Formatting of `charN_t`

Document #:	P3258R0
Date:	2024-05-17
Programming Language C++	
Audience:	LEWG, SG16

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose to make `std::format("{}", u8"🌸")` well-formed.

Motivation

Despite having language support for Unicode character types since C++11 and C++20 (for `char8_t`), the standard library lacks support for these types, making adoption challenging. `std::format` is a great place to start.

Having the ability to format and print sequences of Unicode characters would offer more options when considering the type of strings produced by reflection.

```
int 鳥居;
std::format("{}", meta::name_of(^鳥居)).
```

But there are many other use cases, ie anyone who tries to use `charN_t` will need this facility. It is important for usability, and teachability that the standard library and core language complement each other. Integration of Unicode character types has been unsatisfactory so far.

Design

- `char8_t`, `char16_t`, `char32_t` characters and strings (NTBS, `std::string`, `std::string_view`, arrays, ranges) can be used as a formatted argument to `std::format`.
- They cannot be used as the type of the *format string*.
- We assume the encoding of the output is statically determined by the literal encoding of the *format-string*'s type. This is how `format` works already.
- From there, all string arguments are transcoded:
  - Transcoding a `char` string is a copy.
  - If the ordinary literal encoding is UTF-8, transcoding a `char8_t` string is a copy.

  - Otherwise, transcoding is an implementation-defined process that will inject replacement characters for invalid or unrepresentable characters.
  - Lossy conversions is the best we can achieve here.
- Mixing `char`, `wchar_t` is still not supported to avoid asking the question of the encoding of the argument.
- `std::format(L"{}", u8"")` is supported, as this is something Windows users are likely to find useful.
- Escaping Unicode strings (`format("{:?}", u8"こんにちは")`) works the same as escaping other strings, except characters that have no representation in the associated literal encoding of the format-string's character type are replaced by `\{xxxx}`.

Impact on `std::print`

There are 2 scenarios here:

- For a format string in UTF-N, `std::format` will convert a `charN_t` argument to UTF-N, which is lossless, and on a Unicode terminal, the Unicode API will preserve information (but extra work may be performed)
- For a format string not in `UTF-N`, if `vprint_unicode` is called **manually**, `charN_t` will be first converted to the literal encoding, and the whole formatted string might be converting back to Unicode, potentially losing information.

This is fine because calling `vprint_nonunicode` manually is a limited use case and trying to preserve information here would be extremely challenging: An implementation would have to transcode the formatting string as it is produced, which would be a very different behavior from `format`.

Ultimately `vprint_nonunicode` exists to support Windows terminals and should not be seen as a transcoding facility. Supporting `charN_t` *format-string* would be a better way to ensure information preservation when the literal encoding is not UTF-8 (see "future work").

`basic_format_arg` is not modified

An earlier version of this paper modified `basic_format_arg` so that the newly added `charN_t` would be stored in a handle. However, after talking to Mark de Wever and Jonathan Wakely, we ultimately concluded that:

- We definitively cannot add all 9 types (`charN_T`, `basic_string_view<charN_t>`, `charN_t*`) as this would be an ABI break in several implementations (implementations use a custom variant-like type with a discriminant stored in a few bits).
- We do not care about the cost of type erasing these formatters (one of the motivations to store them in `basic_format_arg`'s variant), as in general the cost of transcoding would dominate, and we expect them to be less use than, eg the `char` formatters.

- Modifying `basic_format_arg`'s variant would technically be a source break if a user provides an exhaustive (ie non-generic) visitor to `basic_format_arg::visit`. This is a lesser concern are few users are likely to be impacted.
- Modifying `basic_format_arg`'s variant would allow users to provide custom u8 strings values to for example a `fill` option. It is also unclear how useful this would be.

This leaves us with 2 options:

- Leave `basic_format_arg` alone (this paper).
- Add custom handling of `char8_t*` and `u8string_view` to `basic_format_arg`, accepting it would be a potential source break. This would not be an ABI break for implementation as they have some margin to extend the variant (but it would eat on future extensions). It is plausible that users would want to parametrize their own formatters with u8 strings arguments. However, there is certainly no motivation to support `char16_t`, `char32_t` there.

