Document Number: N4746
Date: 2018-05-07
Reply to: David Sankel
dsankel@bloomberg.net

Working Draft, C++ Extensions for Reflection

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 Scope [scope]

1 This Technical Specification describes extensions to the C++ Programming Language (Clause 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, provides important context and specification for this Technical Specification. This document is written as a set of changes against that specification, as modified by 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.

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 14882:2017, *Programming languages — C++*

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

2 ISO/IEC 14882:2017 is hereafter called the *C++ Standard*.

3 ISO/IEC TS 19217:2015 is hereafter called the *Concepts TS*.

4 The numbering of clauses, subclauses, and paragraphs in this document reflects the numbering in the C++ Standard and the Concepts TS. References to clauses and subclauses not appearing in this Technical Specification refer to the original, unmodified text in the Concepts TS, or in the C++ Standard for clauses and subclauses not appearing in the Concepts TS.

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`

4 General [intro]

4.1 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are those defined in subclause 4.1 in the Concepts TS, except that references to the Concepts TS or the C++ Standard therein shall be taken as referring to the document that is the result of applying the editing instructions. Similarly, all references to the Concepts TS or 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*]

4.2 Namespaces and headers [intro.namespaces]

1 Whenever a name `x` declared in subclause 21.11 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>`

4.3 Acknowledgements [intro.ack]

1 This work is the result of a collaboration of researchers in industry and academia. We wish to thank people who made valuable contributions within and outside these groups, including Ricardo Fabiano de Andrade, Roland Bock, Chandler Carruth, Klaim-Jol Lamotte, Jens Maurer, and many others not named here who contributed to the discussion.

5 Lexical conventions [lex]

5.1 Keywords [lex.key]

1 In C++ [lex.key], add the keyword `reflexpr` to the list of keywords in Table 5.

6 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*.

6.1 Fundamental types [basic.fundamental]

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

An expression of type *cv* `void` shall be used only as an expression statement (9.2), as an operand of a comma expression (8.19), as a second or third operand of `?:` (8.16), as the operand of `typeid`, `noexcept`, `reflexpr`, or `decltype`, as the expression in a return statement (9.6.3) for a function with the return type *cv* `void`, or as the operand of an explicit conversion to type *cv* `void`.

7 Standard conversions [conv]

No changes are made to Clause 7 of the C++ Standard.

8 Expressions [expr]

No changes are made to Clause 8 of the C++ Standard.

9 Statements [stmt.stmt]

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

10 Declarations [dcl.dcl]

10.1 Specifiers [dcl.spec]

10.1.7 Type specifiers [dcl.type]

10.1.7.2 Simple type specifiers [dcl.type.simple]

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

...

The other *simple-type-specifiers* specify either a previously-declared type, a type determined from an expression, a reflection meta-object type (10.1.7.6), or one of the fundamental types (6.9.1).

Add the following row to Table 11

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

At the end of 10.1.7.2, insert the following paragraph:

> For a *reflexpr-operand* `x`, the type denoted by `reflexpr(x)` is an implementation-defined type that satisfies constraints as laid out in 10.1.7.6.

10.1.7.6 Reflection type specifiers [dcl.type.reflexpr]

Insert the following section:

The *reflexpr* operator yields a type `T` that allows inspection of some properties of its operand through type traits or type transformations on `T` (21.11.4). The operand to the `reflexpr` operator shall be a type, namespace, enumerator, variable, structured binding or data member. Any such `T` satisfies the requirements of `reflect::Object` (21.11.3) 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 incomplete namespace-scope class type ([class]).

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

- `A` and `B` are the same entity or alias,
- `A` is a variable or enumerator and `B` is the type of `A`,
- `A` is an enumeration and `B` is the underlying type of `A`,
- `A` is a class and `B` is a member or base class of `A`,
- `A` is a non-template alias that designates the entity `B`,
- `A` is a class nested in `B` (12.2.5 [class.nest]),
- `A` is not the global namespace and `B` is an enclosing namespace of `A`, or
- `A` is reflection-related to an entity or alias `X` and `X` is reflection-related to `B`.

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

[*Example:*

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

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

Zero or more successive applications of type transformations that yield meta-object types (21.11.4) 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.

