Pattern Matching

Match Me If You Can

Michael Park

Facebook

P1371R0: 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 = /* ... */;

switch (s) {  // error: statement requires expression of integer type
  case "foo":
    std::cout << "got foo\n";
    break;
  case "bar":
    std::cout << "got bar\n";
    break;
  default:
    std::cout << "don't care\n";
}
```

If-Else: Too Flexible

```
struct Expr { virtual ~Expr() = default; };
struct Int : Expr { int value; };
struct Neg : Expr { std::unique_ptr<Expr> expr; };
struct Add : Expr { std::unique_ptr<Expr> lhs, rhs };

int eval(const Expr& expr) {
  if (auto int_ = dynamic_cast<const Int*>(&expr)) {
    return int_->value;
  }
  if (auto neg = dynamic_cast<const Neg*>(&expr)) {
    return -eval(*neg->expr);
  }
  if (auto add = dynamic_cast<const Add*>(&expr)) {
    return eval(*add->lhs) + eval(*add->rhs);
  }
}
```

If-Else: Too Flexible

```
struct Expr { virtual ~Expr() = default; };
struct Int : Expr { int value; };
struct Neg : Expr { std::unique_ptr<Expr> expr; };
struct Add : Expr { std::unique_ptr<Expr> lhs, rhs };

int eval(const Expr& expr) {
  if (auto int_ = dynamic_cast<const Int*>(&expr)) {
    return int_->value;
  }
  if (auto neg = dynamic_cast<const Neg*>(&expr)) {
    return -eval(*neg->expr);
  }
  if (auto add = dynamic_cast<const Add*>(&expr)) {
    return eval(*add->lhs) + eval(*add->rhs);
  }
}
```

Structural Association

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

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

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

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

Principle of Least Power

“Computer Science in the 1960s to 80s spent a lot of effort making languages that were as powerful as possible. Nowadays we have to appreciate the reasons for picking not the most powerful solution but the least powerful. [...]”

TIM BERNERS-LEE, 1998

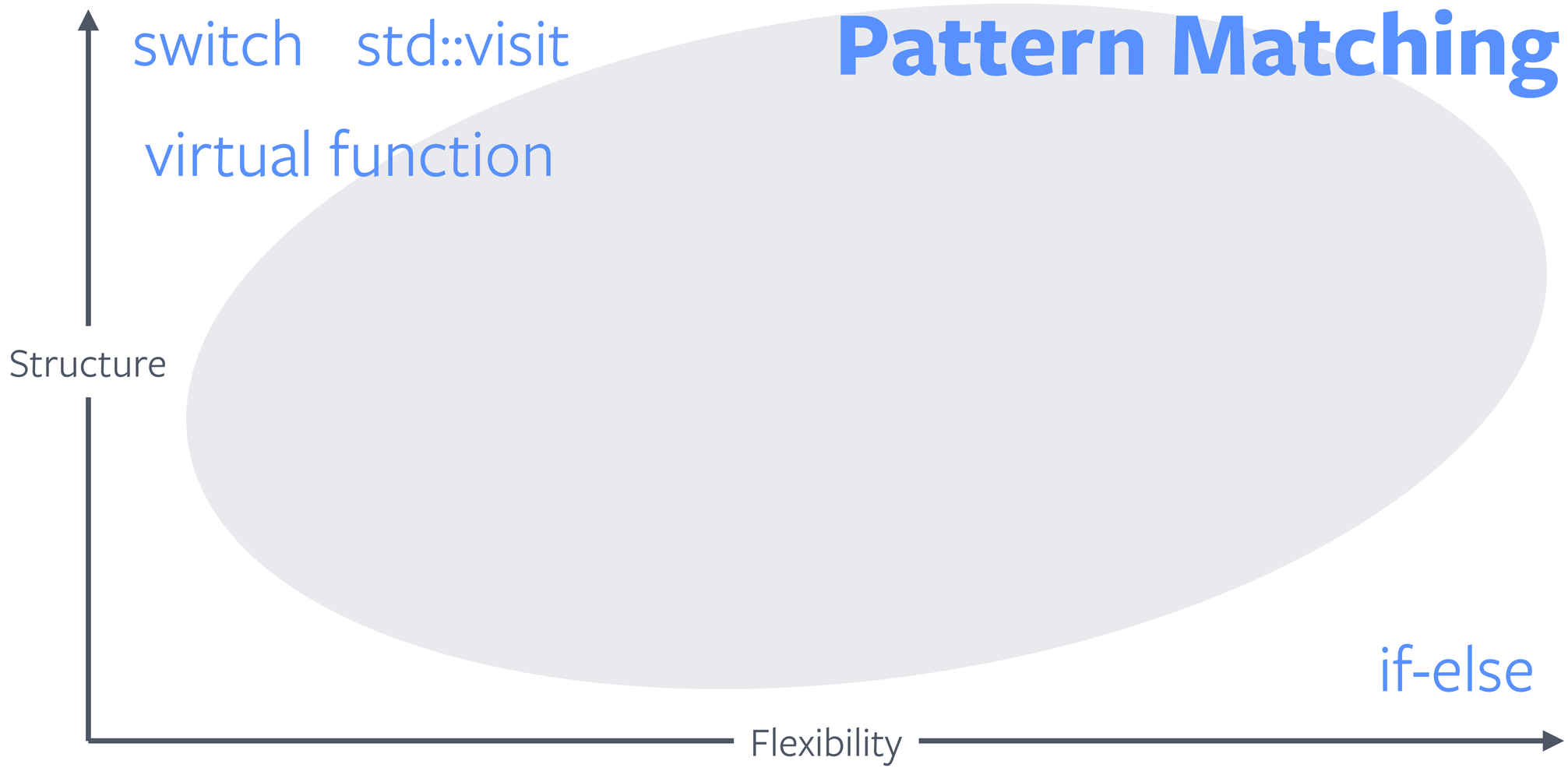
HTTPS://WWW.W3.ORG/DESIGNISSUES/PRINCIPLES.HTML#PLP

Principle of Least Power

“[...] The reason for this is that the less powerful the language, the more you can do with the data stored in that language. If you write it in a simple declarative form, anyone can write a program to analyze it in many ways.”

TIM BERNERS-LEE, 1998

HTTPS://WWW.W3.ORG/DESIGNISSUES/PRINCIPLES.HTML#PLP


switch
std::visit
virtual function
Pattern Matching
Structure
if-else
Flexibility

Bit Bashing

Papers About the author

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

Sep 14, 2017

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

eracpp 1:21 PM

At what point would you start considering

