Pattern Matching

A Sneak Peek

Michael Park

Facebook

P1371R1: Pattern Matching

Sergei Murzin

Michael Park

David Sankel

Dan Sarginson

GOALS

Build Intuition

Gather Feedback

Motivation

SELECTION

Perform different actions depending on a value

DECOMPOSITION

Retrieve components of a value

Switch: Too Limited

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

switch (s) {
  case "foo": std::cout << "got foo\n"; break;
  case "bar": std::cout << "got bar\n"; break;
}

// error: statement requires expression of integer type
```

If/Else: Too Flexible

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

double get_area(const Shape& shape) {
  if (auto circle = dynamic_cast<const Circle*>(&shape)) {
    return 3.14 * circle->radius * circle->radius;
  }
  if (auto rectangle = dynamic_cast<const Rectangle*>(&shape)) {
    return rectangle->width * rectangle->height;
  }
}
```

If/Else: Too Flexible

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

double get_area(const Shape& shape) {
  if (auto circle = dynamic_cast<const Circle*>(&shape)) {
    return 3.14 * circle->radius * circle->radius;
  }
  if (auto rectangle = dynamic_cast<const Rectangle*>(&shape)) {
    return rectangle->width * rectangle->height;
  }
}
```

Structural Association

```
int value = /* ... */;

switch (value) {
  case c1: /* ... */; break;
  case c2: /* ... */; break;
  default: // ...
}
```

```
int value = /* ... */;

if (b1) {
  // ...
} else if (b2) {
  // ...
} else {
  // ...
}
```

Bit Bashing

Papers About the author

std::visit is everything wrong with modern C++

Sep 14, 2017

HTTPS://BITBASHING.IO/STD-VISIT.HTML

Variant Visitation: Too Complex

```
std::variant<bool, int, std::string> v = /* ... */;

struct {
  int operator()(bool b) const {
    std::cout << "got bool: " << b << '\n';
  }
  int operator()(int n) const {
    std::cout << "got int: " << n << '\n';
  }
  int operator()(const std::string& s) const {
    std::cout << "got str: " << s << '\n';
  }
} visitor;

std::visit(visitor, v);
```

Variant Visitation: Too Complex

```
std::variant<bool, int, std::string> v = /* ... */;

std::visit(overload{
  [](bool b) { std::cout << "got bool: " << b << '\n'; },
  [](int n) { std::cout << "got int: " << n << '\n'; },
  [](const std::string& s) { std::cout << "got str: " << s << '\n'; }
}, v);
```

```
template <typename... Fs>
struct overload : Fs... { using Fs::operator()...; };

template <typename... Fs>
overload(Fs... fs) -> overload<Fs...>;
```

Variant Visitation: Too Complex

```
std::variant<bool, int, std::string> v = /* ... */;

std::visit([](const auto& arg) {
  using Arg = std::remove_cvref_t<decltype(arg)>;
  if constexpr (std::is_same_v<Arg, bool>) {
    std::cout << "got bool: " << arg << '\n';
  } else if constexpr (std::is_same_v<Arg, int>) {
    std::cout << "got int: " << arg << '\n';
  } else if constexpr (std::is_same_v<Arg, std::string>) {
    std::cout << "got str: " << arg << '\n';
  } else {
    static_assert(always_false<T>::value, "non-exhaustive visitor!");
  }
}, v);
```

```
template <typename T> struct always_false : std::false_type {};
```

SELECTION

Perform different actions depending on a value

DECOMPOSITION

Retrieve components of a value

Member Access

```
double get_area(const Shape& shape) {
  if (auto circle = dynamic_cast<const Circle*>(&shape)) {
    return 3.14 * circle->radius * circle->radius;
  }
  if (auto rectangle = dynamic_cast<const Rectangle*>(&shape)) {
    return rectangle->width * rectangle->height;
  }
}
```

Member Access

```
double get_area(const Shape& shape) {
  if (auto circle = dynamic_cast<const Circle*>(&shape)) {
    return 3.14 * circle->radius * circle->radius;
  }
  if (auto rectangle = dynamic_cast<const Rectangle*>(&shape)) {
    return rectangle->width * rectangle->height;
  }
}
```

Structured Bindings

```
double get_area(const Shape& shape) {
  if (auto circle = dynamic_cast<const Circle*>(&shape)) {
    const auto& [r] = *circle;
    return 3.14 * r * r;
  }
  if (auto rectangle = dynamic_cast<const Rectangle*>(&shape)) {
    const auto& [w, h] = *rectangle;
    return w * h;
  }
}
```

SELECTION

Perform different actions depending on a value

DECOMPOSITION

Retrieve components of a value

Select + Decompose

```
struct Quit {};
struct Move { int x; int y; };
struct Write { std::string text; };

struct ChangeColor {
  int red; int green; int blue;

};

using Message = std::variant<Quit, Move, Write, ChangeColor>;
```

Select + Decompose

```
Message msg = ChangeColor{0, 160, 255};

std::visit(overload{
  [](const Quit&) { std::cout << "Done\n"; },
  [](const Move& move) {
    const auto& [x, y] = move;
    std::cout << "Move by: " << x << ',' << y << '\n';
  },
  [](const Write& write) {
    const auto& [text] = write;
    std::cout << "Text message: " << text << '\n';
  },
  [](const ChangeColor& change_color) {
    const auto& [r, g, b] = change_color;
    std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
  }
}, msg);
```

Select + Decompose

```
struct Quit {};
struct Move { int x; int y; };
struct Write { std::string text; };

struct ChangeColor {
  int red; int green; int blue;

