Meeting C++ 2018

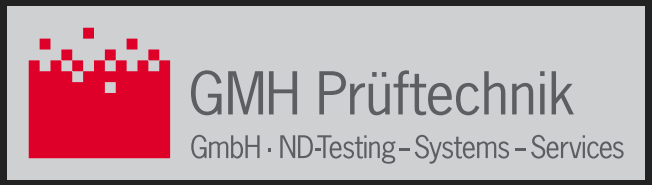
SOCIALIZING WITH {FMT}

Becoming familiar with the {fmt} library

Who am I ?

DANIELA ENGERT

- Electrical Engineer
- For 40 years creating computers and software
- For 30 years developing hardware and software in the field of applied digital signal processing
- For 20 years catching electromagnetic waves travelling around the world
- For 10 years listening to the echoes of ultrasound pings and creating vortices of eddy currents in steel



A small company located in Nürnberg, Germany

About 15 people, 9 of them working in the engineering departments (mechanical, electrical, software), creating custom-built machines for non-destructive testing of steel goods.

Our mission: detect and visualize flaws hidden within the body of any kind of steel pieces, or cracks at their surface (possibly buried below coatings) without doing any kind of harm to them.


GMH Prüftechnik
GmbH - NDT-Testing - Systems - Services

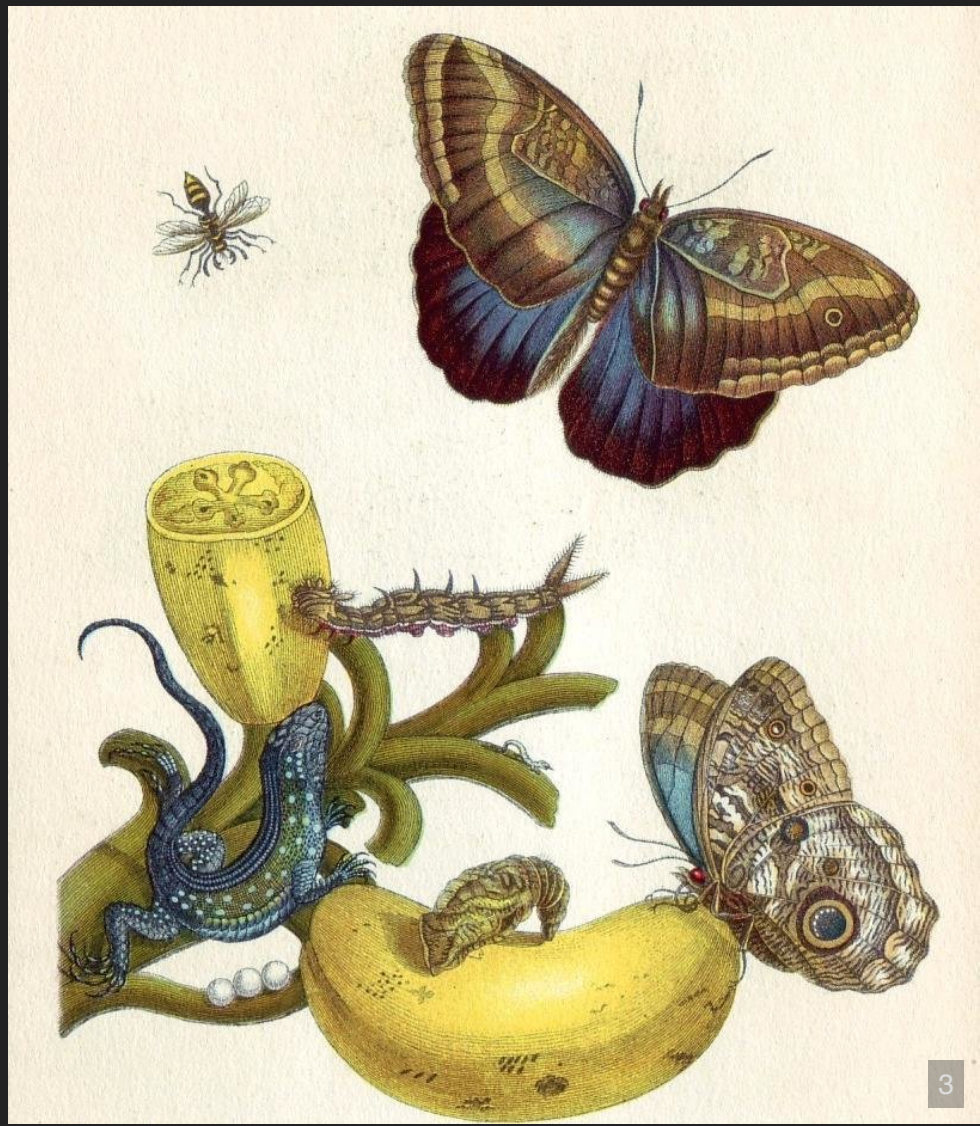

GMH Prüftechnik
GmbH · NDTesting · Systems · Services


GMH Prüftechnik


GMH Prüftechnik

Prelude

My journey
to {fmt}



THE STATUS QUO*

OF FORMATTING IN CURRENT C++

*not the band in search of the fourth chord

OPTION 1

sprintf from the *printf* function family in the C standard library

```
#include <stdio.h>

int n = sprintf(formatted_result, "value %d as hex is %02x", 42, 42);
```

- every parameter and even the return type is unsafe: unchecked or serve a dual purpose
- not extendable, works only with built-in types
- compact format string
- compilers learned to understand the formatting syntax and may prevent face-palm errors
- typically highly optimized and really fast, but performance is affected by locale support

OPTION 2

IO streams from the C++ standard library

```
#include <sstream>
#include <iomanip>

std::ostringstream stream;
stream << "value " << 42 << " as hex is " << hex << setw(2) << 42;
const auto formatted_result = stream.str();
```

- verbose, awkward syntax
- fragmented format string and formatting specifications
- OOP design, virtual functions, fewer optimization opportunities
- stateful, no local reasoning
- extendable
- type-safe

OPTION 3

Formatting libraries, for example Boost.Format

```
#include <boost/format.hpp>

auto formatted_result = str(boost::format("value %|1$| as hex is %|1$02x|") % 42);
```

- type-safe
- compact format string
- extendable
- many formatting options
- reuse of arguments
- slow at compile-time and run-time

OPTION 4

C++ frameworks, for example Qt

```
#include <QString>

auto formatted_string = QString("value %1 as hex is %2").arg(42).arg(42, 2, 16)
                                                        .toStdString();
```

- type-safe
- compact format string
- somewhat verbose
- not extendable
- based on non-standard string type, requires conversion

