Pattern Matching

Document #:	D1260R1
Date:	2018-10-09
Project:	Programming Language C++ Evolution

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

Contents

1 Revision History

- R1
  - Removed the binding (@) pattern
  - Added an **Examples** section
- R0 — Initial Draft

2 Introduction

As algebraic data types gain better support in C++ with facilities such as `tuple` and `variant`, the importance of mechanisms to interact with them have increased. While mechanisms such as `apply` and `visit` have been added, their usage is quite complex and limited even for simple cases. Pattern matching is a widely adopted mechanism across many programming languages to interact with algebraic data types that can help greatly simplify C++. Examples of programming languages include text-based languages such as SNOBOL back in the 1960s, functional languages such as Haskell and OCaml, and "mainstream" languages such as Scala, Swift, and Rust.

Inspired by P0095 [5] — which proposed pattern matching and language-level variant simultaneously — this paper explores a possible direction for pattern matching only, and does not address language-level variant design. This is in correspondence with a straw poll from Kona 2015, which encouraged exploration of a full solution for pattern matching. SF: 16, WF: 6, N: 5, WA: 1, SA: 0.

3 Motivation and Scope

Virtually every program involves branching on some predicates applied to a value and conditionally binding names to some of its components for use in subsequent logic. Today, C++ provides two types of selection

statements: the `if` statement and the `switch` statement.

Since `switch` statements can only operate on a *single* integral value and `if` statements operate on an *arbitrarily* complex boolean expression, there is a significant gap between the two constructs even in inspection of the "vocabulary types" provided by the standard library.

In C++17, structured binding declarations [9] introduced the ability to concisely bind names to components of `tuple`-like values. The proposed direction of this paper aims to naturally extend this notion by performing **structured inspection** prior to forming the **structured bindings** with a third selection statement: the `inspect` statement. The goal of the `inspect` statement is to bridge the gap between `switch` and `if` statements with a **declarative**, **structured**, **cohesive**, and **composable** mechanism.

4 Before/After Comparisons

4.1 Matching Integrals

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";
}
```

4.2 Matching Strings

Before

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

After

```
inspect (s) {
  "foo": std::cout << "got foo";
  "bar": std::cout << "got bar";
  _: std::cout << "don't care";
}
```

4.3 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;
}
```

4.4 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;
}
```

4.5 Matching Polymorphic Types

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

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) {
  inspect (shape) {
    (as<Circle>? [r]): return 3.14 * r * r;
    (as<Rectangle>? [w, h]): return w * h;
  }
}
```

4.6 Evaluating Expressions

```
struct Expr;
struct Neg { std::shared_ptr<Expr> expr; };
struct Add { std::shared_ptr<Expr> lhs, rhs; };
struct Mul { std::shared_ptr<Expr> lhs, rhs; };
struct Expr : std::variant<int, Neg, Add, Mul> { using variant::variant; };

namespace std {
    template <>
    struct variant_size<Expr> : variant_size<Expr::variant> {};

    template <std::size_t I>
    struct variant_alternative<I, Expr> : variant_alternative<I, Expr::variant> {};
}
```

Before

```
int eval(const Expr& expr) {
  struct visitor {
    int operator()(int i) const {
      return i;
    }
    int operator()(const Neg& n) const {
      return -eval(*n.expr);
    int operator()(const Add& a) const {
      return eval(*a.lhs) + eval(*a.rhs);
    }
    int operator()(const Mul& m) const {
      return eval(*m.lhs) * eval(*m.rhs);
    }
  };
  return std::visit(visitor{}, expr);
}
```

After

```
int eval(const Expr& expr) {
  inspect (expr) {
    <int> i: return i;
    <Neg> [e]: return -eval(*e);
    <Add> [l, r]: return eval(*l) + eval(*r);
    <Mul> [l, r]: return eval(*l) * eval(*r);
  }
}
```

5 Design Overview

5.1 Basic Syntax

`inspect` `constexpr`$_{opt}$ `(` *init-statement*$_{opt}$ *condition* `)` `{`
pattern *guard*$_{opt}$ `:` *statement*
pattern *guard*$_{opt}$ `:` *statement*
...
`}`

guard:
`if` `(` *expression* `)`

5.2 Basic Model

Within the parentheses, the `inspect` statement is equivalent to `switch` and `if` statements except that no conversion nor promotion takes place in evaluating the value of its condition.