};

using Message = std::variant<Quit, Move, Write, ChangeColor>;
```

Select + Decompose: Nested

```
+ struct Rgb { int red; int green; int blue; };
+ struct Hsv { int hue; int saturation; int value; };

+ using Color = std::variant<Rgb, Hsv>;

  struct Quit {};
  struct Move { int x; int y; };
  struct Write { std::string text; };

  struct ChangeColor {
-   int red; int green; int blue;
+   Color color;
  };

  using Message = std::variant<Quit, Move, Write, ChangeColor>;
```

Select + Decompose: Nested

```
Message msg = ChangeColor{Rgb{0, 160, 255}};

std::visit(overload{
  [](const Quit&) { std::cout << "Done\n"; },
  // ...
  [](const ChangeColor& change_color) {
    const auto& [color] = change_color;
    std::visit(overload{
      [](const Rgb& rgb) {
        const auto& [r, g, b] = rgb;
        std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
      },
      [](const Hsv& hsv) {
        const auto& [h, s, v] = hsv;
        std::cout << "to HSV: " << h << ',' << s << ',' << v << '\n';
      }
    }, color);
  },
}, msg);
```

Rust

Select + Decompose: Nested

```
let msg = Message::ChangeColor(Color::Rgb(0, 160, 255));

match msg {
    Message::Quit                                => println!("Done"),
    Message::Move{ x, y }                        => println!("Move by: {},{}", x, y),
    Message::Write(text)                         => println!("Text message: {}", text),
    Message::ChangeColor(Color::Rgb(r, g, b)) => println!("to RGB: {},{},{}", r, g, b),
    Message::ChangeColor(Color::Hsv(h, s, v)) => println!("to HSV: {},{},{}", h, s, v),
}
```

HTTPS://DOC.RUST-LANG.ORG/BOOK/CH18-03-PATTERN-SYNTAX.HTML

Takeaways

Selection / decomposition are **extremely common** operations

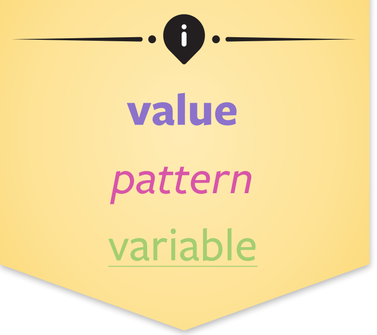
---

We want a general selection mechanism between **switch** / **if-else**

Selection / decomposition very often **nest**

Nested selection / decomposition leads to **indentation**

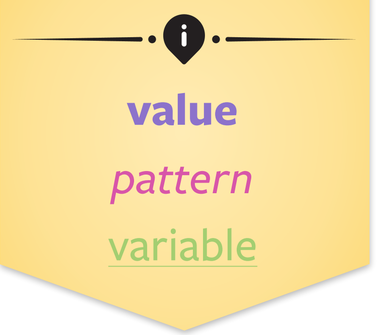
What is Pattern Matching?



“In pattern matching, we attempt to
match values against *patterns* and, if so desired,
bind variables to successful matches.”

HTTPS://EN.WIKIBOOKS.ORG/WIKI/HASKELL/PATTERN_MATCHING

Rust




```
struct Point { x: i32, y: i32 }

let point = Point { x: 0, y: 7 };

match point {
  Point { x   , y: 0 } => println!("X axis: {}", x),
  Point { x: 0, y    } => println!("Y axis: {}", y),
  Point { x   , y    } => println!("{}, {}", x, y),
}

// prints: "Y axis: 7"
```

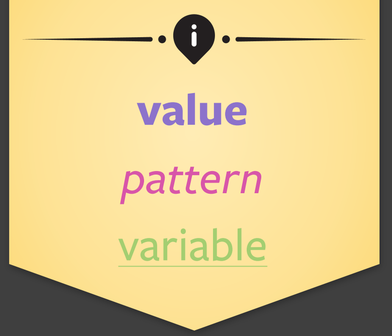
C++

value

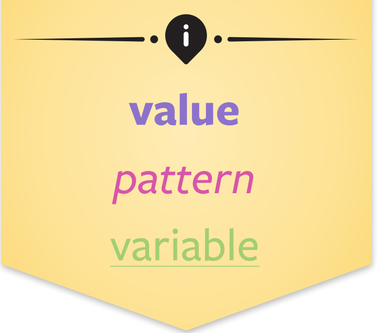
pattern

variable

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

auto point = Point { .x = 0, .y = 7 };

if (point.y == 0) {
  std::cout << "X axis: " << point.x << '\n';
} else if (point.x == 0) {
  std::cout << "Y axis: " << point.y << '\n';
} else {
  std::cout << point.x << ", " << point.y << '\n';
}

// prints: "Y axis: 7"
```



Declarative alternative to manually
testing values with *conditionals* and
extracting the desired components.



