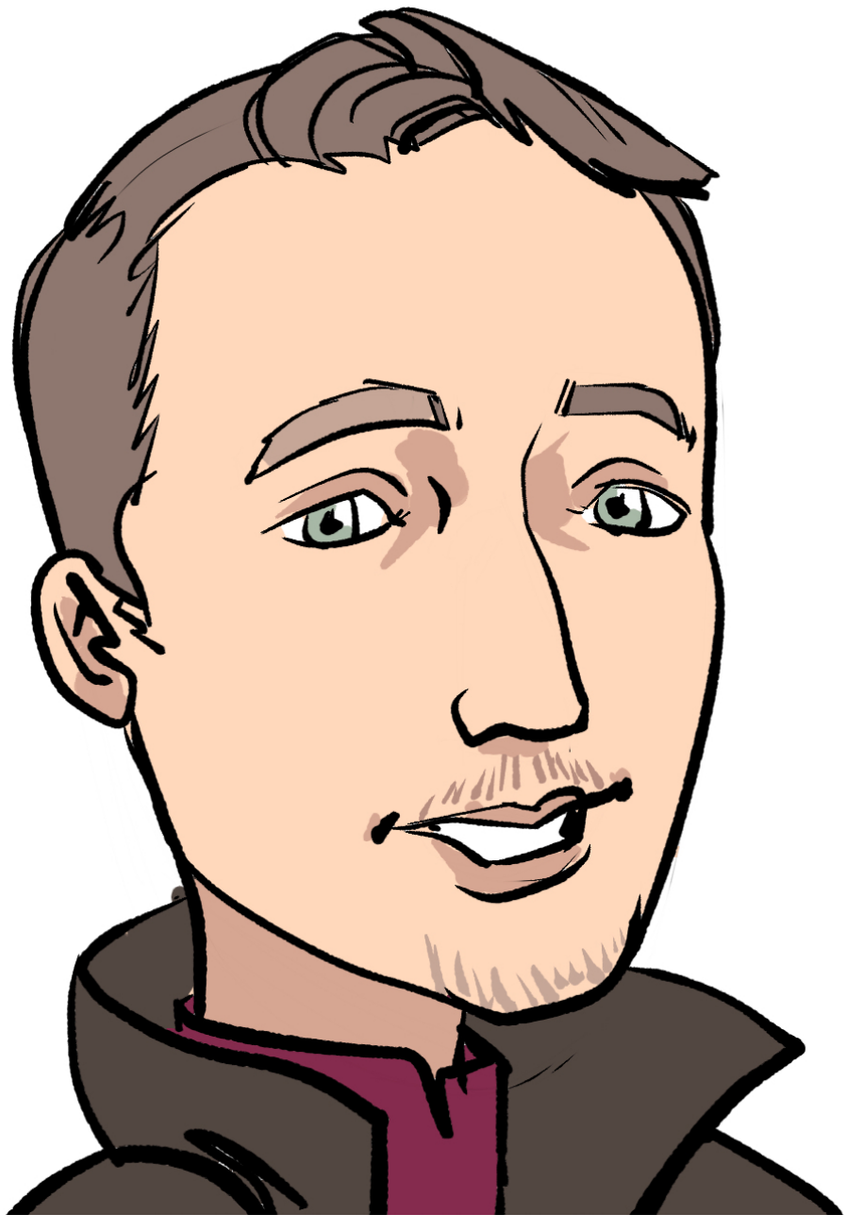
{fmt}: The Cool Parts

Victor Zverovich

About me

- 🔊 VIK-ter ZVE-roh-vich
- Work at Meta on the Thrift RPC & serialization framework
- Author of the {fmt} library, std::format and std::print
- Expert in negative zero
- https://github.com/vitaut
- https://www.threads.net/@signedvoid



Compile time

Compile-time checks

Format string compilation

Compile-time formatting

Compile time checks

Checks in printf

```
printf("%d", "I am not a number”);
```

```
warning: format '%d' expects argument of type 'int', but
argument 2 has type 'const char*' [-Wformat=]
    4 |   printf("%d", "I am not a number");
      |           ~^    ~~~~~~~~~~~~~~~~~~~~
      |            |    |
      |            int const char*
      |           %s
```

when compiled with -Wformat

👍 very nice 👍

Checks in printf

```
size_t n = 42;
printf("%d", n);
```

```
warning: format '%d' expects argument of type 'int', but
argument 2 has type 'size_t' {aka 'long unsigned int'} [-
Wformat=]
    5 |   printf("%d", n);
      |           ~^   ~
      |            |   |
      |            int size_t {aka long unsigned int}
      |           %ld
```

Can you spot the problem?

Checks in printf

```
size_t n = 42;
printf("%ld", n);
```

```
warning: format '%ld' expects argument of type 'long int',
but argument 2 has type 'size_t' {aka 'unsigned int'} [-
Wformat=]
    5 |   printf("%ld", n);
      |           ~~^   ~
      |             |   |
      |             |   size_t {aka unsigned int}
      |             long int
      |           %d
```

when compiled with on a different platform

Checks in {fmt}

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

int main() {
  fmt::print("{:d}", "I am not a number");
}
```

```
$ gcc -std=c++20 test.cc
...
test.cc:4:13: in 'constexpr' expansion of
'fmt::v8::basic_format_string<char, const char (&)
[18]>("{:d}")'
...
fmt/core.h:2783:54: error: call to non-'constexpr' function
'void fmt::v10::detail::throw_format_error(const char*)'
 2308 | throw_format_error("invalid format specifier");
      | ~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Can I haz checks?

Writing your own function with compile-time format string checks:

```
template <typename... T>
void log(fmt::format_string<T...> fmt, T&&... args) {
  fmt::print(log_file, fmt, std::forward<T>(args)...);
}

log("The answer is {}.", 42);
```

How it works

- C++20 `consteval`:

  specifies that a function is an *immediate function*, that is, every call to the function must produce a compile-time constant (cppreference)

- C++20 `type_identity`: suppresses template argument deduction

- Converting constructors

How it works

```
template <typename T> struct basic_format_string {
  std::string_view str;

  template <typename S>
  consteval basic_format_string(const S& s) : str(s) {
    // Parse s and check if format specifiers are valid for types T...
    const char* p = s;
    while (char c = *p++) {
      if (c != '{') continue;
      c = *p++;
      if (c == '}') continue;
      auto f = fmt::formatter<T>();
      auto ctx = fmt::format_parse_context(p);
      p = f.parse(ctx);
    }
  }
};

template <typename... T>
using format_string = basic_format_string<std::type_identity_t<T>...>;
```

How it works

```
template <typename T> struct basic_format_string {
  // ...
};

template <typename... T>
using format_string =
  basic_format_string<std::type_identity_t<T>...>;

template <typename... T>
void check(format_string<T...>, T&&...) {}

check("{:d}", 42); // OK
check("{:s}", 42); // error
```

User-defined types

```
struct answer { int value; };

template <>
struct fmt::formatter<answer> {
  bool hex = false;

  constexpr auto parse(format_parse_context& ctx) {
    auto it = ctx.begin();
    if (it == ctx.end() || *it != 'x') return it;
    ++it;
    hex = true;
    return it;
  }

  auto format(answer a, format_context& ctx) const {
    return hex ? format_to(ctx.out(), "{:x}", a.value) :
                 format_to(ctx.out(), "{}", a.value);
  }
};

fmt::print("{}\n", answer{42});   // 42
fmt::print("{:x}\n", answer{42}); // 2a
fmt::print("{:d}\n", answer{42}); // error
```

Format string compilation

Compilation in sprintf

```c
int main() {
  printf("Hello, world!\n");
}
```

compiles to

```
.LC0:
        .string "Hello, world!"
main:
        push    rbp
        mov     rbp, rsp
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        mov     eax, 0
        pop     rbp
        ret
```

Compilation in sprintf

```
char* format_answer(char* buf) {
  return buf + sprintf(buf, "%d", 42);
}
```

compiles to

```
.LC0:
        .string "%d"
format_answer(char*):
        push    rbx
        mov     edx, 42
        mov     rbx, rdi
        mov     esi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    sprintf
        lea     rax, [rbx+2]
        pop     rbx
        ret
```

Compilation in {fmt}

```
char* format_answer(char* buf) {
  using namespace fmt::literals;
  return fmt::format_to(buf, "{}"_cf, 42);
}
```

compiles to

```
format_answer(char*):
        mov     eax, 12852 ; ('2' << 8) + '4'
        mov     WORD PTR [rdi], ax
        lea     rax, [rdi+2]
        ret
```

