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 1

2 Normative references 2

3 Terms and definitions 3

4 General 4
   4.1 Implementation compliance 4
   4.2 Namespaces and headers 4
   4.3 Feature-testing recommendations 4
   4.4 Acknowledgements 4

5 Lexical conventions 5
   5.1 Keywords 5

6 Basics 6
   6.2 One-definition rule 6
   6.7 Types 6

7 Standard conversions 7

8 Expressions 8
   8.4 Primary expressions 8
   8.5 Compound expressions 8
   8.6 Constant expressions 9

9 Statements 10

10 Declarations 11
   10.1 Specifiers 11

11 Declarators 15
   11.1 Type names 15

12 Classes 16

13 Derived classes 17

14 Member access control 18

15 Special member functions 19

16 Overloading 20

17 Templates 21
   17.7 Name resolution 21

18 Exception handling 22
1 Scope

This document 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.

The International Standard, ISO/IEC 14882, provides important context and specification for this document. This document is written as a set of changes against that specification. 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

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 N4750, Programming languages — C++

2 ISO/IEC N4750 is hereafter called the C++ Standard.

3 The numbering of clauses, subclauses, and paragraphs in this document reflects the numbering in the C++ Standard. References to clauses and subclauses not appearing in this document refer to the original unmodified text in the C++ Standard.
3 Terms and definitions

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

2. ISO Online browsing platform: available at http://www.iso.org/obp
4 General

4.1 Implementation compliance

Conformance requirements for this specification are those defined in subclause 4.1 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]

4.2 Namespaces and headers

Whenever a name x declared in subclause 21.12 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:

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

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

<table>
<thead>
<tr>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;experimental/reflect&gt;</td>
</tr>
</tbody>
</table>

4.3 Feature-testing recommendations

An implementation that provides support for this Technical Specification shall define the feature test macros in Table 2.

Table 2 — Feature-test macros

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_reflection</td>
<td>201902</td>
<td>none</td>
</tr>
<tr>
<td>__cpp_lib_reflection</td>
<td>201902</td>
<td>&lt;reflect&gt;</td>
</tr>
</tbody>
</table>

4.4 Acknowledgements

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, Klaim-Jol Lamotte, Jens Maurer, and many others not named here who contributed to the discussion.
5 Lexical conventions

5.11 Keywords

1 In C++ [lex.key], add the keyword \texttt{reflexpr} to the list of keywords in Table 5.
6 Basics

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.

6.2 One-definition rule

1 In C++ [basic.def.odr], insert a new paragraph after the existing paragraph 8:

A function or variable of static storage duration reflected by T [dcl.type.reflexpr] is odr-used by the specialization std::experimental::reflect::get_pointer<T> [21.12.4.9, 21.12.4.17], as if by taking the address of an id-expression nominating the function or variable.

2 In C++ [basic.def.odr], insert a new bullet (12.2.3) after (12.2.2):

or

or a type implementing std::experimental::reflect::Object [21.12.3.1], as long as all operations [21.12.4] on this type yield the same constant expression results.

6.7 Types

6.7.1 Fundamental types

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

An expression of type cv void shall be used only as an expression statement (9.2), as an operand of a comma expression (8.5.19), as a second or third operand of ?: (8.5.16), 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

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

8.4 Primary expressions

8.4.5 Lambda expressions

8.4.5.2 Captures

1 In C++ [expr.prim.lambda.capture], apply the following change to paragraph 7:

If an expression potentially references a local entity within a declarative region in which it is odr-
usable, and the expression would be potentially evaluated if the effect of any enclosing typeid
expressions (8.5.1.8) or use of a reflexpr-specifier (10.1.7.6) were ignored, the entity is said to
be implicitly captured by each intervening lambda-expression with an associated capture-default
that does not explicitly capture it.

2 In C++ [expr.prim.lambda.capture], apply the following change to paragraph 11:

Every id-expression within the compound-statement of a lambda-expression that is an odr-use
(6.2) 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.

8.5 Compound expressions

8.5.1 Postfix expressions

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

postfix-expression:
  primary-expression
  postfix-expression [ expr-or-braced-init-list ]
  postfix-expression ( expression-listopt )
  function-call-expression
  simple-type-specifier ( expression-listopt )
  typename-specifier ( expression-listopt )
  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-listopt )
8.6 **Constant expressions**

In C++ `expr.const`, insert a new bullet 8.5, adjusting the subsequent bullet numbers:

An expression is potentially constant evaluated if it is:

