Pattern Matching Discussion for Kona 2022

Document #:	P2688R0
Date:	2022-10-14
Project:	Programming Language C++
Audience:	Evolution

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

Contents

1 Introduction

This paper presents an overview of structural changes that are intended to be the next revision of [P1371R3]. It is intended to drive a focused discussion around the fundamental structural differences of patterns proposed in P1371 and P2392.

This paper presents an approach for pattern matching using `match` and `let`. The approach attempts to structurally unify the `inspect` and `is` constructs from [P2392R1].

While the approach described in this paper does not adopt `is`/`as` directly, I want to be clear that I'm not exactly against such facilities. The main goal of this paper is to present Lack of Pattern Composition and Sprawling and Repeating Structures as well as an alternative approach of P1371 that addresses previous feedback given to [P1371R3].

2 Motivation and Scope

The goal and motivation of this paper is to make progress on pattern matching for C++ by focusing the discussion to some of the core structural differences in the current pattern matching proposals in flight.

3 History

[P1371R0] proposed `id` to introduce a new name, and for `^id` to refer to an existing name. [P1371R1] removed the `^` due to feedback about it being a Microsoft C++ extension for smart pointers, as well as its hindrance on code performing simple enumeration matching.

```
enum Color { Red, Green, Blue };

inspect (e) {
  ^Red   => // I'm just trying to
  ^Green => // match simple enums...
  ^Blue  => // What's with these carets?
}
```

[P1371R1] and [P1371R2] kept `id` introducing a new name, but replaced `^id` with `case id` to refer to an existing name and `let id` to explicitly introduce a new name.

One of the motivations for this was to make simple code such as enumeration matching familiar again:

```
enum Color { Red, Green, Blue };

inspect (e) {
  case Red   => // Huh, just looks like `switch` now.
  case Green => // ...
  case Blue  => // ...
}
```

The `case` and `let` applied recursively to identifiers to allow patterns such as `case [a, b]` to match against existing `a` and `b`, and `let [x, y]` to introduce new names `x` and `y`. This recursive behavior received positive feedback from EWG in Cologne 2019:

> "We support the authors' direction of `let` and `case` modes applying to subpatterns.":
>
> SF: 7, F: 7, N: 5, A: 4, SA: 0

The problem with this approach appeared to be due to the overriding behavior of the `case` and `let`. For example, constructs such as `case [a, b, let x]` were allowed to match against existing `a` and `b` for the first two elements and bind `x` to the third.

In more complex examples however, it becomes more difficult to parse through which names are new and which refer to existing.

For example in `let [a, case [let b, c, d], [e]]`, `a`, `b`, `e` are new names and `c`, `d` refer to existing names.

In response, [P1371R3] chose to keep `case` and drop `let`. Because the top-level already had `let` behavior in that it introduced new names, we already effectively had "recursive `let`". We really only needed to provide a way to specify an existing name. This led to `case` becoming non-recursive.

At this point, there were several push backs:

— "We shouldn't to bifurcate expressions like this."

  Some expressions didn't need `case` (e.g., `0`), but some did (e.g., `id`). If `case expr` is pattern matching syntax for expressions, that would at least be consistent.

  This paper aims to address this concern by not requiring `case` for expressions at all.

— "Declaration of new names should have an introducer like most other places in the language."

  The comment is in regard to code like this:

```
inspect (e) {
  x => ...
}
```

  Some people found it surprising that that would introduce a new name, as most identifiers are introduced with an introducer. Notable exceptions being lambda captures and structured bindings. Even then, `[x, y]() {}` doesn't introduce `x` and `y` out of thin-air, it captures existing `x` and `y` at the same time. Though `[x=0, y=1]() {}` actually do. Structured bindings at least has the `auto` in front, like: `auto [x, y]` which doesn't apply to the `x` and `y` at all, but does hint that we're in a declaration context.

  This paper aims to address this concern by reintroducing `let` for new names.