```
std::variant<A, B, ...>
```

over a simple tagged union?

```
struct V {
  union { A a; B b; ... };
  Tag tag; // An enum.
};
```

[...]

quicknir 1:22 PM

I guess my main question would be, why are you avoiding variant?

eracpp 1:22 PM

The complexity of the interface.

bigcheese 1:25 PM

I would almost never go with variant.

Because using it is painful.

[...]

bigcheese 1:29 PM

When we get pattern matching, assuming it doesn't suck, then I'll start using variant.

☝ 1

SELECTION

Perform different actions depending on a value

DECOMPOSITION

Retrieve components of a value

Extracting Fields

```
int eval(const Expr& expr) {
  if (auto int_ = dynamic_cast<const Int*>(&expr)) {
    return int_->value;
  }
  if (auto neg = dynamic_cast<const Neg*>(&expr)) {
    return -eval(*neg->expr);
  }
  if (auto add = dynamic_cast<const Add*>(&expr)) {
    return eval(*add->lhs) + eval(*add->rhs);
  }
}
```

Extracting Fields

```
int eval(const Expr& expr) {
  if (auto int_ = dynamic_cast<const Int*>(&expr)) {
    return int_->value;
  }
  if (auto neg = dynamic_cast<const Neg*>(&expr)) {
    return -eval(*neg->expr);
  }
  if (auto add = dynamic_cast<const Add*>(&expr)) {
    return eval(*add->lhs) + eval(*add->rhs);
  }
}
```

SELECTION

Perform different actions depending on a value

DECOMPOSITION

Retrieve components of a value

Rust

Select-Decompose

```
enum Message {
  Quit,
  Move { x: i32, y: i32 },
  Write(String),
  ChangeColor(i32, i32, i32),
}

let msg = Message::ChangeColor(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(r, g, b)    => println!("to RGB({}, {}, {})", r, g, b),
}

// prints: "to RGB(0, 160, 255)"
```

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

C++

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>;

Message msg = ChangeColor{0, 160, 255};
std::visit(overload{
  [](const Quit&) { fmt::print("Done"); },
  [](const Move& move) { fmt::print("Move by ({}, {})", move.x, move.y); },
  [](const Write& write) { fmt::print("Text message: {}", write.text); },
  [](const ChangeColor& change_color) {
    const auto& [r, g, b] = change_color;
    fmt::print("to RGB({}, {}, {})", r, g, b);
  }
}, msg);

// prints: "to RGB(0, 160, 255)"
```

Rust

Select-Decompose-Select-Decompose

```
enum Color { Rgb(i32, i32, i32), Hsv(i32, i32, i32), }

enum Message {
  Quit,
  Move { x: i32, y: i32 },
  Write(String),
  ChangeColor(Color),
}

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

match msg {
  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),
  _ => (),
}

// prints: "to HSV(0, 160, 255)"
```