[...] (8.5)

| a reflexpr-operand. |
9 Statements

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

10.1 Specifiers

10.1.7 Type specifiers

10.1.7.2 Simple type specifiers

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

The simple type specifiers are

\[
\text{simple-type-specifier:}
\]
\[
\text{nested-name-specifier}_\text{opt} \text{ type-name}
\]
\[
\text{nested-name-specifier template simple-template-id}
\]
\[
\text{nested-name-specifier}_\text{opt} \text{ template-name}
\]

\[
\text{char}
\]
\[
\text{char16_t}
\]
\[
\text{char32_t}
\]
\[
\text{wchar_t}
\]
\[
\text{bool}
\]
\[
\text{short}
\]
\[
\text{int}
\]
\[
\text{long}
\]
\[
\text{signed}
\]
\[
\text{unsigned}
\]
\[
\text{float}
\]
\[
\text{double}
\]
\[
\text{void}
\]
\[
\text{auto}
\]
\[
\text{decltype-specifier}
\]
\[
\text{reflexpr-specifier}
\]

\[
\text{type-name:}
\]
\[
\text{class-name}
\]
\[
\text{enum-name}
\]
\[
\text{typedef-name}
\]
\[
\text{simple-template-id}
\]

\[
\text{decltype-specifier:}
\]
\[
\text{decltype ( expression )}
\]
\[
\text{decltype ( auto )}
\]

\[
\text{reflexpr-specifier:}
\]
\[
\text{reflexpr ( reflexpr-operand )}
\]

\[
\text{reflexpr-operand:}
\]
\[
::
\]
\[
\text{type-id}
\]
\[
\text{nested-name-specifier}_\text{opt} \text{ namespace-name}
\]
\[
\text{id-expression}
\]
\[
( \text{expression} )
\]
\[
\text{function-call-expression}
\]
\[
\text{functional-type-conv-expression}
\]

...
The other simple-type-specifiers specify either a previously-declared type, a type determined from an expression, a reflection meta-object type (10.1.7.6), or one of the fundamental types (6.7.1).

2 Append the following row to Table 11:

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

3 At the end of 10.1.7.2, insert the following paragraph:

For a reflexpr-operand \(x\), the type denoted by \(\text{reflexpr}(x)\) is a type that satisfies the constraints laid out in 10.1.7.6.

10.1.7.6 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 \(T\) (21.12.4). The operand to the reflexpr-specifier shall be a type, namespace, enumerator, variable, structured binding, data member, function parameter, captured entity, parenthesized expression, function-call-expression or functional-type-conv-expression. Any such \(T\) satisfies the requirements of \(\text{reflect::Object}\) (21.12.3) and other reflect concepts, depending on the operand. A type satisfying the requirements of \(\text{reflect::Object}\) is called a meta-object type. A meta-object type is an unnamed, incomplete namespace-scope class type (Clause 12).

