nanofmt Documentation

Sean Middleditch and contributors

Jan 16, 2024

CONTENTS

nanofmt aims to provide a lite-weight semi-compatible implementation of the excellent fmtlib. This can be used in environments or team cultures where neither std::format nor fmtlib are available for use.

Contents

CHAPTER

ONE

API

Contents

1.1 Formatting

The format API is available in the header `nanofmt/format.h`.

The header `nanofmt/forward.h` offers forward declarations of nanofmt types, including the `formatter<T>` template that users must specialize to support custom types.

Extensions for C++ standard library string types are in the header `nanofmt/std_string.h`.

1.1.1 Format to Array

The *nanofmt::format_to()* functions format a given format string and arguments into the target buffer. The result will be NUL-terminated. The return value is a pointer to the terminating NUL character.

The *nanofmt::format_append_to()* functions format a given format string and arguments onto the end of target buffer. The result will be NUL- terminated. The return value is a pointer to the terminating NUL character.

char *nanofmt::**format_to**(char (&dest)[N], *format_string* format_str, Args const&... args)

char *nanofmt::**vformat_to**(char (&dest)[N], *format_string* format_str, *format_args* args)

char *nanofmt::**format_append_to**(char (&dest)[N], *format_string* format_str, Args const&... args)

char *nanofmt::**vformat_append_to**(char (&dest)[N], *format_string* format_str, *format_args* args)

1.1.2 Length-Delimited Formatting

The *nanofmt::format_to_n()* functions format a given format string and arguments into the target buffer, up to the given number of characters. The result will **NOT** be NUL-terminated. The return value is a pointer to one past the last character written.

The *nanofmt::format_append_to_n()* functions format a given format string and arguments onto the end of the target buffer, up to the given number of characters. The result will **NOT** be NUL-terminated. The return value is a pointer to one past the last character written.

char *nanofmt::**format_to_n**(char *dest, std::size_t count, *format_string* format_str, Args const&... args)

char *nanofmt::**vformat_to_n**(char *dest, std::size_t count, *format_string* format_str, *format_args*&&)

char *nanofmt::**format_append_to_n**(char *dest, std::size_t count, *format_string* format_str, Args const&... args)

char *nanofmt::**vformat_append_to_n**(char *dest, std::size_t count, *format_string* format_str, *format_args*&&)

1.1.3 Custom Formatters

The `nanofmt::formatter` template must be specialized to add support for user-provided types.

Two member functions, `parse` and `format`, must be implemented on the specialized structure for nanofmt to work.

template<typename **T**>
struct `nanofmt::`**`formatter`**

Custom formatter. May include any member variables necesasry to convey format information from `parse` to `format`.

char const *`parse`(char const *in, char const *end)

Consumes characters from `in` up to, but not including, `end`. Returns a pointer to one past the last character consumed.

void `format`(*T* const &value, *format_output* &out) const

Formats `value` to `out`.

A header implementing a custom formatter may choose to only depend on `nanofmt/foward.h` header. This header does not offer any of the implementations, nor does it provide declarations of the formatting functions. A formatter may work around this by specifying the `format_output&` parameter of `format` as a template, as in:

```
#include <nanofmt/forward.h>

namespace nanofmt {
  template<>
  struct formatter<my_type> {
    constexpr char const* parse(char const* in, char const*) noexcept;

    template <typename OutputT>
    void format(my_type const& value, OutputT& output);
  }
}
```

1.1.4 Format Length

The *nanofmt::format_length()* function returns the length of result of formatting the given format string and arguments, excluding any terminating NUL character.

size_t nanofmt::**format_length**(*format_string* format_str, Args const&... args)

size_t nanofmt::**vformat_length**(*format_string* format_str, *format_args* args)

1.1.5 Output Buffers

struct nanofmt::**format_output**

1.1.6 Format to Buffer

The nanofmt::format_output& overloads of *nanofmt::format_to()* format a given format string and arguments into the target buffer. The result will **not** be NUL-terminated. The return value is the buffer object itself.

format_output &**format**(format_string fmt, Args const&... args)
: Formats the given format string and argument into the buffer.

format_output &**vformat**(format_string fmt, format_args args)
: Formats the given format string and argument into the buffer.

constexpr format_output &**append**(char const *const zstr) noexcept
: Appends the contents of `zstr` to the buffer.