4 High-Level Comparison Tables

4.1 Matching Integrals

This Paper	P2392

`x match {`
`  0 => { cout << "got zero"; }`
`  1 => { cout << "got one"; }`
`  _ => { cout << "don't care"; }`
`};`
	`inspect (x) {`
	`  is 0 => { cout << "got zero"; }`
	`  is 1 => { cout << "got one"; }`
	`  is _ => { cout << "don't care"; }`
	`}`

4.2 Matching Strings

This Paper	P2392

`s match {`
`  "foo" => { cout << "got foo"; }`
`  "bar" => { cout << "got bar"; }`
`  _ => { cout << "don't care"; }`
`};`
	`inspect (s) {`
	`  is "foo" => { cout << "got foo"; }`
	`  is "bar" => { cout << "got bar"; }`
	`  is _ => { cout << "don't care"; }`
	`}`

4.3 Matching Tuples

This Paper	P2392

```
p match {
  [0, 0] => {
    cout << "on origin";
  }
  [0, let y] => {
    cout << "on y-axis at " << y;
  }
  [let x, 0] => {
    cout << "on x-axis at " << x;
  }
  let [x, y] => {
    cout << x << ',' << y;
  }
};
```

```
inspect (p) {
  is [0, 0] => {
    cout << "on origin";
  }
  [_, y] is [0, _] => {
    cout << "on y-axis at " << y;
  }
  [x, _] is [_, 0] => {
    cout << "on x-axis at " << x;
  }
  [x, y] is _ => {
    cout << x << ',' << y;
  }
}
```

4.4 Matching Variants

This Paper	P2392

```
v match {
  <int32_t> let i32 => {
    cout << "got int32: " << i32;
  }
  <int64_t> let i64 => {
    cout << "got int64: " << i64;
  }
  <float> let f => {
    cout << "got float: " << f;
  }
  <double> let d => {
    cout << "got double: " << d;
  }
};
```

```
inspect (v) {
  i32 as int32_t => {
    cout << "got int32: " << i32;
  }
  i64 as int64_t => {
    cout << "got int32: " << i64;
  }
  f as float => {
    cout << "got float: " << f;
  }
  d as double => {
    cout << "got double: " << d;
  }
}
```

```
v match {
  <std::integral> let i => {
    cout << "got integral: " << i;
  }
  <std::floating_point> let f => {
    cout << "got floating point: " << f;
  }
};
```

```
// Unsupported.
```

This Paper	P2392

`v match {`
`  <int32_t> let i32 => {`
`    cout << "got i32: " << i32;`
`  }`
`  <auto> let x => {`
`    cout << "got something else: " << x;`
`  }`
`};`
	`// Unsupported.`

4.5 Matching Polymorphic Types

This Paper	P2392

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

4.6 Matching Optionals

This Paper	P2392

`void f(const optional<int>& opt) {`
`  opt match {`
`    let ?x => {`
`      cout << "optional is storing: " << x;`
`    }`
`    _ => {`
`      cout << "optional is empty";`
`    }`
`  };`
`}`
	`void f(const optional<int>& opt) {`
	`  inspect (opt) {`
	`    *x is _ => {`
	`      cout << "optional is storing: " << x;`
	`    }`
	`    _ => {`
	`      cout << "optional is empty";`
	`    }`
	`  }`
	`}`

`void f(const std::optional<int>& opt) {`
`  if (opt match let ?x) {`
`    cout << "optional is storing: " << x;`
`  } else {`
`    cout << "optional is empty";`
`  }`
`}`
	`void f(const std::optional<int>& opt) {`
	`  if (auto *x is _ = opt) {`
	`    cout << "optional is storing: " << x;`
	`  } else {`
	`    cout << "optional is empty";`
	`  }`
	`}`

4.7 Matching Nested Structs and Variants

```
struct Rgb { int r, g, b; };
struct Hsv { int h, s, v; };

using Color = variant<Rgb, Hsv>;

struct Quit {};
struct Move { int x, y; };
struct Write { string s; };
struct ChangeColor { Color c; };

using Command = variant<Quit, Move, Write, ChangeColor>;

Command cmd = ChangeColor { Hsv { 0, 160, 255 } };
```

