P2688R1: Pattern Matching

`match` Expression

Michael Park, 2024-03-21

Overview

- Brief Recent History
- Goals for P2688R1
- Anatomy of `match`
- Overview of Patterns
- Pattern Composition
- Static and Dynamic Conditions
- More Design Discussions
- Other Languages: Comparison of various components with 8 other languages

Brief Recent History

- P1371: Pattern Matching R0, R1 (2019), R2, R3 (2020)
- EWG Telecon 2021
  - Poll: Annotate id-pattern with auto

SF	F	N	A	SA
0	3	4	3	8

```
inspect (e) { // like this:
  auto&& x => // binds
  x => // matches
}
```

  - Poll: Annotate id-pattern with let

SF	F	N	A	SA
3	6	6	2	2

```
inspect (e) { // like this:
  let x => // binds
  x => // matches
}
```

  - Poll: Keep the current p1371r3 proposal with case

SF	F	N	A	SA
2	6	8	0	3

```
inspect (e) { // like this:
  x => // binds
  case x => // matches
}
```

Brief Recent History

- P2392: Pattern matching using `is` and `as` R0, R1 (2021), R2 (2022)
- P2688R0: Pattern Matching Discussion for Kona 2022 (2022)
- P2688R1: Pattern Matching: `match` Expression (2024)
- Kona 2022
  - EWG Prefers composition over chaining in pattern matching syntax.

SF	F	N	A	SA
13	9	2	1	0

```
color match {
  Rgb: let [r, g, b] => …
};

command match {
  ChangeColor: [Rgb: let [r, g, b]] => …
};


inspect (color) {
  [r, g, b] as Rgb => ...
}

inspect (command) {
  [[r, g, b]] as ChangeColor as [Rgb]
    => …
}
```

Goals for P2688R1

- Narrow down the scope. Fewer patterns, fewer functionalities
  - P1371 provides too many patterns (my opinion)
  - P2392 stuffs too much semantics into small syntax (my opinion)
- Consider the one-way door options
  - Focus on choices we'll be stuck with.
    e.g. Patterns have distinct syntax from expressions. `*x`, `1 | 2` are just expressions
  - Provide a framework for pattern definitions.
    e.g. Static/dynamic conditions
  - Not get too hung up on functionalities we can add later

Goals for P2688R1

Addressing Concerns

- "*We should be able to perform pattern matching outside of* `inspect`"
  - One of the main concerns of P2392
- "*We shouldn't bifurcate expressions like this*"
  - P1371R3 required `case` on identifiers but not literals
- "*Declaration of new names should have an introducer like most other places in the language.*"

Anatomy of `match`

High-Level Structure

expr `match` {
  pattern => *expr-or-braced-init-list* ;
  pattern => `break` ;
  pattern => `continue` ;
  pattern => `return` *expr-or-braced-init-list*$_{opt}$;
  pattern => `co_return` *expr-or-braced-init-list*$_{opt}$;
  …
}

expr `match` *pattern*

Anatomy of `match`

Guards

```
expr match {
  pattern if condition => expr-or-braced-init-list;
  pattern if condition => break ;
  pattern if condition => continue ;
  pattern if condition => return expr-or-braced-init-list_opt ;
  pattern if condition => co_return expr-or-braced-init-list_opt ;
  ...
}
```

```
expr match pattern if condition
```

Anatomy of `match`

Matching multiple values

```
{ expr... } match {
  pattern => expr-or-braced-init-list;
  pattern => break ;
  pattern => continue ;
  pattern => return expr-or-braced-init-list_opt ;
  pattern => co_return expr-or-braced-init-list_opt ;
  ...
}
```

```
{ expr... } match pattern
```

Anatomy of `match`

Specifying the return type

```
expr match -> return-type {
   pattern => expr-or-braced-init-list;
   pattern => break ;
   pattern => continue ;
   pattern => return expr-or-braced-init-list_opt ;
   pattern => co_return expr-or-braced-init-list_opt ;
   …
}
```

Anatomy of `match`

Matching constexpr

```
expr match constexpr {
   pattern => expr-or-braced-init-list;
   pattern => break ;
   pattern => continue ;
   pattern => return expr-or-braced-init-listopt ;
   pattern => co_return expr-or-braced-init-listopt ;
   …
}
```

Anatomy of `match`

Putting them together

```
expr match {
  pattern₁ => expression₁ ;
  pattern₂ if condition => expression₂ ;
  …
}

{ expr… } match {
  pattern₁ => expression₁ ;
  …
}

expr match constexpr -> return-type {
  pattern₁ => expression₁ ;
  …
}
```

```
bool b = expr match pattern;
f(expr match pattern if condition);

{ expr… } match pattern;

if (expr match pattern) {
  // names injected here
}

while (expr match pattern) {
  // names injected here
}
```

