Document Number: N4856
Date: 2020-03-02
Revises: N4818
Reply to: David Sankel
dsankel@bloomberg.net

C++ Extensions for Reflection

Contents

1 General [intro]

1.1 Scope [intro.scope]

1 This document describes extensions to the C++ Programming Language (Clause 1.2) that enable operations on source code. These extensions include new syntactic forms and modifications to existing language semantics, as well as changes and additions to the existing library facilities.

2 The International Standard, ISO/IEC 14882, together with the C++ Extensions for Concepts, ISO/IEC TS 19217:2015 provide important context and specification for this document. This document is written as a set of changes against the specification of ISO/IEC 14882, as modified by C++ Extensions for Concepts, ISO/IEC TS 19217:2015. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from the International Standard use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

1.2 Normative references [intro.refs]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2014, *Programming Languages — C++*

(1.2) — ISO/IEC TS 19217:2015, *Programming Languages — C++ Extensions for Concepts*

2 ISO/IEC 14882 is hereafter called the *C++ Standard*. ISO/IEC TS 19217:2015 is hereafter called the *Concepts-TS*.

3 The numbering of clauses, subclauses, and paragraphs in this document reflects the numbering in the C++ Standard as modified by the Concepts-TS. References to clauses and subclauses not appearing in this document refer to the original unmodified text in the C++ Standard.

1.3 Terms and definitions [intro.defs]

1 No terms and definitions are listed in this document. ISO and IEC maintain terminological databases for use in standardization at the following addresses:

(1.1) — IEC Electropedia: available at http://www.electropedia.org/

(1.2) — ISO Online browsing platform: available at http://www.iso.org/obp

1.4 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are those defined in subclause 1.4 in the C++ Standard. Similarly, all references to the C++ Standard in the resulting document shall be taken as referring to the resulting document itself. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

1.5 Namespaces and headers [intro.namespaces]

1 Whenever a name `x` declared in subclause 18.11 at namespace scope is mentioned, the name `x` is assumed to be fully qualified as `::std::experimental::reflect::v1::x`, unless otherwise specified. The header described in this specification (see Table 1) shall import the contents of `::std::experimental::reflect::v1` into `::std::experimental::reflect` as if by:

```
namespace std::experimental::reflect {
  inline namespace v1 {}
}
```

2 Whenever a name `x` declared in the standard library at namespace scope is mentioned, the name `x` is assumed to be fully qualified as `::std::x`, unless otherwise specified.

Table 1 — Reflection library headers

`<experimental/reflect>`

1.6 Feature-testing recommendations [intro.features]

1 An implementation that provides support for this Technical Specification shall define each feature test macro defined in Table 2 if no associated headers are indicated for that macro, and if associated headers are indicated for a macro, that macro is defined after inclusion of one of the corresponding headers specified in the table.

Table 2 — Feature-test macros

Macro name	Value	Header
`__cpp_reflection`	201902	none
`__cpp_lib_reflection`	201902	`<experimental/reflect>`

1.7 Acknowledgements [intro.ack]

1 This work is the result of a collaboration of researchers in industry and academia. We wish to thank the original authors of this TS, Matúš Chochlík, Axel Naumann, and David Sankel. We also wish to thank people who made valuable contributions within and outside these groups, including Ricardo Fabiano de Andrade, Roland Bock, Chandler Carruth, Jackie Kay, A. Joël Lamotte, Jens Maurer, and many others not named here who contributed to the discussion.

2 Lexical conventions [lex]

2.12 Keywords [lex.key]

1 In C++ [lex.key], add the keyword `reflexpr` to the list of keywords in Table 4.

3 Basic concepts [basic]

1 In C++ [basic], add the following last paragraph:

An *alias* is a name introduced by a `typedef` declaration, an *alias-declaration*, or a *using-declaration*.

3.2 One definition rule [basic.def.odr]

1 In C++ [basic.def.odr], insert a new paragraph after the existing paragraph 3:

A function or variable of static storage duration reflected by `T` (7.1.6.5) is odr-used by the specialization `std::experimental::reflect::get_pointer<T>` (18.11.4.9, 18.11.4.17), as if by taking the address of an *id-expression* nominating the function or variable.

2 In C++ [basic.def.odr], apply the following changes to the second bullet within paragraph 6:

and the object has the same value in all definitions of D, or a type implementing `std::experimental::reflect::Object` (18.11.3.1), as long as all operations (18.11.4) on this type yield the same constant expression results; and

3.9 Types [basic.types]

3.9.1 Fundamental types [basic.fundamental]

1 In C++ [basic.fundamental], apply the following change to paragraph 9:

An expression of type `void` shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of `?:` (5.16), as the operand of `typeid`, `noexcept`, `reflexpr`, or `decltype`, as the expression in a `return` statement (6.6.3) for a function with the return type `void`, or as the operand of an explicit conversion to type *cv* `void`.

4 Standard conversions [conv]

No changes are made to Clause 4 of the C++ Standard.

5 Expressions [expr]

5.1 Primary expressions [expr.prim]

5.1.2 Lambda expressions [expr.prim.lambda]

1 In C++ [expr.prim.lambda], apply the following change to the second bullet in paragraph 12:

— names the entity in a potentially-evaluated expression (3.2) where the enclosing full-expression depends on a generic lambda parameter declared within the reaching scope of the *lambda-expression*~~.~~;

where, for the process of this determination, `reflexpr` operands are not considered to be unevaluated operands.

2 Also apply the following change to paragraph 18:

Every *id-expression* within the *compound-statement* of a *lambda-expression* that is an odr-use (3.2) of an entity captured by copy, as well as every use of an entity captured by copy in a *reflexpr-operand*, is transformed into an access to the corresponding unnamed data member of the closure type.

5.2 Postfix expressions [expr.post]

1 In C++ [expr.post], apply the following change:

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `[` *braced-init-list* `]`
~~*postfix-expression* `(` *expression-list*$_{opt}$ `)`~~
function-call-expression
~~*simple-type-specifier* `(` *expression-list*$_{opt}$ `)`~~
~~*typename-specifier* `(` *expression-list*$_{opt}$ `)`~~
~~*simple-type-specifier braced-init-list*~~
~~*typename-specifier braced-init-list*~~
functional-type-conv-expression
postfix-expression `.` `template`$_{opt}$ *id-expression*
postfix-expression `->` `template`$_{opt}$ *id-expression*
postfix-expression `.` *pseudo-destructor-name*
postfix-expression `->` *pseudo-destructor-name*
postfix-expression `++`
postfix-expression `--`
`dynamic_cast` `<` *type-id* `>` `(` *expression* `)`
`static_cast` `<` *type-id* `>` `(` *expression* `)`
`reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
`const_cast` `<` *type-id* `>` `(` *expression* `)`
`typeid` `(` *expression* `)`
`typeid` `(` *type-id* `)`

function-call-expression:
postfix-expression `(` *expression-list*$_{opt}$ `)`

functional-type-conv-expression:
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
typename-specifier `(` *expression-list*$_{opt}$ `)`
simple-type-specifier braced-init-list
typename-specifier braced-init-list

expression-list:
initializer-list

6 Statements [stmt.stmt]

No changes are made to Clause 6 of the C++ Standard.

7 Declarations [dcl.dcl]

7.1 Specifiers [dcl.spec]

7.1.6 Type specifiers [dcl.type]

7.1.6.2 Simple type specifiers [dcl.type.simple]

1 In C++ [dcl.type.simple], apply the following change:

The simple type specifiers are

simple-type-specifier:
nested-name-specifier$_{opt}$ *type-name*
nested-name-specifier `template` *simple-template-id*
`char`
`char16_t`
`char32_t`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`
`auto`
decltype-specifier
reflexpr-specifier

type-name:
class-name
enum-name
typedef-name
simple-template-id

decltype-specifier:
`decltype (` *expression* `)`
`decltype ( auto )`

reflexpr-specifier:
`reflexpr (` *reflexpr-operand* `)`

reflexpr-operand:
`::`
type-id
nested-name-specifier$_{opt}$ *namespace-name*
id-expression
`(` *expression* `)`
function-call-expression
functional-type-conv-expression

...

The other *simple-type-specifiers* specify either a previously-declared type, a type determined from an expression, a reflection meta-object type (7.1.6.5), or one of the fundamental types (3.9.1).

2 Append the following row to Table 10:

`reflexpr` (*reflexpr-operand*)	the type as defined below

3 At the end of 7.1.6.2, insert the following paragraph:

For a *reflexpr-operand* `x`, the type denoted by `reflexpr(x)` is a type that satisfies the constraints laid out in 7.1.6.5.

7.1.6.5 Reflection type specifiers [dcl.type.reflexpr]

Insert the following subclause:

1 The *reflexpr-specifier* yields a type `T` that allows inspection of some properties of its operand through type traits or type transformations on `T` (18.11.4). The operand to the *reflexpr-specifier* shall be a type, namespace, enumerator, variable, data member, function parameter, captured entity, parenthesized expression, *function-call-expression* or *functional-type-conv-expression*. Any such `T` satisfies the requirements of `reflect::Object` (18.11.3) and other `reflect` concepts, depending on the operand. A type satisfying the requirements of `reflect::Object` is called a *meta-object type*. A meta-object type is an unnamed, incomplete namespace-scope class type (Clause 9).