How it works

- `constexpr` parse functions in `formatter` specializations
- C++20 class types in non-type template parameters (NTTP)
- C++17 constexpr if
- C++11 user-defined literals with C++20 NTTP support

Non-type template params

```
template <size_t N>
struct fixed_string {
  char data[N] = {};

  constexpr fixed_string(const char (&s)[N]) {
    std::copy_n(s, N, data);
  }
};

template <fixed_string S, typename... T>
std::string format(T&&... args);

auto s = format<"{}">(42); // Not recommended!
```

User-defined literals

```
template <fixed_string S>
struct compiled_format {
  static constexpr
    std::string_view str = S.data;
};

template <fixed_string S>
auto operator ""_cf() {
  return compiled_format<S>();
}

auto s = "{}"_cf; // -> operator""_cf<"{}">()
```

User-defined literals

```
template <fixed_string S>
struct compiled_format {
  static constexpr std::string_view str = S.data;
};
```

```
template <fixed_string S>
auto operator ""_cf() {
  return compiled_format<S>();
}

template <fixed_string S, typename... T>
std::string format(compiled_format<S>, T&&... args);

auto s = format("{}"_cf, 42);
```

constexpr if

```
template <typename T, typename... Tail>
const T& first(const T arg, const Tail&...) {
  return arg;
}
```

```
template <fixed_string S, typename... T>
std::string format(compiled_format<S> fmt,
                   T&&... args) {
  if constexpr (fmt.str == "{}")
    return fmt::to_string(first(args...));
  return fmt::format(fmt.str, args...);
}

auto s = format("{}"_cf, 42);
```

Compiled format

```
constexpr auto compiled = fmt::detail::compile<int>(
  "The answer is {}."_cf);

decltype(compiled):

struct concat<
  text<char>,
  concat<
    field<char, int, 0>, // 0 is arg ID.
    code_unit<char>>>> {

  template <typename OutputIt, typename... T>
  OutputIt format(OutputIt out,
                  const T&... args) const;
};
```

Compiled format

```
template <typename Char> struct text {
  std::basic_string_view<Char> data;

  template <typename OutputIt, typename... Args>
  OutputIt format(OutputIt out, const Args&...) const {
    return write<Char>(out, data);
  }
};

template <typename Char, typename T, int N> struct field {
  template <typename OutputIt, typename... Args>
  OutputIt format(OutputIt out,
                  const Args&... args) const {
    return write<Char>(out,
                       get_arg_checked<T, N>(args...));
  }
};
```

Compiled format

```
char buf[32] = {};
using namespace fmt::literals;

fmt::format_to(buf, "The answer is {}."_cf, 42)
```

is equivalent to

```
// Compilation:
constexpr auto compiled =
  fmt::detail::compile<int>(
    "The answer is {}."_cf);

// Formatting:
compiled.format(buf, 42);
```

Benchmark

https://github.com/miloyip/itoa-benchmark

```
void i32toa(int32_t value, char* buffer);

// fmt_compile:
auto end = fmt::format_to(buffer, FMT_COMPILE("{}"), value);
*end = '\0';

// fmt_runtime:
auto end = fmt::format_to(buffer, "{}", value);
*end = '\0';

// sprintf:
sprintf(buffer, "%d", value);

// to_chars:
auto result = std::to_chars(buffer, buffer + 11, value);
*result.ptr = '\0';

// to_string:
strcpy(buffer, std::to_string(value).c_str());

// ostrstream:
std::ostrstream oss(buffer, 12);
oss << value << std::ends;
```

Benchmark

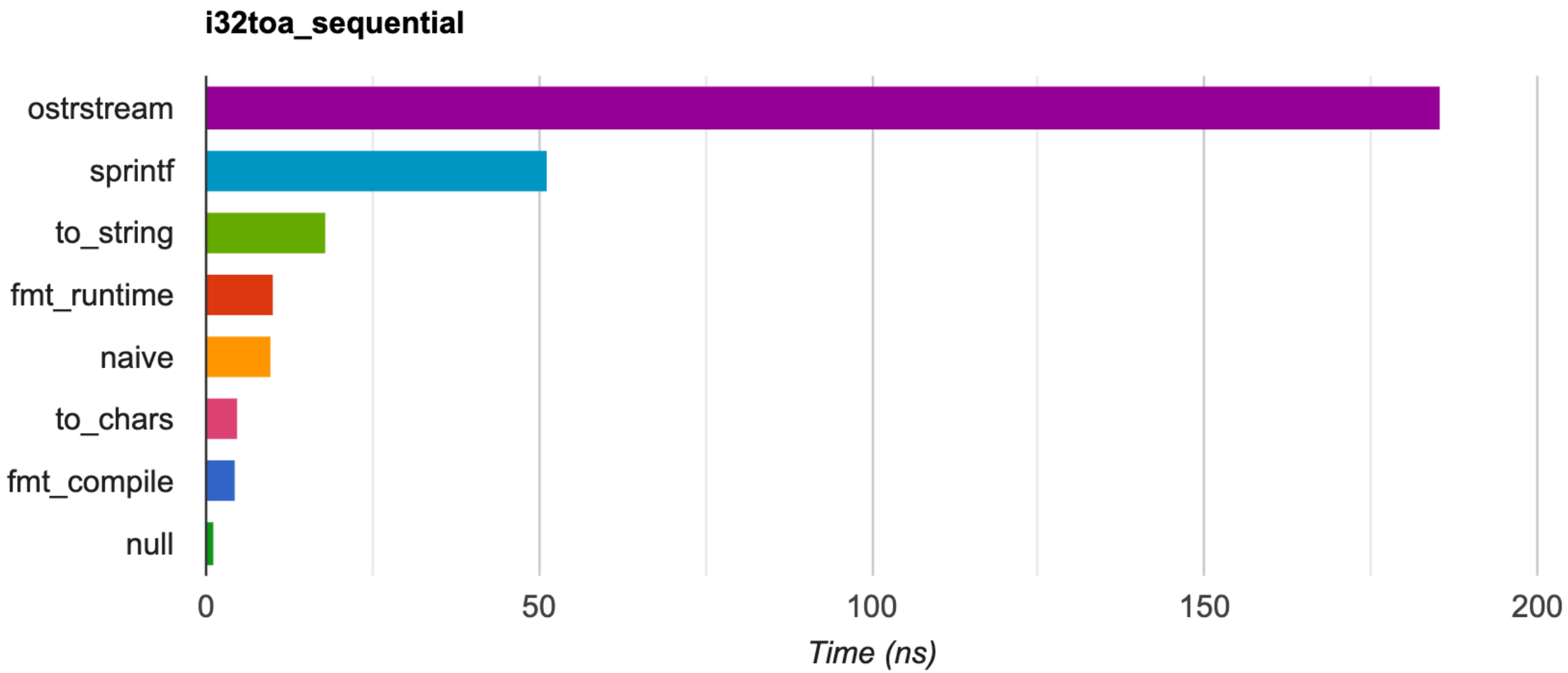
i32toa_sequential

Function	Time (ns)	Speedup
ostrstream	185.284	1.00x
sprintf	51.260	3.61x
to_string	17.952	10.32x
fmt_runtime	9.999	18.53x
naive	9.834	18.84x
to_chars	4.503	41.15x
fmt_compile	4.438	41.75x
null	1.110	166.92x

