A Bite of Pattern Matching in C++

Bowen Fu

Pattern Matching?

Dispatch on Value

```
match meadow.count_rabbits() {
    0 => {}  // nothing to say
    1 => println!("A rabbit is nosing around in the clover."),
    n => println!("There are {} rabbits hopping about in the meadow", n)
}
```

Dispatch on Type

```
fn rough_time_to_english(rt: RoughTime) -> String {
    match rt {
        RoughTime::InThePast(units, count) =>
            format!("{} {} ago", count, units.plural()),
        RoughTime::JustNow =>
            format!("just now"),
        RoughTime::InTheFuture(units, count) =>
            format!("{} {} from now", count, units.plural())
    }
}
```

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}
```

Dispatch on Type

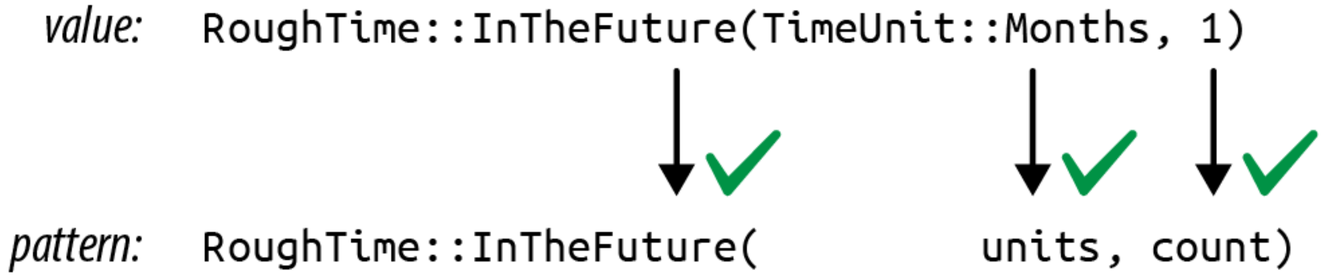
```
fn rough_time_to_english(rt: RoughTime) -> String {
    match rt {
        RoughTime::InThePast(units, count) =>
            format!("{} {} ago", count, units.plural()),
        RoughTime::JustNow =>
            format!("just now"),
        RoughTime::InTheFuture(units, count) =>
            format!("{} {} from now", count, units.plural())
    }
}
```

value: `RoughTime::InTheFuture(TimeUnit::Months, 1)`

pattern: `RoughTime::InThePast(units, count)`

Dispatch on Type

```
fn rough_time_to_english(rt: RoughTime) -> String {
    match rt {
        RoughTime::InThePast(units, count) =>
            format!("{} {} ago", count, units.plural()),
        RoughTime::JustNow =>
            format!("just now"),
        RoughTime::InTheFuture(units, count) =>
            format!("{} {} from now", count, units.plural())
    }
}
```

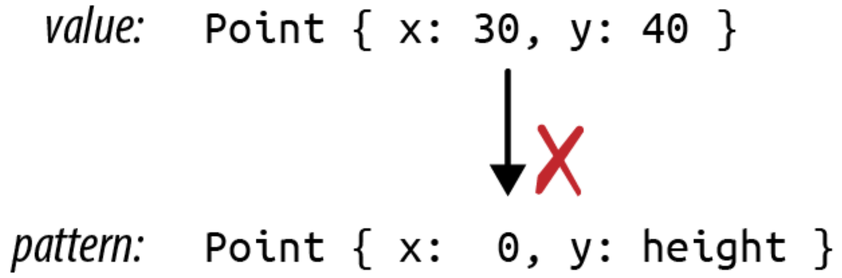


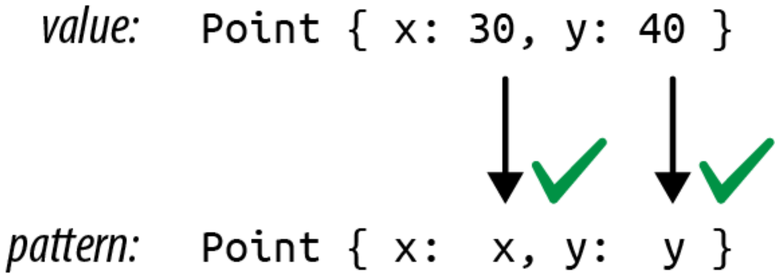
Multiple Dispatch

