Formatters for library types

Document #: P1636R2
Date: 2019-10-06
Project: Programming Language C++
Library
Reply-to: Lars Gullik Bjønnes
<lbjonnes@cisco.com>

0 Revisions

0.1 R2

— more work on wording

0.2 R1

— updates LEWG comments and votes
  — `unique_ptr` and `shared_ptr` should format via cast-to-void*. (If no, then formatting `unique_ptr` and `shared_ptr` is unsupported.)

SF	F	N	A	SA
0	3	5	3	1

  — Throw `invalid_format` (ed. `format_error`) when encountering a leading 0 in the width specifier for complex.

SF	F	N	A	SA
0	1	2	7	0

  — Update paper with "basic_streambuf is not supported by format."
  `unanimous`
  — Make the above clarifications and forward to LWG for C++20.

SF	F	N	A	SA
3	7	0	0	0

— add snippet for each of the formatters
— added `complex` examples
— also base on P1652
— added feature macro
— use `ctx.locale()` in `complex<>::formatter` "Equivalent to"
— use "Equivalent to" instead of "As implemented by"
— some wording/spelling fixes

0.3 R0

— initial proposal

1 Introduction

After [P0645] and [P1361] almost all types that have a output stream `operator<<` overload will also have a `std::formatter` specialization. The following types, which have output stream operators, are missing this specialization:

— `basic_streambuf`
— `bitset`
— `complex`
— `error_code`
— `filesystem::path`
— `shared_ptr`
— `sub_match`
— `thread::id`
— `unique_ptr`

Adding formatter specializations for all or most of these will reduce surprises for users that convert from stream centric output to format centric output:

Before

```cpp
std::filesystem::path p;
os << p; // OK
std::format("{}", p); // error
std::format("{}", p.string()); // OK
```

After

```cpp
std::filesystem::path p;
os << p; // OK
std::format("{}", p); // OK
std::format("{}", p.string()); // OK
```

And similar for most types, but also:

Before

```cpp
std::complex<double> c;
os << c; // OK
std::format("{}", c); // error
std::format("({},{})",
            c.real(), c.imag()); // OK
```

After

```cpp
std::complex<double> c;
os << c; // OK
std::format("{}", c); // OK
std::format("({},{})",
            c.real(), c.imag()); // OK
```

1.1 Some considerations:

1.1.1 `error_code`

Formatted as string, equivalent to using streams for output:

```cpp
basic_ostringstream<charT> o;
o.imbue(locale.classic());
o << ec;
```

1.1.2 `bitset`

`bitset operator<<` uses locale, for that reason this proposal suggest `b.template to_string<charT>()` as the wanted output instead of via streams.

1.1.3 `unique_ptr` and `shared_ptr`

`unique_ptr` and `shared_ptr` use `.get()` in `operator<<`, so one option is to not add formatters for those since `format` formatters are disabled for pointer types in general. The proposal adds formatters for the smart-pointers, but where the result from `.get()` is cast to `void*` before output.

The reason for `format` to not automatically formatting a pointer as its pointer value have been a combination of:

— Avoiding unwanted conversions that would result in conversions to `bool`.
— To prevent formatting when you accidentally pass a pointer instead of an object.

1.1.4 `complex`

This proposal does not suggest to use `ostream` to format `complex`, since stream output of `complex` takes locale into account by default, which is probably not wanted. Also using stream rules to output the `.imag()` and `.real()` parts does not follow how `format` outputs floats. `complex{1.0, 2.0}`, output with streams gives “`(1,2)`”, whereas `format("({},{})", c.imag(), c.real()` gives “`(1.0,2.0)`”. In addition we would loose the possibility to control the floating point type. We have three options when deciding what format specification to use for outputting complex:

1. Use the `std-format-spec`, with the slight change that `["0"]` is ignored, and that the default is right alignment (`>`) as for arithmetic types. The complex number is formatted into a string “(*<real>*,*<imag>*)”. The inner `<real>` and `<imag>` parts will use `[sign] ['#'] ['.' precision] [type]` from the spec and the outer “`(..)`” part will use `[[fill] align] [width]` when creating the formatted output.
2. Use a `complex-format-spec`, almost identical to `std-format-spec`, with `["0"]` not part of the spec.
3. Use a more elaborate `complex-format-spec` with the ability to control the formatting of all parts of the complex number. This means creating a format specification that would be unfamiliar to most.

