Document #: P3249R0
Date: 2024-22-4
Project: Programming Language C++
Audience: SG21, EWG
Reply-to: Ran Regev <regev.ran@gmail.com>

A unified syntax for Pattern Matching and Contracts when introducing a new name

Table Of Contents

Table Of Contents
Abstract
Motivation
Current State
    Examples
Proposal
A Decision is Needed
Possible Extensions (not suggested in this paper)
Other Attempts to unify the syntaxes
A word on keystrokes
Wording
References

Abstract

This paper suggests a unified syntax when introducing a new name in Pattern Matching and Contracts.

Motivation

There are situations where Pattern Matching [P2688] needs to introduce a new name in the *pattern* part that is later referenced in the *expression-statement* part.
There is a situation where Contracts [P2900] needs to introduce a new name in postconditions as part of the post's predicate.

With the current state of the two features these two similar constructs have a different syntax.
Having two constructs that essentially do the same has drawbacks:

- It makes the language more complex.
- It is harder to learn

Uniting the syntax has benefits:

- Recurring syntax is easier to understand, simpler.
- Easier to learn.

Current State

Pattern matching introduces a new name in *pattern* with the contextual keyword *let, the new name* and a two symbols sign *=>*.
Contracts introduce a new name in a postcondition with *the new name* and a *colon*.

Examples

Pattern Matching:

```
int i = f();
i match {
  42 let val => print(val); // introduce new val (val==42)
  let x => print(x); // introduce new x.
}
```

x and val refer to the pattern being matched and may be used in the expression-statement that follows.

Contracts:

```
int foo()
    post(ret : ret > 0) // introduce new ret
```

ret is part of the postcondition predicate and refers to the returned value of foo. It may be used after the colon in the boolean expression.

Proposal

Use *let* and => to introduce a new name in Contracts. (see wording below)
For example:

```
int foo()
    post( let r => r > 0) // introduce new r
```

A Decision is Needed

Both Contracts and Pattern Matching are not standardized. It means that Pattern Matching may end up having a syntax that is not as in [P2688R1]. In the worst scenario we might end up with Contracts having the syntax suggested in P2688R1 and Pattern Matching ends up having a completely different syntax. This is of course not the intent and we do hope to end up with the same syntax.
However, waiting for pattern matching to materialize and only then cooperating with it is not possible - it might be materialized too late, after Contracts are standardized. Moreover, end users don't really care about internal WG21 processes that result in inconsistency - they want a coherent language.
In addition, not trying to merge the syntaxes will *surely* result in two different syntaxes for the same thing, while trying to merge the syntaxes only *may* result in two different syntaxes.
Assuming Contracts materialized before pattern-matching - the fact that we use the same syntax as its R1 version will only add an additional constraint on pattern matching considerations if someone will ask for a different syntax. In this situation it is on pattern-matching (and EWG) to decide which force is stronger - alignment with Contracts or whatever is the reason for deviating from R1 syntax. SG21, however, did its best to unify the syntax.

Possible Extensions (**not** suggested in this paper)

Once the new name is introduced it can be easily used in various expressions, like pattern matching and do expressions:

```
int foo()
    post( let ret =>
            ret match {
                  1 => true;
                  2 => true;
                  _ => false;
            }
    );
```

do expression [P2802]:

```
int foo()
    post( let ret =>
              do {
               if (ret * 10 < 145) {
                 do_return false;
               } else {
                  do_return true;
              }
        );
```

Other Attempts to unify the syntaxes

A.
[P3210] claims that “A Postcondition *is* a Pattern Match”.
This paper [D3249] claims that “A Postcondition *might be* a Pattern Match” but can also be other things and therefore P3210 falls short of suggesting only one of the possible extensions.

P3210 suggests more than one syntax. One syntax for “simple” cases and one for “complex” syntaxes. This complexity is redundant and can be avoided.

P3210 removes completely the need to introduce a new name or the keyword *match*, as it assumes that a postcondition is a pattern matching. This leads to a somewhat awkward syntax when a boolean expression is combined with the defaulted pattern matching.
For example, to a postcondition to be true under `(ret == 1 or ret == 2 or ret > 100)`:

```
int foo()
  post (
      1 => true;
```

```
      2 => true;
      let ret => ret > 100
  );
```

D3249 on the other hand suggests (future extension only) for the same condition to be written:

```
int foo()
  post ( let ret =>
      ret match {
        1 => true;
        2 => true;
      } or
      ret > 100
  );
```

P3210 also suggests the keyword *result* as the name of the returned value but this paper does not tackle this property of P3210.

B.
There was a suggestion for post-MVP in [P2961 Section 6.2] to enable *structured-binding* of the returned value in postconditions:

```
std::pair<int,int> f()
   post( [a,b] : a < b);
```

With this paper and pattern matching this can be done like any other pattern matching:

```
std::pair<int,int> f()
   post( let result => result match let [a,b] if a < b);
```

A word on keystrokes

Some people prefer to minimize keystrokes and completely remove the need in introducing the new name in places where it is “obvious” what is done and implicit understanding of the context is enough (e.g. P3210 where the need to state that we are pattern-matching the result is redundant all together).
This paper (D3249) does not eliminate this option - a *result-name-introducer* is still optional and may be omitted whenever it is redundant. If in the future we discover for example that postcondition is indeed by default a pattern-matching and introducing a name for the returned is redundant, this paper does not close the door to this option. There will be existing codebases that use *let* - that stays the same. A new code may omit the entire construction. On the other hand, if in the first place in some constructs we don’t explicitly state that we are referring to the

returned name, we'll not be able to change it later as codebases will not have the returned values name.

Wording

The proposed changes are relative to P2900R6

Modify [dcl.contract.func]

result-name-introducer :
    let attributed-identifier ~~:~~ =>

References

[P2688R1] Michael Park, Pattern Matching: match Expression
[P2900R6] Joshua Berne, Contracts for C++
[P2806R2] Bruno Cardoso Lopes, do expressions
[P3210R0] Andrew Tomazos, A Postcondition *is* a Pattern Match
[P2961R2] Timur Doumler, A natural syntax for Contracts