Document number: DXXXX=yy-nnnn
Date: 02/10/22
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

C++ generic factory

Experimental generic factories library for C++17.

Contents

Introduction.......... 2
Motivation and Scope.......... 2
Tutorial.......... 4
  Type constructor factory.......... 4
  How to define a class that wouldn't need customization?.......... 5
  How to customize an existing class.......... 5
  How to define a type constructor?.......... 5
  Type constructor helper classes.......... 6
Design rationale.......... 7
  Why to have a generic make function?.......... 7
  Using a class template as type constructor.......... 7
  Why do we need the concept of type constructor then?.......... 8
  What about emplace factories?.......... 9
  Why to have customization points?.......... 10
  Customization point.......... 10
  reference_wrapper<T> overload to deduce T& 10
  Product types factories.......... 10
  High order factory.......... 10
Open points.......... 11
Technical Specification.......... 12
  Header <experimental/meta> Synopsis.......... 12
  Header <experimental/functional> Synopsis.......... 13
  DEDUCED_TYPE(T).......... 13
  Template function make 14
  Template function make_custom 15
Example of customizations.......... 15
  optional 15
  expected 15
  future/shared_future 16
  unique_ptr 16
  shared_ptr 17
Implementation.......... 17
Acknowledgements.......... 17
History.......... 17

References.. 18
Appendix - Helper Classes not part of this proposal.. 18

Introduction

This paper presents a proposal for a generic factory `make()` that allows to make generic algorithms that need to create an instance of a wrapped class from its underlying types.

P0091R0 proposes extending template parameter deduction for functions to constructors of template classes. If P0091R0 is adopted, it could seam that this proposal will 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 a 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); // would this be correct if P0091R0 is accepted?
}
```

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

```
template <class T>
shared_ptr<T> f() {
  T a,
  ...
  return make_shared(a);
  //return shared_ptr(a); // would this be correct if P0091R0 is accepted?
}
```

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.

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

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 x4 = make<future>(v);
auto x5 = make<shared_future>(v);
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>>();
auto x4 = make<future<A>>(v);
auto x5 = make<shared_future<A>>(v, v);
auto x6 = make<expected<A>>(v, v);
```

Note, if P0091R0 is accepted, the following will be already possible

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

However we can not do that for classes having non-member factories as `std::experimental::future`.

It is not clear how P0091R0 or an evolution of this proposal could be used inside a template. What is the type of the template parameter representing the type constructor? What would be the contents of `?T?` and `?R?` in the following function template.

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

Would the following be correct if P0091R0 is adopted?

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

We can also make use of the class 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);
```

Tutorial

Type constructor factory

```
template <class TC>
  meta::apply<TC, int> safe_divide(int i, int j)
{
  if (j == 0)
    return none<TC>();
  else
    return make<TC>(i / j);
}
```

where meta::apply<TC, T> applies a type constructor to a type resulting in another type, `none<TC>` returns a not-a-value associated to `TC` implicitly convertible to `apply<TC,T>` for any `T`.

We can use this function with different type constructor as

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

or

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

Here `optional<_t>` is a type constructor, `_t` is a type placeholder used here to make evident that we are constructing `optional<T>` types.

How to define a class that wouldn't need customization?

`make` is defined by default when the type defines a conversion from the make arguments. `make<TC>(x1, …, xn)` is equivalent to

```
apply<TC, X1, … Xn>(xs...);
```

How to customize an existing class

When the existing class doesn't provide the needed constructor as e.g. `boost::uture<T>`, the user needs to add the missing overloads for the customizable `make_custom` function so that they can be found by ADL. This function has a tag parameter containing the type to construct.

```
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>(x)...);
  }
}
```

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>`.

```
namespace boost
{
  struct future_tc {
    template <class T>
    using apply = 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 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>>;
  };
}
```

Type constructor 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 tboost
{
  template <class E> struct expected<_t, E> :
      std::experimental::meta::reverse_lift<expected, E> {};
}

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

Why to have a generic make function?

The proposed generic make function is more verbose than the specific ones, so what is the advantage of such a generic function? This proposal doesn't proposes to remove the specific factories. When the user knows that s/he wants an `optional` the best is to use `make_optional`. The use of the generic make function has all its sense in the context of generic functions or classes templates.

Using a class template as type constructor

We can start with a factory that builds an instance of a class from its class template and deduce the template parameters from the function parameters.