Implementability

There is sadly no great way to implement the transcoding part of this proposal with existing standard facilities. Here are a few implementation strategies and considerations:

- For an implementation that always uses UTF-8 as the type of character literals (libc++, clang), the only requirement is to weave support for the types, and add UTF-X->UTF-Y conversion routines, that are not hard to implement and already exist in implementations.
- For an implementation that supports additional character encodings, there needs to additionally exist a method that converts a UTF-8 sequence to the literal encoding. This can be achieved by using iconv or ICU, both of which can create a converter from a name (that one can get from `std::text_encoding`). On platforms where this would not be an option, or where no converter for the literal encoding would exist, assuming the execution encoding is a superset of the literal encoding is a reasonable assumption. For example, `c32rtomb` can be used.

The wording should leave enough room to allow an implementation that always produces a bunch of replacement characters.

Adding `charN_t` in the various interfaces of format did not present implementation challenges. Most implementations already support Unicode for escaping and width computation. So the hard work is already done.

Future work

This paper is minimalist by design, here are a few considerations for the future

`constexpr format`

It is reasonable to expect `format` to become `constexpr` soon-ish. To make `charN_t` formatters constexpr, implementation will need some built-in to perform the conversion. This is a reasonable ask: implementations already need to have conversion routines to evaluate string literal, exposing them to the library is doable.

We could imagine that built-in to be very similar to an iconv-style interface.

```
constexpr size_t
__builtin_utf8_to_ordinary(const char8_t*&, size_t & N, char*& Outbuf, size_t & OutputSize);
```

(And would use iconv or ICU or whatever transcoding facility the compiler already uses to encode string literals).

`charN_t` format strings

Ideally, we would support `charN_t` format strings (`std::format(u8"{}", "")`). However, the following questions would need to be answered:

- How does `to_chars` and `char8_t` interract? Unicode has a large set of numbers.
- Are existing locale facilities sufficient to support the needs of Unicode?
- What do we assume the encoding for `char` and `wchar_t` to be?
- What is the implementation burden?
- What is the interaction with user-defined formatters?

`wchar_t` <-> `char`

We do not propose to allow implicit transcoding between `wchar_t` and `char`. This is because we would need a better understanding of the nature of formatting arguments (execution or literal encoding?), and there seems to be less demand for it.

Error handling options

We could imagine letting the user control the replacement character to use or whether to throw a formatting error on non-representable/invalid characters. This can be explored separately as it would impact existing character types.

Better specification

As both C and C++ gain better transcoding facilities in future standards, we can respecify in terms of these facilities.

What about `iostream`?

This is a story for another paper (One that an enthusiastic reader is encouraged to write!)

Wording

� Types [basic.types]

� Fundamental types [basic.fundamental]

Type `bool` is a distinct type that has the same object representation, value representation, and alignment requirements as an implementation-defined unsigned integer type. The values of type `bool` are `true` and `false`. [*Note:* There are no `signed`, `unsigned`, `short`, or `long bool` types or values. — *end note*]

The types `char8_t`, `char16_t`, and `char32_t` are collectively called *Unicode character types*. The The Unicode character types, `char`, and `wchar_t`~~, `char8_t`, `char16_t`, and `char32_t`~~ are collectively called *character types*. The character types, `bool`, the signed and unsigned integer types, and cv-qualified versions[basic.type.qualifier] thereof, are collectively termed *integral types*. A synonym for integral type is *integer type*. [*Note:* Enumerations[dcl.enum] are not integral; however, unscoped enumerations can be promoted to integral types as specified in **??**. — *end note*]

� Formatting [format]

� Standard format specifiers [format.string.std]

The *align* option applies to all argument types. The meaning of the various alignment options is as specified in [format.align]. [*Note:* The *fill*, *align*, and `0` options have no effect when the minimum field width is not greater than the estimated field width because padding width is `0` in that case. Since fill characters are assumed to have a field width of `1`, use of a character with a different field width can produce misaligned output. The UNICODE CLOWN FACE character has a field width of `2`. The examples above that include that character illustrate the effect of the field width when that character is used as a fill character as opposed to when it is used as a formatting argument. — *end note*]

The *sign* option is only valid for arithmetic types other than the Unicode character types, `charT` and `bool` or when an integer presentation type is specified. The meaning of the various options is as specified in [format.sign].