Anatomy of `match`

Still a wrinkle

```
void f(int a, int b) {
  a match { // okay
    0 => std::print("zero");
    1 => std::print("one");
    _ => std::print("don't care");
  };

  int x = { a, b } match { // okay
    [0, 0] => 1;
    _ => 2;
  };
}
```

```
void fizzbuzz() {
  for (int i = 1; i <= 100; ++i) {
    {i%3, i%5} match {
      [0, 0] => std::print("fizzbuzz");
      [0, _] => std::print("fizz");
      [_, 0] => std::print("buzz");
      [_, _] => std::print("{}\n", i);
    };
  }
}
```

Anatomy of `match`

Still a wrinkle

```
void f(int a, int b) {
  a match { // okay
    0 => std::print("zero");
    1 => std::print("one");
    _ => std::print("don't care");
  };

  int x = { a, b } match { // okay
    [0, 0] => 1;
    _ => 2;
  };
}
```

```
void fizzbuzz() {
  for (int i = 1; i <= 100; ++i) {
    {i%3, i%5} match {
      [0, 0] => std::print("fizzbuzz");
      [0, _] => std::print("fizz");
      [_, 0] => std::print("buzz");
      [_, _] => std::print("{}\n", i);
    };
  }
}
```

This is interpreted as a new block scope 😥

Anatomy of match

Still a wrinkle

```
void f(int a, int b) {
  a match { // okay
    0 => std::print("zero");
    1 => std::print("one");
    _ => std::print("don't care");
  };

  int x = { a, b } match { // okay
    [0, 0] => 1;
    _ => 2;
  };
}
```

```
void fizzbuzz() {
  for (int i = 1; i <= 100; ++i) {
    ({i%3, i%5} match {
      [0, 0] => std::print("fizzbuzz");
      [0, _] => std::print("fizz");
      [_, 0] => std::print("buzz");
      [_, _] => std::print("{}\n", i);
    });
  }
}
```

Overview of Patterns

- **Wildcard Pattern** `_`
  - Ignore values
- **Parenthesized Pattern** `(` *pattern* `)`
  - Grouping
  - Useful since not all patterns are delimited
- **Let Pattern** `let` *binding-pattern*
  match-pattern `let` *binding-pattern*
  - Introduce bindings
- **Constant Pattern** *constant-expression*
  - `enum` values
  - `constexpr` variables
- **Optional Pattern** `?` *pattern*
  - pointers
  - `std::unique_ptr`, `std::shared_ptr`
  - `std::optional`
- **Structured Bindings Pattern** `[` *pattern...* `]`
  - arrays, `std::array`, `std::pair`, `std::tuple`
- **Alternative Pattern** *type-id* `:` *pattern*
  type-constraint `:` *pattern*
  - `std::variant`, `std::expected`
  - `std::any`, `std::exception_ptr`
  - polymorphic types

Overview of Patterns

Matching Values

```
int x = 1;

switch (x) {
  case 1: …
  case 2: …
  default: …
}
```

```
x match {
  1 => …
  2 => …
  _ => …
};
```

Overview of Patterns

Matching Values

```
int x = 1;

switch (x) {
  case 1: …
  case 2: …
  default: …
}
```

Constant Pattern *constant-expression*

```
x match {
  1 => …
  2 => …
  _ => …
};
```

Overview of Patterns

Matching Values

```
int x = 1;

switch (x) {
  case 1: …
  case 2: …
  default: …
}
```

Wildcard Pattern _

```
x match {
  1 => …
  2 => …
  _ => …
};
```

Overview of Patterns

Matching Strings

```
std::string s = "hello";

if (s == "hello") {
  …
} else if (s == "world") {
  …
} else {
  …
}
```

```
s match {
  "hello" => …
  "world" => …
  _ => …
};
```

Overview of Patterns

Matching Strings

```
std::string s = "hello";

if (s == "hello") {
  …
} else if (s == "world") {
  …
} else {
  …
}
```

Constant Pattern *constant-expression*

```
s match {
  "hello" => …
  "world" => …
  _ => …
};
```

Overview of Patterns

Matching Enumerations

```
enum Color { Red, Green, Blue };

Color get_color();

switch (get_color()) {
  case Red: …
  case Green: …
  case Blue: …
}
```

```
get_color() match {
  Red => …
  Green => …
  Blue => …
};
```

Overview of Patterns

Matching Enumerations

```
enum Color { Red, Green, Blue };

Color get_color();

switch (get_color()) {
  case Red: …
  case Green: …
  case Blue: …
}
```