2 An entity or alias `B` is *reflection-related* to an entity or alias `A` if

(2.1) — `A` and `B` are the same entity or alias,

(2.2) — `A` is a variable or enumerator and `B` is the type of `A`,

(2.3) — `A` is an enumeration and `B` is the underlying type of `A`,

(2.4) — `A` is a class and `B` is a member or base class of `A`,

(2.5) — `A` is a non-template alias that designates the entity `B`,

(2.6) — `A` is not the global namespace and `B` is an enclosing class or namespace of `A`,

(2.7) — `A` is the parenthesized expression `( B )`,

(2.8) — `A` is a lambda capture of the closure type `B`,

(2.9) — `A` is the closure type of the lambda capture `B`,

(2.10) — `B` is the type specified by the *functional-type-conv-expression* `A`,

(2.11) — `B` is the function selected by overload resolution for a *function-call-expression* `A`,

(2.12) — `B` is the return type, a parameter type, or function type of the function `A`, or

(2.13) — `B` is reflection-related to an entity or alias `X` and `X` is reflection-related to `A`.

3 [*Note:* This relationship is reflexive and transitive, but not symmetric. —*end note*]

4 [*Example:*

```
struct X;
struct B  {
   using X = ::X;
   typedef X Y;
};
struct D : B {
   using B::Y;
};
```

5 The alias `D::Y` is reflection-related to `::X`, but not to `B::Y` or `B::X`. —*end example*]

6 Zero or more successive applications of type transformations that yield meta-object types (18.11.4) to the type denoted by a *reflexpr-specifier* enable inspection of entities and aliases that are reflection-related to the operand; such a meta-object type is said to *reflect* the respective reflection-related entity or alias.

7 [*Example:*


```
template <typename T> std::string get_type_name() {
   namespace reflect = std::experimental::reflect;
   // T_t is an Alias reflecting T:
   using T_t = reflexpr(T);
   // aliased_T_t is a Type reflecting the type for which T is a synonym:
   using aliased_T_t = reflect::get_aliased_t<T_t>;
   return reflect::get_name_v<aliased_T_t>;
}

std::cout << get_type_name<std::string>(); // outputs basic_string
```

—end example]

8 It is unspecified whether repeatedly applying `reflexpr` to the same operand yields the same type or a different type. [*Note:* If a meta-object type reflects an incomplete class type, certain type transformations (18.11.4) cannot be applied. *—end note*]

9 [*Example:*

```
class X;
using X1_m = reflexpr(X);
class X {};
using X2_m = reflexpr(X);
using X_bases_1 = std::experimental::reflect::get_base_classes_t<X1_m>; // OK:
                                                    // X1_m reflects complete class X
using X_bases_2 = std::experimental::reflect::get_base_classes_t<X2_m>; // OK
std::experimental::reflect::get_reflected_type_t<X1_m> x; // OK: type X is complete
```

—end example]

10 For the operand `::`, the type specified by the *reflexpr-specifier* satisfies `reflect::GlobalScope`. Otherwise, the type specified by the *reflexpr-specifier* satisfies concepts depending on the result of name lookup, as shown in Table 11. Any other *reflexpr-operand* renders the program ill-formed.

11 If the *reflexpr-operand* of the form *id-expression* is a constant expression, the type specified by the *reflexpr-specifier* also satisfies `reflect::Constant`.

12 If the *reflexpr-operand* designates a name whose declaration is enclosed in a block scope (3.3.3) and the named entity is neither captured nor a function parameter, the program is ill-formed. If the *reflexpr-operand* designates a class member, the type represented by the *reflexpr-specifier* also satisfies `reflect::RecordMember`. If the *reflexpr-operand* designates an expression, it is an unevaluated operand (Clause 5). If the *reflexpr-operand* designates both an alias and a class name, the type represented by the *reflexpr-specifier* reflects the alias and satisfies `reflect::Alias`.

Table 11 — `reflect` concept (18.11.3) that the type specified by a *reflexpr-specifier* satisfies, for a given *reflexpr-operand*.

Category	***reflexpr-operand***	**`reflect` Concept**
Type	*class-name* designating a union	`reflect::Record`
	class-name designating a closure type	`reflect::Lambda`
	class-name designating a non-union class	`reflect::Class`
	enum-name	`reflect::Enum`
	template *type-parameter*	both `reflect::Type` and `reflect::Alias`
	decltype-specifier	both `reflect::Type` and `reflect::Alias`
	type-name introduced by a *using-declaration*	`reflect::Type`, `reflect::Alias`, and `reflect::ScopeMember`
	any other *typedef-name*	both `reflect::Type` and `reflect::Alias`
	any other *type-id*	`reflect::Type`
Namespace	*namespace-alias*	both `reflect::Namespace` and `reflect::Alias`
	any other *namespace-name*	`reflect::Namespace`
Expression	the name of a data member	`reflect::Variable`
	the name of a variable	`reflect::Variable`
	the name of an enumerator	`reflect::Enumerator`
	the name of a function parameter	`reflect::FunctionParameter`
	the name of a captured entity	`reflect::LambdaCapture`
	parenthesized expression[a]	`reflect::ParenthesizedExpression`
	function-call-expression[b]	`reflect::FunctionCallExpression`
	functional-type-conv-expression[c]	`reflect::FunctionalTypeConversion`

[a] For a *reflexpr-operand* that is a parenthesized expression `(E)`, `E` shall be a *function-call-expression*, *functional-type-conv-expression*, or an expression `(E′)` that satisfies the requirements for being a *reflexpr-operand*.

[b] If the *postfix-expression* of the *function-call-expression* is of class type, the function call shall not resolve to a surrogate call function (13.3.1.1.2). Otherwise, the *postfix-expression* shall name a function that is the unique result of overload resolution.

[c] The usual disambiguation between function-style cast and a *type-id* (8.2) applies. [*Example:* The default constructor of class `X` can be reflected on as `reflexpr((X()))`, while `reflexpr(X())` reflects the type of a function returning `X`. — *end example*]

8 Declarators [dcl.decl]

8.1 Type names [dcl.name]

1 In C++ [dcl.name], apply the following changes:

> To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `new`, ~~`or`~~ `typeid`, or `reflexpr`, the name of a type shall be specified.

9 Classes [class]

No changes are made to Clause 9 of the C++ Standard.

10 Derived classes [class.derived]

No changes are made to Clause 10 of the C++ Standard.

11 Member access control [class.access]

No changes are made to Clause 11 of the C++ Standard.

12 Special member functions [special]

No changes are made to Clause 12 of the C++ Standard.

13 Overloading [over]

No changes are made to Clause 13 of the C++ Standard.

14 Templates [temp]

14.6 Name resolution [temp.res]

14.6.2 Dependent names [temp.dep]

14.6.2.1 Dependent types [temp.dep.type]

1 In C++ [temp.dep.type], apply the following changes to paragraph 8:

A type is dependent if it is
[...]

(8.8) — denoted by `decltype(`*expression*`)`, where *expression* is type-dependent (14.6.2.2)~~.~~, or

(8.9) — denoted by `reflexpr(`*operand*`)`, where *operand* is a type-dependent expression or a (possibly parenthesized) *functional-type-conv-expression* with at least one type-dependent immediate subexpression, or

(8.10) — denoted by `reflexpr(`*operand*`)`, where *operand* designates a dependent type or a member of an unknown specialization or a value-dependent constant expression.

15 Exception handling [except]

No changes are made to Clause 15 of the C++ Standard.

16 Preprocessing directives [cpp]

No changes are made to Clause 16 of the C++ Standard.

17 Library introduction [library]

17.6 Library-wide requirements [requirements]

17.6.1 Library contents and organization [organization]

17.6.1.2 Headers [headers]

1 Add `<experimental/reflect>` to Table 14 – C++ library headers.

17.6.1.3 Freestanding implementations [compliance]

Modify Table 16 as follows.

Table 16 — C++ headers for freestanding implementations

Subclause		Header(s)
		`<ciso646>`
18.2	Types	`<cstddef>`
18.3	Implementation properties	`<cfloat>` `<limits>` `<climits>`
18.4	Integer types	`<cstdint>`
18.5	Start and termination	`<cstdlib>`
18.6	Dynamic memory management	`<new>`
18.7	Type identification	`<typeinfo>`
18.8	Exception handling	`<exception>`
18.9	Initializer lists	`<initializer_list>`
18.10	Other runtime support	`<cstdalign>` `<cstdarg>` `<cstdbool>`
18.11	Static reflection	`<experimental/reflect>`
20.10	Type traits	`<type_traits>`
20	Atomics	`<atomic>`

18 Language support library [language.support]

1 Add a new subclause 18.11 titled "**Static reflection**" as follows:

18.11 Static reflection [reflect]

18.11.1 In general [reflect.general]

1 As laid out in 7.1.6.5, compile-time constant metadata, describing various aspects of a program (static reflection data), can be accessed through meta-object types. The actual metadata is obtained by instantiating templates constituting the interface of the meta-object types. These templates are collectively referred to as *meta-object operations*.