[*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>(); // prints "basic_string"
```


— *end example*]

The type specified by the *reflexpr-specifier* is implementation-defined. 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 (21.11.4) cannot be applied. — *end note*]

[*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*]

For the operand `::`, the type specified by the `reflexpr-specifier` satisfies `reflect::GlobalScope`. For an operand of the form *identifier* where *identifier* is a template *type-parameter*, the type satisfies both `reflect::Type` and `reflect::Alias`.

The *identifier* or *simple-template-id* is looked up using the rules for name lookup (6.4): if a *nested-name-specifier* is included in the operand, qualified lookup (6.4.3) of *nested-name-specifier identifier* or *nested-name-specifier simple-template-id* will be performed, otherwise unqualified lookup (6.4.1) of *identifier* or *simple-template-id* will be performed. The type specified by the *reflexpr-specifier* satisfies concepts depending on the result of the name lookup, as shown in Table 12.

Table 12 — `reflect` concept (21.11.3) that the type specified by a *reflexpr-specifier* satisfies, for a given *reflexpr-operand identifier* or *simple-template-id*.

Category	***identifier* or *simple-template-id* kind**	**`reflect` Concept**
type	*class-name* designating a union	`reflect::Record`
	class-name designating a non-union class	`reflect::Class`
	enum-name	`reflect::Enum`
	type-name introduced by a *using-declaration*	both `reflect::Type` and `reflect::Alias`
	any other *typedef-name*	both `reflect::Type` and `reflect::Alias`
namespace	*namespace-alias*	both `reflect::Namespace` and `reflect::Alias`
	any other *namespace-name*	both `reflect::Namespace` and `reflect::ScopeMember`
data member	the name of a data member	`reflect::Variable`
value	the name of a variable or structured binding that is not a local entity	`reflect::Variable`
	the name of an enumerator	both `reflect::Enumerator` and `reflect::Constant`

If the *reflexpr-operand* designates a *type-id* not explicitly mentioned in Table 12, the type represented by the *reflexpr-specifier* satisfies `reflect::Type`. Any other *reflexpr-operand* renders the program ill-formed.

If the *reflexpr-operand* designates an entity or alias at block scope (6.3.3) or function prototype scope (6.3.4), 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 variable or a data member, it is an unevaluated operand (expr.context). If the *reflexpr-operand* designates both an alias and a class name, the type represented by the *reflexpr-specifier* reflects the alias and satisfies `Alias`.

11 Declarators [dcl.decl]

11.1 Type names [dcl.name]

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.

12 Classes [class]

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

13 Derived classes [class.derived]

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

14 Member access control [class.access]

No changes are made to Clause 14 of the C++ Standard.

15 Special member functions [special]

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

16 Overloading [over]

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

17 Templates [temp]

17.6.2.1 Dependent types [temp.dep.type]

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

— a *simple-template-id* in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent, ~~or~~

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

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

18 Exception handling [except]

No changes are made to Clause 18 of the C++ Standard.

19 Preprocessing directives [cpp]

No changes are made to Clause 19 of the C++ Standard.

20 Library introduction [library]

20.5 Library-wide requirements [requirements]

20.5.1 Library contents and organization [organization]

20.5.1.2 Headers [headers]

Add `<experimental/reflect>` to Table 16 – C++ library headers.

21 Language support library [language.support]

Add a new subclause 21.11 titled "**Static reflection**" as follows:

21.11 Static reflection [reflect]

21.11.1 In general [reflect.general]

As laid out in 10.1.7.6, 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*.

Meta-object types satisfy different concepts (21.11.3) depending on the type they reflect (10.1.7.6). 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 21.11.4.

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

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

// 21.11.3 Concepts for meta-object types
template <class T> concept Object;
template <class T> concept ObjectSequence;
template <class T> concept Named;
template <class T> concept Alias;
template <class T> concept RecordMember;
template <class T> concept Enumerator;
template <class T> concept Variable;
template <class T> concept ScopeMember;
template <class T> concept Typed;
template <class T> concept Namespace;
template <class T> concept GlobalScope;
template <class T> concept Class;
template <class T> concept Enum;
template <class T> concept Record;
template <class T> concept Scope;
template <class T> concept Type;
template <class T> concept Constant;
template <class T> concept Base;


// 21.11.4 Meta-object operations
// 21.11.4.1 Multi-concept operations
template <class T> struct is_public;
template <class T> struct is_protected;
template <class T> struct is_private;

template <class T>
```

```
  constexpr auto is_public_v = is_public<T>::value;
template <class T>
  constexpr auto is_protected_v = is_protected<T>::value;
template <class T>
  constexpr auto is_private_v = is_private<T>::value;

// 21.11.4.2 Object operations
template <Object T1, Object T2> struct reflects_same;
template <class T> struct get_source_line;
template <class T> struct get_source_column;
template <class 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;

// 21.11.4.3 ObjectSequence operations
template <ObjectSequence S> struct get_size;
template <size_t I, ObjectSequence S> struct get_element;
template <template <class...> class Tpl, ObjectSequence S>
  struct unpack_sequence;

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

// 21.11.4.4 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;

// 21.11.4.5 Alias operations
template <Alias T> struct get_aliased;

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

// 21.11.4.6 Type operations
template <Typed T> struct get_type;
template <Type T> struct get_reflected_type;
```



```
template <Type T> struct is_enum;
template <Type T> struct is_class;
template <Type T> struct is_struct;
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 <Type T>
  constexpr auto is_class_v = is_class<T>::value;
template <Type T>
  constexpr auto is_struct_v = is_struct<T>::value;
template <Type T>
  constexpr auto is_union_v = is_union<T>::value;

// 21.11.4.7 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;

// 21.11.4.8 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_types;
template <Record T> struct get_accessible_member_types;
template <Record T> struct get_member_types;
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;

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_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 <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;
template <Class T>
  constexpr auto is_final_v = is_final<T>::value;

// 21.11.4.9 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;

// 21.11.4.10 Value operations
template <Constant T> struct get_constant;
template <Variable T> struct is_constexpr;
template <Variable T> struct is_static;
template <Variable T> struct get_pointer;

template <Constant T>
  constexpr auto get_constant_v = get_constant<T>::value;
template <Variable T>
  constexpr auto is_constexpr_v = is_constexpr<T>::value;
template <Variable T>
  constexpr auto is_static_v = is_static<T>::value;
template <Variable T>
  const auto get_pointer_v = get_pointer<T>::value;

// 21.11.4.11 Base operations
template <Base T> struct get_class;
template <Base T> struct is_virtual;
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;
template <Base T>
  constexpr auto is_virtual_v = is_virtual<T>::value;

// 21.11.4.12 Namespace operations
template <Namespace T> struct is_inline;

template <Namespace T>
  constexpr auto is_inline_v = is_inline<T>::value;

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

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

1 The operations on meta-object types defined here require meta-object types to satisfy certain concepts ([dcl.spec.concept]). These concepts are also used to specify the result type for *TransformationTrait* type transformations that yield meta-object types.

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

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

1 `Object<T>` is satisfied 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.

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

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

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

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

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

1 `Named<T>` is satisfied if and only if `T` is an `Object` with an associated (possibly empty) name.

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

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

1 `Alias<T>` is satisfied if and only if `T` is a `Named` that 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*. Any such `T` also satisfies `ScopeMember`; its scope is the scope that the alias was injected into. [*Example:*