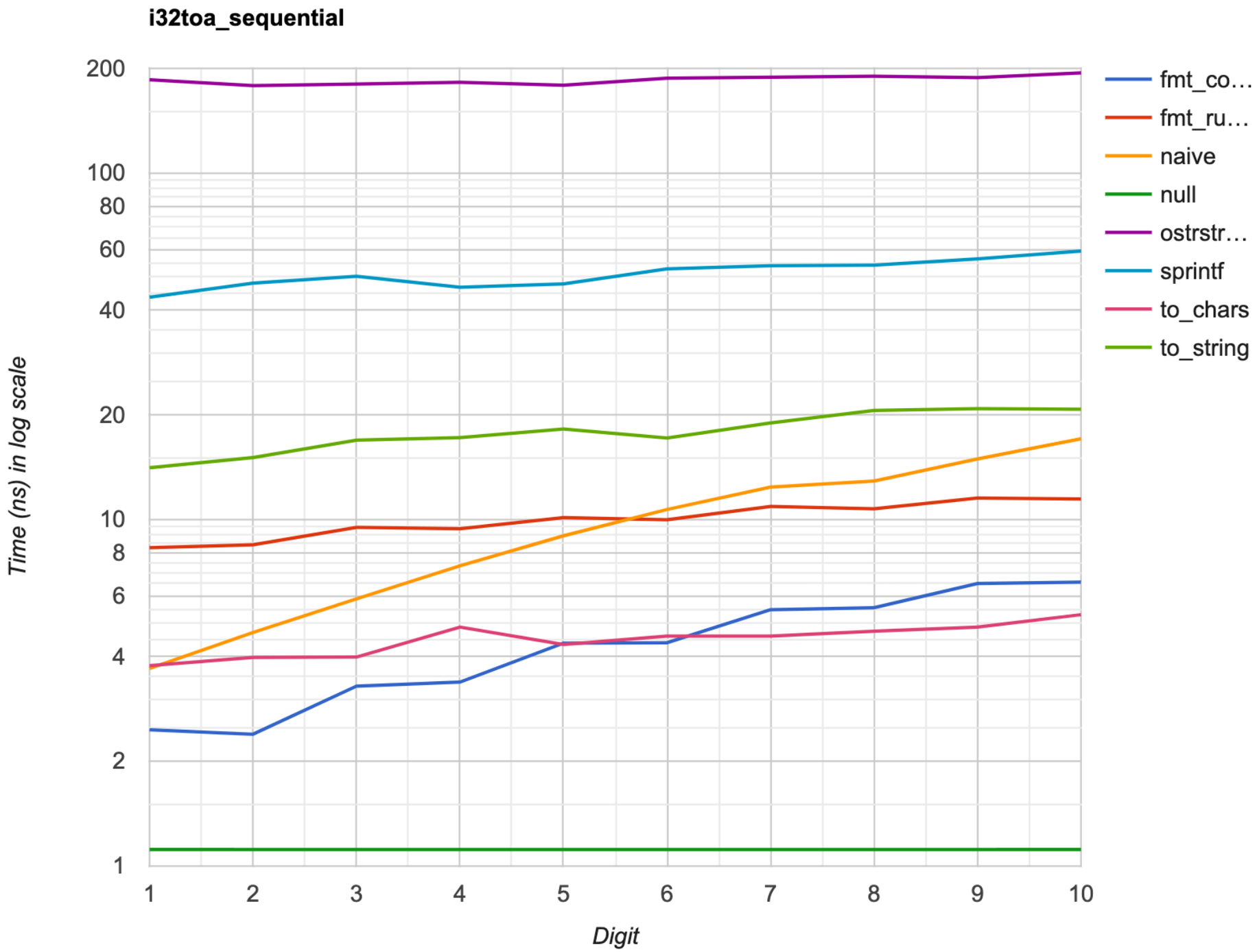
i32toa_random

Function	Time (ns)	Speedup
ostrstream	202.387	1.00x
sprintf	68.246	2.97x
to_string	25.744	7.86x
naive	20.277	9.98x
fmt_runtime	18.952	10.68x
to_chars	15.036	13.46x
fmt_compile	12.283	16.48x
null	1.114	181.68x

