Working Draft, C++ Extensions for Reflection

Note: this is an early draft. It’s known to be incompleat and incorrekt, and it has lots of bad formatting.
## Contents

1 Scope

2 Normative references

3 Terms and definitions

4 General
   4.1 Implementation compliance ................................................. 4
   4.2 Namespaces and headers .................................................. 4
   4.3 Acknowledgements .......................................................... 4

5 Lexical conventions
   5.11 Keywords ................................................................. 5

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

7 Standard conversions

8 Expressions
   8.4 Primary expressions ....................................................... 8
   8.5 Compound expressions .................................................... 8

9 Statements

10 Declarations
   10.1 Specifiers ................................................................. 11

11 Declarators
   11.1 Type names ............................................................... 15

12 Classes

13 Derived classes

14 Member access control

15 Special member functions

16 Overloading

17 Templates
   17.7 Name resolution .......................................................... 21

18 Exception handling

19 Preprocessing directives

© ISO/IEC 2018 – All rights reserved
20 Library introduction
   20.5 Library-wide requirements ........................................... 24

21 Language support library
   21.12 Static reflection .......................................................... 25
1 Scope

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.

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. 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

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++*

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

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 Technical Specification 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:

(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

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

| <experimental/reflect> |

4.3 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

In C++ [lex.key], add the keyword `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 static variable 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):

— 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 [expr]

8.4 Primary expressions [expr.prim]
8.4.5 Lambda expressions [expr.prim.lambda]
8.4.5.2 Captures [expr.prim.lambda.capture]

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 [expr.compound]
8.5.1 Postfix expressions [expr.post]

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

postfix-expression:
  primary-expression
  postfix-expression [ expr-or-braced-init-list ]
  postfix-expression ( expression-list_opt )
  function-call-expression
  simple-type-specifier ( expression-list_opt )
  typename specifier ( expression-list_opt )
  simple-type-specifier-braced-init-list
  typename specifier-braced-init-list
  functional-type-conv-expression
  postfix-expression . template_opt id-expression
  postfix-expression -> template_opt id-expression
  postfix-expression . pseudo-destructor-name
  postfix-expression -> pseudo-destructor-name
  postfix-expression ++
  postfix-expression --
  dynamic_cast < type-id > ( expression )
  static_cast < type-id > ( expression )
  reinterpret_cast < type-id > ( expression )
  const_cast < type-id > ( expression )
  typeid ( expression )
  typeid ( type-id )
  function-call-expression:
    postfix-expression ( expression-list_opt )
**functional-type-conv-expression:**

- `simple-type-specifier ( expression-list_opt )`
- `typename-specifier ( expression-list_opt )`
- `simple-type-specifier braced-init-list`
- `typename-specifier braced-init-list`

**expression-list:**

- `initializer-list`
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

1 In C++, 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
dcltype-specifier

reflexpr-specifier

```

type-name:

```
class-name
enum-name
typedef-name
simple-template-id
```

dcltype-specifier:

```
dcltype ( expression )
dcltype ( auto )
```

reflexpr-specifier:

```
refexpr ( reflexpr-operand )
```

reflexpr-operand:

```
::
type-id
nested-name-specifier_opt identifier
nested-name-specifier_opt simple-template-id
  ( expression )
function-call-expression
functional-type-conv-expression
```

...
The other simple-type-specifiers specify either a previously-declared type, a type determined from an expression, a reflection meta-object type (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 reflexpr(x) is an implementation-defined 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 section:

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 conversion expression. Any such T satisfies the requirements of reflect::Object (21.12.3) 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 (Clause 12).

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

1. A and B are the same entity or alias,
2. A is a variable or enumerator and B is the type of A,
3. A is an enumeration and B is the underlying type of A,
4. A is a class and B is a member or base class of A,
5. A is a non-template alias that designates the entity B,
6. A is a class nested in B (12.2.5),
7. A is not the global namespace and B is an enclosing namespace of A,
8. B is the parenthesized expression ( A ),
9. A is a lambda capture of the closure type B,
10. A is the closure type of the lambda capture B,
11. A is the type specified by the functional-type-conv-expression B,
12. A is the function selected by overload resolution for a function-call-expression B,
13. A is the return type, parameter type, or function type of the function B, or
14. 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:

```cpp
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 (21.12.4) 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.

