Pattern Matching: `match` Expression

Document #:	P2688R1
Date:	2024-02-15
Project:	Programming Language C++
Audience:	Evolution

Reply-to:	Michael Park <mcypark@gmail.com>

Contents

1 Introduction

This paper fleshes out the version of pattern matching that was described in [P2688R0]. It introduces a unified `match` expression that can perform a single pattern match using the syntax:

```
expression match pattern
```

as well as the selection of pattern matches using the following syntax:

```
expression match {
    pattern => expression-statement
    /* ... */
}
```

A single pattern match yields a boolean, and therefore can be used in other contexts such as `if`, `while`, `for`, and `requires`.

`let` is used to introduce new names within patterns, and are always "bindings" like the ones introduced by structured bindings in C++17 [P0144R2].

The set of patterns are trimmed down to match the following entities:

1. Values (e.g. `42`, `"hello"`, `compute_value()`)
2. Pointer-like types (e.g. `T*`, `optional`)
3. Tuple-like types, extending structured bindings (e.g. `pair`, `tuple`)
4. Sum types (e.g. `variant`, `expected`, `any`, `exception_ptr`, polymorphic types)

2 Motivation and Scope

The goal and motivation of this paper is to make further progress on pattern matching for C++.

At the Kona meeting in November 2022, the previous version of this paper [P2688R0] and [P2392R2] was discussed over whether patterns should be composed vs chained. EWG clearly expressed the desire for patterns to be composable (i.e. nested patterns):

> Poll: "EWG prefers composition over chaining in pattern matching syntax."
>
> Result: SF: 13, F: 9, N: 2, A: 1, SA: 0

This paper presents (as did [P1371R3]) a design that continues to offer composable patterns.

At the EWG Telecon July 7, 2021, EWG clearly expressed the desire for pattern matching to be available outside of `inspect`:

> Poll: "Should we spend more time on patmat expressions outside of `inspect` (as proposed in P2392 or otherwise), knowing that time is limited and we already have put in a lot of effort towards another patmat proposal?"
>
> Result: SF: 11, F: 12, N:4, A: 2, SA: 0

This paper offers single pattern matching via *`expression`* `match` *`pattern`* which is similar to the `is`-expression from [P2392R2]. See

Additionally, it aims to address the following pieces of feedback:

> "Declaration of new names should have an introducer like most other places in the language."

New names need the `let` introducer to introduce bindings, just like other new names in most other places in the language.

> "We shouldn't bifurcate expressions like this."

That is, expressions are just expressions without needing anything everywhere else in the language. This is true in this design. That is, `x` by itself is an expression referring to an existing name like it does everywhere else.

> "I don't want the documentation of pattern matching to have to mention a caveat that `x` is a new name and therefore shadows an existing variable."

As mentioned above, `x` is an expression that refers to an existing variable.

Another contribution of this paper is Static and Dynamic Conditions, which aim to more clearly specify and discuss the framework of requirements for patterns. They determine how uses of patterns are checked and/or tested at compile-time and/or runtime, within template contexts and outside.

Features such as predicates, extractors, structured bindings with designators, static type matching by type or concepts, and pattern combinators `and` and `or` are proposed to be deferred as future extensions.

The following is a list of key goals of the paper:

- Introduce `match` expression with `let` bindings.
- Trim down the set of patterns to focus on.
- Allow pattern matching in more places.
- Determine how patterns should be treated in templates.

3 Comparison Tables

The following are 4-way comparison tables between C++23, [P1371R3], [P2392R2], and this paper.

3.1 Matching Integrals

C++23

```
switch (x) {
  case 0: std::print("got zero"); break;
  case 1: std::print("got one"); break;
  default: std::print("don't care");
}
```

P1371R3

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

P2392R2

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

This Paper

```
x match {
  0 => std::print("got zero");
  1 => std::print("got one");
  _ => std::print("don't care");
};
```

3.2 Matching Strings

C++23

```
if (s == "foo") {
  std::print("got foo");
} else if (s == "bar") {
  std::print("got bar");
} else {
  std::print("don't care");
}
```

P1371R3

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

P2392R2

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

This Paper

```
s match {
  "foo" => std::print("got foo");
  "bar" => std::print("got bar");
  _ => std::print("don't care");
};
```

3.3 Matching Tuples

C++23

```
auto&& [x, y] = p;
if (x == 0 && y == 0) {
  std::print("on origin");
} else if (x == 0) {
  std::print("on y-axis at {}", y);
} else if (y == 0) {
  std::print("on x-axis at {}", x);
} else {
  std::print("at {}, {}", x, y);
}
```

P1371R3

```
inspect (p) {
  [0, 0] => std::print("on origin");
  [0, y] => std::print("on y-axis at {}", y);
  [x, 0] => std::print("on x-axis at {}", x);
  [x, y] => std::print("at {}, {}", x, y);
};
```

P2392R2

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

This Paper

```
p match {
  [0, 0] =>
    std::print("on origin");
  [0, let y] =>
    std::print("on y-axis at {}", y);
  [let x, 0] =>
    std::print("on x-axis at {}", x);
  let [x, y] =>
    std::print("at {}, {}", x, y);
};
```

3.4 Matching Variants

C++23

```
struct visitor {
  void operator()(int32_t i32) const {
    std::print("got int32: {}", i32);
  }
  void operator()(int64_t i64) const {
    std::print("got int64: {}", i64);
  }
  void operator()(float f) const {
    std::print("got float: {}", f);
  }
  void operator()(double d) const {
    std::print("got double: {}", d);
  }
};
std::visit(visitor{}, v);
```

P1371R3

```
inspect (v) {
  <int32_t> i32 =>
    std::print("got int32: {}", i32);
  <int64_t> i64 =>
    std::print("got int64: {}", i64);
  <float> f =>
    std::print("got float: {}", f);
  <double> d =>
    std::print("got double: {}", d);
};
```

P2392R2

```
inspect (v) {
  i32 as int32_t =>
    std::print("got int32: {}", i32);
  i64 as int64_t =>
    std::print("got int64: {}", i64);
  f as float =>
    std::print("got float: {}", f);
  d as double =>
    std::print("got double: {}", d);
};
```

This Paper

```
v match {
  int32_t: let i32 =>
    std::print("got int32: {}", i32);
  int64_t: let i64 =>
    std::print("got int64: {}", i64);
  float: let f =>
    std::print("got float: {}", f);
  double: let d =>
    std::print("got double: {}", d);
};
```

This example is matching the variant alternatives using concepts.

