The Syntax of Static Reflection

Andrew Sutton (asutton@lock3software.com)
Wyatt Childers (wchilders@lock3software.com)
Daveed Vandevoorde (daveed@edg.com)

Document	P2320R0
Audience	SG7

1 Introduction

This paper suggests a new syntax for reflection and splicing that differs from both [1] and [2]. The syntax in both proposals has various weaknesses. P1240R1 is largely written using placeholder notation, and while P2237 offers some concrete improvements, it turns out to have some ambiguity issues.

We (the authors) considered several notations for the two operations with respect to several criteria:

- *Expressivity.* The syntax must support a wide array of metaprogramming use cases. Broadly, we need syntax to a) inspect the compile-time properties of expressions, names, declarations, and entities, b) splice entity references and expressions back into source code, and c) expand such splices in contexts where pack expansion is allowed.
- *Readability.* Obviously, we'd like our programs to be readable. We want syntax that is both visually distinctive yet comprehensible. Because metaprogramming (especially splicing) includes several new C++ programming concepts, we think the notation should be different than conventional notations for e.g., function call and template instantiation.
- *Flexibility.* We shouldn't design notation that prevents future extensions. We should also consider future extensions to metaprogramming (like code injection) so that we don't end up with wildly different notations for similar kinds of functionality.
- *Lack of ambiguity.* The grammar for these terms should be as unambiguous as practical, both technically (i.e., not requiring parsing heroics) and visually (i.e., not confusing for a normal C++ programmer).
- *Implementability.* Syntax and semantics that cannot be supported by all implementations is not viable. The variety of syntactic and semantic analysis techniques used by different implementations is known to complicate, if not disqualify, seemingly simple or obvious ideas. We can't force implementers to standardize on technique.

Section 2 presents the notation we've chosen for reflection and splicing. Appendix A presents other notations and brief analyses.

2 Proposal

This section describes specific syntax for three features of static reflection: reflecting names and expressions (Section 2.1), splicing (Section 2.2), and pack expansion (Section 2.3). This corresponds to the scope of [1], but does not cover some metaprogramming mechanisms — like code injection — described in [2]. However, the authors have kept those mechanisms in mind while exploring the syntax options described here and in Appendix A.

2.1 Reflection

We propose to enable reflecting a source construct using ^ as a unary operator. Because reflection is such a fundamental (and primitive) operation for metaprogramming, it should have a simple spelling. The ^ is intended to imply "lifting" or "raising" a term into the metaprogramming environment.

```
meta::info r1 = ^int;  // reflects the type-id int
meta::info r2 = ^x;    // reflects the id-expression x
meta::info r2 = ^f(x); // reflects the call f(x)
```

The suggested grammar for reflection is:

unary-expression
    . . .
    reflection-expression

reflection-expression
    ^ *postfix-expression*
    ^ *type-id*
    ^ *qualified-namespace-specifier*
    ^ *qualified-template-specifier*
    ^ ::

One of the reasons that we opt for this very terse syntax over the prior `reflexpr(X)` form, is that we anticipate that it will be desirable to "pass arguments by reflection" in future proposals. Just as it is convenient to "pass an argument by address/pointer" using the simple * declarator and & operators, having a simple ^ will keep invocation syntax light and readable.

Another (weaker) reason to drop `reflexpr(X)` is that it has proven somewhat unpopular with the many readers of earlier reflection papers (See [3]).

There are two potentially conflicting uses of ^: Apple's blocks extension [4] and C++/CLI/CX's managed pointers.

With respect to blocks, ^ is used two ways: as a *unary-expression* and as part of a *declarator*. A "block literal expression" uses ^ as a unary operator with approximately this grammar in C++:

unary-expression:
    . . .
    ^ *type-specifier-seq*$_{opt}$ (*parameter-declaration-clause*) *compound-statement*

There is an overlap between this production rule and the *type-id* form of the *reflection-expression*. Because a block literal always has a *compound-statement*, there is no ambiguity. In the case where the *type-specifier-seq* is omitted, we would have to distinguish the paren-enclosed parameter list from a *primary-expression*. However, this is the same technique used to differentiate an *expression* from a *declaration* inside a *condition*.