This proposal goes with 1. or 2., wanting some guidance if the `["0"]` should be allowed and ignored or just not allowed at all. The third option can be done later, the format can be extended in a compatible fashion.

1.1.4.1 Examples

```
std::complex c{1.0, 2.0};

"(1.0,2.0)"       == std::format("{}", c);

"(-1.0,-2.0)"     == std::format("{:-}", -c);
"(+1.0,+2.0)"     == std::format("{:+}", c);
"( 1.0, 2.0)"     == std::format("{: }", c);

" (1.0,2.0)"      == std::format("{:10}", c);
"      (1.0,2.0)" == std::format("{:15}", c);

"(1.00,2.00)"     == std::format("{:.3}", c);

"(1.0,2.0)######" == std::format("{:#<15}", c);
"######(1.0,2.0)" == std::format("{:#>15}", c);
"###(1.0,2.0)###" == std::format("{:#^15}", c);

"(+1.000,+2.000)#####" == std::format("{:#<+#20.3f}", c);

"(1.00,2.00)"     == std::format("{:.{}}", c, 3);

"      (1.0,2.0)" == std::format("{:{}}", c, 15);

"    (1.00,2.00)" == std::format("{:15.3}", c);
"    (1.00,2.00)" == std::format("{:{}.{}}", c, 15, 3);
```

1.1.5 `filesystem::path`

Formatted as string, equivalent to using streams for output:

```
basic_ostringstream<charT> o;
o.imbue(locale::classic());
o << p;
```

This means that the output will be quoted.

1.1.6 `sub_match`

Uses the `sub_match::str()` for output.

1.1.7 `thread::id`

Formatted as string, equivalent to using streams for output:

```
basic_ostringstream<charT> o;
o.imbue(locale::classic());
o << id;
```

1.1.8 `basic_streambuf`

Leave the `basic_streambuf` alone and let ostream handle that. The result of this is that `basic_streambuf` is not supported by `format`.

1.2 Proposal

Add `formatter` specializations for:

- `error_code`
- `bitset`
- `unique_ptr`
- `shared_ptr`
- `complex`
- `filesystem::path`
- `sub_match`
- `thread::id`

1.2.1 Design questions

Use std-format-spec with formatting rules specific for `complex`? Use a complex-format-spec instead of a std-format-spec? Want a more elaborate complex-format-spec, with control over more parts?

Add the formatters for `unique_ptr` and `shared_ptr` even if formatting of pointers are normally disabled in `std::format`?

1.3 Impact on the standard

This proposal is a pure library extension.

1.4 Feature macro

If a feature macro is required for this,

___cpp_lib_formatters

is suggested.

1.5 Implementation

All of the formatter specializations in this paper has been implemented on top of [{fmt}].

2 Proposed Wording

Wording is relative to [N4810] with [P0645], [P1361] and [P1652] applied.

2.1 Feature macro

Add to 17.3.1 [support.limits] table 36:

Macro name	Value	Header(s)
__cpp_lib_formatters	202002L	`<system_error>` `<bitset>` `<memory>` `<complex>` `<filesystem>` `<regex>` `<thread>`

2.2 error_code

Add to 19.5.1 Header `<system_error>` synopsis [system.error.syn] just below `operator<<`:

```
// 19.5.3.6, formatter
template<class charT> struct formatter<error_code, charT>;
```

Add to 19.5.3 1 Overview [syserr.errcode.overview], just below `operator<<`:

```
// 19.5.3.6, formatter
template<class charT> struct formatter<error_code, charT>;
```

Add a new section 19.5.3.6 Formatter [syserr.errcode.fmt]

19.5.3.6 Formatter **[syserr.errcode.fmt]**

```
template<class charT>
  struct formatter<error_code, charT> : formatter<basic_string<charT>, charT> {
    template<class FormatContext>
      typename FormatContext::iterator
        format(error_code ec, FormatContext& ctx);
};
```

1 The specialization `formatter<error_code, charT>` meets the *Formatter* requirements (20.20.4.1).

```
template<class FormatContext>
  typename FormatContext::iterator
    format(error_code ec, FormatContext& ctx);
```

2 *Effects:* Equivalent to:

```
basic_ostringstream<charT> o;
o.imbue(locale::classic());
o << ec;
return formatter<basic_string<charT>, charT>::format(o.str(), ctx);
```

2.3 bitset

Add to 20.9.1 Header `<bitset>` synopsis [bitset.syn], just below `operator<<`:

> // 20.9.5, *bitset formatter*

```
template<size_t N, class charT> struct formatter<bitset<N>, charT>;
```

Add a new section: 20.9.5 bitset formatter [bitset.fmt]

20.9.5 bitset formatter **[bitset.fmt]**

```
template<size_t N, class charT>
  struct formatter<bitset<N>, charT> : formatter<basic_string<charT>, charT> {
    template<class FormatContext>
      typename FormatContext::iterator
        format(const bitset<N>& b, FormatContext& ctx);
};
```

1 The specialization `formatter<bitset<N>, charT>` meets the *Formatter* requirements (20.20.4.1).

```
template<class FormatContext>
  typename FormatContext::iterator
    format(const bitset<N>& b, FormatContext& ctx);
```

2 *Effects:* Equivalent to:

```
return formatter<basic_string<charT>, charT>::format(b.template to_string<charT>(), ctx);
```

2.4 unique_ptr

Add to 20.10.2 Header `<memory>` synopsis [memory.syn] just after `operator<<`:

```
template<class T, class D, class charT> struct formatter<unique_ptr<T, D>, charT>;
```

Add a new section 20.11.1.7 Formatter [unique.ptr.fmt]

20.11.1.7 Formatter **[unique.ptr.fmt]**

```
template<class T, class D, class charT>
  struct formatter<unique_ptr<T, D>, charT> : formatter<const void*, charT> {
    template<class FormatContext>
      typename FormatContext::iterator
        format(const unique_ptr<T, D>& p, FormatContext& ctx);
};
```

1 The specialization `formatter<unique_ptr<T, D>, charT>` meets the *Formatter* requirements (20.20.4.1).

```
template<class FormatContext>
  typename FormatContext::iterator
    format(const unique_ptr<T, D>& p, FormatContext& ctx);
```

2 *Constraints:*

```
  const_cast<
    add_const_t<typename unique_ptr<T, D>::pointer>(p.get()))
```

to be a valid expression.

3 *Effects:* Equivalent to:

```
return formatter<const void*, charT>::
         format(const_cast<
             add_const_t<typename unique_ptr<T, D>::pointer>>(p.get()), ctx);
```

2.5 shared_ptr

Add to 20.10.2 Header `<memory>` synopsis [memory.syn] just after `operator<<`:

```
template<class T, class charT> struct formatter<shared_ptr<T>, charT>;
```

Add a new section 20.11.3.12 Formatter [util.smartptr.shared.fmt]

20.11.3.12 Formatter **[util.smartptr.shared.fmt]**

```
template<class T, class charT>
  struct formatter<shared_ptr<T>, charT> : formatter<const void*, charT> {
    template<class FormatContext>
      typename FormatContext::iterator
        format(const shared_ptr<T>& p, FormatContext& ctx);
};
```

1 The specialization `formatter<shared_ptr<T>, charT>` meets the *Formatter* requirements (20.20.4.1).

```
template<class FormatContext>
  typename FormatContext::iterator
    format(const shared_ptr<T>& p, FormatContext& ctx);
```

2 *Effects:* Equivalent to:

```
return formatter<const void*, charT>::
         format(const_cast<
            add_const_t<
              add_pointer_t<typename shared_ptr<T>::element_type>>>(p.get()), ctx);
```

2.6 complex

Add to 26.4.1 Header `<complex>` synopsis [complex.syn] just below `operator<<`:

```
// 26.4.?, formatter
template<class T, class charT> struct formatter<complex<T>, charT>;
```

Add a new section 26.4.? Formatter [complex.fmt]:

26.4.? Formatter **[complex.fmt]**

```
template<class T, class charT>
  struct formatter<complex<T>, charT> {
    typename basic_format_parse_context<charT>::iterator
      parse(basic_format_parse_context<charT>& ctx);
    template<class FormatContext>
      typename FormatContext::iterator
        format(const complex<T>& c, FormatContext& ctx);
};
```

1 The specialization `formatter<complex<T>, charT>` meets the *Formatter* requirements (20.20.4.1).

```
typename basic_format_parse_context::iterator
  parse(basic_format_parse_context&);
```

2 *Effects:* Interprets the *format-spec* as a *std-format-spec* for `T`, except:

- — right alignment ('>') as default as for arithmetic types
- — zero ('0') preceding the width field is ignored

```
template<class FormatContext>
  typename FormatContext::iterator
   format(const complex<T>& c, FormatContext& ctx);
```

3 *Effects:* As if implemented by:

```
// pseudo-code
std::basic_string<charT> s =
  std::format(ctx.locale(),
              "({:" complex_format_spec "},"
              "{:" complex_format_spec "})",
              c.real(), c.imag());
 return std::format("{:" string_format_spec "}", s, ctx);
```

Where *complex_format_spec* is built up using the *sign*, *alternate*, *precision* and *type*, and *string_format_spec* is built using the *fill*, *align*, and *width*, both specs with info aquired from the `parse` member function.