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;
};
```

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

5 Zero or more successive applications of type transformations that yield meta-object types (21.12.4) 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.

6 [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

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 (21.12.4) 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.
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 un-
qualified 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 name lookup, as shown in
Table 12.

Note 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 require-
ments for being a reflexpr-operand.

Note 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 (16.3.1.1.2). Otherwise, the postfix-expression shall
name a function that is the unique result of overload resolution.

Note C: [Note: The usual disambiguation between function-style cast and a type-id (11.2) 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] — end note]
Any other reflexpr-operand renders the program ill-formed.

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

If the reflexpr-operand designates a name whose declaration is enclosed in a block scope (6.3.3)
or function parameter scope (6.3.4) 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 (8.2.3). 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. Reflecting directly or indirectly upon std::align_val_t
6.6.4.4 without inclusion of the header <new> is ill-formed; no diagnostic required.
Table 12 — Reflect concept (21.12.3) that the type specified by a reflect-specifier satisfies, for a given reflect-operand.

<table>
<thead>
<tr>
<th>Category</th>
<th>identifier or simple-template-id kind</th>
<th>reflect::Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>class-name designating a union</td>
<td>reflect::Record</td>
</tr>
<tr>
<td></td>
<td>class-name designating a closure type</td>
<td>reflect::Lambda</td>
</tr>
<tr>
<td></td>
<td>class-name designating a non-union class</td>
<td>reflect::Class</td>
</tr>
<tr>
<td></td>
<td>enum-name</td>
<td>reflect::Enum</td>
</tr>
<tr>
<td></td>
<td>template type-parameter</td>
<td>both reflect::Type and reflect::Alias</td>
</tr>
<tr>
<td></td>
<td>decltype-specifier</td>
<td>both reflect::Type and reflect::Alias</td>
</tr>
<tr>
<td></td>
<td>type-name introduced by a using-declaration</td>
<td>reflect::Type, reflect::Alias, and reflect::ScopeMember</td>
</tr>
<tr>
<td></td>
<td>any other typedef-name</td>
<td>both reflect::Type and reflect::Alias</td>
</tr>
<tr>
<td></td>
<td>any other type-id</td>
<td>reflect::Type</td>
</tr>
<tr>
<td><strong>Namespace</strong></td>
<td>namespace-alias</td>
<td>both reflect::Namespace and reflect::Alias</td>
</tr>
<tr>
<td></td>
<td>any other namespace-name</td>
<td>reflect::Namespace</td>
</tr>
<tr>
<td><strong>Expression</strong></td>
<td>the name of a data member</td>
<td>reflect::Variable</td>
</tr>
<tr>
<td></td>
<td>the name of a variable or structured binding</td>
<td>reflect::Variable</td>
</tr>
<tr>
<td></td>
<td>the name of an enumerator</td>
<td>reflect::Enumerator</td>
</tr>
<tr>
<td></td>
<td>the name of a function parameter</td>
<td>reflect::FunctionParameter</td>
</tr>
<tr>
<td></td>
<td>the name of a captured entity</td>
<td>reflect::LambdaCapture</td>
</tr>
<tr>
<td></td>
<td>parenthesized expression: see Note A, below</td>
<td>reflect::ParenthesizedExpression</td>
</tr>
<tr>
<td></td>
<td>function-call-expression: see Note B, below</td>
<td>reflect::FunctionCallExpression</td>
</tr>
<tr>
<td></td>
<td>functional-type-conv-expression: see Note C, below</td>
<td>reflect::FunctionalTypeConversion</td>
</tr>
</tbody>
</table>
11 Declarators [dcl.decl]

11.1 Type names [dcl.name]

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

To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `new`, `delete`, `typeid`, `or reflexivity`, the name of a type shall be specified.
12 Classes

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

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

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

17.7 Name resolution

17.7.2 Dependent names

17.7.2.1 Dependent types

1 In C++ [temp.dep.type], apply the following changes to paragraph 9:

A type is dependent if it is

— denoted by 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 is a pack expansion [Note: This includes an injected-class-name (Clause 12) of a class template used without a `template-argument-list`. — end note], or

— denoted by `decltype(expression)`, where `expression` is type-dependent (17.7.2.2), or

— denoted by `refexpr(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

— denoted by `refexpr(operand)`, where `operand` designates a dependent type or a member of an unknown specialization or a value-dependent constant expression.
18 Exception handling

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

20.5 Library-wide requirements

20.5.1 Library contents and organization

20.5.1.2 Headers

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

20.5.1.3 Freestanding implementations

Add `<experimental/reflect>` to Table 19 – C++ headers for freestanding implementations.
21 Language support library
[language.support]

1 Add a new subclause 21.12 titled 'Static reflection' as follows:

21.12 Static reflection
[reflect]
21.12.1 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 (21.12.3) 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 21.12.4.

21.12.2 Header <experimental/reflect> synopsis
[reflect.synopsis]

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

    // 21.12.3 Concepts for meta-object types
    template <class T> concept Object = see below; // refines Object
    template <class T> concept ObjectSequence = see below; // refines Object
    template <class T> concept TemplateParameterScope = see below; // refines Scope
    template <class T> concept Named = see below; // refines Object
    template <class T> concept Alias = see below; // refines Named and ScopeMember
    template <class T> concept RecordMember = see below; // refines ScopeMember
    template <class T> concept Enumerator = see below; // refines Constant
    template <class T> concept Variable = see below; // refines Typed and ScopeMember
    template <class T> concept ScopeMember = see below; // refines Named
    template <class T> concept Typed = see below; // refines Object
    template <class T> concept Namespace = see below; // refines Named and Scope
    template <class T> concept GlobalScope = see below; // refines Namespace
    template <class T> concept Class = see below; // refines Record
    template <class T> concept Enum = see below; // refines Type, Scope, and
      // ScopeMember
    template <class T> concept Record = see below; // refines Type, Scope, and
      // ScopeMember
    template <class T> concept Scope = see below; // refines Object
    template <class T> concept Type = see below; // refines Named
    template <class T> concept Constant = see below; // refines Typed and ScopeMember
    template <class T> concept Base = see below; // refines Object
    template <class T> concept FunctionParameter = see below; // refines Typed and
      // ScopeMember
    template <class T> concept Callable = see below; // refines Scope and ScopeMember
    template <class T> concept Expression = see below; // refines Object
    template <class T> concept ParenthesizedExpression = see below; // refines Expression
    template <class T> concept FunctionCallExpression = see below; // refines Expression
  }
}
```

§ 21.12.2 © ISO/IEC 2019 – All rights reserved 25
template <class T> concept FunctionalTypeConversion = see below;  // refines Expression
template <class T> concept Function = see below;  // refines Typed and Callable
template <class T> concept MemberFunction = see below;  // refines RecordMember and Function
template <class T> concept SpecialMemberFunction = see below;  // refines RecordMember
template <class T> concept Constructor = see below;  // refines Callable and RecordMember
template <class T> concept Destructor = see below;  // refines Callable and SpecialMemberFunction
template <class T> concept Operator = see below;  // refines Function
template <class T> concept ConversionOperator = see below;  // refines MemberFunction and Operator
template <class T> concept Lambda = see below;  // refines Type and Scope
template <class T> concept LambdaCapture = see below;  // refines Variable

