Pattern matching using `is` and `as`

Document Number: **P2392 R1**
Date: 2021-07-14
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: EWG

Abstract

This paper aims to build on the strong foundation of existing pattern matching papers, and to contribute:

a clean and regular syntax for expressing match alternatives without name introduction ambiguities,

in a generalizable way that could be used consistently throughout the language, because matching a pattern is a broadly useful feature that ideally should not be limited to `inspect` statements only.

Contents

1 Key goal: Avoid divergent sub-language with special syntax

"I want an integrated set of language features and libraries for C++ …
Don't try to define 'isolated' mini-languages within C++" — B. Stroustrup in [Solodkyy 2014b]

This paper agrees with the existing pattern matching proposals' motivation and implementation "bones." This paper tries to regularize the user interface "skin," to avoid inventing a divergent sub-language with special-purpose one-off syntaxes that end up different from elsewhere in the language, and to embrace the power of pattern matching and make it useful throughout the language and not just inside `inspect`. In particular:

- **Avoid special syntax for match alternatives.** Instead, use a syntax that can be used also outside `inspect` (e.g., with `if`).
- **Avoid special syntax to distinguish introduced names from uses of existing names.** Instead, put the two in unambiguously separate grammar positions.
- **Keep simple things simple.** Give common cases the nice syntax that is a subset of the general syntax, by allowing "advanced" options to be well defaulted and syntactically omitted when not used..

Other syntax choices like `inspect`/`switch` or `=>`/`:` are also important, but I try to follow the previous papers for those to avoid distracting from this key goal. See §4 Appendix for discussion of some alternatives.

1.1 Nutshell overview of this approach

In this approach, throughout the language:

`[ ]` structured bindings syntax is used uniformly for all decomposition. It is generalized to allow nesting and wildcards (e.g., `[a,[b,_]]`), and used for patterns (e.g., `[0,[_,even]]`).

`x is C` can be used uniformly for all constraints. `C` can be a type predicate, specific type, value predicate, or specific value.

`x as T` can be used uniformly to invoke all casting (conversions and coercions). `T` is a type predicate or specific type.

and all pattern matching alternatives are expressed using `is C`, `as T`, or `if Cond`. New names are always introduced to the left of `is`/`as`/`if`, and so never ambiguous with existing names to the right. For example:

```
constexpr auto even (auto const& x) { return x%2 == 0; }  // given this example predicate

// x can be anything suitable, incl. variant, any, optional<int>, future<string>, etc.

void f(auto const& x) {
    inspect (x) {
        i as int           => cout << "int " << i;
        is std::integral   => cout << "non-int integral " << x;
        [a,b] is [int,int] => cout << "2-int tuple " << a << " " << b;
        [_,y] is [0,even]  => cout << "point on y-axis and even y " << y;
        s as string        => cout << "string \"" + s + "\"";
        is _               => cout << "((no matching value))";
    }
}
```

The same matching is allowed consistently throughout the language. For example, in `if` conditions:

```
void f(auto const& x) {
    if      (auto i as int = x)           { cout << "int " << x; }
    else if (x is std::integral)          { cout << "non-int integral " << x; }
    else if (auto [a,b] is [int,int] = x) { cout << "2-int tuple " << a << " " << b; }
    else if (auto [_,y] is [0,even]  = x) { cout << "point on y-axis and even y " << y; }
    else if (auto s as string = x)        { cout << "string \"" + s + "\""; }
    else                                  { cout << "((no matching value))"; }
}
```

… or in `requires` clauses, which do not currently support declaring new names (so those go in the body) or dynamic constraints:

```
void g(auto const& x)  requires requires{x as int;}
                       { auto i = x as int; cout << "int " << x; }

void g(auto const& x)  requires (x is std::integral)
                       { cout << "non-int integral " << x; }

void g(auto const& x)  requires (x is [int,int])
                       { auto [a,b] = x; "2-int tuple " << a << " " << b; }

void g(auto const& x)  requires requires{x as string;}
                       { auto s = x as string; cout << "string \"" + s + "\""; }

void g(auto const& x)  { cout << "((no matching value))"; }
```

1.2 Design principles

Note These principles apply to all design efforts and aren't specific to this paper. Please reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability. — For example, this proposal uses the same decomposition syntax as structured bindings, and extends both for nesting.
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination. — For example, this proposal enables freely using `is` for all matching (including type predicates, specific types, value predicates, and specific values) and `as` for all casting (including directly supporting dynamic typing).
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features. — This proposal uses `is` and `as` that can be allowed generally in the language outside pattern matching (if we want to), and avoids inventing a one-off syntax for introducing names.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

Additional design principles include: Make important things and differences visible. Make unimportant things and differences less visible. — This proposal makes declaring new names vs. using existing names clear by not putting them in the same grammar location; new names always appear before `is`/`as`, existing names always after.

1.3 Acknowledgments

Special thanks to the earlier pattern matching proposers and prototypers for bringing C++ pattern matching to this point, including but not limited to: Bruno Cardoso Lopes, Sergei Murzin, Michael Park, David Sankel, Dan Sarginson, Yuriy Solodkyy, Bjarne Stroustrup.

Thank you also to the following for their feedback and contributions: Kevlin Henney, Corentin Jabot, Tomasz Kamiński, Andrzej Krzemieński, Jens Maurer, Axel Naumann, Balog Pal, Barry Revzin, Richard Smith, Andrew Sutton, Andrew Tomazos, Mads Torgersen, Jeroen Van Antwerpen, Ville Voutilainen, Jarrad Waterloo, Zhihao Yuan, and Reddit user D_0b.

1.4 Revisions

R1: Incorporated EWG feedback on reflector and 2021-07-07 initial review.

- Clarified how indexed access (e.g., `variant`) works for `is` and `as` with updated Godbolt example, including for variants with repeated types.
- Clarified that `operator is` side effects are ignorable.
- Various minor clarifications and bug fixes.

2 Posit for the sake of discussion: is and as expressions

The purpose of this section is to demonstrate that the pattern matching approach in the rest of the paper was not designed in isolation for pattern matching, but that I considered syntax and semantics that could be general and consistent across the whole language (if we want).

For the sake of discussion, say the language had `is` and `as` expressions as described in this section, including a `_` "don't care" wildcard/placeholder (as mentioned in [P0144R2] §3.8 and [P2169R2], and see also [P1110R0] and [P1469R0]).

Let `typeof(x)` mean `std::remove_cvref_t<decltype(x)>`. Just so I don't have to write that everywhere.

Let `Pointer<P>` be a concept that is true iff `P` is dereferenceable using unary `*` and `->`.

2.1 General constraint expression: is

`is` expressions provide a consistent syntax for type and value match queries, that is also generalized to support static and dynamic queries including customized queries.

A type or value constraint `C` can be a type predicate (supports `C<x>` where x is a type, such as a concept or `_v` type trait; see also [P0144R2] §3.7), a specific type (see also [P0144R2] §3.6), a value predicate (supports `C(x)` where `x` is an expression), a specific value, or the general `_` placeholder to match anything.

Let an `is`-expression be of the form

```
x is C                                          // x is an expression or a type
```

where `is` has precedence just below member selection (same as `&`), the type of the expression is `bool`, and:

- If `C` is `_`, then `x is C` means `true`.
- Otherwise, if `x` is a valid value expression:
  - If `operator is` is available for `x`, then use that. (See §2.3.3 and §2.3.4.)
  - Otherwise, if `x == C` is valid, then `x is C` means `x == C`.
  - Otherwise, if `C` is an expression and `x is typeof(C)` is valid and true, then `x is C` means `x as typeof(C) == C`.
  - Otherwise, if `C(x)` is valid and convertible to `bool`, then `x is C` means `C(x)`.
  - Otherwise, if `x is Pointer` and `x as C` is valid, then `x is C` means `x as C is nullptr`. (See §2.2 for `as`, and §2.3.3 for the `operator is` customization for `Pointer` null checking.)
  - Otherwise, if `x is Pointer` and `C` is a `&`-qualified type, then `x is C` means `x as C` is valid and does not throw.
  - Otherwise, if `C<X>` is valid and convertible to `bool` or `C` is a specific type, then `x is C` means `typeof(x) is C`.
  - Otherwise, `x is C` is ill-formed.
- Otherwise, if `x` is a type:
  - If `C` is a specific type, then `x is C` means `std::is_same_v<C,x>`.
  - Otherwise, if `C` is a template-name, then `x is C` is `true` if `x` is a specialization of `C`, else `false`.
  - Otherwise, if `C<X>` is valid and convertible to bool, then `x is C` means `C<x>`.
  - Otherwise, `x is C` is ill-formed.

Notes There are five grammar productions for `is`-expressions: (1) type-id `is` type-id; (2) type-id `is` type-constraint; (3) expression `is` expression; (4) expression `is` type-id; (5) expression `is` type-constraint. Production 3 covers both "expression is value" and "expression is value-constraint."

Regarding `dynamic_cast`: We know statically which `is` cases are specified to mean `dynamic_cast` (via `as`; see §2.2). If a compiler has a way to turn off RTTI, then for those `is` cases it would do the same as if the user had tried to write `dynamic_cast` directly (typically, make it a warning or an error). However, in addition because a customized `is` is preferred to `dynamic_cast`, projects that disable RTTI now have the option of plugging something else in, and/or projects that have a more efficient form of dynamic casting for their type hierarchy (e.g., tagged hierarchies) now have the option of using those in portable code, and use them seamlessly under the common syntax `is`, whereas today they have to resort to a different spelling (they cannot customize `dynamic_cast`).

If `x` is decomposable using structured bindings, then `C` may be a composite condition of the form

```
x is [C1, C2, ..., Cn]
```

where this decomposition must be valid

```
auto&& [x1, x2, ..., xn] = x;
```

and then the meaning is

```
x1 is C1 && x2 is C2 && ... && xn is Cn
```

Note To avoid ambiguities, `is` should not be immediately followed by a lambda-introducer. To write a lambda after `is`, put it in parentheses.

2.1.1 `is` in variable declarations, and structured bindings

For generality, both structured bindings and `is` patterns should permit nested decomposition.[1] For example:

```
pair<int, pair<int,int> > data;
auto [a, [_, c]] = data;                    // A
if( data is [_, [1, _]] ) {...}             // B
```

In a variable declaration, an `is` constraint on the declared name(s) requires the `is` to be true.

- If the variable declaration is an `if` init-statement, then the init-statement is true if the `is` is true. For example, the above two lines A and B could be combined as follows:

  ```
  if (auto&& [a, [_, c]] is [_, [1, _]] = data) {...}  // checks that "is" is true
  ```

  with the additional relaxation that this code is valid even if we know statically that `data` does not decompose to `[_,[_,_]]` (in which case the branch is not taken). And a non-decomposing example:

  ```
  if (auto a is std::integral = f()) {...}             // checks that "is" is true
  ```

- Otherwise, the constraint is applied, and if false an exception is thrown. For example:

  ```
  auto&& [a, [_, c]] is [_, [1, _]] = data;            // requires "is" to be true
  ```

[1] As mentioned as a future direction for the structured bindings proposal [P0144R2] §3.9.

And a non-decomposing example:

```
auto a is std::integral = f();                  // requires "is" to be true
```

Notes In an `is`-expression, attributes are not permitted following `is`, and so decomposition is not ambiguous in examples like `data is [[x,y]]`.

When constraining to a specific type, the major difference between the two forms

```
int                a    = f();                  // C++20 explicit type/concept
std::integral auto b    = g();

auto a is int           = f();                  // "is" constraint
auto b is std::integral = g();
```

is how the requirement is applied. The first C++20 form statically requires that `f` returns an `int` or an object convertible to an `int` (and therefore might perform a conversion) and `g` returns something that satisfies `std::integral`. The second `is` form could be a static or dynamic constraint (if the latter, it could throw an exception if it fails; e.g., if `f` returns a `variant<int,string>` and the returned object does not currently hold an `int`) but if it succeeds it guarantees there is no conversion (it is just applying a constraint).

Similarly for these forms:

```
if (int                a    = f()) {...}       // C++20 explicit type/concept
if (std::integral auto b    = g()) {...}

if (auto a is int           = f()) {...}       // "is" constraint
if (auto b is std::integral = g()) {...}
```

The first form statically requires that `f` returns an `int` or an object convertible to an `int` (and therefore might perform a conversion) and `g` returns something that satisfies `std::integral`, and the branch is taken if the declared variable is not zero. The second `is` form could be a static or dynamic constraint (e.g., if the returned type is a `variant<int,string>`) and if it fails (e.g., if the variant does not currently hold an `int`) the branch is not taken, and if it succeeds the branch is then taken regardless of the value of the declared variable.

The second form is similar to the following, except that in the following `a` and `b` must be initialized before the `is` test:

```
if (auto a = f(); a is int          ) {...}    // "is" constraint, variation
if (auto b = g(); b is std::integral) {...}
```

Finally, putting the `is` in the initializer has its usual expression meaning. For example, given `v` of type `variant<pair<int,int>, string>`, this

```
auto flag = v is pair<int,int>;
```

has the same meaning as this C++20 version (see §2.3.5.1 for `std::variant` and `is`):

```
auto flag = v.index() == 0;      // ≈ std::holds_alternative<pair<int,int>>(v)
```

2.1.2 Example `is` uses (outside pattern matching)

In `if` conditions:

```
// users can write predicate generators as desired like this:
constexpr auto in(auto min, auto max)
    { return [=](const auto& x){ return min<=x && x<=max; }; }

// and directly like this if all the information is static:
template<auto min, auto max>
constexpr bool in(const auto& x) { return min<=x && x<=max; }

// and use them like this:
void test( auto x ) {
    if      (x is 3)                 { ... }    // specific value, x==3
    else if (x is in(1,2))           { ... }    // value predicate (or: in<1,2>)
    else if (x is std::pair<int,int>) { ... }    // specific type, pair<int,int>
    else if (x is std::pair)         { ... }    // type template-name, pair<_,_>
    else if (x is std::integral)     { ... }    // type predicate, integral<typeof(x)>
}
```

In `requires` clauses:

```
template< typename T, auto Size >
auto make_array() { ... };                  // unconstrained template

template< typename T, auto Size >
    requires Size is 3                      // specific value constraint
auto make_array() { ... };

template< typename T, auto Size >
    requires Size is in(1,2)                // value predicate constraint
auto make_array() { ... };

template< typename T, auto Size >
    requires (Size is std::pair<int,int>)   // specific type constraint
auto make_array() { ... };

template< typename T, auto Size >
    requires (Size is std::pair)            // type template-name constraint
auto make_array() { ... };

template< typename T, auto Size >
    requires Size is std::integral          // type predicate constraint
auto make_array() { ... };
```

`is` is not a globally reserved keyword, so it can appear as an identifier:

```
int is = 42;
assert(is is int);
```

2.2 General casting expression: as

as expressions provide a consistent syntax for casting (conversions and coercions), generalized to support static and dynamic casting including customized conversions (useful for dynamic types where static converting constructors and static conversion operators are usually not appropriate, such as optional, variant, any, future; see §2.3.2).

Let refto(T,x) be:

- If T is a reference type, then T.
- Otherwise, if x is an lvalue, then an lvalue reference T&.
- Otherwise (x is an rvalue), an rvalue reference T&&.

Let a cast as-expression be of the form:

```
x as T                                          // x is an expression
```

where as has the same precedence as is, P is a type predicate or specific type, the type of the expression is refto(T,x) unless otherwise specified, and:

- If std::is_same_v<T,typeof(x)>, then x as T means a reference to x.
- Otherwise, if x can be bound to a refto(T,x), then x as T means a refto(T,x) bound to x.
- Otherwise, if operator as<T>(x) or x.operator as<T>() is available, then use that. (See §2.3.3.)
- Otherwise, if typeof(x) is implicitly convertible to T, then x as T means to convert x to an rvalue of type T (e.g., including the case where both are Pointer types and this is a static upcast).
- Otherwise, if T(x) is valid and the result is dereferenceable using unary *, then x as T means *T(x).
- Otherwise, if dynamic_cast<refto(T,x)>(x) is well-formed, then x as T means dynamic_cast<refto(T,x)>(x).
- Otherwise, if dynamic_cast<T>(x) is well-formed, then x as T means dynamic_cast<T>(x).
- Otherwise, if !(x is Pointer) and T(x) is valid, then x as T means to convert x to an rvalue of type T using T(x) (e.g., including the case where both are Pointer types and this is a static upcast, and the case where T is a template-name and this uses CTAD).
- Otherwise, if x as T is ill-formed.

Notes There are two grammar productions for as-expressions: (1) expression as type-id; (2) expression as expression.

T(x) is an extractor extension point, and the one case where T is not a type.

