Standardized Type Ordering

true true

2022-10-29

Contents

1 Abstract

Currently, `std::type_info` provides a stable but *implementation defined* order of types. Despite being unchangeable at runtime, the implementation defined `type_info::before` is not marked `constexpr`. This paper explores a standardized ordering of types in C++, as well as the impact of marking `type_info::before` constexpr.

2 Revision History

New Paper

3 Motivation

There is currently no way in C++ to sort types. Well-performing typesets, required by various policy-based template libraries, require constexpr evaluation of order.

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

The inability to sort these policies into a canonical order results in different types with the same behavior.

Strong ordering of types also allows typesets to produce the same symbol in different compilation units.

4 Proposal

#TODO Mention ordering of meta::info

4.1 Ordering of Types

We propose the following as a canonical way of sorting all types that are generated by the compiler. This order shall be available for constexpr evaluation. This proposal does not encompass type successors (i.e. `typeid(int).next()`), defining only a comparison on two known types. (i.e. `typeid(int).before(typeid(char))`)

4.1.1 Ordering Qualified Types

For any unqualified type T, its qualified versions are ordered immediately after it (preceding any other type or its own qualified versions) in the following manner:

Qualifiers are each assigned a score

&: 1 &&: 2 const: 3 volatile: 6

and ordering lowest-first after summing them.

For any unqualified type `T`, the order of all possible qualified types would be:

```
0  T
1  T &
2  T &&
3  T const
4  T const &
5  T const &&
6  T volatile
7  T volatile &
8  T volatile &&
9  T const volatile
10 T const volatile &
11 T const volatile &&
```

The remainder of the paper concerns itself only with unqualified types.

4.1.2 Ordering Scalar Types

We order scalar types before any compound types; built-in types first, followed by class types.

built-in types with simple names must be ordered before any types that reference other types.

In particular, scalar types should be ordered as follows:

1. `void` comes first because it's not reifiable,
2. `nullptr_t` as the first monostate

3. (any other monostates, if we ever add them, sorted alphabetically)
4. `bool` as the first bi-state
5. (any other bi-states, if we ever add them)
6. Raw-memory types (`char`, `signed char`, `unsigned char`, std::byte)
7. Integral types in order of size, signed before unsigned (`short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, followed by any implementation-defined wider integral types like ___int128_t etc.). Intersperse any implementation-defined built-in integral types as needed between the above.
8. Any remaining character types that are not type-aliases of any of the above, including unicode, according to the following rules: smallest first, unicode-specific variants after non-unicode variants.
9. Floating-point types, in order of size. In case of ties, `float`, `double` and `long double` come before any types from https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1467r9.html.
10. Enumeration types, (internally ordered by rules for class type ordering by name)
11. Function types (internally ordered by rules in section [function-types])
12. Pointer types (internally ordered by their pointee-type)
13. Pointer-to-member types (internally ordered by pointee-type)

Class types shall be ordered according to the rules below, see [ordering-classes]

4.1.3 Kinds

There are the following kinds in the grammar, and shall be ordered as follows from lowest to highest.

— value
— namespace
— function
— member_function
— function_template
— lambda_function
— type
— class_template
— variable_template

4.1.3.1 List of atoms

The following are atoms of ordering tuples. They shall be ordered from lowest to highest.

— scalar types
— — (pointer)
— [] (unknown bound array)
— [n] (known bound array of size n)
— kinds (see kinds)
— name of class
— … (variadic parameter)
— rank{n} (maybe)

4.1.4 namespaces

Namespaces shall be ordered alphabetically by comparing namespace names at the same rank (as if their names were tuples of those strings). A namespace comes before any of its subnamespaces.

Given:

```
namespace outer1 {
  struct i;
}

namespace outer2 {
  namespace inner1 {
    struct i;
  }
  namespace inner2 {
    struct i;
  }
}
```

The order of the three structs w/ type `i` types shall be `outer1::i < outer2::inner1::i < outer2::inner2::i`.

4.1.4.1 anonymous namespace

The anonymous namespace shall be ordered after its enclosing namespace but before any named namespaces.

4.1.5 OrderingTuple (Name TBD)

For the sake of clarity, we're going to define a construct called the `OrderingTuple`.

Types will be broken down into `OrderingTuple`s. Each value in the tuple will be a pair of Kind (see Kinds) + type.

These `OrderingTuples` will be ordered lexicographically.

Given

```
namespace foo::bar {
    struct i;
}

namespace baz {
  struct j;
}
```

`foo::bar::i` produces `((namespace, foo), (namespace, bar), (type, i)}`

`baz::j` produces `((namespace, baz), (type, j))`

When compared, these yield

`baz::j` < `foo::bar::i`, since `namespace baz` precedes `namespace foo`

4.1.6 Ordering Array Types

Array types shall be ordered after scalar types but before class types.

Order arrays first internally by element type, then by rank, then by rank bounds, lowest first. Arrays of unknown bounds come before arrays of known bounds.