```
fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;
    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "at the origin",
        (_, Equal) => "on the x axis",
        (Equal, _) => "on the y axis",
        (Greater, Greater) => "in the first quadrant",
        (Less, Greater) => "in the second quadrant",
        _ => "somewhere else"
    }
}
```

Inside Struct

```
match balloon.location {
    Point { x: 0, y: height } =>
        println!("straight up {} meters", height),
    Point { x: x, y: y } =>
        println!("at ({}m, {}m)", x, y)
}
```





C++ Pattern Matching Proposal P1371R3

Matching Literals

Before

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

After

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

Matching Tuples

Before

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

After

```
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; }
};
```

Matching Variants

Before

```
struct visitor {
  void operator()(int i) const {
    os << "got int: " << i;
  }
  void operator()(float f) const {
    os << "got float: " << f;
  }
  std::ostream& os;
};
std::visit(visitor{strm}, v);
```

After

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

Matching Polymorphic Types

Before

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

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

After

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

A Patch: Case Pattern

```
enum Color { Red, Green, Blue };
Color color = /* ... */;

inspect (color) {
    case Red =>   // ...
    case Green => // ...
//        ^^^^^ id-expression
    case Blue =>  // ...
//  ^^^^^^^^^ case pattern
};
```

```
static constexpr int zero = 0;
int v = /* ... */;

inspect (v) {
    case zero => { std::cout << "got zero"; }
//       ^^^^ id-expression
    case 1 => { std::cout << "got one"; }
//       ^ expression pattern
    case 2 => { std::cout << "got two"; }
//  ^^^^^^ case pattern
};
```

Types of Patterns

- **Primary Patterns**
  - Wildcard Pattern
    - _
  - Identifier Pattern
    - x, y, z
  - Expression/Literal Pattern
    - 1, "1"
- **Compound Patterns**
  - Structured Binding Pattern
    - [x, 1]
  - Alternative Pattern
    - <int>
  - Case Pattern
    - case x
  - ...

Without Pattern Matching

```cpp
template <typename T>
void Node<T>::balance() {
  if (color != Black) return;
  if (lhs && lhs->color == Red) {
    if (const auto& lhs_lhs = lhs->lhs; lhs_lhs && lhs_lhs->color == Red) {
      // left-left case
      //
      //        (Black) z                 (Red) y
      //         /     \                 /      \
      //     (Red) y    d         (Black) x  (Black) z
      //     /      \       ->     /   \      /   \
      //  (Red) x    c             a     b    c     d
      //   /    \
      //  a      b
      *this = Node{
          Red,
          std::make_shared<Node>(Black, lhs_lhs->lhs, lhs_lhs->value, lhs_lhs->rhs),
          lhs->value,
          std::make_shared<Node>(Black, lhs->rhs, value, rhs)};
      return;
    }
    if (const auto& lhs_rhs = lhs->rhs; lhs_rhs && lhs_rhs->color == Red) {
      *this = Node{  // left-right case
          Red,
          std::make_shared<Node>(Black, lhs->lhs, lhs->value, lhs_rhs->lhs),
          lhs_rhs->value,
          std::make_shared<Node>(Black, lhs_rhs->rhs, value, rhs)};
      return;
    }
  }
  if (rhs && rhs->color == Red) {
    if (const auto& rhs_lhs = rhs->lhs; rhs_lhs && rhs_lhs->color == Red) {
      *this = Node{  // right-left case
          Red,
          std::make_shared<Node>(Black, lhs, value, rhs_lhs->lhs),
          rhs_lhs->value,
          std::make_shared<Node>(Black, rhs_lhs->rhs, rhs->value, rhs->rhs)};
      return;
    }
    if (const auto& rhs_rhs = rhs->rhs; rhs_rhs && rhs_rhs->color == Red) {
      *this = Node{  // right-right case
          Red,
          std::make_shared<Node>(Black, lhs, value, rhs->lhs),
          rhs->value,
          std::make_shared<Node>(Black, rhs_rhs->lhs, rhs_rhs->value, rhs_rhs->rhs)};
      return;
    }
```

With Pattern Matching

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

C++ Pattern Matching Library match(it)

https://github.com/BowenFu/matchit.cpp

Matching Single Value