If x is decomposable using structured bindings, then T may be a composite cast of the form

```
x as [C1, C2, ..., Cn]
```

where this decomposition must be valid

```
auto&& [x1, x2, ..., xn] = x;
```

and then the meaning is

```
forward_as_tuple ( x1 as C1, x2 as C2, ..., xn as Cn )
```

2.2.1 as in variable declarations (including structured bindings)

As in §2.1.1, both nested structured bindings and as patterns can be used in a variable declaration. For example, given

```
variant< pair<int,int>, string> v;
```

then this expresses a branch that is taken if the as is statically valid and dynamically succeeds, and any temporaries created are lifetime-extended to the scope of the if:

```
if (auto&& [a, b] as pair<int,int> = v) {...}          // checks "as" is valid and true
```

and this expresses a declaration that throws an exception if the cast is not valid or does not succeed at execution time:

```
auto&& [a, b] as pair<int,int> = v;                    // requires "as" to be true
```

Notes In an as-expression, attributes are not permitted following as, and so decomposition is not ambiguous in examples like data as [[x,y]].

When constraining to a specific type, the major difference between the two forms

```
int a = f();
auto a as int = f();
```

is that the first form statically requires that the returned value is an int or convertible to an int (and therefore might perform a conversion), whereas the second form always performs an explicit cast (and could throw an exception if it fails; e.g., if the returned type is a variant<int,string> and the returned object does not currently hold an int) but if it succeeds it guarantees there is no further implicit conversion.

Similarly, the major difference between the two forms

```
if (int a = f()) {...}
if (auto a as int = f()) {...}
```

is that the first form statically requires that the returned value is an int or convertible to an int (and therefore might perform a conversion) and the branch is taken if a is not zero, whereas the second form always performs an explicit cast (e.g., if the returned type is a variant<int,string>) and if it fails the branch is not taken, and if it succeeds it guarantees there is no further implicit conversion and the branch is then taken regardless of the value of a.

The second form is similar to the following, except that in the following a_ must be initialized before the as test:

```
if (auto a_ = f(); requires{a_ as int;}) {auto a = a_ as int; ...}
```

Finally, putting the as in the initializer has its usual expression meaning. For example, given v of type variant<pair<int,int>, string>, this

```
auto [a, b] = v as pair<int,int>;
```

has the same meaning as this C++20 version (see §2.3.5.1 for std::variant and as):

```
auto [a, b] = std::get<0>(v);     // ≈ std::get<pair<int,int>>(v)
```

2.2.2 Example as uses (outside pattern matching)

Let as be allowed in any expression context. For example, to perform an explicit argument conversion:

```
// Given: void f(std::string) { }
char a;
f(a);                       // error, constructor is explicit

// in C++20 we might try this:
f(string(a));               // error, function-style cast not allowed
f(string{a});               // ok, have to use { }

// with 'as' we could write:
f(a as string);             // ok, can clearly see (and grep for) explicit conversions
```

For example, to select a base explicitly:

```
namespace NS {
    struct A { void f() {} };
    struct B : A { };
    struct C { B i; };
}

struct D : NS::C {
    void foo() {
      this->NS::C::i.NS::A::f();                // using :: (today, a bit token-soupy)
      ((*this as NS::C).i as NS::A).f();        // using as (this section)
    }
};
```

as is not a globally reserved keyword, so it can appear as an identifier:

```
short as = 42;
assert(as as int == 42);
```

2.3 is/as and dynamic types: Customization and std

is and as support general constraints and casts, including for both static and dynamic typing.

C++ already includes language support for some dynamic typing via dynamic_cast and nullptr checking, and library support for dynamic typing such as smart pointers and sum types. For example, a raw or smart Pointer type such as T* or unique_ptr<T> can dynamically point to nothing (nullptr), to an object whose dynamic type is T, or to an object whose dynamic type is derived from T; and an optional<T> can dynamically contain nothing (nullopt) or an object of type T.

2.3.1 Binding and matching dynamic types

Decomposing a Pointer p that can point to an object of a type T (e.g., unique_ptr<T>, T*) is treated as a dereference using unary *p, and is treated as a dynamic is-a that matches (is true) if not p is nullptr.

In general, for all dynamic as-a typing (including a Pointer dereference or a myvariantobj as int), a binding that includes a dynamic pattern requires "is pattern" to be true, otherwise will throw an exception.

For example:

```
struct Node { unique_ptr<Node> left, right; int value; };

void f( Node& root ) {
    auto const& [_,_,v] = root;            // v binds to root.value

    auto const& [*[_,_,v2],_,_] = root;    // v2 binds to root.left->value if present,
                                           //    throws an exception if root.left is_null

    if( root is [*_,_,_] ) {               // aka "if( root.left )"
        auto const& [*[_,_,v3],_,_] = root;// v3 binds to (*root.left).value
```

2.3.2 Customizing operator is/as for dynamic types

Member or non-member operator is and operator as are customization points for types that have dynamic type relationships and conversions, meaning that their effective type is a dynamic property. Overloaded operator is side effects are ignorable.

Classes that use C++'s static typing and static type conversions never need to overload operator is or operator as. Such types "just work" with is and as expressions, which already use C++'s existing static type relationships (e.g., public derivation) and static conversions (implicit or explicit converting constructors/operators).

Note C++ library types tend not to support C++'s static conversion constructors and conversion operators when their type is a dynamic property, because then those functions would have to be able to fail by throwing an exception if the dynamic type conversion failed, and there is not a common spelling for testing in advance that the conversion is valid and will succeed. Instead, C++ libraries tend to provide their own divergently named functions for type tests (e.g., holds_alternative<T>(x), any_cast<T*>(&x)) and type conversions (e.g., get<T>(x), any_cast<T>(x)). The generalized operator is/as customization points regularize these ad-hoc conventions under a common spelling, and for existing types they are usually one-line passthroughs to the existing function. In the future, new dynamic types can just add both operator is and operator as, which are naturally used together by callers to first query is and then use as knowing it will be dynamically valid (or else that they can tolerate the exception).

2.3.3 Value query customization for null check

In namespace std, provide an "is null" customization to test any Pointer value with is nullptr:

```
template<Pointer P>
constexpr bool operator is(P const& p, std::nullptr_t) requires requires{(bool)p==true;}
    { return (bool)p==false; }

template<Pointer P>
constexpr bool operator is(P const& p, std::nullptr_t) requires std::regular<P>
    { return p==P{}; }
```

Notes This uses the contextual conversion to bool, when available, to avoid the temporary P{}.

This lets all indirection types (raw/smart pointer, or iterator) be directly tested to see if they hold a well-known null (default) value to prevent null-dereference errors. Other "not valid to dereference" errors are mitigated elsewhere: out-of-bounds by using span, and use-after-free via [P1179R1]).

This formulations supports optional, which has a pointerlike contextual conversion to bool.

2.3.4 Type query customization forms

A type can provide any of the following type query customization forms. Typically a type provides one form; if there is more than one, the implementation of is can choose the more efficient one (e.g., when doing multiple tests on the same expression, it would typically prefer indexed or type_info over single-type).

"Takes an object x" means as a this argument of a member function call (i.e., x.operator is) or as the first argument of a non-member function call (i.e., operator is(x)). This table shows the non-member syntax.

Type query form	**operator is**	**operator as** (if it cannot return the required type, throw an exception)
Single-type	Takes a static type T and an object x Returns a value B convertible to bool x is T means B converted to bool	Takes a static type T and an object x Returns an object X of type T by value or by reference x as T means X
type_info	Takes an object x Returns a std::type_info const& for the active type [2] x is T means operator is(x) is typeid(T)	(Same as single-type)
Indexed (Example)	Takes an object x Returns a std::integral value I that is the zero-based index of the active type, else -1 cast to that type (consecutive values from 0 to largest I for which sizeof(operator as<I>(x)) is valid)	Takes a static value I and an object x Returns an object of the I-th type (if active) by value or by reference x as T means if(x is T) operator as<I>(x) else throw an exception

[2] Kevlin Henney, original author of [Boost.Any] and coauthor of the std::any proposal [N3804], reports: "I don't recall if performance over any_cast was a driver for including the type function: the primary motivation was for table lookup, e.g., being able to use type_info as a key to look up, say, a type-appropriate callback. If you squint at it just right, that fits the pattern matching case nicely. That type comparison was also clearer in intent than using any_cast just to query whether something was of a particular type would have been another motivation."

`x is T` means `typeof(operator as<I>(x))> is T`

Note For the indexed case, this uses the return type of `operator as<I>(x)` to determine the `I`-th type, and the validity of `operator as<I>(X)` to determine the maximum `I`. My initial design had `operator is` return, not just an integral value, but an integral value that carried in its type the list of types it is an index into. For example:

```
// godbolt.org/z/ohd1EfvYs
template<typename ...Ts> struct current_type_info { size_t index; };
```

or, to emphasize that this isn't a wrapper, just a direct template alias for an integral value:

```
// godbolt.org/z/3MfnqndsT
template<typename ...Ts> using current_type_info = size_t;
```

Early reviewers asked me to just use the return values of `operator as` which were already available.

2.3.5 `operator is`/`as` for dynamic types in the standard library

This section defines `operator is` and `operator as` for the following `std::` types as one-line passthroughs:

std:: type	**for `x is T`, `operator is` invokes**	**for `x as T`, `operator as(x)` invokes**
`variant`	`x.index()`	`get<I>(x)`
`any`	`x.type()`	`any_cast<T>(x)` `*any_cast<T&>(&x)`
`optional`	`x.has_value()`	`x.value()`
`future` and `shared_future`[3]	`x.wait_for(chrono::seconds(0))` `  == future_status::ready`	`x.get()`
`shared_ptr`	`dynamic_pointer_cast<T>(x)`	`dynamic_pointer_cast<T>(x)`

For example, for a `variant<int,string> v`, this

```
if( v is int ) {                        // this "is" check ensures that
    x = v as int;                       // this "as" won't throw bad_variant_access
```

is equivalent to

```
if( std::holds_alternative<int>(v) ) {  // this "holds_alternative" check ensures that
    x = std::get<int>(v);               // this "get" won't throw bad_variant_access
```

2.3.5.1 `std::variant`

In namespace `std`, provide an indexed "is/as type" customization:

```
template<typename... Ts>
constexpr auto operator is( std::variant<Ts...> const& x )
    { return x.index(); }

template<size_t I, typename... Ts>
constexpr auto operator as( std::variant<Ts...> const& x ) -> auto&&
    { return std::get<I>( x ); }
```

[3] See [Wakely 2020], including active optimizations in libstdc++.

Note The auto&& return follows the design of std::get which returns effectively a reference. Note that the programmer can explicitly request an lvalue reference:

```
v as int            // for std::get<int>(a)
v as int&           // for std::get<int&>(a)
```

2.3.5.2 *std::any*

In namespace std, provide a type_info "is/as type" customization:

```
constexpr auto operator is( std::any const& x ) -> type_info const&
    { return x.type(); }

template<typename T>  requires (!std::is_reference_v<T>)
constexpr auto operator as( std::any const& x ) -> T
    { return std::any_cast<T>( x ); }

template<typename T>  requires std::is_reference_v<T>
constexpr auto operator as( std::any& x ) -> T& {
    if (auto p = std::any_cast<std::remove_reference_v<T>*>( &x )) { return *p; }
    throw std::bad_any_cast;
}
```

Note These overloads, both returning T but for one of which that is a reference type, follows the design of any_cast to support both the "by value" and "by pointer" forms:

```
a as int      // for any_cast<int>(a)
a as int&     // for *any_cast<int*>(&a)
```

I think this is a simpler and clearer interface for any, and also lets variant, optional, and future (see below) be used with the consistent interface of the two spellings

- `x as T` to get whichever of value or reference is the designed default for the type, to be used if you don't care; and
- `x as T&` to explicitly get an lvalue reference.

2.3.5.3 *std::optional*

In namespace std, provide a single-type "is/as type" customization:

```
template<typename T>
constexpr auto operator is( std::optional<T> const& x ) -> bool
    { return x.has_value(); }

template<typename T>
constexpr auto operator as( std::optional<T> const& x ) -> auto&&
    { return x.value(); }
```

Note The auto&& return follows the design of .value() which returns by & or && reference.

2.3.5.4 *`std::future` and `std::shared_future`*

In namespace `std`, provide a single-type "is/as type" customization: (`FUTURE` is `std::future` or `std::shared_future`)

```
template<typename T>
constexpr auto operator is( FUTURE<T> const& x ) -> bool
    { return x.wait_for(std::chrono::seconds(0)) == std::future_status::ready; }

template<typename T>
constexpr auto operator as( FUTURE<T> const& x ) -> T
    { return x.get(); }
```

Notes The `T` returns follow the design of `.get()` which returns by `&` or `&&` reference, to support both the "by value" and "by reference" forms:

```
f as int            // for future<T>::get() -> T
f as int&           // for future<T&>::get() -> T&
```

This is the `wait_for` pattern that is currently used to work around the absence of `is_ready`, which was in the Concurrency TS but is not in the standard.

2.3.5.5 Smart pointers

In namespace `std`, provide "as type" customizations:

```
template<typename T, typename U>     // for dynamic conversion to another shared_ptr
constexpr auto operator as( std::shared_ptr<U> const& x ) -> auto&&
    { return dynamic_pointer_cast<T>(x); }

template<typename T, Pointer P>      // for dynamic conversion to a raw pointer
constexpr auto operator as( P const& x ) -> T
    requires requires {x.get() == true;}
    { return dynamic_cast<T>(x.get()); }
```

Notes We don't need a customized `operator is` because for Pointer types `is` in defined in terms of `as`.

The second form is for code like the following:

```
inspect (smart_or_raw_ptr_to_widget_base) {
    p as Button*   => handle_button  (p);
    p as ComboBox* => handle_combobox(p);
}
```

or equivalently:

```
if (Button* p = smart_or_raw_ptr_to_widget_base as Button*) {
    handle_button(p);
} else if (ComboBox* p = smart_or_raw_ptr_to_widget_base as ComboBox*) {
    handle_combobox(p);
}
```

2.3.6 Cleaner form for structured bindings' "tuple-like" customization

The structured bindings proposal [P0144R2] added a tuple-like customization point in response to EWG direction. Using it has been cumbersome (e.g., [P1096R0], [P0326R0], [P0327R3]). Overloading `operator as` could let us do better. Consider these two types (thanks to Ville Voutilainen for this example):

```
struct PlainRect {
    Point topLeft;
    int   width;
    int   height;
};

class EncapsulatedRect { // ...
    Point const& topLeft() const;
    int          width  () const;
    int          height () const;
};
```

Structured bindings directly supports `PlainRect`, but not `EncapsulatedRect`:

```
auto [tl, w, h] = PlainRect();                  // ok

auto [tl, w, h] = EncapsulatedRect();           // error, unless customized
```

To make the second line be not an error, today we must write the following customization ceremony (Godbolt):

```
// This is what we need to write today to make the above structured bindings work.
// Here highlighting the essential information payload, the rest is all ceremony:
namespace std {
    template<> struct tuple_size   <   EncapsulatedRect> { enum { value = 3 }; };
    template<> struct tuple_element<0, EncapsulatedRect> { using type = Point const&; };
    template<> struct tuple_element<1, EncapsulatedRect> { using type = int;   };
    template<> struct tuple_element<2, EncapsulatedRect> { using type = int;   };
}
template<int I>
auto get(EncapsulatedRect const& er)
    -> constexpr typename std::tuple_element<I, EncapsulatedRect>::type
{
    if      constexpr(I == 0) return er.topLeft();
    else if constexpr(I == 1) return er.width();
    else if constexpr(I == 2) return er.height();
}
```

If we added a customization point for structured bindings to consider a well-known function name (any name will do, but if we add "operator as" anyway that could be the name as an additional unary form) that returned something bindable (e.g., a `tuple` or `struct`), we could replace the above code with a single function, including as a member function provided by the type author and with access to privates as needed. For example, a `tuple`:

```
constexpr auto /* "destructure" or "operator as", some recognized name */ () const
    -> std::tuple<Point const&, int, int>
    { return { topLeft(), width(), height() }; }
```

Or a `struct`:

```
constexpr auto /* "destructure" or "operator as", some recognized name */ () const {
    struct Ret { Point const& tl; int w; int h; };
    return Ret { topLeft(), width(), height() };
}
```

This carries exactly the same information payload, but more densely (with less ceremonial boilerplate). It can also be provided as a non-member function.

Notes Both `tuple` and a struct work fine because they transform the type into something bindable. Using `std::tuple` is a good choice for such a custom adaptation because its bindings never copy and it works with existing libraries that understand and manipulate `std::tuple`.

The above is just a more straightforward way to write today's existing customization point. If we want to go further, then as Bjarne Stroustrup points out, the logical minimum is something like this, which can be viewed as a jump table (similar to a vtbl) – the most general form, ideally provided by the class author:

```
structure_map (EncapsulatedRect) { topLeft, width, height };
```

The above code approaches this in conciseness and that the class author can provide appropriate mapping of their type to a `tuple` or `struct`.