The other potential conflict occurs in a declarative context. A block variable is declared like a function pointer (e.g., `int (^b)(int)`). We don't expect that "reflection parameters" would ever be declared in this way: a reflection is always a value, never a function.

C++/CLI/CX also has a potential conflict in the declarative context. A managed pointer is declared with the ^ operator:

```
MyClass ^h_MyClass = gcnew MyClass;
```

We aren't currently proposing using ^ as part of a declarator or parameter declaration, but we are concerned about potential conflicts for future proposals. Reflection variables and parameters will need to be constant expressions, so there's some wiggle room for using ^ as part of a specialized style of reflection declaration. Exactly, what such a declaration should look like is well beyond the scope of this paper, but we are thinking about it.

2.2 Splicing reflections

We propose the notation `[: R :]` to denote the splice of a reflection `R`. Here, the use of bracket notation is explicitly chosen to denote a "gap" in the source code, filled in by the "reflected value" of `R`. The notation is intentionally designed to be visually distinctive because it represents a new programming concept for C++. We prefer to encourage a degree of unfamiliarity.

In general, and without qualification, `[: R :]` splices an *expression* into the program (assuming `R` reflects a variable, function, or some other expression). If `R` reflects a type, template, or namespace, the splice operator must be qualified with an appropriate keyword, except in some contexts where the meaning is obvious. For example:

```
struct S { struct Inner { }; };
template<int N> struct X;

auto refl = ^S;
auto tmpl = ^X;

void f() {
  typename [:refl:] * x;  // OK: declares x to be a pointer-to-S
  [:refl:] * x;           // error: attempt to multiply int by x
  [:refl:]::Inner i;      // OK: splice as part of a nested-name-specifier
  typename [:refl:]{};    // OK: default-constructs an S temporary
  using T = [:refl:];     // OK: operand must be a type
  struct C : [:refl:] {}; // OK: base classes are types
  template [:tmpl:]<0>;   // OK: names the specialization
  [:tmpl:] < 0 > x;       // error: attempt to compare X with 0
}
```

Note that the extra annotations are necessary even in non-dependent contexts.

That said, we expect SG7 will entertain future proposals that relax the requirement for qualifying syntax when the splice operand is not value-dependent (as is done in P2237R0 [2]). However, there are two main obstacles to adopting such proposals. Is readability hurt by the omission of keywords? Does the implementation require parsing heroics? (There is no implementation experience for this abbreviated notation in P2237R0.) Note that requiring keywords now does not limit our ability to make them optional later.

There is one exception to this rule. Annotations differentiating type and non-type *template-argument*s in a *template-argument-list* can be omitted. This is a special case that allows reflections (and packs thereof) to be forwarded to a template taking mixed type/value template arguments or to an overload set where the kind and type of arguments may vary. For example:

```
template<typename T> void f();
template<int N> void f();

template<meta::info Refl>
void g() {
  f<[:Refl:]>();
}
```

Here, `Refl` can reflect either a type or integer constant expression. We don't require the program to explicitly state whether `Refl` reflects types or expressions, as it could reasonably reflect either.

To "force" `Refl` to be spliced as a type or an expression by adding `typename` or enclosing the splice in parentheses.

```
template<meta::info Refl>
void g() {
  f<typename [:Refl:]>(); // splices a type
```

```
  f<([:Refl:])>();        // splices an expression
}
```

We anticipate the later addition of an *identifier-splice* construct (currently we use the `[# str #]` syntax in discussions among authors). However, that construct operates, in part, at the lexical level and has considerably more subtleties that the authors are exploring (in part through prototype implementations). We therefore do not propose syntax for it here, and we expect that the corresponding functionality will be separated out in revisions of P1240 or other proposals.

