Standardized Type Ordering

Document #: P2830R1
Date: 2023-10-06
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. This paper explores a standardized ordering of types in C++, and possible syntaxes for it.

Defining `type_info::before` would consitute an ABI break, so this paper explores various other ways of exposing a standardized type ordering.

Therefore, we split this paper into two parts:

1. the design of the exposition-only function `ORDER(x, y)`, which deals with how such an order should be defined,
2. how to expose this capability to the programmer.

2 Revision History

0. New Paper
1. Revision 1
   — Introduce options to prevent changing `std::type_info::before`
   — Anonymous namespaces can't be empty
   — Add FAQ section
   — Add motivating examples
   — Add proposed syntax
   — Add appendix

3 Motivation

There is currently no way in portable C++ to sort types at compile-time.

Various libraries exist to do this, mostly by utilizing `__PRETTY_FUNCTION__`, but this is non-portable and error-prone (consider forward-declared enums). There are multiple stack-overflow questions on the topic.

Fundamentally, we need a way to canonicalize sets and multisets of types. This is necessary when building any kind of compositional library in C++, from `std::execution` [P2300R7], mixin libraries, libraries of monadic components and operators, policy-based libraries, etc. The inability to sort and unique types leads to the instantiation of an exponential number of templates instead of just one per typeset, which leads to code-bloat and untenable compile-times. The problem is fundamentally that we generate types with a different name but the same behavior, and there is no way to avoid this.

The goal here is to provide a *flexible* language mechanism to let both `Foo<A, B, C>` and `Foo<C, B, A>` produce the same underlying `Foo_impl<A, B, C>`.

The reason we start with `ORDER()` and not "just give me typesets" is that we need flexibility; consider

— a given library might want to deduplicate on a part and keep either last or first, or even make it ill-formed: `Foo<pair<A, X>, pair<B, Y>, pair<A, Z>>` might want to be the same as `Foo_impl<pair<A, X>, pair<B, Y>>` or `Foo_impl<pair<A, Z>, pair<B, Y>>`
— a given library might actually just want canonicalized multisets: `Foo<A, B, A, A, C>` should perhaps be `Foo<A, A, A, B, C>`
— or treat the first one as special: `Matrix<float, policy1, policy2, policy3>` should only deduplicate policies.

We must provide `ORDER` in order to `sort` and `unique`; they are required building blocks for any `set` primitive. Put another way, even if we standardized a `set`, we'd need to somehow canonicalize the order (due to mangling and debug info), leading us back here.

Without such canonicalization, it is also functionally impossible to enumerate the set of function templates to instantiate in a separate compilation unit. For the same reason, it's utterly impossible to type-erase them in a fixed-size virtual function table.

3.1 Motivating Examples

This section introduces the kind of code we would like to write, regardless of how this feature ends up being spelled. To not prejudice the reader in this section, please assume the existence of an exposition-only `consteval` function `ORDER(x, y) -> std::strong_ordering` whose arguments can be anything usable as a template argument. Crucially, also consider the interactions with [P1985R3] and [P2841R1], which introduce `concept` and `variable-template` arguments.

3.1.1 Canonicalizing policy-based libraries

Consider the needs of a library for type-erasure; we'd like the user to specify the capabilities to erase. Fundamentally, these capabilities are a set.

Observe:

```
using T1 = lib::dyn<ostreamable, json_serializable, iterable<int>, scalar_multiplicable<int>>;
```

Different users of the library will likely provide these capabilities in different orders; this becomes especially problematic when writing functions:

```
using T2 = lib::dyn<json_serializable, ostreamable, iterable<int>, scalar_multiplicable<int>>;
                    ~~~~~~ flipped ~~~~~~~~~~~~~~~~
void f(T2 const&);
```

We can solve this by having type aliases, but if these sets are *computed*, we are left without recourse:

```
int sum = 0;
auto pipeline1 =
   log(std::cerr) | sum_into(&sum) | imul(sum) | json_dump(std::cout);
auto pipeline2 =
   sum_into(&sum) | imul(sum) | json_dump(std::cout) | log(std::cerr);
```

The above pipelines need the same type-erased interface for its input, but will likely compute it as `T1` and `T2`, respectively.

3.1.2 Canonicalized variant

The most obvious example is a canonicalized `std::variant`, that is, something like

```
template <typename... Ts>
using canonical_variant = apply_canonicalized<std::variant, Ts...>;
```

Building `apply_canonicalized` is not terribly difficult, as long as we have `ORDER`. Please see the appendices on how to do it.

It *would* be nice if `apply_canonicalized` was a language built-in, but to do that, we need to first define `ORDER(x, y)`. After we define `ORDER`, putting `apply_canonicalized` into `type_traits` is a far simpler proposition.