```cpp
#include "matchit.h"

constexpr int32_t factorial(int32_t n)
{
    using namespace matchit;
    assert(n >= 0);
    return match(n)(
        pattern | 0 = expr(1),
        pattern | _ = [n] { return n * factorial(n - 1); }
    );
}
```

Matching Multiple Values

```
#include "matchit.h"

constexpr int32_t gcd(int32_t a, int32_t b)
{
    using namespace matchit;
    return match(a, b)(
        pattern | ds(_, 0) = [&] { return a >= 0 ? a : -a; },
        pattern | _        = [&] { return gcd(b, a%b); }
    );
}

static_assert(gcd(12, 6) == 6);
```

Using Variables

```
#include "matchit.h"
#include <map>

template <typename Map, typename Key>
constexpr bool contains(Map const& map, Key const& key)
{
    using namespace matchit;
    return match(map.find(key))(
        pattern | map.end() = expr(false),
        pattern | _         = expr(true)
    );
}
```

Predicate Pattern

```cpp
#include "matchit.h"

constexpr double relu(double value)
{
    return match(value)(
        pattern | (_ >= 0) = expr(value),
        pattern | _        = expr(0));
}

static_assert(relu(5) == 5);
static_assert(relu(-5) == 0);
```

Multiple Possibilities

```cpp
#include "matchit.h"

constexpr bool isValid(int32_t n)
{
    using namespace matchit;
    return match(n)(
        pattern | or_(1, 3, 5) = expr(true),
        pattern | _            = expr(false)
    );
}

static_assert(isValid(5));
static_assert(!isValid(6));
```

Applying Transformations

```cpp
#include "matchit.h"

constexpr bool isLarge(double value)
{
    using namespace matchit;
    return match(value)(
        pattern | app(_ * _, _ > 1000) = expr(true),
        pattern | _                    = expr(false)
    );
}

// app with projection returning scalar types is supported by constexpr match.
static_assert(isLarge(100));
```

Identifier Pattern

```
#include <iostream>
#include "matchit.h"

bool checkAndlogLarge(double value)
{
    using namespace matchit;
    Id<double> s;
    return match(value)(
        pattern | app(_ * _, s.at(_ > 1000)) = [&] {
                std::cout << value << "^2 = " << *s << " > 1000!" << std::endl;
                return true; },
        pattern | _ = expr(false));
}
```

Identifier Pattern

```cpp
#include "matchit.h"

constexpr bool symmetric(std::array<int32_t, 5> const& arr)
{
    using namespace matchit;
    Id<int32_t> i, j;
    return match(arr)(
        pattern | ds(i, j, _, j, i) = expr(true),
        pattern | _                 = expr(false)
    );
}

static_assert(symmetric(std::array<int32_t, 5>{5, 0, 3, 7, 10}) == false);
static_assert(symmetric(std::array<int32_t, 5>{5, 0, 3, 0, 5}) == true);
static_assert(symmetric(std::array<int32_t, 5>{5, 1, 3, 0, 5}) == false);
```

Destructuring Struct