SO MANY OPTIONS

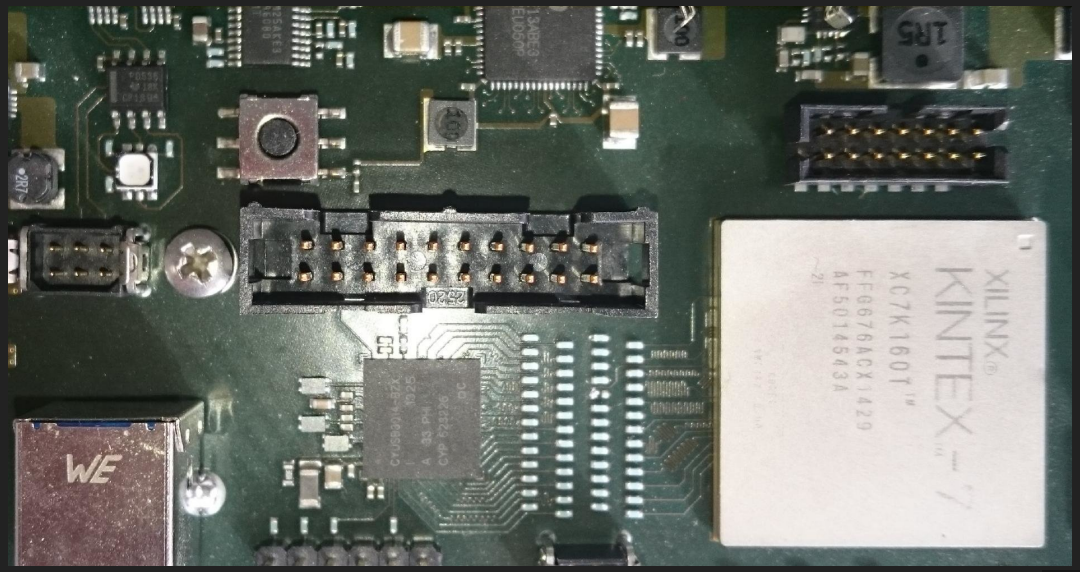
but

NONE OF THEM IS FULFILLING AT LEAST THE MOST BASIC EXPECTATIONS!

THE MOTIVATION

Some communication protocols of 3rd party equipment are text based and their throughput suffers from latencies

Some communication protocols deliver data packets so rapidly that error logging may become a serious bottleneck



THE OBJECTIVES

Never block a thread or an ASIO callback because of

- taking a mutex (possibly hidden within an innocent-looking call)
- hogging the CPU because of inadequate algorithms before issuing the next IO

Which boils down to

- Do not allocate in text formatting
- Format as fast as you can

LOOKING OUT FOR THE ONE LIBRARY ...

... THAT GIVES US TORN SOULS HOPE

... AND RULES THEM ALL

https://devrant.com/users/JeDaYoshi

Enter {fmt}

The new kid on the block

inspired by Python

HISTORY OF {FMT}

- started in 2012 by Victor Zverovich
- designed with performance, ease-of-use, and extendability in mind
- original name 'cppformat'
- later renamed to {fmt} alluding to it's format syntax
- up to version 4.x requiring only C++03
- beginning with version 5.x requiring a modest subset of C++11 which is available from all major compilers of the last 5 years
- BSD license
- in active development, currently 120 contributors

OVERVIEW

- {fmt} 101
- {fmt} under the hood
- {fmt} all the things!
- {fmt} all the strings!
- {fmt} it's about time!
- {fmt} in all languages!
- {fmt} the future

{fmt} 101

functions

formatting syntax

THE MOST BASIC FUNCTION

```
#include <fmt/format.h>

using namespace fmt;

template <typename String, typename... Args>
basic_string<Char> // 'Char' is derived from 'String'
format(const String & format_str, const Args &... args);
```

```
auto str = format("The answer is {}", 42);
```

produces result

```
std::string str("The answer is 42");
```

THE MOST BASIC FUNCTION

```
#include <fmt/format.h>

template <typename String, typename... Args>
basic_string<Char> format(const String & format_str,
                          const Args &... args);
```

The 'format_str' argument determines the internally used character type:

- the type of template parameter *String* is deduced from the argument as 'const char[17]'
- therefore **format** is formatting in terms of char code units
- and produces a 'std::basic_string<char>'

THE MOST BASIC FUNCTION

```
#include <fmt/format.h>

template <typename String, typename... Args>
basic_string<Char> format(const String & format_str,
                          const Args &... args);
```

any character type will do

- char
- wchar_t
- char8_t
- char16_t
- char32_t
- my_type

producing:

- std::string
- std::wstring
- std::u8string
- std::u16string
- std::u32string
- std::basic_string<my_type>

as long as there exists a matching specialization of std::char_traits.

THE MOST BASIC FUNCTION

```
#include <fmt/format.h>

template <typename String, typename... Args>
basic_string<Char> format(const String & format_str,
                          const Args &... args);
```

Numeric formatting uses the 'std::numpunct<Char>' facet of 'std::locale' for localized punctuation only when *explicitly* asked for locale-aware formatting by specifying the 'n' numeric formatting mode.

If you don't order this, it's not on your bill and you don't have to pay for it.

THE MOST BASIC FUNCTION

```
#include <fmt/format.h>

template <typename String, typename... Args>
basic_string<Char> format(const String & format_str,
                          const Args &... args);
```

The underlying character types of all strings or string-like *Args* amongst the formatting arguments 'args' **must** match the character type deduced from the format string 'format_str'.

Violating this requirement will cause a compilation failure.

The generated error message can possibly be improved...

FORMATTING INTO A CONTAINER

```
#include <fmt/format.h>

template <typename OutputIt, typename String, typename... Args>
OutputIt format_to(OutputIt it,
                   const Sring & format_str,
                   const Args &... args);
```

Takes any iterator type that models the *OutputIterator* concept and refers to a **contiguous** container, e.g.

- pointers to non-const characters
- non-const iterators returned by containers
- back_insert_iterators

The returned iterator refers to the next write position.

FORMATTING INTO A CONTAINER

```
#include <fmt/format.h>

vector<wchar_t> result;
format_to(back_inserter(result), L"{}", 42);
```

This produces a vector of size() == 2 with elements L'4' and L'2'

- you may want to reserve enough capacity beforehand to avoid repeated allocations
- you may find it advantageous to hang on to a container and reuse it as a dynamically-sized formatting cache

CONTROL THE MAXIMUM FORMATTED LENGTH

```
#include <fmt/format.h>

template <typename OutputIt, typename String, typename... Args>
auto format_to_n(OutputIt it, size_t size,
                 const String & format_str, const Args &... args);

template <typename OutputIt>
struct format_to_n_result {
  OutputIt out;
  size_t   size;
};
```

As before, the returned iterator 'out' refers to the next write position.

Please note: the 'size' member reports the **untruncated** size!

CONTROL THE MAXIMUM FORMATTED LENGTH

```
#include <fmt/format.h>

u16string result;
auto info = format_to_n(back_inserter(result), result.capacity(),
                        u"{}", 42);
```

This produces an u16string result(u"42") as long as the formatted result is no longer than the allocated capacity (e.g. 7 in this case on 64 bit, VS2017 with SSO).

CONTROL THE MAXIMUM FORMATTED LENGTH

```
#include <fmt/format.h>

u16string result;
auto info = format_to_n(back_inserter(result), result.capacity(),
                        u"{}", 42987654321);
```

The formatted result might be u"4298765" (again, 64 bit VS2017).

The 'size' member of the returned value 'info' contains the untruncated size 11 which denotes the actual capacity required to prevent truncation.

NO ALLOCATIONS

```
#include <fmt/format.h>

template <typename String, typename... Args,
          size_t SIZE, typename Char>
OutputIterator format_to(basic_memory_buffer<Char, SIZE> & buffer,
                         const String & format_str,
                         const Args &... args);

template <typename Char, size_t SIZE = inline_buffer_size,
          typename Allocator = allocator<T>>
class basic_memory_buffer;
```

'fmt::basic_memory_buffer' is a simple container with 'SIZE' Chars worth of inline space (i.e. on the stack) and dynamic overflow capacity allocated from the heap. **format_to** returns an output iterator into the given buffer.

NO ALLOCATIONS

```
#include <fmt/format.h>

basic_memory_buffer<char, 100> buffer;
format_to(buffer, "{}", 42);
string result = to_string(buffer);
```

- you have full control over the amount of inline capacity
- you may chain multiple formatting calls into the same buffer
- you may use such buffers as arguments to other calls to **format** or variants thereof

and all of this without any heap allocations!

