Pattern Matching

Document #:	D1260R0
Date:	2018-05-22
Project:	Programming Language C++ Evolution
Reply-to:	Michael Park <mcypark@gmail.com>

Contents

1 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 [1] — which proposed pattern matching and language-level variant simulteneously — 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.

2 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 [2] introduced the ability to concisely bind names to components of a `tuple`-like value. 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.

3 Before/After Comparisons

3.1 Matching Integrals

Before	After

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

3.2 Matching Strings

Before	After

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

3.3 Matching Tuples

Before	After

`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;`
`}`
	`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;`
	`}`

3.4 Matching Variants

Before	After

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

3.5 Evaluating Expressions

Given the following definition:

```
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; };
```

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);
  }
}
```

4 Design Overview

4.1 Basic Syntax

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

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

4.2 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 evalutes 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.

4.3 Types of Patterns

4.3.1 Primary Patterns

4.3.1.1 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.]

4.3.1.2 Constant Pattern

The constant pattern has the form:

constant-expression

and matches value `v` if `std::strong_equal(c, v) == std::strong_equality::equal` is `true` where `c` is the constant expression.

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

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

[*Note:* The *id-expression* is overriden by the identifier pattern. `+id` or `(id)` is needed for disambiguation.]

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

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

4.3.2 Compound Patterns

4.3.2.1 Structured Binding Pattern

The structured binding pattern has `N` *pattern* instances with the form:

`[` $pattern_0$ `,` $pattern_1$ `, ... ,` $pattern_{N\text{-}1}$ `]`

and matches value `v` if each $pattern_i$ matches the i^{th} component of `v`. The components of `v` are determined by the structured binding declaration: `auto&& [__e`$_0$`, __e`$_1$`, ..., __e`$_{N\text{-}1}$`] = 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
}
```

4.3.2.2 Alternative Pattern

The alternative pattern has the form:

`<Alt>` *pattern*

Let `v` be the value being matched and `V` be its type. There are two cases we consider:

1. **Variant-Like**

   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 current alternative of `v` behaves as a reference of type `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:

   a. If `Alt` is **`auto`**
   b. If `Alt` is a **concept** and `Alt<std::variant_alternative_t<I, V>>()`
   c. If `Alt` is a **type** and `std::is_same_v<Alt, std::variant_alternative_t<I, V>>`
   d. If `Alt` is a **value** and is the same value as `I`.

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

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

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

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

2. **Polymorphic**

// TODO

5 Impact on the Standard

This is a language extension to introduce a new selection statement: `inspect`.

6 Proposed Wording

6.1 Syntax

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 (` *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:
  constant-pattern
  identifier
  wildcard-pattern
  structured-binding-pattern
  alternative-pattern

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

7 Design Decisions

7.1 Conceptual Model: Extending Structured Bindings

The design intends to be consistent and 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` vs `switch`

This proposal introduces a new `inspect` statement rather than trying to extend the `switch` statement for the following reasons:

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

7.3 First Match vs Best Match

// TODO

7.4 Statement vs Expression

This paper diverges from P0095 [1] 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. 1. In the case 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. 2. 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 syntax such as `x inspect { /* ... */ }`. The proposed syntax in this paper is consistent with every other statement in C++ today.

7.5 Language vs Library

There have been three popular pattern matching libraries in existence today.

- Mach7
- Simple Match by jbandela
- MPark.Patterns

The issue of introducing identifiers is burdensome enough that I believe it justifies a language feature.

7.6 Optimizations

Comparison elision?

...

7.7 Ranges

...

7.8 User-defined Patterns

...

8 Examples

...

9 Other Languages and Libraries

9.1 C#

9.2 Rust

Constants: https://github.com/rust-lang/rfcs/blob/master/text/1445-restrict-constants-in-patterns.md

9.2.1 Intersection of semantic / structural equality

9.3 Scala

Scala Tutorial - Pattern Matching: https://www.youtube.com/watch?v=ULcpWn23waw Matching Objects with Patterns: https://infoscience.epfl.ch/record/98468/files/MatchingObjectsWithPatterns-TR.pdf

9.3.1 Extractors

9.4 F#

9.4.1 Active Patterns

10 Future Work

11 Acknowledgements

Thank you to Agustín Bergé, Ori Bernstein, Alexander Chow, Louis Dionne, Michał Dominiak, Eric Fiselier, Zach Laine, Jason Lucas, David Sankel, Tony Van Eerd, and everyone else who contributed to the discussions, and encouraged me to write this paper.

References

[1] David Sankel. 2016. Pattern Matching and Language Variants. *P0095*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0095r1.html

[2] 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