When the `inspect` statement is executed, its condition is evaluated and matched in order (first match semantics) against each pattern. If a pattern successfully matches the value of the condition and the boolean expression in the guard evaluates to `true` (or if there is no guard at all), control is passed to the statement following the matched pattern label. If the guard expression evaluates to `false`, control flows to the subsequent pattern. If no pattern matches, none of the statements are executed.

5.3 Types of Patterns

5.3.1 Primary Patterns

5.3.1.1 Wildcard Pattern

The wildcard pattern has the form:

`_`

and matches any value `v`.

```
int v = /* ... */;

inspect (v) {
    _: std::cout << "ignored";
//  ^ wildcard pattern
}
```

[*Note:* Even though `_` is a valid identifier, it does not introduce a name.]

// TODO: Refer to Jeffrey's paper.

5.3.1.2 Identifier Pattern

The identifier pattern has the form:

identifier

and matches any value `v`. The introduced name behaves as an lvalue referring to `v`, and is in scope from its point of declaration until the end of the statement following the pattern label.

```
int v = /* ... */;

inspect (v) {
    x: std::cout << x;
//  ^ identifier pattern
}
```

[*Note:* If the identifier pattern is used as a top-level pattern, it has the same syntax as a `goto` label.]

5.3.1.3 Constant Pattern

The constant pattern has the form:

constant expression

and matches value `v` if a call to member `c.match(v)` or else a non-member ADL-only `match(c, v)` is contextually convertible to `bool` and evaluates to `true` where `c` is the *constant expression*.

The following is the default behavior of `match(x, y)`.

```
template <typename T, typename U>
constexpr auto match(T&& lhs, U&& rhs)
        -> decltype(std::forward<T>(lhs) == std::forward<U>(rhs)) {
    return std::forward<T>(lhs) == std::forward<U>(rhs);
}
```

```
int v = /* ... */;

inspect (v) {
    0: std::cout << "got zero";
    1: std::cout << "got one";
//  ^ constant pattern
}
```

[*Note:* `(id)` is needed to disambiguate with the identifier pattern.]

```
static constexpr int zero = 0, one = 1;
int v = /* ... */;

inspect (v) {
    (zero): std::cout << "got zero";
    (one): std::cout << "got one";
//  ^^^^^ constant pattern
}
```

5.3.2 Compound Patterns

5.3.2.1 Structured Binding Pattern

The structured binding pattern has the form:

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

and matches value `v` if each *pattern*$_i$ matches the i^{th} component of `v`. The components of `v` are given by the structured binding declaration: `auto&& [__e0, __e1, ..., __eN] = v;` where each `__e`$_i$ are unique exposition-only identifiers.

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

inspect (p) {
    [0, 0]: std::cout << "on origin";
    [0, y]: std::cout << "on y-axis";
//      ^ identifier pattern
    [x, 0]: std::cout << "on x-axis";
//      ^ constant pattern
    [x, y]: std::cout << x << ',' << y;
//  ^^^^^^ structured binding pattern
}
```

5.3.2.2 Alternative Pattern

The alternative pattern has the form:

`<` `auto` `>` *pattern*
`<` *concept* `>` *pattern*
`<` *type* `>` *pattern*
`<` *constant expression* `>` *pattern*

Let `v` be the value being matched and `V` be `std::remove_cvref_t<decltype(v)>`.
Let `Alt` be the entity inside the angle brackets.

If `std::variant_size_v<V>` is well-formed and evaluates to an integral, the alternative pattern matches `v` if `Alt` is compatible with the current index of `v` and *pattern* matches the active alternative of `v`.

Let `I` be the current index of `v` given by a member `v.index()` or else a non-member ADL-only `index(v)`. The active alternative of `v` is given by `std::variant_alternative_t<I, V>&` initialized by a member `v.get<I>()` or else a non-member ADL-only `get<I>(v)`.

`Alt` is compatible with `I` if one of the following four cases is true:

- `Alt` is `auto`
- `Alt` is a *concept* and `std::variant_alternative_t<I, V>` satisfies the *concept*.
- `Alt` is a *type* and `std::is_same_v<Alt, std::variant_alternative_t<I, V>>` is `true`
- `Alt` is a *constant expression* that can be used in a `switch` and is the same value as `I`.

Before	After

`std::visit([&](auto&& x) {`
`  strm << "got auto: " << x;`
`}, v);`
	`inspect (v) {`
	`  <auto> x: strm << "got auto: " << x;`
	`}`

Before