C++23

```
struct visitor {
  void operator()(
      std::integral auto i) const {
    std::print("got integral: {}", i);
  }
  void operator()(
      std::floating_point auto f) const {
    std::print("got float: {}", f);
  }
};
std::visit(visitor{}, v);
```

P1371R3

```
inspect (v) {
  <std::integral> i =>
    std::print("got integral: {}", i);
  <std::floating_point> f =>
    std::print("got float: {}", f);
};
```

P2392R2

```
// not supported
```

This Paper

```
v match {
  std::integral: let i =>
    std::print("got integral: {}", i);
  std::floating_point: let f =>
    std::print("got float: {}", f);
};
```

3.5 Matching Polymorphic Types

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

C++23

```
virtual int Shape::get_area() const = 0;

int Circle::get_area() const override {
  return 3.14 * radius * radius;
}
int Rectangle::get_area() const override {
  return width * height;
}
```

P1371R3

```
int get_area(const Shape& shape) {
  return inspect (shape) {
    <Circle> [r] => 3.14 * r * r;
    <Rectangle> [w, h] => w * h;
  };
}
```

P2392R2

```
int get_area(const Shape& shape) {
  return inspect (shape) {
    [r] as Circle => 3.14 * r * r;
    [w, h] as Rectangle => w * h;
  };
}
```

This Paper

```
int get_area(const Shape& shape) {
  return shape match {
    Circle: let [r] => 3.14 * r * r;
    Rectangle: let [w, h] => w * h;
  };
}
```

3.6 Matching Nested Structures

```
struct Rgb { int r, g, b; };
struct Hsv { int h, s, v; };

using Color = variant<Rgb, Hsv>;

struct Quit {};
struct Move { int x, y; };
struct Write { string s; };
struct ChangeColor { Color c; };

using Command = variant<Quit, Move, Write, ChangeColor>;

Command cmd = ChangeColor { Hsv { 0, 160, 255 } };
```

C++23

```
struct CommandVisitor {
  void operator()(Quit) const {}
  void operator()(const Move& move) const {
    const auto& [x, y] = move;
    // ...
  }
  void operator()(const Write& write) const {
    const auto& text = write.s;
    // ...
  }
  void operator()(
      const ChangeColor& cc) const {
    struct ColorVisitor {
      void operator()(const Rgb& rgb) {
        const auto& [r, g, b] = rgb;
        // ...
      }
      void operator()(const Hsv& hsv) {
        const auto& [h, s, v] = hsv;
        // ...
      }
    };
    std::visit(ColorVisitor{}, cc.c);
  }
};
std::visit(CommandVisitor{}, cmd);
```

P1371R3

```
inspect (cmd) {
  <Quit> _ => // ...
  <Move> [x, y] => // ...
  <Write> [text] => // ...
  <ChangeColor> [<Rgb> [r, g, b]] => // ...
  <ChangeColor> [<Hsv> [h, s, v]] => // ...
};
```

P2392R2

```
inspect (cmd) {
  is Quit => // ...
  [x, y] as Move => // ...
  [text] as Write => // ...
  [[r, g, b]] as ChangeColor as [Rgb] => // ...
  [[h, s, v]] as ChangeColor as [Hsv] => // ...
}
```

This Paper

```
cmd match {
  Quit: _ => // ...
  Move: let [x, y] => // ...
  Write: let [text] => // ...
  ChangeColor: [Rgb: let [r, g, b]] => // ...
  ChangeColor: [Hsv: let [h, s, v]] => // ...
};
```

Example from Destructuring Nested Structs and Enums section from Rust documentation.

4 Design Overview

The overall idea is to introduce a single `match` construct that can be used to perform single pattern match, or a selection of pattern matches.

```
expression match {
  pattern => expression-statement
  // ...
}
```

`let` denotes that an identifier is a new name rather than an existing name.

```
constexpr int x = 42;

expression match {
  x => ... // match against existing `x`
  let x => ... // introduce new x.
}
```

On the right of `=>` is an *expression-statement* rather than a *statement*. This means that only expressions are allowed, and I believe it will be best to pursue `do` expressions [P2806R2] to do *statement* things.

The following is used to match a value against a single pattern.

```
expression match pattern
```

The following is the `match` expression being used within an `if` statement.

```
if (expr match [0, let foo]) {
  // `foo` is available here
} else {
  // but not here
}
```

A optional guard can be added for a single pattern match as well:

```
std::pair<int, int> fetch(int id);

bool is_acceptable(int id, int abs_limit) {
  return fetch(id) match let [min, max] if -abs_limit <= min && max <= abs_limit;
}
```

4.1 Syntax Overview

```
// Single pattern match
expr-or-braced-init-list match constexpr_opt pattern guard_opt

// Selection pattern match
expr-or-braced-init-list match constexpr_opt trailing-return-type_opt {
    pattern guard_opt => expr-or-braced-init-list ;
    pattern guard_opt => break ;
    pattern guard_opt => continue ;
    pattern guard_opt => return expr-or-braced-init-list_opt ;
}

guard:
    if expression
```

```
pattern:
    match-pattern
    let binding-pattern
    match-pattern let binding-pattern
    ...  // only in structured bindings pattern (P1061)

match-pattern:
    _                           // wildcard
    constant-expression         // value
    ( pattern )                 // grouping
    ? pattern                   // pointer-like
    discriminator : pattern     // variant-like, polymorphic, etc.
    [ pattern-0 , … , pattern-N ]    // tuple-like

binding-pattern:
    identifier
    [ binding-pattern-0 , … , binding-pattern-N ]
    ... identifier  // only in structured bindings pattern (P1061)

discriminator:
    type-id
    type-constraint
```

4.2 Pattern Specifications

4.2.1 Wildcard Pattern

_

A wildcard pattern always matches any *subject*.

```
int v = 42;
v match {
    _ => std::print("ignored");
//  ^  wildcard pattern
};
```

This paper reattempts for _ to be the wildcard pattern. See Wildcard Pattern Syntax for further discussion.