constexpr format_output &**append**(char const *source, std::size_t length) noexcept
: Appends `length` characters from `source` to the buffer.

constexpr format_output &**put**(char ch) noexcept
: Appends the character `ch` to the buffer.

constexpr format_output &**fill_n**(char ch, std::size_t count) noexcept
: Appends `count` copies of the character `ch` to the buffer.

constexpr format_output &**advance_to**(char *const p) noexcept
: Updates the buffer position to `p` and adjusts the `advance` member appropriately.

char *pos = nullptr

Current output position of the buffer. For custom formatting operations, use this value for the output position. The *advance_to()* function should always be preferred for mutating the pos member.

char const *end = nullptr

The end pointer for the buffer. Custom formatting code should never advance pos past the end pointer, and should never dereference end.

std::size_t advance = 0

The number of characters that were written to the buffer, ignoring any truncation. Even when pos equals end, operations on the buffer will still increment advance.

The *advance_to()* member function should be preferred over directly mutating advance.

1.1.7 String Utilities

General string utiltities that are useful in implementing formatting.

char *copy_to(char *dest, char const *end, char const *source) noexcept

Copy the source string to the destination buffer, but not extending past the provided buffer end pointer. Returns the pointer one past the last character written.

char *copy_to_n(char *dest, char const *end, char const *source, std::size_t length) noexcept

Copies up to length characters of source string to the destination buffer, but not extending past the provided buffer end pointer. Returns the pointer past the last character written.

char *put(char *dest, char const *end, char ch) noexcept

Copies the provided character ch to the destination buffer, but not extending past the provided buffer end pointer. Returns the pointer past the last character written.

char *fill_n(char *dest, char const *end, char ch, std::size_t count) noexcept

Copies count copies of the charcter ch to the destination buffer, but not extending past the provided buffer end pointer. Returns the pointer past the last character written.

std::size_t strnlen(char const *buffer, std::size_t count) noexcept

Returns the length of the string in buffer, to a maximum of count.

1.1.8 Format Strings

nanofmt uses a *nanofmt::format_string* structure for receiving its format strings, to decouple from and support various string types and classes. Many string types should automatically convert to format_string; for string types that don't already support conversion to format_string, a *nanofmt::to_format_string()* function can be implemented.

struct nanofmt::format_string

Receiver for format strings. Only implicitly constructs from string literals (constant character arrays). Can be explicitly constructed from other string types.

template<typename T>
format_string nanofmt::to_format_string(*T* const &value) noexcept

Converts a given string type to a *nanofmt::format_string*. Works on most standard string types with data() and size() member functions.

Overload to add support for other string types that require different means of converted to a format_string.

struct nanofmt::**format_string_view**

A very simple wrapper around a pointer and length. This is provided to make it easier to write nanofmt::formatter specializations that work on length-delimited string views, by deriving from nanofmt::formatter<format_string_view>.

char const ***string** = nullptr

std::size_t **length** = 0

1.1.9 Variadic Arguments

struct nanofmt::**format_args**

List of format args. Typically only constructed from *nanofmt::make_format_args()*. Does not take ownership of any internal data.

> **Warning:** Storing an instance of format_args can result in dangling references.

auto nanofmt::**make_format_args**(Args const&... args)

> **Danger:** This function should usually only be used directly in a call to a function accepting a *nanofmt::format_args* parameter.

1.2 Character Conversion

The character conversion API is available in the header nanofmt/charconv.h.

char *nanofmt::**to_chars**(char *dest, char const *end, IntegerT value, *int_format* fmt = *int_format*::*decimal*) noexcept

Formats value into the buffer using the base specified in fmt.

char *nanofmt::**to_chars**(char *dest, char const *end, FloatT value, *float_format* fmt) noexcept

Formats value into the buffer using the base specified in fmt. Uses the shortest precision.

char *nanofmt::**to_chars**(char *dest, char const *end, FloatT value, *float_format* fmt, int precision) noexcept

Formats value into the buffer using the base specified in fmt. Uses the given precision, whose meaning depends on the specified format.

enum class nanofmt::**int_format**

Specify whether to use base 10, base 16, or base 2. Base 16 has an uppercase variant.

enumerator **decimal**

Base 10.

enumerator **hex**

Base 16.

enumerator **hex_upper**

enumerator **binary**

Base 2.

enum class nanofmt::**float_format**

Specify whether to use scientific, fixed, or general precision formatting. Scientific and general also have uppercase variants.

enumerator **scientific**

Scientific notation formats floating point values as [-]d.de[+-]dd.

enumerator **scientific_upper**

enumerator **fixed**

Fixed-point notation formats floating point values as [-]d.dddd.

enumerator **general**

General precision notation formats in either scientific or fixed-point notation, depending on the exponent of the value.

enumerator **general_upper**

CHAPTER

TWO

DESIGN

Contents

2.1 Origins

The design of nanofmt started by trying to closely match the interface of `std::format` and fmtlib. This includes the general naming of functions and types as well as the design of the `make_format_args` and related utilities.

