A modern formatting library for C++

Victor Zverovich (victor.zverovich@gmail.com)

“Formatting is something everybody uses but nobody has put much effort to learn.”

– Reviewer 5

Formatting in C++

<table>
<tr><td>stdio</td><td><code>printf("%4d\n", x);</code></td></tr>
<tr><td>iostream</td><td><code>std::cout << std::setw(4) << x << std::endl;</code></td></tr>
<tr><td>Boost Format</td><td><code>std::cout << boost::format("%|4|\n") % x;</code></td></tr>
<tr><td>Fast Format</td><td><code>ff::fmtln(std::cout, "{0,4}\n", x);</code></td></tr>
<tr><td>Folly Format</td><td><code>std::cout << folly::format("{:4}\n", x);</code></td></tr>
</table>

... and a million other ways

The past: stdio

stdio

- C legacy but still widely used in C++
- Pros:
  - Very small client compile time and binary code
  - Available everywhere as a part of the C standard library
  - Good performance
- Cons:
  - Almost no extensibility
  - Lack of type and memory safety

Extensibility

No standard way to extend printf but there is a GNU extension

```
class Widget;

int print_widget(
  FILE *stream, const struct printf_info *info, const void *const *args) {
  const Widget *w = *((const Widget **) (args[0]));
  // Format widget.
}

int print_widget_arginfo(
  const struct printf_info *info, size_t n, int *argtypes) {
  /* We always take exactly one argument and this is a pointer to the
     structure.. */
  if (n > 0)
    argtypes[0] = PA_POINTER;
  return 1;
}

register_printf_function('W', print_widget, print_widget_arginfo);
```

Not type safe, limited number of specifiers (uppercase letters).

Type safety

Type/format specifier mismatch

```
  printf("%2s\n", x);
```

`-Wformat` to the rescue:

```
  warning: format specifies type 'char *' but the argument has
type 'int' [-Wformat]
  printf("%2s\n", x);
          ~~~     ^
          %2d
```

Only works for literal format strings

Strings can be dynamic esp. with localization

Memory safety

`size` chars should be enough for everyone:

```
size_t size =
  ceil(log10(numeric_limits<int>::max())) + 1;
vector<char> buf(size);
int result = sprintf(buf.data(), "%2d", x);
```

Let's check:

```
printf("%d %d", result + 1, size);
```

Solution: `snprintf`

Cannot grow buffer automatically

Fun with specifiers

Did you notice an error in the previous slide?

Fun with specifiers

Did you notice an error in the previous slide?

