Formatted output

P2093R8

Victor Zverovich (victor.zverovich@gmail.com)

Integrate text formatting (`std::format`) with standard I/O facilities

Motivation

Before

```
std::cout << std::format("Hello, {}!", name);
```

After

```
std::print("Hello, {}!", name);
```

- Better usability
- Avoid temporary `std::string`
- Avoid extra formatting call (`operator<<`)
- Less binary code

Changes since R7

- Added a reference to LLVM's `raw_ostream` that implements similar mojibake prevention mechanism to § 14 Implementation.

Changes since R6

- Added new SG16 poll results.
- Rebased the wording onto the latest draft, most importantly adding compile-time checks introduced by [P2216].
- Added "If `out` contains invalid code units, the behavior is undefined and implementations are encouraged to diagnose it." to the definition of `vprint_unicode` per SG16 feedback.
- Replaced "invalid code points are substituted with U+FFFD REPLACEMENT CHARACTER" with " implementations should substitute invalid code units with U+FFFD REPLACEMENT CHARACTER per The Unicode® Standard Version 13.0 – Core Specification, Chapter 3.9" in § 15 Wording per SG16 feedback.
- Added "The Unicode® Standard Version 13.0 – Core Specification" to Normative references in § 15 Wording.
- Clarified the behavior when mixing encodings in § 10 Unicode.

Using the literal encoding is the correct way of establishing encoding expectation in `std::format` and `std::print`

Literal encoding

Consistent with P2216 “std::format improvements” that has been merged into the standard.

With P2216 a format string must be known at compile time, normally be a string literal:

```
  template<class... Args>
    void print(format-string<Args...> fmt, const Args&... args);
```

Therefore the format string is normally in a literal encoding (possibly a subset).

Literal encoding

Both fmt and Microsoft STL (the only implementation shipping `std::format`) do parsing of the format string and width estimation based on whether literal encoding is UTF-8.

LWG3576 “Clarifying fill character in `std::format`”

`fill``:`
`    any ~~character~~codepoint of the literal encoding other than { or }`

If a format string is not in a literal encoding it may fail to parse.

Literal encoding

To be consistent with `std::format` the choice of encoding must be locale-independent.

The active code page and the terminal encoding are unrelated on popular Windows localizations such as Russian where the former is CP1251 while the latter is CP866.

Instead of assuming one encoding regardless of the string origin which would often result in mojibake, an explicit encoding indication, e.g.

```
print("Привет, {}!", locale_encoded(string_in_locale_encoding));
print("Привет, {}!", terminal_encoded(string_in_terminal_encoding));
```

This is already possible to implement via `formatter` specializations.

Literal encoding

Consistent with P2419 “Clarify handling of encodings in localized formatting of chrono types” and the resolution of LWG3565:

```
std::locale::global(std::locale("Russian.1251"));
auto s = std::format("День недели: {:L}", std::chrono::Monday);
```

Require implementations to make `std::chrono` substitutions with `std::format` as if transcoded to UTF-8 when the **literal encoding** E associated with the format string is UTF-8, for an implementation-defined set of locales.

SF	WF	N	WA	SA
1	6	2	0	0

Consensus: Consensus in favour.

Literal encoding

Consistent with P2419 “Clarify handling of encodings in localized formatting of chrono types” and the resolution of LWG3565:

```
std::locale::global(std::locale("Russian.1251"));
auto s = std::format("День недели: {:L}", std::chrono::Monday);
//                                          ^ using literal encoding
// s == "День недели: Пн"
```

Consider:

```
std::locale::global(std::locale("Russian.1251"));
std::print("День недели: {:L}", std::chrono::Monday);
//                               ^ using literal encoding
// Output: ???
```

We already use literal encoding for arguments. If we don’t do the same in print it may result in mojibake that P2419 tried to prevent.

Literal encoding

Consistent with P2419 “Clarify handling of encodings in localized formatting of chrono types” and the resolution of LWG3565:

```
std::locale::global(std::locale("Russian.1251"));
auto s = std::format("{:L}", std::chrono::Monday);
//                           ^ using literal encoding
// s == "Пн"
```

Consider:

```
std::locale::global(std::locale("Russian.1251"));
std::print("{:L}", std::chrono::Monday);
//                 ^ using literal encoding
// Output: ???
```

Note that the format string doesn’t have to contain non-ASCII for the output to be in UTF-8. It’s enough for the literal encoding to be UTF-8.

Implementation

- The proposed `print` function has been implemented in the open-source fmt library and has been in use for about 6 years.
- Rust's standard output facility uses essentially the same approach for preventing mojibake when printing to console on Windows. The main difference is that invalid code units are reported as errors in Rust.
- LLVM's `raw_ostream` also implements this approach when writing to console on Windows. The main difference is that in case of invalid UTF-8 it falls back on writing raw (not transcoded) data.