// 21.12.4 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> || Callable<T>
   constexpr auto is_implicit_v = is_implicit<T>::value;

template <class T>
requires Class<T> || MemberFunction<T>
   constexpr auto is_static_v = is_static<T>::value;

template <class T>
requires Constructor<T> || ConversionOperator<T>
   constexpr auto is_final_v = is_final<T>::value;

template <class T>
requires Variable<T> || Callable<T>
   constexpr auto is_virtual_v = is_virtual<T>::value;

template <class T>
requires Base<T> || MemberFunction<T> || Destructor<T>

§ 21.12.2  © ISO/IEC 2019 – All rights reserved
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;

// 21.12.4.1 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;

// 21.12.4.2 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 = unpack_sequence<Tpl, T>::type;

// 21.12.4.3 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;

// 21.12.4.4 Alias operations
template <Alias T> struct get_aliased;

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

// 21.12.4.5 Type operations

§ 21.12.2 © ISO/IEC 2019 – All rights reserved
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;

// 21.12.4.6 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;

// 21.12.4.7 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;

// 21.12.4.8 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;

// 21.12.4.9 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;

// 21.12.4.10 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;

// 21.12.4.11 Namespace operations
template <Namespace T> struct is_inline<T>;

// 21.12.4.12 FunctionParameter operations
template <FunctionParameter T> struct has_default_argument;

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

// 21.12.4.13 Callable operations
template <Callable T> struct is_vararg;
template <Callable T> struct is constexpr<T>;
template <Callable T> struct is noexcept<T>;
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;

// 21.12.4.14 ParenthesizedExpression operations
template <ParenthesizedExpression T> struct get_subexpression;

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

// 21.12.4.15 FunctionCallExpression operations
template <FunctionCallExpression T> struct get Callable;

template <FunctionCallExpression T>
using get Callable_t = typename get Callable<T>::type;

// 21.12.4.16 FunctionalTypeConversion operations
template <FunctionalTypeConversion T> struct get constructor;

template <FunctionalTypeConversion T>
using get constructor_t = typename get constructor<T>::type;

// 21.12.4.17 Function operations
template <Function T> struct get_pointer<T>;

// 21.12.4.18 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 lvalue ref qualifier;
template <MemberFunction T> struct has rvalue ref qualifier;
template <MemberFunction T> struct is virtual<T>;
template <MemberFunction T> struct is pure virtual<T>;

§ 21.12.2  © ISO/IEC 2019 – All rights reserved 30
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;

// 21.12.4.19 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;

// 21.12.4.20 Constructor operations
template <Constructor T> struct is_explicit<T>;

// 21.12.4.21 Destructor operations
template <Destructor T> struct is_virtual<T>;
template <Destructor T> struct is_pure_virtual<T>;

// 21.12.4.22 ConversionOperator operations
template <ConversionOperator T> struct is_explicit<T>;

// 21.12.4.23 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;

// 21.12.4.24 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>
```cpp
constexpr auto is_init_capture_v = is_init_capture<T>::value;
}
```

// inline namespace v1

// namespace std::experimental::reflect

### 21.12.3 Concepts for meta-object types

The operations on meta-object types defined here require meta-object types to satisfy certain concepts (17.6.8). 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::reflect::is_enum operates on meta-object types. — end note]

### 21.12.3.1 Concept Object

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

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.

### 21.12.3.2 Concept ObjectSequence

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

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

### 21.12.3.3 Concept TemplateParameterScope

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

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 (6.3.9), providing a scope to the Alias reflecting a template type-parameter. — end note]