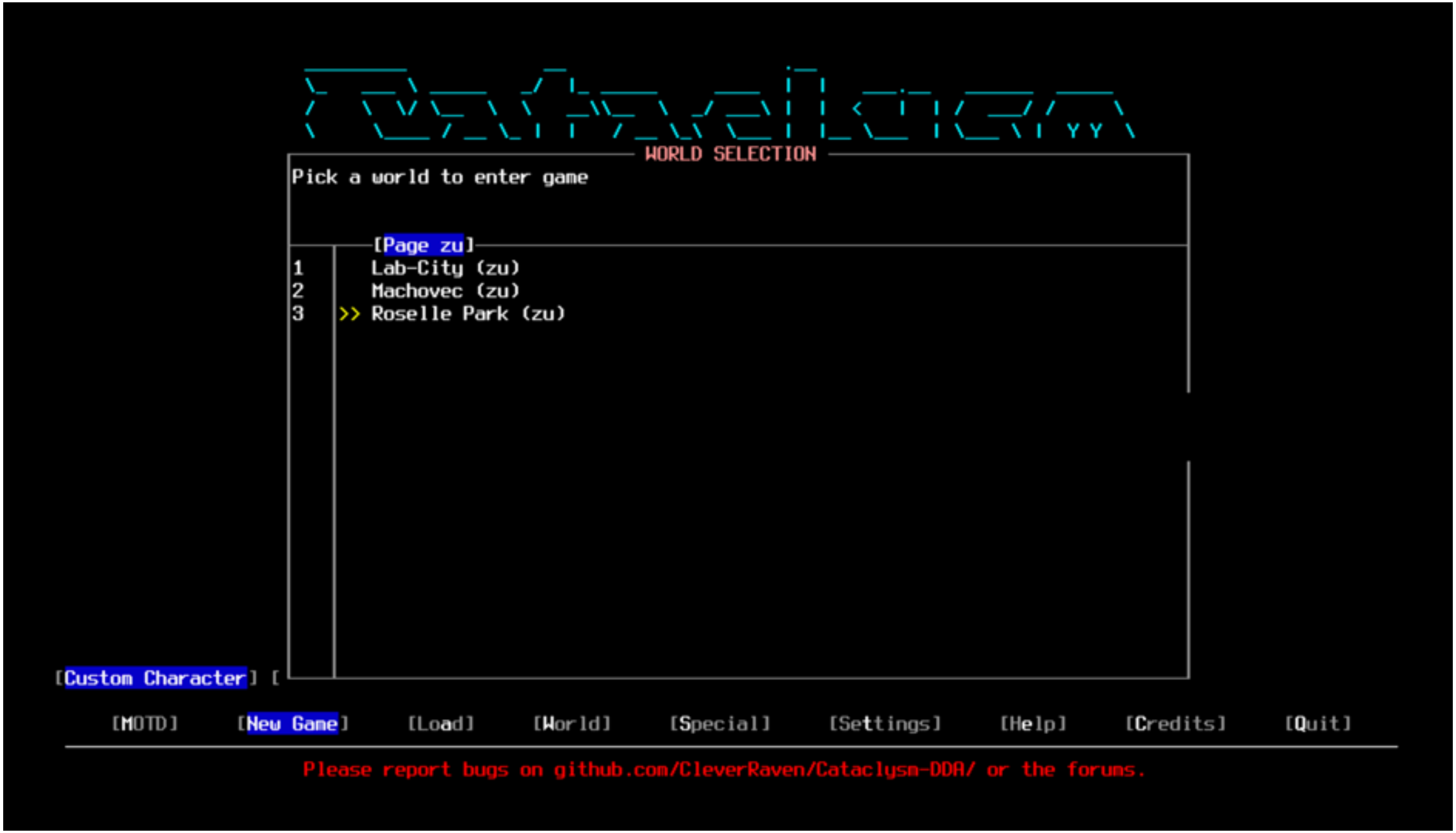
```
  size_t size = ...
  printf("%d %d", result + 1, size);
```

`%d` is not a valid format specifier for **`size_t`**.

```
  warning: format specifies type 'int' but the argument has type
'size_t' (aka 'unsigned long') [-Wformat]
    printf("%d %d", result + 1, size);
               ~~                ^~~~
               %lu
```

But `%lu` is not the correct specifier for **`size_t`** either (compiler lies).

The correct one is `%zu`, but...