```
std::visit([&](auto&& x) {
  using X = std::remove_cvref_t<decltype(x)>;
  if constexpr (C1<X>()) {
    strm << "got C1: " << x;
  } else if constexpr (C2<X>()) {
    strm << "got C2: " << x;
  }
}, v);
```

After

```
inspect (v) {
  <C1> c1: strm << "got C1: " << c1;
  <C2> c2: strm << "got C2: " << c2;
}
```

Before

```
std::visit([&](auto&& x) {
  using X = std::remove_cvref_t<decltype(x)>;
  if constexpr (std::is_same_v<int, X>) {
    strm << "got int: " << x;
  } else if constexpr (
      std::is_same_v<float, X>) {
    strm << "got float: " << x;
  }
}, v);
```

After

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

Before

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

std::visit([&](int x) {
  strm << "got int: " << x;
}, v);
```

After

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

inspect (v) {
  <int> x: strm << "got int: " << x;
}
```

Before

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

std::visit([&](auto&& x) {
  switch (v.index()) {
    case 0: {
      strm << "got first: " << x;
      break;
    }
    case 1: {
      strm << "got second: " << x;
      break;
    }
  }
}, v);
```

After

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

inspect (v) {
  <0> x: strm << "got first: " << x;
  <1> x: strm << "got second: " << x;
}
```

5.3.2.3 Extractor Pattern

The extractor pattern has the following two forms:

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

Let `c` be the *constant expression*. The first form matches value `v` if *pattern* matches `e` where `e` is the result of a call to member `c.extract(v)` or else a non-member ADL-only `extract(c, v)`.

For second form, let `e` be the result of a call to member `c.try_extract(v)` or else a non-member ADL-only `try_extract(c, v)`. It matches value `v` if `e` is contextually convertible to `bool`, evaluates to `true`, and *pattern* matches `*e`.

```
struct {
    std::optional<std::array<std::string_view, 2>>
    try_extract(std::string_view sv) const;
} email;

struct {
    std::optional<std::array<std::string_view, 3>>
    try_extract(std::string_view sv) const;
} phone_number;

inspect (s) {
    (email? [address, domain]): std::cout << "got an email";
    (phone_number? ["415", _, _]): std::cout << "got a San Francisco phone number";
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ extractor pattern
}
```

5.3.2.4 As Pattern

The as pattern is a special instance of the extractor pattern, and behaves as:

```
template <typename Derived>
struct As {
    template <typename Base>
    auto* try_extract(Base& base) const {
        static_assert(std::is_polymophic_v<Base>);
        static_assert(std::is_convertible_v<Derived*, Base*>,
                      "cross-casts are not allowed.");
        using R = /* `Derived` with the same _cv_-qualification as `Base` */;
        return dynamic_cast<R*>(&base);
    }
};

template <typename Derived>
inline constexpr As<Derived> as;
```

While this is a possible library implementation, it will likely benefit from being implemented as a compiler intrinsic for optimization opportunities.

N3449 [8] describes techniques involving vtable pointer caching and hash conflict minimization that are implemented in the Mach7 [6] library, but also mentions further opportunities available for a compiler solution.

Given the following definition of a `Shape` class hierarchy:

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

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

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) {
  inspect (shape) {
    (as<Circle>? [r]): return 3.14 * r * r;
    (as<Rectangle>? [w, h]): return w * h;
//  ^^^^^^^^^^^^^^^^^^^^^^^^ as pattern
  }
}
```

5.4 Pattern Guard

The pattern guard has the form:

`if` (*expression*)

Let `e` be the result of *expression* contextually converted to `bool`. If `e` is `true`, control is passed to the corresponding statement. Otherwise, control flows to the subsequent pattern.

The pattern guard allows to perform complex tests that cannot be performed within the *pattern*. For example, performing tests across multiple bindings:

```
inspect (p) {
    [x, y] if (test(x, y)): std::cout << x << ',' << y << " passed";
//         ^^^^^^^^^^^^^^^ pattern guard
}
```

This also diminishes the desire for fall-through semantics within the statements, an unpopular feature even in `switch` statements. For the reified semantics of the pattern guard, consider the following snippet:

```
switch (x) {
    case c1: if (cond1) { stmt1; break; } [[fallthrough]]
    case c2: if (cond2) { stmt2; break; } [[fallthrough]]
}
```

5.5 `inspect constexpr`

Note that every *pattern* is able to determine whether it matches value `v` as a boolean expression in isolation.