USER-DEFINED LITERALS

```
#include <fmt/format.h>

fmt::literals::operator""_format(const char *s, size_t);
fmt::literals::operator""_format(const wchar_t *s, size_t);
```

These UDLs allow you to write

```
using namespace fmt::literals;

wstring message = L"The answer is {}"_format(42);
```

instead of

```
wstring message = format(L"The answer is {}", 42);
```

USER-DEFINED LITERALS

```
#include <fmt/format.h>

fmt::literals::operator""_a(const char *s, size_t);
fmt::literals::operator""_a(const wchar_t *s, size_t);
```

These UDLs attach a name to an argument.

```
using namespace fmt::literals;

format("The answer is {the_answer}", "the_answer"_a = 42);
```

is probably less clumsy than

```
format(L"The answer is {the_answer}", arg("the_answer", 42));
```

USER-DEFINED LITERALS

- Named arguments help preventing developers from losing track of which argument goes into which replacement field.
- Named arguments may improve the quality of string translation because the translator sees the whole context and intent.

More on named arguments in just a few slides...

COMPILE-TIME FORMAT-STRINGS

```
#define FMT_STRING_ALIAS 1
#include <fmt/format.h>

string result = format(fmt("The answer is {:d}"), "42");
```

A compile-time format string allows checking both the syntax of the format string and the correct types of the arguments *at compile time*.

This applies to user-defined formatters too, as long as they are properly 'constexpr'-qualified.

The example above will not compile because the replacement field asks for a numerical argument, but there's a string supplied instead.

COMPILE-TIME FORMAT-STRINGS

```
#define FMT_STRING_ALIAS 1
#include <fmt/format.h>

string result = format(fmt("The answer is {:d}"), "42");
```

Defining FMT_STRING_ALIAS to 1 before the inclusion of 'format.h' makes the macro fmt available to easily define compile-time format strings the same way on all compilers. This feature requires C++14 relaxed constexpr support from the compiler!

Obviously, compile-time checking in this pre-packaged form precludes runtime translation of the format string.

COMPILE-TIME FORMAT-STRINGS

```
#include <fmt/format.h>

#define translate(...) // to some equivalent of fmt(...)
string result = format(translate("The answer is {:d}"), "42");
```

Some macro magic and proper tooling may overcome this drawback:

- the compiler sees a compile-time string and checks for syntax errors and consistency in some kind of 'pre-check mode'
- in 'regular mode' it sees the lookup into the translation database
- the translation tools build see translation markers

There's probably no one-size-fits-all solution readily at hand.

OTHER TARGETS

printing to the console

```
#include <fmt/format.h>

template <typename String, typename... Args>
void print(const String & format_str, const Args &... args);
```

printing to a file stream

```
#include <fmt/format.h>

template <typename String, typename... Args>
void print(FILE * f, const String & format_str,
           const Args &... args);
```

A DIFFERENT WORLD

Including 'fmt/printf.h' makes POSIX compatible formatting available.

```
#include <fmt/printf.h>

template <typename String, typename... Args>
auto sprintf(const String & format_str, const Args &... args);

template <typename String, typename... Args>
int printf(const String & format_str, const Args &... args);

template <typename String, typename... Args>
int fprintf(FILE * file, const String & format_str,
            const Args &... args);

template <typename String, typename... Args>
int fprintf(basic_ostream<char-type-of-String> & stream,
            const String & format_str, const Args &... args);
```

A DIFFERENT WORLD

```
#include <fmt/printf.h>

auto sprintf(...);
int printf(...);
int fprintf(FILE * file, ...);
int fprintf(basic_ostream<char-type-of-String> & stream, ...);
```

are still type-safe and equivalent to

```
#include <fmt/format.h>

auto format(...);
int print(...);
int print(FILE * file, ...);
basic_ostream<char-type-of-String> << format(...);
```

but with extended POSIX format specification syntax instead of the native {fmt} syntax

FORMATTING SYNTAX

A format string is an alternating sequence of

- literal text fields
- replacement fields surrounded by a pair of curly braces {}

Every field may be empty!

valid

```
" bla {} bla {r1}{r2} {{{r3}}}"
```

invalid

```
" bla {} bla {r1}{r2} {r3}}"
```

FORMATTING SYNTAX

Literal text fields are copied verbatim to the output.

Replacement fields consist of two parts which are separated by a colon ' : ' character

- an argument identifier
- a formatting specification

Each part may be empty. The colon is required only when the formatting specification is not empty.

```
{argument identifier:formatting specification}
```

FORMATTING SYNTAX

valid:

```
{}          default-formatted next argument
{:}        as above, colon is optional here
{0}       first argument, default-formatted
{name}   argument "name", default-formatted
{:<5X}  next argument formatted as specified
{2:#}  third argument formatted as specified
```

invalid:

```
{#:}            invalid argument identifier
{<5x}                 the colon is missing
```

ARGUMENT IDENTIFIERS

Formatting arguments are associated with a replacement field either by

- position, i.e "indexed" argument
- name, i.e. "named" argument

Therefore, each argument may be referenced as many times as you like.

Indexed argument references are a little bit more efficient compared to named argument references because the latter require building an internal dictionary and argument name lookup. In the current implementation, the dictionary is allocated on the default heap.

ARGUMENT IDENTIFIERS

Argument identifiers follow the standard naming C++ rules as you would expect

- numeric positional indices start with number 0, referring to the first argument
- valid names used in named argument identifiers are also valid C++ identifiers (and vice versa) with the one exception that only characters from the 7-bit ASCII character set are allowed (C++ allows almost all Unicode characters in identifiers)

ARGUMENT IDENTIFIERS

The index used for indexed argument references is by default maintained internally by a counter (automatic indexing). Which means the first replacement field references the first argument, the second field the second argument, an so on.

Automatic indexing is valid only if **all** replacement fields use automatically generated references. Mixing modes is an error.

```
// valid

result = format("{}", "a"_a=42);        // automatic indexing
result = format("{0}{a}", "a"_a=42);    // positional indexing

//invalid

result = format("{}{0}{a}", "a"_a=42); // mixed indexing
```

FORMATTING SPECIFICATION

The formatting specification language used by default for all of the built-in types, pointers, and strings is pretty much what you would expect and what you are already familiar with from 'printf' formatting:

```
formatspec ::= [[fill]align][sign]["#"]["0"][width]["." precision][type]
fill       ::= a single character other than '{', '}' or '\0'
align      ::= "<" | ">" | "=" | "^"
sign       ::= "+" | "-" | " "
width      ::= integer | "{" argument identifier "}"
precision  ::= integer | "{" argument identifier "}"
type       ::= inttype | "a" | "A" | "c" | "e" | "E" | "f" | "F"
                       | "g" | "G" | "p" | "s"
inttype    ::= "b" | "B" | "d" | "n" | "o" | "x" | "X"
```

FORMATTING SPECIFICATION

Please note the emphasis on 'by default' and on the list of type to which this formatting language applies!

