Document number:	P0318R1
Date:	2018-03-30
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

`unwrap_ref_decay` and `unwrap_reference`

Abstract

This paper proposes to introduce two new transformation type traits `unwrap_reference` and `unwrap_ref_decay` 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 `unwrap_ref_decay` 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. `UNWRAP_REF_DECAY`.

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<unwrap_ref_decay_t<T1>, unwrap_ref_decay_t<T2>>(std::forward<T1>(x), std::forward<T2>(y));`

If the traits are not adopted, the author suggest to use `UNWRAP_REF_DECAY (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 `unwrap_ref_decay` 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.

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

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

`unwrap_ref_decay` type trait

`unwrap_ref_decay` 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++17.

Proposed wording

This wording is relative to N4727.

General utilities library

20.9 Header `<functional>` synopsis

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

```
namespace std {
    [...]

    template <class T>
      struct unwrap_reference;

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

    template <class T>
      using unwrap_ref_decay_t = typename unwrap_ref_decay<T>::type;

    [...]
}
```

Add a subsection section

Transformation Type trait `unwrap_reference` [function.objects.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.

23.4.3 Specialized algorithms [pairs.spec]

Replace

9 *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 `Viv is` X& `if` Ui `equals` reference_wrapper `, otherwise` Vi `is` Ui`.

by

Let `Vi` is `unwrap_ref_decay_t <Ti>`.

9 *Returns*: `pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y));`.

23.5.3.4 Tuple creation functions [tuple.creation]

Replace

2 The pack `VTypes` is defined as follows. Let `Ui` be `decay_t<Ti>` for each `Ti` in `TTypes`. If `Ui` is a specialization of `reference_wrapper`, then `Vi` in `VTypes` is `Ui::type&`, otherwise `Vi` is `Ui`.

by

2 Let `VTypes...` be `unwrap_ref_decay_t<TTypes>...`.

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 unwrap_ref_decay : unwrap_reference<decay_t<T>> {}

template <class T>
using unwrap_ref_decay_t = typename unwrap_ref_decay<T>::type;
```

Open question

Do we want this to be duplicated on the Fundamental TS?

We have some uses of these traits to simplify the wording of the `not_fn` function object in the Fundamental Ts N4600 and maybe also the `bind_front` function object P0356R2.

If desired the changes in `<functional>` could be duplicated to `<experimental/functional>`.

Do we want the Concurrent Ts to depend on the Fundamental TS?

We have some uses of these traits to simplify the wording of the `make_ready_function` factory the Concurrent Ts.

If the dependency is worth, additional changes could be applied to P0159R0.

2.10 Function template makereadyfuture [futures.makereadyfuture]

```
template <class T>
future<V> make_ready_future(T&& value);

future<void> make_ready_future();
```

Replace

3 Let `U` be `decay_t<T>`. Then `V` is `X&` if `U` equals `reference_wrapper<X>`, otherwise `V` is `U`.

by

3 Let `U` be `unwrap_ref_decay_t<T>`.

Acknowledgements

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

References

- N4600 - Working Draft, C++ Extensions for Library Fundamentals, Version 2

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4600.html#func.not_fn

- N4727 N4727 - Working Draft, Standard for Programming Language C++

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4727.pdf

- 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

- P0356R2 - Simplified partial function application

  www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/P0356R2.html

- make.impl C++ generic factory - Implementation

  https://github.com/viboes/std-make/blob/master/include/experimental/stdmakev1/make.hpp

- Boost.Thread http://www.boost.org/doc/libs/1600/doc/html/thread.html

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

History

Changes since p0318r0

Take in account the LEWG feedback from JAX 2017

- Maintain `unwrap_reference`

- Rename `decay_unwrap` to `unwrap_ref_decay`.
- Remove the open points.
- Update the wording and reduce the change to the IS C++20 and left the possibility to update the TS N4600 so that other TS's can profit of those traits as e.g. the Concurrent Ts V2 P0159R0.
- Added other ongoing proposals that could take advantage of this proposal.