Standardized Type Ordering

Document #:	P2830R0
Date:	2022-10-29
Project:	Programming Language C++
Audience:	EWG
Reply-to:	Nate Nichols <natenichols@cox.net> Gašper Ažman <gasper.azman@gmail.com>

Contents

1 Abstract

Currently, `std::type_info` provides a stable but *implementation defined* order of types. Despite being a compile-time property, the implementation defined `type_info::before` is not marked `constexpr`. The behavior of `type_info::before` cannot be changed with risking backwards compatibility issues. This paper explores a standardized ordering of types in C++, as well as exposing a separate library function called `type_order` that exposes the `<=>` operator for ordering types types.

`std::type_order` will provide the following interface

```
template <typename F>
struct type_order final {
    template <typename G>
    constexpr auto operator<=>(type_order<G> const &rhs) const noexcept
      -> std::strong_ordering;
};
```


2 Revision History

0. New Paper
1. Revision 1
   — Introduce `std::type_order` to prevent changing `std::type_info::before`
   — Anonymous namespaces can't be empty
   — Add section named scopes
   — Add FAQ section

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.

A consistent strong ordering of types would also allow such typesets to produce the same symbol in different compilation units, thus allowing consistent linking of such compilation units.

4 Proposal

This proposal only concerns itself with ordering types. However, it has implications for the whole reflection space as it is a subset of providing strong ordering on `std::meta::info` objects.

Below, we propose a canonical way of sorting all types in c++, which allows us to mark `std::type_info::before` `constexpr`.

This proposal does not propose marking type successor access as `constexpr` (i.e. `typeid(int).next()`), as the result of that is by necessity compilation-unit specific.

4.1 Approach outline

1. We define a **lowering to a *key-tuple*** for every type in the language.

2. The order is then defined on these *key-tuples*.

4.2 Structure of key-tuples

Every *key-tuple* is of the form `($_element_$...)`.

where an element is one of:

- *atom* (see atoms)
- *key-tuple*

These tuples are then ordered lexicographically (ties broken in favor of shorter tuple), atoms before tuples.

Let us name this transformation as `sort_key(entity)`.

The rest of the paper is concerned with defining this transformation.

4.3 Named Scopes

A type is ordered by appending `sort_key(...)` to the named scope it is declared in. The following are *named scopes*:

1. namespaces
2. classes
3. functions
4. lambdas
5. concepts

Starting with the global namespace, `sort_key(global)=()`. Any type `T` declared in the global namespace shall have a defined `sort_key` operation that resolves to `(sort_key(T))`.

If `class foo` is declared in `struct bar`:

```
struct bar { class foo; }
```

```
sort_key(foo) = (sort_key(bar), sort_key(foo)) = ((type, bar), (type, foo, ))
```

This shall hold for any of the above named scopes.

4.3.1 Example:

Given

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

namespace baz {
  struct j;
}
```

Then:

- `sort_key(foo::bar::i)` is `((namespace, foo), (namespace, bar), (type, i, ))`.
- `sort_key(baz::j)` is `((namespace, baz), (type, j, ))`

When compared, the result is that `baz::j` < `foo::bar::i`, since `namespace baz` precedes `namespace foo`.

4.4 Atoms

The atoms of *key-tuples* are ordered as follows:

1. kinds (see kinds)
2. simple names (including empty string) (see names)
3. qualifiers (see qualifiers)
4. `[]` (array of unknown bound)
5. `[n]` (array of known bound n) (ordered by `n` internally)
6. `*` (pointer)
7. ellipsis (`...` in `f(...)`)
8. parameter pack (`...` in `typename...`)

4.4.1 Kinds

There are the following kind tokens that can appear in *key-tuples*.

1. value
2. namespace
3. type
4. class template
5. type alias template
6. variable template
7. concept
8. function

Note: everything but values is pretty simple, but we haven't dealt with values extensively yet with the R0 of this paper.

4.5 Identifiers

4.5.1 Simple Names

Most names are strings that are valid (atomic) identifiers. Those are just themselves:

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

`foo`, `bar` and `baz` are such atomic identifiers.

4.5.2 Anonymous Namespace

Anonymous namespaces shall be represented with the ! character, as it cannot be represented by the empty string and cannot collide with any user defined names;

Example:

```
namespace a { namespace { struct s; } }

sort_key(a::s) = ((namespace, a), (namespace, "!"), (type, s, ))
```

4.5.3 Unnamed entities

Unnamed entities are all given a name that is not an identifier (but is, in fact, a tuple), and are then numbered consecutively, starting with zero, based on their name-scope.

Name-scopes are namespaces, classes, unions, functions, and enumerations.

Function declarations are name-scoped to the function itself.

Consider a lambda that appears as a default argument of a function template:

```
template <typename T>
void f(T x = []{ return T{0}; }());
//           ^^^^^^^^^^^^^^^^^^ this one
```

The *key-tuple* for `f<int>` is:

`(function, (f, (type, int)), (type, void), ((type, int)))`

The *key-tuple* for the lambda is:

`((function, (f, (type, int)), (type, void), ((type, int))), (type, (lambda, 0), ))`.

Note: because of the regular structure of *key-tuples*, such anonymous classes will compare greater than any entity that has a simple identifier, due to tuples comparing greater than atoms (which simple names are).

4.5.3.1 Lambda types

Types of lambda objects are ordered first by where they are declared, then by declaration order.

In effect, we assign them the name `(lambda, #)` where `#` is the count of other unnamed entities in the name scope.

```
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:

```
sort_key(decltype(Banana::i)) = ((namespace, Banana), (type, (lambda, 0), ));
sort_key(decltype(Apple::i))  = ((namespace, Apple), (type, (lambda, 0), ));
sort_key(decltype(Apple::j))  = ((namespace, Apple), (type, (lambda, 1), ));
```

Note: the empty bit after the identifier is the empty qualifier pack.

4.5.3.2 Unnamed `struct` and `union` types

They are named, respectively, `(class, #)` and `(union, #)`.

4.6 Namespaces

The `sort_key(namespace-name)` is `(namespace, identifier)`.

This means that namespaces are ordered alphabetically by comparing namespace names at the same rank. A namespace comes before any of its subnamespaces.

Example:

```
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

`sort_key(outer1::i) < sort_key(outer2::inner1::i) < sort_key(outer2::inner2::i)`.

4.7 Types

The `sort_key` of a type is `(type, <identifier>, <qualifiers>)`.

The `<identifier>` bit is a bit complicated, so let's deal with the qualifiers first.

Note: any name-scopes the `type` is declared in are part of the parent *key-tuple*. The `identifier` portion is complicated because of possible template arguments for types that are template specializations.

4.7.1 Qualifiers

Qualifiers are each assigned a score

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

and ordering lowest-first after summing them.

Therefore, for an 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.

4.8 Ordering Scalar Types

All scalar types are built-in types, except for enumerations, which should be ordered according to their namespaced names.

Unfortunately, some of the built-in types do not have names, only type aliases (such as `decltype(nullptr)`).

The intention is for built-in scalar types to be ordered before any compound types.

Built-in types with simple names should be ordered before any types that reference other types.

In particular, scalar types shall be ordered as follows:

1. `void` comes first because it's not reifiable,
2. the type of `std::nullptr_t` as the first monostate
3. any other monostates, if added, sorted alphabetically by their common names (to be specified explicitly if added)
4. `bool` as the first bi-state
5. any other bi-states, if added, sorted alphabetically.

6. Raw-memory types (`char`, `signed char`, `unsigned char`) (std::byte is an enumeration in `std` so it falls under different rules)
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 floating point types.
10. Function types (internally ordered by rules in section Function Types)
11. Pointer types (internally ordered by their pointee-type)
12. Pointer-to-member types (internally ordered by pointee-type)

Class types shall be ordered according to the rules below, see [Ordering Compound Types]

4.9 Ordering Array Types

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

The `sort_key(T[]) = ([], sort_key(T))` and the `sort_key(T[n]) = ([n], sort_key(T))`.

The intention is to 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]`, and

`sort_key(T[0]) = (type, ([], (type, T, )))`

`sort_key(T[10][2]) = (type, ([2], sort_key(T[10]))) = (type, ([2], (type, ([10], (type, T, ))))`

4.10 Ordering Class Types

4.10.1 Ordering Simple Class Types

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 sort key as:

`sort_key(Apple) = (type, Apple, )`

`sort_key(Banana) = (type, Banana, )`

`sort_key(Carrot) = (type, Carrot, )`

4.10.2 Non Type Template Parameters

NTTPs shall first be ordered by their type, then their value.

Given:

```
template <auto T>
struct s {
    decltype(T) i = T;
};

s<1u> a;
s<1.0f> b;
```

`sort_key(s<1u>) = ((type, (s, sort_key(1u))))`

We can define sort_key of `1u` as: `sort_key(1u) = ( sort_key(decltype(1u)), 1)`

`s<1u>` shall be ordered before `s<1.0f>`, as integral types come before floating point types.

NTTPs of the same type shall be lexicographically ordered by their scalar subobjects. Meaning