4 [*Example*:

```
  complex c{1.0, 2.0};

  format("{}", c);           // (1.0,2.0)
  format("{:#>15}", c);      // ######(1.0,2.0)
  format("{:#<+#20.3f}", c); // (+1.000,+2.000)#####
```

— *end example*]

2.7 filesystem::path

Add to 29.11.5 Header `<filesystem>` synopsis [fs.filesystem.syn] (location up to editors discretion):

```
// 29.11.7.8, formatter
template<class charT> struct formatter<filesystem::path, charT>;
```

Add a new section 29.11.7.8 Formatter [fs.path.fmt]

29.11.7.8 Formatter **[fs.path.fmt]**

```
template<class charT>
  struct formatter<filesystem::path, charT> : formatter<basic_string<charT>, charT> {
    template<class FormatContext>
      typename FormatContext::iterator
        format(const filesystem::path& p, FormatContext& ctx);
};
```

1 The specialization `formatter<filesystem.path, charT>` meets the *Formatter* requirements (20.20.4.1).

```
template<class FormatContext>
  typename FormatContext::iterator
    format(const filesystem::path& p, FormatContext& ctx);
```

2 *Effects:* Equivalent to:

```
basic_ostringstream<charT> o;
o.imbue(locale::classic());
o << p;
return formatter<basic_string<charT>, charT>::format(o.str(), ctx);
```

2.8 sub_match

Add to 30.4 Header `<regex>` synopsis [re.syn] after the `operator<<`:

```
// 30.9.3, formatter
template<class BiIter, class charT> struct formatter<sub_match<BiIter>, charT>;
```

Add a new section 30.9.3 Formatter [re.submatch.fmt]

30.8.3 Formatter **[re.submatch.fmt]**

```
template<class BiIter, class charT>
  struct formatter<sub_match<BiIter>, charT> : formatter<basic_string<charT>, charT> {
    template<class FormatContext>
      typename FormatContext::iterator
        format(const sub_match<BiIter>& s, FormatContext& ctx);
};
```

1 The specialization `formatter<sub_match<BiIter>, charT>` meets the *Formatter* requirements (20.20.4.1).

```
template<class FormatContext>
  typename FormatContext::iterator
    format(const sub_match<BiIter>& s, FormatContext& ctx);
```

2 *Constraints:* `is_same<sub_match<BiIter>::value_type, charT>`

3 *Effects:* Equivalent to:

```
return formatter<basic_string<charT>, charT>::format(s.str(), ctx);
```

2.9 thread::id

Add to 32.3.2.1 Class `thread::id` [thread.thread.id] just after `operator<<`:

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

Add a new paragraph 14 just after paragraph 13:

```
template<class charT>
  struct formatter<thread::id, charT> : formatter<basic_string<charT>, charT> {
    template<class FormatContext>
      typename FormatContext::iterator
        format(thread::id id, FormatContext& ctx);
};
```

14 The specialization `formatter<thread::id, charT>` meets the *Formatter* requirements (20.20.4.1).

```
template<class FormatContext>
  typename FormatContext::iterator
    format<thread::id id, FormatContext& ctx);
```

15 *Effects:* Equivalent to:

```
basic_ostringstream<charT> o;
o.imbue(locale::classic());
o << id;
return formatter<basic_string<charT>, charT>::format(o.str(), ctx);
```

3 Acknowledgements

Victor Zverovich, Jonathan Wakely and Juan Alday for looking through the draft and giving valuable comments and directions.

4 References

[{fmt}] Zverovich Victor. A modern formatting library.
https://github.com/fmtlib/fmt

[N4810] Richard Smith. Working draft, Standard for Programming Language C++.
http://wg21.link/N4810

[P0645] Victor Zverovich. Text Formatting.
http://wg21.link/P0645

[P1361] Victor Zverovich, et al. Integration of chrono with text formatting.
http://wg21.link/P1361

[P1652] Zhihao Yuan, et al. Printf corner cases in std::format.
http://wg21.link/P652