2 Meta-object types satisfy different concepts (18.11.3) depending on the type they reflect (7.1.6.5). These concepts can also be used for meta-object type classification. They form a generalization-specialization hierarchy, with `reflect::Object` being the common generalization for all meta-object types. Unary operations and type transformations used to query static reflection data associated with these concepts are described in 18.11.4.

18.11.2 Header `<experimental/reflect>` synopsis [reflect.synopsis]

```
namespace std {
namespace experimental {
namespace reflect {
inline namespace v1 {

// 18.11.3 Concepts for meta-object types
template <class T>
concept bool Object = see below;
template <class T>
concept bool ObjectSequence = see below; // refines Object
template <class T>
concept bool TemplateParameterScope = see below; // refines Scope
template <class T>
concept bool Named = see below;           // refines Object
template <class T>
concept bool Alias = see below;           // refines Named and ScopeMember
template <class T>
concept bool RecordMember = see below;    // refines ScopeMember
template <class T>
concept bool Enumerator = see below;      // refines Constant
template <class T>
concept bool Variable = see below;        // refines Typed and ScopeMember
template <class T>
concept bool ScopeMember = see below;     // refines Named
template <class T>
concept bool Typed = see below;           // refines Object
template <class T>
concept bool Namespace = see below;       // refines Named and Scope
template <class T>
concept bool GlobalScope = see below;     // refines Namespace
template <class T>
```



```
concept bool Class = see below;           // refines Record
template <class T>
concept bool Enum = see below;            // refines Type, Scope, and ScopeMember
template <class T>
concept bool Record = see below;          // refines Type, Scope, and ScopeMember
template <class T>
concept bool Scope = see below;           // refines Object
template <class T>
concept bool Type = see below;            // refines Named
template <class T>
concept bool Constant = see below;        // refines Typed and ScopeMember
template <class T>
concept bool Base = see below;            // refines Object
template <class T>
concept bool FunctionParameter = see below; // refines Typed and ScopeMember
template <class T>
concept bool Callable = see below;        // refines Scope and ScopeMember
template <class T>
concept bool Expression = see below;      // refines Object
template <class T>
concept bool ParenthesizedExpression = see below;  // refines Expression
template <class T>
concept bool FunctionCallExpression = see below;   // refines Expression
template <class T>
concept bool FunctionalTypeConversion = see below; // refines Expression
template <class T>
concept bool Function = see below;        // refines Typed and Callable
template <class T>
concept bool MemberFunction = see below; // refines RecordMember and Function
template <class T>
concept bool SpecialMemberFunction = see below; // refines RecordMember
template <class T>
concept bool Constructor = see below;     // refines Callable and RecordMember
template <class T>
concept bool Destructor = see below;      // refines Callable and SpecialMemberFunction
template <class T>
concept bool Operator = see below;        // refines Function
template <class T>
concept bool ConversionOperator = see below; // refines MemberFunction and Operator
template <class T>
concept bool Lambda = see below;          // refines Type and Scope
template <class T>
concept bool LambdaCapture = see below;   // refines Variable


// 18.11.4 Meta-object operations
// Multi-concept operations
template <Object T> struct is_public;
template <Object T> struct is_protected;
template <Object T> struct is_private;
template <Object T> struct is_constexpr;
template <Object T> struct is_static;
template <Object T> struct is_final;
template <Object T> struct is_explicit;
template <Object T> struct is_inline;
```



```
template <Object T> struct is_virtual;
template <Object T> struct is_pure_virtual;
template <Object T> struct get_pointer;

template <class T>
requires RecordMember<T> || Base<T>
  constexpr auto is_public_v = is_public<T>::value;
template <class T>
requires RecordMember<T> || Base<T>
  constexpr auto is_protected_v = is_protected<T>::value;
template <class T>
requires RecordMember<T> || Base<T>
  constexpr auto is_private_v = is_private<T>::value;
template <class T>
requires Variable<T> || Callable<T>
   constexpr auto is_constexpr_v = is_constexpr<T>::value;
template <class T>
requires Variable<T> || MemberFunction<T>
   constexpr auto is_static_v = is_static<T>::value;
template <class T>
requires Class<T> || MemberFunction<T>
   constexpr auto is_final_v = is_final<T>::value;
template <class T>
requires Constructor<T> || ConversionOperator<T>
   constexpr auto is_explicit_v = is_explicit<T>::value;
template <class T>
requires Namespace<T> || Callable<T>
   constexpr auto is_inline_v = is_inline<T>::value;
template <class T>
requires Base<T> || MemberFunction<T> || Destructor<T>
   constexpr auto is_virtual_v = is_virtual<T>::value;
template <class T>
requires MemberFunction<T> || Destructor<T>
   constexpr auto is_pure_virtual_v = is_pure_virtual<T>::value;
template <class T>
requires Variable<T> || Function<T>
   constexpr auto get_pointer_v = get_pointer<T>::value;

// 18.11.4.1 Object operations
template <Object T1, Object T2> struct reflects_same;
template <Object T> struct get_source_line;
template <Object T> struct get_source_column;
template <Object T> struct get_source_file_name;

template <Object T1, Object T2>
  constexpr auto reflects_same_v = reflects_same<T1, T2>::value;
template <class T>
  constexpr auto get_source_line_v = get_source_line<T>::value;
template <class T>
  constexpr auto get_source_column_v = get_source_column<T>::value;
template <class T>
  constexpr auto get_source_file_name_v = get_source_file_name<T>::value;

// 18.11.4.2 ObjectSequence operations
template <ObjectSequence T> struct get_size;
```



```
template <size_t I, ObjectSequence T> struct get_element;
template <template <class...> class Tpl, ObjectSequence T>
  struct unpack_sequence;

template <ObjectSequence T>
  constexpr auto get_size_v = get_size<T>::value;
template <size_t I, ObjectSequence T>
  using get_element_t = typename get_element<I, T>::type;
template <template <class...> class Tpl, ObjectSequence T>
  using unpack_sequence_t = typename unpack_sequence<Tpl, T>::type;

// 18.11.4.3 Named operations
template <Named T> struct is_unnamed;
template <Named T> struct get_name;
template <Named T> struct get_display_name;

template <Named T>
  constexpr auto is_unnamed_v = is_unnamed<T>::value;
template <Named T>
  constexpr auto get_name_v = get_name<T>::value;
template <Named T>
  constexpr auto get_display_name_v = get_display_name<T>::value;

// 18.11.4.4 Alias operations
template <Alias T> struct get_aliased;

template <Alias T>
  using get_aliased_t = typename get_aliased<T>::type;

// 18.11.4.5 Type operations
template <Typed T> struct get_type;
template <Type T> struct get_reflected_type;
template <Type T> struct is_enum;
template <Class T> struct uses_class_key;
template <Class T> struct uses_struct_key;
template <Type T> struct is_union;

template <Typed T>
  using get_type_t = typename get_type<T>::type;
template <Type T>
  using get_reflected_type_t = typename get_reflected_type<T>::type;
template <Type T>
  constexpr auto is_enum_v = is_enum<T>::value;
template <Class T>
  constexpr auto uses_class_key_v = uses_class_key<T>::value;
template <Class T>
  constexpr auto uses_struct_key_v = uses_struct_key<T>::value;
template <Type T>
  constexpr auto is_union_v = is_union<T>::value;

// 18.11.4.6 Member operations
template <ScopeMember T> struct get_scope;
template <RecordMember T> struct is_public<T>;
template <RecordMember T> struct is_protected<T>;
template <RecordMember T> struct is_private<T>;
```



```
template <ScopeMember T>
  using get_scope_t = typename get_scope<T>::type;

// 18.11.4.7 Record operations
template <Record T> struct get_public_data_members;
template <Record T> struct get_accessible_data_members;
template <Record T> struct get_data_members;
template <Record T> struct get_public_member_functions;
template <Record T> struct get_accessible_member_functions;
template <Record T> struct get_member_functions;
template <Record T> struct get_public_member_types;
template <Record T> struct get_accessible_member_types;
template <Record T> struct get_member_types;
template <Record T> struct get_constructors;
template <Record T> struct get_destructor;
template <Record T> struct get_operators;
template <Class T> struct get_public_base_classes;
template <Class T> struct get_accessible_base_classes;
template <Class T> struct get_base_classes;
template <Class T> struct is_final<T>;

template <Record T>
  using get_public_data_members_t = typename get_public_data_members<T>::type;
template <Record T>
  using get_accessible_data_members_t = typename get_accessible_data_members<T>::type;
template <Record T>
  using get_data_members_t = typename get_data_members<T>::type;
template <Record T>
  using get_public_member_functions_t = typename get_public_member_functions<T>::type;
template <Record T>
  using get_accessible_member_functions_t = typename get_accessible_member_functions<T>::type;
template <Record T>
  using get_member_functions_t = typename get_member_functions<T>::type;
template <Record T>
  using get_public_member_types_t = typename get_public_member_types<T>::type;
template <Record T>
  using get_accessible_member_types_t = typename get_accessible_member_types<T>::type;
template <Record T>
  using get_member_types_t = typename get_member_types<T>::type;
template <Record T>
  using get_constructors_t = typename get_constructors<T>::type;
template <Record T>
  using get_destructor_t = typename get_destructor<T>::type;
template <Record T>
  using get_operators_t = typename get_operators<T>::type;
template <Class T>
  using get_public_base_classes_t = typename get_public_base_classes<T>::type;
template <Class T>
  using get_accessible_base_classes_t = typename get_accessible_base_classes<T>::type;
template <Class T>
  using get_base_classes_t = typename get_base_classes<T>::type;

// 18.11.4.8 Enum operations
template <Enum T> struct is_scoped_enum;
```



```
template <Enum T> struct get_enumerators;
template <Enum T> struct get_underlying_type;

template <Enum T>
  constexpr auto is_scoped_enum_v = is_scoped_enum<T>::value;
template <Enum T>
  using get_enumerators_t = typename get_enumerators<T>::type;
template <Enum T>
  using get_underlying_type_t = typename get_underlying_type<T>::type;

// 18.11.4.9 Value operations
template <Constant T> struct get_constant;
template <Variable T> struct is_constexpr<T>;
template <Variable T> struct is_static<T>;
template <Variable T> struct is_thread_local;
template <Variable T> struct get_pointer<T>;

template <Constant T>
  constexpr auto get_constant_v = get_constant<T>::value;
template <Variable T>
  constexpr auto is_thread_local_v = is_thread_local<T>::value;

// 18.11.4.10 Base operations
template <Base T> struct get_class;
template <Base T> struct is_virtual<T>;
template <Base T> struct is_public<T>;
template <Base T> struct is_protected<T>;
template <Base T> struct is_private<T>;

template <Base T>
  using get_class_t = typename get_class<T>::type;

// 18.11.4.11 Namespace operations
template <Namespace T> struct is_inline<T>;

// 18.11.4.12 FunctionParameter operations
template <FunctionParameter T> struct has_default_argument;

template <FunctionParameter T>
  constexpr auto has_default_argument_v = has_default_argument<T>::value;

// 18.11.4.13 Callable operations
template <Callable T> struct get_parameters;
template <Callable T> struct is_vararg;
template <Callable T> struct is_constexpr<T>;
template <Callable T> struct is_noexcept;
template <Callable T> struct is_inline<T>;
template <Callable T> struct is_deleted;

template <Callable T>
  using get_parameters_t = typename get_parameters<T>::type;
template <Callable T>
  constexpr auto is_vararg_v = is_vararg<T>::value;
template <Callable T>
  constexpr auto is_deleted_v = is_deleted<T>::value;
```



```
// 18.11.4.14 ParenthesizedExpression operations
template <ParenthesizedExpression T> struct get_subexpression;

template <ParenthesizedExpression T>
   using get_subexpression_t = typename get_subexpression<T>::type;

// 18.11.4.15 FunctionCallExpression operations
template <FunctionCallExpression T> struct get_callable;

template <FunctionCallExpression T>
  using get_callable_t = typename get_callable<T>::type;

// 18.11.4.16 FunctionalTypeConversion operations
template <FunctionalTypeConversion T> struct get_constructor;

template <FunctionalTypeConversion T>
  using get_constructor_t = typename get_constructor<T>::type;

// 18.11.4.17 Function operations
template <Function T> struct get_pointer<T>;

// 18.11.4.18 MemberFunction operations
template <MemberFunction T> struct is_static<T>;
template <MemberFunction T> struct is_const;
template <MemberFunction T> struct is_volatile;
template <MemberFunction T> struct has_lvalueref_qualifier;
template <MemberFunction T> struct has_rvalueref_qualifier;
template <MemberFunction T> struct is_virtual<T>;
template <MemberFunction T> struct is_pure_virtual<T>;
template <MemberFunction T> struct is_override;
template <MemberFunction T> struct is_final<T>;

template <MemberFunction T>
  constexpr auto is_const_v = is_const<T>::value;
template <MemberFunction T>
  constexpr auto is_volatile_v = is_volatile<T>::value;
template <MemberFunction T>
  constexpr auto has_lvalueref_qualifier_v = has_lvalueref_qualifier<T>::value;
template <MemberFunction T>
  constexpr auto has_rvalueref_qualifier_v = has_rvalueref_qualifier<T>::value;
template <MemberFunction T>
  constexpr auto is_override_v = is_override<T>::value;

// 18.11.4.19 SpecialMemberFunction operations
template <SpecialMemberFunction T> struct is_implicitly_declared;
template <SpecialMemberFunction T> struct is_defaulted;

template <SpecialMemberFunction T>
  constexpr auto is_implicitly_declared_v = is_implicitly_declared<T>::value;
template <SpecialMemberFunction T>
  constexpr auto is_defaulted_v = is_defaulted<T>::value;

// 18.11.4.20 Constructor operations
template <Constructor T> struct is_explicit<T>;
```



```
// 18.11.4.21 Destructor operations
template <Destructor T> struct is_virtual<T>;
template <Destructor T> struct is_pure_virtual<T>;

// 18.11.4.22 ConversionOperator operations
template <ConversionOperator T> struct is_explicit<T>;

// 18.11.4.23 Lambda operations
template <Lambda T> struct get_captures;
template <Lambda T> struct uses_default_copy_capture;
template <Lambda T> struct uses_default_reference_capture;
template <Lambda T> struct is_call_operator_const;

template <Lambda T>
  using get_captures_t = typename get_captures<T>::type;
template <Lambda T>
  constexpr auto uses_default_copy_capture_v = uses_default_copy_capture<T>::value;
template <Lambda T>
  constexpr auto uses_default_reference_capture_v = uses_default_reference_capture<T>::value;
template <Lambda T>
  constexpr auto is_call_operator_const_v = is_call_operator_const<T>::value;

// 18.11.4.24 LambdaCapture operations
template <LambdaCapture T> struct is_explicitly_captured;
template <LambdaCapture T> struct is_init_capture;

template <LambdaCapture T>
  constexpr auto is_explicitly_captured_v = is_explicitly_captured<T>::value;
template <LambdaCapture T>
  constexpr auto is_init_capture_v = is_init_capture<T>::value;

} // inline namespace v1
} // namespace reflect
} // namespace experimental
} // namespace std
```