Looking forward, perhaps the right general future language feature that also naturally enables this mapping is *multiple return values*, as suggested by Gor Nishanov and others. There are many motivating use cases already including in the standard library (e.g., `map::insert` returns `-> pair<iterator,bool>` because it cannot directly return `iterator, bool`). If future C++ gets multiple return values, then we could write the conversion even more directly and with greatest efficiency by leveraging guaranteed copy elision (GCE) using the same rules as today and applying them individually to each return value instead of just the current-singular return value:

```
// If in the future C++ allowed multiple return values (strawman syntax)
auto operator as() const -> (Point const&, int, int)
    { return topLeft(), width(), height(); }   // now with GCE efficiency

// Possibly this with suitable return type deduction, and equal efficiency:
auto operator as() const { return topLeft(), width(), height(); }
```

So I am optimistic that the cleanup direction proposed in this section is on a straight-line path from what we have today (where it expresses the same thing with less ceremony) to the ideal language solution that has direct bindings and minimum copies.

3 Pattern matching using `is` and `as`

3.1 Core approach: Expressing alternatives using `is` and `as`

Let pattern matching be written using an `inspect` statement or expression, with an optional explicit result type:

```
inspect ( expr )  -> Result opt  {  /* alternatives */  }
```

where

expr is the expression to be inspected, and may be a comma-delimited list of subexpressions treated as `forward_as_tuple(`*expr*`)`.

Result is the result type; if omitted, it is deduced from the alternatives using the same rules as return type deduction, ignoring alternative results that are `noreturn` blocks or `throw` expression statements.

The body comprises a series of one or more *alternatives* of one of the forms

```
names opt  is C     =>  result
names opt  as T     =>  result
names opt  if Cond  =>  result
```

where the alternatives are considered successively until the first match is found, and:

names optionally introduces either a single identifier or a set of `[` *decomposed-names* `]`. Writing *names* [`is` *C* | `as` *T*] introduces the names the same as `auto&&` *names* [`is` *C* | `as` *T*] = *expr*`;` (using structured bindings if decomposed). For any unused component, use the "don't care" wildcard `_`.

`is` *C* means the alternative is selected if *expr* `is` *C* is well-formed and evaluates to true, and any decomposition in *names* is valid.

`as` *T* means the alternative is selected if *expr* `is` *T* is well-formed and evaluates to true, and any decomposition in *names* is valid, and *expr* `as` *T* is well-formed, and in this alternative an introduced name for *expr* binds to *expr* `as` *T*.

`if` *Cond* means the alternative is selected if the boolean condition *Cond* evaluates to true.

result defines the value of the `inspect` expression if that alternative is selected, and can be a block or else a single-expression statement ending in `;` where the expression is the value produced.

When `inspect` appears at the start of a statement, it is an `inspect` statement and must not specify a result-type. Otherwise, it is an expression that has a deduced or explicit result-type; must not contain `break`, `continue`, or `return`; and must contain an `is _` alternative.

Notes A deduced result type uses the same rules as for a deduced function return type.

To explicitly exhaustively enumerate all alternatives, make the final alternative

```
is _ => noreturn { /* assert(false) or similar */ }
```

or we could provide a direct explicit way to spell this, such as `nodefault;` .

`inspect constexpr` requires *expr* and all alternative conditions to be compile time constant expressions. All non-selected alternative results are discarded, and their `return` statements do not participate in function return type deduction.

3.2 Advanced patterns: is/as combinations, dereference

3.2.1 is/as combinations

Recall that in `inspect (x)`:

- `is C` does not change type or identity: introduced name(s) bind to all/components of `x`.
- `as T` does change type and/or identity: introduced name(s) bind to all/components of `x as T`.

In a sequence of `is` and `as`, each successively refers to the result of the previous (i.e., the most recent `as`, since `is` does not modify the type/identity), including that bound names bind to the last `as`. For example:

```
inspect (e) {

  a is X as Y => ...
    // if e is X,
    // then if e as Y is valid do that,
    // then select this alternative, a binds to the Y

  b as V is W => ...
    // if e as V is valid do that,
    // then if the V (V or V&) is W,
    // then select this alternative, b binds to the V

}
```

For example, given:

```
tuple<optional<WidgetBase**>*, optional<WidgetBase**>*> v = ... ;

inspect (v) {
```

We can test whether `v` has two fields that are `Label` (a class that inherits from `WidgetBase`) widgets that have the texts `"Hello"` and `"world!"` in them by writing a boolean condition that combines `as` and `is`:

```
[****a, ****b] if (a as Label) is "Hello" && (b as Label) is "world!" => ...
```

or by combining `as` and `is` in the pattern:

```
as [****Label is "Hello", ****Label is "world!"] => ...
```

3.2.2 Dereference

`Pointer` types, including iterators, can be dereferenced in patterns using `*`, and match if the contextual conversion to `bool` evaluates to `true`. For example:

```
pair< variant< unique_ptr<Node>, Node*, double*, string >, int > v = ... ;

inspect (v) {
    is [*Node,_] => ...       // first element is either unique_ptr<Node> or Node*
    is [*,_]     => ...       // first element is dereferenceable (therefore double*)
    is _         => ...       // first element is anything (therefore string)
}
```

Note In this proposal, `[ ]` is used only for structural has-a decomposition, which is always lifetime-safe. It is tempting, but I think wrong, to use `[ ]` also for points-to-a decomposition, aka dereferencing, because dereferencing a `Pointer` is inherently a more generous operation having failure modes that are not checkable, notably use-after-free (e.g., a pointer or iterator that points to something that has been deallocated, which must not be dereferenced) and out-of-bounds (e.g., a pointer or iterator to one-past-the-end of an array, which can be formed but must not be dereferenced). In general there is no way for an `inspect` alternative to check automatically whether a `Pointer` is safe to dereference before attempting the deference, and so using a separate syntax, here `*`, makes the dangerous "trust me" operation visible. (For a more general approach to statically ensuring a Pointer is safely deferenceable, which also works for `inspect` uses but is not limited to `inspect`, see [P1179R1].)

3.3 Conveniences: ||, &&, {}

The following options are syntactic conveniences. None increases the fundamental expressive power we already saw, but each makes it more convenient to write common code patterns (and follow "don't repeat yourself") by removing the need to redundantly repeat the same constraint or result. These can be combined to remove multiple forms of redundancy.

3.3.1 Multiple alternatives that have the same result: ||

When multiple successive alternatives use the same result, the common result can be written by using the syntactic sugar `||`.

Note This is useful in two ways: (1) To save repeating the name of the expression being matched in top-level `is` alternatives. For example, we can already write this without any sugar as an ordinary single expression:

```
inspect (x) { x is Foo || x is Bar => result; }      // single expr, no sugar
```

and this feature allows the don't-repeat-yourself convenience of writing the same as:

```
inspect (x) { is Foo || is Bar => result; }          // top-level || sugar
```

(2) To allow introducing the same set of names to refer to different components for a common result, as in the red/black tree rebalancing examples, which is rarely needed but when you need it is useful to bind a common result to different names without repeating the common result body. For example, we can already write this without any sugar:

```
inspect (x) {
    [a,_] is [_,0] => result(a);        // two alternatives, no sugar
    [_,a] is [0,_] => result(a);
    // ...
}
```

and this feature allows the don't-repeat-yourself convenience of writing the same as:

```
inspect (x) {
    [a,_] is [_,0] ||                   // top-level || sugar
    [_,a] is [0,_] => result(a);
    // ...
}
```

The semantics are that when successive alternatives of the form `is` *C* or `if` *Cond* have common results, for example:

```
names₁ is Foo   => result
names₂ if Cond  => result
    ...         => result
namesₙ as Bar   => result
```

where

result is identical in all three cases

names₁, *names₂*, and *namesₙ* introduce the same set of names with the same types
(possibly with different decompositions, such as `[a,_]` vs `[_,a]`)

Note The use cases we currently have for this are satisfied with each name having the same type in each alternative. We could relax this requirement so that the semantics are the shared result is de-sugared (as a rewrite rule) and the compiler spits the redundant result expression into each alternative as if it had been written that way, and then do name lookup and binding appropriately. I think we should start with the "same type" restriction for now until we find compelling use cases, and nothing prevents relaxing this in the future.

then we can replace each non-last `=>` *result* with `||`, with the same meaning:

```
names₁ is Foo  ||
names₂ if Cond ||
    ...        ||
namesₙ as Bar => result
```

Notes An important observation is that the ability for non-first subexpressions to introduce additional names can be viewed as "a logical-expression with the added power to introduce names." Importantly, that ability to introduce names inherently adds a fundamental restriction on how they can be combined, because given predicates `A` and `B` each of which now has the power to introduce names to be used in the result:

- for `A || B`, allowing `B` to introduce additional names naturally has the requirement that `A` and `B` must introduce all the same names with the same types (as above, otherwise we can't write the common result sensibly)
- for `A && B`, allowing `B` to introduce additional names naturally has the requirement that `A` and `B` must *not* introduce any of the same names more than once (otherwise they conflict)
- (future) for mixed chains of `||` and `&&` that could introduce additional names inside the expression, the natural rule would be that all introduced names must be written together at the beginning of the combined alternative, and not allow subparts to introduce names.

We have motivating examples for chains of all-`||` (alternatives) and all-`&&` (sequencing, see next section). We don't currently have compelling motivating examples for mixed chains of `||` and `&&`, so we shouldn't try to support those until we do have compelling examples, but there is nothing in the current design that gets in the way of evolution in that direction later.

Earlier *namesᵢ* are in scope in later parts of the same `||`'d alternative, so if successive `||`'d `is` alternatives specify identical decompositions then all but the first are redundant.

3.3.2 Grouping common names and constraints: { }

When multiple successive alternatives share a common subset of constraints, the common subset can be written once by using the syntactic sugar { } to group them, for example:

```
names_opt is C {
    // alternatives considered if "expr is C"
}
names_opt as T {
    // alternatives considered if "expr as T" is valid
}
names_opt if Cond  {
    // alternatives considered if "Cond"
}
```

Groups can be nested, as in:

```
names_1 is Foo {
    names_2 if Cond {
        ... {
            names_N as Bar => result
        }
    }
}
```

where

names$_1$, *names*$_2$, and *names*$_N$ introduce nonoverlapping names
(possibly introducing multiple names for the same component)

Earlier *names*$_i$ are in scope in nested groups, so if nested groups do not specify two names for the same component, then they can equivalently be merged and written together in the outermost group.

Notes This is useful mainly to not repeat common prefixes by factoring them out. For example, we can already write this without any sugar:

```
inspect (x) {
    // ...
    s is Shape && if s.value() > 42 => result1(s);   // no sugar
    s is Shape && if fobnozz(s)     => result2(s);
    // ... continue considering alternatives here if none of the above match
}
```

and this feature allows the don't-repeat-yourself convenience of writing the same as:

```
inspect (x) {
    // ...
    s is Shape {                                     // grouping sugar
        if s.value() > 42 => result1(s);             // equivalent to above
        if fobnozz(s)     => result2(s);
    }
    // ... continue considering alternatives here if none of the above match
}
```

Note this is different from a nested `inspect` within a result, such as:

```
inspect (x) {
    // ...
    s is Shape => {                                 // legal too, but NOT
        inspect(s) {                                // equivalent to above,
            if s.value() > 42 => result1(s);        // expresses something
            if fobnozz(s)     => result2(s);        // different
        }
    }
    // ... continue considering alternatives here only if “is Shape” is false
}
```

because the latter would enter the `is Shape` alternative set always if `is Shape` is matched. Visually, following a `=>` is a “hard stop” for the current `inspect`, which is correct.

Some reviewers expressed concern that this looks too much like a block, and suggested adding `&&` before the `{` could be a visual reminder that these have “and” semantics.

3.3.3 Multiple constraints required for a result: &&

When multiple successive `is` *C* and `if` *Cond* constraints are required for a result, they can be written using the syntactic sugar `&&`.

The semantics are that when nested constraints lead to a single result:

```
names₁ is Foo {
    names₂ expr {
        ... {
            namesₙ as Bar => result
        }
    }
}
```

then we can replace each `{ }` using `&&`, with the same meaning:

```
names₁ is Foo &&
names₂ expr   &&
    ...       &&
namesₙ as Bar => result
```

Note As already noted for nested `{ }`, the names introduced by $names_i$ must be nonoverlapping.

Earlier $names_i$ are in scope in later parts of the same `&&`’d alternative, so if successive `&&`’d `is` alternatives do not specify two names for the same component, then they can equivalently be merged and written together first.

3.4 Examples

3.4.1 Examples: C-style switch

This is equivalent to a C `switch` statement except the alternatives are scoped and do not have fallthrough:

```
inspect (x) {
    is 0 => cout << "none";
    is 1 => cout << "one";
    is 2 => cout << "two";
    is 3 => cout << "three";
    is _ => cout << "many";
}
```

This is the same written as a subexpression:

```
cout << inspect (x) {
            is 0 => "none";
            is 1 => "one";
            is 2 => "two";
            is 3 => "three";
            is _ => "many";
        };
```

3.4.2 Examples: First match

The first match is selected:

```
auto count(int i) {
    cout << inspect (i) {                 // because the constraints are ordered
        is in(1,2) => "1 or 2";           // the value 2 will match here...
        is in(2,3) => "3";                // ... and not here
        is _       => "something else";
    };
}
```

3.4.3 Examples: Consistency with outside `inspect`

If we did have `is` and `as` elsewhere in the language, everything would behave consistently:

```
void test( auto x ) {
    inspect (x) {
        is Integral => { ... }
        is in(1,2)  => { ... }
        is 3        => { ... }
    }
}
```

Including that `inspect` alternatives can be constrained by compile-time type constraints and/or run-time value constraints:

```
inspect (x) {
    is Number     => { cout << "some type of number"; } // type predicate match
```

```
    is string     => { cout << "a string"; }           // specific type match
    is in(1,2)    => { cout << "1 or 2"; }             // value predicate match
    is 3          => { cout << "3"; }                  // specific value match
    if x<10       => { cout << "<10, but not 1 2 3"; } // boolean condition match
    is _          => { cout << "something else"; }     // match anything
}
```

3.4.4 Examples: Decomposition

For example:

```
struct S { int i; double j; };

S s;

inspect (s) {
    [_,b] is [0,_] => cout << fmt("s.i is zero, s.j is {}", b);
          is _     => cout << fmt("s.i is {}, s.j is {}", s.i, s.j);
}
```

Similarly, multiple values can be passed to `inspect` and treated as a composite:

```
int i; double j;

inspect (i, j) {
    [_,b] is [0,_] => cout << fmt("i is zero, j is {}", b);
          is _     => cout << fmt("i is {}, j is {}", i, j);
}
```

3.4.5 Examples: Start of statement

When `inspect` appears at the start of a statement, it is a statement. No trailing `;` is required. For example:

```
inspect (x) {
    is "xyzzy" => 0;
    is "plugh" => 1;
}
```

To use an `inspect` subexpression in the leftmost position of a more complex expression, enclose it in `( )`. For example:

```
(inspect (x) -> widget& {                  // assign 42 to one of two widgets
    is "xyzzy" => first_widget;
    is _       => some_other_one;
}) = 42;
```

3.4.6 Examples: `std::` dynamic types (`variant`, `any`, `optional`, `future`)

These all work via overloaded `std::operator is`/`as` (see §2.3.2).

```
void f1(variant<int,string> const& x) {
    inspect (x) {
        s as string => cout << "string \"" + s + "\"";
          is _      => cout << "not a string";
    }
```

```
}

void f2(any const& x) {
    inspect (x) {
        s as string => cout << "string \"" + s + "\"";
          is _      => cout << "not a string";
    }
}

void f3(optional<string> const& x) {
    inspect (x) {
        s as string => cout << "has value \"" + s + "\"";
          is _      => cout << "((nullopt))";
    }
}

void f4(future<string> const& x) {
    inspect (x) {
        s as string => cout << "ready: \"" + s + "\"";
          is _      => cout << "Æthelred the Unready";
    }
}
```

One thing that makes a uniform interface powerful is that it enables writing generic code, including that it works with concepts and other type predicates. For example:

```
// this generic function works uniformly when x is any type, including any
// of variant, any, optional<int>, optional<string>, future<int>, future<string>, ...

void f(auto const& x) {
    inspect (x) {
        i as int           => cout << "int " << i;
          is std::integral => cout << "non-int integral " << x;
        s as string        => cout << "string \"" + s + "\"";
          is _             => cout << "((no int or string value))";
    }
}
```

3.4.7 Ville's customization example

Ville Voutilainen suggested the following problem as a test case for customizability: "Consider pattern-matching a variant<X,Y,Z> [call this type V] where the types are classes in a polymorphic class hierarchy, and in some cases it's possible to cross-cast between them with dynamic_cast. I might want to write a match that could convert via static conversions, but not via dynamic_cast. What do I do?"