— Matching Condition: None

4.2.2 Let Pattern

let *binding-pattern*

A let pattern always matches any *subject*. The *binding-pattern* is either an *identifier* or a structured bindings pattern.

```
int v = 42;
v match {
    let x => std::print("ignored");
//  ^^^^^  let pattern
};
```

let can be used to introduce new names individually, or all-in-one.

```
let x           // x is new
[a, let y]      // a is old, y is new
```

```
[let x, b]      // x is new, b is old
let [x, y]      // x and y are both new
let [x, [y, z]] // x, y, z are all new
```

match-pattern `let` *binding-pattern*

A `let` pattern can appear after a *match-pattern* to create bindings to the value that was matched with *match-pattern*.

```
int i = 42;
i match {
  42 => // match 42
  let x => // bind name
  42 let x => // match 42 and bind name at the same time
};

std::pair p = {0, 0};
p match {
  [0, let y] => // match and bind a piece
  let whole => // bind whole pair
  [0, let y] let whole => // do both
};
```

4.2.3 Constant Pattern

constant-expression

A constant pattern tests the value of ***subject*** against the value of the constant pattern. The constant pattern can be any ***constant-expression***, such as literals, `constexpr` variables, or values of an `enum`.

— Matching Condition: `bool(`*subject* `==` *constant-expression*`)`;

4.2.4 Parenthesized Pattern

`(` *pattern* `)`

A parenthesized pattern is used to group undelimited patterns.

— Matching Condition: *subject* `match` *pattern*

Example:

```
void f(const Shape* s) {
    s match {
        ? (Circle: let c) => // ...
        ? (Rectangle: let r) => // ...
        _ => // ...
    };
}
```

```
std::optional<int> maybe_int();

void f() {
    maybe_int() match {
        (? let i) let o => // i is int, o is the whole optional
        _ => // ...
    };
}
```

4.2.5 Optional Pattern

`? pattern`

An optional pattern tests pointer-like objects. It matches if *subject* contextually converts to `true` and `*subject` matches *pattern*.

— Matching Condition: `bool(subject) && *subject match pattern`

4.2.6 Alternative Pattern

type-id `:` *pattern*
type-constraint `:` *pattern*

An alternative pattern tests sum type objects such as `variant`, `any`, and polymorphic types.

Let `s` be *subject*, `S` be `std::remove_cvref_t<decltype(subject)>`.

Case 1: Variant-like

An alternative pattern matches if the `variant`-like object stores a value of type *type-id* or the value of type satisfies *type-constraint*, and the stored value matches *pattern*.

If `std::variant_size<S>` is well-formed and `std::variant_size<S>::value` is an integral, let `I` be the value of `s.index()`. An alternative pattern matches if `std::variant_alternative_t<I, S>` is *type-id* or if it satisfies *type-constraint*, and *pattern* matches `get<I>(s)`.

Case 2: Casts

If `auto* p = cast<S>::operator()<`*type-id*`>(s)` is well-formed, alternative pattern matches if `p` contextually converts to `true` and `std::forward_like<decltype(s)>(*p)` matches *pattern*.

A `cast` customization point is proposed, rather than using `any_cast`. Since `any` has an implicit constructor from anything, overloading `any_cast` which takes `const any&` will likely cause a problem. Moreover, [P2927R0] is in the process of introducing `std::try_cast`.

```
template <typename>
struct cast;

template <>
struct cast<std::any> {
  template <typename T>
  static const T* operator()(const std::any& a) noexcept {
    return std::any_cast<T>(&a);
  }

  template <typename T>
  static T* operator()(std::any& a) noexcept {
    return std::any_cast<T>(&a);
  }
};

template <>
struct cast<std::exception_ptr> {
  template <typename T>
  static const T* operator()(const std::exception_ptr& p) noexcept {
    return std::try_cast<T>(p); // P2927R0
  }
};
```

Case 3: Polymorphic Types

This is listed as a separate case in case it's needed for optimization flexibility. In principle though, the following specialization of `cast` should provide the desired semantics.

```
template <typename Base>
requires requires { std::is_polymorphic_v<T>; }
struct cast<Base> {
  template <typename T>
  static const T* operator()(const Base& b) noexcept {
    return dynamic_cast<const T*>(&b);
  }

  template <typename T>
  static T* operator()(Base& b) noexcept {
    return dynamic_cast<T*>(&b);
  }
};
```

4.2.7 Structured Bindings Pattern

[pattern-0 , … , pattern-N]

Given the following structured binding declaration:

```
auto&& [ e-0, …, e-N ] = subject ;
```

Let *e-i* be a unique exposition-only identifier if *pattern-i* is a *pattern* and an ellipsis (`...`) if *pattern-i* is an ellipsis (`...`). Structured bindings pattern matches *subject* if *e-i* matches *pattern-i* for all i where *e-i* is an identifier.

4.3 Scope of Bindings

The scope of the bindings introduced by `let` are as follows:

- If the *pattern* is left of `=>`, the scope of the binding is the corresponding expression statement.
- If the *pattern* is in ***expression*** `match` ***pattern guard***$_{opt}$, the scope of the binding is the expression including the optional guard, unless:
- If the construct is the *condition* of an `if` statement, the scope of the binding is the *then* substatement of the `if` statement.
- If the construct is the *condition* of a `for`, or `while` statement, the scope of the binding is the substatement of `for` or `while` statement.

Example:

```
bool b1 = e1 match [0, let x] if x > 1;
// x not available here.

bool b2 = e2 match [let x]; // not a redeclaration
// x not available here.

if (e3 match (? let elem)) {
  // elem available here
} else {
  // elem not available here
}

while (queue.next() match (? let elem)) {
  // elem available here
}
```

4.4 Static and Dynamic Conditions

Every *pattern* has a corresponding condition which is tested against the *subject* to determine whether the *pattern* matches the *subject*.

For example, the constant pattern `0` has a condition that it matches if *`subject`* `==` `0` is true. However, there are static and dynamic dimensions to which this condition can be applied. These dimensions are defined here.

4.4.1 Static Conditions

Static conditions are the static requirements of a pattern. The patterns being introduced in this paper have dynamic behavior, and therefore their static conditions are the validity of a *pattern*'s match condition.

