Put `std::monostate` in `<utility>`

- Document number: P0472R3
- Date: 2024-11-18
- Reply-to:
  David Sankel <camior@gmail.com>
  Andrei Zissu <andrziss@gmail.com>
- Audience: Library Evolution

Revision History

Revision 3

- Addressed wording feedback from LWG by reproducing monostate-related declarations in the utility header.

Revision 2

- Added *Alternatives* section, currently referring to `std::nullptr_t`.
- Proposed Wording - redone.

Revision 1

- Keep `std::monostate` in `<variant>` too, for backward compatibility.
- Added a use case (safe invocation utility) in the motivation section.
- Added backward compatibility section.
- Added proposed wording.
- Added co-author.

Revision 0 (2016)

- Original version.

Abstract

`std::monostate` is currently defined and available in the `<variant>` header, but its utility is not limited to `variant`s. We propose adding `std::monostate` availability to `<utility>` to reduce artificial coupling of `std::monostate` clients to the `<variant>` header. We are not proposing removing `std::monostate` from `<variant>`, for backward compatibility.

Motivation

`std::monostate` is a class that has exactly one value. It is default constructible, copyable and supports all the comparison operations, or in other words it is a regular type. `std::monostate` is about as simple of a type as one could concoct. These properties turn out to be useful for writing template code.

The first use case is in testing. Does your custom `vector` or `set` make any undesirable assumptions about the types they are instantiated with? If they work properly with `std::monostate`, then probably not. `std::monostate` can be used in this way as a means to write simple test drivers.

The second use case occurs in more sophisticated template metaprogramming. The well known `std::future` class makes use of a "special" template parameter `void` to indicate that it carries no information aside from when the future is fulfilled. Using the `void` keyword to represent this situation carries a serious implementation burden due to its many strange properties. While this burden may not be a problem for standard library implementers, it would be nice to have a simpler option for the more common developer.

Another similar use case involves writing a safe invocation utility which executes a provided std::invocable and swallows all exceptions (via catch(...)). Such a utility would provide the returned value wrapped in either a std::optional or a std::expected, which would be returned as disengaged/unexpected in case the outcome of the invocation was an exception. This utility would also need to support callables returning void, which for ease of writing generic code we would want to share the same std::optional/std::expected representation. For this end std::monostate would be an ideal choice for value/expected type, as it is a regular type while at the same time cannot be mistakenly assigned or implicitly converted to any other useful type.

`std::monostate` is a much more natural way to represent "no information" than `void` is. It has exactly one value and is a regular type instead of a keyword. Consider how simple the following code is:

```
template<typename ExtraInformation = std::monostate>
class Data
{
  //...
  ExtraInformation m_extraInformation;
};
```

Here we have a `Data` template which optionally carries extra information. The use of `std::monostate` in this example makes it simple in both specification and implementation to represent `Data` objects that carry no additional information.

Alternatives

As has been pointed out on the reflector, there already exists a similar type - `std::nullptr_t`. Why wouldn't we settle for it as a proper general use substitute?

The reason is that misuses of `std::monostate` are much less likely. `std::nullptr_t` can be copied to any pointer type, and can be outputted by both cstdio and iostream families of facilities. `std::monostate` is ill-formed in any assignment as well as in streamed output, and is only well-formed with cstdio functions (with likely QoL warnings). This makes it a better match for a type that should represent nothingness, something that nothing useful can be done with.

Backward Compatibility

This is a library-only proposal, thus it has no impact on the language.

This is strictly an addition to the `<utility>` header. The `<variant>` header is not modified in any way per this proposal, thus requiring no change in existing codebases. Nor are any other parts of `<utility>` modified.

Proposed Wording[1]

Add the following lines to the end of [utility.syn] paragraph 1 prior to the final `}`.

```
// 22.6.8, class monostate
struct monostate;

// 22.6.9, monostate relational operators
constexpr bool operator==(monostate, monostate) noexcept;
```

[1] This wording strategy is to copy the relevant portions of [variant.syn] to [utility.syn]. Also considered, but ultimately rejected in favor of this approach, was adding the following paragraph to the end of [variant.monostate]: In addition to being available via inclusion of the <variant> header, the class monostate, and its associated relational operators, are available when the <utility> header is included.

```
  constexpr strong_ordering operator<=>(monostate, monostate)
noexcept;

  // 22.6.12, hash support
  template<class T> struct hash;
  template<> struct hash<monostate>;
```

Conclusion

`std::monostate` is a generally useful type and therefore belongs in a more appropriate header than `<variant>`. We are proposing adding it to `<utility>`, while preserving `<variant>` as is for backward compatibility.