```
// Another option to destructure your struct / class.
constexpr auto dsByMember(DummyStruct const&v)
{
    using namespace matchit;
    // compose patterns for destructuring struct DummyStruct.
    constexpr auto dsA = dsVia(&DummyStruct::size, &DummyStruct::name);
    Id<char const*> name;
    return match(v)(
        pattern | dsA(2, name) = expr(name),
        pattern | _ = expr("not matched")
    );
};

static_assert(dsByMember(DummyStruct{1, "123"}) == std::string_view{"not matched"});
static_assert(dsByMember(DummyStruct{2, "123"}) == std::string_view{"123"});
```

Pattern Guard

```cpp
#include <array>
#include "matchit.h"

constexpr bool sumIs(std::array<int32_t, 2> const& arr, int32_t s)
{
    using namespace matchit;
    Id<int32_t> i, j;
    return match(arr)(
        pattern | ds(i, j) | when(i + j == s) = expr(true),
        pattern | _                           = expr(false));
}

static_assert(sumIs(std::array<int32_t, 2>{5, 6}, 11));
```

Ooo Pattern

```cpp
#include <array>
#include "matchit.h"

template <typename Tuple>
constexpr int32_t detectTuplePattern(Tuple const& tuple)
{
    using namespace matchit;
    return match(tuple)
    (
        pattern | ds(2, ooo, 2)  = expr(4),
        pattern | ds(2, ooo   )  = expr(3),
        pattern | ds(ooo, 2   )  = expr(2),
        pattern | ds(ooo      )  = expr(1)
    );
}

static_assert(detectTuplePattern(std::make_tuple(2, 3, 5, 7, 2)) == 4);
```

Binding to Ooo Pattern

```
template <typename Range>
constexpr bool recursiveSymmetric(Range const &range)
{
    Id<int32_t> i;
    Id<SubrangeT<Range const>> subrange;
    return match(range)(
        pattern | ds(i, subrange.at(ooo), i) = [&] { return recursiveSymmetric(*subrange); },
        pattern | ds(_, ooo, _)              = expr(false),
        pattern | _                          = expr(true)
    );
```

Some / None Pattern

```cpp
#include "matchit.h"

template <typename T>
constexpr auto square(std::optional<T> const& t)
{
    using namespace matchit;
    Id<T> id;
    return match(t)(
        pattern | some(id) = id * id,
        pattern | none     = expr(0));
}
constexpr auto x = std::make_optional(5);
static_assert(square(x) == 25);
```

Patterns are Composable

```
template <typename T>
constexpr auto cast = [](auto && input) {
    return static_cast<T>(input);
};

constexpr auto deref = [](auto &&x) { return *x; };

constexpr auto some = [](auto const pat) {
    return and_(app(cast<bool>, true), app(deref, pat));
};

constexpr auto none = app(cast<bool>, false);
```

As Pattern

```cpp
#include "matchit.h"
template <typename T>
constexpr auto getClassName(T const& v)
{
    using namespace matchit;
    return match(v)(
        pattern | as<char const*>(_) = expr("chars"),
        pattern | as<int32_t>(_)     = expr("int32_t")
    );
}

constexpr std::variant<int32_t, char const*> v = 123;
static_assert(getClassName(v) == std::string_view{"int32_t"});
```

As Pattern

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

auto getClassName(Shape const &s)
{
    return match(s)(
        pattern | as<Circle>(_) = expr("Circle"),
        pattern | as<Square>(_) = expr("Square")
    );
}
```

Patterns are Composable

```
template <typename T>
constexpr AsPointer<T> asPointer;

template <typename T>
constexpr auto as = [](auto const pat) {
    return app(asPointer<T>, some(pat));
};
```

Three Mechanisms

- Every powerful language has three mechanisms for accomplishing this:
  - **primitive expressions**, which represent the simplest entities the language is concerned with
  - **means of combination**, by which compound elements are built from simpler ones
  - and **means of abstraction**, by which compound elements can be named and manipulated as units.