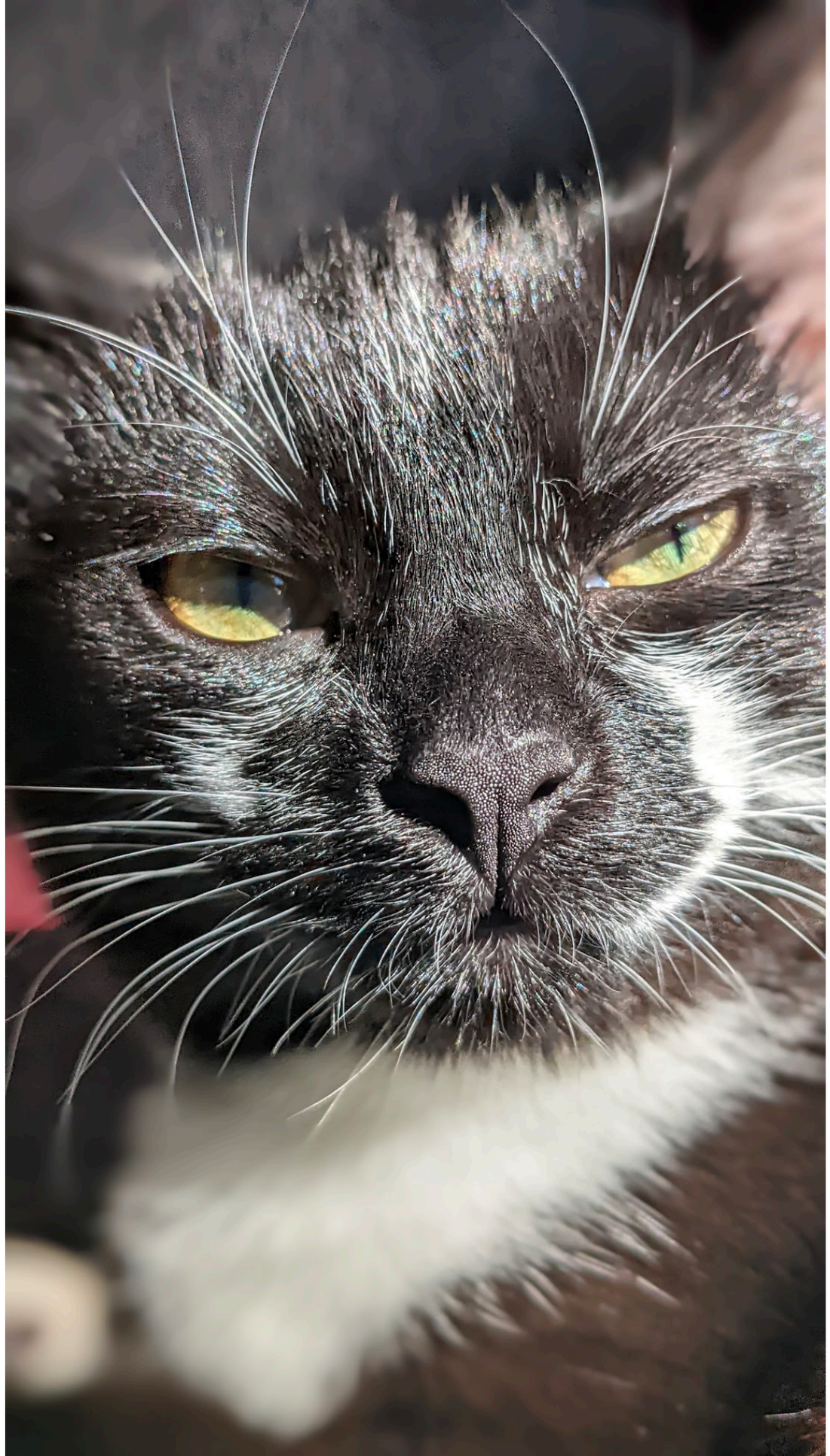
Benchmark

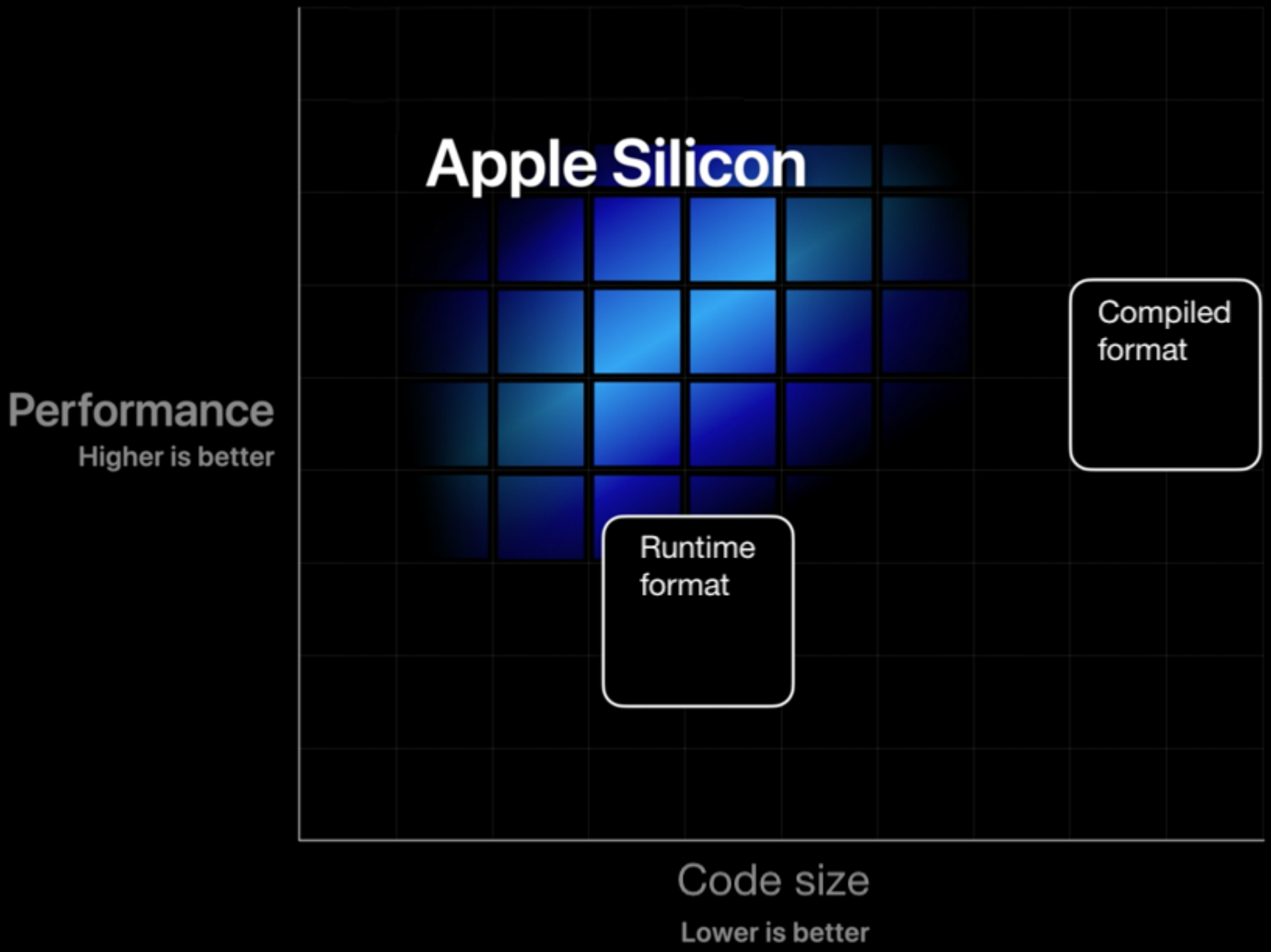

i32toa_sequential
ostrstream
sprintf
to_string
fmt_runtime
naive
to_chars
fmt_compile
null
0
50
100
150
200
Time (ns)

Benchmark


i32toa_sequential
200
100
80
60
40
20
10
8
6
4
2
1
Time (ns) in log scale
1
2
3
4
5
6
7
8
9
10
Digit
fmt_co…
fmt_ru…
naive
null
ostrstr…
sprintf
to_chars
to_string

Hmm, so what's the catch?




Apple Silicon
Compiled format
Runtime format
Performance
Higher is better
Code size
Lower is better

Compile-time formatting

Compile-time formatting

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

constexpr auto compile_time_itoa(int n) {
  auto result = std::array<char, 12>();
  using namespace fmt::literals;
  auto end = fmt::format_to(
    result.data(), "{}"_cf, n);
  *end = '\0';
  return result;
}

constexpr auto answer = compile_time_itoa(42);
```

compiles to

```
answer:
        .byte   52
        .byte   50
        .byte   0
        ...
```

How it works

Constexpr all the things! Caveat: static data in constexpr functions

```
constexpr auto count_digits(uint32_t n) -> int {
  // An optimization by Kendall Willets from https://bit.ly/3uOIQrB.
  constexpr uint64_t table[] = {...};
  auto inc = table[__builtin_clz(n | 1) ^ 31];
  return static_cast<int>((n + inc) >> 32);
}
```

```
static:

error: 'table' declared 'static' in 'constexpr' function only available with '-std=c+
+2b' or '-std=gnu++2b'
    6 |   static constexpr uint64_t table[] = {
      |                             ^~~~~
```

No static:

```
        sub     rsp, 144
        mov     esi, OFFSET FLAT:.LC0
        mov     edx, eax
        lea     rdi, [rsp-120]
        mov     ecx, 32
        or      edx, 1
        rep movsq
```

Life finds a way

Workarounds for static arrays in constexpr functions:

- separate non-constexpr logic via if constexpr
- store data outside of functions (can be problematic for complex build configs: optional header only, static, .so)
- use string literals 

```
constexpr uint32_t fractional_part_rounding_thresholds(int index) {
  // These are stored in a string literal because we cannot have
  // static arrays in constexpr functions and non-static ones are
  // poorly optimized.
  return U"\x9999999a\x828f5c29\x80418938\x80068db9\x8000a7c6"
         U"\x800010c7\x800001ae\x8000002b"[index];
}
```

Type erasure

Type erasure

- Formatting arguments:

```
template <typename... T>
std::string format(format_string<T...> fmt,
                   T&&... args);
```

- Output iterator:

```
template <typename OutputIt, typename... T>
OutputIt format_to(OutputIt out,
                   format_string<T...> fmt,
                   T&&... args);
```

iostreams

```
std::string format_answer() {
  std::ostringstream ss;
  ss << "The answer is " << 42 << ".";
  return ss.str();
}
```

How much code does it generate?

iostreams