See Static Type Checking with Constraint Pattern for an example where this isn't the case.

The main question is, are these static requirements checked or tested? Going back to the constant pattern `0`, its static condition is whether *`subject`* `==` `0` is a valid expression.

```
void f1(int x) {
  x match {
    0 => // ...
    _ => // ...
  };
}
```

In this example, whether `x == 0` is a valid expression is checked at compile-time. If `x` is a `std::string` for example, the program is ill-formed.

```
void f2(std::string x) {
  x match {
    0 => // ill-formed
    _ => // ...
  };
}
```

This behavior is likely to be pretty obvious to folks. But what if `x` were a templated parameter instead?

```
void f3(auto x) {
  x match {
    0 => // fine here
    _ => // ...
  };
}

f3("hello"s); // proposed: ill-formed
```

This paper proposes that this example be ill-formed at the instantiation site. While a model that treats `0` as a no-match would be doable, I believe it'll be better and safer as an opt-in feature. For `f3<std::string>` to have different type-checking behavior than `f2` would be novel and likely lead to subtle bugs.

This means that static conditions of patterns are always checked and enforced at compile-time. See More on Static Conditions for further design discussions, and Testing the Static Conditions with `match requires` which suggests an extension to explicitly treat the static conditions as compile-time tests rather than checks.

The semantics for this was not precisely defined in [P1371R3], and [P2392R2] proposes for `f3("hello"s)` to be well-formed and `0` is a no-match.

4.4.2 Dynamic Conditions

Dynamic conditions are more obvious and straight-forward. The constant pattern `0` matches if *`subject`* `== 0` is true. But true when?

This paper proposes that `match` tests the dynamic condition at runtime, (think `if`) and `match constexpr` tests it at compile-time (think `if constexpr`).

`match`

```
void f(int x) {
  x match {
    0 => // ...
    1 => // ...
    _ => // ...
  };
}
```

`match constexpr`

```
template <std::size_t I>
const auto& get(const S& s) {
  return I match constexpr -> const auto& {
    0 => s.foo();
    1 => s.bar();
    _ => static_assert(false);
  };
}
```

5 Design Decisions and Discussions

5.1 Unified `match` Expression

The `match` expression presented in this paper unifies the syntax for a single pattern match and a selection of pattern matches. Namely, *`expr`* `match` *`pattern`* and *`expr`* `match { ... }`.

The single pattern match *`expr`* `match` *`pattern`* is very similar to *`expr`* `is` *`pattern`* introduced in [P2392R2].

Early attempts at pattern matching with `inspect` also explored the idea of being a statement and an expression depending on its context. In short, if it appears in an expression-only context (e.g. `int x = inspect { ... };`) then it's an expression. If it appears in a context where a statement or an expression can appear (e.g. `{ inspect { ... } }`), then it's interpreted as a statement.

Having to differentiate between the statement-form and expression-form was a novel situation with no other precedent in the language. Additionally, whatever the keyword, it would've needed to be a *full* keyword. Maybe `inspect` would've been okay, but something like `match` was not even a possibility.

With this approach, `match` is feasible as a context-sensitive keyword, and and there is only an expression-form, which simplifies the design.

5.2 Wildcard Pattern Syntax

This paper proposes `_` as the syntax for wildcard patterns. Note that this is different from bindings that are introduced with the name `_`.

For example,

```
e match {
    _ => // ...
//  ^ this is a wildcard
    let [_, _] => // ...
//       ^  ^ these are bindings
};
```

In the bindings case, the semantics are the same as [P2169R4], which was accepted for C++26. That is, a single declaration of `_` is usable but a use after a redeclaration is ill-formed.

In the wildcard case, it is a special rule in that `_` can be an existing variable. For example,

```
int i = 101;
int _ = 202;

i match {
    _ => // 101 != 202 but _ is a wildcard, so this matches.
};
```

The recommended workaround is to use a guard:

```
int i = 101;
int _ = 202;

i match {
    let x if x == _ => // ...
};
```

— [P1371R3] proposed __ which was also the syntax recommended in [P1110R0].
— [P1469R0] proposed disallowing use of _ as an identifier in the context of structured bindings, but this was rejected by EWG. as it's not referred to, and was accepted for C++26.
— [P2392R2] proposed _ as well.

This is a relatively small cost to get _ as the wildcard pattern, given the prevalence and scope of adoption of _ across the industry. Languages such as Python, Rust, Scala, Swift, C#, Erlang, Prolog, Haskell, OCaml and many others already use _. Pattern matching facilities across different languages do vary, but I'm not aware of *any* language that disagree on _.

5.3 Why We Want Expressions in Patterns

If expressions are not supported at all, this would mean we couldn't do some of the most simple operations that `switch` can handle. We should be able to at the very least match integrals, strings, and enums.

So we need to allow expressions at least in *some* capacity. Let's say for example we only allow literals. This would give us matching for integral and string literals, but we wouldn't be able to match against `constexpr` variables of integrals and strings.

It also doesn't get us enums, since enum values are not literals. We need unqualified names to be able to access `enum` values, and qualified names to be able to access `enum class` values.

At this point, we already basically have *primary-expression*. The question of how to handle referring to existing names vs introducing new names have to be addressed. Only allowing *primary-expression* rather than *constant-expression* might still be useful or needed to avoid further grammar complications, but the fundamental issue of existing vs new names I don't think could nor should be avoided.

5.4 Exploration of Variable Declaration Syntax for Alternative Pattern

The proposed syntax in this paper is

```
type-id : pattern
type-constraint : pattern
```

Here's a simple example:

```
std::variant<int, bool, std::string> parse(std::string_view);

parse(some_input) match {
  int: let i => // ...
  bool: let b => // ...
```

```
  std::string: let s => // ...
};
```

This looks more like `case` labels where the alternatives are listed and the appropriate one is chosen. The correponding value is then matched with a nested pattern.

The absolute minimal syntax would be `std::string s`, which is rather appealing but ultimately not what is proposed.

An example using this syntax might be something like:

```
std::variant<int, bool, std::string> parse(std::string_view);

parse(some_input) match {
  int i => // ...
  bool b => // ...
  std::string s => // ...
};
```

Question 1: What **are** `i`, `b`, and `s`?