Constant Pattern *constant-expression*

```
get_color() match {
  Red => …
  Green => …
  Blue => …
};
```

Overview of Patterns

Matching Pointer-Like: Pointers, Smart Pointers, Optionals

```
optional<int> get_optional();

if (auto opt = get_optional()) {
  f(*opt); // or opt.value()
} else {
  …
}
```

```
get_optional() match {
  ? let v => f(v);
  _ => …
};
```

```
void g(const optional<int>& opt) {
  if (opt) {
    f(*opt);
  }
}
```

```
void g(const optional<int>& opt) {
  if (opt match ? let v) {
    f(v);
  }
}
```

Overview of Patterns

Matching Pointer-Like: Pointers, Smart Pointers, Optionals

```
optional<int> get_optional();

if (auto opt = get_optional()) {
  f(*opt); // or opt.value()
} else {
  ...
}

void g(const optional<int>& opt) {
  if (opt) {
    f(*opt);
  }
}
```

Optional Pattern *? pattern*

```
get_optional() match {
  ? let v => f(v);
  _ => ...
};

void g(const optional<int>& opt) {
  if (opt match ? let v) {
    f(v);
  }
}
```

Overview of Patterns

Matching Pointer-Like: Pointers, Smart Pointers, Optionals

```
optional<int> get_optional();

if (auto opt = get_optional()) {
  f(*opt); // or opt.value()
} else {
  …
}

void g(const optional<int>& opt) {
  if (opt) {
    f(*opt);
  }
}
```

Let Pattern `let` *binding-pattern*

```
get_optional() match {
  ? let v => f(v);
  _ => …
};

void g(const optional<int>& opt) {
  if (opt match ? let v) {
    f(v);
  }
}
```

Overview of Patterns

Matching Pointer-Like: Pointers, Smart Pointers, Optionals

```
optional<int> get_optional();

if (auto opt = get_optional()) {
  f(*opt); // or opt.value()
} else {
  …
}
```

```
get_optional() match {
  ? let v => f(v);
  _ => …
};
```

```
void g(const optional<int>& opt) {
  if (opt) {
    f(*opt);
  }
}
```

```
void g(const optional<int>& opt) {
  if (opt match ? let v) {
    f(v);
  }
}
```

Overview of Patterns

Matching Pointer-Like: Pointers, Smart Pointers, Optionals

```
optional<int> get_optional();

if (auto opt = get_optional()) {
  f(*opt); // or opt.value()
} else {
  …
}

void g(const optional<int>& opt) {
  if (opt) {
    f(*opt);
  }
}
```

```
get_optional() match {
  ? let v => f(v);
  _ => …
};

void g(const optional<int>& opt) {
  if (opt match ? let v) {
    f(v);
  }
}
```

Overview of Patterns

Matching Pointer-Like: Pointers, Smart Pointers, Optionals

```
optional<int> get_optional();
```

```
if (auto opt = get_optional()) {
  f(*opt); // or opt.value()
} else {
  …
}
```

```
get_optional() match {
  ? let v => f(v);
  _ => …
};
```

```
void g(const optional<int>& opt) {
  if (opt) {
    f(*opt);
  }
}
```

```
void g(const optional<int>& opt) {
  if (opt match ? let v) {
    f(v);
  }
}
```

Overview of Patterns

Matching Pointer-Like: Pointers, Smart Pointers, Optionals

```
optional<int> get_optional();

if (auto opt = get_optional()) {
  f(*opt); // or opt.value()
} else {
  …
}

void g(const optional<int>& opt) {
  if (opt) {
    f(*opt);
  }
}
```

Parenthesized Pattern (*pattern*)

```
get_optional() match {
  ? let v => f(v);
  _ => …
};

void g(const optional<int>& opt) {
  if (opt match (? let v)) {
    f(v);
  }
}
```

Overview of Patterns

Matching an optional enumeration

```
enum Color { Red, Green, Blue };

optional<Color> get_opt_color();

if (auto opt_color = get_opt_color()) {
  switch (*opt_color) {
    case Red: stop();
    default: handle(*opt_color);
  }
} else {
  …
}
```

Optional Pattern ? *pattern*

```
get_opt_color() match {
  ? Red => stop();
  ? let color => handle(color);
  _ => …
};
```

Overview of Patterns

Matching an optional enumeration

```
enum Color { Red, Green, Blue };

optional<Color> get_opt_color();

if (auto opt_color = get_opt_color()) {
  switch (*opt_color) {
    case Red: stop();
    default: handle(*opt_color);
  }
} else {
  …
}
```