The first question is: How would you write that today? Presumably something like this:

```
template<typename W>
constexpr bool ville_convertible( variant<X,Y,Z>& v ) {
    if constexpr( is_convertible_v<X,W> ) { if (v.index() == 0) return true; }
    if constexpr( is_convertible_v<Y,W> ) { if (v.index() == 1) return true; }
    if constexpr( is_convertible_v<Z,W> ) { if (v.index() == 2) return true; }
    return false;
}
```

We can use this directly as a predicate:

```
inspect (x) {
    is ville_convertible<W> => cout << "yes, x passes Ville's match test";
}
```

Then if we want to use it "as" that type we can write a named conversion:

```
template<typename W>
constexpr auto ville_convert( variant<X,Y,Z>& v) -> W& {
     // as you wish
}
```

and use that in the result:

```
inspect (x) {
    is ville_convertible<W> => ville_convert<W>(x).some_w_function();
}
```

In this example, that may be all we want, so the match and conversion are explicit as shown.

But if we did want to go further, and allow writing an implicit form as

```
inspect (x) {
    w as W => w.some_w_function();   // let's say we wanted to allow this option
}                                    // when our overloaded is/as are in scope
```

then we can overload is and as to enable that (while still allowing the explicit form above) as follows:

```
template<typename W>
constexpr bool operator is( variant<X,Y,Z> const& v ) {
    return ville_is_convertible_to<W>(v);
}

template<typename W>
constexpr auto&& operator as( variant<X,Y,Z> const& v ) {
    return ville_convert<W>(v);
}
```

3.4.8 David Sankel's examples

This paper's approach grammatically separates introduced names from uses of existing names as constraints. For example, instead of introducing and using names in the same grammar position, such as

```
// P1371 example
[col, case Red] => ...        // col is a new introduced name, Red is an existing name
```

this paper's approach separates them grammatically, so all introduced names come first before `is`, `as`, or `if`, and all existing names used as constraints come after:

```
// This paper's equivalent
[col, _] is [_, Red] => ...   // col is a new introduced name, Red is an existing name
```

This separation has several advantages.

(1) It is naturally **unambiguous**. Because the two can never appear in the same grammar position, we never need a syntactic disambiguator to distinguish whether we want to introduce a name rather than refer to an existing one.

(2) It naturally allows **naming and constraining the same component**. This arises especially with predicate constraints. For example, consider inspecting a `Point`:

```
[x,y] is [even,_] => f(x, g(y));
```

Here, even though the first component is constrained by `even`, we still want to introduce a name `x` for the same component so we can refer to it, because we do not know its exact value.

(3) It naturally allows **naming and constraining to take different shapes**. As always with decoupling, by separating two things we enable them to each independently take their own most natural shape for a given situation. For example, again inspecting a `Point`:

```
[x,y] is inside_an_existing_shape => f(x, g(y));
```

Here, we are writing a constraint that applies to the whole value, but we are also decomposing it to refer to the components so we can use them individually… not only in the result, but possibly later in the alternative pattern itself, such as:

```
[x,y] is inside_an_existing_shape && if sin(x)>0.5 => f(x, g(y));
```

3.4.9 Michael Park's examples (patterns outside `inspect`)

I asked Michael Park about the usefulness of a general `match` facility along the lines of [P1260R0] section 4.5… there, `match` is exposition-only, but what if it were a generally usable facility outside `inspect`, so we could use it elsewhere such as `if` and `requires`? Park suggested the following examples.

Consider this use of such a `match`:

```
// Example 1: P1260-like syntax
if (match([let x, 0], point)) {
    // use x
}
```

In this paper's syntax, that would be expressed using `auto` and structured bindings (including cv/ref-qualified as desired):

```
if (auto [x,_] is [_,0] = point) {         // or auto&, etc.
    // use x
}
```

That is, this a structured binding augmented with an “is pattern” clause. (And the _ wildcard.) This way we can get the goodness of patterns generally in the language.

Next, consider:

```
// Example 2: P1260-like syntax
if (cond || match([let x, 0], point)) {
    // we get here if cond is true, is x in scope?
}
```

In this paper’s syntax, that would be expressed as:

```
if (auto [x,_] is [_,0] = point; cond) {
    // we get here if the pattern matches and cond is true, x is in scope
}
```

Note This is similar to the following non-decomposition example:

```
if (auto x is integral = f()) {
    // we get here if the pattern matches, x is in scope
}
```

which expresses something different from the C++20 code

```
if (integral auto x = f()) {     // compile-time error if f() is not integral
    // ...
}
```

Finally, consider:

```
// Example 3: P1260-like syntax
if (!match([let x, 0], point)) {
    // this executes if the pattern does not match, but x is in scope?
}
```

In this paper’s syntax, that could be literally written as:

```
if (auto [x,_] is [_,0] = point) {   // likely get “unused binding” warning here
    // intentionally left blank
} else {
    // this executes if the pattern does not match, x is naturally not mentioned
}
```

but it would be most naturally expressed as:

```
if (!(point is [_,0])) {
    // this executes if the pattern does not match, x is naturally not mentioned
}
```

3.4.10 Sergei Murzin's example

Handling state transition maps:

```
using State = std::variant<PendingInit, Data, Deleted>;
State old = ...;
State new = ...;
inspect (old, new) {
  [_, initialState]  is [PendingInit, _] => processInit(initialState);
  [oldData, _]       is [Data, Deleted]  => processDelete(oldData);
  [oldData, newData] is [Data, Data]     => processDiff(oldData, newData);
  [_, newState]      is [Deleted, _]     => processFromDeleted(newState);
  _                                      => reportUnexpectedTransition(old, new);
};
```

3.4.11 Customizing is and as for [LLVM Variant]

The following customization in namespace `llvm::pdb` would enable [LLVM Variant] to efficiently work with `is` and `as`.

```
constexpr auto operator is(Variant const& x) -> std::underlying_type_t<PDB_VariantType> {
    return x.Type<2 ? -1 : x.Type-2 ;
}

template<std::underlying_type_t<PDB_VariantType> I>
constexpr auto operator as(Variant const& x) {
    if (I < 0 || I != x.type-2) throw bad_variant_access();
    return inspect constexpr (I) {
        is  0 => x.Int8;
        is  1 => x.Int16;
        is  2 => x.Int32;
        is  3 => x.Int64;
        is  4 => x.Single;
        is  5 => x.Double;
        is  6 => x.UInt8;
        is  7 => x.UInt16;
        is  8 => x.UInt32;
        is  9 => x.UInt64;
        is 10 => x.Bool;
        is 11 => x.String;
    };
}
```

3.5 Tony tables: Side by side with other proposals/languages

3.5.1 Michael Park's EWG 2021-04-08 slide examples

Alternatives explored	This paper (proposed)

// R0: Annotate id-expr with ^
inspect (e) {
  name => ...
  0 => ...
  ^value => ...
  [x, y] => ...
  [0, ^b] => ...
  [^a, ^b] => ...
  [x, y, z, ^a] => // more names
  [^a, ^b, ^c, x] => // more refs
  <Circle> circle => ...
  <Rectangle> [width, height] => ...
};
// Pin (^) operator in Elixir
// Weird looking simple uses
enum Color { Red, Blue };
inspect (color) {
  ^Red => ...
  ^Blue => ...
};
	inspect (e) {
	  name is _ => ...
	  is 0 => ...
	  is value => ...
	  [x, y] is _ => ...
	  is [0, b] => ...
	  is [a, b] => ...
	  [x, y, z, _] is [_, _, _, a] => // more names
	  [_, _, _, x] is [a, b, c, _] => // more refs
	  circle as Circle => ...
	  [width, height] as Rectangle => ...
	};
	
	enum Color { Red, Blue };
	inspect (color) {
	  is Red => ...
	  is Blue => ...
	};

// R1/R2: let/case recursive, let default
inspect (e) {
  name => ...
  0 => ...
  case value => ...
  [x, y] => ...
  [0, case b] => ...
  case [a, b] => ...
  [x, y, z, case a] => // more names
  case [a, b, c, let x] => // more refs
  <Circle> circle => ...
  <Rectangle> [width, height] => ...
};
// let from Rust, Swift, many others
// + `switch` looking simple uses
enum Color { Red, Blue };
inspect (color) {
  case Red => ...
  case Blue => ...
};
// - Recursing gets complex with nesting
	// (same as above, plus extra example at the end)
	inspect (e) {
	  name is _ => ...
	  is 0 => ...
	  is value => ...
	  [x, y] is _ => ...
	  is [0, b] => ...
	  is [a, b] => ...
	  [x, y, z, _] is [_, _, _, a] => // more names
	  [_, _, _, x] is [a, b, c, _] => // more refs
	  circle as Circle => ...
	  [width, height] as Rectangle => ...
	};
	
	enum Color { Red, Blue };
	inspect (color) {
	  is Red => ...
	  is Blue => ...
	};

```
inspect (e) {
  let [a, case [let b, c], [d]] => ...
};
```

```
inspect (e) {
    [a, [b, _], [d]] is [_, [_, c], _] => ...
};
```

```
// R3: Annotate id-expr with case
inspect (e) {
  name => ...
  0 => ...
  case value => ...
  [x, y] => ...
  [0, case b] => ...
  [case a, case b] => ...
  [x, y, z, case a] => // more names
  [case a, case b, case c, x] => // more refs
  <Circle> circle => ...
  <Rectangle> [width, height] => ...
};
// + `switch` looking simple uses
enum Color { Red, Blue };
inspect (color) {
  case Red => ...
  case Blue => ...
};
// + Easier to read in nested patterns
inspect (e) {
  [a, [b, case c], [d]] => ...
};
// + No need to introduce let.
// - Abusing case
// - Expressions should be expressions
// - Declaration of names should look
// more like a declaration
// (lambda captures were a mistake?)
```

```
inspect (e) {
  name is _ => ...
  is 0 => ...
  is value => ...
  [x, y] is _ => ...
  is [0, b] => ...
  is [a, b] => ...
  [x, y, z, _] is [_, _, _, a] => // more names
  [_, _, _, x] is [a, b, c, _] => // more refs
  circle as Circle => ...
  [width, height] as Rectangle => ...
};

enum Color { Red, Blue };
inspect (color) {
  is Red => ...
  is Blue => ...
};

inspect (e) {
  [a, [b, _], [d]] is [_, [_, c], _] => ...
};
```

```
// R4?: Annotate id-part instead?
inspect (e) {
  let name => ...
  0 => ...
  value => ...
  [let x, let y] => ...
  [0, b] => ...
  [a, b] => ...
  [let x, let y, let z, a] => // more names
  [a, b, c, let x] => // more refs
  <Circle> let circle => ...
  <Rectangle> [let width, let height] => ...
};
// Clean simple use cases
enum Color { Red, Blue };
```

```
inspect (e) {
  name is _ => ...
  is 0 => ...
  is value => ...
  [x, y] is _ => ...
  is [0, b] => ...
  is [a, b] => ...
  [x, y, z, _] is [_, _, _, a] => // more names
  [_, _, _, x] is [a, b, c, _] => // more refs
  circle as Circle => ...
  [width, height] as Rectangle => ...
};

enum Color { Red, Blue };
```

```
inspect (color) {
  Red => ...
  Blue => ...
};
// Easier to read through deep nesting
inspect (e) {
  [let a, [let b, c], [let d]] => ...
};
// + No longer abusing case
// + Expressions are expressions
// + Declaration of names have
// an indication of a declaration.
// - Apparent inconsistency with
// structured bindings.
// * No longer optimizing for what
// some believe is more common.
```

```
inspect (color) {
  is Red => ...
  is Blue => ...
};

inspect (e) {
  [a, [b, _], [d]] is [_, [_, c], _] => ...
};
```

```
// auto
inspect (e) {
  auto name => ...
  0 => ...
  value => ...
  auto&& [x, y] => ...
  [0, b] => ...
  [a, b] => ...
  [auto x, auto const& y, auto&& z, a] => // ...
  [a, b, c, auto&& x] => // more refs
  <Circle> auto&& circle => ...
  <Rectangle> auto&& [width, height] => ...
};
// + familiar syntax, no new keyword
// - complexity, + finer lifetime control
// - potential unintentional copies
// - more verbose
// - bindings are all or nothing, tied to SB
```

```
inspect (e) {
  name is _ => ...
  is 0 => ...
  is value => ...
  [x, y] is _ => ...
  is [0, b] => ...
  is [a, b] => ...
  [x, y, z, _] is [_, _, _, a] => // more names
  [_, _, _, x] is [a, b, c, _] => // more refs
  circle as Circle => ...
  [width, height] as Rectangle => ...
};
```

3.5.2 Bruno Cardoso Lopes' EWG 2021-02-26 slide examples

P1371R3	This paper (proposed)

`int factorial(int n) {`
`  return inspect(n) -> int { // explicit type`
`    0 => 1;`
`    __ => n * factorial(n-1);`
`  };`
`}`
	`int factorial(int n) {`
	`  return inspect (n) -> int {  // explicit type`
	`    is 0 => 1;`
	`    is _ => n * factorial(n-1);`
	`  };`
	`}`

`int factorial(int n) {`
`  return inspect(n) {  // deduced equivalent`
`    0 => 1;`
`    __ => n * factorial(n-1);`
`  };`
`}`
	`int factorial(int n) {`
	`  return inspect (n) {  // deduced equivalent`
	`    is 0 => 1;`
	`    is _ => n * factorial(n-1);`
	`  };`
	`}`

`int factorial(int n) {`
`  return inspect(n) {`
`    __ if (n==0) => 1;  // expression equivalent`
`    __ => n * factorial(n-1);`
`  };`
`}`
	`int factorial(int n) {`
	`  return inspect (n) {`
	`    if n==0 => 1;  // expression equivalent`
	`    is _ => n * factorial(n-1);`
	`  };`
	`}`

`int factorial(int n) {`
`  const int base_case = 0;`
`  return inspect(n) -> int {`
`    case base_case => 1; // non-literal equiv.`
`    __ => n * factorial(n-1);`
`  };`
`}`
	`int factorial(int n) {`
	`  const int base_case = 0;`
	`  return inspect (n) -> int {`
	`    is base_case => 1;  // non-literal equiv.`
	`    is _ => n * factorial(n-1);`
	`  };`
	`}`

`int factorial(int n) {`
`  return inspect(n) -> int {`
`    0 => 1;`
`    x => x * factorial(x-1); // identifier`
`  };`
`}`
	`int factorial(int n) {`
	`  return inspect (n) -> int {`
	`     is 0 => 1;`
	`    x is _ => x * factorial(x-1); // identifier`
	`  };`
	`}`

`int factorial(int n) {`
`  auto x = 42;  // shadowed`
`  return inspect(n) -> int {`
`    0 => 1;`
`    x => x * factorial(x-1); // mistake?`
`  };`
`}`
	`int factorial(int n) {`
	`  auto x = 42;  // shadowed`
	`  return inspect (n) -> int {`
	`     is 0 => 1;`
	`    x is _ => x * factorial(x-1); // explicit`
	`  };`
	`}`

`int factorial(int n) {`
`  const auto x = 42;  // add "const"`
`  return inspect(n) -> int {`
`    0 => 1;`
	`int factorial(int n) {`
	`  const auto x = 42;`
	`  return inspect (n) -> int {`
	`    is 0 => 1;`

```
    x => x * factorial(x-1); // different mistake?
  };
}
```

```
    is x => x * factorial(x-1);    // a constraint
    is _ => n * factorial(n-1);    // required
  };
}
```

```
enum Color { Red, Green, Blue };
struct ColorPack {
  Color c1, c2;
};
ColorPack cp{Red, Blue};
inspect (cp) {
  [col, case Red] => cout << col; // id, exp
  // ...
}
```

```
enum Color { Red, Green, Blue };
struct ColorPack {
  Color c1, c2;
};
ColorPack cp{Red, Blue};
inspect (cp) {
  [col,_] is [_,Red] => cout << col; // id, exp
  // ...
}
```

```
enum insn_type { Add, Sub, Unknown };
struct insn_fmt { unsigned opc : 16, imm : 16; };
insn_type decode_insn(insn_fmt &insn) {
  return inspect(insn) {
    [1, i] if (i != 0xffff) => Add;
    [2, __] => Sub
    __ => Unknown;
  };
}
```

```
enum insn_type { Add, Sub, Unknown };
struct insn_fmt { unsigned opc : 16, imm : 16; };
insn_type decode_insn(insn_fmt &insn) {
  return inspect (insn) {
    [_,i] is [1,_] && if i != 0xffff => Add;
    is [2, _] => Sub
    is _ => Unknown;
  };
}
```

```
void f(std::tuple<double, char, std::string> t) {
  inspect(t) {
    [3.8, 'A', "Lisa Simpson"] => ;
    // ...
  };
}
```

```
void f(std::tuple<double, char, std::string> t) {
  inspect (t) {
    is [3.8, 'A', "Lisa Simpson"] => ;
    // ...
  }
}
```

```
int f(int n) {
  int x = 0;
  inspect(n) {
    0 => { x += 2; }
    1 => { x *= 2; }
    2 => ;
    __ => !{}
  };
  return x;
}
```

```
int f(int n) {
  int x = 0;
  inspect (n) {
    is 0 => { x += 2; }
    is 1 => { x *= 2; }
    is 2 => ;
    is _ => { }
  }
  return x;
}
```

```
void h(bool b) {
  auto x = inspect(b) -> std::pair<int, int> {
    true  => {1, 2};
    false => {2, 3};
  };
}
```

```
void h(bool b) {
  auto x = inspect (b) -> std::pair<int, int> {
    is true  => {1, 2};
    is _     => {2, 3};
  }
}
```

```
void r(int a) {
  for (int i = a; i < 10; i++) {
    inspect(i) {
      4 => { return; }
      5 => { break; }
      6 => { continue; }
    };
  }
}
```

```
void r(int a) {
  for (int i = a; i < 10; i++) {
    inspect (i) {
      is 4 => return;
      is 5 => break;
      is 6 => continue;
    }
  }
}
```