They certainly look like variable declarations, and I think it'll be too surprising for them to be anything else. So let's for now assume that they are variable declarations. In this case, they should probably used as a general way to introduce new names within a pattern for binding purposes. We want patterns to compose, so this applies to nested patterns as well, but at the top-level this might look like:

```
int parse_int(std::string_view);

parse_int(some_input) match {
  0 => // ...
  1 => // ...
  auto i => // use `i` which is `int` returned by `parse_int`
            // not 0 or 1
}
```

Question 2: How do you disambiguate `auto x` between `variant` itself vs the alternative inside?

`std::variant` is a very unique sum type, in that you are able to handle the "catch-all" case where you can generically access the value inside of it.

C++23	Variable Declaration Approach

`std::visit(`
`  overload(`
`  [](int i) { /* ... */ },`
`  [](auto x) { /* ... */ }),`
`  parse(some_input));`
	`parse(some_input) match {`
	`  int i => // ...`
	`  auto x => // ...`
	`};`

But what is `x`? Is it the unhandled alternatives of the variant, or is it the variant itself?

In the `parse_int` example from above, `auto i` was a binding to the whole value!

Note that for polymorphic types we could actually make this work since there's no way to generically operate on the runtime value of a polymorphic type anyway.

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

```
const Shape& get_shape();

get_shape() match {
  const Circle& c => // runtime downcast to `Circle`.
  const auto& s => // `s` can't generically be `Triangle` or `Rectangle` anyway.
};
```

This is what C# does for example:

```
Shape get_shape();

get_shape() switch {
  Circle c => // runtime downcast to `Circle`
  var s => // `s` is the whole shape.
};
```

While this syntax would work for polymorphic types specifically, there is a general desire to unify the handling of sum types like `variant` and polymorphic types. For example, [P2411R0] points out:

> The 'is'-and-'as' notation [P2392] is cleaner and more general than the [P1371] and successors notation. For example, it eliminates the need to use the different notation styles for variant, optional, and any access. Uniform notation is the backbone of generic programming.

[P1371R3] already had uniform notation at least for `variant`, `any` and polymorphic types, but regardless, the point is that using syntax that works only for polymophic types but not `variant` is not desired.

Question 3: Initialization? Conversions? First-match? Best-match?

Going back to the first example:

```
std::variant<int, bool, std::string> parse(std::string_view);

parse(some_input) match {
  int i => // ...
  bool b => // ...
  std::string s => // ...
};
```

Are these variable declarations initialized by direct-initialization, copy-initialization, list-initialization, something else? Having to answer this question isn't necessarily a blocker, but one needs to be chosen.

Regardless of the answer though, there's no initialization form that disallows conversions in general. If these have first-match semantics (the only form of matching that has been proposed so far), `int i` would match if the variant is in the `bool` state, since all of these are valid:

```
int i1(true); // direct
int i2 = true; // copy
int i3{true}; // list
int i4 = {true}; // copy-list
```

On the other hand, best-match semantics would introduce significant complexity. Value-matching needs to consider becoming best-match, and this would likely mean evaluating more than necessary in order to compute a score to best-match with. If value-matching remained first-match, then we would have best-match semantics weaved info first-match semantics. This is likely very difficult for users.

Note that even with best-match semantics, allowing conversions makes code like this difficult to diagnose missing cases:

```
parse(some_input) match {
  int i => // ...
  std::string s => // ...
  // maybe missing bool case? it is covered by `int` though...
};
```

Question 4: How do we match against an existing value?

Variable declaration syntax isn't conducive to referring to an existing value. Suppose there is a constexpr value `batch_size` that we want to match against. `int batch_size` wouldn't work since that would be introducing a new variable. `batch_size` could be annotated somehow, but annotating existing names rather than the new names has already been attempted.

More generally, variable declaration syntax isn't conducive to composition.

With this paper, the first example would be written as:

```
std::variant<int, bool, std::string> parse(std::string_view);

parse(some_input) match {
  int: let i => // ...
  bool: let b => // ...
  std::string: let s => // ...
};
```

1. `i`, `b`, and `s` are bindings, introduced by `let`.
2. How do you disambiguate `auto x` between `variant` itself vs the alternative inside?

With this paper, `let x` binds the whole value, whereas `auto: let x` binds to the value inside the variant. The following is an example of `let x` binding the whole value:

```
int parse_int(std::string_view);

parse_int(some_input) match {
  0 => // ...
  1 => // ...
  let i => // use `i` which is `int` returned by `parse_int`
           // not 0 or 1
}
```

The following is an example of `auto: let x` where we bind the alternative inside the variant.

```
std::visit(
  overload(
    [](int i) { /* ... */ },
    [](auto x) { /* ... */ }),
  parse(some_input));
```

```
parse(some_input) match {
  int: let i => // ...
  auto: let x => // x is bool or string
};
```

---

3. Initialization? Conversions? First-match? Best-match?

Initialization and conversions are dictated by the rules and principles of bindings as introduced by structured bindings.

The problem of first-match vs best-match is solved by requiring an **exact-match** for alternative types. With exact-match, first-match and best-match become equivalent.

Variable Declaration Approach	This Paper

`parse(some_input) match {`
`  int i => // ...`
`  std::string s => // ...`
`  // missing bool case, but covered by `int`.`
`};`
	`parse(some_input) match {`
	`  int: let i => // ...`
	`  std::string: let s => // ...`
	`  // error: missing bool case`
	`};`

To be precise, the type to the left of the `:` is used to match an alternative **as declared**. This is similar to how `std::get` works. For example:

```
void f(std::variant<const int, std::string> v) {
  v match {
    const int: let i => // `const int` is required here.
    std::string: let s => // ...
  };
}
```

5.5 Discussion on Variant-like Types

We have a few variant-like facilities: `optional`, `expected`, and `variant`. Type-based alternative matching for `std::variant` seems pretty obvious.

```
void f(std::variant<int, std::string> v) {
  v match {
    int: let i => // ...
    std::string: let s => // ...
  };
}
```

The `int` and `string` are the states that a `variant<int, std::string>` can be in, and facilities such as `holds_alternative<int>` and `get<int>` clearly provide type-based access to `variant`.