C++

Select-Decompose-Select-Decompose

```
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 { Color color; };

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

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

std::visit(overload{
  [](const ChangeColor& change_color) {
    std::visit(overload{
      [](const Rgb& rgb) {
        const auto& [r, g, b] = rgb;
        fmt::print("to RGB({}, {}, {})", r, g, b);
      },
      [](const Hsv& hsv) {
        const auto& [h, s, v] = hsv;
        fmt::print("to HSV({}, {}, {})", h, s, v);
      }
    }, change_color.color);
  },
  [](const auto&) {}
}, msg);

// prints: "to HSV(0, 160, 255)"
```

Takeaways

Selection / decomposition are **extremely common** operations

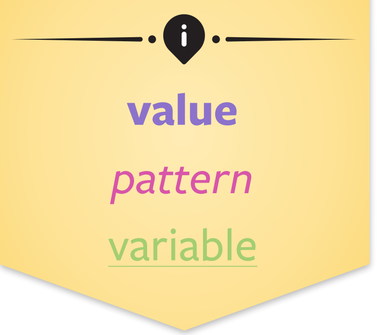
---

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

Selection / decomposition very often **nest**

Nested selection / decomposition leads to **indentation**

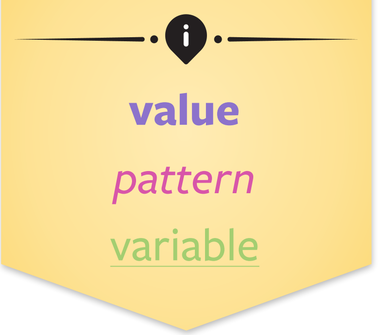
What is Pattern Matching?



“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"
```

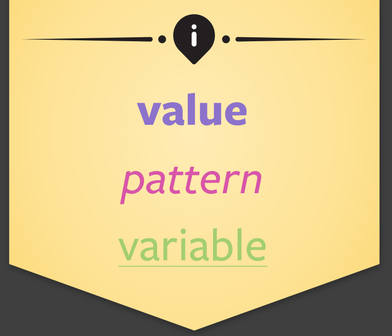
C++

value

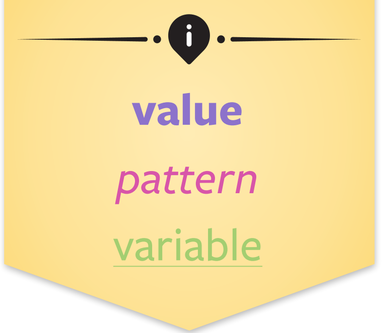
pattern

variable

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

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

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

// 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

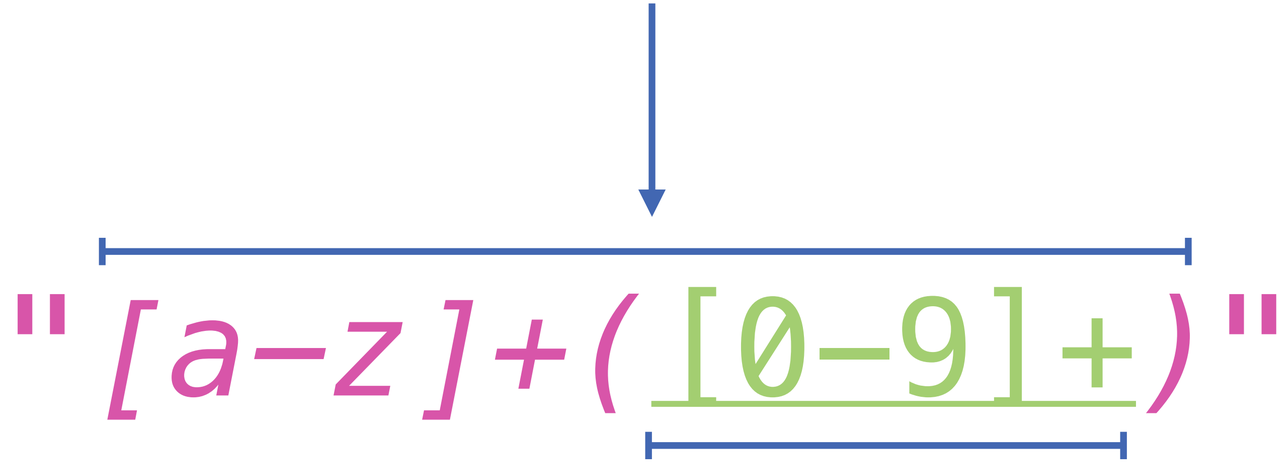
Regular Expressions

value

pattern

variable

test *letters followed by digits*

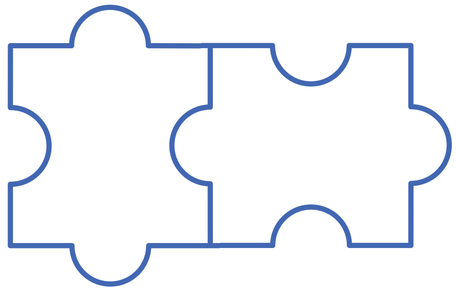


extract digits

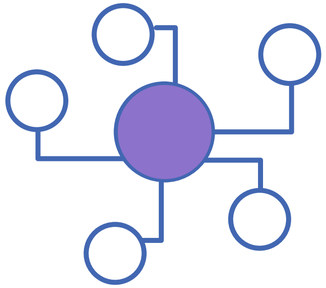
value

pattern

variable



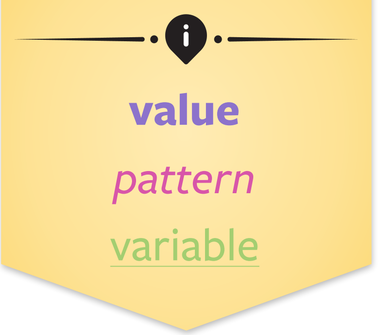
Match



Bind

Rust

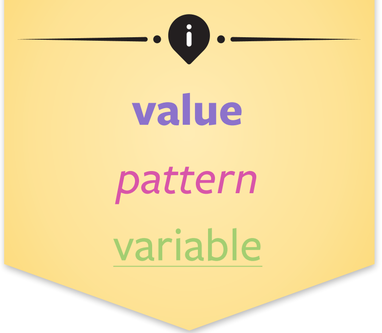
Select-Decompose




