Suggested Reflection TS NB Resolutions

Document #: P1390R0
Date: 2019-01-10
Project: Programming Language C++
SG7/Evolution/LEWG/LWG/Core
Reply-to: Matúš Chochlík
<chochlik@gmail.com>
Axel Naumann
<Axel.Naumann@cern.ch>
David Sankel
<dsankel@bloomberg.net>

Contents

1 Introduction

This paper contains resolution suggestions to national body comments issued in [N5325] which summarizes responses to [N4766], the proposed Reflection TS.

2 Suggested core resolutions (EWG / CWG)

2.1 CH 004 (04.03) - SG10/CWG

See GB 086.

2.2 CH 005 (06p1) - CWG

Recommend to reject the proposed change, in favor of the conflicting solution proposed in CH 006.

2.3 CH 006 (06p1) - CWG

Recommend to accept the proposed change with modification, to clarify that *alias* refers exclusively to types:

> Rename all occurrences of *alias* to *type-alias* throughout the document.

(Note: the modification consists of the added hyphen in "*type-alias*".)

2.4 CA 4 008 (08) - CWG

Recommend to reject the proposed change: the type specified by a *reflexpr-specifier* satisfies `Constant` only if the *reflexpr-operand* is the name of a constexpr variable or an enumerator.

(Note that the example provided as part of CA 4 008 is invalid as `reflexpr(foo(a))` specifies a type that satisfies `FunctionCallExpression`, not `Constant`.)

2.5 PL 009 (10.01.7.2, 10.1.7.6) - CWG

Recommend to accept the proposed change: while the implementation defines the type, including its name, the TS does not actually specify the types. The use of "unspecified type" would be in line with many other occurrences in the IS (see e.g. [std.manip]).

Modify 10.1.7.2/3 as follows:

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

Modify 10.1.7.6/3 as follows:

> The type specified by the *reflexpr-specifier* is implementation-definedunspecified.

2.6 CH 010 (10.01.7.2) - SG7/EWG

Recommend to reject the proposed change, with the rationale to allow future extensions reflecting templates. For those, a leading `template` specifier could be useful. The example's intent can instead be achieved using an intermediary alias:

```
using ABC = A::template B<C>;
using ABC_r = reflexpr(ABC);
```

2.7 CA 5 011 (10.01.7.2) - CWG

Recommend to accept the proposed change.

Modify 10.1.7.2 as follows:

> *reflexpr-operand:*
>
> `::`
>
> *type-id*
> *nested-name-specifier*$_{opt}$ *identifier*
> *nested-name-specifier*$_{opt}$ *simple-template-id*
>
> *id-expression*

Modify 10.1.7.6 as follows:

> For an operand of the form *identifierid-expression* where*identifierid-expression* is a template *type-parameter*,...

Modify the caption of Table 12 as follows:

> `reflect` concept (21.12.3) that the type specified by a *reflexpr-specifier* satisfies, for a given *reflexpr-operand* *identifier* or *simple-template-id**id-expression*.

Modify the description of the second column of Table 12 as follows:

> *identifier* or *simple-template-id* kindkind of name denoted by *id-expression*

2.8 CH 013 (10.01.7.6) - CWG

Recommend to accept the proposed change.

Modify 10.1.7.6 as follows:

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

2.9 CA 12 016 (10.01.7.6 p1) - CWG

Recommend to reject the proposed change. We want to leave the door open for expression reflection.

2.10 CA 11 017 (10.01.7.6 p1) - CWG

Recommend to accept the proposed change.

Add the following line to Table 12, as the last row in the "type" Category:

> | decltype-specifier | both `reflect::Type` and `reflect::Alias` |

2.11 CA 9 019 (10.01.7.6 p1) - CWG

Recommend to accept the proposed change. Modify 10.01.7.6 as follows:

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

2.12 CA 7 021 (10.01.7.6 p1) - CWG

Recommend to accept the proposed change with modification. Modify 10.01.7.6 as follows:

> An entity or alias `AB` is *reflection-related* to an entity or alias `BA` if
>
> - `A` and `B` are the same entity or alias,
> - `A` is a variable or enumerator and `B` is the type of `A`,
> - `A` is an enumeration and `B` is the underlying type of `A`,
> - `A` is a class and `B` is a member or base class of `A`,
> - `A` is a non-template alias that designates the entity `B`, - `A` is a class nested in `B` (12.2.5),
> - `A` is not the global namespace and `B` is an enclosing class or namespace of `A`,
> - `BA` is the parenthesized expression `( [A]{.rm}[B]{.add} )`,
> - `A` is a lambda capture of the closure type `B`,
> - `A` is the closure type of the lambda capture `B`,
> - `AB` is the type specified by the *functional-type-conv-expression* `BA`,

- `AB` is the function selected by overload resolution for a *function-call-expression* `BA`,
- `BA` is the return type, parameter type, or function type of the function `BA`, or
- `AB` is reflection-related to an entity or alias `X` and `X` is reflection-related to `BA`.