18.11.3 Concepts for meta-object types [reflect.concepts]

1 The operations on meta-object types defined here require meta-object types to satisfy certain concepts (7.1.7). These concepts are also used to specify the result type for *TransformationTrait* type transformations that yield meta-object types. [*Note:* Unlike `std::is_enum`, `std::experimental::reflect::is_enum` operates on meta-object types. — *end note*]

18.11.3.1 Concept `Object` [reflect.concepts.object]

```
template <class T> concept bool Object = see below;
```

1 `Object<T>` is true if and only if `T` is a meta-object type, as generated by the `reflexpr` operator or any of the meta-object operations that in turn generate meta-object types.

18.11.3.2 Concept `ObjectSequence` [reflect.concepts.objseq]

```
template <class T> concept bool ObjectSequence = Object<T> && see below;
```

1 `ObjectSequence<T>` is true if and only if `T` is a sequence of `Objects`, generated by a meta-object operation.

18.11.3.3 Concept `TemplateParameterScope` [reflect.concepts.tempparmscope]

```
template <class T> concept bool TemplateParameterScope = Scope<T> && see below;
```

1 `TemplateParameterScope<T>` is `true` if and only if `T` is a `Scope` reflecting the scope of a template *type-parameter*, generated by a metaobject operation. [*Note:* It represents the template parameter scope (3.3.9), providing a scope to the `Alias` reflecting a template *type-parameter*. —*end note*]

18.11.3.4 Concept `Named` [reflect.concepts.named]

```
template <class T> concept bool Named = Object<T> && see below;
```

1 `Named<T>` is true if and only if `T` has an associated (possibly empty) name.

18.11.3.5 Concept `Alias` [reflect.concepts.alias]

```
template <class T> concept bool Alias = Named<T> && ScopeMember<T> && see below;
```

1 `Alias<T>` is true if and only if `T` reflects a `typedef` declaration, an *alias-declaration*, a *namespace-alias*, a template *type-parameter*, a *decltype-specifier*, or a declaration introduced by a *using-declaration*. [*Note:* The `Scope` of an `Alias` is the scope that the alias was injected into. For an `Alias` reflecting a template *type-parameter*, that scope is its `TemplateParameterScope`. —*end note*] [*Example:*

```
namespace N {
   struct A;
}
namespace M {
   using X = N::A;
}
struct B {
   int i;
};
struct C {
   using B::i;
};
using M_X_t = reflexpr(M::X);
using M_X_scope_t = get_scope_t<M_X_t>;
using C_i_t = reflexpr(C::i);
using C_i_scope_t = get_scope_t<C_i_t>;
```

The scope reflected by `M_X_scope_t` is `M`, not `N`; the scope reflected by `C_i_scope_t` is `C`, not `B`. —*end example*]

2 Type transformations (18.11.4) never yield an `Alias`; instead, they yield the aliased entity.

18.11.3.6 Concept `RecordMember` [reflect.concepts.recordmember]

```
template <class T> concept bool RecordMember = ScopeMember<T> && see below;
```

1 `RecordMember<T>` is true if and only if `T` reflects a *member-declaration*.