[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

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 (21.12.4) 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.

For an operand that is a parenthesized expression (8.4.3), the type satisfies reflect::ParenthesizedExpression. For a parenthesized expression (E), whether or not itself nested inside a parenthesized expression, the expression E shall be either a parenthesized expression, a function-call-expression or a functional-type-conv-expression; otherwise the program is ill-formed.

For an operand of the form function-call-expression, the type satisfies reflect::FunctionCallExpression. 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.

For an operand of the form functional-type-conv-expression (8.5.1.3), the type satisfies reflect::FunctionalTypeConversion. [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]

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.

If the reflexpr-opand designates a type-id not explicitly mentioned in Table 12, the type represented by the reflexpr-specifier satisfies reflect::Type. Any other reflexpr-opand renders the program ill-formed.
Table 12 — reflect concept (21.12.3) that the type specified by a reflexpr-specifier satisfies, for a given
reflexpr-operand identifier or simple-template-id.

<table>
<thead>
<tr>
<th>Category</th>
<th>identifier or simple-template-id kind</th>
<th>reflect Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</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>type-name introduced by a using-declaration</td>
<td>both reflect::Type and reflect::Alias</td>
</tr>
<tr>
<td></td>
<td>any other typedef-name</td>
<td>both reflect::Type and reflect::Alias</td>
</tr>
<tr>
<td>namespace</td>
<td>namespace-alias</td>
<td>both reflect::Namespace and reflect::Alias</td>
</tr>
<tr>
<td></td>
<td>any other namespace-name</td>
<td>both reflect::Namespace and reflect::ScopeMember</td>
</tr>
<tr>
<td>data member</td>
<td>the name of a data member</td>
<td>reflect::Variable</td>
</tr>
<tr>
<td>value</td>
<td>the name of a variable or structured binding that is not a local entity</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 (8.4.5.2)</td>
<td>reflect::LambdaCapture</td>
</tr>
</tbody>
</table>

If the reflexpr-operand designates an entity or alias at block scope (6.3.3) or function prototype scope (6.3.4) and the 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 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.

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

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 reflexpr(operand), where operand designates a dependent type or a member of an unknown specialization.
18 Exception handling

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

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.
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]

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

// 21.12.3 Concepts for meta-object types

template <class T> concept Object;       // refines Object
template <class T> concept ObjectSequence; // refines Object
template <class T> concept Named;        // refines Object
template <class T> concept Alias;        // refines Named and ScopeMember
template <class T> concept RecordMember;  // refines ScopeMember
template <class T> concept Enumerator;   // refines Constant
template <class T> concept Variable;     // refines Named
template <class T> concept Typed;        // refines Named
template <class T> concept Scope;        // refines Object
template <class T> concept Type;         // refines Named and ScopeMember
template <class T> concept Base;         // refines Object
template <class T> concept FunctionParameter; // refines Typed and ScopeMember
template <class T> concept Callable;     // refines Scope and ScopeMember
template <class T> concept Expression;   // refines Object
template <class T> concept ParenthesizedExpression; // refines Expression
template <class T> concept FunctionCallExpression; // refines Expression
template <class T> concept FunctionalTypeConversion; // refines Expression
template <class T> concept Function;     // refines Typed and Callable
template <class T> concept MemberFunction; // refines RecordMember and Function
template <class T> concept SpecialMemberFunction; // refines RecordMember

§ 21.12.2 © ISO/IEC 2018 – All rights reserved 25
template <class T> concept Constructor;  // refines Callable and RecordMember
template <class T> concept Destructor;  // refines Callable and SpecialMemberFunction
template <class T> concept Operator;  // refines Function
template <class T> concept ConversionOperator;  // refines MemberFunction and Operator
template <class T> concept Lambda;  // refines Type and Scope
template <class T> concept LambdaCapture;  // 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>
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;
template <class T>
constexpr auto is_constexpr_v = is_constexpr<T>::value;
template <class T>
constexpr auto is_static_v = is_static<T>::value;
template <class T>
constexpr auto is_final_v = is_final<T>::value;
template <class T>
constexpr auto is_explicit_v = is_explicit<T>::value;
template <class T>
constexpr auto is_inline_v = is_inline<T>::value;
template <class T>
constexpr auto is_virtual_v = is_virtual<T>::value;
template <class T>
constexpr auto is_pure_virtual_v = is_pure_virtual<T>::value;
template <class 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>
    constexpr auto 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
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;