Note: `apply_canonicalized` is roughly `mp11::mp_sort` + `mp11::unique` with the order derived from `ORDER`.

3.1.2.1 Uses of a canonicalized variant

Uses are legion; but they all follow same the pattern as the type from `std::execution` needed for the result between suspension points. We call it `notification`, as a nod to `rxcpp`.

Consider the `transfer(scheduler)` algorithm; it receives all `set_value(PACK)`, `set_error(PACK)` and possibly `set_stopped()` parameter lists, *stores them in a* `variant<tuple<CHANNEL, PACK>...>`, and resumes with the values on a different scheduler once the compute resource becomes available.

The code using it looks like this:

```
some_computation | transfer(scheduler) | then([](auto&&...args){/*...*/})...
```

There are many `transfer` calls in a given program, and odds are many of them deal with similar signals, but the order of the 4 or 5 different `set_value()` channels is unlikely to be the same in every pipeline. Moreover, many of these channels exhibit incast semantics - going from many types to few - and so without canonicalization, we are left with a variant with many more types than is necessary. We also generate far more code than required, since we generate switch statments for every ordering of channels, instead of every set.

3.1.3 Canonicalized tuple

Similarly to a canonicalized `variant`, a `canonicalized_tuple` can be implemented as

```
template <typename...Ts>
using canonical_tuple = apply_canonicalized<tuple<Ts...>>;
```

This is far less useful on its own than the `variant`, since initializing that type will be quite the chore. Instead, canonicalized tuples appear as a result of the `environment` monad, again, in implementations of `std::execution`.

`std::execution` mixes in an environment, which consists at least of a `stoppable_token`, a `scheduler`, and possibly other things like a `domain`, and, at least in the authors' implementation, arbitrary other things.

The `environment<pair<Tags, Ts>...>` is a product type that is a map from `Tag` to a value of type `T`, say `std::stop_token` to `never_stop_token{}` and `scheduler` to `thread_pool_scheduler`.

One can push and pop things off the environment in different parts of the pipeline; if the pushes come in the wrong order, the environment is difficult to keep canonicalized.

```
auto ss = std::stop_source();
auto tp = std::static_thread_pool(5);
auto env = empty_env{}
    | push_env<stop_token>(ss.get_token())
    | push_env<scheduler_t>(tp.get_scheduler())
    | push_env<mylib::numa_domain_t>(2);
```

If you imagine those three calls to happen in different algorithm transformers, it's quite likely they'll be in a different order, manifesting a different type, which doesn't make much sense - the type behaves identically, this just leads to code bloat.

3.1.4 ABI and API stability

The authors have observed a necessity to sometimes split parts of such pipeline compositions into different compilation units due to excessive compile times and repetitive compilation.

This has proven difficult due to the brittle nature of type ordering of the names of explicit template specializations involved. By far, the main culprit has been the lack of a sensible canonicalization of names, as the order changes far more frequently than the set of types.

Having a defined, stable order of types proved invaluable (even if we did hack the type order manually).

4 Proposal

4.1 `ORDER(x, y)`

Notionally, this proposal concerns itself with ordering types, but *one does not simply order types.* This *is* C++.

4.1.1 Design Principles

We wish to fulfil the following principles with this design; we are open to ones we might have missed.

4.1.1.1 Self-consistency

The ordering should be self-consistent, that is, for all possible template arguments `T`, `U`, and any unary template `some_template`:

`ORDER(T, U) == ORDER(some_template<T>, some_template<U>)`.

4.1.1.2 Readability

Types that look roughly the same should sort closely together. This is an aesthetic criterion, but sorting `T` close to `T&`, `T const&` and so on is probably a good idea, given that the user's main exposure to these types will be through typelists in error messages.

This criterion asks for sorting things in the same namespace close together, for instance.

4.1.1.3 Stability

The order should be the same across compilation units and across platforms. This is key for generating ABI-compatible vtables, for instance.

4.1.1.4 Reflection compatibility

`operator<=>(std::meta::info, std::meta::info)` should be consistent with this one.

While `std::meta::info` [P2320R0] objects can reflect more entities than just types and values (mainly: expressions), any ordering defined on them should be finer than this one, and specifically consistent with it. We should do something that reflection can subsume.

4.1.2 Approach

1. We define a **lowering to a *key-tuple*** for every entity in the language.
2. The order is then defined on these *key-tuples*.

4.1.3 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.

4.1.4 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))`.

4.1.4.1 Example 1: `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.

4.1.4.2 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`.

4.1.5 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...`)

4.1.6 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 R1 of this paper, though we should just defer to `<=>` and require a default strong structural ordering for values that may be template arguments.

4.1.7 Identifiers

4.1.7.1 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.

