

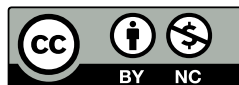
From << to std::formatter

Printing in C++20 era

Core C++ Meetup, May 2024

Yehezkel Bernat

YehezkelShB@gmail.com, @YehezkelShB



Intro

- Adding support for printing our own types is important
- As standard printing utils change, we want to adjust too

iostream – reminder

- To support code like:

```
1  MyType obj;  
2  std::cout << obj << '\n';
```

- We overload the operator:

```
3  std::ostream& operator<<(std::ostream& os,  
                           const MyType& obj);
```

std::format (C++20) / std::print (C++23)

- std::format (outputs a std::string) and std::print use different syntax

```
1  std::print("Hello, {}!\n", "world");  
2  std::puts(std::format("Answer: {}!",  
                        42).c_str());
```

- Reminds printf() but different
 - Format string with placeholders
 - Then, variadic count of arguments
 - But type safe, compile-time checked (and different syntax)

Formatting our type – basic case

```
class MyString
{
public:
    MyString(const std::string& str);

private:
    std::string m_str;
    friend std::formatter<MyString>;
};
```

Formatting our type – basic case (2)

```
template <>
struct std::formatter<MyString>
    : std::formatter<std::string_view> {
    using Base = std::formatter<std::string_view>;

    template <typename FormatContext>
    auto format(const MyString& str,
                FormatContext& ctx) const {
        return Base::format(str.m_str, ctx);
    }
};
```

Formatting our type – basic case (3)

```
MyString str("Hello, world!");  
std::print("{}\n", str);
```

Basic done wrong

```
template <>
struct std::formatter<MyString>
    : std::formatter<std::string_view> {
    auto format(const MyString& str, auto& ctx) const {
        return std::format_to(ctx.out(), "{}", str.m_str);
    }
};
```

[illegible]

Basic done wrong – fixed

```
template <>
struct std::formatter<MyString> { // no inheritance
    template <typename ParseContext>
    constexpr auto parse(ParseContext& ctx) {
        return ctx.begin();
    }

    auto format(const MyString& str, auto& ctx) const {
        return std::format_to(ctx.out(), "{}", str.m_str);
    }
};
```

Support custom format specifiers – parse

```
template <> struct std::formatter<Complex> {  
    constexpr auto parse(auto& ctx) {  
        auto it = ctx.begin();  
        if (it == ctx.end() or *it == '}') return it;  
  
        if (*it++ == 'p') m_polar = true;  
        else throw std::format_error("invalid format");  
  
        return it;  
    }  
};
```

Support custom format specifiers – format

```
template <> struct std::formatter<Complex> {
    auto format(const Complex& complex, auto& ctx) const {
        if (m_polar)
            return std::format_to(ctx.out(), "({} * e^({}i))",
                                   complex.radius(), complex.angle());

        return std::format_to(ctx.out(), "({} + {}i)",
                               complex.real(), complex.imag());
    }
};

std::print("{:p}\n", complex);
```

Support standard format specifiers – parse

```
std::print("{:f}\n", complex);
```

```
template <> struct std::formatter<Complex> {  
    std::formatter<double> m_underlying;  
  
    constexpr auto parse(auto& ctx) {  
        return m_underlying.parse(ctx);  
    }  
};
```

Support standard format specifiers – format

```
auto format(const Complex& complex, auto& ctx) const {  
    auto out = std::format_to(ctx.out(), "(");  
    ctx.advance_to(out);  
    out = m_underlying.format(complex.real(), ctx);  
    out = std::format_to(out, " + ");  
    ctx.advance_to(out);  
    out = m_underlying.format(complex.imag(), ctx);  
    out = std::format_to(out, "i)");  
    return out;  
}
```

Standard + custom – parse

```
std::print("{:p:f}\n", complex);
```

```
constexpr auto parse(auto& ctx) {  
    auto it = ctx.begin(); auto end = ctx.end();  
    if (it == end || *it == '}') return it;  
    auto endOfCustom = std::ranges::find(it, end, ':');  
    if (it != endOfCustom) {  
        if (*it != 'p') throw std::format_error(...);  
        m_polar = true; ++it;  
    }  
    // ...  
}
```

Standard + custom – parse (2)

```
// ...  
    if (it == end || *it == '}') return it;  
    if (it != endOfCustom)  
        throw std::format_error("invalid format");  
  
    ctx.advance_to(++endOfCustom);  
    return m_underlying.parse(ctx);  
}
```

Support standard format specifiers – format

```
auto format(const Complex& complex, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "(");
    ctx.advance_to(out);
    out = m_underlying.format(m_polar ? complex.radius() :
                                complex.real(), ctx);
    out = std::format_to(out, "{}",
                        m_polar ? " * e^(" : " + ");
    ctx.advance_to(out);
    out = m_underlying.format(m_polar ? complex.angle() :
                                complex.imag(), ctx);
    return std::format_to(out, "{}", m_polar ? "i))" :
                                "i)");
```


Conclusions

- We get huge amount of flexibility and options
- With stream-based I/O, standard modifiers requires no work
 - `std::hex` manipulator, for example, affects the stream directly
- Writing our own manipulators is harder in stream-based I/O
- The simple cases are simple

References

- cppreference
- Barry Revzin
 - Formatting ranges paper – <https://wg21.link/p2286>
 - Rust vs C++ Formatting | Barry's C++ Blog – <https://brevzin.github.io/c++/2023/01/02/rust-cpp-format/>
 - The Surprising Complexity of Formatting Ranges in Cpp - Barry Revzin - CppCon 2022 – <https://www.youtube.com/watch?v=EQELdyecZlU>