As a rule of thumb you may assume that every aspect of formatting can be customized in some way:

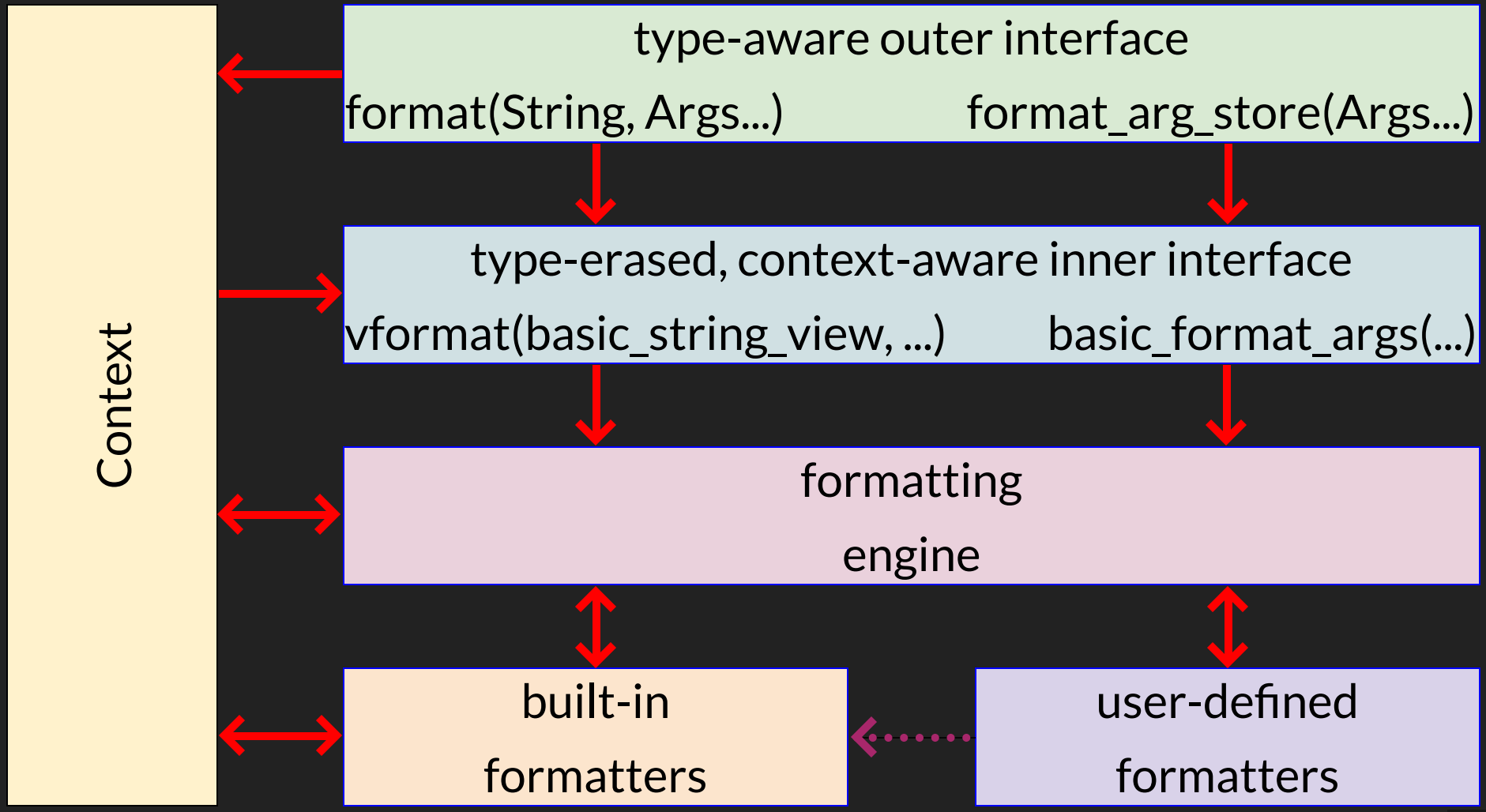
- the interpretation of the parsed formatting specification elements for use with the built-in types may be customized and put into a new ruleset
- a completely different formatting language may be applied. An example is using the extended POSIX language by including **printf.h** instead of **format.h**
- other types beyond the built-in ones may have custom formatters using a custom formatting language

{fmt} under the hood

the engine

replaceable parts

THE MACHINE ROOM



THE FORMATTING CONTEXT

- is created in the interface layer
- holds the character type in terms of which the formatting engine operates
- holds a range referencing the format string
- holds the argument list with references to the formatting arguments
- holds an iterator referencing the next write position in the output sink
- holds the error-handler
- may hold supplemental data supplied at the interface (custom contexts)
- glues formatters to the formatting engine (custom contexts)

TYPE ERASURE

The standard interface is the rich set of function-overloads which is 100% aware of all types, both of the format string and of all of the formatting arguments.

References to the formatting arguments are held in a type-aware 'format_arg_store'.

This layer is meant to be as thin as possible, effectively only stripping the type information from the input types and categorizing them into a small set of well-known inner argument types.

FORMAT

VFORMAT

Here, format is only a representative of all of the type-aware interface functions shown earlier.

After erasing the types of the formatting arguments, the much smaller set of v* functions (e.g. vformat) does the actual work.

Most importantly, the v* functions require much fewer template instantiations (thanks to type erasure), and have a fixed number of arguments rather than a variable number.

Using these v* functions in application code may be a better option in certain use-cases (e.g. logging).

STRING
BASIC STRING VIEW

Eventually, the same type erasure happens to the String template parameter type of the format string.

Like all other strings amongst the formatting arguments, the formatting string is cooked down into the internal *lingua franca* 'fmt::basic_string_view<Char>'.

Being trivially copyable with a small footprint makes string views ideal vehicles to deal with any kind of string.

FORMAT ARGUMENT STORE

BASIC FORMAT ARGS

The fully type-aware 'format_arg_store<Args ...>' is a container holding references to the set of given formatting arguments.

Another type-erased container of same size called 'basic_format_args' hold a list of 'format_args' (effectively a variant type), that can be constructed from a 'format_arg_store'. Each element in the container has the size of two pointers, which is large enough to keep the essential information of every original formatting argument.

When required, the original argument with its full type can be resurrected, e.g. as arguments to custom-formatters.

{fmt} all the things!

extending the library

BEFORE WE START ...

There is another header in fmt called <fmt/color.h>.

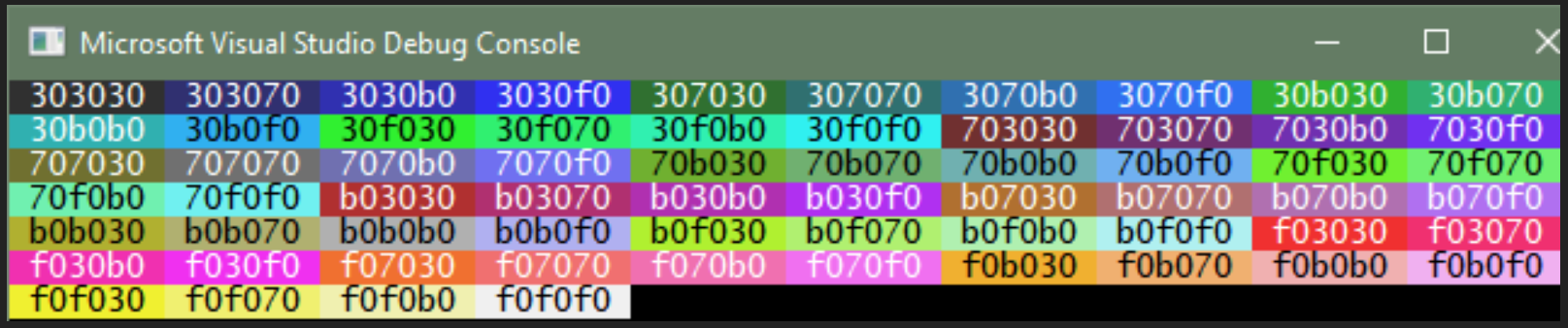
It provides two more overloads of print to print ***colored text*** to the console.

```
#include <fmt/color.h>

struct rgb {
  uint8_t r, g, b;
};

template <typename String, typename... Args>
void print(rgb foreground, const String & format,
           const Args &... args);

template <typename String, typename... Args>
void print(rgb foreground, rgb, background, const String & format,
           const Args &... args);
```

BEFORE WE START ...