```
enum Message {
  Quit,
  Move { x: i32, y: i32 },
  Write(String),
  ChangeColor(i32, i32, i32),
}

let msg = Message::ChangeColor(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(r, g, b) => println!("to RGB({}, {}, {})", r, g, b),
}

// prints: "to RGB(0, 160, 255)"
```

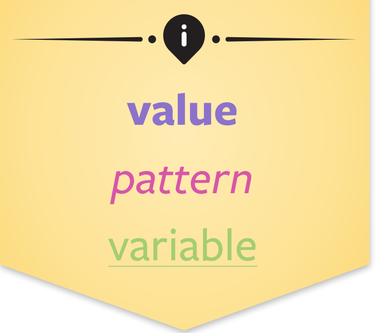
C++ with P1371R0

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>;

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

inspect (msg) {
  <Quit> __: fmt::print("Done");
  <Move> [x, y]: fmt::print("Move by ({}, {})", x, y);
  <Write> [text]: fmt::print("Text message: {}", text);
  <ChangeColor> [r, g, b]: fmt::print("to RGB({}, {}, {})", r, g, b);
}

// prints: "to RGB(0, 160, 255)"
```

Rust

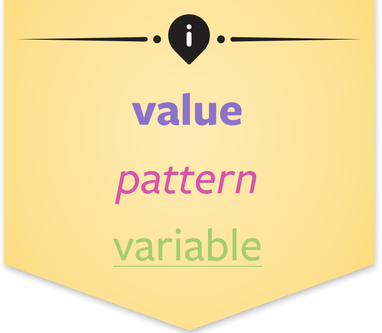


Select-Decompose-Select-Decompose

```
enum Color { Rgb(i32, i32, i32), Hsv(i32, i32, i32), }

enum Message {
  Quit,
  Move { x: i32, y: i32 },
  Write(String),
  ChangeColor(Color),
}

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

match msg {
  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),
  _ => (),
}

// prints: "to HSV(0, 160, 255)"
```

C++ with P1371R0

Select-Decompose-Select-Decompose




```
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 { Color color; };
using Message = std::variant<Quit, Move, Write, ChangeColor>;

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

inspect (msg) {
  <ChangeColor> [<Rgb> [r, g, b]]: fmt::print("to RGB({}, {}, {})", r, g, b);
  <ChangeColor> [<Hsv> [h, s, v]]: fmt::print("to HSV({}, {}, {})", h, s, v);
}

// prints: "to HSV(0, 160, 255)"
```