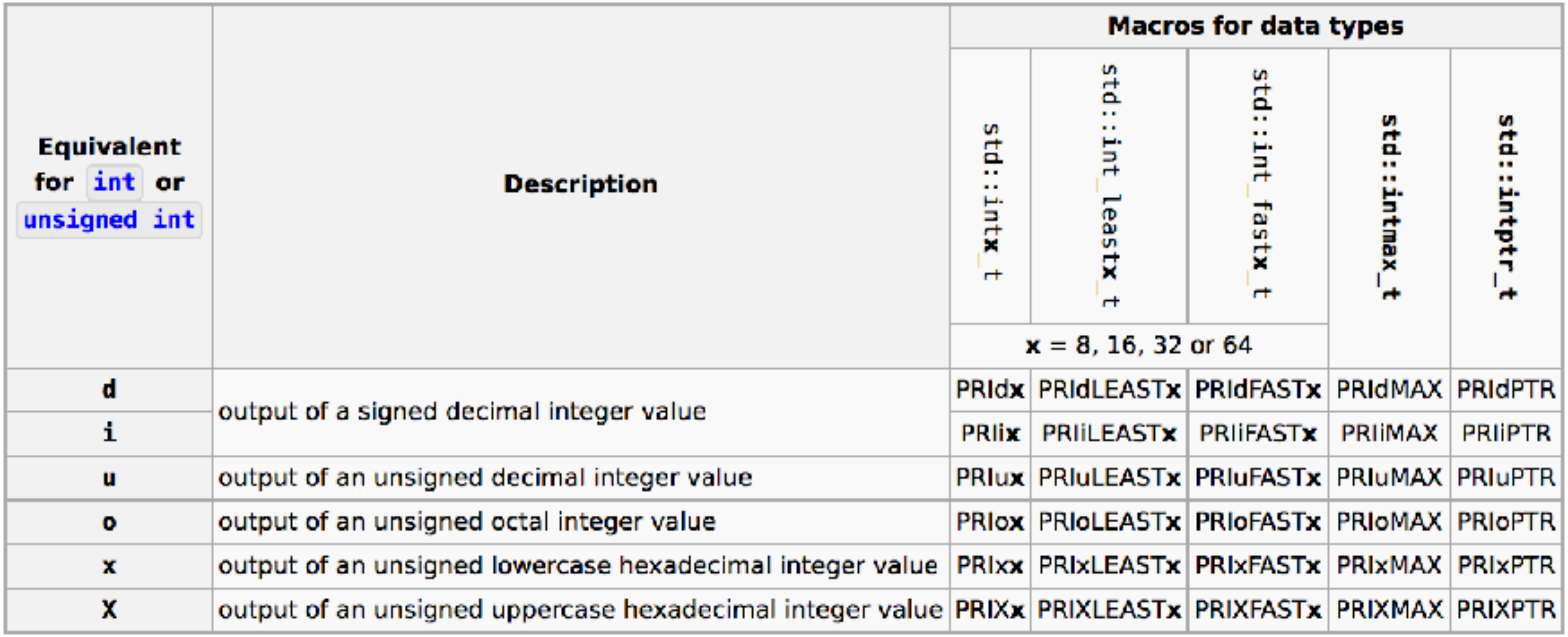


```
                                    WORLD SELECTION
Pick a world to enter game


        [Page zu]
1       Lab-City (zu)
2       Machovec (zu)
3    >> Roselle Park (zu)

[Custom Character] [

  [MOTD]   [New Game]   [Load]   [World]   [Special]   [Settings]   [Help]   [Credits]   [Quit]

        Please report bugs on github.com/CleverRaven/Cataclysm-DDA/ or the forums.
```

2016: Use printf, they said. It's portable, they said.

More specifiers

What about other types?



Equivalent for `int` or `unsigned int`	Description	Macros for data types				
		std::int**x**_t	std::int_least**x**_t	std::int_fast**x**_t	**std::intmax_t**	**std::intptr_t**
		x = 8, 16, 32 or 64				
d	output of a signed decimal integer value	PRIdx	PRIdLEASTx	PRIdFASTx	PRIdMAX	PRIdPTR
i		PRIix	PRIiLEASTx	PRIiFASTx	PRIiMAX	PRIiPTR
u	output of an unsigned decimal integer value	PRIux	PRIuLEASTx	PRIuFASTx	PRIuMAX	PRIuPTR
o	output of an unsigned octal integer value	PRIox	PRIoLEASTx	PRIoFASTx	PRIoMAX	PRIoPTR
x	output of an unsigned lowercase hexadecimal integer value	PRIxx	PRIxLEASTx	PRIxFASTx	PRIxMAX	PRIxPTR
X	output of an unsigned uppercase hexadecimal integer value	PRIXx	PRIXLEASTx	PRIXFASTx	PRIXMAX	PRIXPTR

http://en.cppreference.com/w/cpp/types/integer

And this is just for fixed-width integer types!

Why pass type information in the format string manually, if the compiler knows the types?