The addition of splicing requires updating the grammar as follows.

splice
`[:` *constant-expression* `:]`

primary-expression
. . .
splice

postfix-expression
. . .
postfix-expression `.` `template`$_{opt}$ *splice*
postfix-expression `->` `template`$_{opt}$ *splice*

nested-name-specifier:
. . .
splice `::`

qualified-namespace-specifier:
. . .
`namespace` *splice*

(The `namespace` keyword in a *qualified-namespace-specifier* will be optional in contexts where no other terms could be spliced, such as in a *namespace-alias-definition*. Note that there are only a few contexts where a namespace might be spliced into a program, and the `namespace` keyword is effectively optional in all of them. We could therefore eliminate this addition to the grammar. However, we include it here for completeness and consideration.)

simple-type-specifier:
. . .
`typename`$_{opt}$ *splice*
`template` *splice*

(The `typename` keyword in a *simple-type-specifier* will be optional in very specific grammatical contexts, such as in a *base-specifier*.)

simple-template-id:
. . .
`template` *splice* `<` *template-argument-list* `>`

template-argument:
. . .
`template` *splice*

(The grammar change for *simple-template-id* will unfortunately require quite a few wording adjustments. Ideally, the way templates are "named" should be reworked in the grammar, because it is currently rather unintuitive.)

2.3 Splicing packs

The ability to expand a range of reflections into a list of function arguments, template arguments, base classes, etc. is an important use case for metaprogramming. However, the expansion of non-packs is a novel feature

and requires new syntax to nominate a term as being expandable. Our preferred approach is to require an ellipsis *before* the term being expanded. For example:

```
using T = std::tuple<int, ...[:range_of_types:]..., bool>;
```

Here, `range_of_types` is a sequence (in the range-based-for-loop sense) of type reflections. The leading `...` nominates the splice as expandable, and the trailing `...` explicitly indicates its expansion.

We propose to allow such nominations in almost every context where expansion is allowed (and no others). There are a number of reasons for choosing this syntax. First, a prefix annotation is necessary to be implementable by all vendors.[1] Second, the choice of `...` is chosen specifically because of the symmetry with expansion "operator" and the way in which packs are declared, where the `...` precedes the identifier.

For fold expressions, at most one cast-expression can be nominated as expandable.

```
(...[:range:] && ...)      // Right fold over a splice
(... && ...[:range:])      // Left fold over a splice
(0.0 + ... + ...[:range:]) // Left binary fold over a splice
```

Semantically, the expansion of such expressions is (more or less) just like the expansion of normal template and function parameter packs. Here are some examples:

```
fn(0, 1, ...[:range:]...); // OK: expansion after normal arguments
fn(...[:range:]..., 0, 1); // OK: expansion before normal arguments
fn(...[:range:] * 2..);    // OK: [:range:] * 2 is the pattern
fn(...[:r1:] * [:r2:]...); // OK: iff r1 and r2 have equal size
```

We currently suggest two contexts where nomination is not allowed.

Nominating expressions for use with the `sizeof...` operator seems unnecessary. The size of a range can be queried using `r.size()` or `std::ranges::distance(r)`. We don't need a language-based alternative to these operations.

The second context is in function and template parameter lists:

```
void f(... [:range_of_types:] ...args)
```

This seems like a plausible use of splicing, but there are some deep technical questions we have yet to address. In particular, `args` is kind of like a conventional pack, but not really because it's not dependent. We'll need to introduce new core language machinery to support the declaration of these new kinds of packs. Note that this seems closely related to the declaration of packs in discussed P1061R1 [5], P1858R2 [6], and P2277R0 [7].

There is a potential redundancy in the notation. It can be argued that a term nominated for expansion must always be expanded, so we could omit the trailing ellipsis, and this would be true today. However, this "optimization" is not applicable in fold expressions and may not be future-proof. In the future, we might introduce a "pass-by-reflection" convention that accepts unexpanded (non-splice) parameter packs. So for now, we choose to require ellipses for nomination and expansion.