The available string presentation types are specified in [format.type.string].

The *sign* option applies to floating-point infinity and NaN. [*Example:*

```
double inf = numeric_limits<double>::infinity();
double nan = numeric_limits<double>::quiet_NaN();
string s0 = format("{0:},{0:+},{0:-},{0: }", 1);        // value of s0 is "1,+1,1, 1"
string s1 = format("{0:},{0:+},{0:-},{0: }", -1);       // value of s1 is
"-1,-1,-1,-1"
string s2 = format("{0:},{0:+},{0:-},{0: }", inf);      // value of s2 is
"inf,+inf,inf, inf"
```

Table 1: Meaning of *align* options

Option	Meaning
<	Forces the formatted argument to be aligned to the start of the field by inserting n fill characters after the formatted argument where n is the padding width. This is the default for non-arithmetic non-pointer types, the Unicode character types, `charT`, and `bool`, unless an integer presentation type is specified.
>	Forces the formatted argument to be aligned to the end of the field by inserting n fill characters before the formatted argument where n is the padding width. This is the default for arithmetic types other than the Unicode character types, `charT` and `bool`, pointer types, or when an integer presentation type is specified.
^	Forces the formatted argument to be centered within the field by inserting $\lfloor \frac{n}{2} \rfloor$ fill characters before and $\lceil \frac{n}{2} \rceil$ fill characters after the formatted argument, where n is the padding width.

Table 2: Meaning of *type* options for strings

Type	Meaning
none, `s`	Copies the transcoded [format.string.transcoded] string to the output.
?	Copies the escaped string [format.string.escaped] to the output.

```
    string s3 = format("{0:},{0:+},{0:-},{0: }", nan);      // value of s3 is
"nan,+nan,nan, nan"
```

— *end example*]

The `#` option causes the *alternate form* to be used for the conversion. This option is valid for arithmetic types other than Unicode character types, `charT` and `bool` or when an integer presentation type is specified, and not otherwise. For integral types, the alternate form inserts the base prefix (if any) specified in [format.type.int] into the output after the sign character (possibly space) if there is one, or before the output of `to_chars` otherwise. For floating-point types, the alternate form causes the result of the conversion of finite values to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for `g` and `G` conversions, trailing zeros are not removed from the result.

The `0` option is valid for arithmetic types other than Unicode character types, `charT` and `bool`, pointer types, or when an integer presentation type is specified. For formatting arguments that have a value other than an infinity or a NaN, this option pads the formatted argument by inserting the `0` character n times following the sign or base prefix indicators (if any) where n is `0` if the *align* option is present and is the padding width otherwise. [*Example:*

```
char c = 120;
string s1 = format("{:+06d}", c);       // value of s1 is "+00120"
string s2 = format("{:#06x}", 0xa);     // value of s2 is "0x000a"
```

Table 3: Meaning of *type* options for integer types

Type	Meaning
`b`	`to_chars(first, last, value, 2)`; the base prefix is `0b`.
`B`	The same as `b`, except that the base prefix is `0B`.
`c`	Copies the character `static_cast<charT>(value)` to the output. Throws `format_error` if `value` is not in the range of representable values for `charT`.
`d`	`to_chars(first, last, value)`.
`o`	`to_chars(first, last, value, 8)`; the base prefix is `0` if `value` is nonzero and is empty otherwise.
`x`	`to_chars(first, last, value, 16)`; the base prefix is `0x`.
`X`	The same as `x`, except that it uses uppercase letters for digits above 9 and the base prefix is `0X`.
none	The same as `d`. [*Note:* If the formatting argument type is `charT`, a Unicode character type, or `bool`, the default is instead `c` or `s`, respectively. — *end note*]

Table 4: Meaning of *sign* options

Option	Meaning
`+`	Indicates that a sign should be used for both non-negative and negative numbers. The `+` sign is inserted before the output of `to_chars` for non-negative numbers other than negative zero. [*Note:* For negative numbers and negative zero the output of `to_chars` will already contain the sign so no additional transformation is performed. — *end note*]
`-`	Indicates that a sign should be used for negative numbers and negative zero only (this is the default behavior).
space	Indicates that a leading space should be used for non-negative numbers other than negative zero, and a minus sign for negative numbers and negative zero.

```
string s3 = format("{:<06}", -42);      // value of s3 is "-42   " (0 has no effect)
string s4 = format("{:06}", inf);       // value of s4 is "   inf" (0 has no effect)
```

— *end example*]

The *width* option specifies the minimum field width. If the *width* option is absent, the minimum field width is `0`.