Let `MATCHES` be the condition for which a *pattern* matches a value `v`. Ignoring any potential optimization opportunities, we're able to perform the following transformation:

inspect	if

`inspect (v) {`
`  pattern1 if (cond1): stmt1`
`  pattern2: stmt2`
`  // ...`
`}`
	`if (MATCHES(pattern1. v) && cond1) stmt1`
	`else if (MATCHES(pattern2, v)) stmt2`
	`// ...`

`inspect constexpr` is then formulated by applying `constexpr` to every `if` branch.

inspect constexpr	if constexpr

`inspect constexpr (v) {`
`  pattern1 if (cond1): stmt1`
`  pattern2: stmt2`
`  // ...`
`}`
	`if constexpr (MATCHES(pattern1, v) && cond1) stmt1`
	`else if constexpr (MATCHES(pattern2, v)) stmt2`
	`// ...`

5.6 Exhaustiveness and Usefulness

`inspect` can be declared `[[strict]]` for implementation-defined exhaustiveness and usefulness checking.

Exhaustiveness means that all values of the type of the value being matched is handled by at least one of the cases. For example, having a `_:` case makes any `inspect` statement exhaustive.

Usefulness means that every case handles at least one value of the type of the value being matched. For example, any case that comes after a `_:` case would be useless.

Warnings for pattern matching [2] discusses and outlines an algorithm for exhaustiveness and usefulness for OCaml, and is the algorithm used by Rust.

6 Proposed Wording

The following is the beginning of an attempt at a syntactic structure.

Add to **§8.4 [stmt.select]** of ...

1 Selection statements choose one of several flows of control.

selection-statement:
`if constexpr`$_{opt}$ `(` *init-statement*$_{opt}$ *condition* `)` *statement*
`if constexpr`$_{opt}$ `(` *init-statement*$_{opt}$ *condition* `)` *statement* `else` *statement*
`switch (` *init-statement*$_{opt}$ *condition* `)` *statement*
`inspect constexpr`$_{opt}$ `(` *init-statement*$_{opt}$ *condition* `) {` *inspect-case-seq* `}`

inspect-case-seq:
inspect-case
inspect-case-seq inspect-case

inspect-case:
    attribute-specifier-seq$_{opt}$ *inspect-pattern* *inspect-guard*$_{opt}$ `:` *statement*

inspect-pattern:
    wildcard-pattern
    identifier-pattern
    constant-pattern
    structured-binding-pattern
    alternative-pattern
    binding-pattern
    extractor-pattern

inspect-guard:
    `if (` *expression* `)`

7 Design Decisions

7.1 Extending Structured Bindings Declaration

The design is intended to be consistent and to naturally extend the notions introduced by structured bindings. That is, The subobjects are **referred** to rather than being assigned into new variables.

7.2 `inspect` rather than `switch`

This proposal introduces a new `inspect` statement rather than trying to extend the `switch` statement. P0095R0 [4] had proposed extending `switch` and received feedback to "leave `switch` alone" in Kona 2015.

The following are some of the reasons considered:

- `switch` allows the `case` labels to appear **anywhere**, which hinders the goal of pattern matching in providing **structured** inspection.
- The fall-through semantics of `switch` generally results in `break` being attached to every case, and is known to be error-prone.
- `switch` is purposely restricted to integrals for **guaranteed** efficiency. The primary goal of pattern matching in this paper is expressiveness while being at least as efficient as the naively hand-written code.

7.3 First Match rather than Best Match

The proposed matching algorithm has first match semantics. The choice of first match is mainly due to complexity. Our overload resolution rules for function declarations are extremely complex and is often a mystery.

Best match via overload resolution for function declarations are absolutely necessary due to the non-local and unordered nature of declarations. That is, function declarations live in different files and get pulled in via mechanisms such as `#include` and `using` declarations, and there is no defined order of declarations like Haskell does, for example. If function dispatching depended on the order of `#include` and/or `using` declarations being pulled in from hundreds of files, it would be a complete disaster.

Pattern matching on the other hand do not have this problem because the construct is local and ordered in nature. That is, all of the candidate patterns appear locally within `inspect (x) { /* ... */ }` which

cannot span across multiple files, and appear in a specified order. Note that this is consistent with `try/catch` for the same reasons: locality and order.