```
.LC0:
        .string "basic_string::_M_create"
.LC1:
        .string "The answer is "
.LC2:
        .string "."
.LC3:
        .string "basic_string::_M_replace"
format_answer[abi:cxx11]():
        push    r15
        push    r14
        push    r13
        push    r12
        mov     r12, rdi
        push    rbp
        push    rbx
        sub     rsp, 424
        movq    xmm1, QWORD PTR .LC4[rip]
        lea     rdi, [rsp+144]
        movhps  xmm1, QWORD PTR .LC5[rip]
        movaps  XMMWORD PTR [rsp], xmm1
        call    std::ios_base::ios_base() [base object constructor]
        mov     rbp, QWORD PTR VTT for std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char> >[rip+8]
        xor     eax, eax
        xor     esi, esi
        pxor    xmm0, xmm0
        mov     WORD PTR [rsp+368], ax
        movups  XMMWORD PTR [rsp+376], xmm0
        movups  XMMWORD PTR [rsp+392], xmm0
        mov     rax, QWORD PTR [rbp-24]
        mov     QWORD PTR [rsp+144], OFFSET FLAT:_ZTVSt9basic_iosIcSt11char_traitsIcEE+16
        add     rax, 32
        mov     QWORD PTR [rsp+32], rbp
        mov     QWORD PTR [rsp+360], 0
        lea     rdi, [rsp+rax]
        mov     rax, QWORD PTR VTT for std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char> >[rip+16]
        mov     QWORD PTR [rdi], rax
        call    std::basic_ios<char, std::char_traits<char> >::init(std::basic_streambuf<char, std::char_traits<char> >*)
        movdqa  xmm1, XMMWORD PTR [rsp]
        pxor    xmm0, xmm0
        lea     rdi, [rsp+96]
        mov     QWORD PTR [rsp+144], OFFSET FLAT:vtable for std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char>
>+64
        movaps  XMMWORD PTR [rsp+48], xmm0
        lea     r13, [rsp+128]
        movaps  XMMWORD PTR [rsp+32], xmm1
        movaps  XMMWORD PTR [rsp+64], xmm0
        movaps  XMMWORD PTR [rsp+80], xmm0
        call    std::locale::locale() [complete object constructor]
        lea     rsi, [rsp+40]
        mov     QWORD PTR [rsp+112], r13
        lea     rdi, [rsp+144]
        mov     QWORD PTR [rsp+40], OFFSET FLAT:vtable for std::__cxx11::basic_stringbuf<char, std::char_traits<char>, std::allocator<char> >+16
        mov     DWORD PTR [rsp+104], 16
        mov     QWORD PTR [rsp+120], 0
        mov     BYTE PTR [rsp+128], 0
        call    std::basic_ios<char, std::char_traits<char> >::init(std::basic_streambuf<char, std::char_traits<char> >*)
        mov     edx, 14
        mov     esi, OFFSET FLAT:.LC1
        lea     rdi, [rsp+32]
        call    std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >(std::basic_ostream<char,
std::char_traits<char> >&, char const*, long)
        mov     esi, 42
        lea     rdi, [rsp+32]
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
        mov     rdi, rax
        mov     edx, 1
        mov     esi, OFFSET FLAT:.LC2
        call    std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >(std::basic_ostream<char,
std::char_traits<char> >&, char const*, long)
        mov     rax, QWORD PTR [rsp+80]
        lea     r15, [r12+16]
        mov     QWORD PTR [r12+8], 0
        mov     QWORD PTR [r12], r15
        mov     BYTE PTR [r12+16], 0
        test    rax, rax
        je      .L32
        mov     rdx, QWORD PTR [rsp+64]
        test    rdx, rdx
        je      .L66
        cmp     rax, rdx
        ja      .L66
.L33:
        mov     r14, QWORD PTR [rsp+72]
        mov     rbx, rdx
        sub     rbx, r14
        js      .L79
        cmp     rbx, 15
        ja      .L36
        lea     rcx, [r15+rbx]
        cmp     r14, r15
        je      .L80
        test    rbx, rbx
        je      .L39
        cmp     rbx, 1
        je      .L81
        mov     rdx, rbx
        mov     rsi, r14
        mov     rdi, r15
        mov     QWORD PTR [rsp], rcx
        call    memcpy
        mov     rcx, QWORD PTR [rsp]
.L39:
        mov     QWORD PTR [r12+8], rbx
        mov     BYTE PTR [rcx], 0
.L52:
        movq    xmm0, QWORD PTR .LC4[rip]
        mov     rdi, QWORD PTR [rsp+112]
        mov     QWORD PTR [rsp+144], OFFSET FLAT:vtable for std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char>
>+64
        movhps  xmm0, QWORD PTR .LC6[rip]
        movaps  XMMWORD PTR [rsp+32], xmm0
        cmp     rdi, r13
        je      .L54
        mov     rax, QWORD PTR [rsp+128]
        lea     rsi, [rax+1]
        call    operator delete(void*, unsigned long)
.L54:
        mov     QWORD PTR [rsp+40], OFFSET FLAT:vtable for std::basic_streambuf<char, std::char_traits<char> >+16
        lea     rdi, [rsp+96]
        call    std::locale::~locale() [complete object destructor]
        mov     rax, QWORD PTR [rbp-24]
        mov     QWORD PTR [rsp+32], rbp
        lea     rdi, [rsp+144]
        mov     rcx, QWORD PTR VTT for std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char> >[rip+16]
        mov     QWORD PTR [rsp+32+rax], rcx
        mov     QWORD PTR [rsp+144], OFFSET FLAT:_ZTVSt9basic_iosIcSt11char_traitsIcEE+16
        call    std::ios_base::~ios_base() [base object destructor]
        add     rsp, 424
        mov     rax, r12
        pop     rbx
        pop     rbp
        pop     r12
        pop     r13
        pop     r14
        pop     r15
        ret
.L66:
        mov     rdx, rax
        jmp     .L33
.L80:
        test    rbx, rbx
        je      .L39
        cmp     r15, rdx
        jb      .L41
        cmp     rbx, 1
        je      .L39
        mov     rdx, rbx
        mov     rsi, r15
        mov     rdi, r15
        mov     QWORD PTR [rsp], rcx
        call    memmove
        mov     rcx, QWORD PTR [rsp]
        jmp     .L39
.L81:
        movzx   eax, BYTE PTR [r14]
        mov     BYTE PTR [r12+16], al
        jmp     .L39
.L32:
        lea     rsi, [rsp+112]
        mov     rdi, r12
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_assign(std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > const&)
        jmp     .L52
.L36:
        cmp     rbx, 29
        ja      .L82
        mov     QWORD PTR [rsp], 30
        mov     edi, 31
.L49:
        call    operator new(unsigned long)
        mov     rcx, rax
        test    r14, r14
        je      .L50
        mov     rdx, rbx
        mov     rsi, r14
        mov     rdi, rax
        call    memcpy
        mov     rcx, rax
.L50:
        mov     rdi, QWORD PTR [r12]
        cmp     r15, rdi
        je      .L51
        mov     rax, QWORD PTR [r12+16]
        mov     QWORD PTR [rsp+24], rcx
        lea     rsi, [rax+1]
        call    operator delete(void*, unsigned long)
        mov     rcx, QWORD PTR [rsp+24]
.L51:
        mov     rax, QWORD PTR [rsp]
        mov     QWORD PTR [r12], rcx
        add     rcx, rbx
        mov     QWORD PTR [r12+16], rax
        jmp     .L39
.L82:
        mov     rdi, rbx
        mov     QWORD PTR [rsp], rbx
        add     rdi, 1
        jns     .L49
        call    std::__throw_bad_alloc()
.L41:
        cmp     rbx, 1
        je      .L83
        mov     rsi, rcx
        mov     rdx, rbx
        mov     rdi, r15
        mov     QWORD PTR [rsp], rcx
        call    memcpy
        mov     rcx, QWORD PTR [rsp]
        jmp     .L39
.L83:
        movzx   eax, BYTE PTR [rcx]
        mov     BYTE PTR [r12+16], al
        jmp     .L39
.L79:
        mov     edi, OFFSET FLAT:.LC3
        call    std::__throw_length_error(char const*)
        mov     rbp, rax
        jmp     .L55
        mov     rbp, rax
        jmp     .L31
        mov     rbp, rax
        jmp     .L57
        mov     rbx, rax
        jmp     .L29
format_answer[abi:cxx11]() [clone .cold]:
        lea     rdi, [rsp+32]
        call    std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char> >::~basic_ostringstream() [complete
object destructor]
        mov     rdi, rbp
        call    _Unwind_Resume
.L29:
        mov     QWORD PTR [rsp+40], OFFSET FLAT:vtable for std::__cxx11::basic_stringbuf<char, std::char_traits<char>, std::allocator<char>
>+16
        mov     rdi, QWORD PTR [rsp+112]
        cmp     rdi, r13
        je      .L30
        mov     rax, QWORD PTR [rsp+128]
        lea     rsi, [rax+1]
        call    operator delete(void*, unsigned long)
.L30:
        mov     QWORD PTR [rsp+40], OFFSET FLAT:vtable for std::basic_streambuf<char, std::char_traits<char> >+16
        lea     rdi, [rsp+96]
        call    std::locale::~locale() [complete object destructor]
        mov     rax, QWORD PTR [rbp-24]
        mov     QWORD PTR [rsp+32], rbp
        mov     rbp, rbx
        mov     rcx, QWORD PTR VTT for std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char> >[rip+16]
        mov     QWORD PTR [rsp+32+rax], rcx
.L31:
        lea     rdi, [rsp+144]
        mov     QWORD PTR [rsp+144], OFFSET FLAT:_ZTVSt9basic_iosIcSt11char_traitsIcEE+16
        call    std::ios_base::~ios_base() [base object destructor]
        mov     rdi, rbp
        call    _Unwind_Resume
.LC4:
        .quad   vtable for std::__cxx11::basic_ostringstream<char, std::char_traits<char>, std::allocator<char> >+24
.LC5:
        .quad   vtable for std::basic_streambuf<char, std::char_traits<char> >+16
.LC6:
        .quad   vtable for std::__cxx11::basic_stringbuf<char, std::char_traits<char>, std::allocator<char> >+16
```

Code bloat

Naive implementation:

```
template <typename... T>
std::string format(format_string<T...> fmt,
                   T&&... args) {
  // Parse fmt, get arguments from args and
  // format them.
}

format("int: {}", 42);       // -> format<int>
format("double: {}", 42.0); // -> format<double>
format("string: {}", "42"); // -> format<const char (&)[3]>
```

Type erasure

```
std::string vformat(std::string_view fmt,
                    format_args args);

// Force inline.
template <typename... T>
std::string format(format_string<T...> fmt,
                   T&&... args) {
  return vformat(fmt,
                 make_format_args(args...));
}

format("int: {}", 42);        // -> vformat
format("double: {}", 42.0); // -> vformat
format("string: {}", "42"); // -> vformat
```

Type erasure