3.5.3 [P1371R3] examples

P1371	This paper (proposed)

`inspect (x) {`
`  0  => { std::cout << "got zero"; }`
`  1  => { std::cout << "got one"; }`
`  __ => { std::cout << "don't care"; }`
`};`
	`inspect (x) {`
	`  is 0 => std::cout << "got zero";`
	`  is 1 => std::cout << "got one";`
	`  is _ => std::cout << "don't care";`
	`}`

`inspect (s) {`
`  "foo" => { std::cout << "got foo"; }`
`  "bar" => { std::cout << "got bar"; }`
`  __    => { std::cout << "don't care"; }`
`};`
	`inspect (s) {`
	`  is "foo" => std::cout << "got foo";`
	`  is "bar" => std::cout << "got bar";`
	`  is _     => std::cout << "don't care";`
	`}`

`inspect (p) {`
`  [0, 0] => { std::cout << "on origin"; }`
`  [0, y] => { std::cout << "on y-axis"; }`
`  [x, 0] => { std::cout << "on x-axis"; }`
`  [x, y] => { std::cout << x << ',' << y; }`
`};`
	`inspect (p) {`
	`  is [0, 0]   => std::cout << "on origin";`
	`  is [0, _]   => std::cout << "on y-axis";`
	`  is [_, 0]   => std::cout << "on x-axis";`
	`  [x,y] is _ => std::cout << x << ',' << y;`
	`}`

`inspect (v) {`
`  <int> i => {`
`  strm << "got int: " << i;`
`  }`
`  <float> f => {`
`  strm << "got float: " << f;`
`  }`
`};`
	`inspect (v) {`
	`  i as int   => strm << "got int: "   << i;`
	`  f as float => strm << "got float: " << f;`
	`}`

`// assume Shape has an origin`
`struct Circle : Shape { int radius; };`
`struct Rectangle : Shape { int width, height; };`
`int get_area(const Shape& shape) {`
`  return inspect (shape) {`
`  <Circle>  [r]  => 3.14 * r * r;`
`  <Rectangle> [w, h] => w * h;`
`  };`
`}`
	`// assume Shape has an origin`
	`struct Circle : Shape { int radius; };`
	`struct Rectangle : Shape { int width, height; };`
	`int get_area(const Shape& shape) {`
	`  return inspect (shape) {`
	`  [r]  as Circle  => 3.14 * r * r;`
	`  [w,h] as Rectangle => w * h;`
	`  is _ => 0; // default case is required`
	`  };`
	`}`

`int eval(const Expr& expr) {`
`  return inspect (expr) {`
`  <int> i => i;`
`  <Neg> [(*?) e] => -eval(e);`
`  <Add> [(*?) l, (*?) r] => eval(l) + eval(r);`
 	
`  // Optimize multiplication by 0.`
`  <Mul> [(*?) <int> 0, __] => 0;`
`  <Mul> [__, (*?) <int> 0] => 0;`
	`int eval(const Expr& expr) {`
	`  return inspect (expr) {`
	`  i  as int => i;`
	`  [*e]  as Neg => -eval(e);`
	`  [*l,*r] as Add => eval(l) + eval(r);`
	`  [*l,*r] as Mul {`
	`  // Optimize multiplication by 0.`
	`  if (l as int == 0 || r as int == 0) => 0;`
	`  is _ => eval(l) * eval(r);`

```
    <Mul> [(*?) l, (*?) r] => eval(l) * eval(r);

  };
}
```

```
    }
    is _ => 0; // default case is required
  };
}
```

```
enum class Op { Add, Sub, Mul, Div };

Op parseOp(Parser& parser) {
  return inspect (parser.consumeToken()) {
    '+' => Op::Add;
    '-' => Op::Sub;
    '*' => Op::Mul;
    '/' => Op::Div;
    token => !{
            std::cerr << "Unexpected: " << token;
            std::terminate();
        }
    };
}
```

```
enum class Op { Add, Sub, Mul, Div };

Op parseOp(Parser& parser) {
  return inspect (token = parser.consumeToken()) {
    is '+' => Op::Add;
    is '-' => Op::Sub;
    is '*' => Op::Mul;
    is '/' => Op::Div;
    token is _ => noreturn {
            std::cerr << "Unexpected: " << token;
            std::terminate();
        }
    };
}
```

```
enum Color { Red, Black };

template <typename T>
struct Node {
    void balance();
    Color color;
    shared_ptr<Node> lhs;
    T value;
    shared_ptr<Node> rhs;
};

template <typename T>
void Node<T>::balance() {
  *this = inspect (*this) {
    [case Black, (*?) [case Red, (*?) [case Red, a, x, b], y, c], z, d]
      => Node{Red, make_shared<Node>(Black, a, x, b),
               y, make_shared<Node>(Black, c, z, d)};
    [case Black, (*?) [case Red, a, x, (*?) [case Red, b, y, c]], z, d]
      => Node{Red, make_shared<Node>(Black, a, x, b),
               y, make_shared<Node>(Black, c, z, d)};
    [case Black, a, x, (*?) [case Red, (*?) [case Red, b, y, c], z, d]]
      => Node{Red, make_shared<Node>(Black, a, x, b),
               y, make_shared<Node>(Black, c, z, d)};
    [case Black, a, x, (*?) [case Red, b, y, (*?) [case Red, c, z, d]]]
      => Node{Red, make_shared<Node>(Black, a, x, b),
               y, make_shared<Node>(Black, c, z, d)};
    self => self; // do nothing
  };
}
```

```
enum Color { Red, Black };

template <typename T>
struct Node {
    void balance();
    Color color;
    shared_ptr<Node> lhs;
    T value;
    shared_ptr<Node> rhs;
};

template <typename T>
void Node<T>::balance() {
  *this = inspect (*this) {
       [_,     *[_,   *[_,   a, x, b], y, c], z, d]
    is [Black, *[Red, *[Red, _, _, _], _, _], _, _] ||
       [_,     *[_,   a, x, *[_,   b, y, c]], z, d]
    is [Black, *[Red, _, _, *[Red, _, _, _]], _, _] ||
       [_,     a, x, *[_,   *[_,   b, y, c], z, d]]
    is [Black, _, _, *[Red, *[Red, _, _, _], _, _]] ||
       [_,     a, x, *[_,   b, y, *[_,   c, z, d]]]
    is [Black, _, _, *[Red, _, _, *[Red, _, _, _]]]
      => Node{Red, make_shared<Node>(Black, a, x, b),
               y, make_shared<Node>(Black, c, z, d)};
    is _ => *this;  // do nothing
  }
}
```

3.5.4 [P1308R0] examples

P1308

```
enum color { red, yellow, green, blue };

const Vec3 opengl_color =
  inspect(c) {
    red    => Vec3(1.0, 0.0, 0.0)
    yellow => Vec3(1.0, 1.0, 0.0)
    green  => Vec3(0.0, 1.0, 0.0)
    blue   => Vec3(0.0, 0.0, 1.0)
  };
```

This paper (proposed)

```
enum color { red, yellow, green, blue };

const Vec3 opengl_color =
  inspect (c) {
    is red    => Vec3(1.0, 0.0, 0.0);
    is yellow => Vec3(1.0, 1.0, 0.0);
    is green  => Vec3(0.0, 1.0, 0.0);
    is _      => Vec3(0.0, 0.0, 1.0);
  };
```

P1308

```
struct player { std::string name;
                int hitpoints; int lives; };

void takeDamage(player &p) {
  inspect(p) {
    [hitpoints:   0, lives:0] => gameOver();
    [hitpoints:hp@0, lives:1] => hp=10, l--;
    [hitpoints:hp] if (hp<=3) => { hp--;
                                   almostDead(); }
    [hitpoints:hp] => hp--;
  }
}
```

This paper (proposed)

```
struct player { std::string name;
                int hitpoints; int lives; };

void takeDamage(player &p) {
  inspect (p) {
            is [_,0,0] => gameOver();
    [_,hp,1] is [_,0,1] => hp=10, l--;
    [_,hp,_] is _ {
      if hp<=3 => { hp--; almostDead(); }
      is _     => --hp;
    }
  }
}
```

P1308

```
struct node {
  std::unique_ptr<node> left;
  std::unique_ptr<node> right;
  int value;
};

template <typename Visitor>
void visit_leftmost(const node& n, Visitor&& v)
{
  inspect(n) {
    [left: nullptr] => v(n);
    [left: *left]   => visit_leftmost(left,
                       std::forward<Visitor>(v));
  }
}
```

This paper (proposed)

```
struct node {
  std::unique_ptr<node> left;
  std::unique_ptr<node> right;
  int value;
};

template <typename Visitor>
void visit_leftmost(const node& n, Visitor&& v)
{
  inspect (n) {
    is [nullptr,_,_] => v(n);
    [*left,_,_] is _ => visit_leftmost(left,
                        std::forward<Visitor>(v));
  }
}
```

P1308

```
// alternate: visit leftmost only traversing
// to a left child node if its value >= 5

template <typename Visitor>
void visit_leftmost2(const node& n, Visitor&& v)
{
  inspect(n) {
    [left] if (left && left->value >= 5)
       => visit_leftmost(left,
            std::forward<Visitor>(v));
    __ => v(n);
  }
```

This paper (proposed)

```
// alternate: visit leftmost only traversing
// to a left child node if its value >= 5

template <typename Visitor>
void visit_leftmost2(const node& n, Visitor&& v)
{
  inspect (n) {
    [*left,_,_] if left.value >= 5
         => visit_leftmost(left,
              std::forward<Visitor>(v));
    is _ => v(n);
  }
```

}
 	
class Animal { ... };
class Cat : public Animal { ... };
class Crow : public Animal { ... };
 	
void listen(const Animal &a) {
  inspect(a) {
    Cat c => c.speak();
    Crow c => std::cout << "All crows say "
                << c.speak() << std::endl;
  }
}
	}
	
	class Animal { ... };
	class Cat : public Animal { ... };
	class Crow : public Animal { ... };
	
	void listen(const Animal &a) {
	  inspect (a) {
	    c as Cat => c.speak();
	    c as Crow => std::cout << "All crows say "
	                  << c.speak() << std::endl;
	  }
	}

3.5.5 [P1260R0] examples

P1260	This paper (proposed)

`inspect (x) {`
`  0: std::cout << "got zero";`
`  1: std::cout << "got one";`
`  _: std::cout << "don't care";`
`}`
	`inspect (x) {`
	`  is 0 => std::cout << "got zero";`
	`  is 1 => std::cout << "got one";`
	`  is _ => std::cout << "don't care";`
	`}`

`inspect (s) {`
`  "foo": std::cout << "got foo";`
`  "bar": std::cout << "got bar";`
`  _:     std::cout << "don't care";`
`}`
	`inspect (s) {`
	`  is "foo" => std::cout << "got foo";`
	`  is "bar" => std::cout << "got bar";`
	`  is _     => std::cout << "don't care";`
	`}`

`inspect (t) {`
`  [0, 0]: std::cout << "on origin";`
`  [0, y]: std::cout << "on y-axis";`
`  [x, 0]: std::cout << "on x-axis";`
`  [x, y]: std::cout << x << ',' << y;`
`}`
	`inspect (t) {`
	`  is [0, 0]  => std::cout << "on origin";`
	`  is [0, _]  => std::cout << "on y-axis";`
	`  is [_, 0]  => std::cout << "on x-axis";`
	`  [x,y] is _ => std::cout << x << ',' << y;`
	`}`

`inspect (v) {`
`  <int> i:  strm << "got int: " << i;`
`  <float> f: strm << "got float: " << f;`
`}`
	`inspect (t) {`
	`  i as int   => strm << "got int: "   << i;`
	`  f as float => strm << "got float: " << f;`
	`}`

`// assume Shape has an origin`
`struct Circle : Shape { int radius; };`
`struct Rectangle : Shape { int width, height; };`
`int get_area(const Shape& shape) {`
`  inspect (shape) {`
`  (as<Circle>    ? [r])   : return 3.14 * r * r;`
`  (as<Rectangle> ? [w, h]): return w * h;`
`  }`
`}`
	`// assume Shape has an origin`
	`struct Circle : Shape { int radius; };`
	`struct Rectangle : Shape { int width, height; };`
	`int get_area(const Shape& shape) {`
	`  inspect (shape) {`
	`  [r]   as Circle    => 3.14 * r * r;`
	`  [w,h] as Rectangle => w * h;`
	`  }`
	`}`

`int eval(const Expr& expr) {`
`  inspect (expr) {`
`  <int> i:      return i;`
`  <Neg> [e]:    return -eval(*e);`
`  <Add> [l, r]: return eval(*l) + eval(*r);`
`  <Mul> [l, r]: return eval(*l) * eval(*r);`
`  }`
`}`
	`int eval(const Expr& expr) {`
	`  inspect (expr) {`
	`  i     as int => return i;`
	`  [e]   as Neg => return -eval(*e);`
	`  [l,r] as Add => return eval(*l) + eval(*r);`
	`  [l,r] as Mul => return eval(*l) * eval(*r);`
	`  }`
	`}`

3.5.6 [Solodkyy 2014b] Urbana 2014 examples

Urbana 2014	This paper (proposed)

`double area(const Shape& s) {`
`  inspect (s) {`
`  when Circle: return 2*pi*radius(); // not s.radius()`
`  when Square: return height()*width();`
`  default:  error("unknown shape");`
`  }`
`}`
	`double area(const Shape& s) {`
	`  inspect (s) {`
	`  c as Circle => return 2*pi*c.radius();`
	`  s as Square => return s.height()*s.width();`
	`  is _  => error("unknown shape");`
	`  }`
	`}`

`class Expr  { virtual ~Expr(); };`
`class Value : Expr { int value; };`
`class Plus  : Expr { Expr& a; Expr& b; };`
`class Minus : Expr { Expr& a; Expr& b; };`
`class Times : Expr { Expr& x; Expr& y; };`
`class Divide: Expr { Expr& divident; Expr& divisor; };`
`int eval(const Expr* e) {`
`  inspect (e) {`
`  when Value:  return value;`
`  when Plus:  return eval(a)+eval(b);`
`  when Minus:  return eval(a)-eval(b);`
`  when Times:  return eval(x)*eval(y);`
`  when Divide: return eval(dividend)/eval(divisor);`
`  }`
`}`
	`class Expr  { virtual ~Expr(); };`
	`class Value : Expr { int value; };`
	`class Plus  : Expr { Expr& a; Expr& b; };`
	`class Minus : Expr { Expr& a; Expr& b; };`
	`class Times : Expr { Expr& x; Expr& y; };`
	`class Divide: Expr { Expr& dividend; Expr& divisor; };`
	`int eval(const Expr* e) {`
	`  return inspect (e) {`
	`  e as Value  => e.value;`
	`  e as Plus  => eval(e.a)+eval(e.b);`
	`  e as Minus  => eval(e.a)-eval(e.b);`
	`  e as Times  => eval(e.x)*eval(e.y);`
	`  e as Divide => eval(e.dividend)/eval(e.divisor);`
	`  is _  => an_error; // match-anything required`
	`  };`
	`}`

`istream& operator<<(istream& os,`
`  const variant<int,double>& u)`
`{`
`  inspect (u) {`
`  when {int a}:  return os << a;`
`  when {double d}: return os << d;`
`  }`
`}`
	`istream& operator<<(istream& os,`
	`  const variant<int,double>& u)`
	`{`
	`  return inspect (u) {`
	`  a as int  => os << a;`
	`  d as double => os << d;`
	`  };`
	`}`