To support this feature, we need to change the following grammar additions:

initializer-list:
  ...$_{opt}$ *initializer-clause* *...*$_{opt}$
  initializer-list , *...*$_{opt}$ *initializer-clause* *...*$_{opt}$

template-argument-list:
  ...$_{opt}$ *template-argument* *...*$_{opt}$
  template-argument-list , *...*$_{opt}$ *template-argument* *...*$_{opt}$

base-specifier-list:
  ...$_{opt}$ *base-specifier* *...*$_{opt}$

[1] Not all implementations preserve tokens or construct syntax trees during parsing. The prefix ellipsis in these contexts would alert the compiler that it needs to preserve those tokens for subsequent expansion.

base-specifier-list , ...$_{opt}$ *base-specifier* ...$_{opt}$

mem-initializer-list:
...$_{opt}$ *mem-initializer* ...$_{opt}$
mem-initializer-list , ...$_{opt}$ *mem-initializer* ...$_{opt}$

fold-expression
(...$_{opt}$ *cast-expression fold-operator* ...)
(... *fold-operator* ...$_{opt}$ *cast-expression*)
(...$_{opt}$ *cast-expression fold-operator* ... *fold-operator* ...$_{opt}$ *cast-expression*)

In the fold expression, at most one *cast-expression* can be nominated as expandable.

We are still working through the semantics of this feature, but we are reasonably confident that the syntax works, meaning that it satisfies our design criteria in Section 1. However, there are some cases where the semantics are non-obvious and need further consideration. For example, what happens when a pattern contains both "normal" function argument packs and spliced packs? Are they expanded simultaneously, or is one expanded before the other?

3 Conclusions

We think the notation presented here is concise, visually distinctive, and generally readable and writable. We have also considered the implementability of the proposed grammar and believe that the grammar proposes no serious or novel challenges. Furthermore, if working through use cases, we have found it to be consistent and composable.

We therefore request that SG7 approve further work building on these specific choices. If approved, the next step is to update P1240 with the new syntax along with examples and use, and to start developing core wording for these features.

4 References

[1] D. Vandevoorde, W. Childers, A. Sutton, F. Vali, and D. Vandevoorde, "P1240R1: Scalable reflection in c++." https://wg21.link/p1240r1; WG21, Oct. 2019.

[2] A. Sutton, "P2237R0: metaprogramming." https://wg21.link/p2237r0; WG21, Oct. 2020.

[3] M. Naydenov, "P2087R0: Reflection naming: Fix reflexpr." https://wg21.link/p2087r0; WG21, Jan. 2020.

[4] Apple Inc., "Language specification for blocks." 2009, [Online]. Available: https://opensource.apple.com/source/lldb/lldb-167.2/llvm/tools/clang/docs/BlockLanguageSpec.txt.auto.html.

[5] B. Revzin and J. Wakely, "P1061R1: Structured bindings can introduce a pack." https://wg21.link/p1061r1; WG21, Oct. 2019.

[6] B. Revzin, "P1858R2: Generalized pack declaration and usage." https://wg21.link/p1858r2; WG21, Mar. 2020.

[7] B. Revzin, "P2277R0: Packs outside of templates." https://wg21.link/p2277r0; WG21, Jan. 2021.

[8] D. Sankel, "N4856: C++ extensions for reflection." https://wg21.link/n4856; WG21, Mar. 2020.

[9] M. Naydenov, "P2088R0: Reflection naming: reification." https://wg21.link/p2088r0; WG21, Jan. 2020.

[10] B. Revzin and C. Pike, "P2011R0: A pipeline-rewrite operator." https://wg21.link/p2011r0; WG21, Jan. 2020.

5 Appendix A

This section provides analysis of other notations we considered.