18.11.3.7 Concept `Enumerator` [reflect.concepts.enumerator]

```
template <class T> concept bool Enumerator = Constant<T> && see below;
```

1 `Enumerator<T>` is true if and only if `T` reflects an enumerator. [*Note:* The `Scope` of an `Enumerator` is its type also for enumerations that are unscoped enumeration types. —*end note*]

18.11.3.8 Concept `Variable` [reflect.concepts.variable]

```
template <class T> concept bool Variable = Typed<T> && ScopeMember<T> && see below;
```

1 `Variable<T>` is true if and only if `T` reflects a variable or data member.

18.11.3.9 Concept `ScopeMember` [reflect.concepts.scopemember]

```
template <class T> concept bool ScopeMember = Named<T> && see below;
```

1 `ScopeMember<T>` is true if and only if `T` satisfies `RecordMember`, `Enumerator`, or `Variable`, or if `T` reflects a namespace that is not the global namespace. [*Note:* The scope of members of an unnamed union is the unnamed union; the scope of enumerators is their type. —*end note*]

18.11.3.10 Concept `Typed` [reflect.concepts.typed]

```
template <class T> concept bool Typed = Object<T> && see below;
```

1 `Typed<T>` is true if and only if `T` reflects an entity with a type.

18.11.3.11 Concept `Namespace` [reflect.concepts.namespace]

```
template <class T> concept bool Namespace = Named<T> && Scope<T> && see below;
```

1 `Namespace<T>` is true if and only if `T` reflects a namespace (including the global namespace). [*Note:* Any such `T` that does not reflect the global namespace also satisfies `ScopeMember`. —*end note*]

18.11.3.12 Concept `GlobalScope` [reflect.concepts.globalscope]

```
template <class T> concept bool GlobalScope = Namespace<T> && see below;
```

1 `GlobalScope<T>` is true if and only if `T` reflects the global namespace. [*Note:* Any such `T` does not satisfy `ScopeMember`. —*end note*]

18.11.3.13 Concept `Class` [reflect.concepts.class]

```
template <class T> concept bool Class = Record<T> && see below;
```

1 `Class<T>` is true if and only if `T` reflects a non-union class type.

18.11.3.14 Concept `Enum` [reflect.concepts.enum]

```
template <class T> concept bool Enum = Type<T> && Scope<T> && ScopeMember<T> && see below;
```

1 `Enum<T>` is true if and only if `T` reflects an enumeration type.

18.11.3.15 Concept `Record` [reflect.concepts.record]

```
template <class T> concept bool Record = Type<T> && Scope<T> && ScopeMember<T> && see below;
```

1 `Record<T>` is true if and only if `T` reflects a class type.

18.11.3.16 Concept `Scope` [reflect.concepts.scope]

```
template <class T> concept bool Scope = Object<T> && see below;
```

1 `Scope<T>` is true if and only if `T` reflects a namespace (including the global namespace), class, enumeration, function, closure type, or is a `TemplateParameterScope`. [*Note:* Any such `T` that does not reflect the global namespace also satisfies `ScopeMember`. —*end note*]

18.11.3.17 Concept `Type` [reflect.concepts.type]

```
template <class T> concept bool Type = Named<T> && see below;
```

1 `Type<T>` is true if and only if `T` reflects a type. [*Note:* Some types `T` also satisfy `ScopeMember`; others, for instance those reflecting *cv*-qualified types or fundamental types, do not. —*end note*]

18.11.3.18 Concept `Constant` [reflect.concepts.const]

```
template <class T> concept bool Constant = Typed<T> && ScopeMember<T> && see below;
```

1 `Constant<T>` is true if and only if `T` reflects an enumerator or a `constexpr` variable.

18.11.3.19 Concept `Base` [reflect.concepts.base]

```
template <class T> concept bool Base = Object<T> && see below;
```

1 `Base<T>` is true if and only if `T` reflects a direct base class, as returned by the template `get_base_classes`.

18.11.3.20 Concept `FunctionParameter` [reflect.concepts.fctparam]

```
template <class T> concept bool FunctionParameter = Typed<T> && ScopeMember<T> && see below;
```

1 `FunctionParameter<T>` is `true` if and only if `T` reflects a function parameter. [*Note:* The `Scope` of a `FunctionParameter` is the `Callable` to which this parameter appertains. — *end note*] [*Note:* A `FunctionParameter` does not satisfy `Variable`, and thus does not offer an interface for getting the pointer to a parameter. — *end note*]

18.11.3.21 Concept `Callable` [reflect.concepts.callable]

```
template <class T> concept bool Callable = Scope<T> && ScopeMember<T> && see below;
```

1 `Callable<T>` is `true` if and only if `T` reflects a function, including constructors and destructors.

18.11.3.22 Concept `Expression` [reflect.concepts.expr]

```
template <class T> concept bool Expression = Object<T> && see below;
```

1 `Expression<T>` is `true` if and only if `T` reflects an expression (Clause 5).

18.11.3.23 Concept `ParenthesizedExpression` [reflect.concepts.expr.paren]

```
template <class T> concept bool ParenthesizedExpression = Expression<T> && see below;
```

1 `ParenthesizedExpression<T>` is `true` if and only if `T` reflects a parenthesized expression (5.1.1).

18.11.3.24 Concept `FunctionCallExpression` [reflect.concepts.expr.fctcall]

```
template <class T> concept bool FunctionCallExpression = Expression<T> && see below;
```

1 `FunctionCallExpression<T>` is `true` if and only if `T` reflects a *function-call-expression* (5.2.2).

18.11.3.25 Concept `FunctionalTypeConversion` [reflect.concepts.expr.type.fctconv]

```
template <class T> concept bool FunctionalTypeConversion = Expression<T> && see below;
```

1 `FunctionalTypeConversion<T>` is `true` if and only if `T` reflects a *functional-type-conv-expression* (5.2.3).

18.11.3.26 Concept `Function` [reflect.concepts.fct]

```
template <class T> concept bool Function = Typed<T> && Callable<T> && see below;
```

1 `Function<T>` is `true` if and only if `T` reflects a function, excluding constructors and destructors.

18.11.3.27 Concept `MemberFunction` [reflect.concepts.memfct]

```
template <class T> concept bool MemberFunction = RecordMember<T> && Function<T> && see below;
```

1 `MemberFunction<T>` is `true` if and only if `T` reflects a member function, excluding constructors and destructors.

18.11.3.28 Concept `SpecialMemberFunction` [reflect.concepts.specialfct]

```
template <class T> concept bool SpecialMemberFunction = RecordMember<T> && see below;
```

1 `SpecialMemberFunction<T>` is `true` if and only if `T` reflects a special member function (Clause 12).

18.11.3.29 Concept `Constructor` [reflect.concepts.ctor]

```
template <class T> concept bool Constructor = Callable<T> && RecordMember<T> && see below;
```

1 `Constructor<T>` is `true` if and only if `T` reflects a constructor. [*Note:* Some types that satisfy `Constructor` also satisfy `SpecialMemberFunction`. —*end note*]

18.11.3.30 Concept `Destructor` [reflect.concepts.dtor]

```
template <class T>
concept bool Destructor = Callable<T> && SpecialMemberFunction<T> && see below;
```

1 `Destructor<T>` is `true` if and only if `T` reflects a destructor.

18.11.3.31 Concept `Operator` [reflect.concepts.oper]

```
template <class T> concept bool Operator = Function<T> && see below;
```

1 `Operator<T>` is `true` if and only if `T` reflects an operator function (13.5) or a conversion function (12.3.2). [*Note:* Some types that satisfy `Operator` also satisfy `MemberFunction` or `SpecialMemberFunction`. —*end note*]

18.11.3.32 Concept `ConversionOperator` [reflect.concepts.convfct]

```
template <class T>
concept bool ConversionOperator = MemberFunction<T> && Operator<T> && see below;
```

1 `ConversionOperator<T>` is `true` if and only if `T` reflects a conversion function (12.3.2).

18.11.3.33 Concept `Lambda` [reflect.concepts.lambda]

```
template <class T> concept bool Lambda = Type<T> && Scope<T> && see below;
```

1 `Lambda<T>` is `true` if and only if `T` reflects a closure type (excluding generic lambdas).