Consider also the amount of limitations we face in overload resolution due to the opacity of user-defined types. `T*` is related to `unique_ptr<T>` as it is to `vector<T>` as far as the type system is concerned. This limitation will likely be even bigger in a pattern matching context with the amount of customization points available for user-defined behavior.

7.4 Statement rather than Expression

This paper diverges from P0095 [5] in that it proposes to add `inspect` as a statement only rather than trying to double as a statement and an expression. The main reason here is that the semantic differences between the statement and expression forms are not trivial.

- In the situation where none of the cases match, the statement form simply skips over the entire statement à la `switch`, whereas the expression form throws an exception since it is required to yield a value.
- Resulting type of the statement form of `inspect` within an "immediately-invoked-lambda" is required to be explicitly specified, or is determined by the first `return` statement. In contrast, the expression form will probably need to use `std::common_type_t<Ts...>` where `Ts...` are types of `N` expressions to be consistent with the ternary operator.

While an expression form of `inspect` would be useful, the author believes that it can and should be introduced later, with different enough syntax such as `x inspect {` $pattern_0$ `=>` $expression_0$`,` $pattern_1$ `=>` $expression_1$ `}`. The proposed syntax of the `inspect` statement in this paper consistent with every other statement in C++ today.

7.5 Language rather than Library

There are three popular pattern matching libraries for C++ today: Mach7 [6], MPark.Patterns [3], and `simple_match` [1].

While the libraries have been useful for gaining experience with interfaces and implementation, the issue of introducing identifiers, syntactic overhead of the patterns, and the reduced optimization opportunities justify support as a language feature from a usability standpoint.

7.6 User-defined Patterns

Many languages provide a wide array of patterns through various syntactic forms. While this is a potential direction for C++, it would mean that every new type of matching requires new syntax to be added to the language. This would result in a narrow set of types being supported through limited customization points.

User-defined patterns (via matchers and extractors) are supported in order to minimize the number of patterns with special syntax. The following are example user-defined patterns that commonly have special syntax in other languages.

User-defined Pattern	**Other Languages**
`any_of{1, 2, 3}`	`1 | 2 | 3`
`range{iota_view{1, 10}}`	`1..10`
`(both! [[x, 0], [0, y]])`	`[x, 0] & [0, y]`
`(at! [p, [x, y]])`	`p @ [x, y]`

// TODO: links.

8 Runtime Performance

The following are few of the optimizations that are worth noting.

8.0.1 Structured Binding Pattern

Structured binding patterns can be optimized by performing `switch` over the columns with the duplicates removed, rather than the naive approach of performing a comparison per element. This removes unnecessary duplicate comparisons that would be performed otherwise. This would likely require some wording around "comparison elision" in order to enable such optimizations.

8.0.2 Alternative Pattern

The sequence of alternative patterns can be executed in a `switch`.

8.0.3 Open Class Hierarchy

N3449 [8] describes techniques involving vtable pointer caching and hash conflict minimization that are implemented in the Mach7 [6] library, but also mentions further opportunities available for a compiler solution.

9 Examples

9.1 Predicate-based Discriminator

Short-string optimization using a **predicate** as a discriminator rather than an explicitly stored **value**. Adapted from the pattern matching presentation given by Bjarne Stroustrup at Urbana-Champaign 2015.

```
struct String {
    enum Storage { Local, Remote };

    int size;
    union {
        char local[32];
        struct { char *ptr; int unused_allocated_space; } remote;
    };

    // Predicate-based discriminator derived from `size`.
    Storage index() const { return size > sizeof(local) ? Remote : Local; }

    // Opt into Variant-Like protocol.
    template <Storage S>
    auto &&get() {
        if constexpr (S == Local) return local;
        else if constexpr (S == Remote) return remote;
```

```
    }

    char *data();
};

namespace std {
    // Opt into Variant-Like protocol.

    template <>
    struct variant_size<String> : std::integral_constant<std::size_t, 2> {};

    template <>
    struct variant_alternative<String::Local, String> {
        using type = decltype(String::local);
    };

    template <>
    struct variant_alternative<String::Remote, String> {
        using type = decltype(String::remote);
    };
}

char* String::data() {
    inspect (*this) {
        <Local> l: return l;
        <Remote> r: return r.ptr;
    }
    // switch (index()) {
    //     case Local: {
    //         std::variant_alternative_t<Local, String> &l = get<Local>();
    //         return l;
    //     }
    //     case Remote: {
    //         std::variant_alternative_t<Remote, String> &r = get<Remote>();
    //         return r.ptr;
    //     }
    // }
}
```

9.2 "Closed" Class Hierarchy

A class hierarchy can effectively be closed with an `enum` that maintains the list of its members, and provide efficient dispatching by opting into the Variant-Like protocol.