5.1 P1240

P1240 is our (the authors') initial (and revised) proposal for static reflection and splicing. For reflecting source constructs, we adopted the `reflexpr` operator from the Reflection TS [8], making it an expression. We introduced a similar notation for splicing (i.e., a keyword followed by `()`s).

As noted, there has always been some baseline dislike for the name `reflexpr`, which occasionally appears in various proposals ([2], [3]).

The splice notations in P1240 are not very visually distinctive. It's easy, especially for non-experts, to mistake `typename(x)` as being somehow related to a *typename-specifier*. After all, in many contexts, parentheses are used only for grouping. A number of more specific concerns are discussed in P2088 [9].

The required use of keywords can lead to some unfortunate compositions. For example:

```
template<typename meta::info x>
void f() {
  typename typename(x)::type var;
}
```

Historically, repetition of keywords (i.e., `noexcept(noexcept(E))` and `requires requires`) is often viewed as a design failure by the broader C++ community (even when it is not). It seems like there should be some contexts in which the extra annotations can be elided because only a limited subset of terms are allowed, but this requires a careful analysis of contexts where these terms can appear, but that may have other grammatical consequences. For example, allowing the elision of the 2nd `typename` above effectively means that we would need a new splice notation as `(x)` is not a viable choice.

5.2 P2237

P2237 did not propose a replacement for `reflexpr`. An early draft suggested the name `reify`, but subsequent (offline) discussions showed that it was not an improvement to the status quo.

The splicing notation presented in P2237 was made with two goals in mind:

1. To be visually distinctive from conventional syntax, and
2. Avoid requiring annotations where they can be elided.

For example:

```
void f() {
  |^int| x = 42; // |^int| is a simple-type-specifier
  |^0|;          // |^0| is an expression
}
```

In cases where multiple productions can start with the same sequence of tokens, implementations typically produce synthetic tokens containing the fully parsed and analyzed sequence, effectively caching the parse. For example, GCC and Clang both do this with *nested-name-specifier*s and *template-id*s. The same technique would be used here, and we can label the synthetic tokens according to their computed grammatical category.

Inside templates, the usual rules for adding `typename` and `template` apply.

However, P2237's use of plain `|`s to delimit splices produces some unfortunate lexical issues. For example:

```
constexpr meta::info x = ^0;
constexpr meta::info y = ^x;
int z = ||^y||; // error: expected expression
```

In order to parse that as a nested splice, the parser would have to perform some seriously speculative lexical gymnastics. Similar (but resolvable) issues arise when the splice appears in juxtaposition to other `|` tokens such as the proposed pipeline rewrite operator. [10] All other considerations aside, this alone kills the plain `|x|` notation.

5.3 Alternative splicing notations

We discussed (among the authors) a number of alternative notations that fall into roughly three categories:

- a single bracketed splice notation (`[:x:]`),
- multiple bracketed splice operators (`[:e:]`, `[/t/]`, etc.), and
- a unary splice operator (e.g., `%x`).

We also discuss the impact that strongly typed reflections might have on the splice notation.

5.3.1 Bracketed single splice

The single bracketed approach is what we suggest in our proposal.

There are two syntactic motivating reasons for making the splice operator a bracketed expression: to represent "gaps" or "blanks" filled by a reflection and to allow its use in member access expressions:

```
cout << x.[:get_member():];
```

We could have avoided this extension by just using the member-to-pointer syntax, but it feels a little too arcane:

```
cout << x.*[:get_member():];
cout << (x->*[:get_member_fn():])(args);
```

Splicing member reflections directly through the `.` feels a bit more ergonomic.