18.11.3.34 Concept `LambdaCapture` [reflect.concepts.lambdacapture]

```
template <class T> concept bool LambdaCapture = Variable<T> && see below;
```

1 `LambdaCapture<T>` is `true` if and only if `T` reflects a lambda capture as introduced by the capture list or by capture defaults. [*Note:* The `Scope` of a `LambdaCapture` is its immediately enclosing `Lambda`. —*end note*]

18.11.4 Meta-object operations [reflect.ops]

1 A meta-object operation extracts information from meta-object types. It is a class template taking one or more arguments, at least one of which models the `Object` concept. The result of a meta-object operation can be either a constant expression (5.19) or a type.

2 Some operations specify result types with a nested type called `type` that satisfies one of the concepts in `reflect`. These nested types will possibly satisfy other concepts, for instance more specific ones, or independent ones, as applicable for the entity reflected by the nested type. [*Example:*

```
struct X {};
X x;
using x_t = get_type_t<reflexpr(x)>;
```


While `get_type_t` is specified to be a `Type`, `x_t` also satisfies `Class`. —*end example*] Alias entities are not returned by meta-object operations (18.11.3.5).

3 If subsequent specializations of operations on the same reflected entity could give different constant expression results (for instance for `get_name_v` because the parameter's function is re-declared with a different parameter name between the two points of instantiation), the program is ill-formed, no diagnostic required. [*Example:*

```
void func(int a);
auto x1 = get_name_v<get_element_t<0, get_parameters_t<reflexpr(func(42))>>>;
void func(int b);
auto x2 = get_name_v<get_element_t<0, get_parameters_t<reflexpr(func(42))>>>; // ill-formed,
                                                                              // no diagnostic required
```

—*end example*]

18.11.4.1 Object operations [reflect.ops.object]

```
template <Object T1, Object T2> struct reflects_same;
```

1 All specializations of `reflects_same<T1, T2>` shall meet the `BinaryTypeTrait` requirements (20.10.1), with a base characteristic of `true_type` if

(1.1) — `T1` and `T2` reflect the same alias, or

(1.2) — neither `T1` nor `T2` reflect an alias and `T1` and `T2` reflect the same aspect;

otherwise, with a base characteristic of `false_type`.

2 [*Example:* With

```
class A;
using a0 = reflexpr(A);
using a1 = reflexpr(A);
class A {};
using a2 = reflexpr(A);
constexpr bool b1 = is_same_v<a0, a1>; // unspecified value
constexpr bool b2 = reflects_same_v<a0, a1>; // true
constexpr bool b3 = reflects_same_v<a0, a2>; // true

struct C { };
using C1 = C;
using C2 = C;
constexpr bool b4 = reflects_same_v<reflexpr(C1), reflexpr(C2)>; // false
```

—*end example*]

```
template <Object T> struct get_source_line;
template <Object T> struct get_source_column;
```

3 All specializations of above templates shall meet the `UnaryTypeTrait` requirements (20.10.1) with a base characteristic of `integral_constant<uint_least32_t>` and a value of the presumed line number (16.8) (for `get_source_line<T>`) and an implementation-defined value representing some offset from the start of the line (for `get_source_column<T>`) of a declaration of the entity or typedef described by `T`.

```
template <Object T> struct get_source_file_name;
```

4 All specializations of `get_source_file_name<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1) with a static data member named `value` of type `const char (&)[N]`, referencing a static, constant expression character array (NTBS) of length `N`, as if declared as `static constexpr char STR[N] = ...;`. The value of the NTBS is the presumed name of the source file (16.8) of a declaration of the entity or typedef described by `T`.

18.11.4.2 ObjectSequence operations [reflect.ops.objseq]

```
template <ObjectSequence T> struct get_size;
```

1 All specializations of `get_size<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1) with a base characteristic of `integral_constant<size_t, N>`, where `N` is the number of elements in the object sequence.

```
template <size_t I, ObjectSequence T> struct get_element;
```

2 All specializations of `get_element<I, T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` corresponds to the `I`th element `Object` in `T`, where the indexing is zero-based.

```
template <template <class...> class Tpl, ObjectSequence T> struct unpack_sequence;
```

3 All specializations of `unpack_sequence<Tpl, T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates the template `Tpl` specialized with the element `Objects` in `T`.

18.11.4.3 Named operations [reflect.ops.named]

```
template <Named T> struct is_unnamed;
template <Named T> struct get_name;
template <Named T> struct get_display_name;
```

1 All specializations of `is_unnamed<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1) with a base characteristic as specified below.

2 All specializations of `get_name<T>` and `get_display_name<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1) with a static data member named `value` of type `const char (&)[N]`, referencing a static, constant expression character array (NTMBS) of length `N`, as if declared as `static constexpr char STR[N] = ...;`.

3 The value of `get_display_name_v<T>` is the empty string if `T` reflects an unnamed entity; otherwise the value is implementation defined.

4 The value of `get_name_v<T>` refers to a string literal whose *s-char-sequence* is obtained by the first matching case in the following list:

(4.1) — for `T` reflecting an alias, the unqualified name of the aliasing declaration;

(4.2) — for `T` reflecting an unnamed entity, the empty string;

(4.3) — for `T` reflecting a specialization of a class, function (except for conversion functions, constructors, operator functions, or literal operators), or variable template, its *template-name*;

(4.4) — for `T` reflecting a *cv*-unqualified type that is

(4.4.1) — a *type-parameter*, the identifier introduced by the *type-parameter*;

(4.4.2) — a class type, its *class-name*;

(4.4.3) — an enumeration type, its *enum-name*;

(4.4.4) — a fundamental type other than `std::nullptr_t`, the name stated in the "Type" column of Table 10 in 7.1.6.2;

(4.5) — for `T` reflecting

(4.5.1) — a namespace, its *namespace-name*;

(4.5.2) — a variable, enumerator, data member, or function (except for conversion functions, constructors, operator functions, or literal operators), its unqualified name;

(4.5.3) — a function parameter, its name;

(4.5.4) — a constructor, the *injected-class-name* of its class;

(4.5.5) — a destructor, the *injected-class-name* of its class, prefixed by the character `~`;

(4.5.6) — an operator function, the operator element of the relevant *operator-function-id*;

(4.5.7) — a literal operator, the *s-char-sequence* `\"\"` followed by the literal suffix identifier of the operator's *literal-operator-id*;

(4.5.8) — a conversion function, the same characters as `get_name_v<R>`, with `R` reflecting the type represented by the *conversion-type-id*.

(4.6) — In all other cases, the string's value is the empty string.

5 [*Note:* With

```
namespace n { template <class T> class A; }
using a_m = reflexpr(n::A<int>);
```

the value of `get_name_v<a_m>` is `"A"` while the value of `get_display_name_v<a_m>` might be `"n::A<int>"`. —*end note*]

6 The base characteristic of `is_unnamed<T>` is `true_type` if the value of `get_name_v<T>` is the empty string, otherwise it is `false_type`.

7 Subsequent specializations of `get_name<T>` on the same reflected function parameter can render the program ill-formed, no diagnostic required (18.11.4).

18.11.4.4 Alias operations [reflect.ops.alias]

```
template <Alias T> struct get_aliased;
```

1 All specializations of `get_aliased<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` is the `Named` meta-object type reflecting

(1.1) — the redefined name, if `T` reflects an alias;

(1.2) — the template specialization's template argument type, if `T` reflects a template *type-parameter*;

(1.3) — the original declaration introduced by a *using-declaration*;

(1.4) — the aliased namespace of a *namespace-alias*;

(1.5) — the type denoted by the *decltype-specifier*.

2 The nested type named `type` is not an `Alias`; instead, it is reflecting the underlying non-`Alias` entity.

3 [*Example:* For

```
using i0 = int; using i1 = i0;
```

`get_aliased_t<reflexpr(i1)>` reflects `int`. —*end example*]

18.11.4.5 Type operations [reflect.ops.type]

```
template <Typed T> struct get_type;
```

1 All specializations of `get_type<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` is the `Type` reflecting the type of the entity reflected by `T`.

2 [*Example:* For

```
int v; using v_m = reflexpr(v);
```

`get_type_t<v_m>` reflects `int`. —*end example*]

3 If the entity reflected by `T` is a static data member that is declared to have a type array of unknown bound in the class definition, possible specifications of the array bound will only be accessible when the *reflexpr-operand* is the data member.

4 [*Note:* For

```
struct C {
   static int arr[][17];
};
int C::arr[42][17];
using C1 = get_type_t<get_element_t<0, get_data_members_t<reflexpr(C)>>>;
using C2 = get_type_t<reflexpr(C::arr)>;
```

`C1` reflects `int[][17]` while `C2` reflects `int[42][17]`. —*end note*]

```
template <Type T> struct get_reflected_type;
```

5 All specializations of `get_reflected_type<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` is the type reflected by `T`.