(Note: the modification to the suggested change consists in the clarification of enclosing classes, and the removal of the superfluous nested class case. The latter is subsumed by "member or base class of `A`".)

2.13 US 024 (10.01.7.6 p1) - CWG

Recommend to reject. Many other occurrences of "at block scope" in the standard, e.g. (6.3.10/3). The term *entity* is introduced in [basic]/3 as "value, object, reference, structured binding, function, enumerator, type, class member, bit-field, template, template specialization, namespace, or pack" which seems approriate in this context, too.

2.14 US 025 (10.01.7.6 p1) - CWG

Recommend to accept proposal with modification:

Modify 10.1.7.6 as follows:

> If the *reflexpr-operand* designates an entity or alias atenclosed in a block scope (6.3.3) or function prototype scope (6.3.4) and the entity is neither captured nor a function parameter,...

2.15 CH 026 (17.07.2.1 p09.10) - CWG

Recommend to accept to proposed modification of 17.7.2.1:

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

2.16 CA 13 027 (17.07.2.1 p1) - CWG

Recommend to accept the proposed modification of 17.7.2.1.

Add a new bullet after 9.9:

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

2.17 CH 029 (21.12) - CWG

Recommend to accept the proposed modification of 21.12.2:

For every `constexpr` variable defined in 21.12.2, add the `inline` specifier as in the following example:

```
template <class T>
- constexpr auto is_public_v = is_public<T>::value;
+ inline constexpr auto is_public_v = is_public<T>::value;
```

2.18 CA 16 030 (21.12) - CWG

Recommend to accept the proposed change.

Modify 21.12.3.6 as follows:

```
- template <class T> concept Enumerator = Typed<T> && ScopeMember<T> && see below;
+ template <class T> concept Enumerator = Constant<T> && see below;
```

Modify 21.12.3.7 as follows:

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

Modify 21.12.3.9 as follows:

```
- template <class T> concept Typed = Named<T> && see below;
+ template <class T> concept Typed = Object<T> && see below;
```

Modify 21.12.3.10 as follows:

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

Modify 21.12.3.17 as follows:

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

> Constant is true if and only if T reflects a constant expression (8.6) , an enumerator, or a `constexpr` variable.

Modify 21.12.3.20 as follows (for consistency's sake):

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

Modify 21.12.3.25 as follows (for consistency's sake):

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

Modify 21.12.3.31 as follows (for consistency's sake):

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

2.19 CA 15 031 (21.12) - SG7/EWG

Recommend to accept the proposed change by modifying 21.12.4.7 as follows:

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 their original declaration.

2.20 CH 032 (21.12.02) - CWG

Recommend to accept the proposed change.

Modify 21.12.2 as follows:

```
- template <class T>
+ template <Object T>
  constexpr auto is_public_v = is_public<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_protected_v = is_protected<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_private_v = is_private<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_constexpr_v = is_constexpr<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_static_v = is_static<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_final_v = is_final<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_explicit_v = is_explicit<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_inline_v = is_inline<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_virtual_v = is_virtual<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto is_pure_virtual_v = is_pure_virtual<T>::value;
- template <class T>
+ template <Object T>
   constexpr auto get_pointer_v = get_pointer<T>::value;
```

2.21 PL 035 (21.12.02) - CWG

Recommend to reject. As 21.12.3.27 notes:

> Some types that satisfy `Constructor` also satisfy `SpecialMemberFunction`.

The definition of `SpecialMemberFunction` (21.12.3.27) uses "special member function (Clause 15)" and thus clearly includes special member functions that are constructors.

2.22 PL 036 (21.12.02) - SG7/EWG

Recommend to reject the proposed change of renaming the concept `Object` to `Metaobject`:

The meta-nature of `std::reflect::Object` is stated by the namespace it is defined in. There is no need to repeat this meta-nature by prepending "Meta".

2.23 CH 039 (21.12.02, 21.12.3.9) - CWG

See CA 16 030.