If `{` *arg-id*$_{opt}$ `}` is used in a *width* or *precision* option, the value of the corresponding formatting argument is used as the value of the option. The option is valid only if the corresponding formatting argument is of standard signed or unsigned integer type. If its value is negative, an exception of type `format_error` is thrown.

If *positive-integer* is used in a *width* option, the value of the *positive-integer* is interpreted as a decimal integer and used as the value of the option.

For the purposes of width computation, ~~a~~ an ordinary or wide character string is assumed to be in a locale-independent, implementation-defined encoding. Implementations should use either UTF-8, UTF-16, or UTF-32, on platforms capable of displaying Unicode text in a terminal. [*Note:* This is the case for Windows-based and many POSIX-based operating systems. — *end note*]

[*Note:* `char8_t`, `char16_t`, `char32_t` strings are assumed to be in UTF-8, UTF-16, or UTF-32 respectively. — *end note*]

The available Unicode character type anf `charT` presentation types are specified in [*Editor's note:* the table below]

Table 5: Meaning of *type* options for `charT`

Type	Meaning
none, `c`	Copies the transcoded [format.string.transcoded] character to the output.
`b`, `B`, `d`, `o`, `x`, `X`	As specified in [format.type.int] with `value` converted to the unsigned version of the underlying type.
`?`	Copies the escaped character [format.string.escaped] to the output.

� Formatter specializations [format.formatter.spec]

The functions defined in **??** use specializations of the class template `formatter` to format individual arguments.

Let `charT` be either `char` or `wchar_t`. Each specialization of `formatter` is either enabled or disabled, as described below. A *debug-enabled* specialization of `formatter` additionally provides a public, constexpr, non-static member function `set_debug_format()` which modifies the state of the `formatter` to be as if the type of the *std-format-spec* parsed by the last call to `parse` were `?`. Each header that declares the template `formatter` provides the following enabled specializations:

- The debug-enabled specializations

```
template<> struct formatter<char, char>;
template<> struct formatter<char, wchar_t>;
template<> struct formatter<wchar_t, wchar_t>;

template<> struct formatter<char8_t, char>;
template<> struct formatter<char16_t, char>;
template<> struct formatter<char32_t, char>;
template<> struct formatter<char8_t,  wchar_t>;
template<> struct formatter<char16_t, wchar_t>;
template<> struct formatter<char32_t, wchar_t>;
```

- For each `charT`, for each Unicode character type UcharT, the debug-enabled string type specializations

```
template<> struct formatter<charT*, charT>;
```

```
template<> struct formatter<const charT*, charT>;
template<size_t N> struct formatter<charT[N], charT>;
template<class traits, class Allocator>
struct formatter<basic_string<charT, traits, Allocator>, charT>;
template<class traits>
struct formatter<basic_string_view<charT, traits>, charT>;

template<> struct formatter<UcharT*, charT>;
template<> struct formatter<const UcharT*, charT>;
template<size_t N> struct formatter<UcharT[N], charT>;
template<class traits, class Allocator>
struct formatter<basic_string<UcharT, traits, Allocator>, charT>;
template<class traits>
struct formatter<basic_string_view<UcharT, traits>, charT>;
```

- For each `charT`, for each cv-unqualified arithmetic type `ArithmeticT` other than `char`, `wchar_t`, `char8_t`, `char16_t`, or `char32_t`, a specialization

```
template<> struct formatter<ArithmeticT, charT>;
```

- For each `charT`, the pointer type specializations

```
template<> struct formatter<nullptr_t, charT>;
template<> struct formatter<void*, charT>;
template<> struct formatter<const void*, charT>;
```

