Conditional `final` *class-virt-specifier*

ISO C++23 Standard — Feature Proposal

Metrics

Document №:	Draft R0
Date:	08.07.2020
Project:	ISO/IEC JTC 1/SC 22/WG 21/C++
Audience:	Core Working Group Library Working Group
Reply-to:	Paweł Benetkiewicz <isocpp@benio.me>

Contents

Proposal

Make `final` *class-virt-specifier* conditional.

Syntax

Current	Proposed		
`final`	**`final`**	(1)	
	`final` (*expression*)	(2)	(since C++23)
	`final` < *type-parameter-key name*(optional) > (*expression*)	(3)	(since C++23)

1) Same as **`final ( true )`**

2–3) if *expression* evaluates to true upon final conditioning, the program is ill-formed.

expression — Sequence of logical operations and operands that specifies negation of requirements on base class.

type-parameter-key — Either **typename** or **class**. There is no difference between these keywords in context of final conditioning declaration.

name — Temporary identifier corresponding to derived class upon final conditioning. It's optional to keep consistency with templates.

In case (3), every occurrence of *name* (if any) will be interpreted as a derived class reference in final conditioning. If *name* refers to another class, program is ill-formed.

In cases (2, 3), final conditioning will be performed in the context of deriving class, evaluating *expression*. If evaluation of final conditioning is `true`, program is ill-formed.

Motivation

Conditioning inheritance was widely requested for many years, but it was not possible to add such feature until C++20, which introduced constraints — *constraint-logical-or-expression* allows making `final` *class-virt-specifier* conditional, with optional help of a temporary identifier corresponding to derived class upon final conditioning — evaluation of constraint in the context of class declaration.

This proposal is very similar to P0892R2 (C++20) "explicit(bool)". Both proposals adds conditional expression to specifier with special meaning. Two main differences in syntax are:

1. `final` requires additional temporary replacement for a derived class.
2. `final` uses *constraint-logical-or-expression* while `explicit` uses *constant-expression*.

Conditional `final` *class-virt-specifier* shall heavily reduce amount of hard to deal with common problems with inheriting invalid, outdated or incompatible: interfaces, components and extensions.

This proposal adds feature to heavily increase **compatibility** correctness by inheritance constraints.

Examples

Conditional `final` *class-virt-specifier* allows *i.a.*:

1. Restricting inheritance of interface for it's implementation only.

```
class foo_impl {};
class foo_interface final<class B>(!same_as<B, foo_impl>) {};
// foo_interface can be inherited from foo_impl class only.
```

2. Restricting inheritance of component for compatible classes only.

```
class enemy_component final<class B>(!derived_from<B, enemy>) {};
// enemy_component can be inherited from classes that inherits from enemy only.
```

3. Ensure that version of a derived class is not less than version of base class.

```
class extension final<class B>(B::api_version < extension::api_version) {};
// extension can be inherited from base class with not less api version only.
```

Requirements

1. **Library**: Small change needs to be done to `is_final` type trait, to accept either one or two template arguments, as specified in *Wording: § 7 – 9* of this proposal (§ 20.15.2 [meta.type.synop], § 20.15.4.3: Table 49 [tab:meta.unary.prop], § 20.15.6.2: Table 51 [tab:meta.rel]).

2. **Compiler**: Compilers will have to implement final conditioning procedure: up to one temporary template argument that refers to a derived class, and a constraint expression evaluation in the context of derived class declaration. Fortunately, this should be trivial up to very easy since we got concepts in C++20.

Disadvantages

None.

Advantages

1. **Similarity**: Proposed syntax for conditional `final` *class-virt-specifier* combines syntaxes of well–known `template` declaration and two other specifiers: `noexcept` and `explicit`:

   conditional `final` *class-virt-specifier*:

   `final` < `class` B > (*expression*)

   `template` declaration:

   `template` < `class` T >

   other specifiers with special meanings:

   `noexcept` (*expression*)
   `explicit` (*expression*)

2. **Design tradeoffs**: None.