4.1.7.2 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, ))
```

4.1.7.3 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).

4.1.7.3.1 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.

4.1.7.3.2 Unnamed `struct` and `union` types

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

4.1.8 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)`.

4.1.9 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.

4.1.9.1 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.

4.1.10 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]

4.1.11 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, ))))`

4.1.12 Ordering Class Types

4.1.12.1 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, )`

4.1.12.2 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.

4.1.12.3 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>`.

4.1.12.4 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)`

4.1.12.5 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, ))))`

4.1.12.6 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, ...))`

4.1.12.7 Parameter Packs

Parameter are ordered as class templates.

Given:

```cpp
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>`

4.1.12.8 Ordering Class Templates

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

Given:

```cpp
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))))`

4.1.12.9 Variable Templates

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

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

```cpp
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)))`

4.1.12.10 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)`

4.1.12.11 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)))`

4.2 Syntax

The authors aren't too hung-up on the syntax. Any syntax will do for the use-case, but some better than others. This section explores the trade-offs in syntax choice.

Authors recommend one of the first 3 options.

4.2.1 Option 1: `constexpr bool std::type_info::before()`

Pros:

- doesn't require a new token, just add `constexpr` in front of `before`
- `std::type_info` is discoverable
- built-in type-erasure (you can sort them in `std::array<>` with consteval `std::sort`!)
  - going the other way is more difficult though, so doesn't quite pan out

Cons:

- breaks ABI
- still requires including `<typeinfo>`, which many organizations ban
- doesn't work for nontypes (contrast `entity_ordering<T, U>`)

4.2.2 Option 2: `constexpr std::strong_order(std::type_info, std::type_info)`

Pros:

- doesn't require a new token, we just need to define the overload.
- `std::type_info` is discoverable, but `std::strong_order()` of `type_info` a bit less-so
- built-in type-erasure, though only helps so much.

Cons:

- requires including `<typeinfo>`
- doesn't work for nontypes

4.2.3 Option 3: variable template `std::entity_ordering`

This option means to put

```
template <universal template T, universal template U>
inline constexpr std::strong_order entity_ordering = ORDER(T, U); /* see paper */
```

into either `<type_traits>` or `<compare>`, depending on what EWG/LEWG recommend.

Obviously, the name is bikesheddable.

Pros:

— completely new name, obviously available at compile-time
— no need to include `<typeinfo>`

Cons:

— new name
— not as discoverable (people look at `<typeinfo>`)

4.2.4 Option 4: `constexpr std::type_identity::operator<=>`

Pros:

— kinda obvious

Cons:

— adds `<compare>` to `<type_traits>`, since that's where `type_identity` is
— too cute?
— doesn't work for nontypes

4.2.5 Option 5: `constexpr std::__lift<arg>::operator<=>`

This option means we add `template <universal template> struct __lift {};` into `<type_traits>` and define `operator<=>` for it.

Pros:

— … we'll need a lift sooner or later?

Cons:

— really nonobvious
— still needs a new name
— needs a tutorial to find
— yet-another-`type_info`, basically

5 FAQ

5.1 Why should this be standardized?

`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.

5.2 Why not wait for reflection?

It's a good question. However, reflection will do *nothing* for this problem by itself; the user will still have to implement ordering using `consteval` functions, which have no hope of being as fast as a compiler-provided built-in.

User-programmed functions also won't adapt to language evolution; this feature will.

Finally, sorting is arbitrary; having it be consistent throughout the software ecosystem is potentially a great enabler of interoperability.

5.3 But couldn't this be done faster with reflection?

No; Peter Dimov shares an interesting anecdote.

I have in Mp11 the algorithm mp_unique, which takes a list of types and removes the duplicates. In the course of writing the reflection papers, their authors occasionally took Mp11 code examples and tried to show how they are elegantly implemented using value-based reflection metaprogramming.

So, you take a vector that contains types, and then you simply apply the existing algorithm std::unique to it, et voila… oh wait.

std::unique wants a sorted range, and you can't std::sort the info vector, because info objects aren't ordered, even when they refer to types.

6 Acknowledgements

Thanks to all of the following:

- Davis Herring for his suggestions on ordering non-type template parameters.
- Ville Voutilainen for his critique of examples, and providing a simple way of explaining the motivation
- Peter Dimov for a helpful anecdote, now in the FAQ.

