories]

X.Y.1 In General

X.Y.2 Header synopsis

```
namespace std::experimental {
```
inline namespace fundamental_v3 {
namespace type_constructible {

    template <class T>
    struct traits_default;
    template <class T>
    struct traits : traits_default<T> {};

    // make() overload
    template <template <class ...> class M>
        M<void> make();

    // requires a type constructor
    template <class TC>
        meta::invoke<TC, void> make();

    // requires a template class parameter, deduce the underlying type
    template <template <class ...> class Tmpl, class ...Xs>
        Tmpl<meta::decay_unwrap<Xs>...> make(Xs&& ...xs);

    // requires a type constructor, deduce the underlying types
    template <class TC, class ...Xs>
        meta::invoke<TC, decay_unwrap<Xs>...> make(Xs&& ...xs);

    // make overload: don't deduce the underlying types,
    // don't deduce the underlying type from Xs
    // requires M is not a type constructor
    template <class M, class ...Xs>
        M make(Xs&& ...xs);

    template <class TC>
    struct maker_tc;

    template <template <class> class T>
    struct maker_tmpl;

    template <class T>
    struct maker_t;

    // requires a type constructor
    template <class TC>
        maker_tc<TC> maker();

    // requires T is not a type constructor
    template <class T>
        maker_t<T> maker();

    template <template <class ...> class TC>
        maker_tmpl<TC> maker();
}
}
X.Y.3 Template function `make`

X.Y.4 template + void

```cpp
template <template <class ...> class M>
M<void> make();
```

Equivalent to:

```cpp
make<type_constructor_t<meta::quote<M>>>();
```

X.Y.5 template + deduced underlying type

```cpp
template <template <class ...> class M, class ...Xs>
M<decay_unwrap_t<Xs>...> make(Xs&& ...xs);
```

Equivalent to:

```cpp
make<type_constructor_t<meta::quote<M>>>(std::forward<Xs>(xs)...);
```

X.Y.6 type constructor + deduced underlying types

```cpp
template <class TC, class ...Xs>
meta::invoke<TC, decay_unwrap_t<Xs>...> make(Xs&& ...xs);
```

**Effects:** Forwards to the customization point. As if

```cpp
return traits<meta::invoke<TC, decay_unwrap_t<Xs>...>::make(std::forward<Xs>(xs))...;
```

**Remark:** This function shall not participate in overload resolution until `TC` is a type constructor invocable with the deduced type of the parameters.

X.Y.7 type + non deduced underlying type

```cpp
template <class M, class ...Xs>
M make(Xs&& ...xs);
```

**Effects:** Forwards to the customization point. As if
\texttt{return traits-M::make(std::forward<Xs>(xs)...);}\

\textit{Remark:} This function shall not participate in overload resolution if \texttt{meta::is_callable<TC(deduced_type_t<Xs>...)>::value}.

\textbf{X.Y.8 Class template traits_default}

\begin{verbatim}
template <class T>
struct traits_default
{
    template <class ...Xs>
    static constexpr auto make(Xs&& ...xs)
    -> decltype(T(std::forward<Xs>(xs)...))
    {
        return T(std::forward<Xs>(xs)...);
    }
};
\end{verbatim}

Default customization point for classes defining the constructor.

\textit{Returns:} A \texttt{T} constructed using the constructor \texttt{T(std::forward<Xs>(xs)...)}

\textit{Throws:} Any exception thrown by the constructor.

\textit{Remark:} \texttt{traits_default<T>::make} function shall not participate in overload resolution until \texttt{T(std::forward<Xs>(xs)...)} is well formed.

\section*{Example of customizations}

Next follows some examples of customizations that could be included in the standard

\textbf{optional}

Nothing to do other than saying that the \texttt{make} overloads are included in \texttt{<experimental/optional>}. Say that the \texttt{make} overloads are included in \texttt{<experimental/optional>}. 

\begin{verbatim}
namespace std {
namespace experimental {
    template <>
    struct optional<_t> : meta::quote<optional> {};
    template <class T>
    struct type_constructor<optional<T>> : meta::id<optional<_t>> {};
}
}
\end{verbatim}
Say that the `make` overloads are included in `<experimental/expected>`.

```cpp
namespace std {
namespace experimental {
    template <class E>
    struct expected<std::experimental::_t, E>
    : std::experimental::meta::bind_back<expected, E> {}

    template <class T, class E>
    struct type_constructor<expected<T, E>> : meta::id<expected<_t, E>> {};
}
}
```