6 [*Example:* For


```
using int_m = reflexpr(int);
get_reflected_type_t<int_m> x; // x is of type int
```

— end example]

```
template <Type T> struct is_enum;
template <Type T> struct is_union;
```

7 All specializations of `is_enum<T>` and `is_union<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects an enumeration type (a union), the base characteristic of `is_enum<T>` (`is_union<T>`) is `true_type`, otherwise it is `false_type`.

```
template <Class T> struct uses_class_key;
template <Class T> struct uses_struct_key;
```

8 All specializations of `uses_class_key<T>` and `uses_struct_key<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a class for which all declarations use *class-key* `class` (for `uses_class_key<T>`) or `struct` (for `uses_struct_key<T>`), the base characteristic of the respective template specialization is `true_type`. If `T` reflects a class for which no declaration uses *class-key* `class` (for `uses_class_key<T>`) or `struct` (for `uses_struct_key<T>`), the base characteristic of the respective template specialization is `false_type`. Otherwise, it is unspecified whether the base characteristic is `true_type` or `false_type`.

18.11.4.6 Member operations [reflect.ops.member]

1 A specialization of any of these templates with a meta-object type that is reflecting an incomplete type renders the program ill-formed. Such errors are not in the immediate context (14.8.2).

```
template <ScopeMember T> struct get_scope;
```

2 All specializations of `get_scope<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` is the `Scope` reflecting a scope *S*. With *ST* being the scope of the declaration of the entity, alias or value reflected by `T`, *S* is found as the innermost scope enclosing *ST* that is either a namespace scope (including global scope), class scope, enumeration scope, function scope, or immediately enclosing closure type (for lambda captures). For members of an unnamed union, this innermost scope is the unnamed union. For enumerators of unscoped enumeration types, this innermost scope is their enumeration type. For a template *type-parameter*, this innermost scope is the `TemplateParameterScope` representing the template parameter scope in which it has been declared.

```
template <RecordMember T> struct is_public<T>;
template <RecordMember T> struct is_protected<T>;
template <RecordMember T> struct is_private<T>;
```

3 All specializations of these partial template specializations shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a public member (for `is_public`), protected member (for `is_protected`), or private member (for `is_private`), the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.7 Record operations [reflect.ops.record]

1 A specialization of any of these templates with a meta-object type that is reflecting an incomplete type renders the program ill-formed. Such errors are not in the immediate context (14.8.2). Members introduced by *using-declaration*s (7.3.3) are included in the sequences below where applicable; the `Scope` of these members remains that of the declaration of the referenced entity. [*Note:* These members are not `Aliases`, see 18.11.4. A member injected into a derived class may have different access. *— end note*]

```
template <Record T> struct get_public_data_members;
template <Record T> struct get_accessible_data_members;
template <Record T> struct get_data_members;
template <Record T> struct get_public_member_functions;
template <Record T> struct get_accessible_member_functions;
template <Record T> struct get_member_functions;
```


2 All specializations of these templates shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type satisfying `ObjectSequence`, containing elements which satisfy `RecordMember` and reflect the following subset of non-template members that are either declared in the class reflected by `T` or that are introduced by a using-declaration in the class reflected by `T`:

(2.1) — for `get_data_members` (`get_member_functions`), all data (function, including constructor and destructor) members.

(2.2) — for `get_public_data_members` (`get_public_member_functions`), all public data (function, including constructor and destructor) members;

(2.3) — for `get_accessible_data_members` (`get_accessible_member_functions`), all data (function, including constructor and destructor) members that are accessible from the context of the invocation of `reflexpr` which (directly or indirectly) generated `T`. [*Example:*

```
class X {
   int a;

   friend struct Y;
};

struct Y {
  using X_t = reflexpr(X);
};

using X_mem_t = get_accessible_data_members_t<Y::X_t>;
static_assert(get_size_v<X_mem_t> == 1, ""); // passes.
```

— *end example*]

3 The order of the elements in the `ObjectSequence` is the order of the declaration of the members in the class reflected by `T`.

4 *Remarks:* The program is ill-formed if `T` reflects a closure type.

```
template <Record T> struct get_constructors;
template <Record T> struct get_operators;
```

5 All specializations of these templates shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type satisfying `ObjectSequence`, containing elements which satisfy `RecordMember` and reflect the following subset of non-template function members that are either declared in the class reflected by `T` or that are introduced by a using-declaration in function members of the class reflected by `T`:

(5.1) — for `get_constructors`, all constructors.

(5.2) — for `get_operators`, all conversion functions (12.3.2) and operator functions (13.5).

6 The order of the elements in the `ObjectSequence` is the order of the declaration of the members in the class reflected by `T`.

7 *Remarks:* The program is ill-formed if `T` reflects a closure type.

```
template <Record T> struct get_destructor;
```

8 All specializations of `get_destructor<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a `Destructor` type that reflects the destructor declared in the class reflected by `T`.

9 *Remarks:* The program is ill-formed if `T` reflects a closure type.

```
template <Record T> struct get_public_member_types;
template <Record T> struct get_accessible_member_types;
template <Record T> struct get_member_types;
```

10 All specializations of these templates shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type satisfying `ObjectSequence`, containing elements which satisfy `Type` and reflect the following subset of non-template types that are either declared in the class reflected by `T` or that are introduced by a using declaration in the class reflected by `T`:

(10.1) — for `get_public_member_types`, all public nested class types, enum types, or member typedefs;

(10.2) — for `get_accessible_member_types`, all nested class types, enum types, or member typedefs that are accessible from the scope of the invocation of `reflexpr` which (directly or indirectly) generated `T`;

(10.3) — for `get_member_types`, all nested class types, enum types, or member typedefs.

11 The order of the elements in the `ObjectSequence` is the order of the first declaration of the types in the class reflected by `T`.

12 *Remarks:* The program is ill-formed if `T` reflects a closure type.

```
template <Class T> struct get_public_base_classes;
template <Class T> struct get_accessible_base_classes;
template <Class T> struct get_base_classes;
```

13 All specializations of these templates shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type satisfying `ObjectSequence`, containing elements which satisfy `Base` and reflect the following subset of base classes of the class reflected by `T`:

(13.1) — for `get_public_base_classes`, all public direct base classes;

(13.2) — for `get_accessible_base_classes`, all direct base classes whose public members are accessible from the scope of the invocation of `reflexpr` which (directly or indirectly) generated `T`;

(13.3) — for `get_base_classes`, all direct base classes.

14 The order of the elements in the `ObjectSequence` is the order of the declaration of the base classes in the class reflected by `T`.

15 *Remarks:* The program is ill-formed if `T` reflects a closure type.

```
template <Class T> struct is_final<T>;
```

16 All specializations of `is_final<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a class that is marked with the *class-virt-specifier* `final`, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.8 Enum operations [reflect.ops.enum]

```
template <Enum T> struct is_scoped_enum;
```

1 All specializations of `is_scoped_enum<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a scoped enumeration, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

```
template <Enum T> struct get_enumerators;
```

2 All specializations of `get_enumerators<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type satisfying `ObjectSequence`, containing elements which satisfy `Enumerator` and reflect the enumerators of the enumeration type reflected by `T`.

3 *Remarks:* A specialization of this template with a meta-object type that is reflecting an incomplete type renders the program ill-formed. Such errors are not in the immediate context (14.8.2).

```
template <Enum T> struct get_underlying_type;
```

4 All specializations of `get_underlying_type<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type that reflects the underlying type (7.2) of the enumeration reflected by `T`.

5 *Remarks:* A specialization of this template with a meta-object type that is reflecting an incomplete type renders the program ill-formed. Such errors are not in the immediate context (14.8.2).

18.11.4.9 Value operations [reflect.ops.value]

```
template <Constant T> struct get_constant;
```

1 All specializations of `get_constant<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). The type and value of the static data member named `value` are those of the constant expression of the constant reflected by `T`.

```
template <Variable T> struct is_constexpr<T>;
```

2 All specializations of this partial template specialization shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a variable declared with the *decl-specifier* `constexpr`, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

```
template <Variable T> struct is_static<T>;
template <Variable T> struct is_thread_local;
```

3 All specializations of `is_static<T>` and `is_thread_local<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a variable with static (for `is_static`) or thread (for `is_thread_local`) storage duration, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

```
template <Variable T> struct get_pointer<T>;
```

4 If `T` reflects a reference with static storage duration, and the reference has no prior initialization or has not been initialized with an object of static storage duration, the specialization of `get_pointer<T>` has no member named `type`. Otherwise, the specialization of `get_pointer<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1), with a static data member named `value` of type `X` and value `x`, where

(4.1) — for variables with static storage duration: `X` is `add_pointer<Y>`, where `Y` is the type of the variable reflected by `T`, and `x` is the address of that variable; otherwise,

(4.2) — `X` is the pointer-to-member type of the non-static data member reflected by `T` and `x` a pointer to that member.