```
std::string format_answer() {
  return format("The answer is {}.", 42);
}
```

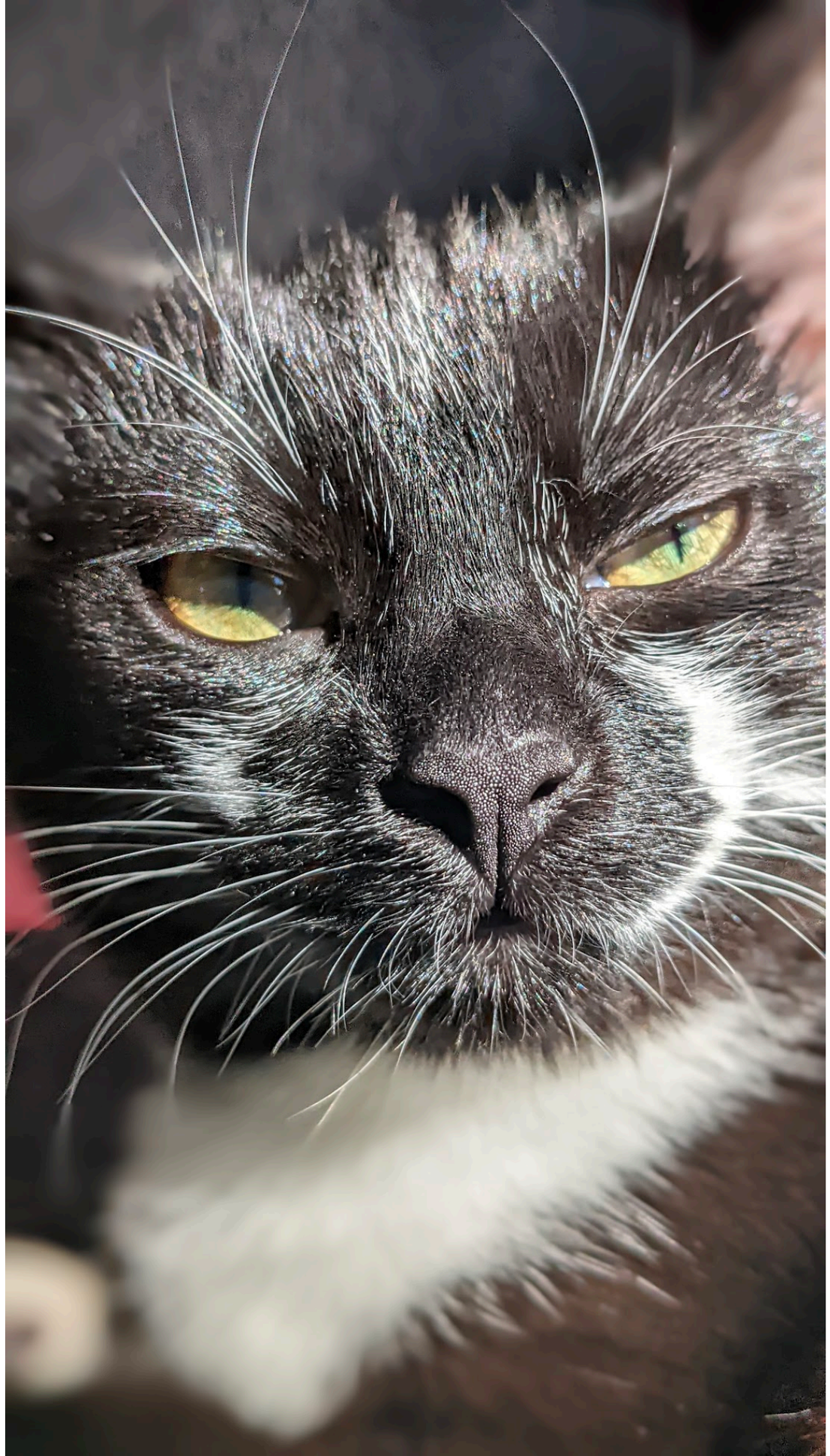
compiles to

```
.LC0:
        .string "The answer is {}."
format_answer[abi:cxx11]():
        push    r12
        mov     ecx, 1
        mov     esi, OFFSET FLAT:.LC0
        mov     r12, rdi
        mov     edx, 17
        sub     rsp, 16
        mov     r8, rsp
        mov     DWORD PTR [rsp], 42
        call    fmt::v8::vformat[abi:cxx11](fmt::v8::basic_string_view<char>,
fmt::v8::basic_format_args<...>)
        add     rsp, 16
        mov     rax, r12
        pop     r12
        ret
```

How it works

```
using format_arg  = std::variant<int, double, ...>; // Conceptually
using format_args = std::span<format_arg>;

template <size_t N>
struct format_args_store {
  std::array<format_arg, N> data;
  operator format_args() { return {data.data(), N}; }
};

template <typename... T>
auto make_format_args(const T&... args) {
  return format_args_store<sizeof...(T)>{{args...}};
}

std::string vformat(fmt::string_view fmt, format_args args);

template <typename... T>
std::string format(format_string<T...> fmt, T&&... args) {
  return vformat(fmt, make_format_args(args...));
}
```

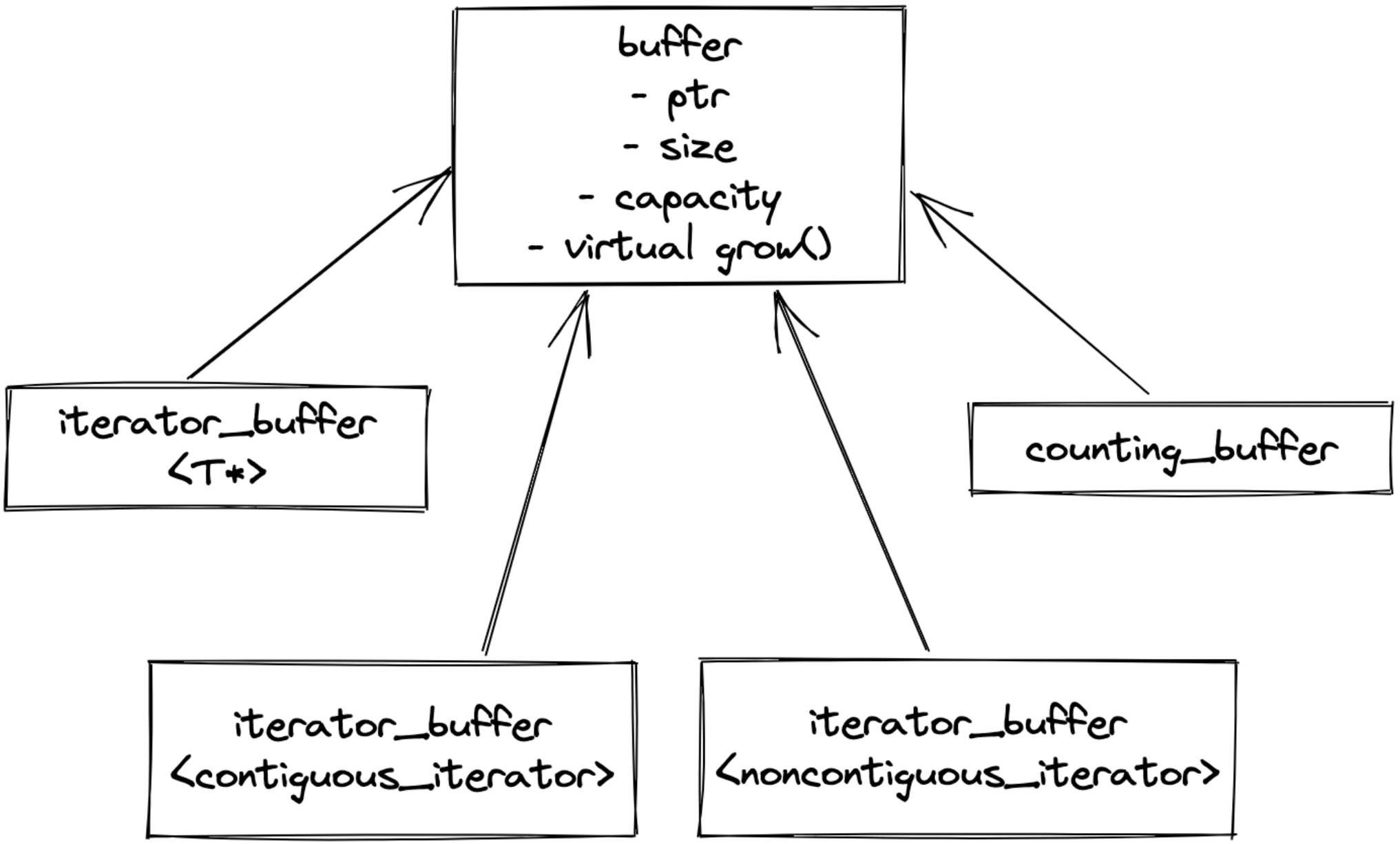
Hmm, this looks familiar.



Prior art

```
int printf(const char* fmt, ...);

int vprintf(const char* fmt, va_list args);
```

Iterator type erasure



Iterator type erasure

```
template <typename OutputIt>
auto vformat_to(OutputIt out,
                std::string_view fmt,
                format_args args) {
  // iterator -> buffer
  auto&& buf = get_buffer(out);

  // Perform formatting.
  vformat_to(buf, fmt, args);

  // buffer -> iterator
  return get_iterator(buf);
}
```

Does it matter?

~5x reduction in per-call binary code size in the Folly logging library:

https://github.com/facebook/folly/commit/da41ae504804260fa371595daed100d03440670e

Floating-point formatting

0.30000000000000004

- Formatting defaults are suboptimal in C & C++ (lose precision):

```
std::cout << (0.1 + 0.2) << " == " << 0.3 << " is "
          << std::boolalpha << (0.1 + 0.2 == 0.3) << "\n";
```

prints `"0.3 == 0.3 is false"`

- The issue is not specific to C++ but some languages have better defaults: https://0.30000000000000004.com/

Desired properties

Steele & White (1990):

1. No information loss
2. Shortest output
3. Correct rounding
4. ~~Left to right generation~~ - irrelevant with buffering

(public domain)

No information loss

Round trip guarantee: parsing the output gives the original value.

Most libraries/functions lack this property unless you explicitly specify big enough precision: C stdio, C++ iostreams & `to_string` (until C++23), Python's `str.format` until version 3, etc.