### 21.12.3.4 Concept Named

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

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

### 21.12.3.5 Concept Alias

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

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:

```cpp
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\langle M_{X\_t}\rangle;
using C_{i\_t} = reflexpr(C::i);
using C_{i\_scope\_t} = get_scope_t\langle C_{i\_t}\rangle;

The scope reflected by \( M_{X\_scope\_t} \) is \( M \), not \( N \); the scope reflected by \( C_{i\_scope\_t} \) is \( C \), not \( B \).

Type transformations \((21.12.4)\) never yield an Alias; instead, they yield the aliased entity.

21.12.3.6 Concept \texttt{RecordMember} \hfill [reflect.concepts.recordmember]

\[
\text{template <class T> concept RecordMember = ScopeMember\langle T\rangle \&\& \text{ see below};}
\]

\texttt{RecordMember\langle T\rangle} is true if and only if \( T \) reflects a member-declaration.

21.12.3.7 Concept \texttt{Enumerator} \hfill [reflect.concepts.enumerator]

\[
\text{template <class T> concept Enumerator = Constant\langle T\rangle \&\& \text{ see below};}
\]

\texttt{Enumerator\langle T\rangle} is true if and only if \( T \) reflects an enumerator. \([\text{Note: The scope of an Enumerator is its type also for enumerations that are unscoped enumeration types. — end note}]\)

21.12.3.8 Concept \texttt{Variable} \hfill [reflect.concepts.variable]

\[
\text{template <class T> concept Variable = Typed\langle T\rangle \&\& \text{ ScopeMember\langle T\rangle \text{ see below};}}
\]

\texttt{Variable\langle T\rangle} is true if and only if \( T \) reflects a variable or data member.

21.12.3.9 Concept \texttt{ScopeMember} \hfill [reflect.concepts.scopemember]

\[
\text{template <class T> concept ScopeMember = Named\langle T\rangle \&\& \text{ see below};}
\]

\texttt{ScopeMember\langle T\rangle} is true if and only if \( T \) satisfies \texttt{RecordMember}, \texttt{Enumerator}, or \texttt{Variable}, or if \( T \) reflects a namespace that is not the global namespace. \([\text{Note: The scope of members of an unnamed union is the unnamed union; the scope of enumerators is their type. — end note}]\)

21.12.3.10 Concept \texttt{Typed} \hfill [reflect.concepts.typed]

\[
\text{template <class T> concept Typed = Object\langle T\rangle \&\& \text{ see below};}
\]

\texttt{Typed\langle T\rangle} is true if and only if \( T \) reflects an entity with a type.

21.12.3.11 Concept \texttt{Namespace} \hfill [reflect.concepts.namespace]

\[
\text{template <class T> concept Namespace = Named\langle T\rangle \&\& \text{ Scope\langle T\rangle \&\& \text{ see below};}}
\]

\texttt{Namespace\langle T\rangle} is true if and only if \( T \) reflects a namespace (including the global namespace). \([\text{Note: Any such } T \text{ that does not reflect the global namespace also satisfies ScopeMember. — end note}]\)

21.12.3.12 Concept \texttt{GlobalScope} \hfill [reflect.concepts.globalscope]

\[
\text{template <class T> concept GlobalScope = Namespace\langle T\rangle \&\& \text{ see below};}
\]

\texttt{GlobalScope\langle T\rangle} is true if and only if \( T \) reflects the global namespace. \([\text{Note: Any such } T \text{ does not satisfy ScopeMember. — end note}]\)

21.12.3.13 Concept \texttt{Class} \hfill [reflect.concepts.class]

\[
\text{template <class T> concept Class = Record\langle T\rangle \&\& \text{ see below};}
\]

\texttt{Class\langle T\rangle} is true if and only if \( T \) reflects a non-union class type.
21.12.3.14 Concept Enum

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

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

21.12.3.15 Concept Record

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

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

21.12.3.16 Concept Scope

```cpp
template <class T> concept 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 or closure type. [Note: Any such `T` that does not reflect the global namespace also satisfies `ScopeMember`. — end note]

21.12.3.17 Concept Type

```cpp
template <class T> concept 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]

21.12.3.18 Concept Constant

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

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

21.12.3.19 Concept Base

```cpp
template <class T> concept 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`.