Three Mechanisms

- Inside `match(it)`
  - **primitive expressions** include `Expression Pattern`, `Wildcard Pattern`, `Predicate Pattern`, `Identifier Pattern`, `Match Guard`, `Ooo Pattern`
  - **Combinators** include `Or Pattern`, `And Pattern`, `Not Pattern`, `App Pattern`, `Destructure Pattern`, `At Pattern`
  - **Aliasing compound patterns** and **defining functions / function objects that return patterns** are the **means of abstraction**.

With P1371R3

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

With match(it)

```
template <typename T>
void Node<T>::balance()
{
  using namespace matchit;

  …

  Id<std::shared_ptr<Node<T>>> a, b, c, d;
  Id<T> x, y, z;
  Id<Node> self;
  *this = match(*this)(
     pattern | blackN(some(redN(some(redN(a, x, b)), y, c)), z, d) // left-left case
      = [&]
      { return Node{Red, std::make_shared<Node>(Black, *a, *x, *b), *y,
                    std::make_shared<Node>(Black, *c, *z, *d)}; },
      pattern | blackN(some(redN(a, x, some(redN(b, y, c)))), z, d) // left-right case
      = [&]
      { return Node{Red, std::make_shared<Node>(Black, *a, *x, *b), *y,
                    std::make_shared<Node>(Black, *c, *z, *d)}; },
      pattern | blackN(a, x, some(redN(some(redN(b, y, c)), z, d))) // right-left case
      = [&]
      { return Node{Red, std::make_shared<Node>(Black, *a, *x, *b), *y,
                    std::make_shared<Node>(Black, *c, *z, *d)}; },
      pattern | blackN(a, x, some(redN(b, y, some(redN(c, z, d))))) // right-right case
      = [&]
      { return Node{Red, std::make_shared<Node>(Black, *a, *x, *b), *y,
                    std::make_shared<Node>(Black, *c, *z, *d)}; },
      pattern | self = expr(self) // do nothing
  );
}
```

With match(it)

```
template <typename T>
void Node<T>::balance()
{
  using namespace matchit;

  constexpr auto dsN = [](auto &&color, auto &&lhs, auto &&value, auto &&rhs)
  {
    return and_(app(&Node<T>::color, color), app(&Node<T>::lhs, lhs),
                app(&Node<T>::value, value), app(&Node<T>::rhs, rhs));
  };

  constexpr auto blackN = [dsN](auto &&lhs, auto &&value, auto &&rhs)
  {
    return dsN(Black, lhs, value, rhs);
  };

  constexpr auto redN = [dsN](auto &&lhs, auto &&value, auto &&rhs)
  {
    return dsN(Red, lhs, value, rhs);
  };

  …
}
```

Sample for Symbolic Computation

```
ExprPtr operator<(ExprPtr const &lhs, ExprPtr const &rhs)
{
    if (equal(lhs, rhs))
    {
        return false_;
    }
    return match(lhs, rhs)(
        pattern | ds(asDouble, asDouble) = [&]
        { return evald(lhs) < evald(rhs) ? true_ : false_; },
        pattern | _ = [&]
        { return makeSharedExprPtr(Relational{RelationalKind::kLESS, lhs, rhs}); });
}
```

Sample for Symbolic Computation

```
// the basic quotient transformation
ExprPtr operator/(ExprPtr const &lhs, ExprPtr const &rhs)
{
    using namespace matchit;
    Id<Integer> il, ir;
    return match(*lhs, *rhs)(
        // clang-format off
        pattern | ds(as<Integer>(il), as<Integer>(ir)) = [&] { return
simplifyRational(fraction(*il, *ir)); },
        // basic identity transformation
        pattern | ds(_, as<Integer>(0)) = [&] { throw std::runtime_error{"undefined!"};
return 0_i; },
        pattern | ds(as<Integer>(0), _) = expr(0_i),
        pattern | ds(_, as<Integer>(1)) = expr(lhs),
        pattern | _                                  = expr(lhs * (rhs ^ -1_i))
    // clang-format on
);
```

New Proposal: P2392 R1

Pattern matching using is and as

Pattern Matching Using Is and As

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

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

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

Pattern Matching Using Is and As

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

Q&A