Doc. No. D3332R0
Date: 2024-06-18
Audience: EWG
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

A simpler notation for PM

Bjarne Stroustrup

Abstract

Looking at P2688R1, I decided that I liked the general design but found the notation cumbersome and distracting. I would like to see PM is C++26 but I fear that the notation would be a long-term burden. We can do better.

My design is based on the idea that every pattern match can name the expression that it matched.

pattern name => action *// the action can refer to name*

This introduces the identifier **name** for the pattern matched, just as identifiers are introduced in structured binding. That is, **name** refers to whatever was matched and has the type of whatever was matched exactly as the **let** mechanism in P2688R1. Incidentally, structured binding was meant to be a first step in the direction of pattern matching, so this is a reversion to the original.

Naming is optional. Thus

pattern => action *// doesn't introduce a new name.*

Names not introduced in this way refer to names in the enclosing scope.

Like P2688R1, I use _ for "match everything", but now we can name what it matched. For example:

_ name => action *// the action can refer name*

None of what I say would change if we decided on __ rather than _.

With permission from Michael Park, with whom I have discussed my suggested notation, I will list the examples from P2688R1 with my suggested alternatives.

This note is not meant to take sides in the discussion between the two proposals for PM (the other being P2392R2), simply to show that a simplification of P2688R1 is possible.

Also, maybe this can help re-ignite the discussion about Pattern Matching which seems not to have progressed despite many expressions of support for the general idea and its (obvious?) benefits to expression of ideas in code and to type safety.

1. The **Let** Pattern

Form P2688R1

> A wildcard pattern always matches any **subject**.
>
> A let pattern always matches any **subject**. The **binding-pattern** is either an **identifier** or a structured bindings pattern.
>
> ```
> int v = 42;
> v match {
>   let x => std::print("ignored");
> //  ^^^^^  let pattern
> };
> ```
>
> `let` can be used to introduce new names individually, or all-in-one.
>
> ```
> let x            // x is new
> [a, let y]       // a is old, y is new
> [let x, b]       // x is new, b is old
> let [x, y]       // x and y are both new
> let [x, [y, z]]  // x, y, z are all new
> ```

In other words, a let-pattern is a wildcard-pattern that also introduces a name. Using _ rather than let, we get

```
v match {
        _ => std::print("ignored");
};

_ x              // x is new
[a, _ y]         // a is old, y is new
[_ x, b]         // x is new, b is old
_ [x, y]         // x and y are both new
_ [x, [y, z]]    // x, y, z are all new
```

That last _ **[x, [y, z]]** looks a bit magical to me. I think I'd prefer to require [**_x, _[y, z]]** rather than having a special rule for nesting.

My idea here is to allow every pattern to name its match, not just whatever matches everything.

2. Examples from P2688R1

The initial, simplest examples are identical.

In every case, the P2688R1 variant is the first followed by the "_ variant".

2.1. Integers

The _ without a name has the same meaning. Simple integer matches require no change:

```
x match {
        0 => std::print("got zero");
        1 => std::print("got one");
        _ => std::print("don't care");
};
```

2.2. Strings

Same for strings (and other values)

```
s match {
        "foo"    => std::print("got foo");
         "bar"   => std::print("got bar");
        _        => std::print("don't care");
};
```

2.3. Tuples

```
p match {
        [0, 0]          => std::print("on origin");
        [0, let y]      => std::print("on y-axis at {}", y);
        [let x, 0]      => std::print("on x-axis at {}", x);
        let [x, y]      => std::print("at {}, {}", x, y);
};
```

The [...] is the notation for looking into a nested object. In all cases [...] does a structured binding on the representation of the object.

```
p match {
        [0, 0]          => std::print("on origin");
        [0, _ y]        =>  std::print("on y-axis at {}", y);
        [_ x, 0]        => std::print("on x-axis at {}", x);
        _ [x, y]        => std::print("at {}, {}", x, y);
};
```

Here, I find the **let**s distracting, and an unnecessary added concept. With code coloring, those **let**s becomes far louder and distracting from the main logic of the code.

Note that the space between _ and **x** is necessary.

A pair is a kind of tuple, so we get

```
void f(pair<int,int> p)
{
        p match {
                [_ first, 42] => // first names  the first int if the second equals 42
        };
}
```

2.4. Template parameters

This should work, though I don't see it explicitly mentioned in P2688R1:

```
Template<class T>
void f(T x)
{
        X match {
                int i => // ...
                _ xx => // ...
        };
}
```

An argument has been made that given a construct like **int i** , people will expect a new variable to be introduced and type conversion rules to be applied. In that case, people's expectations would be wrong. The rules are the ones for structured binding: simpler and more efficient (no temporaries and no implicit conversions). My conjecture is that people would soon be used to this and that conversely, they would soon tire of frequently having to write **:let** and start complaining about verbosity.