`void advance(Iterator p, int n)`
`{`
`  inspect(Iterator) {`
`  when Forward_iterator: // ?fallthrough if empty?`
`  when Bidirectional_iterator: while(--n>0) ++p;`
`  when Randomaccess_iterator:  p+=n;`
`  }`
`}`
	`void advance(Iterator p, int n)`
	`{`
	`  inspect(Iterator) {`
	`  is Forward_iterator ||`
	`  is Bidirectional_iterator => while(--n>0) ++p;`
	`  is Randomaccess_iterator  => p+=n;`
	`  }`
	`}`

`template<typename T, typename U>`
`void f(T& x, U xx)`
`{`
`  inspect (x,xx) {`
`  when {int* p,0}: p=nullptr;`
	`template<typename T, typename U>`
	`void f(T& x, U xx)`
	`{`
	`  inspect {x,xx} {`
	`  [p,_] is [int*,0] => p=nullptr;`

```
  when {_a,int}:   ...  // _a is an introduced name
  }
}
```

```
  [a,_] is [_,int] => ...
  }
}
```

```
double factorial(int n)
{
  assert(0<=n);
  inspect(n)  {
  when 0:          return 1;
  when {double m}: return m*factorial(m-1);
     // m initialized by n
  }
}
```

```
double factorial(int n)
{
  assert(0<=n);
  inspect(n)  {
    is 0      => return 1;
  m as double => return m*factorial(m-1);
  }
}
```

```
template<typename ...Ts>
void print(tuple<Ts...>& t)
{
  inspect (t) {
  when {}:         ;
  when {auto a}:   cout<<a;
  when {_a,_tail}: cout<<a; print(tail);
  }
}
```

```
template<typename ...Ts>
void print(tuple<Ts...>& t)
{
  inspect (t) {
              is []  => ;
  [a]         is [_] => cout<<a;
  [a,...tail] is _   => cout<<a; print(tail...);
  }
}
// Note: this would be an extension...
//       do want to allow variadic names?
```

```
template<typename ...Ts, typename ...Us>
bool operator==(tuple<Ts...>& t, tuple<Us...>& u)
{
  inspect (t,u) {
  when {{},{}}: return true;
  when {_,{}}:  return false;
  when {{},_}:  return false;
  default:      if (head(t)!=head(u)) return false;
                return tail(t)==tail(u);
  }
}
```

```
// (1) a direct respelling in this paper's syntax
template<typename ...Ts, typename ...Us>
bool operator==(tuple<Ts...>& t, tuple<Us...>& u)
{
  inspect (t,u) {
  is [[],[]] => return true;
  is [_ ,[]] => return false;
  is [[],_ ] => return false;
  is _       => if (head(t)!=head(u)) return false;
                return tail(t)==tail(u);
  }
}
```

```
(same as previous)
```

```
// (2) my preferred way, in this paper's syntax...
//     regular + makes the recursive case visible
template<typename ...Ts, typename ...Us>
bool operator==(tuple<Ts...>& t, tuple<Us...>& u)
{
  return inspect (t,u) {
  is [[],[]]          => true;
  is [_ ,[]]          => false;
  is [[],_ ]          => false;
  if head(t)!=head(u) => false;
  is _                => tail(t)==tail(u);
  };
}
```

(same as previous)	// (3) or this, fully regular
	template<typename ...Ts, typename ...Us>
	bool operator==(tuple<Ts...>& t, tuple<Us...>& u)
	{
	  return inspect (t,u) {
	  is [[],[]] => true;
	  is [_ ,[]] => false;
	  is [[],_] => false;
	  if head(t)!=head(u) => false;
	  if tail(t)==tail(u) => true;
	  is _ => false;
	  };
	}

void print(Range<T> r) // use PM?
{
  inspect(r) {
  when {}: ;
  when {_p,_q}: cout << *_p; print(++_p,_q);
  }
}
	void print(Range<T> r) // has suitable operator is/as
	{
	  inspect(r) {
	  is ranges::empty => ;
	  [p,q] is _ => { cout << *_p; print(++p,q); }
	  }
	}

void print(Range<T> r) // use PM?
{
  inspect(begin(r),end(r)) {
  when {_b,_e} | _b==_e: return; // conditional match
  when {Iterator _b, Iterator _e}:
  cout << *_b; print(++_e);
  }
}
	void print(Range<T> r) // use PM?
	{
	  inspect(begin(r),end(r)) {
	  [b,e] b==e => return; // conditional match
	  [b,e] is [Iterator,Iterator]
	  => { cout << *_b; print(++_e); }
	  }
	}

3.5.7 [Rust PM] examples

Rust	This paper (proposed)

let x = 1;
match x {
  1 => println!("one"),
  2 => println!("two"),
  3 => println!("three"),
  _ => println!("anything"),
}
	auto x = 1;
	cout << inspect (x) {
	  is 1 => "one";
	  is 2 => "two";
	  is 3 => "three";
	  is _ => "anything";
	};

let x = 'x';
let c = 'c';
match c {
  x => println!("x: {} c: {}", x, c),
} // implicit shadowing pitfall
	auto x = 'x';
	auto c = 'c';
	inspect (c) {
	  x is _ => cout<<fmt("x: {} c: {}", x, c),
	} // explicit: new names always before 'is'

let x = 1;
match x {
  1 | 2 => println!("one or two"),
  3 => println!("three"),
  _ => println!("anything"),
}
	auto x = 1;
	cout << inspect (x) {
	  is 1 || is 2 => "one or two";
	  is 3 => "three";
	  is _ => "anything";
	};

let x = 1;
match x {
  e @ 1 ... 5 => println!("in the range"),
  _ => println!("out of the range"),
}
	auto x = 1;
	inspect (x) {
	  is in(1,5) => cout << "in the range";
	  is _ => cout << "out of the range";
	}

let x = 1;
match x {
  e @ 1 ... 5 |
  e @ 8 ... 10 => println!("got value {}", e),
  _ => println!("anything"),}
	auto x = 1;
	inspect (x) {
	  e is in(1,5) ||
	  is in(8,10) => cout<<fmt("got value {}", e);
	  is _ => cout << "anything";
	}

struct Point { x: i32, y: i32 }
let origin = Point { x: 0, y: 0 };
match origin {
  Point { x, y } => println!("({},{})", x, y),
}
match origin {
  Point { x: x1, y: y1 } => println!("({},{})", x1, y1),
}
match origin {
  Point { .., y: yy } => println!("y is {}", yy),
}
	struct Point { int x; int y; };
	Point origin(0, 0);
	inspect (origin) {
	  [x,y] is _ => { cout<<fmt("({},{})", x, y); }
	}
	inspect (origin) {
	  [x1,y1] is _ => { cout<<fmt("({},{})", x1, y1); }
	}
	inspect (origin) {
	  [_,yy] is _ => { cout<<fmt("y is {}", yy); }
	}

3.5.8 [Swift PM] examples

Swift	This paper (proposed)

let point = (3,2);
switch point {
  case let (x,y):
  print("Point is at (\(x), \(y))")
}
	pair point(3,2);
	inspect (point) {
	  [x,y] is _ =>
	  cout<<fmt("Point is at ({}, {})", x, y);
	}

let point = (1,2);
switch point {
case (0, 0):
  print("(0, 0) is at the origin.")
case (-2...2, -2...2):
  print("(\(point.0), \(point.1)) is near the origin.")
default:
  print("Point is at (\(point.0), \(point.1)).")
}
	pair point (1,2);
	inspect (point) {
	  [_,_] is [0, 0] =>
	  cout << "(0, 0) is at the origin.";
	  [x,y] is [in(-2,2), in(-2,2)] =>
	  cout << fmt("({}, {}) is near the origin", x, y);
	  [x,y] is _ =>
	  cout << fmt("Point is at ({}, {})", x, y);
	}

switch x {
case _ where x > 0:
  print("positive")
case _ where x < 0:
  print("negative")
default:
  print("zero")
}
	inspect (x) {
	if x > 0 =>
	  cout << "positive";
	if x < 0 =>
	  cout << "negative";
	is _ =>
	  cout << "zero";
	}

3.5.9 Bjarne's syntax examples

Examples from Bjarne's Apr 2020 draft paper that are not already covered above.

Examples like the red-black tree balancing and small string optimization are classic use cases for pattern matching. See also [N3449], [Solodkyy 2012], and [Solodkyy 2014b].

Bjarne's paper

```
enum Color { Red, Black };

template <typename T>
struct Node {
    void balance();
    Color color;
    shared_ptr<Node> lhs;
    T value;
    shared_ptr<Node> rhs;
};

template <typename T>
void Node<T>::balance() {
  *this = inspect (*this) {
    // left-left case
    [case Black, *[case Red, *[case Red, a, x, b], y, c], z, d]:
      Node{Red, make_shared<Node>(Black, a, x, b),
            y, make_shared<Node>(Black, c, z, d)};
    // left-right case
    [case Black, *[case Red, a, x, *[case Red, b, y, c]], z, d]:
      Node{Red, make_shared<Node>(Black, a, x, b),
            y, make_shared<Node>(Black, c, z, d)};
    // right-left case
    [case Black, a, x, *[case Red, *[case Red, b, y, c], z, d]]:
      Node{Red, make_shared<Node>(Black, a, x, b),
            y, make_shared<Node>(Black, c, z, d)};
    // right-right case
    [case Black, a, x, *[case Red, b, y, *[case Red, c, z, d]]]:
      Node{Red, make_shared<Node>(Black, a, x, b),
            y, make_shared<Node>(Black, c, z, d)};
    // do nothing
    __ :  *this
  };
}
```

This paper (proposed)

```
enum Color { Red, Black };

template <typename T>
struct Node {
    void balance();
    Color color;
    shared_ptr<Node> lhs;
    T value;
    shared_ptr<Node> rhs;
};

template <typename T>
void Node<T>::balance() {
  *this = inspect (*this) {
    // left-left case
       [_,     *[_,   *[_,   a, x, b], y, c], z, d]
    is [Black, *[Red, *[Red, _, _, _], _, _], _, _] ||
    // left-right case
       [_,     *[_,   a, x, *[_,   b, y, c]], z, d]
    is [Black, *[Red, _, _, *[Red, _, _, _]], _, _] ||
    // right-left case
       [_,     a, x, *[_,   *[_,   b, y, c], z, d]]
    is [Black, _, _, *[Red, *[Red, _, _, _], _, _]] ||
    // right-right case
       [_,     a, x, *[_,   b, y, *[_,   c, z, d]]]
    is [Black, _, _, *[Red, _, _, *[Red, _, _, _]]]
      => Node{Red, make_shared<Node>(Black, a, x, b),
                y, make_shared<Node>(Black, c, z, d)};
    is _ => *this;  // do nothing
  }
}
```

```
// short string optimization
char* String::data() {
    inspect (*this) {
        Local [i]: return i;
        Remote [r]: return r.ptr;
    }
}
```

```
// short string optimization
char* String::data() {
    inspect (*this) {
        [i] is Local => return i;
        [r] is Remote => return r.ptr;
    }
}
```

```
int fac(int n) {
    return inspect(n) {
        0:       1;
        m ? m>1: m*fac(m-1);
        __:      throw Negative_fac{};
    };
}
```

```
int fac(int n) {
    return inspect (n) {
        is 0   => 1;
        is n>1 => n*fac(n-1);
        is _   => throw Negative_fac{};
    };
}
```

```
int fac(int n) {   // not proposed, would be special case
    return inspect(n) {
        0:     1;
        ? n>1: n*fac(n-1);
        __:    throw Negative_fac{};
    };
}
```

```
int fac(int n) {   // naturally works, no special case
    return inspect (n) {
          is 0   => 1;
        m is m>1 => m*fac(m-1);
          is _   => throw Negative_fac{};
    };
}
```

```
// concepts
template<Iterator Iter>
Iter advance(Iter p, int n) {
    return inspect(Iter) {
        input_iterator: while (n--) ++p;
        random_access_iterator: p+n;
        auto: static_assert("bad type for advance()");
    };
}
```

```
// concepts
template<Iterator Iter>
Iter advance(Iter p, int n) {
    return inspect (Iter) {
        is input_iterator => while (n--) ++p;
        is random_access_iterator => p+n;
        is _ => static_assert("bad type for advance()");
    };
}
```

```
// types
inspect(decltype(x)) {
    Foo P:       …       // P names a type
    Bar Q:       …       // Q names a type
    auto:        static_assert("no type I know of");
};
```

```
// types
inspect (decltype(x)) {
    P is Foo =>  …       // P names a type
    Q is Bar =>  …       // Q names a type
      is _   =>  static_assert("no type I know of");
}
```

```
inspect (x) {
  [a,b] is [_,odd] && is [<100,b] && check(a,b)
    => do_something(a,b);
}
```

```
inspect (x) {
  [a,b] b is odd && a<100 && check(a,b)
    => do_something(a,b);
}
```

4 Appendix: Notes on implementation, optimization, and syntax

4.1 Implementation notes: Grammar, avoiding ambiguities

Thanks to Richard Smith for his feedback that led to much of the content of this section, including suggesting these grammar productions.

4.1.1 `is` and `as` are context-sensitive keywords

The words `is` and `as` are not globally reserved, so we have to make sure that uses of those names as identifiers (e.g., names of types, variables, namespaces) all work including being aware of any backward compatibility issues.

4.1.2 Grammar productions for `is` and `as` expressions

There are five grammar productions for `is`-expressions:

1) type-id is type-id

2) type-id is type-constraint

3) expression is expression

4) expression is type-id

5) expression is type-constraint

#3 covers both “expression is value” and “expression is value-constraint.”

“expression has operator is” can be either #3 or #4.

Note If the grammar doesn’t easily allow cases like `unsigned int is Modulo` for the C fundamental types whose names are multiple tokens, we shouldn’t let them complicate this design. The user can use a typedef/using alias as the usual workaround for today’s problem cases such as a function-style cast like `unsigned int(5)`. — In the future we could perhaps generally solve the problem with those names by some other general approach instead of fixing individual cases in the language For example, we could make those names parse as single whitespace tokens (i.e., “unsigned /*arbitrary whitespace*/ int” would become a single token and a new globally reserved keyword, as we did for “ref class” etc. in C++/CLI), and then the main questions are relatively small details like whether macros should be able to create such a token.

In cases like `int(int())` that grammatically can be valid value expressions and valid function types, we interpret them as value expressions. To use a type, give it a name.

There is one grammar production for `as`-expressions:

6) expression as type-id

The grammar productions would be:

pm-expression:

~~<del>cast-expression</del>~~ <ins>as-is-expression</ins>

(and similar for the other pm-expression productions)

as-is-expression:
 cast-expression
 as-is-expression is is-constraint
 as-is-expression as type-id
 type-id is type-id
 type-id is type-constraint

is-constraint:
 cast-expression
 type-id
 type-constraint
 [is-constraint-list[opt]]

is-constraint-list:
 is-constraint ...[opt]
 is-constraint-list , is-constraint ...[opt]

For example:

- `z += 3 * x as y` would parse as `z += (3 * (x as y) )`
- `++a is b++` would parse as `(++a) is (b++)`

Note We'll want to allow pack expansion in a `[...]` list. At least `tuple is [PackOfTypes...]` makes sense here, but we should also consider whether to allow constructs like `tuple is [int...]` to mean "these are all ints and I don't care how many there are," as Barry Revzin proposed for `auto [x...] = tuple;` .

This grammar allows `is` followed by a lambda, since lambda-expressions are primary-expressions and so can appear basically anywhere. (See below for further discussion of lambdas.)

4.1.3 `operator is` and `operator as`

When written as member functions, the new `operator is` and `operator as` look similar to existing conversion functions to types named `is` or `as`, including that they can have a *template-head*:

```
template<typename T = void> struct is { };
template<typename T = void> struct as { };

struct X {
    operator is() const;            // conversion operator to type 'is'

    template<typename T>
    operator is<T>() const;         // conversion operator to type 'is<T>'

    operator as() const;            // conversion operator to type 'as'

    template<typename T>
    operator as<T>() const;         // conversion operator to type 'as<T>'
```

The new operators `is` and `as` are distinct because they require a return type, which is unambiguous:

```
    bool operator is(X) const;       // new member 'operator is', value of same type

    template<typename Y>
    bool operator is(Y) const;       // new member 'operator is', value of different type

    template<typename ValPred>
    bool operator is(ValPred) const; // new member 'operator is', value predicate

    template<typename Type>
    bool operator is() const;        // new member 'operator is', type

    template<typename T>
    T    operator as() const;        // new member 'operator as', type
};
```

Alternatively, these can be written using trailing return type syntax, in which case `auto` still comes first which it cannot with a conversion operator.

Parsing these new operators requires no lookahead, because today `is` and `as` are not allowed in that position. In C++20, when parsing a member function declarator that starts with a return type (or `auto`) followed by `operator`, we have we have already excluded conversion operators grammatically and the next token must be one of the overloadable operator symbols. So if the next token is `is` or `as`, it is unambiguously the new operator, no lookahead is required.