// 21.12.4.6 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_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 get_pointer<T>;

template <Constant T>
  constexpr auto get_constant_v = get_constant<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 get_parameters;
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;
\textbf{template <Callable T>}
\begin{verbatim}
   using get_parameters_t = typename get_parameters<T>::type;
\end{verbatim}
\textbf{template <Callable T>}
\begin{verbatim}
   constexpr auto is_vararg_v = is_vararg<T>::value;
\end{verbatim}
\textbf{template <Callable T>}
\begin{verbatim}
   constexpr auto is_deleted_v = is_deleted<T>::value;
\end{verbatim}

// 21.12.4.14 ParenthesizedExpression operations
\textbf{template <ParenthesizedExpression T> struct get_subexpression;}
\textbf{template <ParenthesizedExpression T>}
\begin{verbatim}
   using get_subexpression_t = typename get_subexpression<T>::type;
\end{verbatim}

// 21.12.4.15 FunctionCallExpression operations
\textbf{template <FunctionCallExpression T> struct get_callable;}
\textbf{template <FunctionCallExpression T>}
\begin{verbatim}
   using get_callable_t = typename get_callable<T>::type;
\end{verbatim}

// 21.12.4.16 FunctionalTypeConversion operations
\textbf{template <FunctionalTypeConversion T> struct get_constructor;}
\textbf{template <FunctionalTypeConversion T>}
\begin{verbatim}
   using get_constructor_t = typename get_constructor<T>::type;
\end{verbatim}

// 21.12.4.17 Function operations
\textbf{template <Function T> struct get_pointer<T>;

// 21.12.4.18 MemberFunction operations
\textbf{template <MemberFunction T> struct is_static<T>;
\textbf{template <MemberFunction T> struct is_const;
\textbf{template <MemberFunction T> struct is_volatile;
\textbf{template <MemberFunction T> struct has_lvalueref_qualifier;
\textbf{template <MemberFunction T> struct has_rvalueref_qualifier;
\textbf{template <MemberFunction T> struct is_virtual<T>;
\textbf{template <MemberFunction T> struct is_pure_virtual<T>;
\textbf{template <MemberFunction T> struct is_override;
\textbf{template <MemberFunction T> struct is_final<T>;
\textbf{template <MemberFunction T>}
\begin{verbatim}
   constexpr auto is_const_v = is_const<T>::value;
\end{verbatim}
\textbf{template <MemberFunction T>}
\begin{verbatim}
   constexpr auto is_volatile_v = is_volatile<T>::value;
\end{verbatim}
\textbf{template <MemberFunction T>}
\begin{verbatim}
   constexpr auto has_lvalueref_qualifier_v = has_lvalueref_qualifier<T>::value;
\end{verbatim}
\textbf{template <MemberFunction T>}
\begin{verbatim}
   constexpr auto has_rvalueref_qualifier_v = has_rvalueref_qualifier<T>::value;
\end{verbatim}
\textbf{template <MemberFunction T>}
\begin{verbatim}
   constexpr auto is_override_v = is_override<T>::value;
\end{verbatim}

// 21.12.4.19 SpecialMemberFunction operations
\textbf{template <SpecialMemberFunction T> struct isimplicitly_declared;
\textbf{template <SpecialMemberFunction T> struct isdefaulted;
\textbf{template <SpecialMemberFunction T>
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.