Of course, in general there's more to it. The `variant` could be in a valueless-by-exception state, or we can have `std::variant<T, T>`. Let's table these for now.

The `? pattern` specifically supports the pointer-like usage pattern, so we can write:

```
void f(int* p) {
  p match {
    ? let i => // ...
    nullptr => // ...
  };
}
```

`optional` and `expected` are "`variant`-like" in that they have "one-of" states. However, their interfaces are not `std::variant`-like at all. They carry much more semantic implications. `optional<T>` behaves more like `T` than `variant<std::nullopt_t, T>` would. `expected<T, E>` behaves more like `T` than `E`, and again, more like `T` than `variant<T, E>` would. Their interfaces are also pointer-like rather than `std::variant`-like.

Given this, it seems natural enough to match on an `optional` like this:

```
void f(std::optional<int> o) {
  o match {
    ? let i => // ...
    std::nullopt => // ...
```

```
  };
}
```

A `std::variant`-like approach would look like this:

```
void f(std::optional<int> o) {
  o match {
    int: let i => // ...
    std::nullopt_t: _ => // ...
  };
}
```

Here, if we changed `std::optional<int>` to say, a `std::optional<double>` the `int: let i` pattern would be ill-formed, whereas the `?` would continue to work. This is consistent with the usage of `optional` today:

```
void f(std::optional<int> o) {
  // no mention of `int` in the below usage.
  if (o) {
    use(*o);
  } else {
    // ...
  }
}
```

Open Question: For exhaustiveness checking purposes, matching with `?` then `_` will always be sufficient. But this means `?` will need to be matched first. For types like `T*` and `unique_ptr`, it should be possible to say matching with `?` and `nullptr` is exhaustive, and `nullptr` can be matched first as well. For `optional` though, the null state is `std::nullopt`. To use `nullptr` for this seems wrong, given that `optional` design explicitly introduced `nullopt` over using `nullptr` itself. The solution in [P2392R2] is to introduce `is void`, but this seems problematic at least for `expected<void, error>` where the question becomes ambiguous.

But `expected<T, E>` gets more tricky. The "no value" case is not just some sentinel type/value, but is some type `E` retrieved by `.error()`.

```
void f(std::expected<int, parse_error> e) {
  e match {
    ? let i => // ...
    // How do we match and access `.error()` ?
  };
}
```

So perhaps a `variant`-like approach would be better here:

```
void f(std::expected<int, parse_error> e) {
  e match {
    int: let i => // ...
    parse_error: let err => // ...
  };
}
```

This seems simple and clean enough. Similar to `variant` however, we can have `expected<T, T>`. Unlike `variant` though, it actually goes out of its way to store a `std::unexpected<T>` as the error state to distinguish the two. It's conceivable to use this unexpected type to support `expected<T, T>`:

```
void f(std::expected<int, int> e) {
  e match {
    int: let i => // ...
    std::unexpected<int>: let err => // distinguish
```

```
  };
}
```

But that would really hinder the by-far more common use cases:

```
void f(std::expected<int, parse_error> e) {
  e match {
    int: let i => // ...
    std::unexpected<parse_error>: let err => // yuck
  };
}
```

It was considered to allow matching `std::expected<T, T>` with `T` and `std::unexpected<T>` while matching `std::expected<T, E>` with `T` and `E`. But it's a bit weird for `std::unexpected<E>` to then not work at all, and also weird for `err` in `std::unexpected<T>: let err` to not be a binding to a `std::unexpected<T>`, but rather a binding to a `T`. A reference to the underlying `std::unexpected<T>` is also not an interface that `std::expected` exposes. Furthermore, this wouldn't solve the problem of `variant<T, T>` in a consistent manner. At best it'd be a special case for `std::expected`.

Ideally, `value` and `error` would be **names** associated to the **types** `T` and `E`, such that they can be used even when `T` and `E` are the same, and are stable even when `T` and `E` changes.

This is essentially how the `Result` type in Rust is defined, as well as many other languages that provide similar functionalities.

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

This is matched like this:

```
match parse(some_input) {
  Ok(v) => // use `v`
  Err(err) => // use `err`
}
```

A few approaches were considered to emulate this "name-based" dispatch.

1. Introduce a parallel `enum class` with the desired names.

```
enum class expected_state { value, error };

template <typename T, typename E>
class expected {
  // ...

  expected_state index() const {
    return has_value() ? expected_state::value : expected_state::error;
  }

  template <expected_state S>
  auto&& get(this auto&& self) {
    if constexpr (S == expected_state::value) {
      return *std::forward<decltype(self)>(self);
    } else if constexpr (S == expected_state::error) {
      return std::forward<decltype(self)>(self).error();
    } else {
```

```
      static_assert(false);
    }
  }
};

template <typename T, typename E>
struct variant_size<expected<T, E>> : std::integral_constant<std::size_t, 2> {};

template <typename T, typename E>
struct variant_alternative<(std::size_t)expected_state::value, expected<T, E>> {
  using type = T;
};

template <typename T, typename E>
struct variant_alternative<(std::size_t)expected_state::error, expected<T, E>> {
  using type = E;
};
```

The usage would need to be something along the lines of:

```
std::expected<int, parse_error> parse(std::string_view sv);

void f() {
  parse(some_input) match {
    using enum std::expected_state;
    value: let v => // ...
    error: let err => // ...
  };
}
```

While the introduction of `std::expected_state` seems a bit odd on first glance, it actually doesn't seem any more odd than other related helper types such as `std::in_place_t`, `std::unexpect_t`, `std::unexpected`, etc.

2. Use the existing tag types

We already have tag types, and they roughly correspond with the various states. For example, `std::expected` uses `std::in_place_t` and `std::unexpect_t`.

```
void f(std::expected<int, parse_error> e) {
  e match {
    std::in_place_t: let v => // ...
    std::unexpect_t: let err => // ...
  };
}
```

The names `std::in_place_t` and `std::unexpect_t` are terrible substitute for `value` and `error`. We'd be better off with just using the types directly, and not fully supporting the `std::expected<T, T>` case.