test **point**.*x* == *0*

```
Point { x: 0, y }
```

extract point.y

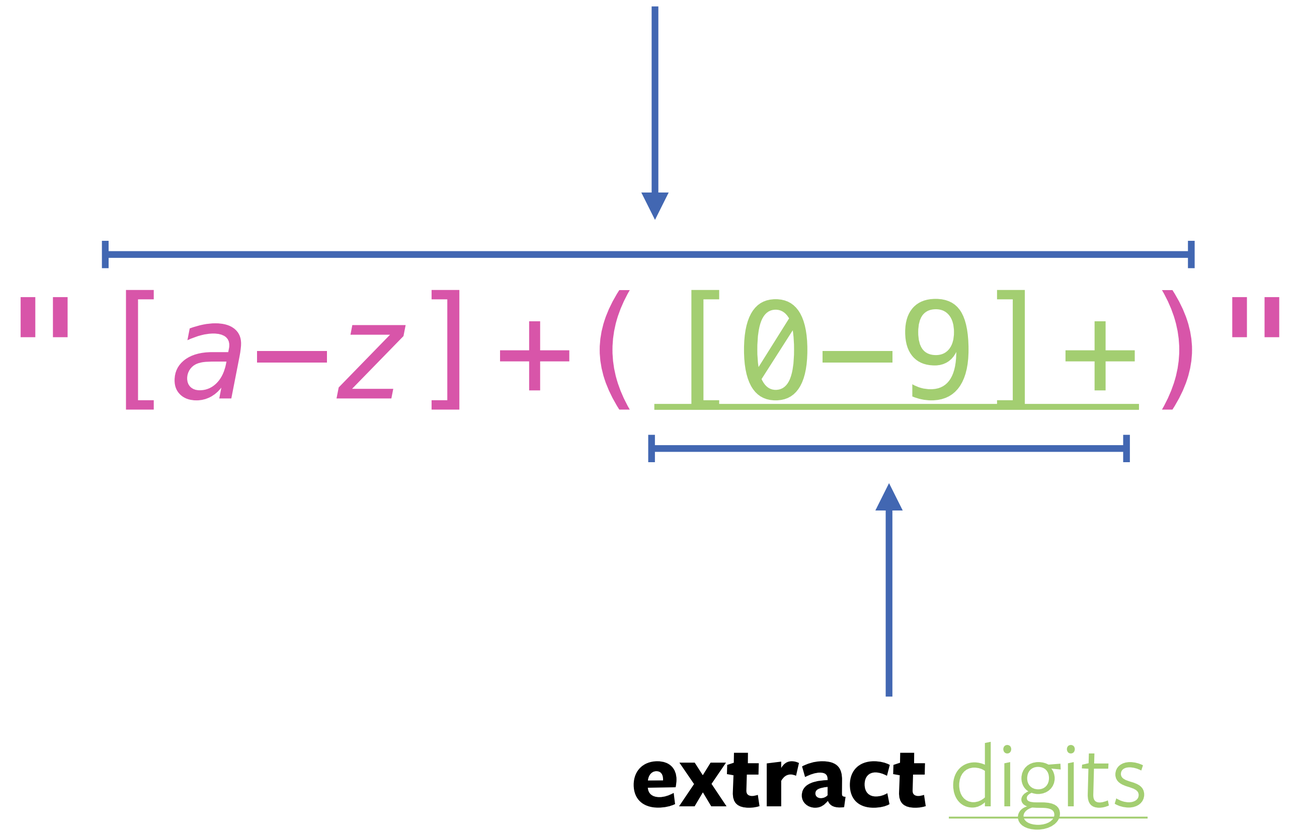
Regular Expressions

value

pattern

variable

test *letters followed by digits*

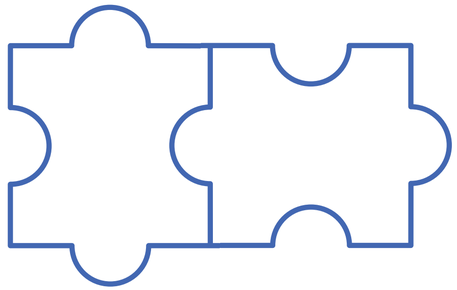


extract digits

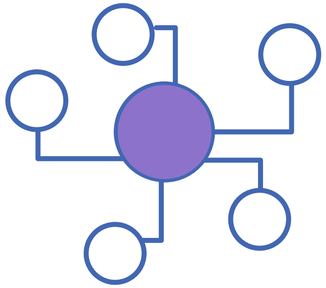
value

pattern

variable

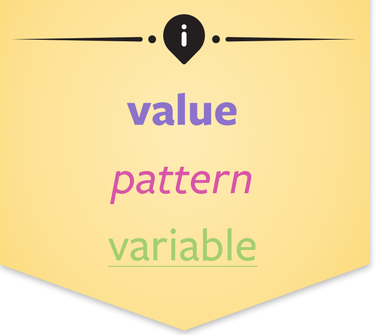


Match



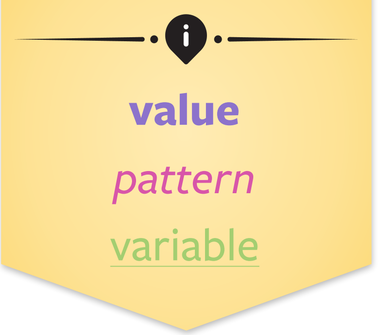
Bind

Select + Decompose



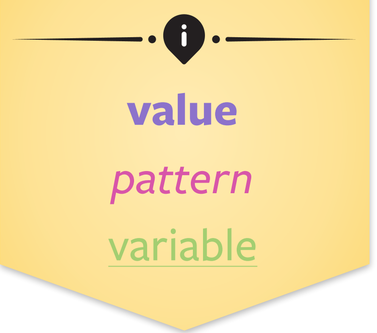

```
Message msg = ChangeColor{0, 160, 255};

std::visit(overload{
  [](const Quit&) { std::cout << "Done\n"; },
  [](const Move& move) {
    const auto& [x, y] = move;
    std::cout << "Move by: " << x << ',' << y << '\n';
  },
  [](const Write& write) {
    const auto& [text] = write;
    std::cout << "Text message: " << text << '\n';
  },
  [](const ChangeColor& change_color) {
    const auto& [r, g, b] = change_color;
    std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
  }
}, msg);
```

