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.

`P0091R0` 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 the case, as `P0091R0` is a feature for the user that cannot be used in generic code.

**Table of Contents**

1. Introduction
2. Motivation and scope
3. Proposal
4. Design rationale
5. Proposed wording
6. Implementability
7. Open points
8. Acknowledgements
9. References

**Introduction**

This paper presents a proposal for a family of generic factories `make<TC>(args...)` that allow to make generic algorithms that need to create an instance of a wrapping class `TC` from the underlying types.

`P0091R0` 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 the case.

**Motivation and scope**

**Possible valued types**

All these types, `shared_ptr<T>`, `unique_ptr<T,D>`, `optional<T>`, `expected<T,E>` (see `P0323R0`) and
There are two kinds of factories:

- type constructor with the underlying types as parameter
  - `back_inserter`
  - `make_optional`
  - `make_ready_future`
  - `make_expected`
- emplace construction of the underlying type given the constructor parameters
  - `make_shared`
  - `make_unique`

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(a);
  //return unique_ptr(a); // this should be correct with [P0091R0]
}
```

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(a);
  //return shared_ptr(a); // this should be correct with [P0091R0]
}
```

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

```cpp
template <template <class> class TC, class T>
TC<T> f() {
  T a,
  ...
  return make<TC>(a);
  //return TC(a); // This should not work even with [P0091R0]
}
```

Another generic example: Suppose that `PossiblyValued` types define a `pv.value()` and a `pv.has_value()` functions and the traits `value_type_t<pv>`. For these kind of classes, we can define the `functor_map` function as follows.
template <class Callable, class PossiblyValued>
auto functor_map f(Callable c, PossiblyValued pv)
  -> rebind_t<PossiblyValued, decltype(c(pv.value()))>[
  
    if (pv.has_value())
      return make_type_constructor_t<PossiblyValued>(c(pv.value()));
    else
      return none_type_constructor_t<PossiblyValued>();
]

Product types

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

- make_pair
- make_tuple

Comparasion

<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_shared_ptr&lt;int&gt;(v);</td>
</tr>
<tr>
<td>auto x2 = make_unique&lt;int&gt;(v);</td>
<td>auto x2 = make_unique_ptr&lt;int&gt;(v);</td>
</tr>
<tr>
<td>auto x3 = make_optional(v);</td>
<td>auto x3 = make_optional&lt;int&gt;(v);</td>
</tr>
<tr>
<td>auto x4v = make_ready_future();</td>
<td>auto x4v = make_future();</td>
</tr>
<tr>
<td>auto x4 = make_ready_future(v);</td>
<td>auto x4 = make_future(v);</td>
</tr>
<tr>
<td>auto x5v = make_ready_future().share();</td>
<td>auto x5v = make_shared_future();</td>
</tr>
<tr>
<td>auto x5 = make_ready_future(v).share();</td>
<td>auto x5 = make_shared_future(v);</td>
</tr>
<tr>
<td>auto x6v = make_expected();</td>
<td>auto x6v = make_expected();</td>
</tr>
<tr>
<td>auto x6 = make_expected(v);</td>
<td>auto x6 = make_expected(v);</td>
</tr>
<tr>
<td>auto x7 = make_pair(v, v);</td>
<td>auto x7 = make_pair(v, v);</td>
</tr>
<tr>
<td>auto x8 = make_tuple(v, v, 1u);</td>
<td>auto x8 = make_tuple(v, v, 1u);</td>
</tr>
<tr>
<td>future&lt;int&amp;&gt; x4r = make_ready_future(std::ref(v));</td>
<td>future&lt;int&amp;&gt; x4r = make_future(std::ref(v));</td>
</tr>
<tr>
<td>auto x1 = make_shared&lt;A&gt;(v, v);</td>
<td>auto x1 = make_shared_ptr&lt;A&gt;(v, v);</td>
</tr>
<tr>
<td>auto x2 = make_unique&lt;A&gt;(v, v);</td>
<td>auto x2 = make_unique_ptr&lt;A&gt;(v, v);</td>
</tr>
<tr>
<td>auto x3 = make_optional&lt;A&gt;(v, v);</td>
<td>auto x3 = make_optional&lt;A&gt;(v, v);</td>
</tr>
<tr>
<td>auto x4 = make_ready_future&lt;A&gt;(v,v);</td>
<td>auto x4 = make_future&lt;A&gt;(v,v);</td>
</tr>
<tr>
<td>auto x5 = make&gt;(v, v);</td>
<td>auto x5 = make_shared_future&lt;A&gt;(v, v);</td>
</tr>
<tr>
<td>auto x6 = make_expected&lt;A&gt;(v, v);</td>
<td>auto x6 = make_expected&lt;A&gt;(v, v);</td>
</tr>
</tbody>
</table>

We can use the class template name as a type constructor
vector<int> vi1 = { 0, 1, 2, 3, 5, 8 };
vector<int> vi2;
copy_n(vi1, 3, make_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

int v=0;
future<int&> x4 = make_future(std::ref(v));

or use the class name to build to support in place construction

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 `P0091R0`, the following is already possible

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

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`.