```c
double a = 1.0 / 3.0;
char buf[20];
sprintf(buf, "%g", a);
double b = atof(buf);
assert(a == b);

// fails:
// a == 0.33333333333333333
// b == 0.333333
```

```c
double a = 1.0 / 3.0;

auto s = 🦄(a);
double b = atof(s.c_str());
assert(a == b);

// succeeds:
// a == 0.33333333333333333
// b == 0.33333333333333333
```

How much is enough?

- "17 digits ought to be enough for anyone"
  — some famous person (paraphrased)

- *In-and-out conversions*,
  David W. Matula (1968):

Conversions from base B round-trip through base v when $B^n < v^{m-1}$, where n is the number of base B digits, and m is the number of base v digits.

$$\lceil \log_{10}(2^{53}) + 1 \rceil = 17$$

Photo of a random famous person
(public domain)

Shortest output

The number of digits in the output is as small as possible.

It is easy to satisfy the round-trip property by printing unnecessary "garbage" digits:

```
sprintf("%.17g", 0.1);
```

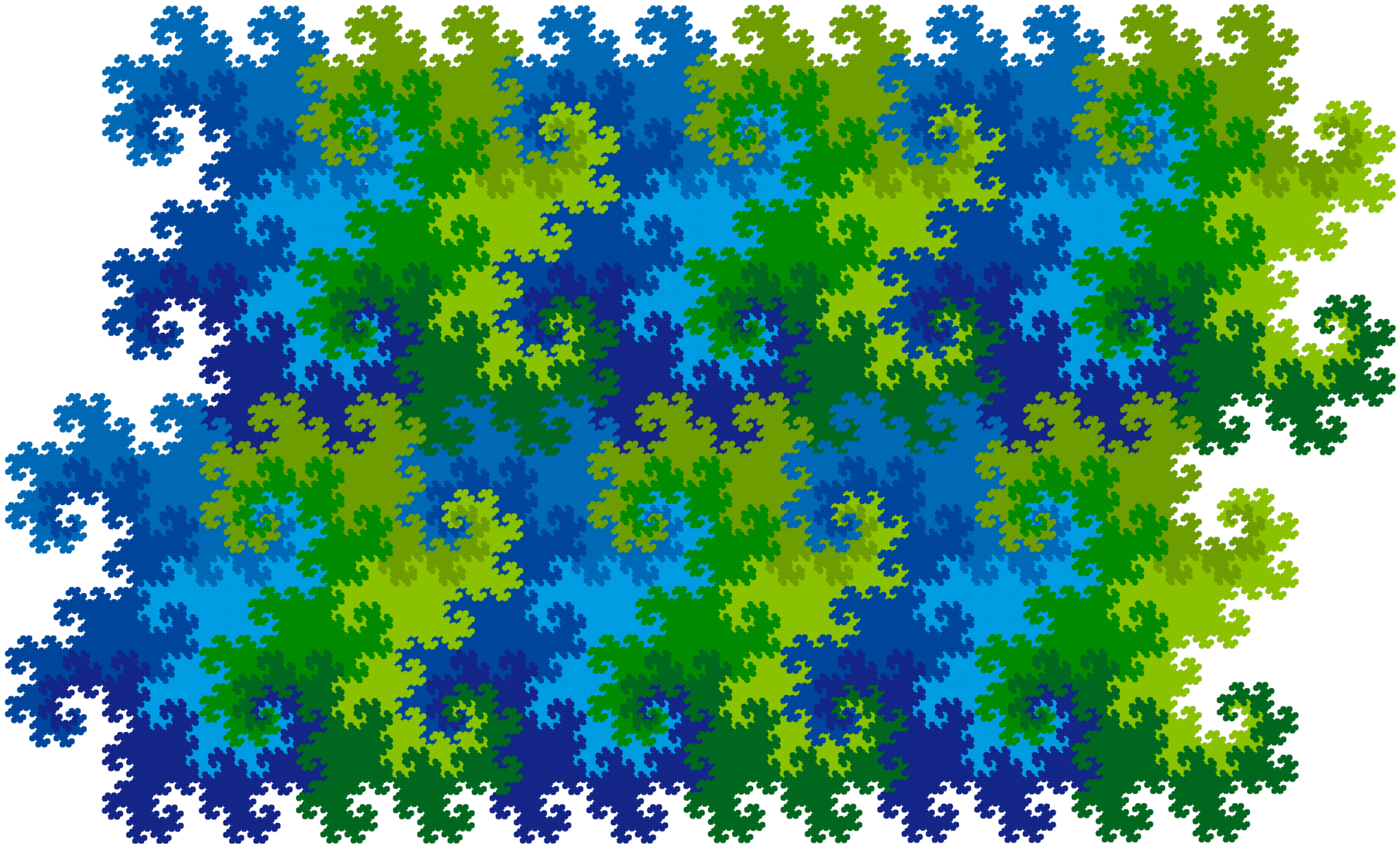
prints `"0.10000000000000001"`

```
🦄(0.1);
```

prints `"0.1"`

Correct rounding

- Usually round to nearest, ties to even (IEEE 754 default).
- Most implementations have this, but MSVC/CRT had a long history of bugs (both from and to decimal):
  - https://www.exploringbinary.com/incorrect-round-trip-conversions-in-visual-c-plus-plus/
  - https://www.exploringbinary.com/incorrectly-rounded-conversions-in-visual-c-plus-plus/
  - Had to disable some floating-point tests on MSVC due to broken rounding in `printf` and iostreams



(GNU Free Documentation License)

Here be dragons: FP formatting algorithms

Algorithms

- Dragonbox (2020) by Junekey Jeon: https://github.com/jk-jeon/dragonbox - {fmt}'s default.
  - Based on Schubfach (2020) by Raffaello Giulietti.
- Ryu (2018) by Ulf Adams.
- Grisu family of algorithms (2010+) by Florian Loitsch. Used in Google's double conversion.
- Dragon family of algorithms (70s-80s) by Steele and White. Used as a fallback for exotic FP in {fmt}.

printf

- Full exponent range for IEEE double: 10^{-324} - 10^{308}
- May require multiple precision arithmetic
- glibc pulls in a GNU multiple precision library for `printf`:

Overhead	Command	Shared Object	Symbol
57.96%	a.out	libc-2.17.so	[.] __printf_fp
15.28%	a.out	libc-2.17.so	[.] __mpn_mul_1
15.19%	a.out	libc-2.17.so	[.] __mpn_divrem
5.79%	a.out	libc-2.17.so	[.] hack_digit.13638
5.79%	a.out	libc-2.17.so	[.] vfprintf

printf

- Full exponent range for IEEE double: 10^{-324} - 10^{308}
- May require multiple precision arithmetic
- glibc pulls in a GNU multiple precision library for `printf`:

Overhead	Command	Shared Object	Symbol
57.96%	a.out	libc-2.17.so	[.] __printf_fp
15.28%	a.out	libc-2.17.so	[.] __mpn_mul_1
15.19%	a.out	libc-2.17.so	[.] __mpn_divrem
5.79%	a.out	libc-2.17.so	[.] hack_digit.13638
5.79%	a.out	libc-2.17.so	[.] vfprintf

FP in {fmt}

- The default is shortest decimal representation with round-trip guarantees and correct rounding 🦄
- Rich formatting mini-language
- Supports iterators, size computation, buffer preallocation
- High performance
- Zero dynamic memory allocations possible
- Locale control
- Portability: requires only a subset of C++11

Round-trip

```cpp
#include <fmt/core.h>

int main() {
  double a = 1.0 / 3.0;
  auto s = fmt::format("{}", a);
  double b = atof(s.c_str());
  assert(a == b);

  // succeeds:
  // a == 0.3333333333333333
  // b == 0.3333333333333333
}
```

Locale

Locale-independent by default:

```
fmt::print("{}", 4.2); // prints 4.2
```

Locale-specific formatting is available via a separate format specifier:

```
std::locale::global(
  std::locale(“uk_UA.UTF-8”));
fmt::print("{:L}", 4.2); // prints 4,2
```

Zero allocations

- Dynamic memory allocations can be completely avoided & in particular the default will never allocate for float/double.

- No allocation & no need to specify buffer size:

```
auto buf = fmt::memory_buffer();
fmt::format_to(buf, "{}", 1.2345);
// std::string_view(buf.data(), buf.size())
// contains "1.2345"
```

- Single exact allocation & no extra copy (unlike `to_chars`):

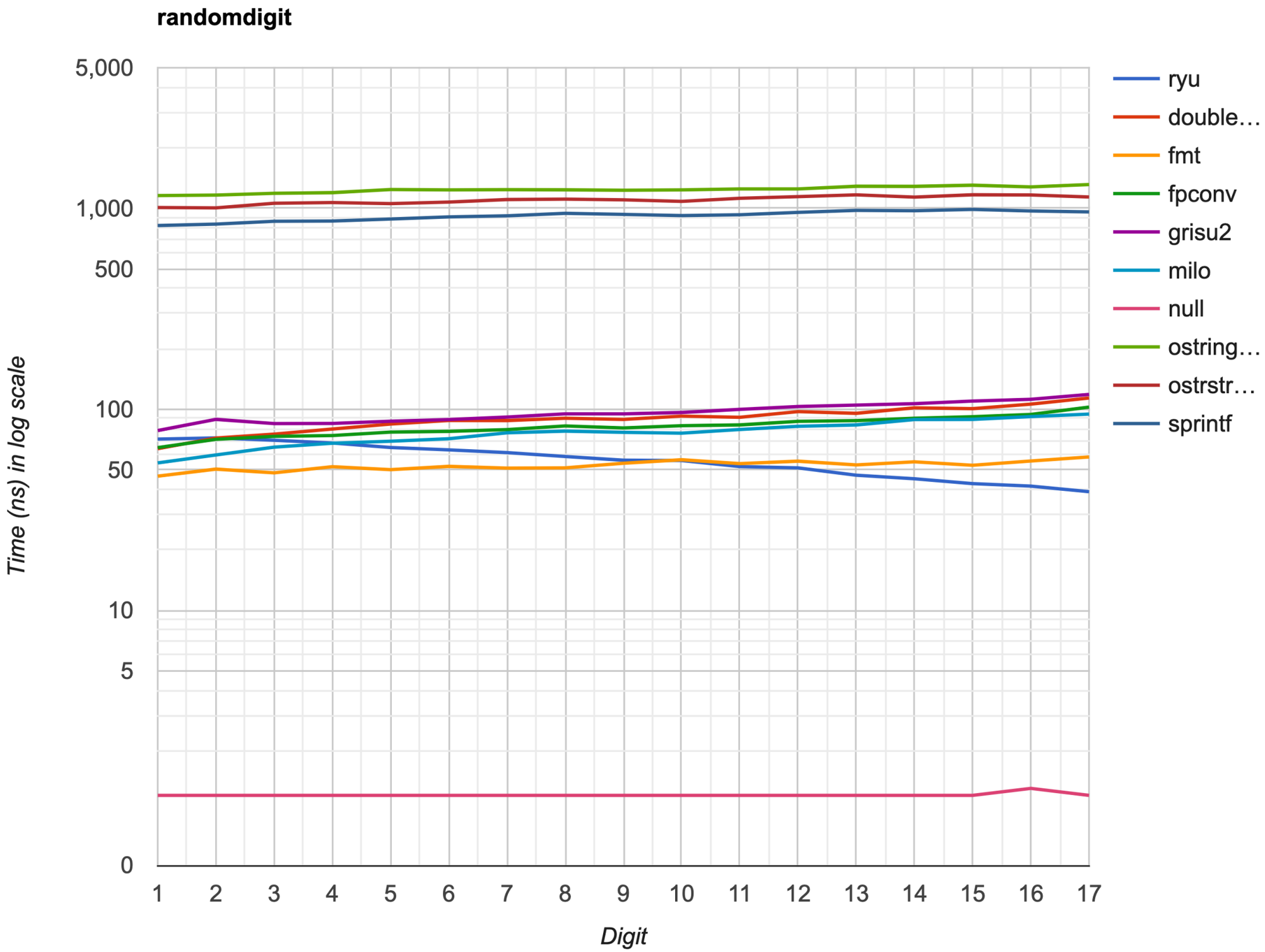
```
auto s = std::string();
fmt::format_to(std::back_inserter(s), "{}", 1.2345);
```

Benchmark

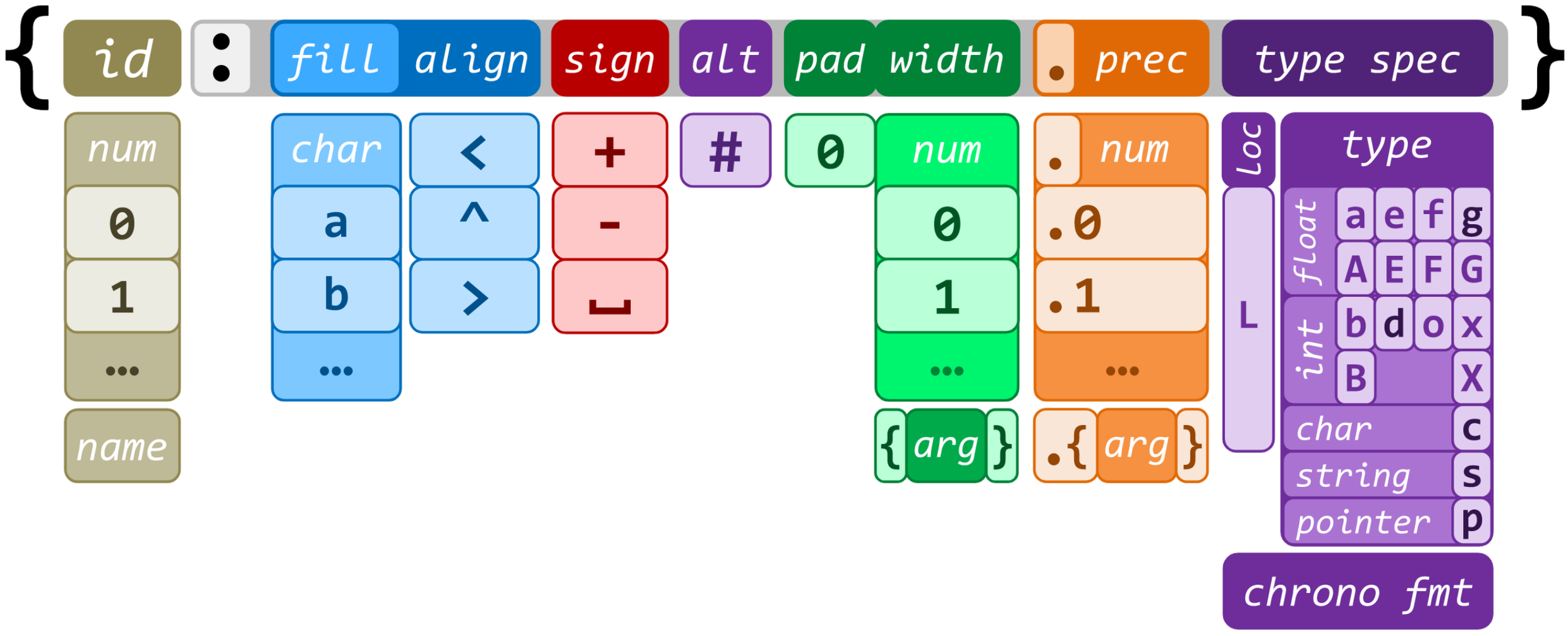
Function	Time (ns)	Speedup
ostringstream	1,239.747	1.00x
ostrstream	1,100.371	1.13x
sprintf	919.418	1.35x
grisu2	96.859	12.80x
doubleconv	89.965	13.78x
fpconv	82.365	15.05x
milo	76.700	16.16x
ryu	56.318	22.01x
fmt	52.571	23.58x
null	1.206	1,028.08x

Roundtrip precision: https://github.com/fmtlib/dtoa-benchmark (based on miloyip/dtoa-benchmark)

Benchmark


randomdigit
5,000
1,000
500
100
50
10
5
0
Time (ns) in log scale
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
Digit
ryu
double…
fmt
fpconv
grisu2
milo
null
ostring…
ostrstr…
sprintf

Cheat sheets



https://hackingcpp.com/cpp/libs/fmt.html

Want to become a C++ meta programmer?

We are hiring: https://www.metacareers.com/

Questions?