Document number:	P0318R0
Date:	2016-05-22
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

`decay_unwrap` and `unwrap_reference`

Abstract

This paper proposes to introduce two new transformation type traits `unwrap_reference` and `decay_unwrap` associated to the type deduction when `reference_wrapper<T>` can be used to mean `T&`.

Table of Contents

Introduction

This paper proposes to introduce two new transformation type traits `unwrap_reference` and `decay_unwrap` associated to the type deduction when `reference_wrapper<T>` can be used to mean `T&`.

Motivation

There are some places in the standard where we can find wording such as

Returns: `pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y));` where `V1` and `V2` are

determined as follows: Let `Ui` be `decay_t<Ti>` for each `Ti`. Then each `Vi` is `X&` if `Ui` equals `reference_wrapper<X>`, otherwise `Vi` is `Ui`.

The intent is hard to catch and should be described only once as it is the case of `DECAY_COPY`, e.g. `DECAY_UNWRAP`.

In addition the author believes that using these kind of macros when we are able to define them using functions or traits makes the standard less clear.

Compare the previous wording to

Returns:
`pair<decay_unwrap_t<T1>, decay_unwrap_t<T2>>(std::forward<T1>(x), std::forward<T2>(y));`

If the traits are not adopted, the author suggest to use `DECAY_UNWRAP(T)` and define it only once on the standard.

This trait can already be used in the following cases

- [pair.spec] p8
- [tuple.creation] p2,3
- Concurrent TS P0159R0 `make_ready_future`

To the knowledge of the author `decay_unwrap` is used already in HPX, and in Boost.Thread as `deduced_type`.

The author plans to use it also in other factory proposals as the ongoing P0338R0 and P0319R0.

Proposal

We propose to:

- add an `unwrap_reference` type trait that unwraps a `reference_wrapper`;
- add a `decay_unwrap` type trait that decay and then unwraps if wrapped.

Design rationale

`unwrap_reference` type trait

Having a way to wrap a reference with `reference_wrapper` needs a way to unwrap it.

`decay_unwrap` can be defined in function of `decay` and a `unwrap_reference`.

It could be seen as an implementation detail, but seems useful.

`decay_unwrap` type trait

`decay_unwrap` can be considered as an implementation detail as it is equivalent to `unwrap_reference<decay_t<T>>` . However, the author find that it makes the wording much simpler.

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

This wording is relative to N4480.

General utilities library

20.9 Header `<functional>` synopsis

Change [function.objects], header synopsis, after reference_wrapper

```
namespace std {
  namespace experimental {
  inline namespace fundamentals_v3 {
    [...]

    template <class T>
      struct unwrap_reference;

    template <class T>
      struct decay_unwrap : unwrap_reference<decay_t<T>> {}

    template <class T>
      using decay_unwrap_t = typename decay_unwrap<T>::type;

    [...]
  }}
}
```

Add a subsection section

Transformation Type trait `unwrap_reference` [unwrapref]

```
template <class T>
struct unwrap_reference;
```

The member typedef type of `unwrap_reference <T>` shall equal `X&` if `T` equals `reference_wrapper<X>`, `T` otherwise.

20.3.3 Specialized algorithms [pairs.spec]

Replace 8 where V1 and V2 are ... by

where Vi is decay_unwrap.

220.4.2.4 Tuple creation functions [tuple.creation]

Replace 2 Let Ui ... by

Let `Ti` in `Types`, then each `Vi` in `VTypes` is `decay_unwrap_t<Ti>`.

Alternatively

If the traits are not adopted, the author suggest to use `DECAY_UNWRAP(T)` and define it only once on the standard as we do for `DECAY_COPY`.

Implementability

The implementation is really simple

```
template <class T>
struct unwrap_reference { using type = T; }
template <class T>
struct unwrap_reference<reference_wrapper<T>> { using type = T&; }

template <class T>
struct decay_unwrap : unwrap_reference<decay_t<T>> {}

template <class T>
using decay_unwrap_t = typename decay_unwrap<T>::type;
```

Open Points

The authors would like to have an answer to the following points if there is at all an interest in this proposal. Most of them are bike-shedding about the name of the proposed functions:

Do we want a `decay_unwrap` type trait?

If the traits is not adopted, the author suggest to use `DECAY_UNWRAP(T)`, define it only once on the standard and adapt [pair.spec] p8 and [tuple.creation] p2,3.

Do we want `DECAY_UNWRAP` instead?

Should it be named `unwrap_decay` instead?

As what it is really done is to first decay and then unwrap reversing would swapping the two words be better in English? A better name for `decay_unwrap` ?

Do we want a `unwrap_reference` ?

Acknowledgements

Thanks to Agustín Bergé K-ballo who show me that HPX uses these traits already.

References

- N4480 N4480 - Working Draft, C++ Extensions for Library Fundamentals

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html

- P0159R0 - Draft of Technical Specification for C++ Extensions for Concurrency

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0159r0.html

- P0319R0 Adding Emplace Factories for promise/future

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0319r0.pdf

- P0338R0 - C++ generic factories

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0338r0.pdf

- make.impl C++ generic factory - Implementation

  https://github.com/viboes/std-make/blob/master/include/experimental/std*make*v1/make.hpp

- Boost.Thread http://www.boost.org/doc/libs/1*60*0/doc/html/thread.html

- HPX http://stellar.cct.lsu.edu/files/hpx_0.9.8/html/hpx.html