Optional Pattern ? *pattern*

```
get_opt_color() match {
  ? Red => stop();
  ? let color => handle(color);
  _ => …
};
```

Overview of Patterns

Matching an optional enumeration

```
enum Color { Red, Green, Blue };

optional<Color> get_opt_color();

if (auto opt_color = get_opt_color()) {
  switch (*opt_color) {
    case Red: stop();
    default: handle(*opt_color);
  }
} else {
  …
}
```

Constant Pattern *constant-expression*

```
get_opt_color() match {
  ? Red => stop();
  ? let color => handle(color);
  _ => …
};
```

Overview of Patterns

Matching an optional enumeration

```
enum Color { Red, Green, Blue };

optional<Color> get_opt_color();

if (auto opt_color = get_opt_color()) {
  switch (*opt_color) {
    case Red: stop();
    default: handle(*opt_color);
  }
} else {
  …
}
```

Let Pattern `let` *identifier*

```
get_color() match {
  ? Red => stop();
  ? let color => handle(color);
  _ => …
};
```

Overview of Patterns

Let Pattern

match-pattern `let` *binding-pattern*

`let` *binding-pattern*

```
expr match {
  [0, 0] let whole => …
  let [p, q] if p == q => …
  let [...xs] if pred(xs...) => …
  let x => …
};
```

Overview of Patterns

Matching on Tuple-Like

```
struct Point { int x, int y };

Point p = { 101, 202 };

auto&& [x, y] = p;
if (x == 0 && y == 0) { … }
else if (x == 0) { … }
else if (y == 0) { … }
else { … }
```

```
p match {
  [0, 0] => …
  [0, let y] => …
  [let x, 0] => …
  let [x, y] => …
};
```

Overview of Patterns

Matching on Tuple-Like

```
struct Point { int x, int y };

Point p = { 101, 202 };

auto&& [x, y] = p;
if (x == 0 && y == 0) { … }
else if (x == 0) { … }
else if (y == 0) { … }
else { … }
```

Structured Bindings Pattern [*pattern...*]

```
p match {
  [0, 0] => …
  [0, let y] => …
  [let x, 0] => …
  let [x, y] => …
};
```

Overview of Patterns

Matching on Tuple-Like

```
struct Point { int x, int y };

Point p = { 101, 202 };

auto&& [x, y] = p;
if (x == 0 && y == 0) { … }
else if (x == 0) { … }
else if (y == 0) { … }
else { … }
```

Let Pattern	`let` *identifier*
	`let [` *identifier…* `]`

```
p match {
  [0, 0] => …
  [0, let y] => …
  [let x, 0] => …
  let [x, y] => …
};
```

Overview of Patterns

Matching on Variant-Like

```
variant<int, string> v = 42;

struct visitor {
  void operator()(int i) const {
    …
  }
  void operator()(const string& s) const {
    …
  }
};
std::visit(visitor{}, v);
```

```
v match {
  int: let i => …
  string: let s => …
};
```

Overview of Patterns

Matching on Variant-Like

```
variant<int, string> v = 42;

struct visitor {
  void operator()(int i) const {
    …
  }
  void operator()(const string& s) const {
    …
  }
};
std::visit(visitor{}, v);
```

Alternative Pattern *type-id* : *pattern*

```
v match {
  int: let i => …
  string: let s => …
};
```

Overview of Patterns

Matching on Variant-Like

```
variant<int, string> v = 42;

struct visitor {
  void operator()(int i) const {
    …
  }
  void operator()(const string& s) const {
    …
  }
};
std::visit(visitor{}, v);
```

Let Pattern `let` *identifier*

```
v match {
  int: let i => …
  string: let s => …
};
```

Pattern Composition

Matching Tuple-Likes and Variant-Likes

```
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 text; };
struct ChangeColor { Color color; };

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

Pattern Composition

Matching Tuple-Likes and Variant-Likes

```
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.text;
  }
  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.color);
  }
};
std::visit(CommandVisitor{}, cmd);
```

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

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

Pattern Composition

Matching Tuple-Likes and Variant-Likes

```
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.text;
  }
  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.color);
  }
};
std::visit(CommandVisitor{}, cmd);
```

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

cmd match {
  Quit: _ => ...
  Move: [0, 0] => // going to origin
  Move: [0, let y] => // going to y-axis
  Move: [let x, 0] => // going to x-axis
  Move: let [x, y] => ...
  Write: [quit_message] => // did you mean to quit instead?
  Write: let [text] => // write text
  ChangeColor: [Rgb: let [r, g, b]] => ...
  ChangeColor: [Hsv: let [h, s, v]] => ...
};
```