Let's explore the options that we have to print the background color components colored to the console. Our baseline is this simple code that comes up with some colors:

```
void baseline() {
  forSomeColors([](rgb foreground, rgb background) {
    print(foreground, background, " {:02x}{:02x}{:02x} ",
          background.r, background.g, background.b);
  });
}
```



BEFORE WE START ...

Manually decomposing class members and feeding the individual elements to a formatting or printing facility is the traditional way to deal with C++ aggregates.

Alas, it's both tedious and error-prone.

Let's teach {fmt} new tricks how to handle types other than the built-in ones, in this case 'struct rgb'.

FORMATTERS

This is the base template of (almost) all formatters. Some built-in formatters and all user-defined formatters are (partial) specializations of it.

```
// A formatter for objects of type T.
template <typename T,              // the type to be formatted
          typename Char = char,    // the formatters's Char type
          typename Enable = void> // enable SFINAE
struct formatter {
  template <typename ParseContext>
  typename ParseContext::iterator parse(ParseContext &);

  template <typename FormatContext>
  auto format(const T & val,
              FormatContext & ctx) -> decltype(ctx.out());
};
```

FORMATTERS

You may specialize your custom formatter for only one character type

```
// A formatter for objects of type T1.
template <>
struct formatter<T1, wchar_t> { ... };
```

or SFINAE the specialization on predicates of your choice

```
// A formatter for objects of type T2.
template <typename T2, typename Char>
struct formatter<T2, Char, enable_if_t<
                 sizeof(Char) == 2 && is_base_of_v<BaseType, T2>>
{
  ...
};
```

FORMATTERS

The parse function is responsible for parsing the formatting specifier part of a replacement field

```
template <typename T>
struct formatter {
  template <typename ParseContext>
  typename ParseContext::iterator parse(ParseContext &);
};
```

The given parse context models a *RandomAccessRange*, either beginning right after the colon (if there is one) or the opening curly brace, and extends to the end of the format string.

The parse function owns the front part of the range up to, but not including the next matching closing curly brace. The return value is an iterator that **must** reference this curly brace!

FORMATTERS

The format function generates a character stream from the given value into the output sink of the format context.

```
template <typename T>
struct formatter {
  template <typename FormatContext>
  auto format(const T & val, FormatContext & ctx)
    -> decltype(ctx.out());
};
```

The iterator returned by the out() function of the given context models an *Outputiterator* into a *BackInsertRange*. The return value is an iterator that refers to the next write position in the output sink.

A SIMPLE FORMATTER

Goal: achieve the same formatted output as the one generated from the baseline code

- all 3 components
- side-by-side
- using the same formatting specification

As an additional requirement, as much existing code as possible shall be reused to avoid duplication and possible errors.

A SIMPLE FORMATTER

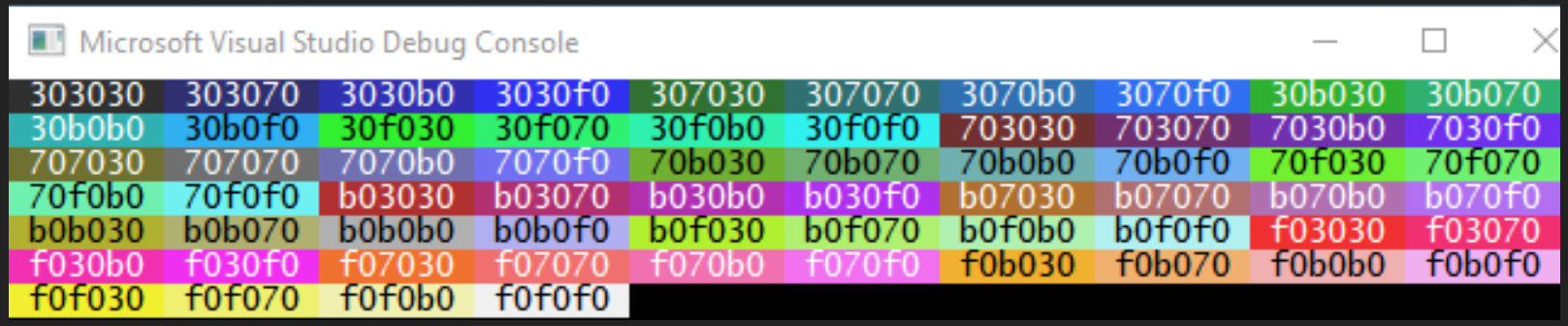
```
template <typename Char>
struct formatter<rgb, Char> : formatter<uint8_t, Char> {
  using base = formatter<uint8_t, Char>;

  template <typename FormatContext>
  auto format(const rgb_c & color, FormatContext &ctx) {
    base::format(color.r, ctx);
    base::format(color.g, ctx);
    return base::format(color.b, ctx);
  }
};
```

- No parse function required, it's inherited from the base class.
- The format function is trivial, it delegates the actual formatting of each component to the base class.

A SIMPLE FORMATTER

The result is identical to the one from the baseline, just as expected.

```
void simple() {
  forSomeColors([](rgb foreground, rgb background) {
    print(foreground, background, " {:02x} ", background);
  });
}
```



A BIT MORE FANCY

So far, each component is rendered side-by-side to its siblings. Let's change the requirements a bit:

- the component shall be separated by a comma
- the triplet shall be surrounded by a pair of braces

This doesn't affect parsing, but the formatting code needs some additions.

A BIT MORE FANCY

```
template <typename Char>
struct formatter<rgb, Char> : formatter<uint8_t, Char> {
  using base = formatter<uint8_t, Char>;

  template <typename FormatContext>
  auto format(const rgb_c & color, FormatContext &ctx) {
    auto it = ctx.begin();
    it = '(';
    it = base::format(color.r, ctx);
    it = ',';
    it = base::format(color.g, ctx);
    it = ',';
    it = base::format(color.b, ctx);
    it = ')';
    return it;
  }
};
```

The back insert iterator 'it' must be obtained from the context and the return values of the base formatters.

A BIT MORE FANCY

At the call site, nothing needs to be changed. For the sake of variation we use a different formatting specification.

```
void hardcoded() {
  forSomeColors([](rgb foreground, rgb background) {
    print(foreground, background, " {:3}  ", background);
  });
}
```

EVEN MORE FANCY

Let's change the requirements once again:

- the component shall be optionally separated by a user-defined character
- only in case of an alternate decimal formatting mode, the triplet shall be surrounded by a pair of braces
- only in case of an alternate hexadecimal formatting mode, the triplet shall be prepended by a # character

This doesn't affect the formatting code much over the previous one, but the format specification language must be extended. Therefore a default parser won't do any longer.

EVEN MORE FANCY

Remember:

- we have control only over the formatting specification part of the replacement field.
- if existing parsing code shall be reused, any extended format specification language must fit unambiguously into the standard one.

There are two options:

- prepend the standard formatting part with the optional extension part.
- or append the optional extension.

In both cases, the first part must be delimited by a pair of curly braces to fit the restrictions.

EVEN MORE FANCY

The **optional** extension part shall be the one in braces, so extended the language this way:

```
extendedspec ::= ["{" alternate [delimiter] "}"][formatspec]
delimiter    ::= a single character other than '{', '}' or '\0'
alternate    ::= ":" | "#"
```

Unfortunately, the current implemention of the library offers no accessors to the format specification parsed by the inherited formatter. Therefore custom formatters need to look into the formatspec part by themselves if there is some vital information in there.