3. **Compatibility**: Legacy code will be fully compatible with a proposed syntax. In order to keep backwards compatibility of modern code with syntax from before this proposal, we can take use from `__cpp_lib_is_final` support macro and define a new, simply one:

```
#if defined(__cpp_lib_is_final) && __cpp_lib_is_final >= 202007L
   #define FINAL(...) final(__VA_ARGS__)
#else
   #define FINAL(...)
#endif
```

Wording

1. In § 11.1.1 [class.pre] and A.8 [gram.class], add conditional `final`:

   class-virt-specifier:
   `final` *final-clause*$_{opt}$
   final-clause:
   final-head$_{opt}$ (*constraint-logical-or-expression*)
   final-head:
   < *type-parameter-key identifier*$_{opt}$ >

2. In § 11.1.5 [class.pre], change first paragraph:

   If a class is marked with the *class-virt-specifier* that begins with `final` *identifier* and it appears as a *class-or-decltype* in a *base-clause*, then the program is ill-formed unconditionally if there is no *final-clause* in *class-virt-specifier* of this class, or a final conditioning will be issued [*Note:* See below. — *end note*], yielding program ill-formed if constraint fails. Whenever a *class-key* is followed by a *class-head-name*, the *identifier* `final` (optionally followed by a *final-clause*), and a colon or left brace, `final` is interpreted as beginning of a *class-virt-specifier*.

3. Change numbering of sections § 11.1.6–7 to § 11.1.7–8 [class.pre].
   After § 11.1.5 [class.pre], add new section (§ 11.1.6) with the following content:

   If *final-head* with *identifier* appears in a *final-clause*, then every occurrence of *identifier* must be interpreted as a derived class reference in final conditioning. If such *identifier* already refers to a class, the program is ill-formed. Otherwise, a final conditioning will be performed, evaluating *constraint-logical-or-expression* of issuing *final-clause* and yielding result of evaluated constraint. [*Note:* Program is ill-formed if evaluation of final conditioning is `true`. — *end note*]

4. In § 11.1.5 [class.pre], add two line breaks and the following code to the end of an example:

```
struct D final<class B>(!derived_from<B, E>) {};
// struct D can be derived only from classes that derives from E

struct E {};
struct F : E {};
struct G {};

struct H : F, D {};       // OK: F derives from E
struct I : G, D {};       // ill-formed: G does not derive from E
```

5. In § 16.5.5.12.4 [derivation], change first sentence:

   All types specified in the C++ standard library shall not be unconditionally non-`final` types unless otherwise specified.

6. In § 17.3.2 [version.syn], change line with __cpp_lib_is_final:

```
#define __cpp_lib_is_final                          201420027L // also in <type_traits>
```

7. In § 20.15.2 [meta.type.synop], change `template is_final_v`:

```
template<class...>
  inline constexpr bool is_final_v;
template<class T>
  inline constexpr bool is_final_v<T> = is_final<T>::value;
template<class Base, class Derived>
  inline constexpr bool is_final_v<Base, Derived> = is_final<Base, Derived>::value;
```

8. In § 20.15.4.3 [meta.unary.prop], modify Table 49 [tab:meta.unary.prop], changing first sentence in *Condition* colum of row with `template<class T> struct is_final` *Template*:

   T is a class type marked with the *class-virt-specifier* that begins with `final` ([class.pre]). [*Note:* `is_final` has specializations for both unary and relation types. — *end note*]

9. In § 20.15.6.2 [meta.rel], add a row to Table 51 [tab:meta.rel]:

Template	Condition	Comments
`template<class Base, class Derived> struct is_final<Base, Derived>;`	`Derived` cannot derive from `Base` ([class.pre]). [*Note:* `is_final` has specializations for both unary and relation types. — *end note*]	Every template argument that is a class type, shall be a complete type.