Explicit return type deduction for `std::numeric_limits` and `<numbers>`

Thomas Mejstrik

December 19, 2024

1 Revision history

1.1 R0

Current revision

2 Motivation

In spite of the *always auto* movement, their are some places where `auto` cannot be used, leading to the need to write types multiple times. Two places where `auto` cannot be used are

- non-static data members in types, and
- type specifiers in function declarations.

This paper proposes

- a new tag type `std::deduce`, which can be used to indicate that return type deduction shall take place and to eliminate the need for writing out type names multiple times.

Although return type deduction in C++ is possible, it is rarely used, since it is not very C++ like, can easily be overseen where it happens, and thus can easily be used wrongly. By introducing a tag `std::deduce` this is solved. Indeed, the tag clearly indicates that return type deduction shall occur, and it is easy to search for the tag in source code.

2.1 Tony Tables

```
/* (1) - For structs */
struct F {
  float f1 = std::numeric_limits< float >::max();         // Before
  float f2 = std::numeric_limits< std::deduce >::max();   // After
};

/* (2) - For functions */
void func1( float f = std::numeric_limits< float >::max() );         // Before
void func2( float f = std::numeric_limits< std::deduce >::max() );   // After
```

The latter gets much easier to read, especially when the type (here: `float`) is not that simple, but a rather complicated, maybe user defined, type.

3 Implementation proposal

This proposal can be implemented as a library extension. A specialization of the types in `std::numeric_limits` with the type `std::deduce` is added An example is given for `numeric_limits::max`:

```
namespace std {
  template<>
  struct numeric_limits< std::deduce > {
    struct max {
      template< typename Out >
      operator Out() { return numeric_limits< Out >::max(); }
    };
    /* ... */
  };
}
```

For the mathematical constants the following code is a possible implementation. We present the code for the fictitious constant `M_ONES`.

```
inline constexpr auto M_ONES = 1.11111111111111111111111111111111111111L;

template< typename T = double >
struct m_ones_v_c {
    constexpr operator T() const {
        return M_ONES;
    }
};
template<> struct m_ones_v_c< tt::deduce > {
    template< typename Out >
    constexpr operator Out() const {
        return M_ONES;
    }
};
template< typename T > inline constexpr auto m_ones =  m_ones_v_c< T >{};
inline constexpr auto m_ones =  m_ones_v_c< double >{};
```

This implementation is fully compatible with the current implementation of the mathematical constants.

4 Header

The tag type `std::deduce` should go into the header `<utility>`. This way, users can easily also use this type for their own functions, without the need to include `<numeric_limits>` or `<numbers>`.

5 Not suggested variants

5.1 `auto` for non-static data members

In this paper we do not suggest to allow `auto` for non-static data members, because

- it does not solve point (2).
- it is more or less already rejected, see N3897.

```
/* (1) */
struct F {
  float f1 = std::numeric_limits< float >::max();   // Before
  auto   f0 = std::numeric_limits< float >::max();   // Not suggested
};

/* (2) */
void func1( float f = std::numeric_limits< float >::max() );     // Before
// void func0( auto f = std::numeric_limits< float >::max() );   // senseless
```

5.2 `auto` instead of `std::deduce`

Instead of introducing a new tag type, one could also allow `auto` to be used in template instantiations. This proposal seems to be rejected already - There is a note about this in P0849.

```
/* (1), (2) is similar */
struct F {
  float f1 = std::numeric_limits< float >::max();   // Before
  float f3 = std::numeric_limits< auto >::max();    // Not suggested
};
```

5.3 void instead of std::deduce

Instead of introducing a new tag type, one could also overload the classes for type void, similar as it is done for std::less. We do not propose this for the following reasons:

- It's hard to search for it in the source code
- It is not obviously clear that return type deduction shall take place.
- It's not consistent with std::less: Indeed, void in std::less is used to make the function deduce its instantiation from the **input** type, where std::deduce is used to deduce its instantiation from the **output** type

```
/* (1), (2) is similar */
struct F {
  float f1 = std::numeric_limits< float >::max();  // Before
  float f4 = std::numeric_limits< void >::max();   // Not suggested
};
```

5.4 Empty brackets instead of std::deduce

This is similar to the void case above, since std::less's template parameter defaults to void. Furthermore, this is even more less search-and-findable then the case above.

```
/* (1), (2) is similar */
struct F {
  float f1 = std::numeric_limits< float >::max();  // Before
  float f4 = std::numeric_limits<>::max();   // Not suggested
};
```