```
struct F final {
  struct G final {
    int h;
    int i;
  } g;
  int j;
};

F f{{0,1}, 2};
F f2{{1,2}, 3};
```

`sort_key(s<f>) < sort_key(s<f2>);`

NTTPs of the same pointer or reference type shall be ordered by instantiation order.

4.10.3 Ordering Class Template Specializations

Class templates shall be 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>;
```

Note, `sort_key(<parameter>)...` will be used to denote a tuple where `sort_key` has been applied to all parameters.

For `void f(Foo, Bar)` `sort_key(<parameter>)...` would mean `(sort_key(Foo), sort_key(Bar))`

`sort_key` of a class template shall be defined as:

```
sort_key(<class template>) = (type, (<name>, (sort_key(<parameter>)...)))
```

So

```
sort_key(Apple<Banana, Carrot> = (type, (Apple, (sort_key(Banana), sort_key(Carrot)), )
```

```
sort_key(Apple<Banana, Carrot> = (type, (Apple, ((type, Banana, ), (type, Carrot, )), )
```

Note: the empty bit after the identifier is the empty qualifier pack.

The above would be ordered sort_key(Apple<Banana, Banana>), sort_key(Apple<Banana, Carrot>), sort_key(Apple<Carrot, Carrot>.

4.10.4 Function Types

Function types shall be ordered by

1. Return type
2. Parameters, lexicographically.

The sort_key of a function shall be defined as:

```
sort_key(<function>) = (function, <name>, sort_key(<return type>), (sort_key(<parameter>)...))
```

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

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

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

```
sort_key(void foo(int)) = (function, foo, (type, void), ((type, int)))
```

```
sort_key(void foo(int, double)) = (function, foo, (type, void), ((type, int), (type, double)))
```

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

4.10.5 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.

The sort key of a member function shall be defined as:

```
sort_key(<member function>) =
```

```
(function, (<name>, sort_key(<class>)), sort_key(<return type>), (sort_key(<parameter>)...))))
```

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

```
sort_key(Foo::bar) =
```

```
(type, Foo, ), (function, (bar, (type, Foo, )), (type, void), ((type, int, ), (type, float, ))))
```

4.10.6 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, ...))`

4.10.7 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>`

4.10.8 Ordering Class Templates

Kinds of templates are ordered first by name, then by template arguments.

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:

`sort_key(zero<int>) = (type, (zero, (type, int)))`

`sort_key(one<zero>) = (type, (one, (class_template, zero))))`

`sort_key(two<one>) = (type, (two, (class_template, one))))`

4.10.9 Variable Templates

Variable templates are ordered by name, then by template parameter.

`sort_key(<variable_template>) = (variable_template, (<name>, (sort_key(<template_parameter>)...)))`

```
template <typename F, typename S>
constexpr std::pair<F, S> pair_one_two = {1, 2};
```

the type of `pair_one_two<int, double>` can be represented as:

`sort_key(pair_one_two<int, double>) = (variable_template, (pair_one_two, (type, int), (type, double)))`

4.10.10 Alias Templates

Alias templates are ordered alphabetically by name.

`sort_key(<alias_template>) = (alias_template, <name>)`

Given

```
template< class T >
using remove_cvref_t = typename remove_cvref<T>::type;
```

`sort_key(remove_cvref_t) = (alias_template, remove_cvref_t)`

4.10.11 Concepts

Concepts are ordered in a similar manner to variable templates.

`sort_key(<concept>) = (concept, (<name>, (sort_key(<template_parameter>)...)))`

```
template <typename T, typename F = decltype([](T){})>
concept f = requires (T i, F f = [](T){}) {
    {f(i)} -> std::convertible_to<void>;
};
```

In order to order the type of the lambda declared in `concept f`, `concept f` must be comparable with other types.

Concepts shall be ordered first by name, then by template arguments.

`sort_key(f<int>) = (concept, (f, (type, int), (lambda, 0)))`

5 FAQ

Why should this be standardized?

Currently, type_info::before only provides implementation defined type order. This means that two different compilation units could order a type differently. Doing this would allow portable typesets across compilers.

6 Acknowledgements

Thanks to all of the following:

— Davis Herring for his suggestions on ordering non-type template parameters.

7 References

[1] Jens Maurer. 2019. Inconsistencies with non-type template parameters. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1907r1.html