18.11.4.10 Base operations [reflect.ops.derived]

```
template <Base T> struct get_class;
```

1 All specializations of `get_class<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates `reflexpr(`*`X`*`)`, where *X* is the base class (without retaining possible `Alias` properties, see 18.11.3.5) reflected by `T`.

```
template <Base T> struct is_virtual<T>;
template <Base T> struct is_public<T>;
template <Base T> struct is_protected<T>;
template <Base T> struct is_private<T>;
```

2 All specializations of these partial template specializations shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a direct base class with the `virtual` specifier (for `is_virtual`), with the `public` specifier or with an assumed (see 11.2) `public` specifier (for `is_public`), with the `protected` specifier (for `is_protected`), or with the `private` specifier or with an assumed `private` specifier (for `is_private`), then the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.11 Namespace operations [reflect.ops.namespace]

```
template <Namespace T> struct is_inline<T>;
```

1 All specializations of `is_inline<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects an inline namespace, the base characteristic of the template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.12 FunctionParameter operations [reflect.ops.fctparam]

```
template <FunctionParameter T> struct has_default_argument;
```

1 All specializations of this template shall meet the `UnaryTypeTrait` requirements (20.10.1). If `T` reflects a parameter with a default argument, the base characteristic of `has_default_argument<T>` is `true_type`, otherwise it is `false_type`.

2 *Remarks:* Subsequent specializations of `has_default_argument<T>` on the same reflected function parameter can render the program ill-formed, no diagnostic required (18.11.4).

18.11.4.13 Callable operations [reflect.ops.callable]

```
template <Callable T> struct get_parameters;
```

1 All specializations of this template shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type satisfying `ObjectSequence`, containing elements which satisfy `FunctionParameter` and reflect the parameters of the function reflected by `T`. If that function's *parameter-declaration-clause* (8.3.5) terminates with an ellipsis, the `ObjectSequence` does not contain any additional elements reflecting that. The `is_vararg_v<Callable>` trait can be used to determine if the terminating ellipsis is in its parameter list.

```
template <Callable T> struct is_vararg;
template <Callable T> struct is_constexpr<T>;
template <Callable T> struct is_noexcept;
template <Callable T> struct is_inline<T>;
template <Callable T> struct is_deleted;
```

2 All specializations of these templates and partial template specializations shall meet the `UnaryTypeTrait` requirements (20.10.1). If their template parameter reflects an entity with an ellipsis terminating the *parameter-declaration-clause* (8.3.5) (for `is_vararg`), or an entity that is (where applicable implicitly or explicitly) declared as `constexpr` (for `is_constexpr`), as non-throwing (15.4) (for `is_noexcept`), as an inline function (7.1.2) (for `is_inline`), or as deleted (for `is_deleted`), the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

3 *Remarks:* Subsequent specializations of `is_inline<T>` on the same reflected function can render the program ill-formed, no diagnostic required (18.11.4).

18.11.4.14 ParenthesizedExpression operations [reflect.ops.expr.paren]

```
template <ParenthesizedExpression T> struct get_subexpression;
```

1 All specializations of `get_subexpression<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` is the `Expression` type reflecting the expression `E` of the parenthesized expression `(E)` reflected by `T`.

18.11.4.15 FunctionCallExpression operations [reflect.ops.expr.fctcall]

```
template <FunctionCallExpression T> struct get_callable;
```

1 All specializations of `get_callable<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` is the `Callable` type reflecting the function invoked by the *function-call-expression* which is reflected by `T`.

18.11.4.16 FunctionalTypeConversion operations [reflect.ops.expr.fcttypeconv]

```
template <FunctionalTypeConversion T> struct get_constructor;
```

1 All specializations of `get_constructor<T>` shall meet the `TransformationTrait` requirements (20.10.1). For a *functional-type-conv-expression* reflected by `T`, let `S` be the type specified by the type conversion (5.2.3). If a constructor is used for the initialization of `S`, the type `get_constructor<T>::type` is the `Constructor` reflecting that constructor; otherwise, `get_constructor<T>` has no member named `type`. [*Note:* For instance fundamental types (3.9.1) do not have constructors. —*end note*]

18.11.4.17 Function operations [reflect.ops.fct]

```
template <Function T> struct get_pointer<T>;
```

1 All specializations of `get_pointer<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1), with a static data member named `value` of type `X` and value `x`, where

(1.1) — for non-static member functions, `X` is the pointer-to-member-function type of the member function reflected by `T` and `x` a pointer to the member function; otherwise,

(1.2) — `X` is `add_pointer<Y>`, where `Y` is the type of the function reflected by `T` and `x` is the address of that function.

18.11.4.18 MemberFunction operations [reflect.ops.memfct]

```
template <MemberFunction T> struct is_static<T>;
template <MemberFunction T> struct is_const;
template <MemberFunction T> struct is_volatile;
template <MemberFunction T> struct has_lvalueref_qualifier;
template <MemberFunction T> struct has_rvalueref_qualifier;
template <MemberFunction T> struct is_virtual<T>;
template <MemberFunction T> struct is_pure_virtual<T>;
template <MemberFunction T> struct is_override;
template <MemberFunction T> struct is_final<T>;
```

1 All specializations of these templates and partial template specializations shall meet the `UnaryTypeTrait` requirements (20.10.1). If their template parameter reflects a member function that is `static` (for `is_static`), `const` (for `is_const`), `volatile` (for `is_volatile`), declared with a *ref-qualifier* `&` (for `has_lvalueref_qualifier`) or `&&` (for `has_rvalueref_qualifier`), implicitly or explicitly `virtual` (for `is_virtual`), pure virtual (for `is_pure_virtual`), or overrides a member function of a base class (for `is_override`) or `final` (for `is_final`), the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.19 SpecialMemberFunction operations [reflect.ops.specialfct]

```
template <SpecialMemberFunction T> struct is_implicitly_declared;
template <SpecialMemberFunction T> struct is_defaulted;
```

1 All specializations of these templates shall meet the `UnaryTypeTrait` requirements (20.10.1). If their template parameter reflects a special member function that is implicitly declared (for `is_implicitly_declared`) or that is defaulted in its first declaration (for `is_defaulted`), the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.20 Constructor operations [reflect.ops.ctor]

```
template <Constructor T> struct is_explicit<T>;
```

1 All specializations of this partial template specialization shall meet the `UnaryTypeTrait` requirements (20.10.1). If the template parameter reflects an explicit constructor, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.21 Destructor operations [reflect.ops.dtor]

```
template <Destructor T> struct is_virtual<T>;
template <Destructor T> struct is_pure_virtual<T>;
```

1 All specializations of these partial template specializations shall meet the `UnaryTypeTrait` requirements (20.10.1). If the template parameter reflects a virtual (for `is_virtual`) or pure virtual (for `is_pure_virtual`) destructor, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.22 ConversionOperator operations [reflect.ops.convfct]

```
template <ConversionOperator T> struct is_explicit<T>;
```

1 All specializations of `is_explicit<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If the template parameter reflects an explicit conversion function, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.23 Lambda operations [reflect.ops.lambda]

```
template <Lambda T> struct get_captures;
```

1 All specializations of `get_captures<T>` shall meet the `TransformationTrait` requirements (20.10.1). The nested type named `type` designates a meta-object type satisfying `ObjectSequence`, containing elements which satisfy `LambdaCapture` and reflect the captures of the closure object reflected by `T`. The elements are in order of appearance in the *lambda-capture*; captures captured because of a *capture-default* have no defined order among the default captures.

```
template <Lambda T> struct uses_default_copy_capture;
template <Lambda T> struct uses_default_reference_capture;
```

2 All specializations of these templates shall meet the `UnaryTypeTrait` requirements (20.10.1). If the template parameter reflects a closure object with a *capture-default* that is `=` (for `uses_default_copy_capture`) or `&` (for `uses_default_reference_capture`), the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

```
template <Lambda T> struct is_call_operator_const;
```

3 All specializations of `is_call_operator_const<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If the template parameter reflects a closure object with a `const` function call operator, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

18.11.4.24 LambdaCapture operations [reflect.ops.lambdacapture]

```
template <LambdaCapture T> struct is_explicitly_captured;
```

1 All specializations of `is_explicitly_captured<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If the template parameter reflects an explicitly captured entity, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

```
template <LambdaCapture T> struct is_init_capture;
```

2 All specializations of `is_init_capture<T>` shall meet the `UnaryTypeTrait` requirements (20.10.1). If the template parameter reflects an *init-capture*, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

Annex A (informative) Compatibility [diff]

Modify title for section A.1 to "C++ extensions for Concepts with Reflection and ISO C++ 2014".

A.1 C++ extensions for Concepts with Reflection and ISO C++ 2014 [diff.iso]

In C++ [diff.iso], modify the first paragraph:

This subclause lists the differences between C++ with Reflection and Concepts and ISO C++ by the chapters of this document.

A.1.1 Clause 2: lexical conventions [diff.lex]

In C++ [diff.lex], modify the first paragraph:

1 **Affected subclause:** 2.12
Change: New keywords.
Rationale: Required for new features. The `requires` keyword is added to introduce constraints through a *requires-clause* or a *requires-expression*. The `concept` keyword is added to enable the definition of concepts (7.1.7). The `reflexpr` keyword is added to introduce meta-data through a *reflexpr-specifier*.
Effect on original feature: Valid ISO C++ 2014 code using `concept` ~~or~~, `requires`, or `reflexpr` as an identifier is not valid in this International Standard.