3. Use the reflection of the accessors as the tags

This idea would be to come up with a new variant-like protocol using reflection. If a type let's say were to advertise its alternatives through `std::vector<std::meta::info>`, and we use those as the tags for dispatching…

```
template <typename T, typename E>
struct expected {
  static consteval std::vector<std::meta::info> alternatives() {
    return { ^value, ^error };
  }
```

```
  constexpr const T& value() const&;
  constexpr const E& error() const& noexcept;
  // other qualified versions...
};
```

With this, perhaps we could pull off something like:

```
void f(std::expected<int, parse_error> e) {
  e match {
    e.value: let v => // ...
    e.error: let err => // ...
  };
}
```

I think this is a very interesting direction for both tuple-like and variant-like protocols, but I haven't been able to flesh out the details. See Reflection-based Tuple-like and Variant-like Customization Points.

In the end, the suggested path for now is:

T*

```
ptr match {
  ? let x => // ...
  nullptr => // ...
};
```

std::optional<T>

```
opt match {
  ? let x => // ...
  std::nullopt => // ...
};
```

std::expected<T, E>

```
e match {
  T: let v => // ...
  E: let err => // ...
};
```

std::variant<T, U>

```
v match {
  T: let t => // ...
  U: let u => // ...
};
```

5.6 Reflection-based Tuple-like and Variant-like Customization Points

"Tuple-like" customization today involves specializing `std::tuple_size`, `std::tuple_element`, and implementing a `get<I>` function. Section 2.3.6 "Cleaner form for structured bindings'"tuple-like" customization" from [P2392R2] has a good summary of the problem.

It also also says:

> 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 };
> ```

and as Bjarne Stroustup points out in [P2411R0]:

> The mapping from an encapsulating type to a set of values used by pattern matching must be simple and declarative. The use of `get<>()` for structured binding is an expert-only mess. Any code-based, as opposed to declarative, mapping will have such problems in use and complicate optimization. We can do much better.

Perhaps this problem can be tackled with reflection.

```
struct EcapsulatedRect {
  static consteval std::vector<std::meta::info> elements() {
    return { ^topLeft, ^width, ^height };
  };

  Point topLeft() const;
  int width() const;
  int height() const;
};
```

The advantage of this is that we can put data members as well as member functions into `elements` as opaque reflections and apply them when needed.

Similarly, it seems feasible for there to be a reflection-based variant-like protocol as well.

```
template <typename... Ts>
struct variant {
  static consteval std::vector<std::meta::info> alternatives() {
    return { ^Ts... };
  };

  // ...
};
```

Note that for tuple-like protocol, even if we were to come up with something better, I think we'll still have to continue supporting the current protocol. There are types written that opted into that protocol that use structured bindings and `std::apply` and other things today.

Variant-like protocol is actually a different story. Unlike tuple-like protocol, The variant helpers such as `std::variant_size`, `std::variant_alternative` are solely used by `std::variant`. `std::visit`, the only thing that might already be using a "variant-like protocol" does not support non-`std::variant`s. It does support types that directly inherit from `std::variant` [P2162R2], but they work by being converted into `std::variant` beforehand.

As such, there's a bigger opportunity for variant-like protocol to not bless the existing set of facilities but to come up with something better.

5.7 More on Static Conditions

This is an elaboration of the discussion from Static Conditions. The question is: how are the requirements and validity of patterns handled? The proposed solution in this paper is for the static conditions to always be checked. For templates, this means the they are checked at instantiation.

Another approach is for some patterns to allow to be invalid if the *subject* is a dependent value. Since in this case, the pattern **can be** valid under some instantiations.

This can be made to work, and would certainly useful. As the default behavior however, it seems like it will likely cause subtle bugs.

Consider an example like this:

```
template <typename Operator>
void f(const Operator& op) {
  op.kind() match {
    '+' => // ...
    '-' => // ...
    '*' => // ...
    "/" => // ...
```

```
    _ => throw UnknownOperator{};
  };
}
```

Let's say `op.kind()` returns a `char`, but we can't be sure of that since `op` is templated. With the approach in this proposal, the typo of `"/"` (should be `'/'`!) will be detected as a compile-time error. In a model where a pattern can be invalid because the *subject* is dependent, this will likely be well-formed, fallthrough to the `_` case, and throw an exception at runtime.

It's true that this function should probably be better constrained using concepts, but the reality is that this kind of code is extremely prevalent today. Note that just using `if`, we would have been provided this safety:

```
template <typename Operator>
void f(const Operator& op) {
  if (op.kind() == '+') {
    // ...
  } else if (op.kind() == '-') {
    // ...
  } else if (op.kind() == '*') {
    // ...
  } else if (op.kind() == "/") { // error: comparison between pointer and integer
    // ...
  } else {
    throw UnknownOperator{};
  }
}
```

Testing the Static Conditions with `match requires` is described as a future extension where users can explicitly opt in to relax this requirement on *static conditions*.

6 Future Extension Exploration

The following lists patterns and features excluded from this paper, but could still be useful future extensions.

6.1 Static Type Checking with Constraint Pattern

A constraint pattern could be used to perform static type checks.

type-constraint

The static condition of a constraint pattern would be that `decltype(`*`subject`*`)` satisfies the *type-constraint*.

For example,

```
void f(auto p) {
  p match {
    [std::convertible_to<int>, 0] => // statically check that first elem converts to int.
    // ...
  };
}
```

If used with structured bindings, this becomes very similar to the static type checking proposed in [P0480R1].

```
auto [std::same_as<std::string> a, std::same_as<int> b] = f();
```

The syntax changes would be:

```
  match-pattern
      // ...
+     type-constraint
```

```
  binding-pattern
      // ...
+     type-constraint identifier
```

6.2 Testing the Static Conditions with `match requires`

The sections Static Conditions and More on Static Conditions described what static conditions are. They also described why by default, `match` and `match constexpr` should both always check the static conditions.

`match requires` (or some other spelling) would offer a way to test the static conditions instead.