2.24 CH 040 (21.12.03) - SG7/EWG

Recommend to accept the proposed modification by adding the following note to 21.12.03:

[*Note:* unlike `std::is_enum`, `std::reflect::is_enum` operates on meta-object types. — *end note*]

2.25 CH 041 (21.12.3.10) - SG7/EWG

See CA 16 030.

2.26 CH 042 (21.12.3.18) - SG7/EWG

This is a misunderstanding. A 'base' in this case has a class which is accessible with `get_class`. This class then has a name. Suggest no action.

2.27 CH 043 (21.12.03.6) - CWG

See CA 16 030.

2.28 CH 044 (21.12.03.7) - CWG

See CA 16 030.

2.29 CA 19 045 (21.12.04) - CWG

Recommend to accept the suggested change to 21.12.4.3, while leaving 21.12.4.1 unmodified, as it specifically references (19.8).

Add a new sentence after 21.12.4.3/5:

6 The value of the NTBS is first formed using the basic source character set (with *universal-character-names* as necessary) then mapped in the manner applied to string literals with no encoding prefix in phases 5 and 6 of translation.

2.30 CA 18 046 (21.12.04) - CWG

No recommendation from the authors.

2.31 PL 047 (21.12.04) - SG7/EWG

It is unclear how (or if) such a thing can be implemented. At any rate, this is something that can be added in the future if we get implementation experience that suggests this is possible and desirable.

2.32 CA 20 049 (21.12.04.1) - CWG

Recommend to accept with modification, to match e.g. "subsequent redeclaration" in 10.6.4.:

In 21.12.4.1, apply the following changes:

3 ... representing some offset from the start of the line (for `get_source_column<T>`) of the most recent redeclaration of the entity or typedef described by `T`.

4 ... The value of the NTBS is the presumed name of the source file (19.8) of the most recent redeclaration of the entity or typedef described by `T`.

2.33 CH 050 (21.12.04.10 p1) - CWG

Recommend to accept the proposed modification, by applying the following change to 21.12.4.10:

1 ... The nested type named `type` is an alias to `reflexpr(X)`, where `X` is the base class (without retaining possible `Alias` properties, see 21.12.3.4) reflected by `T`.

2.34 CA 36 052 (21.12.04.16 p1) - CWG

No recommendation from the authors.

2.35 CA 35 053 (21.12.04.16) - SG7/EWG

Recommend to reject the proposed change: the current wording states (21.12.4.16):

> 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.

In the NB comment's example, `get_constructor_t<reflexpr(T{"Popeye"})>` is instantiated with `T` being `const S&`; the "type specified by the type conversion" is thus `S`. The constructors under consideration are those of `S`, and the one "selected by overload resolution" is `S::S(const std::string&)`.

2.36 PL 054 (21.12.04.18) - SG7/EWG

Recommend to accept.

1 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 marked with `override`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`.

2.37 CH 057 (21.12.04.3 p2.1) - SG7/EWG

This is already the existing behavior and lambda objects are specifically called out. See Section 21.12.4.3 paragraph 2.5. Lambda objects are not unnamed entities.

2.38 CH 058 (21.12.04.3) - SG7/EWG

Recommend to accept, by applying the following modification to 21.12.4.3:

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

Template aliases are not supported by N4766.

2.39 CA 32 060 (21.12.04.3 p2) - CWG

Recommend to accept, by applying the following modification to 21.12.4.3:

2.4.15 - for `T` reflecting an operator function, the *operator* element of the relevant *operator- function-id*; or, in the case of a literal operator, the string `"\"\""` followed by the *literal suffix identifier* of the operator's *literal-operator-id*, followed by a terminating `'\0'` character;

2.40 CA 31 061 (21.12.04.3 p2) - CWG

See CH 058.

2.41 CA 28 064 (21.12.04.3 p2) - CWG

Recommend to accept, by applying the following modification to 21.12.4.3:

2.4.6 - for `T` reflecting all other *simple-type-specifiers*a cv-unqualified fundamental type other than `std::nullptr_t`, the name stated in the "Type" column of Table 9 in (10.1.7.2);

2.42 CA 27 065 and CA 26 066 (21.12.04.3 p2) - CWG

For CA 26 066, see also CH 075.

Recommend to accept, by applying the following modification to 21.12.4.3:

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*alias-declaration*, or a `typedef` declaration;

Then insert a new bullet (2.4.2), moving the existing (2.4.2) to (2.4.3) etc:

2.4.2 - for `T` reflecting a *type-parameter*, the identifier introduced by the *type-parameter*.]{.add}

2.43 CA 24 068 (21.12.04.3 p2) - CWG

See CH 058.