Static and Dynamic Conditions

```
void f(int x) {
  x match {
    0 => // well-formed
    _ => …
  };
}
```

```
void g(string x) {
  x match {
    0 => // ill-formed
    _ => …
  };
}
```

Static and Dynamic Conditions

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

f("hello"s); // what about here?
```

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

Static and Dynamic Conditions

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

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

```
f("hello"s); // P2688R1: ill-formed at instantiation
```

Static and Dynamic Conditions

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

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

```
f("hello"s); // P2688R1: ill-formed at instantiation
             // P2392R2: well-formed and no-match
             // Carbon: well-formed and no-match
```

Static and Dynamic Conditions

```
void f(const auto& op) {
  op.kind() match {
    '+' => …
    '-' => …
    '*' => …
    "/" => …
    _ => throw UnknownOperator{};
  };
}
```

Static and Dynamic Conditions

```
void f(const auto& op) {
  op.kind() match {
    '+'  => ...
    '-'  => ...
    '*'  => ...
    "/"  => // this is just a no-match if static conditions are not enforced.
    _ => throw UnknownOperator{};
  };
}
```

Static and Dynamic Conditions

```
void f(const auto& 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{};
  }
}
```

Static and Dynamic Conditions

Every pattern has static and dynamic conditions

Example: Constant Pattern `0`

Static Condition: `expr == 0` is valid. i.e., `requires { expr == 0; }` is true
Dynamic Condition: `expr == 0` is true

Example: Structured Bindings Pattern `[0, 0]`

Static Condition: `auto&& [x, y] = expr; x == 0` and `y == 0` are all valid
Dynamic Condition: `auto&& [x, y] = expr; (x == 0 && y == 0)` is true

Static and Dynamic Conditions

`match` **enforces** static conditions (think `static_assert`), and
tests dynamic conditions at **runtime** (think `if`)

`match constexpr` **tests** dynamic conditions at **compile-time** (think `if constexpr`).

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

Static and Dynamic Conditions

`match requires` (**not proposed**) **tests** static conditions at **compile-time** (think `if constexpr`).

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

Variable Declarations for Alternative Pattern

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

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

Variable Declarations for Alternative Pattern

Variable or Bindings?

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

parse(some_input) match {
  int i => …
  bool b => …
  string s => …
};
```

`Q1:` What **are** `i`, `b`, and `s`?

Variable Declarations for Alternative Pattern

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

parse(some_input) match {
  int i => …
  bool b => …
  const string& s => …
};
```

Variable Declarations for Alternative Pattern

`auto` ambiguity for variant

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

```
parse(some_input) match {
  int i => ...
  auto x => ...
};
```

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

Q2: What is x? Is it a `variant`? or is it `bool` or `string`?
If it's a `bool` or `string`, what does this match? `[0, auto x]`

Variable Declarations for Alternative Pattern

Initialization?

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

parse(some_input) match {
  int i => …
  bool b => …
  const string& s => …
};
```

Q3: Which initialization? Direct? copy? list? copy-list?

Variable Declarations for Alternative Pattern

Conversions?

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

parse(some_input) match {
  int i => …
  bool b => …
  const string& s => …
};
```

`Q4:` Are conversions allowed? No form of initialization disallows **all** conversions

Variable Declarations for Alternative Pattern

First-Match? or Best-Match?

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

parse(some_input) match {
  int i => …
  bool b => …
  const string& s => …
};
```

Q5: First-match? or is this overload resolution?

If conversions are allowed, first-match is problematic.

Variable Declarations for Alternative Pattern

```
std::any parse(string_view);

parse(some_input) match {
  int i => ...
  bool b => ...
  const string& s => ...
};
```

`Q5:` First-match? or is this overload resolution?
This would need to be first-match.

Variable Declarations for Alternative Pattern

```
std::any parse(string_view);

parse(some_input) match {
  int i => ...
  bool b => ...
  const string& s => ...
  auto x => // not ambiguous. generically getting any's value isn't possible.
};
```

Variable Declarations for Alternative Pattern

```
std::any parse(string_view);
```

```
parse(some_input) match {
  int i => …
  bool b => …
  const string& s => …
  auto x => …
};
```

```
parse(some_input) match {
  int: let i => …
  bool: let b => …
  string: let s => …
  let x => …
};
```

Variable Declarations for Alternative Pattern

```
const Shape& parse(string_view);

parse(some_input) match {
  const Circle& c => ...
  const Rectangle& r => ...
  const Triangle& t => ...
  const auto& x => // not ambiguous, same reason.
};
```