`match requires`

```
void f(auto x) {
  x match requires { // not proposed
    0 => // ...
    "hello" => // ...
    _ => // ...
  };
}

f("hello"s); // fine, skips 0
```

`if constexpr (requires { ... })`

```
void f(auto x) {
  if constexpr (requires { x == 0; }) {
    if (x == 0) {
      // ...
      goto done;
    }
  }
  if constexpr (requires { x == "hello"; }) {
    if (x == "hello") {
      // ...
      goto done;
    }
  }
  // ...
  done:;
}
```

Using the constraint pattern from Static Type Checking with Constraint Pattern, we can perform a sequence of static type tests.

```
void f(auto x) {
  x match requires { // not proposed
    std::same_as<bool> => // ...
    std::integral => // ...
    std::same_as<std::string_view> => // ...
    std::range => // ...
  };
}
```

Using the `let` pattern, we can even bind names to each of these:

```
void f(auto x) {
  x match requires { // not proposed
    std::same_as<bool> let b => // ...
    std::integral let i => // ...
    std::same_as<std::string_view> let sv => // ...
    std::range let r => // ...
```

```
  };
}
```

Another example with structured bindings patterns:

```
void f(auto x) {
  x match requires { // not proposed
    let [x] => // ...
    let [x, y] => // ...
    let [x, y, z] => // ...
    let [...xs] => // ...
  };
}
```

Rather than the static condition (matching size requirement) of structured bindings pattern being checked, they are `if constexpr` tested instead.

`match`	`match constexpr`

`if (condition) {`
`  // ...`
`}`
	`// match constexpr`
	`if constexpr (condition) {`
	`  // ...`
	`}`

`match requires` (not proposed)	`match requires constexpr` (not proposed)

`if constexpr (requires { condition ; }) {`
`  if (condition) {`
`  // ...`
`  }`
`}`
	`if constexpr (requires { condition ; }) {`
	`  if constexpr (condition) {`
	`  // ...`
	`  }`
	`}`

6.3 Pattern Combinators

Pattern combinators provide a way to succinctly combine multiple patterns.

```
or ( pattern-0 , … , pattern-N )
and ( pattern-0 , … , pattern-N )
```

Example:

This Paper	**With `or`:**

`direction match {`
`  'N' => f();`
`  'E' => g();`
`  'S' => f();`
`  'W' => g();`
`};`
	`direction match {`
	`  or('N', 'S') => f();`
	`  or('E', 'W') => g();`
	`};`

This Paper	With `or`:

`e match {`
`  A: let a => f();`
`  B: let b => f();`
`  C: let c => g();`
`};`
	`e match {`
	`  or(`
	`  A: let a,`
	`  B: let b`
	`  ) => f();`
	`  C: let c => g();`
	`};`

6.4 Designator Support for Structured Bindings

This would extend structured bindings to allow designators (i.e. `.field_name`) to match on that field.

```
  match-pattern
      // ...
+     [ designator-0 : pattern-0 , … , designator-N : pattern-N ]
```

```
  binding-pattern
      // ...
+     [ designator-0 : binding-pattern-0 , … designator-N : binding-pattern-N ]
```

Example:

```
  return scope match {
    GlobalScope: _ => Cxx::Scope::global_();
    NamespaceScope: [.fact: let f] => Cxx::Scope::namespace_(f);
    ClassScope: [.fact: let f] => Cxx::Scope::recordWithAccess(f, access(acs));
    LocalScope: [.fact: let f] => Cxx::Scope::local(f);
//                ^^^^^^^^^^^^^^
  };
```

6.5 Value-based discriminators

This would extend the alternative pattern to allow value-based discriminators.

```
  discriminator:
      type-id
      type-constraint
+     constant-expression
```

From Discussion on Variant-like Types, the example of `enum` values `value` and `error`:

```
enum class expected_state { value, error };

std::expected<int, parse_error> parse(std::string_view sv);

void f() {
  parse(some_input) match {
    using enum std::expected_state;
    value: let v => // ...
    error: let err => // ...
  };
}
```

variant<T, T>	expected<T, T>

void f(variant<int, int> v) {
  v match {
  0: let first => // ...
  1: let second => // ...
  };
}
	void f(expected<int, int> e) {
	  e match {
	  0: let value => // ...
	  1: let error => // ...
	  };
	}

7 Acknowledgements

Thank you to all of the following folks

— Zach Laine, Barry Revzin, and Bruno Cardoso Lopes for the encouragement and long discussions over much of what is proposed and discussed in this paper.
— David Sankel, Sergei Murzin, Bruno Cardoso Lopes, Dan Sarginson, and Bjarne Stroustrup for our prior work on [P1371R3].
— Herb Sutter for valuable feedback and the work done in P2392R2.
— David Sankel, Sergei Murzin, Alex Chow, Yedidya Feldblum, and Jason Lucas for recent discussions.
— Everyone else who have had discussions about pattern matching and provided feedback in prior meetings and telecons!

8 References

[P0144R2] Herb Sutter. 2016-03-16. Structured Bindings.
https://wg21.link/p0144r2

[P0480R1] Ville Voutilainen. 2018-10-08. Structured bindings with explicit types.
https://wg21.link/p0480r1

[P1110R0] Jeffrey Yasskin, JF Bastien. 2018-06-07. A placeholder with no name.
https://wg21.link/p1110r0

[P1371R3] Michael Park, Bruno Cardoso Lopes, Sergei Murzin, David Sankel, Dan Sarginson, Bjarne Stroustrup. 2020-09-15. Pattern Matching.
https://wg21.link/p1371r3

[P1469R0] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2019-01-21. Disallow _ Usage in C++20 for Pattern Matching in C++23.
https://wg21.link/p1469r0

[P2162R2] Barry Revzin. 2021-02-18. Inheriting from std::variant (resolving LWG3052).
https://wg21.link/p2162r2

[P2169R4] Corentin Jabot, Michael Park. 2023-06-16. A Nice Placeholder With No Name.
https://wg21.link/p2169r4

[P2392R2] Herb Sutter. 2022-09-25. Pattern matching using is and as.
https://wg21.link/p2392r2

[P2411R0] Bjarne Stroustrup. 2021-07-22. Thoughts on pattern matching.
https://wg21.link/p2411r0

[P2688R0] Michael Park. 2022-10-16. Pattern Matching Discussion for Kona 2022.
https://wg21.link/p2688r0

[P2806R2] Barry Revzin, Bruno Cardoso Lopez, Zach Laine, Michael Park. 2023-11-16. do expressions.
https://wg21.link/p2806r2

[P2927R0] Gor Nishanov. 2023-10-15. Observing exceptions stored in exception_ptr.
https://wg21.link/p2927r0