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

Type constructor factory

```cpp
template <class TC>
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

```cpp
C();
```

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

```cpp
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::factory_traits<T>` class providing the needed overloads for `make`.
namespace experimental
{
    template <class T>
    struct factory_traits<future<T>> {

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

    template <>
    struct factory_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 simple case is when the class has a single template parameter as is the case for `future<T>`.

```cpp
```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`,

```cpp
```namespace boost
{
    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.
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> {};
}
```

```cpp
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>
        struct 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 {
    namespace experimental {
    
    inline namespace fundamental_v3 {
    
    template <class T>
    struct make_traits_default {
    
        template <class ...Xs>
        constexpr auto make(Xs&& xs) {
            return T{forward<Xs>(xs)...};
        }
    
    }
    
    }
    
    }
}

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

    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
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 <typename ...Xs>
TC<decay_unwrap_t<Xs>...> make(Xs&& ...xs);

// make product factory overload: Deduce the resulting `Us`
template <typename ...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

```cpp
template <typename ...X>
struct maker {
    template <typename ...X>
    constexpr auto operator()(X&& ...x) const
    {
        return make<T>(forward<X>(x)...);
    }
};
```

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.
template <template <class> class Tmpl>
struct maker_tmpl {
    template <typename ...X>
    constexpr auto operator()(X&& ...x) const
    {
        return make-Tmpl>(forward<X>(x)...);
    }
};

template <class TC>
struct maker_tc {
    template <typename ...Args>
    constexpr auto operator()(Args&& ...args) const
    {
        return make-TC>(forward<Args>(args)...);
    }
};

template <class T>
struct maker_t {
    template <class ...Args>
    constexpr auto operator()(Args&& ...args) const
    {
        return make<T>(std::forward<Args>(args)...);
    }
};

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 the Working Draft - C++ Extensions for Library Fundamentals V2.

The current wording make use of \texttt{decay\_unwrap\_t} as proposed in P0318R0, 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.

General utilities library

-------------------------------------------------------
Insert a new section.  
-------------------------------------------------------

X.Y Factories [functional.factories]

X.Y.1 In General

X.Y.2 Header synopsis

\begin{verbatim}
namespace std
{
namespace experimental
{
inline namespace fundamental_v3
{

    template <class T>
    struct factory_traits_default;
    template <class T>
    struct factory_traits : factory_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();

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

    // make overload: 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,
\end{verbatim}
// 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

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

Effects: Forwards to the customization point. As if

    return make<type_constructor_t<meta::quote-M>>();

X.Y.5 template + deduced underlying type

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

Effects: Forwards to the customization point. As if

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

X.Y.6 type constructor + deduced underlying type
template <class TC, class ...Xs>
meta::invoke<TC, decay_unwrap<Xs>...> make(Xs&& ...xs);

Effects: Forwards to the customization point. As if

\[
\text{return factory_traits<meta::invoke<TC, deduced_type_t<Xs>...>::make(std::forward<Xs>(xs)...)};
\]

Remark: This function shall not participate in overload resolution until

\[
\text{meta::is_callable<TC(deduced_type_t<Xs>...)>::value}.
\]

X.Y.7 type + non deduced underlying type

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

Effects: Forwards to the customization point. As if

\[
\text{return factory_traits<M>::make(std::forward<Xs>(xs)...});
\]

Remark: This function shall not participate in overload resolution if

\[
\text{meta::is_callable<TC(deduced_type_t<Xs>...)>::value}.
\]

X.Y.8 Class template factory\texttt{traits\texttt{default}}

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

Default customization point for classes defining the constructor.

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

Throws: Any exception thrown by the constructor.

Remark: \texttt{factory_traits_default<T>::make} function shall not participate in overload resolution until

\( T(\text{std::forward<Xs>(xs)...}) \) is well formed.

Example of customizations

Next follows some examples of customizations that could be included in the standard
namespace std {
namespace experimental {
    template <class T>
    struct factory_traits<future<T>> {
        template <class ...Xs>
        static future<T> make(Xs&& ...xs) {
            return make_ready_future<T>((forward<Xs>(xs))...);
        }
    };
    template <>
    struct factory_traits<future<void>> {
        static future<void> make() {
            return make_ready_future();
        }
    };
    template <class T>
    struct factory_traits<shared_future<T>> {
        template <class ...Xs>
        static shared_future<T> make(Xs&& ...xs) {
            return make_ready_future<DX>(forward<Xs>(xs)).share();
        }
    };
    template <>
    struct factory_traits<shared_future<void>> {
        static //constexpr
        shared_future<void> make() {
            return make_ready_future().share();
        }
    };
}
namespace std {
namespace experimental {
    template <class T, class D>
    struct factory_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 {
namespace experimental {
    template <class T>
    struct factory_traits<shared_ptr<T>>
    {
        template <class ...Xs>
        static
        shared_ptr<T> make(Xs&& ...xs)
        {
            return make_shared<T>(forward<Xs>(xs)...);
        }
    }
}
}

Nothing to do.

Nothing to do.

Implementability

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

Open points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:
• Is there an interest on the `make` factories?

• 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. An 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-ballos 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 `P0091R0` proposal that would even help to avoid the factories in the common cases.

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
  

• `P0091R0` - Template parameter deduction for constructors (Rev. 3)
  
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html

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

• `P0319R0` - Adding Emplace functions for `promise<T>/future<T>`
  
  http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0319r0.pdf
- **P0323R0** - A proposal to add a utility class to represent expected monad (Revision 2)
  
  [http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0323r0.pdf](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0323r0.pdf)

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

- **Range-V3**
  
  [https://github.com/ericniebler/range-v3](https://github.com/ericniebler/range-v3)

- **Meta**
  
  [https://github.com/ericniebler/meta](https://github.com/ericniebler/meta)

- **Boost.Hana**
  
  [https://github.com/ldionne/hana](https://github.com/ldionne/hana)

- **Pure**
  
  [https://github.com/splinterofchaos/Pure](https://github.com/splinterofchaos/Pure)

- **Fit**
  
  [https://github.com/pfultz2/Fit](https://github.com/pfultz2/Fit)