So the order of the following, for a given type T:

```
T[]
T[10]
T[11]
T[][2]
```

```
T[10][2]
T[3][2]
```

shall be ordered `T[] < T[10] < T[11] < T[][2] < T[3][2] < T[10][2]`,

defining `OrderingTuples ((type, T), []) ((type, T), [10]) ((type, T), [11]) ((type, T), [], [2]) ((type, T), [10], [3]) ((type, T), [3], [2])`

4.1.7 Ordering Compound Types:

4.1.8 Ordering Class Types

Class types can be defined as either a `struct` or a `class`. As far as ordering is concerned, structs and classes should be treated the same.

Class types shall be greater than scalar types.

Since we cannot redeclare two types with the same name, class types shall be ordered alphabetically.

```
struct Apple {};
class Banana {};
struct Carrot {};
```

Would be ordered as `Apple < Banana < Carrot`

As such, we define the ordering tuples:

`(type, Apple) (type, Banana) (type, Carrot)`

5 Non Type Template Parameters

NTTPs are lexicographically ordered by their scalar subobjects.

5.1 Class Templates

Lets start with the simple case. Class templates are ordered by: 1. Class name, alphabetically 2. Template arguments, applied lexicographically.

For example, given:

```
template <typename T, typename U>
struct Apple;

struct Banana;
struct Carrot;

Apple<Banana, Carrot>;
Apple<Banana, Banana>;
Apple<Carrot, Carrot>;
```

would be ordered `Apple<Banana, Banana> < Apple<Banana, Carrot> < Apple<Carrot, Carrot>`.

We can represent this with tuples: `(class_template, Apple, (type, Banana), (type, Carrot))` `(class_template, Apple, (type, Banana), (type, Banana))` `(class_template, Apple, (type, Carrot), (type, Ca`

5.1.1 Function Types

Function types shall be ordered by

1. Return type

2. Parameters, lexicographically.

```
void foo(int i);
```

This function can be represented by: `(function, (type, void), (type, int))`

```
void foo(int)
void foo(int, double)
```

We can represent these types with `(function, (type, void), (type, int))` `(function, (type, void), (type, int), (t`

So, the type of `void foo(int)` would precede the type of `void foo(int, double)`

5.1.2 Member Function Types

Function types shall be ordered by

1. Return type
2. The type of the class it is a member of.
3. Parameters, lexicographically.

```
struct Foo {
  void bar(int i, float j);
};
```

Produces the following tuple representation `(member_function, (type, void), (type, Foo), (type, int), (type, floa`

5.1.3 Variadic Function Types

Variadic function shall be ordered in a similar way. In a variadic function, the last argument is a variadic argument. A variadic argument shall be ordered immediately after its underlying type.

Given:

```
void foo(Foo);
void foo(Foo...);
```

In this case, the type of `void foo(Foo...)` is ordered immediately after the type of `void foo(Foo)`.

We can represent these as: `(function (type, void) (type, Foo))` `(function (type, void) (type, Foo...))`

5.1.4 Function Template Types

Function templates are ordered after member functions. They shall be ordered: 1) By return type 2) By template parameters, lexicographically 3) By function parameters, lexicographically

Given

```
template <typename T, typename U>
T f(U);
```

The type of `f<char, int>` would produce the representation: `(function_template, (type, int), ((type, char), (type,`

5.2 Lambda Types

Lambda Types shall be ordered in the same manner as functions. Ties are broken by the point of instantiation.

```
namespace Banana {
 auto i = [](int) -> void {}; // 0th lambda instantiated in Banana
}

namespace Apple {
```

```
auto i = [](float) -> int {}; // 0th lambda instantiated in Apple
auto j = []() -> std::string {}; // 1st lambda instantiated in Apple
}
```

These would produce the following tuples: `((namespace Banana), (lambda, (type, void), (type, int), (value, 0))` `((namespace Apple), (lambda, (type, int), (type, float), (value, 0))` `((namespace Apple), (lambda, (type, s`

5.3 Parameter Packs

Parameter are ordered as class templates.

Given:

```
template<class... Types>
struct Tuple {};

class Foo {};
class Bar {};

Tuple<> t0;
Tuple<int> t1;
Tuple<Foo> t2;
Tuple<Bar> t3;
Tuple<Foo, Bar> t4;
```

would be ordered: `Tuple<>` < `Tuple<int>` < `Tuple<Bar>` < `Tuple<Foo>` < `Tuple<Foo, Bar>`

5.4 Kinds of Templates

Kinds of templates shall first be ordered by arity.

Given:

```
template <template <template<typename> class> class Template>
struct two{};

template <template <typename> class> struct one{};

template <typename> struct zero{};

zero<int> value0;
one<zero> value1;
two<one> value2;
```

These are represented by tuples: `(type, zero, (class_template, int))` `(type, one, (class_template, (type, zero, (` `(type, two, (class_template, (type, one, (class_template, rank{1}))))`

5.5 Variable Templates

Variable templates are ordered by the type of their template parameters.

```
template <typename T>
constexpr T pi = T(3.1415);
```

the type of pi can be represented as: `(variable_template, (type, int))`

6 Acknowledgements

Thanks to all of the following: - Davis Herring for his suggestions on ordering non-type template parameters.