Some examples can be found in the LLVM project.

// TODO link

```
struct Shape { enum Kind { Circle, Rectangle } kind; };

struct Circle : Shape {
```

```
    Circle(int radius) : Shape{Shape::Kind::Circle}, radius(radius) {}

    int radius;
};

struct Rectangle : Shape {
    Rectangle(int width, int height)
        : Shape{Shape::Kind::Rectangle}, width(width), height(height) {}

    int width, height;
};

namespace std {
    template <>
    struct variant_size<Shape> : std::integral_constant<std::size_t, 2> {};

    template <>
    struct variant_alternative<Shape::Circle, Shape> { using type = Circle; };

    template <>
    struct variant_alternative<Shape::Rectangle, Shape> { using type = Rectangle; };
}

Shape::Kind index(const Shape& shape) { return shape.kind; }

template <Kind K>
auto&& get(const Shape& shape) {
    return static_cast<const std::variant_alternative_t<K, Shape>&>(shape);
}

int get_area(const Shape& shape) {
    inspect (shape) {
        <Circle> c: return 3.14 * c.radius * c.radius;
        <Rectangle> r: return r.width * r.height;
    }
    // switch (index(shape)) {
    //     case Shape::Circle: {
    //         const std::variant_alternative_t<Shape::Circle, Shape>& c =
    //             get<Shape::Circle>(shape);
    //         return 3.14 * c.radius * c.radius;
    //     }
    //     case Shape::Rectangle: {
    //         const std::variant_alternative_t<Shape::Rectangle, Shape>& r =
    //             get<Shape::Rectangle>(shape);
    //         return r.width * r.height;
    //     }
    // }
}
```

9.3 User-defined Matcher: `any_of`

The logical-or pattern in other languages is typically spelled $pattern_0$ `|` $pattern_1$ `|` `...` `|` $pattern_N$, and matches value `v` if any $pattern_i$ matches `v`.

This provides a restricted form (constant-only) of the logical-or pattern.

```
template <typename... Ts>
struct any_of : std::tuple<Ts...> {
    using tuple::tuple;

    template <typename U>
    bool match(const U& u) const {
        return std::apply([&](const auto&... xs) {
            return (... || xs == u);
        }, *this);
    }
};
```

```
int fib(int n) {
    inspect (n) {
        x if x < 0: return 0;
        any_of{1, 2}: return n;
        x: return fib(x - 1) + fib(x - 2);
    }
}
```

9.4 User-defined Matcher: `range`

The range pattern in other languages is typically spelled `first..last`, and matches `v` if `v` ∈ `[first, last)`.

This is a generalized form of the range pattern.

```
template <typename R>
struct range {
    R r;

    template <typename U>
    bool match(const U& u) const { return std::ranges::find(r, u) != r.end(); }

    // TODO: Add deduction guide.
};
```

Example using `std::ranges::iota_view` which covers the `first..last` case:

```
inspect (n) {
    range{std::ranges::iota_view{1, 10}}: {
        std::cout << n << " is in [1, 10).";
    }
    _: {
        std::cout << n << " is not in [1, 10).";
    }
}
```

9.5 User-defined Extractor: `both`

The logical-and pattern in other languages is typically spelled *pattern*$_0$ `&` *pattern*$_1$ `&` `...` `&` *pattern*$_N$, and matches `v` if all of *pattern*$_i$ matches `v`.

This extractor emulates binary logical-and with a `std::pair` where both elements are references to value `v`.

```
inline constexpr struct {
    template <typename U>
    std::pair<U&&, U&&> extract(U&& u) const {
        return {std::forward<U>(u), std::forward<U>(u)};
    }
} both;
```

```
inspect (v) {
    (both! [[x, 0], [0, y]]): // ...
}
```

9.6 User-defined Extractor: `at`

The binding pattern in other languages is typically spelled *identifier* `@` *pattern*, binds *identifier* to `v` and matches if *pattern* matches `v`. Note that this is a special case of the logical-and pattern (*pattern*$_0$ `&` *pattern*$_1$) where *pattern*$_0$ is an *identifier*. That is, *identifier* `&` *pattern* has the same semantics as *identifier* `@` *pattern*, which means we get `at` for free from `both` above.

```
inline constexpr at = both;
```

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

10 Future Work

10.1 Language Support for Variant

The design of this proposal also accounts for a potential language support for variant. It achieves this by keeping the alternative pattern flexible for new extensions via `<` *new_entity* `>` *pattern*.