Variable Declarations for Alternative Pattern

```
const Shape& parse(string_view);
```

```
parse(some_input) match {
  const Circle& c => …
  const Rectangle& r => …
  const Triangle& t => …
  const auto& x => …
};
```

```
parse(some_input) match {
  Circle: let c => …
  Rectangle: let r => …
  Triangle: let t => …
  let x => …
};
```

Variable Declarations for Alternative Pattern

Pattern Composition

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

```
parse(some_input) match {
  int i => …
  bool b => …
  const string& s => …
};
```

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

Variable Declarations for Alternative Pattern

Pattern Composition

```
inline constexpr int batch_size = 32;
inline constexpr std::string_view start_message = "let's go";
variant<int, bool, string> parse(string_view sv);
```

```
parse(some_input) match {
  int i => …
  bool b => …
  const string& s => …
};
```

```
parse(some_input) match {
  int: 42 => …
  int: batch_size => …
  int: let i => …
  bool: let b => …
  string: start_message => …
  string: "hello" => …
  string: let s => …
};
```

Labels for Variant-like Things

```
void f(std::vector<int>* x) {
  if (x) {
    g(*x);
  } else {
    …
  }
}
```

```
void f(std::vector<int>* x) {
  x match {
    ? let v => g(v);
    _ => …
  }
}
```

Labels for Variant-like Things

```
void f(std::vector<string>* x) {
  if (x) {
    g(*x);
  } else {
    …
  }
}
```

```
void f(std::vector<string>* x) {
  x match {
    ? let v => g(v);
    _ => …
  }
}
```

Labels for Variant-like Things

```
void f(const optional<vector<string>>& x) {
  if (x) {
    g(*x);
  } else {
    …
  }
}
```

```
void f(const optional<vector<string>>& x) {
  x match {
    ? let v => g(v);
    _ => …
  }
}
```

```
void f(const optional<vector<string>>& x) {
  x match {
    std::nullopt => …
    ? let v => g(v);
  }
}
```

Labels for Variant-like Things

std::expected

```
expected<int, my_error> compute();
```

```
if (auto exp = compute()) {
  g(*exp);
} else {
  handle(exp.error());
}
```

```
compute() match {
  ? let v => g(v);
  let exp => handle(exp.error());
};
```

Labels for Variant-like Things

std::expected

```
expected<int, my_error> compute();
```

```
if (auto exp = compute()) {
  g(*exp);
} else {
  handle(exp.error());
}
```

```
compute() match {
  int: let v => g(v);
  my_error: let err => handle(err);
};
```

Labels for Variant-like Things

std::expected

```
expected<int, int> compute();
```

```
if (auto exp = compute()) {
  g(*exp);
} else {
  handle(exp.error());
}
```

```
compute() match {
  int: let v => g(v);
  int: let err => handle(err);
};
```

This of course doesn't work

Labels for Variant-like Things

`std::expected`

```
expected<std::vector<int>, my_error> compute();
```

```
if (auto exp = compute()) {
  g(*exp);
} else {
  handle(exp.error());
}
```

```
compute() match {
  std::vector<int>: let v => g(v);
  my_error: let err => handle(err);
};
```

Labels for Variant-like Things

std::expected

```
expected<std::vector<int>, my_error> compute();
```

```
if (auto exp = compute()) {
  g(*exp);
} else {
  handle(exp.error());
}
```

```
compute() match {
  ? let v => g(v);
  my_error: let err => handle(err);
};
```

Labels for Variant-like Things

std::expected

```
expected<std::vector<int>, my_error> compute();
```

```
if (auto exp = compute()) {
  g(*exp);
} else {
  handle(exp.error());
}
```

```
compute() match {
  .value: let v => g(v);
  .error: let err => handle(err);
};
```

Ideally something like this

Labels for Variant-like Things

`std::expected`

```
expected<std::vector<int>, my_error> compute();
```

```
if (auto exp = compute()) {
  g(*exp);
} else {
  handle(exp.error());
}
```

```
compute() match {
  std::vector<int>: let v => g(v);
  my_error: let err => handle(err);
};
```

For now, this

Customization Points

- Tuple-protocol (`tuple_size`, `tuple_element`, `get`) is already established and is being used.
  e.g. Structured bindings, `std::apply`

- Variant-protocol is actually not really a thing. Even if you specialize your type for `variant_size`, `variant_alternative`, `get`, it isn't used by anything. Not even `std::visit`.

  `std::visit` only supports types that inherit from `std::variant`.
  P2162R2 "Inheriting from `std::variant`"

- **Conclusion**: While improving tuple-protocol is still desired, we can't just drop support for the current mechanism. There's a bigger opportunity for variant-protocol to introduce a better mechanism, rather than blessing `variant_size`, `variant_alternative`, `get` as variant-protocol.