21.12.3.1 Concept Object

```
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

```
template <class T> concept ObjectSequence = Object<T> && see below;
```
ObjectSequence\textless T\textgreater  is true if and only if \text{T} is a sequence of Objects, generated by a meta-object operation.

21.12.3.3 Concept Named

\texttt{template <\text{class} \text{T}> \text{concept} \text{Named} = \text{Object}\langle \text{T} \rangle \&\& \text{see below};}

\text{T is true if and only if \text{T} has an associated (possibly empty) name.}

21.12.3.4 Concept Alias

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

\text{Alias\langle \text{T} \rangle is true if and only if \text{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. \textbf{[Note: The Scope of an Alias is the scope that the alias was injected into. ---end note]} \textbf{[Example:}}

\begin{verbatim}
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>;
\end{verbatim}

\text{The scope reflected by \text{M_X_scope_t} is \text{M}, not \text{N}. ---end example]\textbf{]}

\text{Except for the type represented by the reflexpr operator, Alias properties resulting from type transformations (21.12.4) are not retained.}

21.12.3.5 Concept RecordMember

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

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

21.12.3.6 Concept Enumerator

\texttt{template <\text{class} \text{T}> \text{concept} \text{Enumerator} = \text{Typed}\langle \text{T} \rangle \&\& \text{ScopeMember}\langle \text{T} \rangle \&\& \text{see below};}

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

21.12.3.7 Concept Variable

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

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

21.12.3.8 Concept ScopeMember

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

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

21.12.3.9 Concept Typed

\texttt{template <\text{class} \text{T}> \text{concept} \text{Typed} = \text{Named}\langle \text{T} \rangle \&\& \text{see below};}

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

\textsection {21.12.3.9}
21.12.3.10 Concept Namespace

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

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

21.12.3.11 Concept GlobalScope

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

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

21.12.3.12 Concept Class

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

\[
\text{Class}<T> \text{ is true if and only if } T \text{ reflects a non-union class type.}
\]

21.12.3.13 Concept Enum

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

\[
\text{Enum}<T> \text{ is true if and only if } T \text{ reflects an enumeration type.}
\]

21.12.3.14 Concept Record

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

\[
\text{Record}<T> \text{ is true if and only if } T \text{ reflects a class type.}
\]

21.12.3.15 Concept Scope

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

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

21.12.3.16 Concept Type

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

\[
\text{Type}<T> \text{ is true if and only if } T \text{ reflects a type.}
\]

21.12.3.17 Concept Constant

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

\[
\text{Constant}<T> \text{ is true if and only if } T \text{ reflects a constant expression (8.6).}
\]

21.12.3.18 Concept Base

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

\[
\text{Base}<T> \text{ is true if and only if } T \text{ reflects a direct base class, as returned by the template get_base_classes.}
\]

21.12.3.19 Concept FunctionParameter

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

\[
\text{FunctionParameter}<T> \text{ is true if and only if } T \text{ reflects a function parameter. \ [Note: The Scope of a FunctionParameter is the Callable to which this parameter appertains.} \text{ — end note]}
\]

\[
\text{[Note: A FunctionParameter does not satisfy Variable, and thus does not offer an interface for getting the pointer to a parameter.} \text{ — end note]}
\]

§ 21.12.3.19 © ISO/IEC 2018 – All rights reserved 33
21.12.3.20  Concept Callable

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

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

21.12.3.21  Concept Expression

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

Expression<T> is true if and only if T reflects an expression (Clause 8).

21.12.3.22  Concept ParenthesizedExpression

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

ParenthesizedExpression<T> is true if and only if T reflects a parenthesized expression (8.4.3).

21.12.3.23  Concept FunctionCallExpression

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

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

21.12.3.24  Concept FunctionalTypeConversion

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

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

21.12.3.25  Concept Function

\[\text{template } \text{<class T> concept Function} = \text{Callable<T> \&\& Typed<T> \&\& see below;}\]

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

21.12.3.26  Concept MemberFunction

\[\text{template } \text{<class T> concept MemberFunction} = \text{RecordMember<T> \&\& Function<T> \&\& see below;}\]

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

21.12.3.27  Concept SpecialMemberFunction

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

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

21.12.3.28  Concept Constructor

\[\text{template } \text{<class T> concept Constructor} = \text{Callable<T> \&\& RecordMember<T> \&\& see below;}\]

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

21.12.3.29  Concept Destructor

\[\text{template } \text{<class T> concept Destructor} = \text{Callable<T> \&\& SpecialMemberFunction<T> \&\& see below;}\]

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

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

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

21.12.3.31 Concept ConversionOperator

\[
\text{template } \langle \text{class } T \rangle \ \text{concept } \text{ConversionOperator} = \text{Operator}<T> \&\& \text{MemberFunction}<T> \&\& \text{see below};
\]

ConversionOperator<\text{T}> is true if and only if \text{T} reflects a conversion function (15.3.2).

21.12.3.32 Concept Lambda

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

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

21.12.3.33 Concept LambdaCapture

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

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

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. \[\text{Example:}\]

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

While get_type_t is specified to be a Type, x_t also satisfies Class. — end example

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. \[\text{Example:}\]

\[
\text{void } \text{func}(\ \text{int } a);
\text{auto } x1 = \text{get_name}\_v<\text{get_element}\_t<0, \ \text{get_parameters}\_t<\text{reflexpr}(\text{func}(42))>>;\]
\[
\text{void } \text{func}(\ \text{int } b);
\text{auto } x2 = \text{get_name}\_v<\text{get_element}\_t<0, \ \text{get_parameters}\_t<\text{reflexpr}(\text{func}(42))>>; \ // \text{ill-formed, no diagnostic required}
\]

— end example

§ 21.12.4 © ISO/IEC 2018 – All rights reserved 35
21.12.4.1 Object operations

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

    All specializations of reflects_same<T1, T2> shall meet the BinaryTypeTrait requirements
    (23.15.1), with a base characteristic of true_type if
    (1.1) T1 and T2 reflect the same alias, or
    (1.2) neither T1 nor T2 reflect an alias and T1 and T2 reflect the same entity;
    otherwise, with a base characteristic of false_type.

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

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

    — end example]
```