2.2 Output Iterators

A major design note of nanofmt is that the only output target supported are `char` arrays. This is one of the largest simplifying factors in nanofmt vs either `std::format` or fmtlib.

This is feasible since fixed-size buffers and the complete avoidance of any allocating routines is exceedingly common in nanofmt's target domains. In the rare cases that arbitrary-length formatting is required, the use of `format_length` allows pre-allocating any necessary buffer.

For one, arbitrary output iterator support effectively requires all formatting code to live in headers, with only the occassional pieces living in individual TUs. This is because the `format_context<>` in `std::format`/fmtlib is itself a type template.

Supporting arbitrary output iterators requires having full output iterator machinery in the first place. Output iterators that need to functions with fixed-length outputs are non-trivial to write.

More importantly, making such generalized output iterator support reasonably fast requires *even more* complexity in terms of temporary buffers, buffer flushing, and so on. Such buffering mechanisms are also required to enable any kind of efficient type-erasure of the output iterators.

There's even further machinery necessary for making common output iterators efficient. Consider `back_inserter<T>`. An efficient implementation of `std::format` needs to detect that its output iterator is a `back_inserter` and transparently replace calls to `push_back` to more efficient `append` or `insert` invocations.

The use of these output iterators is mostly to support things like streaming to console IO, to support `push_back` into containers like `std::string`, or esoteric filtering mechanisms. None of these are essential to nanofmt's target use cases.

The design and implementation pioneered by Victor Zverovich for fmtlib is some honestly amazing engineering! Victor and other fmtlib contributors deserve nothing but praise and respect for the incredible amount of work done to make fmtlib (and by extension `std::format`) feel natural, intuitive, and unsurprising in C++ while still having exceptionally good runtime efficiency.

nanofmt however is more for teams that feel that C-like APIs like `snprintf` are already the epitome of being natural, intuitive, and unsurprising; except of course for the limitations imposed by C's varargs vs C++'s variadic tempaltes.

Ultimately, the complexity cost of supporting other kinds of output iterators is high, and the benefit for nanofmt users is low.

2.3 Localization

nanofmt only supports `char` and does not bother with any of `wchar_t`/ `char8_t`, `char16_t`, or `char32_t`.

Additionally, the `L` format specification flag is parsed but ignored.

nanofmt only supports `char` because that is, by and large, the only character type in active use in its target domains. Target systems can and do assume that all `char*` strings are UTF-8. The type-system bifurcation created by `char8_t` has caused problems for the few projects that took to using `u8""` literals, as it required every function taking a `char*` to offer a second overload that also accepted `char8_t*`. The result has mostly been projects starting to use `u8` literals in C++17 and abandoning them soon after since C++20 compatibility changed the types in a very incompatible and gratuitous way.

The localization flags are unsupported as the target uses of nanofmt tend not to ever bother with localization. Logging can actually be harmed by localization as it makes log parsers and alert systems far more difficult to deploy and maintain. "User-facing" formatting in nanofmt's target domains is generally just developer tools and utilities, which are effectively never localized due both to the cost of localizing propriety in-house tools and to the rapid rate of change in such tools; keeping localization up-to-date is not especially feasible in those environments.

While nanofmt targets games developers in particular, and games *are* heavily and frequently localized, it is not expected that nanofmt would be used for player-facing text. Game UI text tends to use heavily specialized toolkits and rely on iconography, layout, color and style, and other factors to convey information; nanofmt-like text formatting is exceptionally rare in such UI.

2.4 Character Conversion

nanofmt includes implementations of `to_chars` mostly because there are shipping "C++17" implementations that are very much in common use in target industries but which do not offer complete `to_chars` implementations. This is especially true for `float/double`.

nanofmt uses the Dragonbox reference implementation in its floating-point `to_chars` implementations. There are no fallbacks to other implementations as found in fmtlib or `std::to_chars`. This in particular limits the precision of fixed-point formats in nanofmt.

