std::monostate_function<>

ISO/IEC JTC1 SC22 WG21
Document Number: P0477R0
Audience: Library Evolution Working Group
Matt Calabrese (metaprogrammingtheworld@gmail.com)
2016-10-14

Abstract

This paper presents a template named `std::monostate_function` that may be instantiated with a function pointer or member function pointer, producing the type of a `function object` that invokes the corresponding `Callable`. The primary use-cases of this facility are as the `Comparator` for an associative container, and as the deleter of a `std::unique_ptr`.

Motivation

The standard library has a number of type templates that take a `function object` as a template parameter. The instantiated type of such a template often has one or more constructors that allow for initialization of an instance of that `Callable`. Examples of this pattern include but are not limited to `std::set`, `std::map`, `std::priority_queue`, and `std::unique_ptr`.

In the case where a user wishes to specify a function pointer as the `Callable` with such a template, that user must provide the function pointer type and also pass in the function pointer itself through the constructor of the instantiation. For example:

```
bool custom_compare(int, int) { /*...*/ }  // Pre-existing function

std::set<int, decltype(&custom_compare)> ints(&custom_compare);
```

There are numerous drawbacks to this approach:

1. The user must properly specify a function pointer type when instantiating the template.

2. The user must explicitly pass in the desired function pointer value during construction of the encapsulating type.

3. Forgetting to pass the value during construction would result in undefined behavior if the `Callable` were to be invoked.

4. The instantiation may be suboptimal since an implementation is less-likely to inline calls to that function.

5. There will be an increase in size of the type due to the need to contain a function pointer.

6. Specification of a member function pointer is not directly possible.

An alternative to the above approach is to create a new, empty function object type that forwards along its parameters to the existing function. This type is then used instead of the function pointer directly. For example:

```
struct custom_compare_t
{
  bool operator()(int lhs, int rhs) const { return custom_compare(lhs, rhs); }
};

std::set<int, custom_compare_t> ints;
```

The creation of a type such as `custom_compare_t` is mechanical. An equivalent type can instead be concisely generated by a template, thanks in part to a new C++ language facility: declaring non-type template arguments with `auto` [1]. With the template `std::monostate_function` that is proposed in this paper, the above user-created `custom_compare_t` type may be replaced by the template instantiation `std::monostate_function<&custom_compare>`.

Synopsis

```
namespace std {

template<auto Callable>
struct monostate_function
{
  using value_type = decltype(Callable);
  static value_type constexpr value = Callable;

  // Invoke Callable with the specified parameters, returning the result.
  // Requires: Callable is a standard Callable for parameter types P&&...
  template<class... P>
  constexpr std::result_of_t<value_type(P&&...)> operator()(P&&... args) const;
};

template< auto Callable >
constexpr bool operator==
(monostate_function<Callable>, monostate_function<Callable>) noexcept;

template< auto Callable >
constexpr bool operator!=
(monostate_function<Callable>, monostate_function<Callable>) noexcept;

}
```

Possible Implementation

```
namespace std {

template<auto Callable>
struct monostate_function
{
  using value_type = decltype(Callable);
  static value_type constexpr value = Callable;

  template<class... P>
  constexpr std::result_of_t<value_type(P&&...)> operator()(P&&... args) const
  noexcept(noexcept(std::invoke(Callable, std::declval<P>()...)))
  {
    return std::invoke(Callable, std::forward<P>(args)...);
  }
};

template<auto Callable>
constexpr bool operator==
(monostate_function<Callable>, monostate_function<Callable>) noexcept
{
  return true;
}

template<auto Callable>
constexpr bool operator!=
(monostate_function<Callable>, monostate_function<Callable>) noexcept
{
  return false;
}

}
```

Additional Considerations

This proposal does not require the `operator()` of `std::monostate_function` instantiations to be conditionally `noexcept`. This is only because it is based on existing practice with respect to the application of `noexcept` to standard library facilities. The author believes that this function should be required to be conditionally `noexcept` and so should `std::invoke`, though that recommended divergence regarding specification of `noexcept` is beyond the scope of this paper.

For completeness, in addition to the operations described above, instantiations of `std::monostate_function` could be fully comparable and hashable, as well. Relational operations, specifically, are left out of this paper only due to anticipation of recommendations fom LEWG and LWG regarding how preceisely such operations should be specified in the next C++ standard. It is unclear at this time whether recommended practice for new facilities such as this will be to define overloaded relational operators or to define a specialization of `std::default_order`. The choice may be further influenced by a possible acceptance of a default comparisons proposal, such as a revision of "Default comparisons"[2], and so this proposal only focuses on the essential functionality for the time being.

Acknowledgments

Thanks to James Touton and Mike Spertus for their work on P0127[1].

References

[1] James Touton, Mike Spertus, Symantec: "Declaring non-type template parameters with `auto`" P0127r2 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0127r2.html

[2] Bjarne Stroustrup: "Default comparisons" N4475 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4475.pdf