Abstract

This paper provides wording in reply to NB comments suggesting to adopt P1636R2 [1] (Formatters for library types) and to add formatters for std::stacktrace.

History

LEWG approved P1636R2 [1] in 2019 (Cologne) for C++20. The paper was subsequently reviewed in 2021 by LWG, who requested wording changes.

SG16 had significant concerns with the formatting of filesystem::path and asked for that formatter to be removed.

Another paper P2197R0 [2] explored different options for formatting std::complex but was not pursued, nor was it warmly received by LEWG when presented (summer 2020 telecon).

P1636R2 [1] has been stuck in need of a revision and attempts to contact the author have failed.

Design

This paper provides wording for

- std::thread::id
- std::basic_stacktrace
- std::stacktrace_entry

Note that P1636R2 [1] additionally proposed to support

- std::complex
- std::bitset
- std::error_code
- std::unique_ptr
Why do we need thread::id formatting in C++23?

Two reasons. First, it is very commonly used by loggers. But most importantly that information is not exposed by any other means than an ostream << overload. There is no accessor of any kind, so the only well-defined way to extract a thread::id is to use streams

```cpp
std::ostringstream ss;
ss << thread.get_id();
std::print("called a nice API on thread ", ss.str());
```

Note that a quick search on GitHub reveals that users, when their expectations are subverted, will find a way, and won't let such thing as well-definedness stop them. They will, for example, exploit the amazing flexibility of printf to get what they want:

```c
printf("[Thread %d Profiling: %ld microseconds] ",
    std::this_thread::get_id(), microseconds); // UB
```

By properly supporting thread::id in format, we can avoid the proliferation of undefined, non portable, and dangerous code.

Why not std::complex?

Formatting of std::complex is more... complex. In particular, P1636R2 [1] (which, to be fair, was approved by LEWG) proposed to use the same notation as ostream. Given the wide variance in how complex are formatted in other programming languages and the interaction with locales (including the need to support the L specifier), it seems wise to punt this question to a later C++ version.

<table>
<thead>
<tr>
<th></th>
<th>(1.0, 2.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1.0, 2.0)</td>
</tr>
<tr>
<td>P1636R2</td>
<td>(1.0, 2.0)</td>
</tr>
<tr>
<td>P2197R0</td>
<td>(1.0, 2.0i)</td>
</tr>
<tr>
<td>Rust</td>
<td>1+2i</td>
</tr>
<tr>
<td>Python</td>
<td>1+2j</td>
</tr>
<tr>
<td>Julia</td>
<td>1.0+ 2.0im</td>
</tr>
<tr>
<td>Nim</td>
<td>(1.0, 2.0)</td>
</tr>
<tr>
<td>R</td>
<td>1+2i</td>
</tr>
<tr>
<td>Swift</td>
<td>(1.0, 2.0)</td>
</tr>
<tr>
<td>C#</td>
<td>1 + 2i</td>
</tr>
</tbody>
</table>

Why not std::error_code/bitset/smart pointers?

These don't seem sufficiently useful to be processed as part of NB comments.

Moreover, there were plans to remove smart pointer formatters from P1636 for consistency with raw pointers which are intentionally not formattable by default.
Stacktrace

We propose adding formatters for std::stacktrace and std::stacktrace_entry in addition to existing std::to_string overloads such that the following would be equivalent:

to_string(a_stacktrace);
std::format("{}, a_stacktrace);

Do we really need both? Beside arithmetic types, the only other type to have a std::to_string method is std::bitset.

Should stacktrace be formatted as a range?

This would add a lot of complexity to something that would probably never be used. The range behavior can be opt-in this way instead:

std::format("{}", std::views::all(a_stacktrace));

Wording

Header <stacktrace> synopsis [stacktrace.syn]

#include <compare>  // see ??

namespace std {

  // ??, class stacktrace_entry
  class stacktrace_entry;

  // ??, class template basic_stacktrace
  template<class Allocator>
  class basic_stacktrace;

  // basic_stacktrace typedef-names
  using stacktrace = basic_stacktrace<allocator<stacktrace_entry>>;

  // ??, non-member functions
  template<class Allocator>
  void swap(basic_stacktrace<Allocator>& a, basic_stacktrace<Allocator>& b)
  noexcept(noexcept(a.swap(b)));

  string to_string(const stacktrace_entry& f);

  template<class Allocator>
  string to_string(const basic_stacktrace<Allocator>& st);

  template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const stacktrace_entry& f);
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const basic_stacktrace<Allocator>& st);

// ??, formatting support
template<> struct formatter<stacktrace_entry>;
template<class Allocator> struct formatter<basic_stacktrace<Allocator>>;

namespace pmr {
    using stacktrace = basic_stacktrace<polymorphic_allocator<stacktrace_entry>>;
}

// ??, hash support
template<class T> struct hash;
template<> struct hash<stacktrace_entry>;
template<class Allocator> struct hash<basic_stacktrace<Allocator>>;

Effects: Equivalent to: return os << to_string(st);

[...]
Concurrency support library [thread]

Class thread::id [thread.thread.id]

namespace std {
    class thread::id {
        public:
            id() noexcept;
    };

    bool operator==(thread::id x, thread::id y) noexcept;
    strong_ordering operator<=>(thread::id x, thread::id y) noexcept;

    template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& out, thread::id id);

    // hash support
    template<class T> struct hash;
    template<> struct hash<thread::id>;

    template<class charT>
    struct formatter<thread::id, charT>;
}

An object of type thread::id provides a unique identifier for each thread of execution and a single distinct value for all thread objects that do not represent a thread of execution. Each thread of execution has an associated thread::id object that is not equal to the thread::id object of any other thread of execution and that is not equal to the thread::id object of any thread object that does not represent threads of execution.

The text representation for the character type charT of an object of type thread::id is an unspecified value such that, for two objects of type thread::id x and y, if x == y the thread::id objects have the same text representation and if x != y the thread::id objects have distinct text representations.

thread::id is a trivially copyable class. The library may reuse the value of a thread::id of a terminated thread that can no longer be joined.

Let P(x, y) be an unspecified total ordering over thread::id as described in ??.

Returns: strong_ordering::less if P(x, y) is true. Otherwise, strong_ordering::greater if P(y, x) is true. Otherwise, strong_ordering::equal.

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& out, thread::id id);

Effects: Inserts an unspecified the text representation of id into out. For two objects of type thread::id x and y, if x == y the thread::id objects have the same text representation and if x != y the thread::id objects have distinct text representations.
Returns: out.

```cpp
template<class charT> class formatter<thread::id, charT>;
```

formatter<thread::id, charT> interprets format-spec as a thread-id-format-spec. The syntax of format specifications is as follows:

```
thread-id-format-spec:
  fill-and-align_opt width_opt
```

[Note: The productions fill-and-align and width are described in [format.string]. — end note]

If the align option is omitted it defaults to >.

A thread::id object is formatted by writing its text representation to the output with thread-id-format-spec applied.

**Feature test macro**

[Editor's note: define __cpp_lib_formatters set to the date of adoption in <version>, <stacktrace> and <thread>].

**Acknowledgments**

Thanks to Lars Gullik Bjønnes for their initial work on P1636!

**References**