However, note this case:

```
struct Y : X {
  using X::operator is;              // what does this mean?
};
```

The way we interpret this name depends on what we find in `X`, including if `X` is a dependent base class. If `X` has both a conversion operator to type `is` and an overloaded `bool operator is`, both are brought into scope.

If `X` has both a conversion operator to type `is` and an overloaded `bool operator is`, then we need a rule to make invoking `operator is` by name using member function call syntax unambiguous:

```
X().operator is()                    // today, invokes a conversion operator
```

For this case, we can either prefer the type query operator or the conversion operator. I don't currently have a preference.

> **Note** If we were very concerned about visual ambiguity and/or about types that provide both a conversion `operator is` and the new type query `operator is` (and similarly for `as`) we could just reserve `operator is` and `operator as` even as conversion operators and break any code that uses them. Such conversion operators can be rewritten to `operator std::identity_t<is>` and `operator std::identity_t<as>`, if any such code actually exists. Richard Smith reports that Google's large internal code base has zero occurrences of "operator is" or "operator as," and codesearch.isocpp.org's ~2.5M source files there are also zero occurrences.
>
> I don't think we should do that. It would not remove any potential teachability issue even there is one, because the syntax for writing `operator is` and `operator as` would still be the same so we would still have to teach it. And it would create something new to teach, namely the asymmetry that conversion operators are allowed except not for types with two particular names.

4.1.4 is and as munching

Consider:

```
new unsigned is nullptr             // do we max-munch the 'is' as a type here?
```

We do not max-munch `is`.

It should be rare to find a new-expression on the left-hand side of an `is`, so it doesn't seem incredibly important to be able to give a good diagnostic. But if we do discover an issue with this, we could require `( )` treat it the same as with:

```
new widget -> f();                  // error, parens required
(new widget) -> f();                // ok
```

so that we could require

```
new unsigned is nullptr             // if this is problematic for some reason
(new unsigned) is nullptr           // this is always unambiguous
```

Consider also:

```
unsigned int is Modulo
```

This could have the same kinds of parsing problems (or lack thereof) as

```
unsigned int n = unsigned int(5);
```

as a function-style cast. I don't think the very-few C fundamental multi-token type names alone should influence this design. If those are the only things not covered by the grammar we pick, that's fine and today programmers can use an alias to rename them to a single-token name as the usual workaround.

> **Note** If we want to deal with multi-token type names, we can do it in a more general way in the future rather than as a narrow tweak for just one particular feature, such as just for `is`/`as` or just for function-style casts. For example, we could broadly name the multi-token type names parse as single whitespace tokens (i.e., "`unsigned` /*arbitrary whitespace*/ `int`" would become a single token, as we did for `ref class` and similar in C++/CLI, and then the main issues are questions like whether macros should be able to create the token).

4.1.5 Function and variable declarations named is or as

Consider:

```
int is(int());                      // a function declaration (vexing parse)
bool is(true);                      // a variable declaration (non-vexing)
```

These would not be valid `is`-expressions because if we have a type-id on the left then we must have a type or type predicate on the right hand side; "*type-id* is *expression*" is not one of the productions allowed in §4.1.2.

Parsing this can require (minor) additional work. In C++20, if we see a type-specifier at the start of a statement that is not followed by `(` or `{`, we know it's a declaration and can parse it as such. With this new rule, we would additional need to check if the next token is `is`, and if so, look ahead one more token to see if it's followed by something that could plausibly be a type-id or type-constraint (i.e., not `(` or `{` or `=` or `;`) or something that could plausibly follow the declared name in a variable declaration (i.e., `(` or `{` or `=` or `;`).

4.1.6 cv-qualifiers

Consider this example in the case where `is` is a type:

```
const is x;
```

This is declaring a variable `x` of type `const is`. It is not comparing the (grammatically-valid but meaningless) type-id `const` against the type or constraint `x`, but we will need a disambiguation rule to say so. Richard Smith suggests we could generalize [dcl.spec.general]/3 to cover it:

> If a type-name is encountered while parsing a decl-specifier-seq<ins>, defining-type-specifier-seq, or type-specifier-seq S</ins>, it is interpreted as part of ~~<del>the decl-specifier-seq</del>~~ <ins>S</ins> if and only if there is no previous defining-type-specifier <ins>or type-specifier</ins> other than a cv-qualifier in ~~<del>the decl-specifier-seq</del>~~ <ins>S</ins>."

4.1.7 Decompositions and attributes

Consider:

```
if( data is [_, [1, _]] ) { ... }   // e.g., struct { int; struct { int; int; } }
if( data is [[1, _], _] ) { ... }   // e.g., struct { struct { int; int; } int; }
if( data is [[1, _]]    ) { ... }   // e.g., struct { struct { int; int; } }
```

These cases would be ambiguous if attributes were permitted following `is`, so we do not allow attributes in the middle of an is-expression.

We want to allow decomposition symmetrically also in structured bindings, so:

```
auto [_, [1, _]] = ... ;
auto [[1, _], _] = ... ;
auto [[1, _]]    = ... ;
```

Option 1: We could disallow attributes in the position of structured bindings or after `is`, and likely very little structured bindings code would break.

Option 2: Alternatively, we could adjust our current blanket rule that `[[` can only ever be used to introduce an attribute, to support decompositions that start with `[[` or end with `]]`. This might have some non-trivial consequences for some implementations. (We aren't aware of any implementation that is still blindly deleting all tokens between paired `[[` and `]]` any more, at least, but should ask vendors for feedback.)

Option 3: Alternatively, we could require an absence of whitespace between the pair of `[`s in an attribute (and symmetrically for the `]`s), so that then the ambiguity can be solved with whitespace (e.g., `auto [[attr]]` versus `auto [ [nested] ]`).

4.1.8 `requires` and `is` (basic)

Consider:

```
template< typename T, auto Size >
    requires Size is Number              // type predicate constraint
```

This should be equivalent in meaning to

```
    requires Number<Size>
```

because the meaning should not change because of writing it using `is`. Note that this mean this particular expression should participate in constraint-based partial ordering, rather than being treated as an atomic constraint, so `is`-expressions will be treated specially during constraint normalization.

4.2 Optimization notes: Static matches, integers, strings, and more

[Mach7] and [Solodkyy 2013] are a primary resource for high performance pattern matching in C++. This section is not exhaustive, but seeks to illustrate a few important implementation strategies to show this model does not interfere with getting the optimizations we expect from modern pattern matching using this example:

```
constexpr int f(auto x) {
    return inspect (x) {
        is string     => g1(x);                          // group 1

        is 2          => g2(x);                          // group 2
        is 3          => g3(x);
        is 4          => g4(x);

        is "daffy"    => g5(x);                          // group 3
        is "da vinci" => g6(x);
        is "daffodil" => g7(x);

        is _          => g8(x);                          // group 4
    }
}
```

This is equivalent to four groups of alternatives, and a quality implementation should group them, such as:

```
constexpr int f(auto x) {
    if constexpr( x is string ) return g1();             // group 1

    else if constexpr( requires{ x==2; } ) {             // group 2
        // see §4.2.2
    }

    else if constexpr( requires{ x=="string-literal"; } ) {    // group 3
        // see §4.2.3
    }

    else return g8(x);                                   // group 4
}
```

4.2.1 Static matches: Types and compile-time values

All static tests can cause alternatives to be selected or elided at compile time as usual. For example, when instantiating `f<string>` we know the selected alternative statically, and can emit this body:

```
return g1(x);
```

For example, when instantiating `f<int>` we know can prune irrelevant alternatives and emit this body:

```
return inspect (x) {
    is 2          => g2(x);                              // group 2
    is 3          => g3(x);
    is 4          => g4(x);

    is _          => g8(x);                              // group 4
};
```

4.2.2 Integer series (C switch)

What we teach programmers: "For optimal efficiency, arrange integer alternatives to be consecutive where possible."

All consecutive integer alternatives can be optimized the same way as a C `switch` today.

For example, this sequence of consecutive alternatives from this section's example:

```
is 2          => g2(x);
is 3          => g3(x);
is 4          => g4(x);
```

could be implemented by rewriting it to a C `switch`:

```
if constexpr( requires{ x==2; } ) {
    switch (x) {                            // C switch, including all usual
    case 2: return g2(x); // break;         // optimizations such as jump tables
    case 3: return g3(x); // break;
    case 4: return g4(x); // break;
    }
}
```

or emitting a table lookup directly:

```
if constexpr( requires{ x==2; } ) {
    constexpr __pfna[] = { &g2, &g3, &g4 };
    if( x is in(2,4) )
        __pfna[x-2](x);
}
```

4.2.3 String series (string matching FSM)

All string alternatives can be optimized using traditional string match approaches, including using the length to avoid character comparisons, generating a FSM to guarantee single-pass comparison, and/or doing a hash table lookup for the result.

The alternatives need not be consecutive to benefit.

For example, this sequence of consecutive alternatives from this section's example:

```
is "daffy"    => g5(x);
is "da vinci" => g6(x);
is "daffodil" => g7(x);
```

could be emitted as if this short-circuited single-pass logic:

```
if constexpr( requires{ x=="string-literal"; } ) {
    if( (x.length() == 5 || x.length() == 8)  // quick short-circuit based on length
            && x[0] == 'd' && x[1] == 'a' )   // single pass over the string contents
        if( x[2] == 'f' && x[3] == 'f' ) {    // (more generally, generate regex FSM)
            if( /* ... rest is "y" ... */ )
```

```
                return g7();
            else if( /* ... rest is "odil" ... */ )
                return g6();
        } else if( /* ... rest is " vinci" ...*/ )
            return g8();
    }
```

4.2.4 O(1) indexed/type_info query (e.g., variant, any)

See §2.3.3 and §2.3.4.

4.2.5 Additional resources

See [Mach7] and [Solodkyy 2013] as a primary resource for high performance pattern matching implementation.

Thanks to Bruno Cardoso Lopes for the following additional notes and references regarding pattern matching optimizations:

[Augustsson 2005] is a good introduction.

[Maranget 2008] provides interesting ideas useful for machine learning, and other useful work from the author is available in the References section of the paper.

We expect to get many optimizations for free or with little effort by leveraging what is already implemented in existing optimizers for switches and branch chains. LLVM is very smart about optimizing CFGs coming out of switches/branch-chains, much of which applies to pattern matching. [Sayle 2008] describes GCC jump threading, transformations on jump-tables/lookup-tables/if-trees/if-chains and other optimizations. Other more ad-hoc topics include also using PGO, or treating vectors specially in face of structural binding patterns which can allow for some neat SIMD codegen using predicates/masks/shuffle.

4.3 Syntax notes: `inspect`, `=>`, `( )`

The following alternate syntax choices don't affect the basic design or semantics of the approach in this paper. This approach can switch to use those at any point if we want.

4.3.1 Introducing the pattern match: `switch` vs. `match` vs. `inspect` (etc.)

Priorities: The key is to pick a syntax that

- is short and clear — short because a pattern match expression wants a concise syntax, but not just a symbol because a pattern match statement wants a syntax that is readable and should use a nice word;
- directly connotes the right thing;
- is unambiguous and parseable without excessive lookahead; and
- doesn't make it hard to teach programmers to use pattern matching instead of C-style switching in modern code.

Options: Major options discussed in this section are

- `switch`
- `match` contextual word
- `inspect` contextual word
- uglified new reserved word
- multiple new contextual words
- existing reserved word (besides just `switch`)
- existing reserved word/symbol plus another word

`switch`. For example

```
switch (x) { /* need more lookahead to disambiguate less-usual flavors of switch */
```

I would prefer `switch` (as in C, C++, and Swift), if the lookahead for disambiguation is not prohibitive, and if it were acceptable to WG21, but at this point that seems unlikely because of parsing and teaching concerns. The word `switch`:

- is short and clear (and already reserved);
- directly connotes the right thing since this is a strict generalization of C and C++ `switch`, that covers exactly a superset of the uses, and removes the one known pitfall (implicit `case` fallthrough) — so this is still a `switch`, just enabling patterns in the body rather than specific integer equality cases only;
- is unambiguous with existing uses of `switch`, but because legal `switch` bodies can be unusual[4] the disambiguation can require lookahead potentially as far as the closing `}`, possibly with a tentative parse; and
- makes it easy to tell programmers "avoid `case` in new code, `is` is better" because `is` is more flexible than `case` and more general because it could be used everywhere in the language.

`match`. For example

```
match (x) { /* need more lookahead to disambiguate from a declaration */
```

The word `match`:

[4] For example, `switch(x){}`, `switch(x){label:...}`, `switch(x){unreachable_code()}`, `switch(x){int i; case:..}`.

- is short and clear, and the most similar to `switch`;
- directly says what we are going to do — it's a word we always use anyway when describing the new feature's operation even if we call it something else (e.g., "an *inspect* statement *matches* the...");
- is unambiguous with lookahead; and
- makes it easy to tell programmers "avoid `switch` and `case` in new code, `match` is better because it allows `is`," since `is` is more flexible than `case` and more general because it could be used everywhere in the language.

`inspect` or other single words. For example

```
inspect (x) { /* need more lookahead to disambiguate from a declaration */
```

and possibly other similar single English verbs. These meet most of the criteria, but they no longer directly say what we are going to do. If we use a new word like `inspect` or `match`, we should make it contextual in a currently-unused place in the grammar, as we did with `override`.

Disambiguation is needed in any of the above cases (extending the reserved word `switch` or adding a new contextual word), and it's okay to require some token lookahead to disambiguate from today's code. It's not okay to require not tentative parses with backtracking, or parsing that relies on semantic analysis (even though we already have some of that, we mustn't add more).

An uglified new reserved word. If we feel we must take a new reserved word, it must be uglified to be uncommon enough to avoid significant code breakage. For example, `inspect` is less widely used than `match`, but even `inspect` is probably too difficult to reserve because it used in enough major projects as ABI-sensitive identifiers (such as virtual functions and namespaces which are difficult or impossible to change). Candidates that are uglified enough to avoid this problem would include options like

```
patternmatch (x)                    // note: one word
__match (x)
```

I find options in this category aesthetically unpleasant and in some cases embarrassing. We should not resort to such a syntax unless there are serious problems with every better one, and I think the earlier-mentioned syntax options do not have serious enough problems to warrant considering such ugly names.

Multiple new contextual words. C++ does not allow whitespace between identifiers, so we could use a multi-word introducer. For example

```
pattern match (x)                   // note: two words
match pattern (x)
on match (x)
match with (x)
```

or with more lookahead

```
match (x) with
with (x) match
```

I find options in this category less bad than an uglified new reserved word, but needlessly verbose which is undesirable for pattern matching expressions which need to be concise.

A single existing reserved word. If we want to prioritize a syntax that is easy to parse in the first two tokens, we can do no better than a keyword that is already reserved. For example, we could consider

```
switch (x)
case (x)
? (x)
```

However, I still think the best of these is `switch`, for the advantages mentioned above. And `?` is likely too terse for the statement form; in my opinion we want a word here.

An existing reserved word (possibly a symbol) plus an additional word. Examples include two reserved words such as

```
switch if (X)
if switch (X)
case if (X)
if case (X)
?? (x)
```

or one reserved word and one non-reserved word such as

```
match if (x)
if match (x)
match? (x)                      // note: could use match? for non-exhaustive and
match! (x)                      //       match! for exhaustive (I don't like this)
```

or with more lookahead

```
match (x) if
match (x) using
```

These are easy to parse, and not as terrible as uglified reserved words, but still not desirable in my opinion.

4.3.2 Introducing the new names: Whitespace or :

Priorities: The key is to pick a syntax that

- is short and clear;
- directly connotes the right thing; and
- is unambiguous (for compilers and for humans).

Consider using `is` in constrained declarations, boolean expressions and pattern matching alternatives. First, using decomposition:

```
if (auto&&  [x,y] is [even,odd] = z)     f(x, y*2);     // A: declaration: introduces x,y

if (std::tie(x,y) is [even,odd])         f(x, y*2);     // B: expression: preexisting x,y
    // Note: "if ([x,y] is [even,odd])" is not currently legal,
    //       although it has been considered as a synonym for B

inspect (z) {
    [x,y] is [even,odd] =>             { f(x, y*2); }   // C: alternative: introduces x,y
}
```

Note that C is both visually and semantically like A, not like B. Because we do not allow the same syntax for B (as mentioned in the above Note comment), there is no real visual ambiguity in the case where we are introducing decomposed names.

If we are introducing a non-decomposed name, there is a potential visual (not parsing) ambiguity. For example:

```
if (auto&& x is even = z)      f(x);                  // A2: declaration: introduces x

if (x is even)                 f(x);                  // B2: expression: preexisting x

inspect (z) {
    x is even =>               f(x);                  // C2: alternative: introduces x
}
```

Even though in this case all three are visually similar, we think that C2 is clear as written.