This Paper	P2392

`cmd match {`
`  <Quit> => ...`
`  <Move> let [x, y] => ...`
`  <Write> let [text] => ...`
`  <ChangeColor> [<Rgb> let [r, g, b]] => ...`
`  <ChangeColor> [<Hsv> let [h, s, v]] => ...`
`  _ => ...`
`};`
	`inspect (cmd) {`
	`  is Quit => ...`
	`  [x, y] as Move => ...`
	`  [text] as Write => ...`
	`  [[r, g, b]] as ChangeColor as [Rgb] => ...`
	`  [[h, s, v]] as ChangeColor as [Hsv] => ...`
	`  _ => ...`
	`}`

5 Design Overview

The overall idea is to introduce a single `match` construct, along with a context-sensitive keyword `let`. Together they can be used to select a branch, test whether a value matches a single pattern, and in `if`/`while` statements.

```
expr match {
  pattern1 => statement1;
  pattern2 => statement2;
  // ...
}
```

`let` denotes that an identifier is a new name rather than an existing name.

```
int x = 42;

expr match {
  x => ... // match against existing `x`
  let x => ... // introduce new x.
}
```

The following is used to match a value against a single pattern.

```
expr match pattern
```

The following is the match expression being used within an `if` statement.

```
if (expr match [0, let foo]) {
  // `foo` is available
}
```

A optional guard can be added for a single pattern match as well:

```
pair<int, int> fetch(int id);

bool is_acceptable(int id, int abs_limit) {
  return fetch(id) match let [min, max] if min >= -abs_limit && max <= abs_limit;
}
```

The scope of the bindings introduced by `let` are as follows:

- — If the pattern is left of `=>`, the scope of the binding is the corresponding statement.
- — If the pattern is in a *expr* `match` *pattern* *guard*$_{opt}$ expression, the scope of the binding is the expression unless:
- — If the construct immediately enclosing the expression is an `if` or `while` statement, the scope of the binding is the `if` or `while` statement.

5.1 Syntax Overview

expression `match` *pattern* *guard*$_{opt}$

expression `match` *trailing-return-type*$_{opt}$ `{`
  pattern *guard*$_{opt}$ `=>` *statement*
  …
`}`

guard:
  `if` *expression*

pattern:
  `_`
  expression
  type-id
  concept
  `(` *pattern* `)`
  `<` *discriminator* `>` *pattern*$_{opt}$
  `[` *pattern*$_{0}$ `,` *pattern*$_{1}$ `,` … `,` *pattern*$_{N}$ `]`
  `[` *designator*$_{0}$ `:` *pattern*$_{0}$ `,` *designator*$_{1}$ `:` *pattern*$_{1}$ `,` … `,` *designator*$_{N}$ `:` *pattern*$_{N}$ `]`
  `?` *pattern*
  expression `:` *pattern*
  pattern `&&` *pattern*
  pattern `||` *pattern*
  `let` *let-pattern*

let-pattern:
  `_`
  identifier
  `(` *let-pattern* `)`
  `<` *discriminator* `>` *let-pattern*$_{opt}$
  `[` *let-pattern*$_{0}$ `,` *let-pattern*$_{1}$ `,` … `,` *let-pattern*$_{N}$ `]`
  `[` *designator*$_{0}$ `:` *let-pattern*$_{0}$ `,` *designator*$_{1}$ `:` *let-pattern*$_{1}$ `,` … `,` *designator*$_{N}$ `:` *let-pattern*$_{N}$ `]`
  `?` *let-pattern*
  expression `:` *let-pattern*

discriminator: one of
  `auto`, *concept*, *type-id*, *constant-expression*

6 Addressing Feedback

6.1 Pattern Matching Outside of `inspect`

This paper proposes a `match` construct which can be used as a selection mechanism, `expr match { p1 => s1; ... }` as well as `expr match pattern` expression. These correspond to `inspect` and `is` of P2392 respectively.

This was a piece of feedback from P2392 which is to allow pattern matching outside of `inspect` which only allowed to select a branch.

6.2 Expressions vs Bindings

The History section covered the previous attempts around `case` and `let`, during which the following feedback were given:

> "We shouldn't to bifurcate expressions like this."

That is, expressions are just expressions without needing anything everywhere else in the language. This is the case in this paper. That is, `x` is an expression referring to an existing variable like it does everywhere else in the language.