Select + Decompose



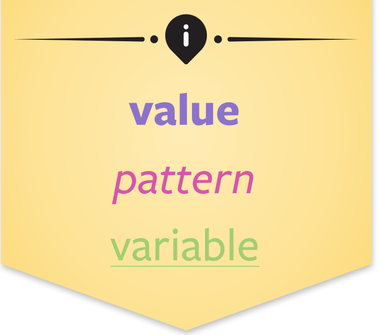

```
Message msg = ChangeColor{0, 160, 255};

std::visit(overload{
  [](const Quit&) { std::cout << "Done\n"; },
  [](const Move& move) {
    const auto& [x, y] = move;
    std::cout << "Move by: " << x << ',' << y << '\n';
  },
  [](const Write& write) {
    const auto& [text] = write;
    std::cout << "Text message: " << text << '\n';
  },
  [](const ChangeColor& change_color) {
    const auto& [r, g, b] = change_color;
    std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
  }
}, msg);
```

Select + Decompose



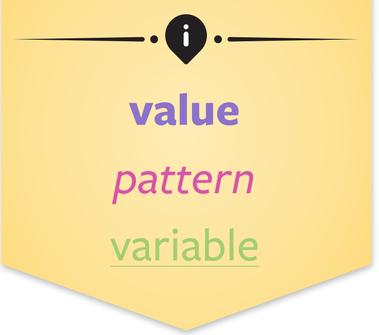

```
Message msg = ChangeColor{0, 160, 255};

inspect (msg) {
  <Quit> __: std::cout << "Done\n";
  <Move> [x, y]: std::cout << "Move by: " << x << ',' << y << '\n';
  <Write> [text]: std::cout << "Text message: " << text << '\n';
  <ChangeColor> [r, g, b]: {
    std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
  }
}
```

Select + Decompose: Nested



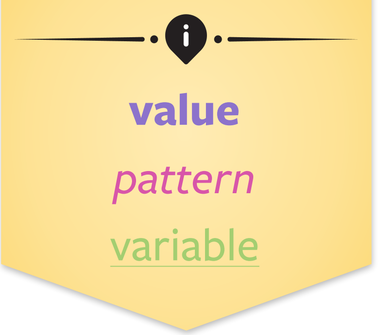

```
Message msg = ChangeColor{Rgb{0, 160, 255}};

std::visit(overload{
  [](const Quit&) { std::cout << "Done\n"; },
  // ...
  [](const ChangeColor& change_color) {
    const auto& [color] = change_color;
    std::visit(overload{
      [](const Rgb& rgb) {
        const auto& [r, g, b] = rgb;
        std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
      },
      [](const Hsv& hsv) {
        const auto& [h, s, v] = hsv;
        std::cout << "to HSV: " << h << ',' << s << ',' << v << '\n';
      }
    }, color);
  },
}, msg);
```

Select + Decompose: Nested




```
Message msg = ChangeColor{Rgb{0, 160, 255}};

std::visit(overload{
  [](const Quit&) { std::cout << "Done\n"; },
  // ...
  [](const ChangeColor& change_color) {
    const auto& [color] = change_color;
    std::visit(overload{
      [](const Rgb& rgb) {
        const auto& [r, g, b] = rgb;
        std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
      },
      [](const Hsv& hsv) {
        const auto& [h, s, v] = hsv;
        std::cout << "to HSV: " << h << ',' << s << ',' << v << '\n';
      }
    }, color);
  },
}, msg);
```

Select + Decompose: Nested



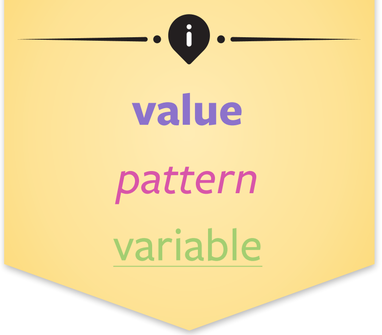

```
Message msg = ChangeColor{Rgb{0, 160, 255}};

inspect (msg) {
  <Quit> __: std::cout << "Done\n";
  <Move> [x, y]: std::cout << "Move by: " << x << ',' << y << '\n';
  <Write> [text]: std::cout << "Text message: " << text << '\n';\
  <ChangeColor> [ <Rgb> [r, g, b] ]: {
    std::cout << "to RGB: " << r << ',' << g << ',' << b << '\n';
  }
  <ChangeColor> [ <Hsv> [h, s, v] ]: {
    std::cout << "to HSV: " << h << ',' << s << ',' << v << '\n';
  }
}
```

KEY IDEA

Patterns and values are both built via composition

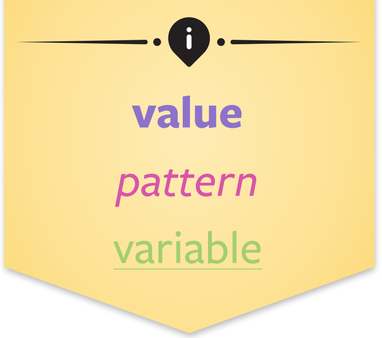
Structure

Statement Form



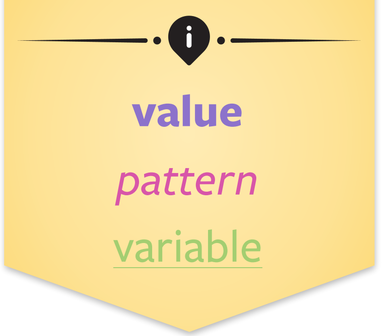

```
inspect constexpr_opt (init-statement_opt value) {
  pattern guard_opt : statement
  pattern guard_opt : statement
  ...
}
```