The precision limitation is not currently believed to be a showstopper, but may be revisited if use cases from nanofmt users illustrates a strong need for more intricate fixed-point formatting.

The Dragonbox reference implementation is used for the work-horse portions of floating-point to decimal conversion.

2.5 constexpr

Most of nanofmt is not constexpr. This is an intentional choice.

Making a constexpr-friendly formatting library unfortunately requires that most of the implementation of the formatters and all supporting machinery also be constexpr, which in turn means it all has to live in headers.

This might be a much smaller issue once we're all living with C++20 modules, but today, the cost paid by every user for constexpr capabilties is very high.

That said, constexpr formatting is a generally useful feature. nanofmt may, if use cases arise, offer a second `const_format.h` header or the like which includes/imports constexpr definitions of the format implementations. Such an approach would allow individual TUs to opt-in to pulling in all the machinery if and only if they actually need it.

Note that we have thus far kept the parsing* part of nanofmt all constexpr capable, since we may wish to enable compile-time format string checking capabilities for projects that (wisely) prefer such a feature. All known potential users of nanofmt are not yet using C++20 so compile-time checking isn't a priority. We may instead opt to just entirely drop the constexpr parsing support if we decide we're not going to support it anytime soon.

2.6 Alignment, Width, Alternate Forms, and Other Specifiers

nanofmt implements are relatively limited set of the fmtlib/`std::format` specifiers. All are parsed, but most are ignored.

This isn't a "design" so much as just not having had the use cases made for supporting all of them yet. Alternate form for integers is high on the list, though.

The goal isn't to be feature-complete, and some of these specifiers are *juuust* annoying enough to implement that it'll only be done on-demand.

CHAPTER

THREE

FAQ

Contents

3.1 Who is This For?

Developers and teams who are still choosing to use snprintf or related technologies instead of fmtlib or std::format.

Developers and teams who are using `snprintf` and have explicitly rejected fmtlib, but still want to get its two biggest features: type-aware formatting and user-extensible type support.

Said developers are still okay with accepting `snprintf`'s other limitation of only being able to write to character buffers.

3.2 Why??

The author has worked on multiples teams in the AAA games industry that have strong cultural tendencies to prefering `snprintf` for all string formatting. Reasons cited are usually something like:

- Minimizing dependencies, as a reason not to use fmtlib.
- Supporting older compilers, as a reason not to use `std::format`.
- Deep distrust of standard headers, as a reason to avoid `std::format`; and any library that uses many standard headers, as a reason not to use fmtlib.
- Dislike of namespaces. Note that nanofmt at this time doesn't "help" here but it's a very possible/probable future evolution.
- Compile times, as a reason to reinvent every wheel.

3.3 Compile Times? Really?

That's two questions, but yes and yes.

It is common (though not universal!) in industries like AAA game development to heavily optimize for compilation times. The primary reason for this is *iteration speed*.

This is a large, complex, and nuanced topic. It is very inaccurate to say that "all game developers" need or care about X.

The best summary for this FAQ, though: some developers critically value having very fast edit-run-test cycles and will spend considerable time and effort up-front to make those cycles faster later.

nanofmt is meant for teams who include "minimizing C++ compilation times" in their efforts to achieve the fastest edit-run-test cycles.

3.4 Why Avoid Standard Headears?

Historically, many standard headers have been very "heavy," introducing tens or hundreds of thousands of lines of complex C++ into any translation unit including them. Compile times have suffered.

Beyond the headers themselves, standard library implementations are generally written to a level of completeness and foolproof-ness that imposes costs some developers don't wish to pay for.

For example, the *checked iterators_* machinery in MSVC and the equivalents in libstdc++ and libc++ impose a cost. Even when disabled, the cost is found in numerous small wrapper functions and other utilities that tend to decrease compiler throughput for an otherwise disabled feature.

Teams that care about these items will often write code that is more like C, with plenty of raw pointers and thin abstractions, simply because it's easier and faster for the compiler to process.

3.5 How Does nanofmt Help With Compile Times?

To be clear, nanofmt at this point has not been extensively benchmarked with any rigor, and we're only _assuming_ it helps.