KEY IDEA

Patterns and values are both built via composition

Three Forms

Statement Form

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

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,
};
```

Declaration Form

```
auto [x, [y, z]] = /* ... */;
```

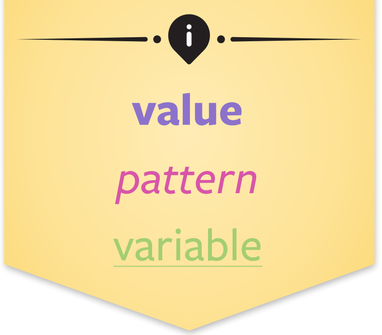
Overview of Patterns

DISCLAIMER

Details subject to change in newer revisions

Primary Patterns

Wildcard Pattern

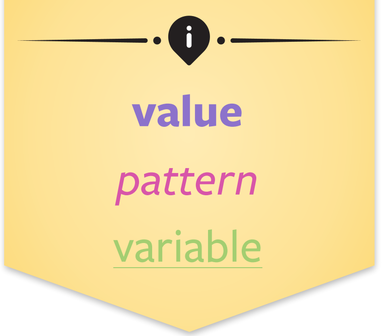


P1110, P1469

- `__` (double underscore)

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

// prints: "ignored"
```

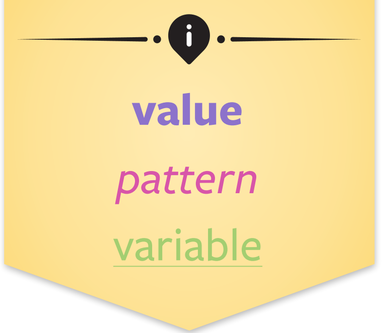
Identifier Pattern



- *identifier*

```
int value = 42;

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

// prints: "42"
```

Expression Pattern



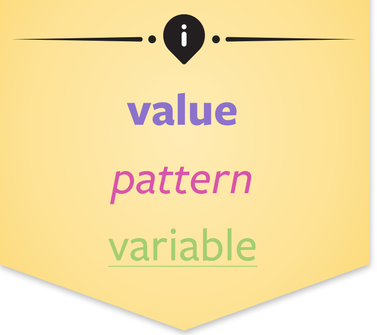
- *literal*

```
int value = 0;

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

// prints: "got zero"
```

- ^ *primary-expression*

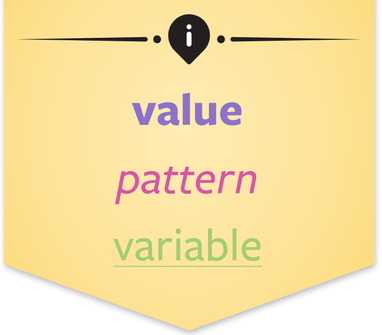
```
static constexpr int zero = 0;
static constexpr int one = 1;

int value = 0;

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

// prints: "got zero"
```

Matcher Example: within




```
struct within {
  int first, last;

  constexpr bool match(int n) const { return first <= n && n <= last; }
};

int n = 1;

inspect (n) {
  ^(within{0, 9}): std::cout << n << " is in [0, 9].";
  __: std::cout << n << " is not in [0, 9].";
}

// prints: "1 is in [0, 9]."
```

Parenthesized Pattern

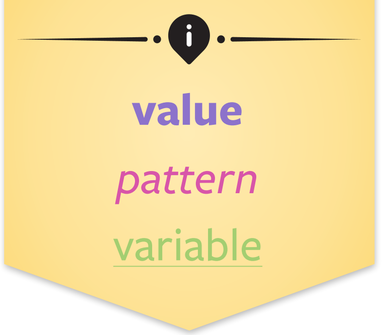


- (*pattern*)

```
static constexpr int one = 1;

int value = 0;

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

// prints: "got zero"
```

Compound Patterns

Structured Binding Pattern (1)