2.5. Concepts

```
v match {
        std::integral: let i            =>  std::print("got integral: {}", i);
        std::floating_point: let f      =>  std::print("got float: {}", f);
};
```

Eliminating the **:let**, we get:

```
v match {
        std::integral i         => std::print("got integral: {}", i);
        std::floating_point f   => std::print("got float: {}", f);
};
```

This matches the P2688R1 design, but shouldn't match of a concept yield a type? Well, PM is an expression so a non-type value is the only choice and people can **decltype** on the selected value.

2.6. Nested Structures

An example from P2688R1:

```
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 } };

cmd match {
        Quit: _                          => // ...
        Move: let [x, y]                 => // ...
        Write: let [text]                => // ...
        ChangeColor: [Rgb: let [r, g, b]]  => // ...
        ChangeColor: [Hsv: let [h, s, v]]  => // ...
};
```

Instead we get

```
cmd match {
        Quit                          => // ...
        Move [x, y]                   => // ...
        Write [text]                  => // ...
        ChangeColor [Rgb [r, g, b]]   => // ...
        ChangeColor [Hsv [h, s, v]]   => // ...
};
```

Again, I don't use **:let** or _ because there already is a pattern that has been matched so we know that if there is a name, it is the name of the match.

2.7. Class hierarchies

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

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

With _ , we get:

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

I have a problem with this requiring the representation of the shapes to be exposed, but that's a separate issue.

We can also (alternatively) do the match and naming inside the [...]:

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

That possibility becomes important in other examples.

3. Special cases

The standard library has several vocabulary classes with semantics and use cases that are "special"; that is, have use cases that don't match simple types.

- Optional
- Varian
- Pointers
- Expected

Unfortunately, it seems that these types also require special treatment from PM.

3.1. Variant

Matching a variant implicitly goes to the active alternative:

```
std::variant<int, bool, std::string> parse(std::string_view);

parse(some_input) match {
        int: let i                  => // ...
        bool: let b                 => // ...
        std::string: let s          => // ...
};
```

This indirection (into the variant) seems necessary for convenient use. Using matches that name their values:

```
parse(some_input) match {
        int i               => // ...
        bool b              => // ...
        std::string s       => // ...
};
```

The semantics is still identical for the two versions.

So far, so good, but the P2688R1 gives this example

```
parse(some_input) match {
        int i      => // ...
        auto x   => // ...
};
```

The **auto x** binds to the whole value, but how did that" **i**" enter the picture without a **let**? I might have misunderstood something, but using _, we get

```
parse(some_input) match {
        int i      => // ... binds i to the int alternative
        _ x        => // ... binds x to the whole variant
};
```

Basically, _ meaning match everything does the job of **auto**.

3.2. Pointers

Safely looking at what a pointer points to requires a nullptr check. PPP2688R1 Has that test explicit

```
void f(int* p) {
        p match {
                ? let i    => // ...
                nullptr  => // ...
        };
}
```

Without the ? a match would look at the pointer itself.

Using _ we get

```
void f(int* p) {
        p match {
                ? _  i     => // ...
                nullptr  => // ...
        };
}
```

3.3. Optional

Std::optional (like pointers) requires a run-time check of validity.

```
void f(std::optional<int> o) {
        o match {
                ? let i              => // ...
                std::nullopt      => // ...
        };
}
```

With _ we get

```
void f(std::optional<int> o) {
        o match {
                ? _ i           => // ... check for validity and then match
                std::nullopt    => // ...
        };
}
```

3.4. Expected

P2688R1 seems a bit undecided about std::expected. See §5.5. It seems to me that it ought to be handled similarly to pointers and optional. In all three cases, there is a normal/expected alternative and a less desirable one that must be handled

```
e match {       // e is an std::expected
        ? _ x    => // ... the expected case with value x
        _ err    => // ... fall back on examining the error case
};
```

Or even

```
e match {       // e is an std::expected
        ? int I         => // special valid case
        ? _ x           => // ... the expected case with value x
        [_, _ err ]     => // ... fall back on examining the error case
};
```

Summary

Letting every match optionally name what it matches simplifies and generalizes the P2688R1 proposal.