The `parse` member functions of these formatters interpret the format specification as a *std-format-spec* as described in **??**. In addition, for each type `T` for which a `formatter` specialization is provided above, each of the headers provides the following specialization:

```
template<> inline constexpr bool enable_nonlocking_formatter_optimization<T> = true;
```

[*Note:* Specializations such as `formatter<wchar_t, char>` and `formatter<const char*, wchar_t>` that would require implicit multibyte / wide string or character conversion are disabled. — *end note*]

For any types `T` and `charT` for which neither the library nor the user provides an explicit or partial specialization of the class template `formatter`, `formatter<T, charT>` is disabled.

If the library provides an explicit or partial specialization of `formatter<T, charT>`, that specialization is enabled and meets the *Cpp17Formatter* requirements except as noted otherwise.

If `F` is a disabled specialization of `formatter`, these values are `false`:

- `is_default_constructible_v<F>`,
- `is_copy_constructible_v<F>`,
- `is_move_constructible_v<F>`,
- `is_copy_assignable_v<F>`, and

- is_move_assignable_v<F>.

An enabled specialization formatter<T, charT> meets the *Cpp17BasicFormatter* requirements [formatter.requirements]. [*Example:*

```
#include <format>
#include <string>

enum color { red, green, blue };
const char* color_names[] = { "red", "green", "blue" };

template<> struct std::formatter<color> : std::formatter<const char*> {
    auto format(color c, format_context& ctx) const {
        return formatter<const char*>::format(color_names[c], ctx);
    }
};

struct err {};

std::string s0 = std::format("{}", 42);         // OK, library-provided formatter
std::string s1 = std::format("{}", L"foo");     // error: disabled formatter
std::string s2 = std::format("{}", red);        // OK, user-provided formatter
std::string s3 = std::format("{}", err{});      // error: disabled formatter
```

— *end example*]

� Formatting transcoded characters and strings [format.string.transcoded]

The transcoded string *E* representation of a string *S* is constructed by encoding a sequence of characters as follows:

Let *ReplacementCharacter* be an implementation-defined code unit sequence in the the associated character encoding *TE* for charT ([lex.string.literal])

If *TE* is the same as the associated encoding *SE* of the character type *ST* of *S* [*Editor's note:* This needs to be defined] , each code unit of *S* is appended to E.

Otherwise (*ST* is a Unicode character type), for each code unit sequence *X* in *S* that either encodes a single character, or is a sequence of ill-formed code units, processing is in order as follows:

- If *X* encodes a single character *C*, then, if there exist an implementation-defined representation *R* of *C* in *TE*, each code unit of *R* is appended to *E*. If no such representation exists, *ReplacementCharacter* is appended to *E*.
- Otherwise *ReplacementCharacter* is appended to *E*.

The transcoded string representation of a character *C* is equivalent to the transcoded string representation of a string of *C*.

� Formatting escaped characters and strings [format.string.escaped]

A character or string can be formatted as *escaped* to make it more suitable for debugging or for logging.

The escaped string *E* representation of a string *S* is constructed by encoding a sequence of characters as follows.

~~The associated character encoding *CE* for `charT` ([lex.string.literal]) is used to both interpret *S* and construct *E*.~~

Let *TE* be the associated character encoding for `charT`. Let *SE* be the associated character encoding for the character type of *S*.

- U+0022 quotation mark (") is appended to *E*.
- For each code unit sequence *X* in *S* that either encodes a single character, is a shift sequence, or is a sequence of ill-formed code units, processing is in order as follows:
  - If *X* encodes a single character *C*, then:
    - If *C* is one of the characters in [format.escape.sequences], then the two characters shown as the corresponding escape sequence are appended to *E*.
    - Otherwise, if *C* is not U+0020 space and
      - ~~*CE*~~ *SE* is UTF-8, UTF-16, or UTF-32 and *C* corresponds to a Unicode scalar value whose Unicode property `General_Category` has a value in the groups `Separator` (`Z`) or `Other` (`C`), as described by UAX 44 of the Unicode Standard, or
      - ~~*CE*~~ *SE* is UTF-8, UTF-16, or UTF-32 and *C* corresponds to a Unicode scalar value with the Unicode property `Grapheme_Extend=Yes` as described by UAX 44 of the Unicode Standard and *C* is not immediately preceded in *S* by a character *P* appended to *E* without translation to an escape sequence, or
      - ~~*CE*~~ *SE* is neither UTF-8, UTF-16, nor UTF-32 and *C* is one of an implementation-defined set of separator or non-printable characters
      - *SE* and *TE* do not denote the same encoding and *C* has no implementation-defined representation in *TE*

