Document number:	DXXXX=yy-nnnn
Date:	2016-05-02
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

C++ generic factories

---------------------- DRAFT ----------------------

Abstract

Experimental generic factories library for C++.

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 generic factories `make<TC>(v)` that allows to make generic algorithms that need to create an instance of a wrapped class `TC` from their underlying types.

P0091R0 extends template parameter deduction for functions to constructors of template classes. With this feature, it would seam clear that this proposal lost most of its added value but this is not the case.

Motivation and scope

All these types, `shared_ptr<T>` , `unique_ptr<T,D>` , `optional<T>` , `expected<T,E>` and `future<T>` , 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`
    - `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>` .

```
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

```
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.

```
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]
}
```

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

- `make_pair`
- `make_tuple`

We can use the class template name as a type constructor

```
vector<int> vi1 = { 0, 1, 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

```
template <class TC>
  apply<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 constructor as

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

Emplace factory

TBC

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 to add the missing overloads for `make_custom` so that they can be found by ADL.

```
namespace boost {
  future<void> make_custom(meta::id<future<void>>)
  {
    return make_ready_future();
  }
  template <class T, class ...Args>
  future<T> make_custom(meta::id<future<T>>, Args&& ...args)
  {
    return make_ready_future<T>(forward<Args>(args)...);
  }
}
```

How to define a type constructor?

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 apply = future<T>;
  };
}
```

When the class has two parameter 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 apply = expected<T, E>;
  };
}
```

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

```
namespace boost
{
  namespace detail
  {
    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;
  }

  template <>
    struct default_delete<experimental::_t>
  {
    template<class T>
    using apply = default_delete<T>;
  };

  template <class D>
    struct unique_ptr<experimental::_t, D>
  {
    template<class T>
    using apply = 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.

```
  // type holder
  struct _t {};

namespace meta
{
  // identity meta-function
  template<class T>
    struct id
    {
      using type = T;
    };

  // lift a class template to a type constructor
  template <template <class ...> class TC, class... Args>
    struct lift;

  // reverse lift a class template to a type constructor
  template <template <class ...> class TC, class... Args>
    struct reverse_lift;

  template <class M, class ...U>
  struct rebind : id<typename M::template rebind<U...>> {};

  template <template<class ...> class TC, class ...Ts, class ...Us>
  struct rebind<TC<Ts...>, Us...> : id<TC<Us...>> {};

  template <class M, class ...Us>
  using rebind_t = eval<rebind<M, Us...>>;

}
```

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

```
namespace boost
{
  template <> struct future<_t> : std::experimental::meta::lift<future> {};
}
```

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

```
namespace boost
{

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

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

Design rationale

Customization point

This proposal takes advantage of overloading the `make_custom` functions adding the tag `id<T>`.

We have named the customization point `make_custom` to make more evident that these are customization point.

Alternatively, we could use a trait

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

};
}}}
```

```
namespace std
{
namespace experimental
{
inline namespace fundamental_v3
{

template <>
struct make_traits<future<void>>
{
    constexpr future<void> make()
    {
        return make_ready_future();
    }
};

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

}}}
```

Why a default customization point?

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

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

The third factory `make` 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&`.

Product types factories

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

```
  // 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>
    apply<TC, decay_unwrap_t<Xs>...> make(Xs&& ...xs);
```

```
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

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

where

```
  template <template <class> class T>
  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 `maker` class template has a template class parameter. We need an additional class that takes a type constructor or a type.

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

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

Now we can define a `maker` factory for high-order make functions as follows

```cpp
template <class T>
// requires is_type_constructor<T>()==false
maker_t<T> maker() { return maker_t<T>{}; }

template <class TC>
// requires is_type_constructor<TC>()==false
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

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

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.

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 `decay_unwrap_t` as proposed in P0318R0, but if this is not accepted the wording can be changed without too much troubles.

General utilities library

--- Insert a new section. ---

X.Y Factories [functional.factorires]

X.Y.1 In General

X.Y.2 Header synopsis

```
namespace std
{
namespace experimental
{
inline namespace fundamental_v3
{
namespace meta
{
  // apply a type constuctor TC to the type parameters Xs
  template<class TC, class... Xs>
    using apply = typename TC::template apply<Xs...>;

  // identity meta-function
  template <class T>
    struct id { using type = T; };
}

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

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

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

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

  // make overload: don't deduce the underlying types,
  // don't deduce the underlying type from Xs
  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();



namespace meta
{
  // default customization point for TC<void> default constructor
  template <class M>
    M make_custom(meta::id<M>);

  // default customization point for constructor from Xs
  template <class M, class ...Xs>
    M make_custom(meta::id<M>, Xs&& xs);
}
}
}
```

```
}
```

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 `make` with a template constructor `id<M<void>>`. As if

```
    return make_custom(meta::id<M<void>>{});
```

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 `make_custom` with a template constructor `meta::id<M<decay_unwrap<Xs>...>>`. As if

```
    return make_custom(meta::id<M<decay_unwrap<Xs>...>>{}, std::forward<T>(x));
```

X.Y.6 type constructor + deduced underlying type

```
  template <class TC, class ...Xs>
    meta::apply<TC, decay_unwrap<Xs>...> make(Xs&& ...xs);
```

Requires: `TC` is a type constructor.