7 Appendix A: building `apply_canonicalized`

We will need a small metaprogramming library; a filter is difficult to do otherwise.

```cpp
struct undefined;
template <typename... Ts> struct list {};

// apply<F, list<Ts...>> -> F<Ts...>
template <template <typename...> typename, typename> extern undefined _apply;
template <template <typename...> typename F, template <typename...> typename L,
          typename... Ts>
F<Ts...> _apply<F, L<Ts...>>;
template <template <typename...> typename F, typename List>
using apply = decltype(_apply<F, List>);

// concatenate<list<Ts...>, list<Us...>, list<Vs...>> -> list<Ts..., Us..., Vs...>
template <typename...> extern undefined _concatenate;
template <typename... Ts> list<Ts...> _concatenate<list<Ts...>>;
template <typename... Ts, typename... Us, typename... Lists>
decltype(_concatenate<list<Ts..., Us...>, Lists...>)
    _concatenate<list<Ts...>, list<Us...>, Lists...>;
template <typename... Ts>
using concatenate = decltype(_concatenate<Ts...>);

// select: list<T> if true, list<> if false
template <bool v, typename T> extern list<> _select;
template <typename T> list<T> _select<true, T>;
template <bool v, typename T>
using select = decltype(_select<v, T>);
```

Canonicalization is now just a basic not-in-place quicksort-ish thing:

```
template <typename.../*empty*/> extern list<> _canon;
template <typename... Ts>
using canonicalized = decltype(_canon<Ts...>);

// a canonicalized T is just T
template <typename T>
list<T> _canon<T>;

template <typename T, typename... Ts>
concatenate<
    // canonicalized things less than T
    apply<canonicalized, concatenate<select<(ORDER(Ts, T) < 0), Ts>...>>,
    list<T> /*T*/, //                          ~~~~~~~~~~~~~~~~
    // canonicalized things greater than T
    apply<canonicalized, concatenate<select<(ORDER(Ts, T) > 0), Ts>... >>
    > //                                       ~~~~~~~~~~~~~~~~
_canon<T, Ts...>;
```

We now have `canonicalized<Ts...>` - but this still leaves `list` as a special type which we'd rather not expose to the user. Onto `apply_canonicalized`:

```
template <template <typename...> typename F, typename... Ts>
using apply_canonicalized = apply<F, canonicalized<Ts...>>;
```

7.1 Full code listing as tested and implemented

Here for completeness, feel free to skip.

```
#include <compare>
#include <type_traits>

struct undefined;

#define ORDER(x, y) type_order_v<x, y>

// in <type_traits>
template <typename X, typename Y>
constexpr inline std::strong_ordering type_order_v;

template <template <typename...> typename, typename>
extern undefined _apply;

template <template <typename...> typename F, template <typename...> typename L,
          typename... Ts>
F<Ts...> _apply<F, L<Ts...>>;

template <template <typename...> typename F, typename List>
using apply = decltype(_apply<F, List>);

// some user-type
template <auto x>
```

```
struct value_t : std::integral_constant<decltype(x), x> {};
template <auto x>
inline constexpr value_t<x> value_v{};

// built-in
template <auto x, auto y>
constexpr inline std::strong_ordering type_order_v<value_t<x>, value_t<y>> =
    x <=> y;

template <typename... Ts>
struct list {};

template <typename...>
extern undefined _concatenate;
template <typename... Ts>
list<Ts...> _concatenate<list<Ts...>>;
template <typename... Ts, typename... Us, typename... Lists>
decltype(_concatenate<list<Ts..., Us...>, Lists...>)
    _concatenate<list<Ts...>, list<Us...>, Lists...>;

template <typename... Ts>
using concatenate = decltype(_concatenate<Ts...>);

template <bool v, typename T>
extern list<> _select;
template <typename T>
list<T> _select<true, T>;

template <bool v, typename T>
using select = decltype(_select<v, T>);

template <typename...>
extern list<> _canon;
template <typename... Ts>
using canonicalized = decltype(_canon<Ts...>);

template <typename T>
list<T> _canon<T>;

template <typename T, typename... Ts>
concatenate<
    apply<canonicalized, concatenate<select<(ORDER(Ts, T) < 0), Ts>...>>,
    list<T>,
    apply<canonicalized, concatenate<select<(ORDER(Ts, T) > 0), Ts>... >>
    >
_canon<T, Ts...>;


static_assert(std::same_as<canonicalized<value_t<0>, value_t<-1>, value_t<-1>, value_t<1>>, list<value_t<
```

8 References

[P1985R3] Gašper Ažman, Mateusz Pusz, Colin MacLean, Bengt Gustafsonn, Corentin Jabot. 2022-09-17. Universal template parameters.
https://wg21.link/p1985r3

[P2300R7] Eric Niebler, Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Bryce Adelstein Lelbach. 2023-04-21. ‘std::execution‘.
https://wg21.link/p2300r7

[P2320R0] Andrew Sutton, Wyatt Childers, Daveed Vandevoorde. 2021-02-15. The Syntax of Static Reflection.

https://wg21.link/p2320r0

[P2841R1] Corentin Jabot, Gašper Ažman. 2023-10-14. Concept Template Parameters.
https://wg21.link/p2841r1