- [$pattern_0$, $pattern_1$, ..., $pattern_N$]

```
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)

value

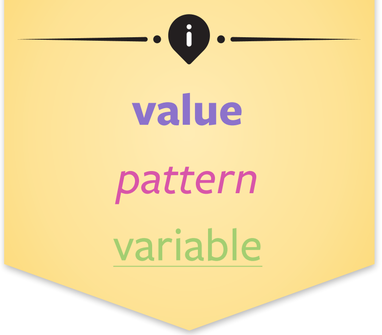
pattern

variable

- [*designator*$_0$: *pattern*$_0$, *designator*$_1$: *pattern*$_1$, ..., *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

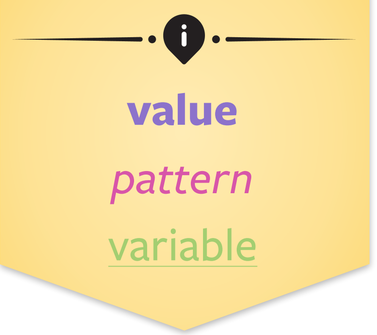


VariantLike

- < *type* > *pattern*

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

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

Alternative Pattern

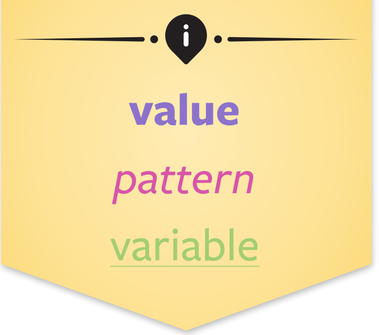


VariantLike

- < *type* > *pattern*

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

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

Alternative Pattern

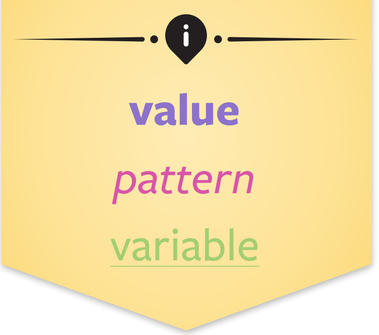


VariantLike

- < *constant-expression* > *pattern*

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

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

Alternative Pattern

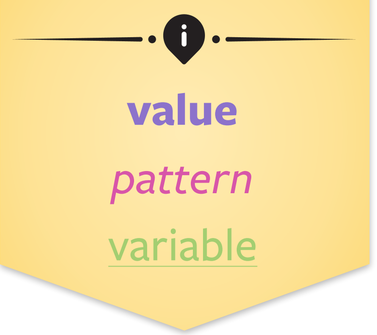


VariantLike

- `< auto >` *pattern*

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

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

Alternative Pattern

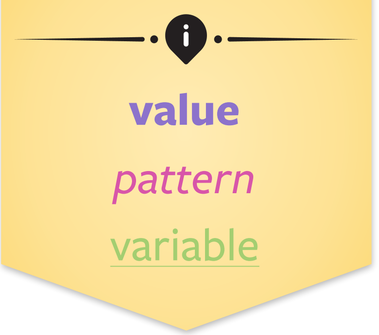


VariantLike

- < *concept* > *pattern*

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

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

Alternative Pattern

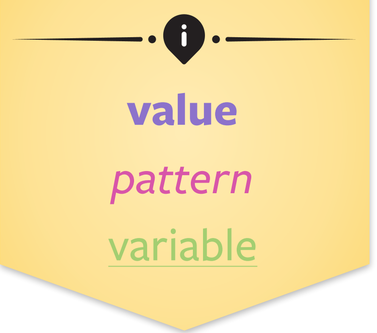


AnyLike

- < *type* > *pattern*

```
std::any a = 42;

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

Alternative Pattern

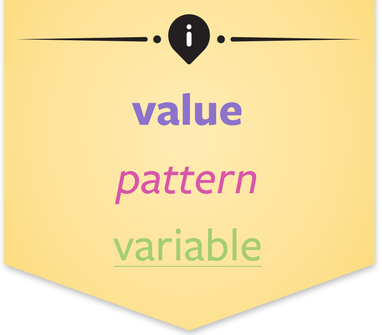


Polymorphic Types

- < *type* > *pattern*

```
struct Shape { virtual ~Shape() = default; };

struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

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

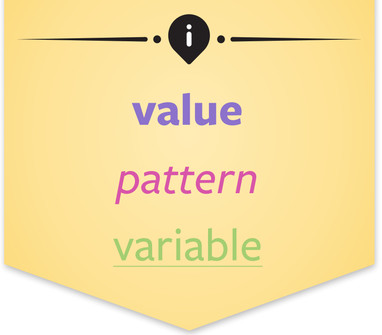
Binding Pattern



- *identifier* @ *pattern*

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

inspect (v) {
  <Point> (p @ [x, y]): // ...
}
```

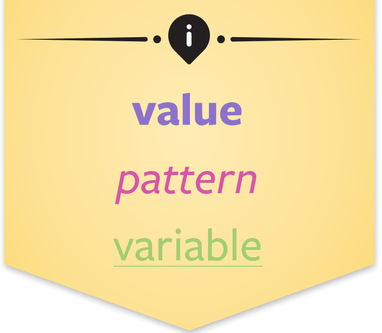
Binding Pattern



- *identifier* `@` *pattern*

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

inspect (v) {
  <Point> (p @ [x, y]): // ...
}
```

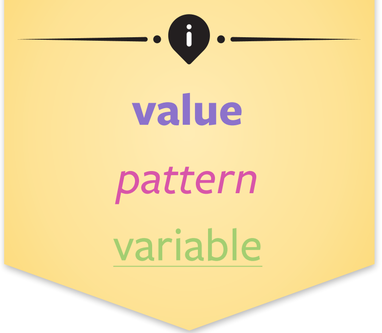
Dereference Pattern



- `*` *pattern*

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

void f(const Node& node) {
  inspect (node) {
    [.value: 0, .next: *[.value: 0]]: { std::cout << "00\n"; }
    __: std::cout << "otherwise\n";
  }
}
```