Statement Form



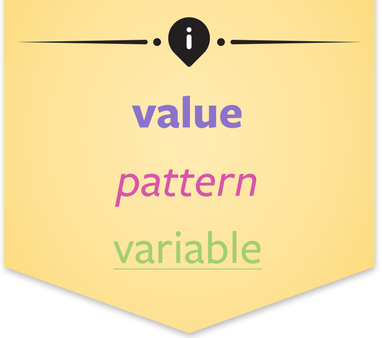

```
inspect (s) {
  "foo": std::cout << "got foo\n";
  "bar": std::cout << "got bar\n";
}
```

Expression Form



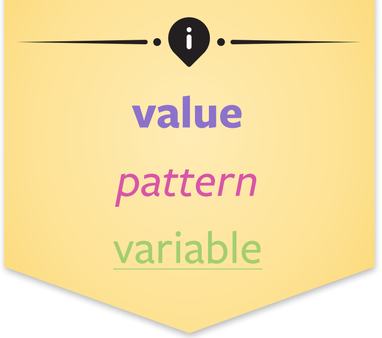

```
inspect constexpr_opt (init-statement_opt value) trailing-return-type_opt {
  pattern guard_opt => expression ,
  pattern guard_opt => expression ,
  ...
};
```

Expression Form



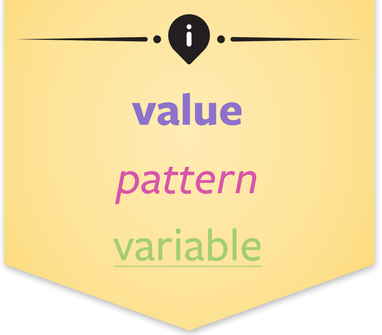

```
auto s = inspect (x) {
  0 => "zero"s,
  1 => "one"s,
};
```

Expression Form



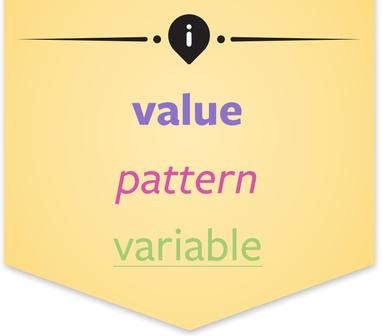

```
auto s = inspect (x) -> std::string {
  0 => "zero",
  1 => "one"s,
};
```

Pattern Guard



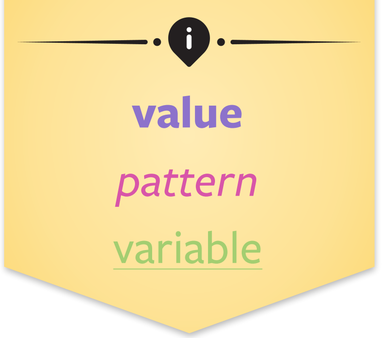

```
inspect (x) {
  [x, y] if (x == y): std::cout << "same\n";
  [x, y]: std::cout << "diff\n";
}
```

Pattern Guard




```
inspect (x) {
  [x, y] if (x == y): std::cout << "same\n";
  [x, y]: std::cout << "diff\n";
}
```

Declaration Form




```
auto pattern = value;
```

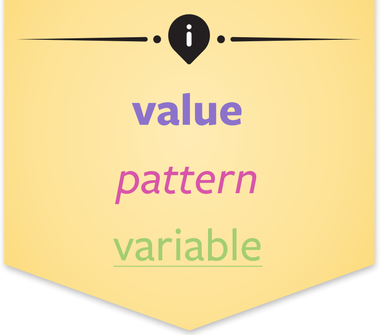
Overview of Patterns

DISCLAIMER

Details subject to change in newer revisions

Primary Patterns

Wildcard Pattern



P1110, P1469

- `__` (*double underscore*)

```
inspect (value) {
  __: {
    std::cout << "ignored\n";
  }
}

// prints: "ignored"
```

```
std::lock_guard<std::mutex> __(a);
// ...
std::lock_guard<std::mutex> __(b);



auto [x, __, __] = value;
```

Identifier Pattern

value

pattern

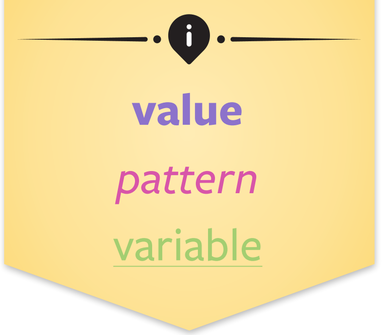
variable

- identifier

```
int value = 42;

inspect (value) {
  x: std::cout << x << '\n';
}

// prints: "42"
```

```
auto x = value;
```

Expression Pattern

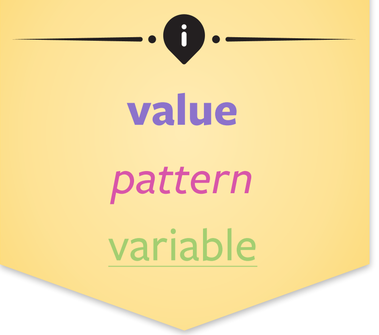


- *constant-expression* ... **except** identifier !

```
int value = 0;

inspect (value) {
  0: std::cout << "zero\n";
  1: std::cout << "one\n";
}

// prints: "zero"
```

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

Color color = Color::Red;

inspect (color) {
  Color::Red: std::cout << "red\n";
  Color::Green: std::cout << "green\n";
  Color::Blue: std::cout << "blue\n";
}