Effects: Forwards to the customization point `make_custom` with a template constructor `meta::id<meta::apply<TC, decay_unwrap<Xs>>>`. As if

```
    return make_custom(meta::id<meta::apply<TC, decay_unwrap<Xs>>>{}, std::forward<X
```

X.Y.7 type + non deduced underlying type

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

Requires: `M` is not a type constructor.

Effects: Forwards to the customization point `make_custom` with a template constructor `meta::id<M>` . As if

```
    return make_custom(meta::id<M>{}, std::forward<Xs>(xs)...);
```

X.Y.8 constructor customization point

```
template <class M, class ...Xs>
  M make_custom(meta::id<M>, Xs&& ...xs);
```

Returns: A `M` constructed using the constructor `M(std::forward<Xs>(xs)...)`

Throws: Any exception thrown by the constructor.

Example of customizations

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

`optional`

```
namespace std {
namespace experimental {

  // Holder specialization
  template <>
  struct optional<_t>;

}
}
```

`expected`

```
namespace std {
namespace experimental {

  // Holder specialization
  template <class E>
  struct expected<_t, E>;
}
}
```

`future` / `shared_future`

```
namespace std {

  // customization point for template
  // (needed because std::experimental::future doesn't has a default constructor)
  future<void> make_custom(experimental::meta::id<future<void>>);

  // customization point for template
  // (needed because std::experimental::future doesn't has a conversion constructor)
  template <class DX, class X>
    future<DX> make_custom(experimental::meta::id<future<DX>>, X&& x);

  // customization point for template
  // (needed because std::experimental::shared_future doesn't has a default construc
  shared_future<void> make_custom(experimental::meta::id<shared_future<void>>);

  // customization point for template
  // (needed because std::experimental::shared_future<X> doesn't has a constructor f
  template <class DX, class X>
    shared_future<DX> make_custom(experimental::meta::id<shared_future<DX>>, X&& x);

  // Holder specializations
  template <>
    struct future<experimental::_t>;
  template <>
    struct future<experimental::_t&>;
  template <>
    struct shared_future<experimental::_t>;
  template <>
    struct shared_future<experimental::_t&>;
}
```

`unique_ptr`

```
namespace std {

  // customization point for template
  // (needed because std::unique_ptr doesn't has a conversion constructor)
  template <class DX, class ...Xs>
    unique_ptr<DX> make_custom(experimental::meta::id<unique_ptr<DX>>, Xs&& xs);

  // Holder customization
  template <class D>
  struct unique_ptr<experimental::_t, D>;

  template <>
  struct default_delete<experimental::_t>;

}
```

`shared_ptr`

```
namespace std {

  // customization point for template
  // (needed because std::shared_ptr doesn't has a conversion constructor)
  template <class DX, class ...Xs>
  shared_ptr<DX> make_custom(experimental::meta::id<shared_ptr<DX>>, Xs&& xs...);

  // Holder customization
  template <>
  struct shared_ptr<experimental::_t>;

}
```

Implementability

This proposal can be implemented as pure library extension, without any compiler magic support, in C++14.

There is an implementation at https://github.com/viboes/std-make.

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` functions?

* Should the customization be done with overloading or with traits?

  The current proposal uses overloading as customization point. The alternative is to use traits as e.g. the library Boost.Hana uses.

  If overloading is preferred,

  * should the customization function names be suffixed e.g. with `_custom` ?

* Should the namespace `meta` be used for the meta programming utilities `apply` and `id` ?
* Should the high-order function factory `maker` be part of the proposal?
* Should the function factories `make` be 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.

* Is there an interest on placeholder type `_t` ?

  While not need absolutely, it helps to define friendly the type constructors.

* Is there an interest on the helper meta-functions `id` , `lift` , `lift_reverse` and `rebind` ?

  If yes, should them be part of a separated proposal?

  There is much more on meta-programming utilities as show on the Meta library.

* Should the customization of the standard classes `pair` , `tuple` , `optional` , `future` , `unique_ptr` , `shared_ptr` be part of this proposal? Are there others standard types to customize?

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 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

- N4480 - Programming Languages — C++ Extensions for Library Fundamentals

  http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4480.html

- P0091R0 - Template parameter deduction for constructors (Rev. 3)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html

- P0318R0 `decay_unwrap` and `unwrap_reference`

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0318r0.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

- 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

Appendix - Helper Classes

In the original proposal there were some helper classes as `lift`, `reverse_lift`, `_t` and `id` that are not mandatory for this proposal. If the committee has interest, a specific proposal can be written.

```
namespace std
{
namespace experimental
{
inline namespace fundamental_v3
{
  // type placeholder
  struct _t {};

namespace meta
{
  // lift a class template to a type constructor
  template <template <class ...> class TC, class... Args>
    struct lift;

  // reverse lift a class template to a type constructor
  template <template <class ...> class TC, class... Args>
    struct reverse_lift;

  template <class M, class ...U>
  struct rebind : id<typename M::template rebind<U...>> {};

  template <template<class ...> class TC, class ...Ts, class ...Us>
  struct rebind<TC<Ts...>, Us...> : id<TC<Us...>> {};

  template <class M, class ...Us>
  using rebind_t = typename rebind<M, Us...>::type;

}}}}
```