```
namespace N {
   struct A;
}
namespace M {
   using X = N::A;
}
using M_X_t = reflexpr(M::X);
using M_X_scope_t = get_scope_t<M_X_t>;
```

The scope reflected by `M_X_scope_t` is `M`, not `N`. — *end example*]

2 Except for the type represented by the `reflexpr` operator, `Alias` properties resulting from type transformations (21.11.4) are not retained.

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

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

1 `RecordMember<T>` is satisfied if and only if `T` reflects a *member-declaration*. Any such `T` also satisfies `ScopeMember`.

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

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

1 `Enumerator<T>` is satisfied if and only if `T` reflects an enumerator. Any such `T` also satisfies `Typed` and `ScopeMember`; the `Scope` of an `Enumerator` is its type also for enumerations that are unscoped enumeration types.

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

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

1 `Variable<T>` is satisfied if and only if `T` reflects a variable or non-static data member. Any such `T` also satisfies `Typed`.

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

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

1 `ScopeMember<T>` is satisfied if and only if `T` satisfies `RecordMember`, `Enumerator`, or `Variable`, or if `T` reflects a namespace that is not the global namespace. Any such `T` also satisfies `Named`. The scope of members of an unnamed union is the unnamed union; the scope of enumerators is their type.

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

```
template <class T> concept Typed = Variable<T> || Constant<T>;
```

1 `Typed<T>` is satisfied if and only if `T` reflects a variable or enumerator. Any such `T` also satisfies `Named`.

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

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

1 `Namespace<T>` is satisfied if and only if `T` reflects a namespace (including the global namespace). Any such `T` also satisfies `Scope`. Any such `T` that does not reflect the global namespace also satisfies `ScopeMember`.

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

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

1 `GlobalScope<T>` is satisfied if and only if `T` reflects the global namespace. Any such `T` also satisfies `Namespace`; it does not satisfy `ScopeMember`.

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

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

1 `Class<T>` is satisfied if and only if `T` reflects a non-union class type. Any such `T` also satisfies `Record`.

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