2.44 CH 072 (21.12.04.5) - SG7/EWG

Recommend to accept: `struct` and `class` are conceptually the same, and `std::reflect::is_struct` wrongly suggests the need of `std::is_struct`. To clearly distinguish the `std::is_class` type trait from these meta functions, we recommend to rename them as proposed in the NB comment.

Apply the following modification to 21.12.2:

```
- template <Type T> struct is_class;
+ template <Type T> struct uses_class_key;
- template <Type T> struct is_struct;
+ template <Type 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 <Type T>
```

```
-  constexpr auto is_class_v = is_class<T>::value;
+  constexpr auto is_class_v = uses_class_key<T>::value;
template <Type T>
-  constexpr auto is_struct_v = is_struct<T>::value;
+  constexpr auto is_struct_v = uses_struct_key<T>::value;
```

Apply the following modification to 21.12.4.5:

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

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

8 All specializations of is_classuses_class_key`<T>` and is_struct[`uses_struct_key`]{`.add`}`<T>` shall meet the `UnaryTypeTrait` require ments (23.15.1). If `T` reflects a class with *class-key* `class` (for is_classuses_class_key`<T>`) or `struct` (for is_structuses_struct_key`<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_classuses_class_key`<T>` and is_structuses_struct_key can be `true_type`, the other template specialization is `false_type`; the actual choice of value is unspecified.

2.45 CH 074 (21.12.04.6) - SG7/EWG

To allow future extensions of template reflection, we recommend to accept this comment with modification.

To represent the scope of template parameters also for variable templates, a new concept needs to be introduced, serving as the scope of template parameters. For the time being, this concept has no operations (beyond that of `Object`) defined.

Modify 21.12.2 as follows:

```
template <class T> concept ObjectSequence; // refines Object
+ template <class T> concept TemplateArgumentList; // refines Object
template <class T> concept Named; // refines Object
```

Add a new paragraph 21.12.3.3, moving the existing 21.12.3.3 to 21.12.3.4, etc:

21.12.3.3 Concept `TemplateArgumentList`

`template <class T> concept TemplateArgumentList = Object<T> && see below; TemplateArgumentList<T>` is `true` if and only if `T` is a sequence of `Alias` types reflecting template *type-parameter*s, generated by a metaobject operation. [*Note:* It represents the template argument list (17.2), and provides a scope to the `Alias` reflecting a template *type-parameter* (6.3.9). — *end note*].

Modify 21.12.3.4 as follows:

[*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 a `TemplateArgumentList`. — *end note*]

Add the following sentence to the end of 21.12.4.6, paragraph 2:

2 ... For template *type-parameter*s, this innermost scope is the `TemplateArgumentList` representing the template parameter scope in which they have been declared.

2.46 US 075 (21.12.04.6) - CWG

Recommend to accept the proposed change, by modifying 21.12.4.6 as follows:

2 ... class scope, enumeration scope, function scope (in the case where the function's parameters), or ...

2.47 CH 078 (21.12.04.7 and many others) - CWG

Recommend to accept the proposed change.

Modify 21.12.4.7 as follows:

2 ... The nested type named `type` is an alias to an `ObjectSequence` specialized with `RecordMember` types thata meta-object type satisfying `ObjectSequence`, containing elements which satisfy `RecordMember` and reflect the following subset of non-template members of the class reflected by `T`:

5 ... The nested type named `type` is an alias to an `ObjectSequence` specialized with `RecordMember` types thata meta-object type satisfying `ObjectSequence`, containing elements which satisfy `RecordMember` and reflect the following subset of function members of the class reflected by `T`:

10 ... The nested type named `type` is an alias to an `ObjectSequence` specialized with `Type` types thata meta-object type satisfying `ObjectSequence`, containing elements which satisfy `Type` and reflect the following subset of types declared in the class reflected by `T`:

13 ... The nested type named `type` is an alias to an `ObjectSequence` specialized with `Base` types thata meta-object type satisfying `ObjectSequence`, containing elements which satisfy `Base` and reflect the following subset of base classes of the class reflected by `T`:

Modify 21.12.4.8 as follows:

2 ... The nested type named `type` is an alias to an `ObjectSequence` specialized with `Enumerator` types thata meta-object type satisfying `ObjectSequence`, containing elements which satisfy `Enumerator` and reflect the enumerators of the enumeration type reflected by `T`. ...

