Pattern Matching

Document #:	DxxxxR0
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 `std::tuple` and `std::variant`, the importance of mechanisms to interact with them have increased. While mechanisms such as `std::apply` and `std::visit` have been added, they leave much to be desired. Pattern matching is a mechanism that has been widely adopted across many programming languages. These 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 full solution 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 its components for use in subsequent logic. Today, C++ provides two types of selection statements which choose between one of several flows of control: the `switch` statement and the `if` 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 for inspection of the "vocabulary types" provided by the standard library such as `tuple`, `variant`, `string`, and `vector`.

Consider a variable `p` of type `Point` and a function `position` which prints whether `p` is positioned at the origin, on the x-axis or y-axis, or not on any axes.

Before

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

```
void position(const Point& p) {
  if (p.x == 0 && p.y == 0) {
    cout << "at the origin";
  } else if (p.x == 0) {
    cout << "on the x-axis";
  } else if (p.y == 0) {
    cout << "on the y-axis";
  } else {
    cout << "not on any axes";
  }
}
```

After

```
void position(const Point& p) {
  inspect (p) {
    [0, 0]: cout << "at the origin";
    [0, y]: cout << "on the x-axis";
    [x, 0]: cout << "on the y-axis";
    [x, y]: cout << "not on any axes";
  }
}
```

Structured binding declarations [2] in C++17 introduced the ability to concisely bind names to components of a value. Pattern matching aims to naturally extend this notion by performing **structured inspection** prior to forming the **structured bindings**. The proposed direction of this paper is to introduce an `inspect` statement as the third selection statement to fill the gap between the `switch` statement and the `if` statement.

3 Design Overview

3.1 Basic Syntax

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

3.2 Basic Model

Within the parenthesis, the `inspect` statement is equivalent to `if` and `switch` 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 against each pattern in order (first match). If a pattern is successfully matched with the value of the condition, control is passed to the statement following the matched pattern label. If there is a guard present, the boolean expression must evaluate to true in order for control to be passed to the statement following the label. If no pattern matches, then none of the statements are executed.

A name introduced by a pattern is in scope from its point of declaration until the end of the statement following the pattern label.

3.3 Requirements

Each pattern enforces a set of compile-time requirements that, if violated, results in the program being ill-formed.

3.4 Primitive Patterns

3.4.1 Constant Pattern

The constant pattern has the form:

constant-expression

Let `c` be the constant pattern and `v` the value being matched.

Requires: The expression `compare_3way(c, v)` must return `std::strong_equality`.

Matches: If `compare_3way(c, v) == 0` is `true`.

```cpp
int factorial(int n) {
  inspect (n) {
    0: return 1;
//  ^ constant pattern
    _: return n * factorial(n - 1);
  }
}
```

3.4.2 Identifier Pattern

The identifier pattern has the form:

unparenthesized *identifier*

Let `id` be the identifier pattern and `v` the value being matched.

Requires: None.

Matches: Any value `v`. `id` is an lvalue referring to `v`, and is in scope from its point of declaration until the end of the statement following the pattern label. This implies that identifiers cannot be repeated within he same pattern but can reused in the subsequent pattern.

```cpp
int n = 101;
inspect (n) {
  x: cout << x; // prints 101
}
```

3.5 Compound Patterns

3.5.1 Structured Binding Pattern

The structured binding pattern has the form:

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

Let `v` the value being matched.

Requires: `std::tuple_size_v<std::remove_cv_t<decltype(v)>> == N`

Matches: If $pattern_i$ matches `GET<i>(v)` for all $0 \leq i < N$.

3.5.2 Alternative Pattern

The alternative pattern has the form:

`<Alternative>` *pattern*

Let `v` the value being matched and `V` be `std::remove_cv_t<decltype(v)>`.

Requires: - `std::variant_size_v<V>` is defined. - `discriminator(v)` is a valid expression returning an integral, enumeration, or a class type contextually convertible to an integral type. - `std::variant_discriminator_v<Alternative, V>` is defined and is an integral, enumaration, or a class type contextually convertible to an integral type. - `get<std::variant_discriminator_v<Alternative, V>>(v)` is defined.

Matches: If `discriminator(v)` has the same value as `std::variant_discriminator_v<Alternative, V>`, and *pattern* matches `get<std::variant_discriminator_v<Alternative, V>>(v)`.

```
std::variant<T, U> v;
inspect (v) {
  <T> t: /* ... */
  <U> u: /* ... */
}
```

```
const Base& b = /* ... */;
inspect (v) {
  <Derived1> d1: /* ... */
  <Derived2> d2: /* ... */
}
```

4 Impact on the Standard

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

5 Proposed Wording

5.1 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:
identifier
constant-expression
wildcard-pattern
structured-binding-pattern
alternative-pattern

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

6 Design Decisions

6.1 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.

6.2 `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` requires `break`
- `switch` is purposely restricted to integrals for **guaranteed** efficiency. The primary goal of pattern matching is expressivity, while being as efficient as hand-written code.

6.3 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.

6.4 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 Examples

7.1 Matching strings

```
std::string s = "hello";
inspect (s) {
  "hello": std::cout << "hello";
  "world": std::cout << "world";
}
```

8 Other Languages and Libraries

8.1 C#

8.2 Rust

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

8.2.1 Intersection of semantic / structural equality

8.3 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

8.3.1 Extractors

8.4 F#

8.4.1 Active Patterns

9 Future Work

10 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