21.12.3.20 Concept FunctionParameter

```cpp
template <class T> concept 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]

21.12.3.21 Concept Callable

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

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

21.12.3.22 Concept Expression

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

1. `Expression<T>` is true if and only if `T` reflects an expression (Clause 8).
21.12.3.23 Concept ParenthesizedExpression [reflect.concepts.expr.paren]

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

ParenthesizedExpression<T> is true if and only if T reflects a parenthesized expression (8.3.4).

21.12.3.24 Concept FunctionCallExpression [reflect.concepts.expr.fctcall]

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

FunctionCallExpression<T> is true if and only if T reflects a function-call-expression (8.5.1.2).

21.12.3.25 Concept FunctionalTypeConversion [reflect.concepts.expr.type.fctconv]

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

FunctionalTypeConversion<T> is true if and only if T reflects a functional-type-conv-expression (8.5.1.3).

21.12.3.26 Concept Function [reflect.concepts.fct]

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

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

21.12.3.27 Concept MemberFunction [reflect.concepts.memfct]

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

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

21.12.3.28 Concept SpecialMemberFunction [reflect.concepts.specialfct]

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

SpecialMemberFunction<T> is true if and only if T reflects a special member function (Clause 15).

21.12.3.29 Concept Constructor [reflect.concepts.ctor]

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

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

21.12.3.30 Concept Destructor [reflect.concepts.dtor]

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

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

21.12.3.31 Concept Operator [reflect.concepts.oper]

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

Operator<T> is true if and only if T reflects an operator function (16.5) or a conversion function (15.3.2). [Note: Some types that satisfy Operator also satisfy MemberFunction or SpecialMemberFunction. — end note]

21.12.3.32 Concept ConversionOperator [reflect.concepts.convfct]

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

ConversionOperator<T> is true if and only if T reflects a conversion function (15.3.2).
21.12.3.33 Concept Lambda

\[\text{template <class } T \text{> concept Lambda = Type}<T> \&\& \text{Scope}<T> \&\& \text{see below;}\]

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

21.12.3.34 Concept LambdaCapture

\[\text{template <class } T \text{> concept LambdaCapture = Variable}<T> \&\& \text{see below;}\]

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]

21.12.4 Meta-object operations

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 (8.6) or a type.

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:

\[\text{struct } X \{};\]
\[\text{X x;}\]
\[\text{using } x_t = \text{get_type_t}<\text{reflexpr}(x)>;\]

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 (21.12.3.5).

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:

\[\text{void } \text{func}(\text{int } a);\]
\[\text{auto } x1 = \text{get_name_v}<\text{get_element_v}<0, \text{get_parameters_v}<\text{reflexpr}(\text{func}(42))>>;\]
\[\text{void } \text{func}(\text{int } b);\]
\[\text{auto } x2 = \text{get_name_v}<\text{get_element_v}<0, \text{get_parameters_v}<\text{reflexpr}(\text{func}(42))>>; // ill-formed,}
\[\text{ // no diagnostic required}\]

—end example]

21.12.4.1 Object operations

\[\text{template <Object } T1, \text{ Object } T2> \text{ struct reflects_same;}\]

1 All specializations of \text{reflects_same}<T1, T2> shall meet the BinaryTypeTrait requirements (23.15.1), with a base characteristic of true_type if

\[\begin{align*}
\text{(1.1)} & \quad \text{T1 and T2 reflect the same alias, or} \\
\text{(1.2)} & \quad \text{neither T1 nor T2 reflect an alias and T1 and T2 reflect the same entity;}
\end{align*}\]

otherwise, with a base characteristic of false_type.

2 [Example: With

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

All specializations of above templates shall meet the UnaryTypeTrait requirements (23.15.1) with a base characteristic of integral_constant<uint_least32_t> and a value of the presumed line number (19.8) (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;

All specializations of get_source_file_name<T> shall meet the UnaryTypeTrait requirements (23.15.1) with a static data member named value of type constexpr 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 (19.8) of a declaration of the entity or typedef described by T.

21.12.4.2 ObjectSequence operations

template <ObjectSequence T> struct get_size;

All specializations of get_size<T> shall meet the UnaryTypeTrait requirements (23.15.1) 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;

All specializations of get_element<I, T> shall meet the TransformationTrait requirements (23.15.1). The nested type named type corresponds to the Ith element Object in T, where the indexing is zero-based.

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

All specializations of unpack_sequence<Tpl, T> shall meet the TransformationTrait requirements (23.15.1). The nested type named type designates the template Tpl specialized with the element Objects in T.

21.12.4.3 Named operations

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

All specializations of is_unnamed<T> shall meet the UnaryTypeTrait requirements (23.15.1) with a base characteristic as specified below.

All specializations of get_name<T> and get_display_name<T> shall meet the UnaryTypeTrait requirements (23.15.1) with a static data member named value of type const char (&)[N], referencing a static, constant expression character array (NMTBS) of length N, as if declared as static constexpr char STR[N] = ...; The value of the NMTBS is the empty string if T reflects an unnamed entity; otherwise the value is implementation defined.

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:

— for T reflecting an alias, the unqualified name of the aliasing declaration;
— for T reflecting an unnamed entity, the empty string.
— 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;

— for \( T \) reflecting a cv-unqualified type that is
  — a type-parameter, the identifier introduced by the type-parameter;
  — a class type, its class-name;
  — an enumeration type, its enum-name;
  — a fundamental type other than std::nullptr_t, the name stated in the "Type" column of 11 in (10.1.7.2);

— for \( T \) reflecting
  — a namespace, its namespace-name;
  — a variable, enumerator, data member, or function, its unqualified name;
  — a function parameter, its name;
  — a constructor, the injected-class-name of its class;
  — a destructor, the injected-class-name of its class, prefixed by the character ‘~’;
  — an operator function, the operator element of the relevant operator-function-id;
  — a literal operator, the s-char-sequence "\" followed by the literal suffix identifier of the operator’s literal-operator-id;
  — a conversion function, the same characters as get_name_v<\( R \)>, with \( R \) reflecting the type represented by the conversion-type-id.

— In all other cases, the string’s value is the empty string.

[Note: With

```cpp
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]

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

Subsequent specializations of get_name<T> on the same reflected function parameter can render the program ill-formed, no diagnostic required [21.12.4].

### 21.12.4.4 Alias operations

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

All specializations of get_aliased<T> shall meet the TransformationTrait requirements (23.15.1).

The nested type named type is the Named meta-object type reflecting

1. the redefined name, if \( T \) reflects an alias;
2. the template specialization’s template argument type, if \( T \) reflects a template type-parameter;
3. the original declaration introduced by a using declaration;
4. the aliased namespace of a namespace-alias;
5. the type denoted by the decltype-specifier.

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

[Example: For

```cpp
using i0 = int; using i1 = i0;
get_aliased_t<reflexpr(i1)> reflects int. —end example]
```
21.12.4.5 Type operations [reflect.ops.type]

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

1. All specializations of get_type<T> shall meet the TransformationTrait requirements (23.15.1). The nested type named type is the Type reflecting the type of the entity reflected by T.