Reflection-based Customization Points

Better tuple-protocol

Bjarne Stroustrup points out in P2411R0: "Thoughts on pattern matching"

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.*

Code example from P2392R2:

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

Reflection-based Customization Points

Better tuple-protocol?

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

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

Reflection-based Customization Points

Better tuple-protocol?

```
struct EncapsulatedRect {
  ...

  // shown only for concrete discussion
  template <std::size_t I>
  decltype(auto) get(this auto&& self) {
    return std::forward<decltype(self)>(self).[:elements()[I]:]();
  }
};

// shown only for concrete discussion
namespace std {
  template <>
  struct tuple_size<S>
    : std::integral_constant<std::size_t, S::elements().size()> {};

  template <std::size_t I>
  struct tuple_element<I, S> {
    using type = decltype(std::declval<S>().[:S::elements()[I]:]());
  };
}
```

Reflection-based Customization Points

Better variant-protocol?

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

  …
};

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

  …
};
```

Other Languages

High-Level Structure

Haskell
```
case expr of
  pattern -> expr
```

Rust
```
match expr {
  pattern => expr,
}
```

Scala
```
expr match {
  case pattern => expr
}
```

OCaml
```
match expr with
  | pattern -> expr
```

Python
```
match expr:
  case pattern: stmt
```

Swift
```
// statement
switch expr {
  case pattern: stmt;
}
```
```
// expression
switch expr {
  case pattern: expr;
}
```

Java
```
switch (expr) {
  case pattern: stmt; break;
}
```
```
switch (expr) {
  case pattern -> expr;
}
```

C#
```
switch (expr) {
  case pattern: stmt; break;
}
```
```
expr switch {
  pattern => expr,
}
```

C++
```
switch (expr) {
  case constant: stmt; break;
}
```
```
expr match {   // P2688R1
  pattern => expr;
}
```

Other Languages

Cases

Haskell		*pattern*	|	*condition*	->	...
Rust		*pattern*	if	*condition*	=>	...
Scala	case	*pattern*	if	*condition*	=>	...
OCaml	|	*pattern*	when	*condition*	->	...
Python	case	*pattern*	if	*condition*	:	...
Swift	case	*pattern*	where	*condition*	:	...
Java	case	*pattern*	when	*condition*	->	...
C#		*pattern*	when	*condition*	=>	...
P2688R1		*pattern*	if	*condition*	=>	...

Other Languages

Cases: Introducers

Haskell		*pattern*	|	*condition*	->	...
Rust		*pattern*	if	*condition*	=>	...
Scala	case	*pattern*	if	*condition*	=>	...
OCaml	|	*pattern*	when	*condition*	->	...
Python	case	*pattern*	if	*condition*	:	...
Swift	case	*pattern*	where	*condition*	:	...
Java	case	*pattern*	when	*condition*	->	...
C#		*pattern*	when	*condition*	=>	...
P2688R1		*pattern*	if	*condition*	=>	...

Other Languages

Cases: Guards

Haskell		*pattern*	|	*condition*	->	...
Rust		*pattern*	if	*condition*	=>	...
Scala	case	*pattern*	if	*condition*	=>	...
OCaml	|	*pattern*	when	*condition*	->	...
Python	case	*pattern*	if	*condition*	:	...
Swift	case	*pattern*	where	*condition*	:	...
Java	case	*pattern*	when	*condition*	->	...
C#		*pattern*	when	*condition*	=>	...
P2688R1		*pattern*	if	*condition*	=>	...

Other Languages

Cases: Arrows

Haskell		*pattern*	|	*condition*	->	...
Rust		*pattern*	if	*condition*	=>	...
Scala	case	*pattern*	if	*condition*	=>	...
OCaml	|	*pattern*	when	*condition*	->	...
Python	case	*pattern*	if	*condition*	:	...
Swift	case	*pattern*	where	*condition*	:	...
Java	case	*pattern*	when	*condition*	->	...
C#		*pattern*	when	*condition*	=>	...
P2688R1		*pattern*	if	*condition*	=>	...

Other Languages

Identifiers

	// New name	// Existing name
Haskell	`foo`	`N/A`
Rust	`foo`	`FOO` // uses the value if look-up finds a constant. // relies on naming convention to avoid issues.
Scala	`case foo`	``case `foo` `` // "stable identifiers" via backquotes `case Foo` // or names that start with capitals
OCaml	`| foo`	`N/A`
Python	`case foo`	`case Qualified.foo` // only supports qualified names
Swift	`case let foo` `case var foo`	`case foo`
Java	`case` *type* `foo` `case` *type*`(var foo, …)`	`case foo`