      then the sequence `\u{`*`hex-digit-sequence`*`}` is appended to *E*, where *`hex-digit-sequence`* is the shortest hexadecimal representation of *C* using lower-case hexadecimal digits.

      Otherwise, if *SE* and *TE* do not denote the same encoding, the implementation-defined representation of *C* in *TE* is appended to *E*.
    - Otherwise, *C* is appended to *E*.
  - Otherwise, if *X* is a shift sequence, the effect on *E* and further decoding of *S* is unspecified.

Recommended practice: A shift sequence should be represented in *E* such that the original code unit sequence of *S* can be reconstructed.

- Otherwise (*X* is a sequence of ill-formed code units), each code unit *U* is appended to *E* in order as the sequence `\x{`*`hex-digit-sequence`*`}`, where *`hex-digit-sequence`* is the shortest hexadecimal representation of *U* using lower-case hexadecimal digits.

- Finally, u+0022 quotation mark (") is appended to *E*.

Table 6: Mapping of characters to escape sequences

Character	Escape sequence
u+0009 character tabulation	`\t`
u+000a line feed	`\n`
u+000d carriage return	`\r`
u+0022 quotation mark	`\"`
u+005c reverse solidus	`\\`

The escaped string representation of a character *C* is equivalent to the escaped string representation of a string of *C*, except that:

- the result starts and ends with u+0027 apostrophe (') instead of u+0022 quotation mark ("), and
- if *C* is u+0027 apostrophe, the two characters `\'` are appended to *E*, and
- if *C* is u+0022 quotation mark, then *C* is appended unchanged.

[*Example:* — *end example*]

[*Editor's note:* [...]]

� Class template `range_formatter` [format.range.formatter]

The class template `range_formatter` is a utility for implementing `formatter` specializations for range types.

`range_formatter` interprets *format-spec* as a *range-format-spec*. The syntax of format specifications is as follows:

range-format-spec:
    range-fill-and-align$_{opt}$ *width*$_{opt}$ `n`$_{opt}$ *range-type*$_{opt}$ *range-underlying-spec*$_{opt}$

range-fill-and-align:
    range-fill$_{opt}$ *align*

range-fill:
    any character other than `{` or `}` or `:`

range-type:
    `m`
    `s`
    `?s`

range-underlying-spec:
  : *format-spec*

For `range_formatter<T, charT>`, the *format-spec* in a *range-underlying-spec*, if any, is interpreted by `formatter<T, charT>`.

The *range-fill-and-align* is interpreted the same way as a *fill-and-align*[format.string.std]. The productions *align* and *width* are described in **??**.

The `n` option causes the range to be formatted without the opening and closing brackets. [*Note:* This is equivalent to invoking `set_brackets({}, {})`. — *end note*]

The *range-type* specifier changes the way a range is formatted, with certain options only valid with certain argument types. The meaning of the various type options is as specified in [formatter.range.type].

Table 7: Meaning of *range-type* options **[formatter.range.type]**

Option	Requirements	Meaning
`m`	`T` shall be either a specialization of `pair` or a specialization of `tuple` such that `tuple_size_v<T>` is `2`.	Indicates that the opening bracket should be `"{"`, the closing bracket should be `"}"`, the separator should be `", "`, and each range element should be formatted as if `m` were specified for its *tuple-type*. [*Note:* If the `n` option is provided in addition to the `m` option, both the opening and closing brackets are still empty. — *end note*]
`s`	`T` shall be a Unicode character type or `charT`.	Indicates that the range should be formatted as ~~a~~ a transcoded string [format.string.transcoded] `string`.
`?s`	`T` shall be a Unicode character type or `charT`.	Indicates that the range should be formatted as an escaped string [format.string.escaped].