```
template <template <class ...> class TC, class ...X>
TC<meta::deduced_type_t<X>...> make(X&& ...x);
```

This overload is enough to cover with `make_optional(v)`, `make_ready_future(v)`, `make_pair(v1,v2)` and `make_tuple(v1,v2, v3)`.

```
auto x = make<optional>(v);
auto x = make<future>(v);
auto x = make<pair>(v1,v2);
auto x = make<tuple>(v1,v2,v3);
```

However this overload doesn't work when there are no parameters, as there is no X to make `TC<meta::deduced_type_t<X>...>` well formed, so we need to add an overload for void

```
template <template <class> class TC>
TC<void> make();
```

This is needed in particular for `future<void> make_ready_future()`,

```
auto x = make<future>();
```

Why do we need the concept of type constructor then?

While templates can be used to build types providing all the template parameters, sometimes we have already fixed one of the parameters and need to provide an additional parameter. This is the case for the proposed class `expected`. When we build an `expected<int, error_code>` from an `int`, we have fixed the second type `error_code`. We need a type transformation from `int` to `expected<int, error_code>`. Let me call this transformation `expected_error_code` for example, so `expected_error_code<int>` should be `expected<int, error_code>`.

Now we can use the previous make function using the type transformation `expected_error_code`.

```
auto x = make<expected_error_code>(i);
```

If we need a transformation from `int` to `expected<int, exception_ptr>`, we could define a type constructor `expected_exception_ptr<int>`. But this doesn't scales. We need a way to build the type `expected<int, E>` where `E` can be previously fixed. We can obtain this by defining a class that acts as a type transformation function

```
template <class E>
```

```
struct expected_tc<E> {
  template <class T>
  using apply = expected<T, E>;
};
```

Now we need a type transformation that takes a transformation function and a type to build a type. The class `meta::apply` must be defined so that `meta::apply<expected_tc<error_code>, int>` is `expected<int, error_code>`. We say that `expected_tc<error_code>` is a type constructor.

The definition of meta::apply is very simple

```
template<class TC, class T>
using apply = typename TC::template apply<T>;
```

Now we can add `make` overloads having a type constructor instead of a class template

```
template <class TC>
  meta::apply<TC, void>  make();

template <class TC, class ...X>
  meta::apply<TC, meta::deduced_type_t<X>...> make(X&& ...x);
```

Note that these overload make sense only when TC is a type constructor and the types `meta::apply<TC, void>` and `meta::apply<TC, meta::deduced_type_t<X>...>` are well formed respectively.

Now we can construct

```
auto e = make<expected_tc<error_code>>(i);
```

What about emplace factories?

The previous examples deduces the type to build from the parameters type. However emplace factories as `make_shared` and `make_unique` don't deduce the result type from the parameters type. We need to add another overload to take care of this case

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

The problem introducing this 5^{th} overload is that it conflicts with the 4^{th} one. This means that we need to use SFINAE to avoid the the ambiguity. Once this is resolved, we can now use make as follows

```
auto sp = make<shared_ptr<X>>(a1, … an);
auto up = make<unique_ptr<X>>(a1, … an);
```

Note however that we can also deduce the type when we have a single parameter

```
auto sp = make<shared_ptr>(1);
auto up = make<unique_ptr>(1);
```

Why to have customization points?

The proposed library contains 5 overload of the function make. The user needs to customize only two overloads, one for `TC<void>` and one for `TC<Xs>`.

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.

Having these customization points allows to leverage the user of the difficulties to implement these overload. The user customizing this factories has simpler interface to customize, as the type to build is already deduced.

Customization point

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

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

`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 T, class ...Ts>
  T<Us...> make(Ts&& ...args);
// make product factory overload: Deduce the resulting `Us`
template <class TC, class ...Ts>
  apply<TC, Us...> make(Ts&& ...args);

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>{});

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 can not have the same class with different template parameters. The `maker` class template has a template class parameter. We need an additional classes that takes a meta-function class and a type.

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

template <class MFC> // requires MFC is a type constructor
  struct maker_mfc {
    template <class ...Xs>
    constexpr auto
    operator()(Xs&& ...xs)
    {
      return make<MFC>(std::forward<Xs>(xs)...);
    }
  };

template <class M> // requires M is a type
  struct maker_t
  {
    template <class ...Args>
    constexpr M operator()(Args&& ...args) const
    {
      return make<M>(std::forward<Args>(args)...);
    }
  };
```

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?**
- **Is there an interest on the none 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 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 function object factories be part of the proposal?**

  The function objects `maker_tc`, `maker_mfc` and `maker_t` could be quite useful.