// prints: "red"
```

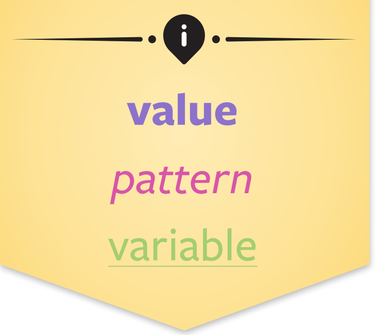
Why?



identifier

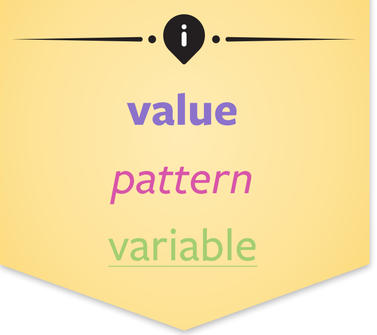
id-expression

Why?

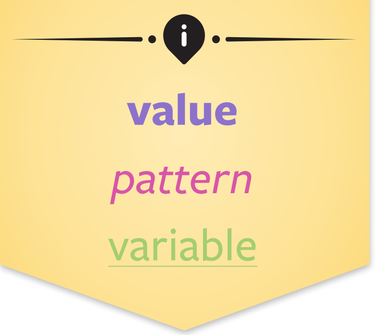



```
static constexpr int x = 101;

int value = 42;

inspect (value) {
  x: std::cout << x << '\n';
  __: std::cout << "no match\n";
}

// prints: ?
```

Why?




```
static constexpr int x = 101;

int value = 42;

inspect (value) {
  x: std::cout << x << '\n';
  __: std::cout << "no match\n";
}

// prints: ?
```



Bind Mode

- `bind` *pattern*
- Identifiers are identifier patterns

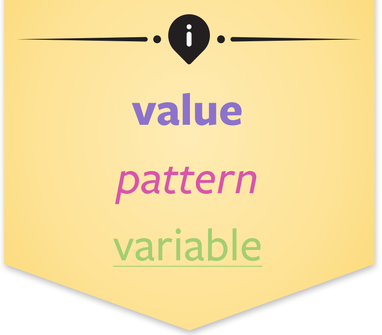
```
int value = 42;

inspect (value) {
  bind x: std::cout << x << '\n';
}

// prints: "42"
```

Expr Mode

- `expr` *pattern*
- Identifiers are *id-expressions*

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

Color color = Red;

inspect (color) {
  expr Red: std::cout << "red\n";
  expr Green: std::cout << "green\n";
  expr Blue: std::cout << "blue\n";
}

// prints: "red"
```

Observation



- Existing declarations
- Identifiers are identifier patterns

```
auto x = value;

auto [foo, bar] = value;
```

- Switch statement
- Identifiers are *id-expressions*

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

Color color = Red;

switch (color) {
  case Red:   /* ... */; break;
  case Green: /* ... */; break;
  case Blue:  /* ... */; break;
}
```

Let Pattern

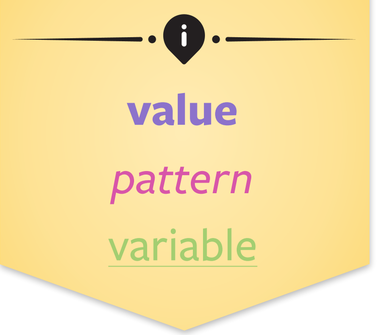
- `let` *pattern*
- Top-level is implicitly `let`

```
int value = 42;

inspect (value) {
  let x: std::cout << x << '\n';
//^^^ optional
}

// prints: "42"
```

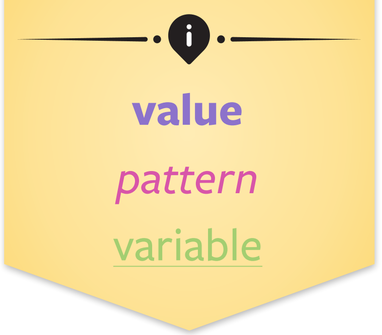
Case Pattern

- `case` *pattern*

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

Color color = Red;

inspect (color) {
  case Red: std::cout << "red\n";
  case Green: std::cout << "green\n";
  case Blue: std::cout << "blue\n";
}

// prints: "red"
```



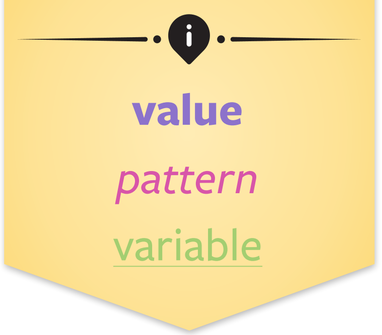
Putting Them Together




```
static constexpr int zero = 0, one = 1;

std::pair<int, std::pair<int, int>> p = /* ... */;

inspect (p) {
  case [zero, let [x, y]]: /* ... */;
  //    ^^^^ id-expression
  case [one , let [x, y]]: /* ... */;
  //                ^  ^ id-pattern
}
```

Compound Patterns

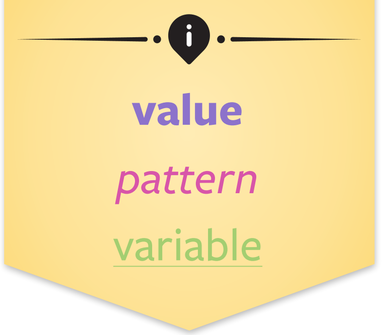
Structured Binding Pattern (1)



- [$pattern_1$, $pattern_2$, ..., $pattern_N$]

```
std::pair<int, int> p = /* ... */;

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;
}
```