2. [Example: For

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

   ```cpp
   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

   ```cpp
   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)>>; // will reflect int[17][]
   using C2 = get_type_t<reflexpr(C::arr)>; // will reflect int[17][42]. — end note]
```

5. template <Type T> struct get_reflected_type;

6. All specializations of get_reflected_type<T> shall meet the TransformationTrait requirements (23.15.1). The nested type named type is the type reflected by T.

7. [Example: For

   ```cpp
   using int_m = reflexpr(int);
   ```

   ```cpp
   get_reflected_type_t<int_m> x; // x is of type int
   ```

   — end example]

8. 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 (23.15.1). 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.

8. 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 (23.15.1). 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.

21.12.4.6 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 (17.9.2).

2. template <ScopeMember T> struct get_scope;

2. All specializations of get_scope<T> shall meet the TransformationTrait requirements (23.15.1). The nested type named type is the Scope reflecting a scope S. With ST being the scope of the declaration of the entity, typedef 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.

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

All specializations of these partial template specializations shall meet the UnaryTypeTrait requirements (23.15.1). 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.

### 21.12.4.7 Record operations

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 (17.9.2). Members introduced by using-declarations (10.3.3) 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 21.12.4. A member injected into a derived class may have different access. — end note]

```cpp
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 (23.15.1). The nested type named type designates a meta-object type satisfying ObjectSequence, containing elements which satisfy RecordMember and reflect the following subset of direct non-template members of the class reflected by T:

1. for get_data_members (get_member_functions), all data (function, including constructor and destructor) members.
2. for get_public_data_members (get_public_member_functions), all public data (function, including constructor and destructor) members;
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:

```cpp
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;

All specializations of these templates shall meet the TransformationTrait requirements
(23.15.1). The nested type named type Designates a meta-object type satisfying ObjectSequence,
containing elements which satisfy RecordMember and reflect the following subset of function
members of the class reflected by T:

— for get_constructors, all constructors,
— for get_operators, all conversion functions (15.3.2) and operator functions (16.5).

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

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

template <Record T> struct get_destructor;

All specializations of get_destructor<T> shall meet the TransformationTrait requirements
(23.15.1). The nested type named type Designates a Destructor type that reflects the destructor of the class reflected by T.

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;

All specializations of these templates shall meet the TransformationTrait requirements
(23.15.1). The nested type named type Designates a meta-object type satisfying ObjectSequence,
containing elements which satisfy Type and reflect the following subset of types declared in
the class reflected by T:

— for get_public_member_types, all public nested class types, enum types, or member
typedefs;
— 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;
— for get_member_types, all nested class types, enum types, or member typedefs.

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

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;

All specializations of these templates shall meet the TransformationTrait requirements
(23.15.1). 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:

— for get_public_base_classes, all public direct base classes;
— 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;
— for get_base_classes, all direct base classes.

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

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

template <Class T> struct is_final<T>;

All specializations of is_final<T> shall meet the UnaryTypeTrait requirements (23.15.1). 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.
21.12.4.8 Enum operations

```
template <Enum T> struct is_scoped_enum;
1
All specializations of is_scoped_enum<T> shall meet the UnaryTypeTrait requirements (23.15.1).
If T reflects a scoped enumeration, the base characteristic of the respective template special-
ization 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
(23.15.1). 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.
```

```
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 (17.9.2).
```

```
template <Enum T> struct get_underlying_type;
3
All specializations of get_underlying_type<T> shall meet the TransformationTrait require-
ments (23.15.1). The nested type named type designates a meta-object type that reflects
the underlying type (10.2) of the enumeration reflected by T.
```

```
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 (17.9.2).
```

21.12.4.9 Value operations

```
template <Constant T> struct get_constant;
1
All specializations of get_constant<T> shall meet the UnaryTypeTrait requirements (23.15.1).
The type and value of the static data member named value are those of the constant expres-
sion of the constant reflected by T.
```

```
template <Variable T> struct is_constexpr<T>;
2
All specializations of is_constexpr<T> shall meet the UnaryTypeTrait requirements (23.15.1).
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>;
3
All specializations of is_static<T> and is_thread_local<T> shall meet the UnaryTypeTrait
requirements (23.15.1). 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 spe-
cialization 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 ini-
tialization or has not been initialized with an object of static storage duration, the special-
isation of get_pointer<T> has no member named type. Otherwise, the specialization of
get_pointer<T> shall meet the UnaryTypeTrait requirements (23.15.1), 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.
```

§ 21.12.4.9 © ISO/IEC 2019 – All rights reserved 42
21.12.4.10 Base operations
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 (17.9.2).

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

```
All specializations of get_class<T> shall meet the TransformationTrait requirements (23.15.1).
The nested type named type designates reflexpr(X), where X is the base class (without retaining possible Alias properties, see 21.12.3.5) reflected by T.
```