The design around not using *output iterators* is one way nanofmt aims to improve over fmtlib or `std::format` for teams that deeply care about compile times.

In general, nanofmt is written to be more C-like in the sense that abstractions are minimal, plain values are used where possible, and raw pointers are used exclusively in place RAII containers, smart pointers, or iterator wrappers.

The result is simply a lot less code to do (some of) the same stuff. The loss of abstractions necessarily comes with a large loss of functionality and flexibility. nanofmt is the result when the choice is made to lean in favor of simpler (and faster to compile) code rather than more featureful but complicated code.

3.6 No IO Support For Real?

Not at the time of writing, no. The author's experience is that actual direct-to-console writing is (relatively) rare, even with developer tools, in target domains. Direct IO to files is very rare, and directly IO to socket streams is close to unheard of, in target domains.

Where console IO does happen, these are usually either tools that are far more open to using standard libraries or librarieies like fmtlib. The few remaining cases can make do just fine with using `char` arrays as a temporary buffer.

Yes, it's slower and more constricting to writing to a buffer before writing to (already buffered) standard output facilities. These aren't the areas of performance that the kinds of teams who might use nanofmt really care about, though.

3.7 Won't C++20 Modules Make This Obsolute?

Maybe? Hopefully? Less duplicate code to maintain is only a good thing. I will only be happy if I never have to personally think about a floating- point string conversion routine again.

The reality is likely a bit more murky. For one, nanofmt exists *now* but C++20 modules are possibly still years away from even being a viable option for new projects. At the time of writing, compiler support is incomplete and very buggy; build system support is nearly non-existant; module-aware linters and doc generators and like is also non-existant; and the general user and ecosystem support can best be described as nascent.

Further, note that modules only help *part* of the compile time overhead. At best, we can expect modules to reduce the cost of parsing large header hierarchies. While that is a significant amount of the time incurred with compiling libraries like fmtlib or `std::format`, another large chunk of the time goes into instantiating templates, resolving function overloads, evaluating constexpr functions, and so on.

nanofmt, by virtue of steeply limiting its feature set and general applicability, aims to reduce the need for as much of that "use time" overhead as possible. While it's almost certainly impossible to hit the minimal compile-time of `snprintf`, the goal is to keep the difference small enough that the "developer time" benefits of a type-safe user- extensible format library outweighs the compile time costs.

3.8 What Does nanofmt Support?

In general, it supports type-aware and user-extensible formatting using the *standard format specification*<*std-format-spec*>, mostly.

It supports writing to length-delimited `char*` arrays.

Support exists for formatting most standard built-in C++ types, including typical integer and floating-point types, booleans, characters, raw pointers, and C-style strings.

The `std_string.h` header may be included for `std::basic_string` and `std::basic_string_view` support.

3.9 What Does nanofmt Not Support?

There is no support for output iterators other than `char*`.

There is no support for character types other than `char`.

There is no support for locales.

There is no formatter support for standard library types. The `std_string.h` header enables support for standard string types.

There is no support for console or file IO.

There is no support for versions of the language older than C++17.

There is no drop-in API compatibility with either fmtlib or `std::format`.

There is no support for `long double` and no suport for `(u)int128_t`.

Any feature of fmtlib or `std::format` not explicitly named in this or the prior section should likely be considered unsupported.

3.10 Does it Support Floating-Point Types?

Yes, nanofmt has support for both `float` and `double`.

The Dragonbox reference implementation is used for the work-horse portions of float to decimal conversion.

3.11 Was it Worth It?

Probably not.

The nanofmt author has implemented several fmtlib replacements for work.

In comparison, the author has been working on nanofmt for 12+ hours/day for about a week; and that doesn't include all the time the author spent building the precursors to nanofmt in personal projects, going all the way back to formatxx (an "ancient" C++11 library), and including re-writing formatxx to incorporate into commercial game codebases with specialized requirements (like drop-in Boost.Format compatibility).

3.12 Will This be Maintained?

Excellent question.

… too soon to tell. If having a dedicated maintainer is important to you, this library might be a little too new and untested for your needs.

As stated in the *C++ modules FAQ question*, there's a very real future where this entire library is obsolete. To that end, while nanofmt is not a direct drop-in replacement for `std::format`, it aims to be "close enough" that migrating from nanofmt to the standard equivalent is meant to be straightforward.