We considered a number of different bracket operators, but ended up choosing `[:` and `:]`. Some alternatives included:

- `[| R |]`. We think this might be too similar to the attribute brackets. Interestingly, this notation is used by Template Haskell for quotes (like source code fragments in P2237). These brackets are also a visual approximation of evaluation functions in various semantics models.
- `<: R :>`. Not bad, but it combines poorly in template argument lists: `vec<<:R:>>`. We might use these brackets for source code fragments, since they rarely appear in template argument lists.
- `(: R :)`. Looks like smiley faces.
- `<| R |>`. Also not bad, but the closing token is proposed for the pipeline operator [10].
- `(| R |)`. Considered briefly.
- `{| R |}`. Not considered.
- `[< R >]`. Proposed in P1240 for splicing template arguments and implemented experimentally in Clang (with good feedback). However, we wanted to preserve `<>` notations for features a little more template-y.

There are a lot of ways we can combine tokens to create brackets. The current proposal seems to be a reasonable choice and has been found pleasant enough while working through use-cases.

5.3.2 Multiple bracketed splice notations

We don't need to limit ourselves to a single splice notation. We could choose to use different splice brackets for the different categories of grammatical constructs being inserted into the program. In fact, P1240 does this for two of its splicers (which it calls "reifiers"): identifiers (`[:x:]` and template arguments (`[<x>]`). The identifier splice (`|#x#|`) in P2237 can also be considered an application of this approach. Template Haskell takes a similar approach with its splice operator(s).

```
[:expr:] // splice an expression
[/type/] // splice a type
[<temp>] // splice a template
[:ns:]   // splice a namespace
```

The choice of some brackets here approximate some aspect of the thing reflected: template-ids have `<>`s and namespaces appear in *nested-name-specifier*s. The choice of brackets for expressions and types are chosen somewhat arbitrarily (types appear in italics?).

One plausible benefit of this approach is that it eliminates the need for keywords in many contexts. For example:

```
// template-id
[<temp>]<int>
typename [<temp>]<int>

// simple-type-specifier
[/type/]
foo::[/type/]

// nested-name-specifiers
[/type/]::id
[:ns:]::id
[<temp>]<int>::id

foo::[:ns:]::id
```

Note that we still need a leading `typename` when splicing a *template-id* as a type. That seems unavoidable.

The benefit is more illusory than real. We're not eliminating keywords, we're replacing them with notation, which can be a more cryptic than beneficial. It also means that programmers have to choose the right splice notation in contexts where only one would be allowed.

5.3.3 Unary splice

There's no strict requirement for splice notation to be bracketed. The design in P2237 prefers brackets for its visual appeal, but we could easily choose to do this with a unary operator, replacing the suggested `[:x:]` notation with, say, `%x`.

As above, without qualification a splice is an expression:

```
template<meta::info v, // reflects a variable
         meta::info x> // reflects a non-static data member m in T
void f(T& t) {
  cout << %x;   // OK: prints the value of V
  cout << t.%x; // OK: prints the value of t.m
}
```

For a splice of anything else (in certain contexts), keywords are required. Again, some example can reveal the flavor implied by such an approach:

```
// unqualified-id (as an expression)
template %temp<int>

// typename-specifier
typename %type
typename template %temp<int>
typename foo::%type
typename foo::template %temp<int>

// nested-name-specifiers
%type::id
%ns::id
template %temp<int>::id
foo::%type::id
foo::%ns::id
foo::template %temp<int>::id
```

If we choose this direction, then we should also choose an alternative for the `reflexpr` operator so that they naturally complement each other. For example, we could choose `/` for the reflection operator.

```
constexpr meta::info x = /int; // reflect
int n = %x;                    // splice
```

and we could choose `\` (yes, backslash) as the splice operator.

```
constexpr meta::info x = /int; // reflect
int n = \x;                    // splice
```

We could also choose to make the splice operator a suffix instead of a prefix.

```
constexpr meta::info x = /int; // reflect
int n = x\;                    // splice
```

Giving the operator lower precedence than a unary operator would allow this somewhat clever construction:

```
/int\ // splices the reflection of int (i.e., identity)
```

A downside of this approach is that single character unary operators are not very visually distinctive, and we have found that splice constructs standing out really helps readability. This also seems just a little too cute.