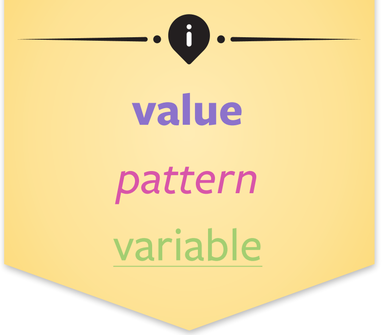
Structured Binding Pattern (2)



- [*designator*$_1$: *pattern*$_1$, *designator*$_2$: *pattern*$_2$, ..., *designator*$_N$: *pattern*$_N$]

```
struct Player { int hitpoints; int coins; };

void get_hint(const Player& player) {
  inspect (player) {
    [.hitpoints: 1]: std::cout << "You're almost destroyed!\n";
    [.hitpoints: 10, .coins: 10]: {
      std::cout << "I need the hints from you!\n";
    }
    [.coins: 10]: std::cout << "Get more hitpoints!\n";
    [.hitpoints: 10]: std::cout << "Get more ammo!\n";
  }
}
```

Alternative Pattern

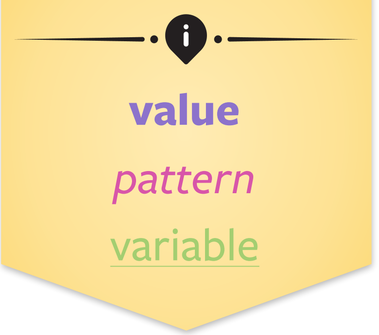


VariantLike

- < *type* > *pattern*

```
std::variant<int, float> v = /* ... */;

inspect (v) {
  <int> i: std::cout << "got int: " << i << '\n';
  <float> f: std::cout << "got float: " << f << '\n';
}
```

Alternative Pattern

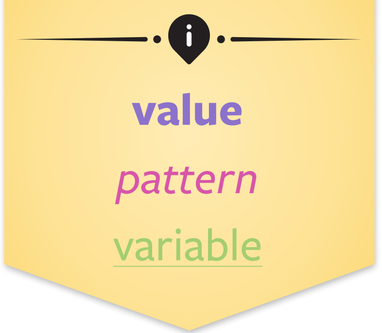


VariantLike

- < *type* > *pattern*

```
std::variant<int, int> v = /* ... */;

inspect (v) {
  <int> i: std::cout << "got int: " << i << '\n';
}
```

Alternative Pattern

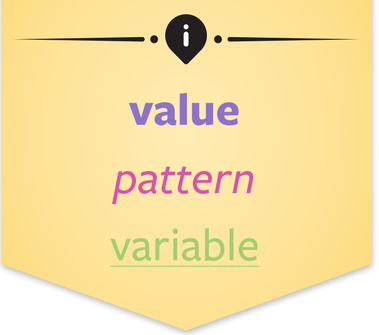


VariantLike

- < *constant-expression* > *pattern*

```
std::variant<int, int> v = /* ... */;

inspect (v) {
  <0> first: std::cout << "got first int: " << first << '\n';
  <1> second: std::cout << "got second int: " << second << '\n';
}
```

Alternative Pattern

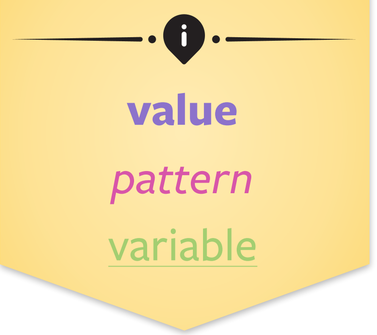


VariantLike

- `< auto >` *pattern*

```
std::variant<int, std::string> v = /* ... */;

inspect (v) {
  <auto> x: std::cout << "got: " << x << '\n';
}
```

Alternative Pattern

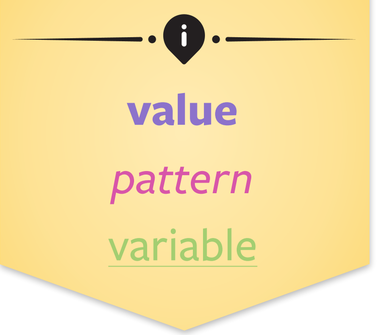


VariantLike

- < *concept* > *pattern*

```
std::variant<bool, char, int, float, std::string> v = /* ... */;

inspect (v) {
  <Integral> i: std::cout << "got an integral: " << i << '\n';
  <auto> x: std::cout << "got : " << x << '\n';
}
```

Alternative Pattern

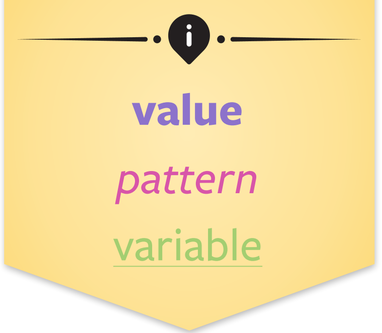


AnyLike

- < *type* > *pattern*

```
std::any a = 42;

inspect (a) {
  <int> i: std::cout << "got int: " << i << '\n';
  <float> f: std::cout << "got float: " << f << '\n';
}
```

