Standardized Type Ordering

true true

2022-10-29

Contents

1 Abstract **1**

2 Revision History **1**

3 Motivation **2**

4 Proposal **2**
4.1 Ordering of Types 2
4.1.1 Ordering Qualified Types 2
4.1.2 Ordering Native Types 2
4.1.3 Ordering Compound types: 3
4.1.4 namespaces 3
4.1.5 anonymous namespace 3
4.1.6 Class Types 4

5 Value Ordering **4**
5.1 Class Templates 4
5.2 Function Types 5
5.2.1 Template Specializations 5
5.2.2 Variadic Function Types 5
5.2.3 Type Aliases 6
5.2.4 Type Alias Templates 6
5.3 Lambda Types 6
5.4 Concepts 6
5.5 Parameter Packs 6
5.6 Kinds of Templates 7

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.

4 Proposal

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 will 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 Native Types

We order scalar types before any compound types; bult-in types first, followed by user-defined 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)

4.1.3 Ordering Compound types:

Array types will 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]
```

would be ordered `T[] < T[10] < T[11] < T[][2] < T[3][2] < T[10][2]`

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

4.1.4 namespaces

Namespaces will 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 would be `outer1::i < outer2::inner1::i < outer2::inner2::i`.

4.1.5 anonymous namespace

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

4.1.6 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 native types.

Ordering these is simple, since we cannot redeclare two types with the same name, we'll just order simple class types alphabetically.

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

Would be ordered as `Apple < Banana < Carrot`

5 Value Ordering

Values can be used to create template specializations. Take, for example:

```
template <int N>
class Foo;
....
Foo<1>;
Foo<2>;
```

To account for this, we must define an order for values. Values shall be ordered first by their types, then by their individual values.

so in the case of

```
Foo<1>
Foo<2>
```

`Foo<1> < Foo<2>`

When ordering types with value template arguments, `typeinfo` shall use the `operator<=>`

Values without `operator<=>` cannot be compared, and shall be considered ill-formed. An example of this would be

```
struct Foo {
  int i;
  // Notice, no way to order two Foos
  constexpr Foo(int i_) : i{i_} {};

  friend auto operator<(Foo const& lhs, Foo const& rhs) -> bool = delete;
};

template <Foo T>
struct Bar {
    int j = T.i;
};
```

5.1 Class Templates

Lets start with the simple case. Class templates are ordered by: 1. Class name, alphabetically 2. Number of template arguments. 3. Order of template arguments applied left to right.

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>`.

5.2 Function Types

Function types will be ordered by

1. Name (alphabetically)
2. Number of parameters
3. Order of parameters. Applied left to right and ordered as described above.

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

First, order by name, so `void bar(int) < void foo(int)`

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

Next, order by number of parameters, so `void foo(int) < void foo(int, double)`

```
struct Apple {};
struct Banana {};

void foo(Apple);
void foo(Banana);
```

Finally, order each parameter left to right as described above, so `void foo(Apple) < void foo(Banana)`

Left to right meaning the order of the following:

```
struct Apple {};
struct Banana {};

void foo(Apple, Apple);
void foo(Apple, Banana);
void foo(Banana, Apple);
void foo(Banana, Banana);
```

Gives us the ordering `void foo(Apple, Apple) < void foo(Apple, Banana) < void foo(Banana, Apple) < void foo(B`

5.2.1 Template Specializations

Template specializations will be orded in the same way as class templates. First by name, then by the length of the argument list, compared as a tuple of arguments.

5.2.2 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 will be ordered after all possible qualified types of its underlying type.

Given:

```
void foo(int);
void foo(float, int);
void foo(int, float);
void foo(float, int...);
void foo(int, float...);
```

According to [ordering-qualified-types], int is ordered before float, so `foo(int) < foo(int, float) < foo(int, float...) <`

5.2.3 Type Aliases

Type aliases are not types, and we don't need to concern ourselves with how we order them.

They will be ordered exactly the same as the type they are aliased to.

5.2.4 Type Alias Templates

```
template <typename T>
using Foo = SomeType<T>;
```

Foo will be ordered exactly the same way as `SomeType<T>`

5.3 Lambda Types

Lambda Types will be ordered in the namespace in which they are declared. Ties are broken by the point of instantiation.

```
namespace Banana {
 auto i = [](){};
}

namespace Apple {
auto i = [](){};
auto j = [](){};
}
```

would be ordered:

`decltype(Apple::i)` < `decltype(Apple::j)` < `decltype(Banana::i)`

5.4 Concepts

Concepts do not need to be ordered, since they are not types, only restrict what types can be.

5.5 Parameter Packs

Parameter packs are ordered by size first, then types compared left to right.

Given:

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

class Foo {};
class Bar {};

Tuple<> t0;
Tuple<int> t1;
```

```cpp
Tuple<Foo> t2;
Tuple<Bar> t3;
Tuple<Foo, Bar> t4;
```

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

5.6 Kinds of Templates

Kinds of templates shall first be ordered by arity.

Given:

```cpp
template <template <template<typename> class> class Template>
struct two_arity{};

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

template <typename> struct zero_arity{};

zero_arity<int> value0;
one_arity<zero_arity> value1;
two_arity<one_arity> value2;
```

shall be ordered `decltype(value0) < decltype(value1) < decltype(value2)`.

If two types have the same arity, they will be ordered based on name.

Given:

```cpp
template <template <template<typename> class> class Template>
struct two_arityA{};
template <template <template<typename> class> class Template>
struct two_arityB{};

template <template <typename> class> struct one_arityA{};
template <template <typename> class> struct one_arityB{};

two_arityA<one_arityA> value1;
two_arityA<one_arityB> value2;
two_arityB<one_arityA> value3;
two_arityB<one_arityB> value4;
```

would be ordered `decltype(value1) < decltype(value2) < decltype(value3) < decltype(value1)`, as `two_arityA` comes before `two_arityB` and `one_arityA` comes before `one_arityB`.