Consider an extension to `union` that allows it to be tagged by an integral, and has proper lifetime management such that the active alternative need not be destroyed manually.

```
// `: type` specifies the type of the underlying tag value.
union U : int { char small[32]; std::vector<char> big; };
```

We could then allow `<` *qualified-id* `>` that refers to a `union` alternative to support pattern matching.

```
U u = /* ... */;

inspect (u) {
    <U::small> s: std::cout << s;
```

```
    <U::big> b: std::cout << b;
}
```

The main point is that whatever entity is introduced as the discriminator, the presented form of alternative pattern should be extendable to support it.

10.2 Patterns in range-based `for` loop

```
for (auto&& [0, y] : points) {
    // only operate on points on the y-axis.
}
```

Structured binding declaration is allowed in range-based `for` loop:

```
for (auto&& [x, y] : points) { /* ... */ }
```

The `[x, y]` part can also be a pattern of an `inspect` statement rather than a structured binding declaration.

Before	After

`for (auto&& p : points) {`
`  auto&& [x, y] = p;`
`  // ...`
`}`
	`for (auto&& p : points) {`
	`  inspect (p) {`
	`  [x, y]: // ...`
	`  }`
	`}`

With this model, allowing patterns directly in range-based `for` loop becomes natural.

Code	Expanded

`for (auto&& [0, y] : points) {`
`  // only points on the y-axis.`
`}`
	`for (auto&& p : points) {`
	`  inspect (p) {`
	`  [0, y]: // ...`
	`  }`
	`  // falls through if no match`
	`}`

10.3 Note on Ranges

The benefit of pattern matching for ranges is unclear. While it's possible to come up with a ranges pattern, e.g., `{x, y, z}` to match against a fixed-size range, it's not clear whether there is a worthwhile benefit.

The typical pattern found in functional languages of matching a range on head and tail doesn't seem to be all that common or useful in C++ since ranges are generally handled via loops rather than recursion.

Ranges likely will be best served by the range adaptors / algorithms, but further investigation is needed.

11 Acknowledgements

Thanks to all of the following:

- Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup for their prior work on N3449 [8], Open Pattern Matching for C++ [7], and the Mach7 [6] library.
- Pattern matching presentation by Bjarne Stroustrup at Urbana-Champaign 2015.
- David Sankel for his work on P0095 [5].
- Jeffrey Yasskin for his work on TODO.
- (In alphabetical order by last name) John Bandela, Agustín Bergé, Ori Bernstein, Matt Calabrese, Alexander Chow, Louis Dionne, Michał Dominiak, Eric Fiselier, Zach Laine, Jason Lucas, David Sankel, Bjarne Stroustrup, Tony Van Eerd, and everyone else who contributed to the discussions, and encouraged me to write this paper.

// TODO: Add Jeffrey's paper

References

[1] John Bandela. Simple, Extensible C++ Pattern Matching Library. Retrieved from https://github.com/jbandela/simple_match

[2] Luc Maranget. Warnings for pattern matching. Retrieved from http://moscova.inria.fr/~maranget/papers/warn/index.html

[3] Michael Park. Pattern Matching in C++. *MPark.Patterns*. Retrieved from https://github.com/mpark/patterns

[4] David Sankel. 2015. Pattern Matching and Language Variants. *P0095R0*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0095r0.html

[5] David Sankel. 2016. Pattern Matching and Language Variants. *P0095R1*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0095r1.html

[6] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Mach7: Pattern Matching for C++. *Mach7*. Retrieved from https://github.com/solodon4/Mach7

[7] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open Pattern Matching for C++. Retrieved from http://www.stroustrup.com/OpenPatternMatching.pdf

[8] Bjarne Stroustrup. 2012. Open and Efficient Type Switch for C++. *N3449*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3449.pdf

[9] Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis. 2016. Structured bindings. *P0144*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0144r2.pdf