```cpp
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>;
```

```
All specializations of these templates shall meet the UnaryTypeTrait requirements (23.15.1).
If T reflects a direct base class with the virtual specifier (for is_virtual), with the public specifier or with an assumed (see 14.2) 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.
```

21.12.4.11 Namespace operations

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

```
All specializations of is_inline<T> shall meet the UnaryTypeTrait requirements (23.15.1).
If T reflects an inline namespace, the base characteristic of the template specialization is true_type, otherwise it is false_type.
```

21.12.4.12 FunctionParameter operations

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

```
All specializations of this template shall meet the UnaryTypeTrait requirements (23.15.1).
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.
```

Remarks: Subsequent specializations of has_default_argument<T> on the same reflected function parameter can render the program ill-formed, no diagnostic required (21.12.4).

21.12.4.13 Callable operations

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

```
All specializations of this template shall meet the TransformationTrait requirements (23.15.1).
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 (11.3.5) 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.
```

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

```
All specializations of these templates shall meet the UnaryTypeTrait requirements (23.15.1).
If their template parameter reflects an entity with an ellipsis terminating the parameter-declaration-clause (11.3.5) (for is_vararg), or an entity that is (where applicable implicitly or explicitly) declared as constexpr (for is_constexpr), as noexcept (for is_noexcept), as an inline function (10.1.6) (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.
```

Remarks: Subsequent specializations of is_inline<T> on the same reflected function can render the program ill-formed, no diagnostic required (21.12.4).
21.12.4.14 ParenthesizedExpression operations

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

All specializations of `get_subexpression<T>` shall meet the TransformationTrait requirements (23.15.1). The nested type named `type` is the Expression type reflecting the expression `E` of the parenthesized expression `(E)` reflected by `T`.

21.12.4.15 FunctionCallExpression operations

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

All specializations of `get_callable<T>` shall meet the TransformationTrait requirements (23.15.1). The nested type named `type` is the Callable type reflecting the function invoked by the `function-call-expression` which is reflected by `T`.

21.12.4.16 FunctionalTypeConversion operations

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

All specializations of `get_constructor<T>` shall meet the TransformationTrait requirements (23.15.1). For a `functional-type-conv-expression` reflected by `T`, let `S` be the type specified by the type conversion (8.5.1.3). 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>::type` has no member named `type`. [Note: For instance fundamental types (6.7.1) do not have constructors. — end note] [Note: `get_constructor_t<get_subexpression<reflexpr((T(T())))>>` is ill-formed because the `functional-type-conv` expression does not perform overload resolution for a constructor (). — end note]

21.12.4.17 Function operations

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

All specializations of `get_pointer<T>` shall meet the UnaryTypeTrait requirements (23.15.1), 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.

21.12.4.18 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>;
```

All specializations of these templates shall meet the UnaryTypeTrait requirements (23.15.1). 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`.

§ 21.12.4.18 © ISO/IEC 2019 – All rights reserved 44
21.12.4.19 SpecialMemberFunction operations

template <SpecialMemberFunction T> struct is_implicitly_declared;
template <SpecialMemberFunction T> struct is_defaulted;
1
All specializations of these templates shall meet the UnaryTypeTrait requirements (23.15.1).
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.

21.12.4.20 Constructor operations

template <Constructor T> struct is_explicit<T>;
1
All specializations of this template shall meet the UnaryTypeTrait requirements (23.15.1).
If the template parameter reflects an explicit constructor, the base characteristic of the respective template specialization is true_type, otherwise it is false_type.

21.12.4.21 Destructor operations

template <Destructor T> struct is_virtual<T>;
template <Destructor T> struct is_pure_virtual<T>;
1
All specializations of these templates shall meet the UnaryTypeTrait requirements (23.15.1).
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.

21.12.4.22 ConversionOperator operations

template <ConversionOperator T> struct is_explicit<T>;
1
All specializations of is_explicit<T> shall meet the UnaryTypeTrait requirements (23.15.1).
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.

21.12.4.23 Lambda operations

template <Lambda T> struct get_captures;
1
All specializations of get_captures<T> shall meet the TransformationTrait requirements (23.15.1). 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 (23.15.1).
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 (23.15.1). 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.
21.12.4.24 LambdaCapture operations

```
template <LambdaCapture T> struct is_explicitly_captured;
1
All specializations of is_explicitly_captured<T> shall meet the UnaryTypeTrait requirements (23.15.1). 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 (23.15.1). 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 C  (informative)
Compatibility

C.5  C++ and ISO C++ 2017

C.5.1  Clause 5: lexical conventions

In C++ [diff.cpp17.lex], modify the first paragraph:

1  Affected subclause: 5.11
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 (17.6.8). The reflexpr keyword is added to introduce meta-data through a reflexpr-specifier.
Effect on original feature: Valid ISO C++ 2017 code using concept or requires or reflexpr as an identifier is not valid in this International Standard.