If the *range-type* is `s` or `?s`, then there shall be no `n` option and no *range-underlying-spec*.

```
constexpr void set_separator(basic_string_view<charT> sep) noexcept;
```

Effects: Equivalent to: `separator_ = sep;`

```
constexpr void set_brackets(basic_string_view<charT> opening,
basic_string_view<charT> closing) noexcept;
```

Effects: Equivalent to:

```
opening-bracket_ = opening;
closing-bracket_ = closing;
```

```
template<class ParseContext>
constexpr typename ParseContext::iterator
```

```
parse(ParseContext& ctx);
```

Effects: Parses the format specifiers as a *range-format-spec* and stores the parsed specifiers in `*this`. Calls `underlying_.parse(ctx)` to parse *format-spec* in *range-format-spec* or, if the latter is not present, an empty *format-spec*. The values of `opening-bracket_`, `closing-bracket_`, and `separator_` are modified if and only if required by the *range-type* or the `n` option, if present. If:

- the *range-type* is neither `s` nor `?s`,
- `underlying_.set_debug_format()` is a valid expression, and
- there is no *range-underlying-spec*,

then calls `underlying_.set_debug_format()`.

Returns: An iterator past the end of the *range-format-spec*.

```
template<ranges::input_range R, class FormatContext>
requires formattable<ranges::range_reference_t<R>, charT> &&
same_as<remove_cvref_t<ranges::range_reference_t<R>>, T>
typename FormatContext::iterator
format(R&& r, FormatContext& ctx) const;
```

Effects: Writes the following into `ctx.out()`, adjusted according to the *range-format-spec*:

- If the *range-type* was `s`, then as if by formatting `basic_string<charT>(from_range, r)` as a transcoded string [format.string.transcoded].
- Otherwise, if the *range-type* was `?s`, then as if by formatting `basic_string<charT>(from_range, r)` as an escaped string [format.string.escaped].
- Otherwise,
  - `opening-bracket_`,
  - for each element `e` of the range `r`:
    - the result of writing `e` via `underlying_` and
    - `separator_`, unless `e` is the last element of `r`, and
  - `closing-bracket_`.

Returns: An iterator past the end of the output range.

� Specialization of *range-default-formatter* for strings [format.range.fmtstr]

```
namespace std {
template<range_format K, ranges::input_range R, class charT>
requires (K == range_format::string || K == range_format::debug_string)
struct range-default-formatter<K, R, charT> {
    private:
    using char_type = remove_cvref_t<range_reference_t<R>;// exposition only
    formatter<basic_string<~~charT~~ char_type>, charT> underlying_;
```

```
    public:
    template<class ParseContext>
    constexpr typename ParseContext::iterator
    parse(ParseContext& ctx);

    template<class FormatContext>
    typename FormatContext::iterator
    format(see below& str, FormatContext& ctx) const;
};
}
```

Mandates: ~~same_as<remove_cvref_t<range_reference_t<R>>, charT> is true~~ char_type denotes charT or a Unicode character type.

```
template<class ParseContext>
constexpr typename ParseContext::iterator
parse(ParseContext& ctx);
```

Effects: Equivalent to:

```
auto i = underlying_.parse(ctx);
if constexpr (K == range_format::debug_string) {
    underlying_.set_debug_format();
}
return i;
```

```
template<class FormatContext>
typename FormatContext::iterator
format(see below& r, FormatContext& ctx) const;
```

The type of r is const R& if ranges::input_range<const R> is true and R& otherwise.

Effects: Let *s* be a basic_string<~~charT~~ char_type> such that ranges::equal(*s*, r) is true. Equivalent to: return *underlying_*.format(*s*, ctx);

Feature test macros

[*Editor's note:* In <format>, bump __cpp_lib_format_uchar to the date of adoption] .

Acknowledgments

I'd like to thank Victor Zverovich for the incredible work on std::format, and SG-16 for motivating me to write this paper. Thanks to Mark de Wever and Jonathan Wakely to help me understand the use cases and extensibility limitations of basic_format_arg

References

[N4958] Thomas Köppe *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4958