```
template <class T> concept Enum = see below;
```

1 `Enum<T>` is satisfied if and only if `T` reflects an enumeration type. Any such `T` also satisfies `Type` and `Scope`.

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

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

1 `Record<T>` is satisfied if and only if `T` reflects a class type. Any such `T` also satisfies `Type` and `Scope`.

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

```
template <class T> concept Scope = Namespace<T> || Record<T> || Enum<T>;
```

1 `Scope<T>` is satisfied if and only if `T` reflects a namespace (including the global namespace), class, or enumeration. Any such `T` that does not reflect the global namespace also satisfies `ScopeMember`.

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

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

1 `Type<T>` is satisfied if and only if `T` reflects a type. Any such `T` also satisfies `Named` and `ScopeMember`.

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

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

1 `Constant<T>` is satisfied if and only if `T` reflects a constant expression ([expr.const]). Any such `T` also satisfies `ScopeMember` and `Typed`.

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

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

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

21.11.4 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 ([expr.const]) or a type.

21.11.4.1 Multi-concept operations [reflect.ops.over]

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

1 These meta-object operations are applicable to both `RecordMember` and `Base`. The generic templates do not have a definition. When multiple concepts implement the same meta-object operation, its template will be partially specialized for the concepts implementing the operation. [*Note:* For these overloaded operations, any meta-object type will always satisfy at most one of the concepts that the operation is applicable to. — *end note*]

2 [*Example:* An operation `OP` applicable to concepts `A` and `B` can be defined as follows:

```
template <class T> concept A = is_signed_v<T>;
template <class T> concept B = is_class_v<T>;
template <class T> struct OP; // undefined
template <A T> struct OP<T> {...};
template <B T> struct OP<T> {...};
```

— *end example*]

21.11.4.2 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 ([meta.rqmts]), 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 entity;

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

3 All specializations of above templates shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]) with a base characteristic of `integral_constant<uint_least32_t>` and a value of the presumed line number ([cpp.predefined]) (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 the most recent declaration of the entity or typedef described by `T`.

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

4 All specializations of `get_source_file_name<T>` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]) 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 ([cpp.predefined]) of the most recent declaration of the entity or typedef described by `T`.

21.11.4.3 ObjectSequence operations [reflect.ops.objseq]

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

1 All specializations of `get_size<S>` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]) 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 S> struct get_element;
```

2 *Remarks:* All specializations of `get_element<I, S>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` corresponds to the `I`th element `Object` in `S`, where the indexing is zero-based.

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

3 *Remarks:* All specializations of `unpack_sequence<Tpl, S>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` is an alias to the template `Tpl` specialized with the types in `S`.

21.11.4.4 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 ([meta.rqmts]) with a base characteristic as specified below.

2 All specializations of `get_name` and `get_display_name` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]) 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] = ...;`.

(2.1) — For `T` reflecting an unnamed entity, the string's value is the empty string.

(2.2) — For `T` reflecting a *decltype-specifier*, the string's value is the empty string for `get_name<T>` and implementation-defined for `get_display_name<T>`.

(2.3) — For `T` reflecting an array, pointer, reference of function type, or a *cv*-qualified type, the string's value is the empty string for `get_name<T>` and implementation-defined for `get_display_name<T>`.

(2.4) — In the following cases, the string's value is implementation-defined for `get_display_name<T>` and has the following value for `get_name<T>`:

(2.4.1) — for `T` reflecting an `Alias`, the unqualified name of the aliasing declaration: the identifier introduced by a *type-parameter* or a type name introduced by a *using-declaration*, alias;

(2.4.2) — for `T` reflecting a specialization of a class template, its *template-name*;

(2.4.3) — for `T` reflecting a class type, its *class-name*;

(2.4.4) — for `T` reflecting a namespace, its *namespace-name*;

(2.4.5) — for `T` reflecting an enumeration type, its *enum-name*;

(2.4.6) — for `T` reflecting all other *simple-type-specifiers*, the name stated in the "Type" column of Table 9 in ([dcl.type.simple]);

(2.4.7) — for `T` reflecting a variable, its unqualified name;

(2.4.8) — for `T` reflecting an enumerator, its unqualified name;

(2.4.9) — for `T` reflecting a class data member, its unqualified name.

(2.5) — In all other cases, the string's value is the empty string for <code>get_name<T></code> and implementation-defined for <code>get_display_name<T></code>.