> "Declarations of new names should have an introducer like most other places."

New names need the `let` introducer to introduce bindings, just like other new names in most other places in the language.

> "I don't want the documentation of pattern matching to have to mention a caveat that `x` is a new name and therefore shadows an existing variable."

As mentioned above, `x` is an expression that refers to an existing variable.

7 Observations of P2392

The following are a collection of observations P2392 as I best as I understand. Apologies for any inaccuracies.

7.1 Lack of Pattern Composition

Consider the example: Matching Nested Structs and Variants:

```
struct Rgb { int r, g, b; };
struct Hsv { int h, s, v; };

using Color = variant<Rgb, Hsv>;

struct Quit {};
struct Move { int x, y; };
struct Write { string s; };
struct ChangeColor { Color c; };

using Command = variant<Quit, Move, Write, ChangeColor>;

Color c = Rgb { 0, 160, 255 };
Command cmd = ChangeColor { c };
```

The following is what it would look like to match a `Color`:

This Paper	P2392
`c match { <Rgb> let [r, g, b] => ... };`	`inspect (c) { [r, g, b] as Rgb => ... }`

```
c match {
  <Rgb> let [r, g, b] => ...
};
```

```
inspect (c) {
  [r, g, b] as Rgb => ...
}
```

Now consider the code that matches a `ChangeColor` which contains `Color`:

This Paper	P2392

```
c match {
  <ChangeColor> [<Rgb> let [r, g, b]] => ...
};
```

```
inspect (c) {
  [[r, g, b]] as ChangeColor as [Rgb] => ...
}
```

The pattern `<Rgb> let [r, g, b]` which matches a `Color` is composed verbatim when a `Color` is composed within `ChangeColor`. The pattern `[r, g, b] as Rgb` in the P2392 example do not compose this way.

This means that the patterns in P2392 don't compose in the same way that the values they describe. The mechanism used in P2392 seem more like chaining of operations rather than composition of patterns. I believe that this is a big loss in usability and consider it to be the biggest fundamental difference.

7.2 Sprawling and Repeating Structures

Consider matching a pair of `int`s and we want to test first element for `0` and bind the second.

This Paper	P2392

```
c match {
  [0, let y] => ...
};
```

```
inspect (c) {
  [_, y] is [0, _] => ...
}
```

We see that the structure of pair, `[_, _]`, is repeated in P2392.

The repeated structure starts to spread with `as` conversions in play. Consider the example: Matching Nested Structs and Variants:

```
struct Rgb { int r, g, b; };
struct Hsv { int h, s, v; };

using Color = variant<Rgb, Hsv>;

struct Quit {};
struct Move { int x, y; };
struct Write { string s; };
struct ChangeColor { Color c; };

using Command = variant<Quit, Move, Write, ChangeColor>;

Command cmd = ChangeColor { Rgb { 0, 160, 255 } };
```

Suppose we want to test for specific values of `r` and `g`:

This Paper	P2392
`c match { <ChangeColor> [<Rgb> [0, 160, let b]] => { // use `b` here } };`	`inspect (c) { [[_, _, b]] as ChangeColor as [Rgb] is [[0, 160, _]] => { // use `b` here } }`

Now the structure of `[[_, _, _]]` need to be repeated, and they're even further away. In my opinion, this is even more difficult to parse through and understand the structure of the value.

7.3 Dereference Syntax Seems Problematic

The use of `*` *pattern* syntax seems problematic for multiple reasons.

7.3.1 Ambiguity with pointer declaration.

```
inspect (v) {
  x is _ => // ...
  *y is _ => // binds `y` to `*v`
}

auto x = v;
auto *y = v; // pointer declaration
auto *y is _ = v; // the `is` makes this not be a pointer declaration?
auto &y is _ = v; // the `is` doesn't change that this is a reference?
```

7.3.2 Inconsistency and Evolution

```
inspect (v) {
  is &a => // matches if `v == &a`
  is *a => // matches if `v && *v == a`
}
```

This is because `*` is a pattern but `&` is not. Aside from being a bit odd, I view this as a problem for the evolution of pattern matching since this means that no other unary expressions can later become a pattern.

a) Are we sure that folks won't be confused by the inconsistency?
b) Are we confident that `*` is the only unary expression we'll ever want to make into a pattern?