C#	*type* `foo` `var foo`	`foo`
P2688R1	`let foo`	`foo`

Other Languages

Identifiers

	// New name	// Existing name
`Haskell`	`foo`	`N/A`
`Rust`	`foo`	`FOO` // uses the value if look-up finds a constant. // relies on naming convention to avoid issues.
`Scala`	`case foo`	``case `foo` `` // "stable identifiers" via backquotes `case Foo` // or names that start with capitals
`OCaml`	`| foo`	`N/A`
`Python`	`case foo`	`case Qualified.foo` // only supports qualified names
`Swift`	`case let foo` `case var foo`	`case foo`
`Java`	`case` *type* `foo` `case` *type*`(var foo, …)`	`case foo`
`C#`	*type* `foo` `var foo`	`foo`
`P2688R1`	`let foo`	`foo`

Other Languages

Identifiers

	// New name	// Existing name	
Haskell	`foo`	N/A	
Rust	`foo`	`FOO`	// uses the value if look-up finds a constant. // relies on naming convention to avoid issues.
Scala	`case foo`	``case `foo` `` `case Foo`	// "stable identifiers" via backquotes // or names that start with capitals
OCaml	`| foo`	N/A	
Python	`case foo`	`case Qualified.foo`	// only supports qualified names
Swift	`case let foo` `case var foo`	`case foo`	
Java	`case` *type* `foo` `case` *type*`(var foo, …)`	`case foo`	
C#	*type* `foo` `var foo`	`foo`	
P2688R1	`let foo`	`foo`	

Other Languages

One-off Pattern Matches

Haskell	N/A
Rust	`if let` *pattern* = *expr* { … }
Scala	N/A
OCaml	N/A
Python	N/A
Swift	`if case` *pattern* = *expr* { … }
Java	`if` (*expr* `instanceof` *pattern*) { … }
C#	`if` (*expr* `is` *pattern*) { … }
P2688R1	`if` (*expr* `match` *pattern*) { … }

Other Languages

One-off Pattern Matches

Haskell	N/A
Rust	`if let` *pattern* = *expr* && *condition* { … }
Scala	N/A
OCaml	N/A
Python	N/A
Swift	`if case` *pattern* = *expr*, *condition* { … }
Java	`if` (*expr* `instanceof` *pattern* && *condition*) { … }
C#	`if` (*expr* `is` *pattern* && *condition*) { … }
P2688	`if` (*expr* `match` *pattern* && *condition*) { … } // extension

Other Languages

Binding the whole match

`Haskell`	*identifier* `@` *pattern*
`Rust`	*identifier* `@` *pattern*
`Scala`	*identifier* `@` *pattern*
`OCaml`	*pattern* `as` *identifier*
`Python`	*pattern* `as` *identifier*
`Swift`	`N/A`
`Java`	`N/A`
`C#`	`N/A`
`P2688R1`	*pattern* `let` *binding-pattern*

Other Languages

References

`Haskell`	https://www.haskell.org/tutorial/patterns.html
`Rust`	https://doc.rust-lang.org/reference/expressions/match-expr.html
	https://doc.rust-lang.org/reference/expressions/if-expr.html
	https://doc.rust-lang.org/reference/patterns.html
`Scala`	https://docs.scala-lang.org/scala3/book/control-structures.html#match-expressions
`OCaml`	https://v2.ocaml.org/manual/patterns.html
`Python`	https://peps.python.org/pep-0634/
	https://peps.python.org/pep-0635/
	https://peps.python.org/pep-0636/
`Swift`	https://docs.swift.org/swift-book/documentation/the-swift-programming-language/patterns
`Java`	https://docs.oracle.com/en/java/javase/21/language/pattern-matching-switch-expressions-and-statements.html
`C#`	https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/switch-expression
	https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns
	https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/is

Precedence of `match`

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right →
2	`a++ a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left ←
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer-to-member	Left-to-right →
5	`a*b a/b a%b`	Multiplication, division, and remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`< <= > >=`	For relational operators < and ≤ and > and ≥ respectively	
10	`== !=`	For equality operators = and ≠ respectively	
11	`a&b`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left ←
	`throw`	throw operator	
	`co_yield`	yield-expression (C++20)	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+= -=`	Compound assignment by sum and difference	
	`*= /= %=`	Compound assignment by product, quotient, and remainder	
	`<<= >>=`	Compound assignment by bitwise left shift and right shift	
	`&= ^= |=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right →

P2392 `is` P2688 `match`

C# `is`