If we want to make this visually explicit, we could require `:` after introduced names:

```
inspect (z) {
    [x,y]: is [even,odd] => ...
    x: is even => ...
}
```

4.3.3 Introducing the selected result: `=` vs. `=>` vs. `case` vs. `then` vs. `:` vs. `->`

Priorities: The key is to pick a syntax that

- is short and clear;
- directly connotes the right thing;
- is unambiguous; and
- is general (not a special-purpose syntax that works only in pattern matching).

I prefer either `=` or `=>`, because both:

- are short and visible;
- effectively convey that the full `switch`/`match`/`inspect` evaluates to this result;
- is unambiguous in this position; and
- is generalizable for us outside `inspect` only (`=` is already used to mean "set to this result," and `=>` has already been proposed for terse lambdas, see below).

`case` is good and familiar and already reserved. Based on discussion so far, however, it is probably too tainted to use because it's the current construct that has a pitfall (fallthough) we are trying to remove and in my experience provokes the most antibodies of any suggestion on this list: `case` the only pattern matching keyword that I've seen get more "but that's the thing we're trying to get rid of!" resistance than `switch`.

`then` is excellent and general, and a keyword that is arguably logically missing from `if…else` already (C omitted the middle of `if…then…else`). But if we use it for pattern matching we will either not be general (it will be used there only) or we will face pressure to add it to `if…else` where it would be always redundant with the existing mandatory `( )` as well as with `{ }` when those are used, as discussed further in §4.3.4.

`:` works well today because it always comes as part of the triplet "`case` *value* `:`" which is visible as a whole. In the pattern matching examples where it's the only token between a pattern and a result, it's too short and tends to disappear. (Some may argue `=` has the same problem; I don't think it does, though of course `=>` is comparatively still more visible.)

`->` does not work generally because it already has a meaning in some places outside pattern matching where we might want to express "evaluates to this result," notably in lambdas…

Generalizability example: Terse lambdas. Note that `=>` has already been suggested for such a non-pattern-matching use, namely for single-expression lambdas in P0573:

```
[](x,y) => x+y                  // P0573
```

I personally like `=` here too equally with `=>`:

```
[](x,y) = x+y                   // equally good as => I think
```

And `case`, `:`, and `->` do not work as well:

```
[](x,y) case x+y                // doesn't make sense
[](x,y) : x+y                   // plausible, but not my preference
[](x,y) -> x+y                  // collides with return type syntax
```

If we do use `=>`: Taking it as a new token without other mitigation, as the P1371 prototype does, would break obscure code like this:

```
x<&y::operator=>                    // valid but extremely rare
```

which might be fine because this is very rare (codesearch has **3 hits** in 2.5 million source files) and could be fixed by the programmer adding a space. Alternatively, we can also add a language workaround rule to break the `=>` token into the two tokens `=` and `>` when it follows the keyword `operator`, similar to what we did for `>>`.

I think `operator=>` is the only currently legal token sequence that would be affected. Other cases found in codesearch include ones like this, which might be fine depending on what the implementation does internally:

```
#pragma data(heap_size => 3000)     // wouldn't be broken (example hit)
```

4.3.4 Delimiting the condition/body:) required or optional

Priorities: The key is to pick a syntax that

- is short and clear;
- directly connotes the right thing;
- is unambiguous; and
- is general (not a special-purpose syntax that works only in pattern matching).

I slightly prefer that the redundant `( )` parentheses can be omitted for `inspect`, if for consistency we now (or someday) also allowed omitting them for all uses of `if`, range-`for`, `switch`, `do`/`while`, and `while` that use `{ }`, because in all of those cases they are already redundant in all legal C and C++ code. For example, I would allow these:

```
switch a { ...
if a { ...
for x : y { ...
while a { ...
do { ... } while a ;
```

For the constructs that support loop-scoped variables, if you want those then write the full syntax with `( )`.

In all languages, the key is that there must be "a" well-known delimiter between the condition and what follows it. For example, all of these are unambiguous:

`IF a THEN thing`	Algol
`IF a DO thing`	BCPL
`if a then thing`	Pascal, Ada
`if a :` `  thing`	Python, : plus indented block
`if (a) thing`	C, C++
`if a { thing }`	Rust, Swift, and my suggestion here for future C++… just make optional the existing syntax that is already redundant

Two delimiters is always redundant, including every C and C++ `if`/etc. we've ever written using both `( )` and `{ }`:

`if (a) { thing }`	C, C++, Rust, Swift… *but having both) and { is always redundant* (Rust emits a "redundant parens" warning)

Note that there is a place C and C++ already do not require `( )` around branch conditions, which is fine because there is already a delimiter:

`a ? thing : thing2`	C, C++ — following BCPL's "`a -> thing1 , thing2`" (see below)

Historically, why did C require `( )` around control flow conditions, when BCPL did not? Well, BCPL didn't need to because it always had a delimiter in the required grammar:

```
IF a DO thing
TEST a THEN thing
TEST a DO thing
a -> thing
UNLESS a DO thing
WHILE a DO thing
UNTIL a DO thing
thing REPEATWHILE a
thing REPEATUNTIL a
SWITCHON a CASE ...
FOR a = b TO c DO thing
FOR a = b TO c by d DO thing
```

But when C made { } optional and did not add a keyword like then or similar in the grammar, it lost a reliable marker between the condition and the branch body. So it had to arrange for one, and picked). Perhaps it was following Fortran, which did require ( ), even though they were always redundant in Fortran because of the mandatory THEN:

```
IF (a) THEN thing        Fortran
```

In all C and C++ code ever written, these mandatory ( ) have always been redundant when { } are used around the body. Furthermore, we now understand the consequences of making { } optional are that code is more brittle especially under maintenance, when the programmer intends for two statements to be in a branch but they accidentally are not (even apart from macro expansion issues); if we had a time machine, with the benefit of hindsight I'm reasonably certain sure we would make ( ) optional and { } mandatory, instead of the reverse.

Why do I mention this now? Because we're talking about adding yet a new place we'd require ( ), and yet that ceremony is always redundant for proposed pattern matching if we require { } anyway.

So I think it's worth discussing the option of making the new pattern matching construct cleaner by not requiring ( ), and at the same time keep the language consistent by just also allowing ( ) to be omitted when it's already redundant (i.e., when { } are used) in if, switch, range-for, while, and do/while, and with zero code breakage (perfect backward compatibility).

4.4 Extension notes

4.4.1 Exceptional alternatives

[P2381R0] proposes allowing exceptions to be handled by pattern matching alternatives. For example:

P1371	P2381

`try { inspect (a()) {`
`  <reta> => { /*...*/ }`
`} }`
`catch(b const&) { /*...*/ }`
`catch(c const&) { /*...*/ }`
`catch(d const&) { /*...*/ }`
	`inspect (a()) {`
	`  <reta> => { /*...*/ }`
	`  <b>    => { /*...*/ }`
	`  <c>    => { /*...*/ }`
	`  <d>    => { /*...*/ }`
	`}`

This may be plausible, but we would have to think carefully about a few things.

First, we probably want an annotation like `catch` before the exceptional cases, for two reasons:

- Normal and error/exceptional control flows are fundamentally different and should be visually distinct. They are usually handled by different code, not the same code: A normal result is always used immediately at the call site and is a natural thing to `inspect` immediately, whereas an error is typically not handled immediately (it is atypical for the immediate call site to also know how to resolve an error).
- Any function could throw a type `E` to report errors, and also return the same type `E`, or a variant that includes `E`, etc. to report normal results. So the syntax for match a thrown `E` must be somehow distinct from all other patterns for a return type that could otherwise mention `E`.

Second, we would like generality: Is there a more general feature that would address the syntactic overhead in the left-hand example above? In [P0709R4] §4.5.3, I proposed a `catch` sugar that would allow writing the example as follows:

This paper, with P0709R4 §4.5.3 "catch" sugar	This paper, with exceptional alternatives and "catch"

`inspect (a()) {`
`  is some_thing  => { /*...*/ }`
`  is other_thing => { /*...*/ }`
`  // ...`
`}`
`catch(b) { /*...*/ }`
`catch(c) { /*...*/ }`
`catch(d) { /*...*/ }`
	`inspect (a()) {`
	`  is some_thing  => { /*...*/ }`
	`  is other_thing => { /*...*/ }`
	`  // ...`
	`  catch b        => { /*...*/ }`
	`  catch c        => { /*...*/ }`
	`  catch d        => { /*...*/ }`
	`}`

My inclination is to not pursue the right-hand approach unless we find compelling examples. There is little syntactic benefit, and the left-hand side has the advantage of keeping normal and error paths strictly separate.

Note that although error handling code should be separate, it could make sense to enable using `inspect` *within* error handling code. For example:

```
catch(auto const& err) {            // a potential extension, that would
  inspect(err)   {                  // let us use inspect naturally ehre
    is b => { /*...*/ }
```

```
      is c => { /*...*/ }
      is d => { /*...*/ }
    }
  }
```

A potential compelling example emphasized in [P2381R0] is the desire to treat errors uniformly, whether they are encoded in the return type (e.g., `result<R,E>`) which can be handled naturally in any pattern matching design, or thrown using an exception would require an extension. However, most of the motivation for proliferating errors encoded in the return type is because of exception handling semantics and costs, and so for this example my preference would be to fix exceptions so that we can use them uniformly instead of proliferating ways of reporting errors (see [P0709R4]). And even if we did want to allow pattern matching to handle errors uniformly whether returned by codes or thrown as exceptions, we will won't achieve that unless we have a uniform syntax for the two, which likely would require a way to identify return type that bear errors so they can be treated uniformly by `catch` alternatives.

4.5 History and related work

In 2013-14, modern C++ pattern matching work was prominently pioneered by Yuriy Solodkyy et al. in [Mach7] and [Solodkyy 2013]. This formed the basis for the Urbana 2014 evening session [Solodkyy 2014b]. Interestingly, this paper's proposed syntax is remarkably consistent with the form shown in Urbana; see §3.5.6.

While exploring ways to simplify C++ during 2015-2016, I had noticed the usefulness of general `is` and `as`, probably influenced by seeing that basic syntax in C#. In WG21, I initially proposed `is` and `as` specifically for metaclasses in 2017 in [P0707R0], and then later in private conversations further developed how those could be generalized in C++ to cover general type and value constraints broadly throughout the language, from simple `if` conditions to template requirements/specializations, including pattern matching. This paper existed in basically its current form in early 2019; it now includes refinements from private discussions during March-June 2021.

Independently, in 2017 C# released version 7.0 with pattern matching that embodied very similar parallel ideas. C# had `is` and `switch` since version 1.0 (2002) but they were separate features until they were unified/generalized via pattern matching in C# 7.0 (2017) in much the same way that this proposal unifies/generalizes `is` with the generalized `switch` (aka `inspect`). The C# language designers commented on how well and naturally the original C# `is` and `switch` generalized to pattern matching, although not initially designed for patterns.

> **Note** One major difference between C# and this proposal is that in C# `is` and `as` are not currently customizable; they never run user-written code. This is an extension C# could allow in the future.

Also independently, in 2018-2020, Michael Park's [P1260R0], David Sankel et al.'s [P1308R0], and Bruno Cardoso Lopes et al.'s in [P1371R3] pattern matching proposals for C++ again embodied similar parallel ideas:

- `is` allows `v is pattern` as a general form of [P1260R0] §4.5 **`pattern matches v`** and [P1371R3] §5.5 **`MATCHES(pattern,v)`**. In the previous papers, `matches` and `MATCHES` are exposition-only, but both hint at making that feature usable also outside `inspect` statements, albeit as pseudocode. In this paper, **`v is pattern`** is first-class boolean expression allowed throughout the language.
- `as` is a general form of [P1260R0] §4.3.2.5 **`as<T>`** and [P1371R3] **`<T>`**. Although [P1260R0] shows `as<>` being used inside `inspect` statements, the implementation suggested in section 4.3.2.5 hints at something usable outside `inspect` statements as well. In this paper, **`v as T`** is a first-class expression allowed throughout the language.

I think it's a promising sign that this similar approach of expressing pattern matching using a generalized "is" and a generalized "switch" has been independently reinvented several times, as well as the directly related of a generalized "as" in result alternatives, and that the basic form remains consistent with the ideals of Urbana 2014.

Additionally, for decomposition and name introduction, this paper reuses (and expands) `[ ]` structured bindings, which also is parallel to the previous papers:

- `[ ]` is a similar approach to the `[ ]` syntax in the three recent prior papers. In this paper, it strictly hews to structured bindings syntax and semantics, including extending both symmetrically to allow generalized predicates, nested patterns, and wildcards.

In "surface syntax," this paper tries to minimize distracting differences from the current pattern matching proposals:

- `=>` to introduce alternative results directly follows the `=>` syntax in [P1308R0] and [P1371R3].
- `inspect` as the main introducer directly follows the `inspect` syntax in the three recent prior papers.

5 Bibliography

[Augustsson 2005] L. Augustsson. “Compiling pattern matching” (*Lecture Notes in Computer Science, vol. 201*, June 2005).

[Bandela 2018] J. Bandela. “simple_match: Simple, Extensible C++ Pattern Matching Library” (GitHub, updated June 2018).

[Boost.Any] K. Henney and A. Polukhin. “Boost.Any” (Boost).

[LLVM Variant] `llvm::pdb::Variant` type (LLVM.org).

[Mach7] Y. Solodkyy, G. Dos Reis, B. Stroustrup. “Mach7: Pattern Matching for C++” (GitHub, updated December 2019).

[Maranget 2008] L. Maranget. “Compiling Pattern Matching to good Decision Trees” (*Proceedings of the 2008 ACM SIGPLAN workshop on ML*, September 2008).

[N3449] B. Stroustrup. “Open and Efficient Type Switch for C++” (WG21 paper, September 2012).

[N3804] B. Dawes, K. Henney, D. Krügler. “*Any* Library Proposal” (WG21, paper, October 2013).

[P0095R1] D. Sankel. “Pattern Matching and Language Variants” (WG21 paper, May 2016).

[P0144R2] H. Sutter, B. Stroustrup, G. Dos Reis. “Structured bindings” (WG21 paper, March 2016).

[P0326R0] V. J. Botet Escribá. “Structured binding: Customization points issues” (WG21 paper, May 2016).

[P0327R3] V. J. Botet Escribá. “Product-type access (revision 3)” (WG21 paper, October 2017).

[P0707R0] H. Sutter. “Metaclasses” (WG21 paper, June 2017).

[P0709R4] H. Sutter. “Zero-overhead deterministic exceptions: Throwing values” (WG21 paper, August 2019).

[P1096R0] T. Doumler. “Simplify the customization point for structured bindings” (WG21 paper, October 2018).

[P1110R0] J. Yasskin, JF Bastien. “A placeholder with no name” (WG21 paper, June 2018).

[P1179R1] H. Sutter. “Lifetime safety: Preventing common dangling” (WG21 paper, November 2019).

[P1260R0] M. Park. “Pattern Matching” (WG21 paper, May 2018).

[P1308R0] D. Sankel, D. Sarginson, S. Murzin. “Pattern Matching” (WG21 paper, October 2018).

[P1371R3] B. Cardoso Lopes, S. Murzin, M. Park, D. Sankel, D. Sarginson, B. Stroustrup. “Pattern Matching” (WG21 paper, September 2020).

[P1469R0] S. Murzin, M. Park, D. Sankel, D. Sarginson. “Disallow _ Usage in C++20 for Pattern Matching in C++23” (WG21 paper, January 2019).

[P2169R2] C. Jabot and M.Park. “A nice placeholder with no name” (WG21 paper, September 2020).

[P2381R0] J. Waterloo. “Pattern Matching with Exception Handling” (WG21 paper, April 2021).

[Park 2019] M. Park. “MPark.Patterns” (GitHub, updated October 2019).

[Rust PM] “Patterns” (Rust documentation).

[Sayle 2008] R. A. Sayle. “A Superoptimizer Analysis of Multiway Branch Code Generation” (Proceedings of the GCC Developers’ Summit, June 2008]

[Solodkyy 2012] Y. Solodkyy, G. Dos Reis, B. Stroustrup. “Open and Efficient Type Switch for C++” (OOPSLA, October 2012).

[Solodkyy 2013] Y. Solodkyy, G. Dos Reis, B. Stroustrup. “Open Pattern Matching for C++” (ACM International Conference on Generative Programming and Component Engineering, October 2013).

[Solodkyy 2014] Y. Solodkyy. “Mach7: The Design and Evolution of a Pattern Matching Library for C++” (C++ Now, May 2014).

[Solodkyy 2014a] Y. Solodkyy. “Accept No Visitors” (CppCon, September 2014).

[Solodkyy 2014b] Y. Solodkyy, G. Dos Reis, B. Stroustrup. “Pattern Matching for C++” (WG21 presentation at Urbana-Champaign, November 2014).

[Swift PM] “Patterns” (Swift documentation, last major section update September 2019).

[Wakely 2020] J. Wakely. Top answer to “Get the status of a std::future” (StackOverflow, June 2012 and updated November 2020).