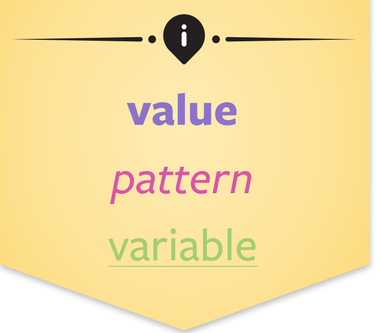
Dereference Pattern



- `*` *pattern*

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

void f(const Node& node) {
  inspect (node) {
    [.value: 0, .next: *[.value: 0]]: { std::cout << "00\n"; }
    __: std::cout << "otherwise\n";
  }
}
```

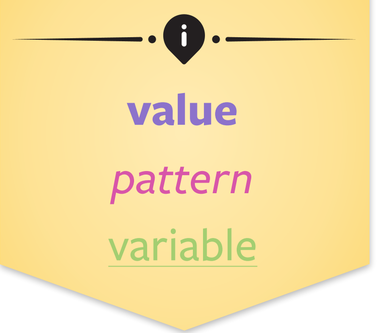
Extractor Pattern



- (*constant-expression* ! *pattern*)
- (*constant-expression* ? *pattern*)

- Matchers only perform matching
- Unchecked extractors enable binding
- Checked extractors enable matching + binding

Extractor Pattern



- (*constant-expression* ! *pattern*)
- (*constant-expression* ? *pattern*)

- `( e ! p ):` match `p` against `e.extract(value)`
- `( e ? p ):`
  - `if (auto result = e.try_extract(value)) {`
    match `p` against `*result`
    `}`

Extractor Example: deref

```
struct Deref {
  template <typename T>
  auto&& extract(T&& arg) const {
    return *std::forward<T>(arg);
  }

  template <typename T>
  auto&& try_extract(T&& arg) const {
    return std::forward<T>(arg);
  }
};

inline constexpr Deref deref;
```

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

void f(const Node& node) {
  inspect (node) {
    [.value: 0, .next: (deref? [.value: 0])]: {
      std::cout << "00\n";
    }
    __: std::cout << "otherwise\n";
  }
}
```

Compile-Time Regular Expressions

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

Extractor Example: CTRE

```
template <ctl::fixed_string fs>
struct re {
  constexpr auto try_extract(std::string_view sv) const {
    return ctre::match<fs>(sv);
  }
};

inline constexpr auto number = re<"[a-z]+([0-9]+)">{};

inline constexpr auto date = re<"(\\d{4})/(\\d{1,2}+)/(\\d{1,2}+)">{};

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

Extractor Example: CTRE

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

inline constexpr auto date =
    ctre::match<"(\\d{4})/(\\d{1,2}+)/(\\d{1,2}+)">;

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

Extractor Example: CTRE

```
inspect (s) {

  (ctre::match<"[a-z]+([0-9]+)">?
      [whole, n]): // ...

  (ctre::match<"(\\d{4})/(\\d{1,2}+)/(\\d{1,2}+)">?
      [whole, year, month, day]): // ...

}
```

Patterns vs Expressions

Example: Identifiers

```
std::pair p = {101, 202};

constexpr int x = 42;

inspect (p) {
  [x, y]: std::cout << "A\n";
  __: std::cout << "B\n";
}
```

Without Identifiers

```
int x = /* ... */;

inspect (x) {
  1 | 2: // bitwise-or? or alternation pattern?
}
```

Rust

```
let x = 1 | 2;

match x {
  1 | 2 => println!("1 | 2"),
  _ => println!("otherwise"),
}

// prints: "otherwise"
```

Swift

```
let x = 1 | 2;

switch x {
  case 1 | 2: print("1 | 2");
  case _: print("otherwise");
}

// prints: "1 | 2"
```

Pattern

```
auto x = 1 | 2;

inspect (x) {
  1 | 2: std::cout << "1 | 2\n";
  __: std::cout << "otherwise\n";
}

// prints: "otherwise"
```

Expression

```
auto x = 1 | 2;

inspect (x) {
  ^(1 | 2): std::cout << "1 | 2\n";
  __: std::cout << "otherwise\n";
}