CHAPTER

FOUR

BENCHMARKS

4.1 Compilation

TBD

4.2 Execution

TBD

CHAPTER

FIVE

LICENSE

nanofmt is released under the MIT license.

nanofmt uses the Dragonbox reference implementation by Junekey Jeon which is released under either the Apache License Version 2.0 with LLVM Exceptions or the Boost Software License Version 1.0.

CHAPTER

SIX

BASIC USAGE

Example usage of writing a string with two variables formatted into the output.

```
char buffer[128];
nanofmt::format_to(buffer, "Hello, {0}! You are visitor {1}.",
  UserName,
  VisitorCount);
```

See also our *Detailed Examples*.

CHAPTER

SEVEN

API OVERVIEW

The crux of the API is writing to a `char` arrays. Writing directly to an array just works:

```
char buffer[128];
format_to(buffer, "Format this {}", 128);
```

A pointer to a buffer and a length can also be provided for safe writing.

```
format_to_n(buffer, size, "Format this {}", 128);
```

The format functions all return a `char*` pointing to the terminating NUL that is always written to the buffer.

```
char* end = format_to(buffer, "{} is {}", seven, 7);
size_t const size = end - buffer;
```

The nanofmt functions understand the same positional arguments and most format flags of std::format.

```
format_to(buffer, "{1} then {0}", "Second", "First");
// buffer: First then Second
```

If the length of formatted text is required, e.g. for allocating buffer space, the *nanofmt::format_length()* function can be used:

```
size_t const length = format_length("{} plus {}", 7, 7);

char* dest = (char*)malloc(length + 1/*NUL byte*/);
format_to_n(buffer, length + 1, "{} plus {}", 7, 7);
```

nanofmt also includes implementation of the C++17 standard to_chars functions, for codebases that are unable or unwilling to use the standard versions, or who are using older compilers that lack support for floating-point `std::to_chars`

See the *API* for more in-depth coverage of the nanofmt facilities.

CHAPTER

EIGHT

THE CASE FOR NANOFMT

nanofmt may be a good fit for teams or codebases which are unable or unwilling to use fmtlib or std::format, particularly if the reasons involve compilation time or standard library header dependencies.

The only headers nanofmt relies on are <type_traits>, <cstddef>, <cstdint>, and <cmath>.

Teams that might otherwise continue to prefer using `snprintf` may find that nanofmt is far more to their tastes.

Anyone unsure of whether they should use nanofmt as their should almost certainly consider using fmtlib or `std::format` instead.

Both fmtlib and the standard formatting facilities offer far more features, far more idiomatic C++ support, and integrate far better with both the rest of the C++ ecosystem and core IO.

CHAPTER

NINE

DETAILED EXAMPLES

9.1 Custom Types

Provide a specialization of *nanofmt::formatter* to enable nanofmt to consume values of a custom type.

```
struct MyType {
  std::string FirstName;
  std::string LastName;
};

namespace nanofmt {
  template <>
  struct formatter<MyType> {
    bool reverse = false;

    constexpr char const* parse(char const* in, char const* end) noexcept;
    inline void format(MyType const& value, buffer& output) noexcept;
  };
}

char buffer[128];
format_into(buffer, "Greetings {:r}!", MyType{"Bob", "Bobson"});

// buffer would contain:
// Greetings Bobson, Bob
```

What does that `:r` do? That's a custom formatter flag supported by the `MyType` formatter. A possible implementation:

```
constexpr char const* nanofmt::formatter<MyType>::parse(char const* in, char const* end)␣
↪noexcept {
  if (in != end && *in == 'r')
    ++in;
    reverse = true;
  }
  return reverse;
}

void nanofmt::formatter<MyType>::format(MyType const& value, buffer& output) noexcept {
  if (reverse)
    format_into(output, "{}, {}", LastName, FirstName);
  else
```

(continues on next page)

(continued from previous page)

```
    format_into(output, "{} {}", FirstName, LastName);
}
```

9.2 Length-Delimited Buffers

nanofmt automatically length-delimits any attempt to write to a *char[]* array. When using raw pointers, or to futher constrain the output length, the *format_into_n* functions may be used.

The format functions return a pointer to the last character written (excluding the NUL byte). This can be used to chain format calls, or to calculate the written length.

```
char* ptr = GetBufferData();
size_t size = GetBufferSize();

char* end = format_into_n(ptr, size, "Format this! {}", value);

size_t const length = end - ptr;
```

INDEX