This customization depends on P0319R1. This means that it will be difficult to add it until P0319R1 or this proposal is in the IS. Otherwise we will introduce dependencies between two TS.
namespace std::experimental::type_constructible {
    template <class T>
    struct traits<future<T>> {
        template <class ...Xs>
        static future<T> make(Xs&& ...xs)
        {
            return make_ready_future<T>(forward<Xs>(xs)...);
        }
    };
    template <>
    struct traits<future<void>> {
        static future<void> make()
        {
            return make_ready_future();
        }
    };
    template <class T>
    struct traits<shared_future<T>> {
        template <class ...Xs>
        static shared_future<T> make(Xs&& ...xs)
        {
            return make_ready_future<T>(forward<Xs>(xs)...).share();
        }
    };
    template <>
    struct traits<shared_future<void>> {
        static shared_future<void> make()
        {
            return make_ready_future().share();
        }
    };
}

unique_ptr

Say that the make overloads are included in <experimental/memory>.
namespace std::experimental::type_constructible {
    template <class T, class D>
    struct traits<unique_ptr<T, D>>
    {
        template <class ...Xs>
        static
        unique_ptr<T, D> make(Xs&& ...xs)
        {
            return make_unique<T>(forward<Xs>(xs)...);
        }
    }
}

namespace std::experimental::type_constructible {
    template <class T>
    struct traits<shared_ptr<T>>
    {
        template <class ...Xs>
        static
        shared_ptr<T> make(Xs&& ...xs)
        {
            return make_shared<T>(forward<Xs>(xs)...);
        }
    }
}

Nothing to do other than saying that the make overloads are included in //experimental/utility>.

tuple

Nothing to do other than saying that the make overloads are included in //experimental/tuple>.

array

Say that the make overloads are included in //experimental/array>.
namespace std::experimental::type_constructible {
    struct array_tc {
    
        template <class ...T>
        using invoke = array<common_type_t<decay_t<T>..., sizeof...(T)>;}
    
    // type_constructor customization
    template <class T, size_t N>
    struct type_constructor<array<T, N>> : meta::id<array_tc> {}; 
    
    template <class T, size_t N>
    struct traits<array<T, N>>
    {
        template <class ...Xs>
        static constexpr array<T, sizeof...(Xs)> make(Xs&& ...xs)
        {
            return {{forward<Xs>(xs)...}};
        }
    }
};

Implementability

There is a partial implementation at https://github.com/viboes/std-make/include/experimental/funcdamental/v3/factory.

Open points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:

- Should the customization be done with overloading or with traits?
  
  The current proposal uses traits. The alternative is to use overloading.

- If overloading is preferred, should the customization function names be suffixed e.g. with _custom?

- Should the high-order function factory maker be part of the proposal?

- Should the resulting Callable from the high-order function factory maker be implementation defined as it is the result of std::bind?

- Should the function factories make be high-order function objects?

  N4381 proposes to use function objects as customized points, so that ADL is not involved.
This has the advantages to solve the function and the high order function at once.

The same technique is used a lot in other functional libraries as Range-V3, Fit and Pure.

The authors don't know how to manage with a single function object for the 3 kind of interfaces. And so there will be 3 function objects that should be named. The authors believe that the proposed high-order function factory maker is more appropriated.

Acknowledgements

Many thanks to Agustín K-ballo Bergé from which I learn the trick to implement the different overloads. Scott Pager helped me to identify a minimal proposal, making optional the helper classes and of course the addition high order functional factory and the missing reference_wrapper overload.

Thanks to Mike Spertus for its P0091R4 proposal that help to avoid the factories in some cases.

Special thanks and recognition goes to Technical Center of Nokia - Lannion for supporting in part the production of this proposal.

References

- **N4381** - Suggested Design for Customization Points  
  http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4381.html

- **N4564** N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 PDTS  

- **P0091R4** - Template parameter deduction for constructors (Rev. 6)  
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0091r4.html

- **P0196R2** - Generic none() factories for Nullable types  
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0196r2.pdf

- **P0318R0** decay_unwrap and unwrap_reference  
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0318r0.pdf

- **P0319R1** - Adding Emplace functions for promise<T>/future<T>  

- **P0323R2** - A proposal to add a utility class to represent expected monad (Revision 2)  

- **P0343R0** - Meta-programming High-Order functions
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0343r0.pdf

- **Range-V3**
  - https://github.com/ericniebler/range-v3

- **Meta**
  - https://github.com/ericniebler/meta

- **Boost.Hana**
  - https://github.com/ldionne/hana

- **Pure**
  - https://github.com/splinterofchaos/Pure

- **Fit**
  - https://github.com/pfultz2/Fit