Modify 21.12.4.13 as follows:

1 ... The nested type named `type` is an alias to an `ObjectSequence` specialized with `FunctionParameter` types thata meta-object type satisfying `ObjectSequence`, containing elements which satisfy `FunctionParameter` and reflect the parameters of the function reflected by `T`. ...

Modify 21.12.4.23 as follows:

1 ... The nested type named `type` is an alias to an `ObjectSequence` specialized with `LambdaCapture` types thata meta-object type satisfying `ObjectSequence`, containing elements which satisfy `LambdaCapture` and reflect the captures of the closure object reflected by `T`. ...

2.48 CH 079 (21.12.04.9) - SG7/EWG

The lack of an interface to determine thread-local storage duration is surprising; we recommend to accept the proposed modification.

Modify 21.12.2 as follows:

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

Modify paragraph 4 of 21.12.4.9 as follows:

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

`template <Variable T> struct is_thread_local;`

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

2.49 CH 080 (21.12.04.9) - SG7/EWG

Recommend to accept the proposed change, by adding a new paragraph after paragraph 4:

5 `reflect::get_pointer<T>` is ill-formed if `T` reflects a reference and the name of the reference in a context with no lvalue-to-rvalue conversion would not be a constant expression."

2.50 CH 082 (Annex C) - CWG

Recommend to accept the proposed change, by editing paragraph 1 in C.5.1 Clause 5: lexical conventions [diff.cpp17.lex]:

1 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*.

2.51 GB 083 (Annex C) - CWG

See CH 082.

2.52 GB 084 (Annex C) - CWG

See CH 013.

2.53 GB 086 (General, 04.2) - SG10/CWG

Propose to accept the suggested modification, by inserting a new 4.3 between 4.2 and 4.3:

4.3 Feature-testing recommendations [intro.features]

1 An implementation that provides support for this Technical Specification shall define the feature test macro(s) in Table 2.

Macro name	Value
___cpp_reflect	201811

Table 2 - Feature-test macro(s)

3 Suggested library resolutions (LEWG/LWG)

3.1 CA 14 028 (20.05.1) - LEWG

Suggestion to accept this request.

- Add `<experimental/reflect>` to Table 19 referenced by 20.5.1.3.

3.2 CH 056 (21.12.04.2) - SG7/LWG

Modify definition of `unpack_sequence_t` from [reflect.synopsis] as follows.

```
template <template <class...> class Tpl, ObjectSequence T>
-   constexpr auto unpack_sequence_t = unpack_sequence<Tpl, T>::type;
+   using unpack_sequence_t = typename unpack_sequence<Tpl, T>::type;
```

3.3 CA 1 085 (General) - SG7/LEWG; recommend: CWG!

The authors believe that this comment should be addressed to CWG, rather than SG7/LEWG.

The authors recommend that the TS does not need a definition of `align_val_t` for the code example in the NB comment:

```
#include <experimental/reflect>
#include <stdio.h>
#include <stdlib.h>
struct A {
  template <typename T> operator T();
};
template <typename T>
A::operator T() {
  using std::experimental::reflect;
  printf("%s\n", get_name_v<get_scope_t<get_aliased_t<reflexpr(T)>>>);
  exit(0);
}
int main(void) {
  (void) ::operator new(sizeof(int), A());
}
```

None of the operations require a complete type. If they did, the code would be ill-formed.

Regarding the spelling of the type name, 6.6.4.4 states:

> The implicit declarations do not introduce the names `std`, `std::size_t`, `std::align_val_t`,... However, referring to `std` or `std::size_t` or `std::align_val_t` is ill-formed unless the name has been declared by including the appropriate header.

The example code provided in the NB comment does not *refer* to `std::align_val_t`; it uses an implicit declaration of a function and the spelling of one of its parameter types. This usage is similar to the usage of the parameter type for this implementation of the conversion function:

```
template <typename T>
A::operator T() {
  printf(std::experimental::source_location::current().function_name());
  exit(0);
}
```

This, too, does not require the inclusion of "the appropriate header".

4 Acknowledgements

Special thanks to Michael Park who is responsible for the pandoc extensions [PBLOG] used to create this paper.

5 References

[N4766] Matúš Chochlík, Axel Naumann, and David Sankel. 2018. Working Draft, C++ Extensions for Reflection.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf

[N5325] Secretariat: ANSI (United States). 2018. Summary of Voting on PDTS 23619, Technical Specification – C++ Extensions for Reflection.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n5325.pdf

[PBLOG] Michael Park. 2018. How I format my C++ papers.
https://mpark.github.io/programming/2018/11/16/how-i-format-my-cpp-papers/