Make `type_info` ordering methods `constexpr`

Gašper Ažman

2018-01-07

Document #: DxxxxR0
Date: 2018-01-07
Audience: Library Working Group
Reply-to: Gašper Ažman, gasper.azman@gmail.com

Contents

1 Overview

The `type_info::before(const type_info& rhs)` method provides for a run-time implementation-defined (but stable!) ordering of `type_info` objects, together with `operator==` and `operator!=`.[1]

Despite the fact that this ordering information is unchangeable at run-time, this method is not marked `constexpr`.

This paper explores the issues and their resolutions one might encounter when trying to mark this method `constexpr`, and proposes the obvious extensions of such a move.

2 Problem description

There is, in C++, currently no way to reliably and correctly sort types at compile time. Various hacks have been implemented in order to get around the issue this presents, from quadratic typesets to hardcoding sort keys in types, to tricking the compiler to give the name of the type (which incorrectly handles anonymous namespaces).

Sorting types at compile time is essential for well-performing typesets, such as ones required by various policy-based template libraries[2].

This presents unsolvable problems for libraries that provide types whose behaviour is configured using a set (not a list) of policies.

The inability to sort these policies into a canonical order results in having several ways of constructing an otherwise identical type.

[1] The equality operator is stronger than more than address equality - it is required to be consistent with `std::is_same`.
[2] polly, liberasure, layout-optimizing tuple implementations etc.

As an example, in `liberasure`, these two types should be identical:

```
using namespace erasure; using namespace std;
using movable_1 = any<move_constructible, move_assignable>;
using movable_2 = any<move_assignable, move_constructible>;
static_assert(is_same<movable_1, movable_2, "... unfortunately fails.");
```

However, due to the inability to establish a canonical order of the policies, there is no way of making the assertion pass in general.

This creates additional template instantiations every time such two `any` objects are used, prevents trivial move-assignment, and all kinds of other interoperability problems which could have been easily avoidable were we able to canonicalize the order at type creation.

There are similar issues with constructing `variant` objects from typelists (how do I make sure my dynamic type lists are in the same order everywhere?), and optimizing tuple layouts:

```
template <typename... Ts>
struct optimized_tuple
  : apply<std::tuple, stable_sort<size_of, typesort<Ts...>> {};

template <typename... Ts>
struct stable_variant
  : apply<std::variant, unique<typesort<Ts...>> {};
```

While `apply`, `stable_sort`, `size_of` and `unique`[3] are straightforward to implement (and just about every metaprogramming library has the implementations), `typesort` is currently impossible.

In effect, what is needed is the ability to define (and use) the following snippet:

```
template <typename T, typename U>
using is_before = std::integral_constant<bool, typeid(T).before(typeid(U))>;
```

Note that `T` and `U` cannot be glvalues of a polymorphic type, since they are not values.

3 Proposal

In [type.info], add:

```
bool operator==(const type_info& rhs) const noexcept constexpr;
bool operator!=(const type_info& rhs) const noexcept constexpr;
bool type_info::before(const type_info& rhs) const noexcept constexpr;
```

The `type_index` methods are implemented directly with `before`, so these should be constexpr as well. The constructor is obviously just a missing constexpr annotation:

In [type.index.overview], add:

```
type_index(const type_info& rhs) noexcept constexpr;
bool operator==(const type_index& rhs) const noexcept constexpr;
bool operator!=(const type_index& rhs) const noexcept constexpr;
bool operator< (const type_index& rhs) const noexcept constexpr;
bool operator<= (const type_index& rhs) const noexcept constexpr;
bool operator> (const type_index& rhs) const noexcept constexpr;
bool operator>= (const type_index& rhs) const noexcept constexpr;
```

4 Acknowledgments

[3] The $O(n^2)$ version, not the linear one.