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 `tuple` and `variant`, corresponding mechanisms for interacting with them have become increasingly more important. Pattern matching is one such mechanism that has been widely adopted by many programming languages. These include text-based languages such as SNOBOL back in the 1960s, functional languages such as Haskell and OCaml, as well as "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 bindings [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 Structure

```
inspect (init-statement_opt expression) {
  pattern_0 guard_0: statement_0
  pattern_1 guard_1: statement_1
  /* ... */
}
```

3.2 Types of Patterns

3.2.1 Constant pattern

Constant patterns have the form:

constant-expression

A constant pattern with *constant-expression* `c` matches a value `v` if `v == c` is `true`.

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

3.2.2 Identifier Pattern

An identifier pattern is denoted by any valid identifier. It matches any value and binds it to the provided identifier.

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

3.2.3 Tuple Pattern

Syntax: [$pattern_0$, $pattern_1$, $\ldots$, $pattern_N$]

Matches values that fulfill the structured bindings protocol.

$pattern_i$ matches

3.2.4 Variant Pattern

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

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

Match the following values: - Scalar - Product Type (i.e., Structured bindable, (e.g., `std::tuple`) - Closed Polymorphism (e.g., `variant`) - Range (e.g., `string`) - Open Polymorphism (e.g., `std::any`, abstract base class)

3.3 Matching strings

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

3.4 Usability of `std::variant`

`variant` is hard to use. // ...

4 Impact on the Standard

This is a language extension to introduce an `inspect` statement.

5 Proposed Wording

```
inspect (int i = 42) {
  0: std::cout << "foo";
  1: std::cout << "bar";
}
```

https://wandbox.org/permlink/okgMcTpzXqcvN700

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, we attempt to **refer** to subobjects rather than introducing new variables to extract subobjects to.

6.2 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 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 expression since it is required to yield a value. 2. The resulting type of a 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 à la the ternary operator.

While an expression-form would be useful, the author believes that it can/should be introduced later, with different syntax such as `x inspect { /* ... */ }`. The proposed syntax in this paper is consistent with other statements today.

6.3 Language vs Library

7 Implementation Experience

8 Other Languages

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