EVEN MORE FANCY

The framework:

```
template <typename Char>
struct formatter<rgb, Char> : formatter<uint8_t, Char> {
  using base = formatter<uint8_t, Char>;

  Char delimiter = {};
  bool isHex = false;
  bool isAlt = false;

  static constexpr bool isArgId(Char ch) {
    return ((ch >= '0') & (ch <= '9')) | ((ch >= 'a') & (ch <= 'z')) |
           ((ch >= 'A') & (ch <= 'Z')) | (ch == '_');
  };

  template <typename ParseContext>
  auto parse(ParseContext & ctx);

  template <typename FormatContext>
  auto format(const rgb & color, FormatContext & ctx);
};
```

EVEN MORE FANCY

The parser, pull out the optional extended part:

```
auto parse(ParseContext & ctx) {
  bool inBraces = false;
  unsigned pos = 0;
  unsigned prefix = 0;
  Char previous = {};

  for (auto ch : ctx) {
    if (inBraces) {
      if (ch == '}')
        inBraces = false;
      if (pos < 4) {
        if (ch == '}') {
          if ((!isAlt & !delimiter & (pos == 1)) | (isAlt & (pos == 2))
              | (delimiter & (pos == 3)))
            prefix = pos + 1;
        } else if ((pos == 1) & (ch == '#')) {
          isAlt = true;
        } else if (pos == 2) {
          if (isAlt | (previous == ':')) {
            delimiter = ch;
```

EVEN MORE FANCY

Check for the standard format specification:

```
auto parse(ParseContext & ctx) {
  ...
  for (auto ch : ctx) {
    ...
          } else if (!isArgId(previous)) {
            ctx.on_error(
              "invalid prefix: first character must be ':' or '#'");
          }
        } else if (pos == 3) {
          ctx.on_error("invalid prefix: missing closing '}'");
        }
      }
    } else {
      if (ch == '{') {
        inBraces = true;
      } else if (ch == '}') {
        isHex = (previous == 'x') | (previous == 'X');
        break;
      }
    }
```

EVEN MORE FANCY

Trace the amount of parsed characters so far, chop off the consumed prefix, and feed the rest into the parser of the inherited formatter class.

```
auto parse(ParseContext & ctx) {
  ...
  for (auto ch : ctx) {
    ...
    ++pos;
    previous = ch;
  }

  ctx.advance_to(ctx.begin() + prefix);
  return base::parse(ctx);
}
```

EVEN MORE FANCY

The format function is no longer using hardcoded values but rather the parsed properties instead.

```
auto format(const rgb & color, FormatContext &ctx) {
  auto it = ctx.begin();
  if (isAlt) it = isHex ? Char{ '#' } : Char{ '(' };
  it = base::format(color.r, ctx);
  if (delimiter) it = delimiter;
  it = base::format(color.g, ctx);
  if (delimiter) it = delimiter;
  it = base::format(color.b, ctx);
  if (isAlt && !isHex) it = Char{ ')' };
  return it;
}
```

EVEN MORE FANCY

Alternative mode hexadecimal formatting without delimiters will look like this:

```
void fancy() {
  forSomeColors([](rgb Foreground, rgb Background) {
    print(Foreground, Background, " {:{#}02X}  ", Background);
  });
}
```

EVEN MORE FANCY

And dash-delimited decimal formatting looks like this:

```
void fancy() {
  forSomeColors([](rgb Foreground, rgb Background) {
    print(Foreground, Background, "   {:{:-}03}  ", Background);
  });
}
```

OUTPUT STREAMS

If a class has an associated output-stream 'operator <<', then {fmt} can use this, too, by generating a matching formatter.

```
#include <fmt/ostream.h>

template <typename Char>
basic_ostream<Char> & operator<<(basic_ostream<Char> & os,
                                 const rgb &) {
  return os << setw(3) << color.r << ','
            << setw(3) << color.g << ','
            << setw(3) << color.b;
}

void custom_ostream() {
  forSomeColors([](rgb Foreground, rgb Background) {
    print(Foreground, Background, " {} ", Background);
  });
}
```

OUTPUT STREAMS

The generated formatter creates a temporary output stream into an internal buffer, calls the output-stream 'operator <<', and pushes the characters collected from the stream into the format sink.

```
template <typename Char, typename T>
void format(basic_buffer<Char> & buffer, const T & value) {
  internal::formatbuf<Char> format_buf(buffer);
  basic_ostream<Char> output(&format_buf);
  output.exceptions(failbit | badbit);
  output << value;
}
```

OUTPUT STREAMS

The drawback is the lack of control over the formatting process (everything is hardcoded).

Caution: **every** type with an output-stream 'operator <<' gets an automatically generated formatter slapped on as soon as 'ostream.h' is included - even if there is an explicitly user-defined one. This will cause compile failures because of the ambiguity of two equally viable specializations: the generated one and the user-defined one!

This makes incrementally moving from ostream-based formatting to {fmt} difficult.

{fmt} all the strings!

beyond STL

STRINGS

{fmt} unifies all string types, from the C heritage up to the C++ string (and string_view if available) classes, into a single view type using a built-in, very simple 'fmt::basic_string_view'. As it is independent from the STL, it can be used uniformly on each compiler in every compilation mode. It has constexpr implicit converting constructors from

- const Char *
- basic_string<Char>
- basic_string_view<Char>

STRINGS

Because of those constexpr implicit converting constructors from all C++ string types, 'fmt::basic_string_view<Char>' looks like the prefect choice as receive type for format strings!

Except when it is not.

There are so many more string types or string-like types out there:

- in the STL, e.g. array<Char> or vector<Char>
- outside of the STL, e.g. QString, CString, XStr, BStr, ...

In fact, every contiguous sequence of Chars can be treated as a string.

STRINGS

Adding implict conversion constructors to fmt's string_view for all string-like types out there is obviously a non-starter.

Adding an implicit conversion operator to fmt's string_view to those string-like types is practically impossible.

And it is **dangerous**, too, because of unwittingly taking references to temporaries that are gone at the time of dereferencing!

Manually converting to a standard C++ string or string_view or fmt's string view is possible but tedious and may introduce inefficiencies that are overlooked by untrained developers.

There must be a better way ...

AN IDEA

If all string types from the C++ standard are converted to the internal *lingua franca* **fmt::basic_string_view<Char>** anyway, why not using a free function as an explicit conversion operator, make it a customization point, and express all conversions to the internal string_view by this conversion operator?

Even better, such a conversion operator already existed as an internal utility, it wasn't just used universally and it wasn't publicly visible.

THE PLAN

- **replace** the exposed 'fmt::basic_string_view' in the format string parameter slot by a **String template** parameter which accepts only strings and string-like types (metaprogramming galore)
- express **all** conversions to 'fmt::basic_string_view' **only** in terms of the conversion operator
- put the conversion operator into the **public** interface of {fmt}
- make it a customization point and **detect** by ADL if there exists this conversion operator (or not) for **any** given type (even more metaprogramming)

CONVERSION TO STRINGVIEW

```
namespace ns_of_string_type {
  constexpr auto to_string_view(const string_type & string) noexcept;
}
```

- must be declared in exactly the same namespace where 'string_type' is declared.
- must return an instance of 'fmt::basic_string_view<Char>'.
- Char must match the character type of 'string'.
- is typically parameterized on the character type.
- the Char **reference** in the returned view must be **valid** for the whole duration of use.

```
namespace ns_of_string_type {
  template <typename Char>
  constexpr fmt::basic_string_view<Char>
    to_string_view(const string_type<Char> & string) noexcept;
}
```

CONVERSION TO STRINGVIEW

In fact, the whole concept of a *String* in {fmt} is now built upon the convertability of a given type to a 'fmt::basic_string_view<Char>'.