7.3.3 Previous EWG Feedback

[P1371R0] had proposed `* pattern` syntax and presented in EWG Kona 2019, the overwhelming feedback was to not use that syntax since it is too confusing with expressions.

I agree with this sentiment, considering a simple example from Inconsistency and Evolution.

7.4 Non-Type Variant Discriminators

Consider a variant with short-string optimization using a predicate as a discriminator rather than an explicitly stored value. This example is adapted from Bjarne Stroustrup's pattern matching presentation at Urbana-Champaign 2014 [PatMatPres].

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

  StorageKind index() const;
  char *data();

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

The discriminator is `StorageKind`, retrieved via an `index()` function as per current variant-like protocol:

```
StorageKind String::index() const {
  return size > sizeof(local) ? Remote : Local;
}
```

Ultimately, after opting into the rest of variant-like protocol the use looks like this:

```
char* String::data() {
  return inspect (*this) {
    <Local> let local => local;
    <Remote> let remote => remote.ptr;
  };
}
```

where `Local` and `Remote` are not types, but rather `enum` values.

In section 3.5.9 of [P2392R1], the following example appears:

```
// short string optimization
char* String::data() {
  inspect (*this) {
    [i] is Local => return i;
    [r] is Remote => return r.ptr;
  }
}
```

But this example doesn't really seem to work. As far as I can understand, the types corresponding to the discriminators have to be used in order to trigger the `operator is`/`operator as` mechanism. For example, the `Local` example needs to be something like `is char[32]`. The `Remote` example seems to not really be spellable at all since the type is anonymous.

7.5 `let` vs Trailing `is _`

As far as I understand, the following is a consistency that P2392 tries to make.

```
auto <names> is <constraint> = v;
//   ^^^^^^^^^^^^^^^^^^^^^^^

auto <names> as <target> = v;
//   ^^^^^^^^^^^^^^^^^^^

inspect (v) {
    <names> is <constraint> => ...
//  ^^^^^^^^^^^^^^^^^^^^^^^

    <names> as <target> => ...
//  ^^^^^^^^^^^^^^^^^^^
}
```

When neither constraint nor target exists, it can be omitted in familiar fashion in the declaration form.

```
auto <names> = v;  // e.g., auto [x, y] = v;
```

This doesn't work in `inspect` and a trailing `is _` is required.

```
inspect (v) {
    <names> is _ => // e.g., x is _ => ...
                    //       *x is _ => ...
                    //       [x, y] is _ => ...
}
```

The use of `let` seem to be a slightly better spelling than a trailing `is _`.

```
inspect (v) {
    let <names> => // e.g., let x => ...
                   //       let ?x => ...
                   //       let [x, y] => ...
}
```

The trailing `is _` appears in other contexts as well such as:

```
if (auto *x is _ = opt) { ... }
```

which is expected to be a common use case.

7.6 `is` is an `&&` Combinator in Disguise

A P2392 pattern like `[a, b, c] is [1, 0, 1]` is really two patterns combined with `&&`. Consider that the following has equivalent meaning: `[a, b, c] is _ && is [1, 0, 1]`. In the approach presented in this paper, this would be `let [a, b, c] && [1, 0, 1]`.

8 References

[P1371R0] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2019-01-21. Pattern Matching.
https://wg21.link/p1371r0

[P1371R1] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2019-06-17. Pattern Matching.
https://wg21.link/p1371r1

[P1371R2] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2020-01-13. Pattern Matching.
https://wg21.link/p1371r2

[P1371R3] Michael Park, Bruno Cardoso Lopes, Sergei Murzin, David Sankel, Dan Sarginson, Bjarne Stroustrup. 2020-09-15. Pattern Matching.
https://wg21.link/p1371r3

[P2392R1] Herb Sutter. 2021-07-19. Pattern matching using "is" and "as"
https://wg21.link/p2392r1

[PatMatPres] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. “Pattern Matching for C++” presentation at Urbana-Champaign 2014.