Alternative Pattern

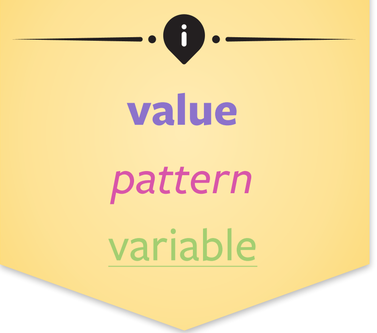


Polymorphic Types

- `<` *type* `>` *pattern*

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

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

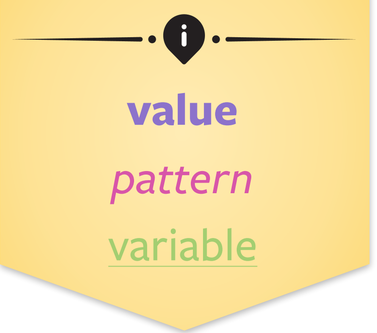
Dereference Pattern



- `*!` *pattern*
- `*?` *pattern*

```
auto v = { 3, 9, 1, 4, 2, 5, 9 };
auto [min_iter, max_iter] = std::ranges::minmax_element(v);

std::cout << "min = " << *min_iter << ", "
          << "max = " << *max_iter << '\n';
```

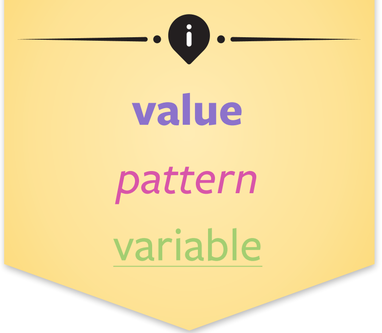
Dereference Pattern



- `*!` *pattern*
- `*?` *pattern*

```
auto v = { 3, 9, 1, 4, 2, 5, 9 };
auto [*! min, *! max] = std::ranges::minmax_element(v);

std::cout << "min = " << min << ", max = " << max << '\n';
```

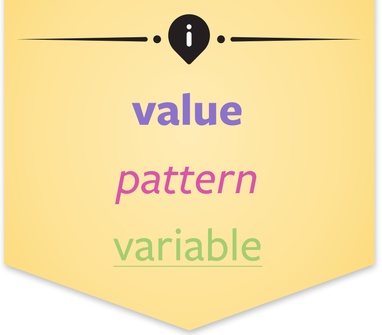
Dereference Pattern



- `*!` *pattern*
- `*?` *pattern*

```
struct Employee { int id; std::string name; };

std::vector<Employee> employees = /* ... */;

auto [*! [.name: min_name], *! [.name: max_name]] =
    std::ranges::minmax_element(employees, {}, &Employee::id);

std::cout << "min id employee = " << min_name << ", "
          << "max id employee = " << max_name << '\n';
```

Dereference Pattern



- `*!` *pattern*
- `*?` *pattern*

```
struct Node {
  int value;
  std::unique_ptr<Node> next;
};

bool starts_with_two_zeros(const Node& node) {
  return inspect (node) {
    [.value: 0, .next: *? [.value: 0]] => true,
    __ => false,
  };
}
```

Dereference Pattern

value

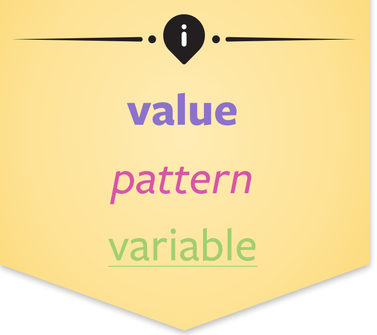
pattern

variable

- `*!` *pattern*
- `*?` *pattern*

```
struct Node {
  int value;
  std::unique_ptr<Node> next;
};

bool starts_with_two_zeros(const Node& node) {
  return inspect (node) {
    [.value: 0, .next: *? [.value: 0]] => true,
    __ => false,
  };
}
```

Extractor Pattern



- (*constant-expression* ! *pattern*)
- Unchecked extractor pattern

```
( e! pattern ) matches value if:

auto&& x = e.extract(value);

if (pattern matches x)
```

- (*constant-expression* ? *pattern*)
- Checked extractor pattern

```
( e? pattern ) matches value if:

auto&& x = e.try_extract(value);

if (x && pattern matches *x)
```

Compile-Time Regular Expressions

https://github.com/hanickadot/compile-time-regular-expressions

CTRE: Basic Example

```
bool is_a_date(std::string_view input) noexcept {
  return ctre::match<"[0-9]{4}/[0-9]{2}/[0-9]{2}">(input);
}
```

ctre::match<”REGEX”>

```
void f(std::string_view input) {
  constexpr auto id =
    ctre::match<"[a-z]+([0-9]+)">;

  auto result = id.try_extract(input);

  // implicitly convertible to bool
  if (result) {

    // unpack with structured bindings
    auto [whole, digits] = *result;

  }
}
```

ctre::match<”REGEX”>

```
void f(std::string_view input) {
  constexpr auto id =
    ctre::match<"[a-z]+([0-9]+)">;

  auto result = id.try_extract(input);

  // implicitly convertible to bool
  if (result) {

    // unpack with structured bindings
    auto [whole, digits] = *result;

  }
}
```

```
( e? pattern ) matches value if:

auto&& x = e.try_extract(value);

if (x && pattern matches *x)
```

ctre::match<”REGEX”>

```
void f(std::string_view input) {
  constexpr auto id =
    ctre::match<"[a-z]+([0-9]+)">;

  auto result = id.try_extract(input);

  // implicitly convertible to bool
  if (result) {

    // unpack with structured bindings
    auto [whole, digits] = *result;

  }
}
```

```
( id? [w, d] ) matches input if:

auto&& x = id.try_extract(input);

if (x && [w, d] matches *x)
```

Extractor Example: CTRE

```
inline constexpr auto id = ctre::match<"[a-z]+([0-9]+)">;

inline constexpr auto date = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">;

inspect (s) {
  (id? [whole, digits]): // ...
  (date? [whole, year, month, day]): // ...
}
```

Thank you!

Michael Park

mcypark@gmail.com

@mcypark