If a type is **not convertible**, then it's **not a String**

CONVERSION TO STRING VIEW

MFC/ATL strings

```
#include <atlstr.h>

namespace ATL {

template <typename Char>
constexpr fmt::basic_string_view<Char> to_string_view(
  const CStringT<Char,
                 StrTraitATL<Char,
                 ChTraitsCRT<Char>>> & String) noexcept {
  return { reinterpret_cast<const Char *>(String),
           static_cast<size_t>(String.GetLength()) };
}

} // namespace ATL
```

CONVERSION TO STRING_VIEW

QStrings

```
#include <QString>

QT_BEGIN_NAMESPACE

constexpr fmt::basic_string_view<char16_t>
to_string_view(const QString & String) noexcept {
  return { reinterpret_cast<const char16_t *>(String.utf16()),
           static_cast<size_t>(String.length()) };
}

QT_END_NAMESPACE
```

At least on Windows, replacing char16_t by wchar_t is possible, too.

THE RESULT

```
#include <QString>
#include <atlstr.h>
#include <string>

const auto qstr = QString("Hello {} - {} ");
const auto wstr = wstring(L"Meeting C++ 🤘");
const auto cstr = CString("Grüße aus Nürnberg");
result = format(qstr, wstr, cstr);
```

format returns a 'std::wstring' with contents

"Hello Meeting C++ 🤘 - Grüße aus Nürnberg"

(in this case, the conversion from QString is defined to use character type wchar_t)

{fmt} in all languages

localization

INTERNATIONALIZATION

Generating formatted output for different cultures in various languages is affected by three aspects:

- text translation
- number rendering
- date/time rendering

or in C++ speak:

- the std::messages facet
- the std::numpunct facet
- /* crickets */

of a locale.

{fmt} is aware of the numpunct facet of the global locale.

TEXT TRANSLATION

needs to be handled by machinery completely outside of {fmt}. There are many options to get at translated text according to the currently selected global locale. Popular ones are

- GNU gettext
- Boost.Locale, a C++ implementation of 'gettext' (plus more)
- Qt TS
- ...

TEXT TRANSLATION

There is little to say about GNU gettext because it is based on C strings, and {fmt} knows how to deal with C strings.

The Qt translation system is fully based on Qt's QStrings, and those can be easily made interoperable with {fmt} by means of a 'to_string_view' conversion operator.

This is probably true for many other translation systems as well.

Boost.Locale is different, though.

BOOST.LOCALE

```
// gnu gettext
//   'translate' is a macro
//   returns a C string

#include <libintl.h>
auto translated = translate("text");

// Boost.Locale
//   'translate' is a function
//   returns a proxy object of type
//
//   template <typename Char>
//   boost::locale::basic_message<Char>

#include <boost/locale.hpp>
using namespace boost::locale;

auto proxy = translate("text");

string explicitly_translated = proxy.str(); // fine, but tedious
string implicitly_translated = proxy;       // nice, but dangerous!
```

BOOST.LOCALE

A conversion operator for Boost.Locale's 'basic_message':

```
#include <boost/locale.hpp>

namespace boost::locale {

template <typename Char>
inline fmt::basic_string_view<Char>
to_string_view(const basic_message<Char> & message) noexcept {
  const auto translated = message; // does the actual translation
  return { translated.data(), translated.size() };
}

} // namespace boost::locale
```

Easy, mission accomplished! {fmt} accepts 'basic_message'.

Well, say *hello* to UB ....

BOOST.LOCALE

There is no way to define a conversion operator without UB.
But a user-defined formatter will do!

```
namespace fmt {

template <typename Char>
struct formatter<boost::locale::basic_message<Char>, Char> :
       formatter<basic_string_view<Char>, Char> {
  using base = formatter<basic_string_view<Char>, Char>;

  template <typename FormatContext>
  auto format(const boost::locale::basic_message<Char> & message,
              FormatContext & ctx) {
    basic_string<Char> translated = message;
    return base::format({translated.data(), translated.size()},
                        ctx);
  }
};

} // namespace fmt
```

TEXT TRANSLATION

Putting all pieces together ...

With the conversion operator for QString, the user-defined formatter for 'boost::locale::basic_message' as shown before, and a German locale in place:

```
print(translate(L"{} - Greetings from Nuremberg").str(),
      QObject::tr("© 2018 All rights reserved"));

print(QObject::tr("{} - Greetings from Nuremberg"),
      translate(L"© 2018 All rights reserved"));
```

Prints in both cases

"© 2018 Alle Rechte vorbehalten - Grüße aus Nürnberg"

BOOST.LOCALE (REPRISE)

A conversion operator for Boost.Locale's 'basic_message':

```
// requires an augmented version of Boost.Locale 1.68 or later
// that can return a std::string_view<Char> with *stable*
// references as an alternative to returning an ephemeral
// std::basic_string<Char> object

#include <boost/locale.hpp>

namespace boost::locale {

template <typename Char>
inline fmt::basic_string_view<Char>
to_string_view(const basic_message<Char> & message) noexcept {
  const auto translated = message.sv(); // does the translation
  return { translated.data(), translated.size() };
}

} // namespace boost::locale
```

NUMERIC FORMATTING

In its current implementation, {fmt} uses the std::numpunct facet of the current global locale to format numbers when the 'n' numeric formatting specifier is given.

A problem is the fact that standard libraries are not required to provide locale facets for character types other than 'char' and 'wchar_t'.

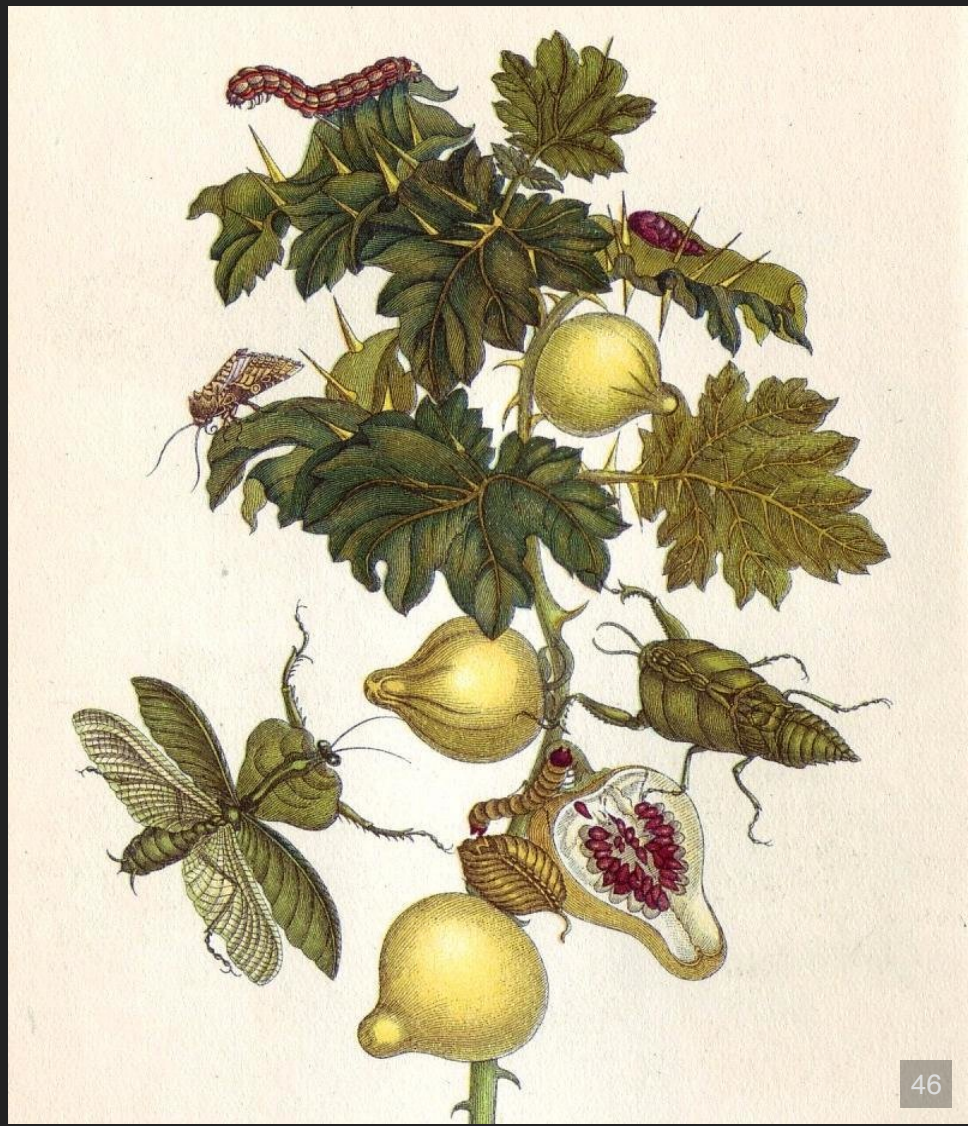
Formatting of floating-point types is backed by the compiler's library implementation which is usually locale-aware.