  What should be the default for `maker`?
- **Should the function factories `make` and `none` 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, Fit and Pure.
- **Is there an interest on the helper holder `_t`?**

  While not need absolutely, it helps to define friendly type constructors.
- **Is there an interest on the helper meta-functions `types(type_list)`, `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?**

Technical Specification

Header <experimental/meta> Synopsis

Add the following declaration in experimental/functional.

```
namespace std
{
namespace experimental
{
inline namespace fundamental_v2
{
  namespace meta
  {
    template <class TC, class... Args>
      using apply = typename TC::template apply<Args...>;

    template<class T>
```

```
      struct id
    {
      using type = T;
    };
  }
}
```

Header <experimental/functional> Synopsis

Add the following declaration in experimental/functional.

```
namespace std
{
namespace experimental
{
inline namespace fundamental_v2
{

  template <class TC>
  constexpr auto none();

  template <template <class ...> class TC>
  constexpr auto none();


  template <class TC>
    meta::apply<TC, void> make();

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

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

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

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


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

  }
}
}
}
```

DEDUCED_TYPE(T)

Let `U` be `decay_t<T>`. Then `V` is `X&` if `U` Is the same as `reference_wrapper<X>`, otherwise `V` is `U`.

Template function `make`

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>>{});
```

Remarks: This overload would not participate in overload resolution until `make_custom(meta::id<M<void>>{})` is well formed.

template + deduced underlying types

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

where `Vs` is DEDUCED_TYPE(Ts):

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

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

Remarks: This overload would not participate in overload resolution until `make_custom(meta::id<M<Vs...>>{}, std::forward<Ts>(xs)...)` is well formed.

type constructor + deduced underlying types

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

where `V` is `DEDUCED_TYPE(T)`.

Requires: `TC` is a type constructor.

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

```
return make_custom(meta::id<meta::apply<TC, V...>>{}, std::forward<Ts>(xs)...);
```

Remarks: This overload would not participate in overload resolution until `make_custom(meta::id<meta::apply<TC, V...>>{}, std::forward<Ts>(xs)...)` is well formed.

type + non deduced underlying type

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

Requires: `M` is not a type constructor and the underlying type of `M` is convertible from `Xs....`

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)...);
```

Template function `make_custom`

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.

Remarks: This overload would not participate in overload resolution until `is_constructible_v<M, Xs&&>`.

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 {

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

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

  // (needed because std::experimental::shared_future doesn't has a default
constructor)
  shared_future<void> make_custom(experimental::meta::id<shared_future<void>>);

  // (needed because std::experimental::shared_future<X> doesn't has a
constructor from X)
  template <class DX, class ...Xs>
    shared_future<DX> make_custom(experimental::meta::id<shared_future<DX>>,
Xs&& xs...);


  // 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 customizations
  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>;

}
```

Implementation

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

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 of high order functional factory and the missing `reference_wrapper` overload.

Thanks to Mike Spertus for its P0091R0 proposal that would help to avoid the factories in the application code cases.

History

v0.1 Creation

v0.2 Take in account comments from the ML

- Moved `apply` and `type` to `meta` namespace.
- Added `constexpr`.
- Added product type factory overload `make` to support `pair/tuple` types.
- Fix the signature of `make` to support `reference_wrapper` types.
- Added factory function object `maker`.
- Added `none` factory.
- Removed the emplace `make` factory specialization.
- Remove `type_constructor` as out of the scope of the proposal. It was used by `unique_ptr<_t, D>` specialization, but this can be seen as an implementation detail.
- Remove `type_constructor_tag` as this was an implementation detail.
- Refactored `rebind`.
- Moved `rebind`, `lift`, `reverse_lift`, `_t` and `id` to appendix Helper Classes not part of this proposal and to `meta` namespace.

v0.3 Take in account comments from the ML

- Fix some product type and emplace factories issues.
- Rename customization point `make` to `make_custom`.
- Reference P0091R0.
- Replace meta::type by meta::id

References

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

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html
- 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
- N4015 - A proposal to add a utility class to represent expected monad

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4015.pdf
- Range-V3

  https://github.com/ericniebler/range-v3
- Meta

  https://github.com/ericniebler/meta
- Hana

  https://github.com/ldionne/hana
- Pure

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

  https://github.com/pfultz2/Fit

Appendix - Helper Classes not part of this proposal

In the original proposal there were some helper classes as `lift`, `reverse_lift`, `rebind`, and `_t` 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_v2
{
  // type holder
  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;

}}}}
```