Doc. No. P3572R0
Date: 2025-01-12
Audience: EWG
Author: Bjarne Stroustrup
Reply to: bjarne@stroustrup.com

Pattern matching

Bjarne Stroustrup
Columbia University

C++ badly needs functional style pattern matching. It is what we lack to make C++ code safer, simpler to write, and for C++ to be seen (correctly) as modern. However, to serve the larger C++ community well it must be simple to use for simple tasks (as simple as or simpler than using union, optional, and variant), as efficient as the basic uses of those older ways of selecting while also supporting more advanced uses.

This is not a new idea. I presented some experiments and implementation ideas in a WG21 evening session back in 2014 Pattern Matching for C++ and restated the case for pattern matching in 2021: Thoughts on pattern matching P2411R0. Since then, two proposals have emerged

- [P1371] B. Cardoso Lopes, S. Murzin, M. Park, D. Sankel, D. Sarginson, B. Stroustrup: Pattern Matching (R3). 2020-09.
- [P2392] H. Sutter: Pattern matching using is and as. 2021-06-13.

Both proposals have evolved since then [PH2024; HS2024]. I have followed their development pretty closely, but I don't see a consensus for either. To avoid personalizing the discussion, I will refer to Michael Park's version the "**match** design" and Herb Sutter's version the "**is/as** design."

The general notion of pattern matching (PM) for C++ has wide support:

- The Direction Group recommends progress on PM: DIRECTION FOR ISO C++
- Ville Voutilainen in To boldly suggest an overall plan for C++26 recommends "making progress " on PM (§5.3).

In my opinion, both proposals need simplification, but the **is/as** design is by far the better design framework and also have more implementation experience. We should accept a slightly slimmed down version of that for C++26. This opinion differs from the Wroclaw EWG vote that preferred the **match** proposal ("Consensus, but sizable objection").

What makes the **is/as** design a good framework for pattern matching:

- It is a framework: adding new forms of selection can be done without adding new syntax
- It is simple to use simply
- It can be (has been) implemented efficiently
- It has a consistent (simple) general syntax

Selecting alternatives

Both proposals can match a variety of alternatives

- Values
- Types
- **concep**ts
- Classes in class hierarchies
- **optional**s
- **any**s
- **variant**s
- Pointers
- **tuple**s

However, there is a major difference in how they map an expression to the type of a pattern. The **match** design uses up-front syntax and the **is/as** proposal use operators for customization. I strongly prefer the second approach because it leaves complexity to implementation. If in the future we want to add a new alternative, in **is/as** we can do so without changing the notation used in application code. In particular, this will allow developers to add their own alternatives. This is very much in the tradition of C++.

Opinion

I would like to see pattern matching now but could wait for **is**/**as** as general constructs (e.g., in **if** statements). In particular, **as** outside pattern matching could become a complication given the timescale for C++26. I do think **as** is necessary for pattern matching in a class hierarchy.

The **is/as** examples are consistently as short or shorter than the **match** examples, and as logically simple or simpler than the **match** examples.

I would like to see “Tony tables” just for the **is/as** and **match** examples. I was occasionally confused by the variety of designs discussed and compared in the latest **is/as** paper.

The **is/as** design is consistent with structured binding (as was stated as an aim when structured binding was approved).

Objections

I can summarize the objections to the **is/as** proposal that I heard most in Wroclaw as "but I want to say exactly what I mean". That's a simplification, of course, but what I think it shows is the difference between focusing on specification and on implementation.

If someone wants to be precise and know the diverse notations for section (e.g., for optional, any, and variant) those existing constructs are still be available for use. I prefer to have the type system select the implementation.

"Other languages use the match style." Yes, but "other languages" aren't C++ and don't have the range of constructs that C++ does. For example, the match design suggests **case**, **let**, **?**, *****, **?***, and **^** to handle such variations, and **<…>** to distinguish types. This is a burden to programmers; at best "expert friendly." The **is/as** design has only ***** for dereferencing. Both use **[…]** for tuples just as structured bindings.

Notation

Comments on §4.3 of [HS2024]. I don't care much about the detailed syntax used (e.g., **match** or i**nspect**) but the notation should directly reflects the semantic while not requiring the programmer to know all the (often obscure) implementation details. Notation should not directly reflect implementation unless if absolutely necessary.

If I had to choose, I'd pick a single introducing keyword (**match** or **inspect**) but not **switch** because that's likely to confuse pattern matching with old style code (e.g., a **switch** statement that doesn't start with a **case** or cause people to mistakenly over-or-underuse **break** out of habit).

I'd also pick **=>** over alternatives because it's short and doesn't usually imply assignment (like **=** does).

I like that every alternative starts with a prefix keyword (**is** or **as**). That eases readability and layout.

References

[MP2024] Michael Park: Pattern Matching: match Expression. P2688R4. 2014-12-17.

[HS2024] Herb Sutter: Pattern matching using is and as. P2392R3 2014-10-15.