A super-fast floating-point formatter based on the Grisu2 algorithm is upcoming.

{fmt} it's about time

date

time



TIME FORMATTING

{fmt} has built-in support to format 'std::time_t' values from the C library:

```
#include <fmt/time.h>

const auto Time = localtime(time(nullptr));
string s = format("It is now {:%Y-%B-%d %H:%M:%S}.", Time);
```

Prints e.g.

"It is now 2018-November-05 19:38:43."

DATE FORMATTING

Formatting C++ date and time types requires the help of Howard Hinnant's *date* library or C++20.

This requires custom formatters for the types from std::chrono (and/or *date*):

- duration
- time_point
- zoned_time
- and many more

DATE FORMATTING

The layout of a custom-formatter base class for most 'std::chrono' and *date* types:

```
namespace fmt {

template <typename Char>
struct base_formatter {
  template <typename ParseContext>
  auto parse(ParseContext & ctx);

  template <typename T, typename FormatContext>
  auto base_format(const T & t, FormatContext & ctx);

private:
  basic_memory_buffer<Char> date_format; // holds the format spec
};

} // namespace fmt
```

DATE FORMATTING

The common parse function:

```
template <typename ParseContext>
auto parse(ParseContext & ctx) {
  const auto begin = ctx.begin();
  const auto end   = find(begin, ctx.end(), '}');
  if (end == begin) {
    // empty specification
    // provide some meaningful default
    date_format.push_back('%');
    date_format.push_back('x');
  } else {
    // copy given specification for later use
    date_format.reserve(end - begin + 1);
    date_format.append(begin, end);
  }
  date_format.push_back(0);
  return end;
}
```

DATE FORMATTING

The base_format function. It must be templated on the actual value type.

```
template <typename T, typename FormatContext>
auto base_format(const T & t, FormatContext & ctx) {
  // a stream-buffer adapter for back-insert iterators
  // (the implementation is not shown here)
  back_insert_buffer<FormatContext> buffer(ctx);

  // a temporary output stream required by the
  // formatting function
  basic_ostream<Char> output(&buffer);

  // the actual formatting function from date or C++20
  to_stream(output, date_format.data(), t);

  if (output.fail())
     ctx.on_error("invalid format specification");
  return buffer;
}
```

DATE FORMATTING

Derive the actual formatters for C++ time-point-related types.

```
// std::chrono::time_point
template <typename Clock, typename Duration, typename Char>
struct formatter<time_point<Clock, Duration>, Char> : base_formatter<Char> {
  template <typename FormatContext>
  auto format(const time_point<Clock, Duration> & tp, FormatContext & ctx) {
    return this->base_format(tp, ctx);
  }
};

// std::chrono::zoned_time or date::zoned_time
template <typename Duration, typename TimeZone, typename Char>
struct formatter<zoned_time<Duration, TimeZone>, Char> : base_formatter<Char> {
  template <typename FormatContext>
  auto format(const zoned_time<Duration, TimeZone> & zt, FormatContext & ctx) {
    return this->base_format(zt, ctx);
  }
};
```

DATE FORMATTING

Plus a special one for 'std::chrono::duration'. It's derived from the built-in numeric formatters to reuse their formatting specs and parsing.

```
// std::chrono::duration
template <typename Rep, typename Period, typename Char>
struct formatter<duration<Rep, Period>, Char> : formatter<Rep, Char> {
  using base_formatter = formatter<Rep, Char>;
  using unit_formatter = formatter<basic_string_view<Char>, Char>;

  template <typename FormatContext>
  auto format(const duration<Rep, Period> & d, FormatContext & ctx) {
    // the numeric value part
    auto it = base_formatter::format(d.count(), ctx);
    // a single space character according to SI formatting rules
    it = ' ';
    // the unit part
    const auto u = detail::get_units<Char>(typename Period::type{});
    return unit_formatter{}.format({ u.data(), u.size() }, ctx);
  }
};
```

DATE FORMATTING

The result:

```
const auto myBirthDate = local_days{ 11_d / mar / 1963 } + 5min;
auto result = format("My birth date is {:%A %d.%m.%Y %H:%M}",
                     myBirthDate);
```

returns "My birth date is Monday 11.03.1963 00:05"

```
const auto myBirthDate = local_days{ 11_d / mar / 1963 } + 5min;
const auto myBirthZoned = make_zoned(current_zone(), myBirthDate);
const auto minutesAfterMidnight = round<minutes>(myBirthDate)
                                - floor<days>(myBirthDate);

auto result = format("My birth date is {:%A %d.%m.%Y %H:%M %Z},"
                     " {} after midnight.",
                     myBirthZoned, minutesAfterMidnight);
```

"My birth date is Monday 11.03.1963 00:05 CET, 5 min after midnight."

{fmt} the future

further development

standardization

DEVELOPMENT

From my personal perspective, I see a lot of progress in the past couple of weeks which widened the scope considerably. But I also see a few points that might need to be improved:

- show better error messages which help users to easily identify mistakes
- the numeric formatting is still a bit weak on the locale-awareness
- create additional overloads which allow feeding locales into the formatting machinery for a single call only
- allow easier use of allocators
- give custom formatters access to formatting specifiers that are parsed by formatters that they derive from

STANDARDIZATION

The progress so far:

- 2017-05-22 P0645R0 first proposal
- 2017-11-10 P0645R1 second proposal
- 2018-04-28 P0645R2 third proposal
- 2018-10-07 P0645R3 fourth proposal

It was discussed last week in the San Diego meeting. So far, the LEWG has approved the design and forwarded the proposal to LWG for proper wording. At this stage, inclusion into C++20 possible.

I see one problem though: the proposal is (necessarily) much narrower in its scope than the reference implementation which contains all of the features presented here.

RESOURCES

Library

- fmtlib.net
- github.com/fmtlib/fmt
- CppCon 2017 talk
- standards proposal P0645

Contact:

- dani@ngrt.de
- github.com/DanielaE/fmt-MeetingCPP

Images: Maria Sibylla Merian (1705)

source: WikiMedia Commons, public domain

QUESTIONS?