3 [*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*]

4 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`.

21.11.4.5 Alias operations [reflect.ops.alias]

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

1 All specializations of `get_aliased<T>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). 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*]

21.11.4.6 Type operations [reflect.ops.type]

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

1 All specializations of `get_type<T>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). 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[17][42];
using C1 = get_type_t<get_element_t<0, get_data_members_t<reflexpr(C)>>>;
using C2 = get_type_t<reflexpr(C::arr)>;
```

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

template <Type T> struct get_reflected_type;

5 All specializations of `get_reflected_type<T>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). 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 ([meta.rqmts]). 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 <Type T> struct is_class;
template <Type T> struct is_struct;
```

8 All specializations of these templates shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]). If `T` reflects a class with *class-key* `class` (for `is_class<T>`) or `struct` (for `is_struct<T>`), the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`. If the same class has redeclarations with both *class-key* `class` and *class-key* `struct`, the base characteristic of the template specialization of exactly one of `is_class<T>` and `is_struct<T>` can be `true_type`, the other template specialization is `false_type`; the actual choice of value is unspecified.

21.11.4.7 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 ([temp.deduct]).

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

2 All specializations of `get_scope<T>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` is the `Scope` reflecting a scope *S*. With *ST* being the scope of the declaration of the entity or typedef reflected by `T`, *S* is found as the innermost scope enclosing *ST* that is either a namespace scope (including global scope), class scope, or enumeration scope.


```
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 ([meta.rqmts]). 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`.

21.11.4.8 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 ([temp.deduct]).

```
template <Record T> struct get_public_data_members;
template <Record T> struct get_accessible_data_members;
template <Record T> struct get_data_members;
```

2 All specializations of these templates shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` is an alias to an `ObjectSequence` specialized with `RecordMember` types that reflect the following subset of data members of the class reflected by `T`:

(2.1) — for `get_data_members`, all data members.

(2.2) — for `get_public_data_members`, all public data members;

(2.3) — for `get_accessible_data_members`, all data members that are accessible from the scope of the invocation of `reflexpr` which (directly or indirectly) generated `T`.

3 The order of the elements in the `ObjectSequence` is the order of the declaration of the data 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_public_member_types;
template <Record T> struct get_accessible_member_types;
template <Record T> struct get_member_types;
```

5 All specializations of these templates shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` is an alias to an `ObjectSequence` specialized with `Type` types that reflect the following subset of types declared in the class reflected by `T`:

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

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

(5.3) — 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`.

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

7 *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;
```

8 All specializations of these templates shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` is an alias to an `ObjectSequence` specialized with `Base` types that reflect the following subset of base classes of the class reflected by `T`:

(8.1) — for `get_base_classes`, all direct base classes;

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

(8.3) — 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`.

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

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

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

11 All specializations of `is_final<T>` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]). 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`.

21.11.4.9 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 ([meta.rqmts]). 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 ([meta.rqmts]). The nested type named `type` is an alias to an `ObjectSequence` specialized with `Enumerator` types that 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 ([temp.deduct]).

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

4 All specializations of `get_underlying_type<T>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` is an alias to a meta-object type that reflects the underlying type (10.2) 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 ([temp.deduct]).

21.11.4.10 Value operations [reflect.ops.value]

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

1 All specializations of `get_constant<T>` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]). It has a static data member named `value` whose type and value are those of the constant expression of the constant reflected by `T`.

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

2 All specializations of `is_constexpr<T>` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]). 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;
```

3 All specializations of `is_static<T>` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]). If `T` reflects a variable with static storage duration, the base characteristic of the respective template specialization is `true_type`, otherwise it is `false_type`.

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

4 All specializations of `get_pointer<T>` shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]), 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 member variable reflected by `T` and `x` a pointer to the member.

21.11.4.11 Base operations [reflect.ops.derived]

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 ([temp.deduct]).

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

1 All specializations of `get_class<T>` shall meet the `TransformationTrait` requirements ([meta.rqmts]). The nested type named `type` is an alias to `reflexpr(`*`X`*`)`, where *`X`* is the base class reflected by `T`.

```
template <Base T> struct is_virtual;
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 the template and of these partial template specializations shall meet the `UnaryTypeTrait` requirements ([meta.rqmts]). If `T` reflects a direct base class with the `virtual` specifier (for `is_virtual`), with the `public` specifier or with an assumed (see C++ [class.access.base]) `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`.

21.11.4.12 Namespace operations [reflect.ops.namespace]

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

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