// prints: "1 | 2"
```

Declaration Form

Refutability

Refutable: Pattern **can** fail to match for some value.

- `101`
- `[x, 0]`
- `<int> x`

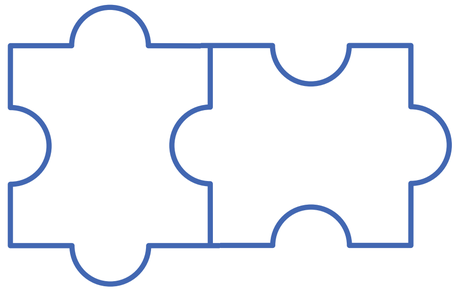
Irrefutable: Pattern **cannot** fail to match for any value.

- `_`
- `id`
- `[x, y]`

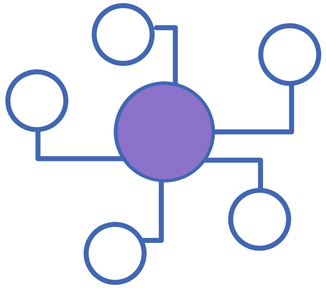
value

pattern

variable



Match

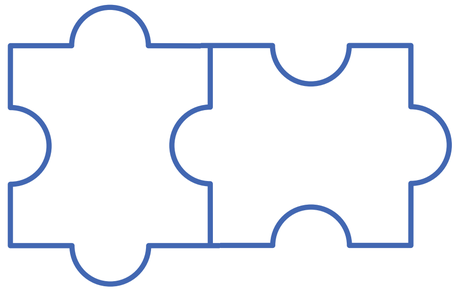


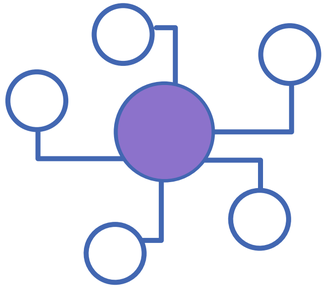
Bind

value

pattern

variable





Match

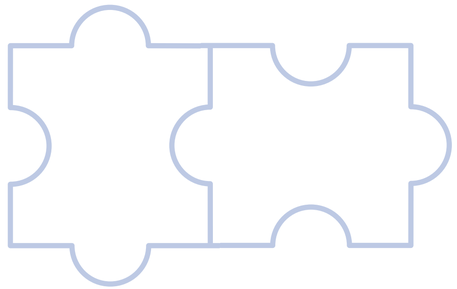
Bind

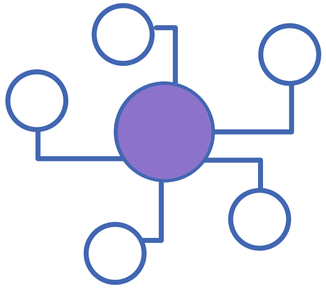
Refutable

value

pattern

variable



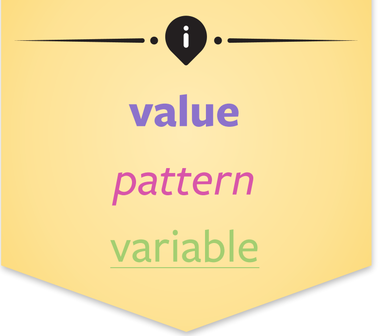


Match

Bind

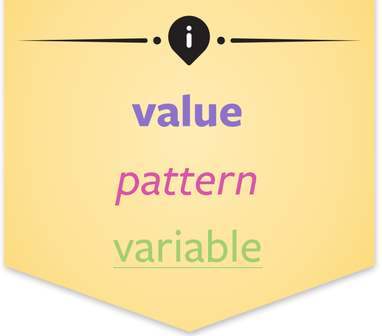
Irrefutable

Declaration Form



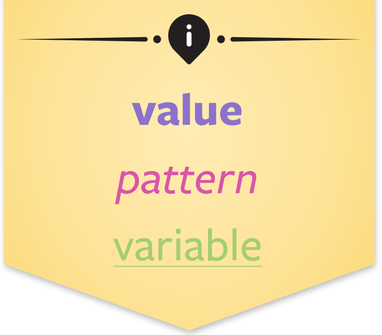

```
auto irrefutable-pattern = expr;
```

Variable Declaration



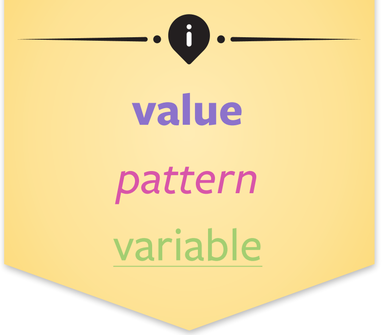

```
auto x = expr;
```

Structured Bindings (1)




```
auto [x, y] = expr;  // id-only structured bindings
```

Structured Bindings (2)




```
// field extracting structured bindings
auto [.x: x, .z: z] = expr;
```

Thank you!

Michael Park

mcypark@gmail.com

@mcypark