template <Object T> struct get_source_line;

template <Object T> struct get_source_column;

    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 the most
    recent 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 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 (19.8) of the most recent 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 require-
    ments (23.15.1). The nested type named type is an alias to the template Tpl specialized
    with the types 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;
```

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

2. All specializations of `get_name<T>` and `get_display_name<T>` shall meet the UnaryTypeTrait requirements (23.15.1) with a static data member named value of type const char (&)[N], referencing a static, constant expression character array (NTBS) of length N, as if declared as static constexpr char STR[N] = ...;

   — For T reflecting an unnamed entity, the string’s value is the empty string.

   — 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>`.

   — 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>`.

   — 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 (10.1.7.2);
     - (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;
     - (2.4.10) for T reflecting a function, its unqualified name;
     - (2.4.11) for T reflecting a specialization of a template function, its template-name;
     - (2.4.12) for T reflecting a function parameter, its unqualified name;
     - (2.4.13) for T reflecting a constructor, the injected-class-name of its class;
     - (2.4.14) for T reflecting a destructor, the injected-class-name of its class, prefixed by the character ‘~’;
     - (2.4.15) for T reflecting an operator function, the operator element of the relevant operator-function-id;
     - (2.4.16) for T reflecting 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 (for instance for T reflecting a lambda object), the string’s value is the empty string for `get_name<T>` and implementation-defined for `get_display_name<T>`.

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]

   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.

4. 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:**
```
using i0 = int; using i1 = i0;
get_aliased_t<reflexpr(i1)> reflects int. — end example
```

### 21.12.4.5 Type operations

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

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.

**Example:**
```
int v; using v_m = reflexpr(v);
get_type_t<v_m> reflects int. — end example
```

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.

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

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

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.

**Example:**
```
using int_m = reflexpr(int);
get_reflected_type_t<int_m> x; // x is of type int
— end example
```

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

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.
template <Type T> struct is_class;
template <Type T> struct is_struct;

8 All specializations of is_class<T> and is_struct<T> shall meet the UnaryTypeTrait requirements (23.15.1). 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.

21.12.4.6 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).

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 (for the function’s parameters), 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.

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 (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 [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 (17.9.2).

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 is an alias to an ObjectSequence specialized with RecordMember types that reflect the following subset of non-template members of the class reflected by T:

(2.1) for get_data_members (get_member_functions), all data (function, including constructor and destructor) members.
(2.2) for get_public_data_members (get_public_member_functions), all public data (function, including constructor and destructor) members;
(2.3) for get_accessible_data_members (get_accessible_member_functions), all data (function, including constructor and destructor) members that are accessible from the context of the invocation of reflexpr which (directly or indirectly) generated T. [Example:

```c
class X {
    int a;
};
```
friend struct Y;
};

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

using X_mem_t = get_accessible_data_members_t<X::X_t>
static_assert(get_size_v<X_mem_t> == 1); // passes.

— end example

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

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

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

5 All specializations of these templates shall meet the TransformationTrait requirements (23.15.1). The nested type named type is an alias to an ObjectSequence specialized with RecordMember types that reflect the following subset of function members of the class reflected by T:

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

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

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

template <Record T> struct getDestructor;

template <Record T> struct getPublicMemberTypes;
template <Record T> struct getAccessibleMemberTypes;
template <Record T> struct getMemberTypes;

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

10.1 for getPublicMemberTypes, all public nested class types, enum types, or member typedefs;
10.2 for getAccessibleMemberTypes, all nested class types, enum types, or member typedefs that are accessible from the scope of the invocation of reflexpr which (directly or indirectly) generated T;
10.3 for getMemberTypes, all nested class types, enum types, or member typedefs.

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

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

template <Class T> struct getPublicBaseClasses;
template <Class T> struct getAccessibleBaseClasses;
template <Class T> struct getBaseClasses;

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

13.1 for getPublicBaseClasses, 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.

```cpp
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

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

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 specialization is `true_type`, otherwise it is `false_type`.

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

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

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).

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

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

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

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

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 expression of the constant reflected by `T`.

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

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`.

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

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

```cpp
template <Variable 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

- 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,

- `X` is the pointer-to-member type of the member variable reflected by `T` and `x` a pointer to that member.

§ 21.12.4.9 © ISO/IEC 2018 – All rights reserved
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` is an alias to `reflexpr(X)`, where `X` is the base class 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>;
```

Some 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` is an alias to an `ObjectSequence` specialized with `FunctionParameter` types that 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;
```

Some 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

\[\text{template } \langle \text{ParenthesizedExpression } T \rangle \text{ struct } \text{get_subexpression};\]

All specializations of get_subexpression\langle T \rangle 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

\[\text{template } \langle \text{FunctionCallExpression } T \rangle \text{ struct } \text{get_callable};\]

All specializations of get_callable\langle T \rangle 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

\[\text{template } \langle \text{FunctionalTypeConversion } T \rangle \text{ struct } \text{get_constructor};\]

All specializations of get_constructor\langle T \rangle shall meet the TransformationTrait requirements (23.15.1). For a type conversion reflected by T, the nested type named type is the Constructor reflecting the constructor of the type specified by the type conversion, as selected by overload resolution. The program is ill-formed if no such constructor exists. [Note: For instance fundamental types (6.7.1) do not have constructors. — end note]

21.12.4.17 Function operations

\[\text{template } \langle \text{Function } T \rangle \text{ struct } \text{get_pointer}<T>;\]

All specializations of get_pointer\langle T \rangle 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\langle Y \rangle, 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

\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{is_static}<T>; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{is_const}; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{is_volatile}; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{has_lvalueref_qualifier}; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{has_rvalueref_qualifier}; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{is_virtual}<T>; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{is_pure_virtual}<T>; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{is_override}; \]
\[\text{template } \langle \text{MemberFunction } T \rangle \text{ struct } \text{is_final}<T>; \]

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 k (for has_lvalueref_qualifier) or && (for has_rvalueref_qualifier), implicitly or explicitly virtual (for is_virtual), pure virtual (for is_pure_virtual), or marked with override (for is_override) or final (for is_final), the base characteristic of the respective template specialization is true_type, otherwise it is false_type.
21.12.4.19 SpecialMemberFunction operations

```cpp
template <SpecialMemberFunction T> struct is_implicitlyDeclared;
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_implicitlyDeclared`) 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

```cpp
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

```cpp
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

```cpp
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

```cpp
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` is an alias to an `ObjectSequence` specialized with `LambdaCapture` types that 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.

```cpp
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`.

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

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;

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.