VARARGS
HISTORY.COM
HD
imgflip.com

varargs

- Non-inlinable
- Require saving a bunch of registers on x86-64

```
int mysprintf(char *buffer, const char *format, ...) {
  va_list args;
  va_start(args, format);
  int result = vsprintf(
    buffer, format, args);
  va_end(args);
  return result;
}
```

```
mysprintf(char*, char
const*, ...):
        subq    $216, %rsp
        testb   %al, %al
        movq    %rdx, 48(%rsp)
        movq    %rcx, 56(%rsp)
        movq    %r8, 64(%rsp)
        movq    %r9, 72(%rsp)
        je      .L9
        movaps  %xmm0, 80(%rsp)
        movaps  %xmm1, 96(%rsp)
        movaps  %xmm2, 112(%rsp)
        movaps  %xmm3, 128(%rsp)
        movaps  %xmm4, 144(%rsp)
        movaps  %xmm5, 160(%rsp)
        movaps  %xmm6, 176(%rsp)
        movaps  %xmm7, 192(%rsp)
.L9:
        leaq    224(%rsp), %rax
        leaq    8(%rsp), %rdx
        movq    %rax, 16(%rsp)
        leaq    32(%rsp), %rax
        movl    $16, 8(%rsp)
        movl    $48, 12(%rsp)
        movq    %rax, 24(%rsp)
        call    vsprintf
        addq    $216, %rsp
        ret
```

varargs

```
char buf[16];
for (int i = 0; i < 10000000; ++i) {
  sprintf(buf, "%d", i);
}
```

```
Overhead  Command  Shared Object         Symbol
  36.96%  a.out    libc-2.17.so          [.] vfprintf
  14.78%  a.out    libc-2.17.so          [.] _itoa_word
  10.73%  a.out    libc-2.17.so          [.] _IO_default_xsputn
   7.49%  a.out    libc-2.17.so          [.] _IO_old_init
   6.16%  a.out    libc-2.17.so          [.] _IO_str_init_static_internal
   5.64%  a.out    libc-2.17.so          [.] __strchrnul
   5.52%  a.out    libc-2.17.so          [.] _IO_vsprintf
   3.20%  a.out    libc-2.17.so          [.] _IO_no_init
   2.53%  a.out    libc-2.17.so          [.] sprintf
```

Not a big deal, but uncalled for (and more profound if formatting is optimized).

varargs

No random access, so need to setup extra arrays when dealing with positional arguments.

```
for (int i = 0; i < 10000000; ++i) {
  sprintf(buf, "%d", i);
}

Time: 0m0.738s

for (int i = 0; i < 10000000; ++i) {
  sprintf(buf, "%1$d", i);
}

Time: 0m1.361s
```

Lessons learned

Varargs are a poor choice for modern formatting API:

1. Manual type management
2. Don't play well with positional arguments due to lack of random access
3. Require saving a bunch of registers on x86-64
4. Non-inlinable causing with (3) small but noticeable (few %) overhead on simple in-memory formatting

We can do better with variadic templates!

The present: iostreams

iostreams

- The standard C++ formatting & I/O library
- Pros:
  - Available everywhere
  - Type and memory safe
  - Extensible
- Cons:
  - Verbose
  - Performance isn't great on some implementations
  - Doesn't play well with localization

Chevron hell

stdio:

```
  printf("0x%04x\n", 0x42);
```

iostream:

```
  std::cout << "0x" << std::hex << std::setfill('0')
            << std::setw(4) << 0x42 << '\n';
```

Which is more readable?

C++11 finally gave in to format strings for time:

```
std::cout << std::put_time(&tm, "%c %Z");
```

Translation

stdio - whole message is available for translation:

```
printf(translate("String `%s' has %d characters\n"),
       string, length(string));
```

iostream - message mixed with arguments:

```
cout << "String `" << string << "' has "
     << length(string) << " characters\n";
```

Other issues:

- Reordering arguments
- Access to arguments for pluralization

Manipulators

Let's print a number in hexadecimal:

```
cout << hex << setw(8) << setfill('0') << 42 << endl;
```

and now print something else:

```
cout << 42 << endl;
```

Oops, this still prints "2a" because we forgot to switch the stream back to decimal.

Some flags are sticky, some are not. ¯_(ツ)_/¯

Solution: `boost::io::ios_flags_saver`

Locales

Let's write some JSON:

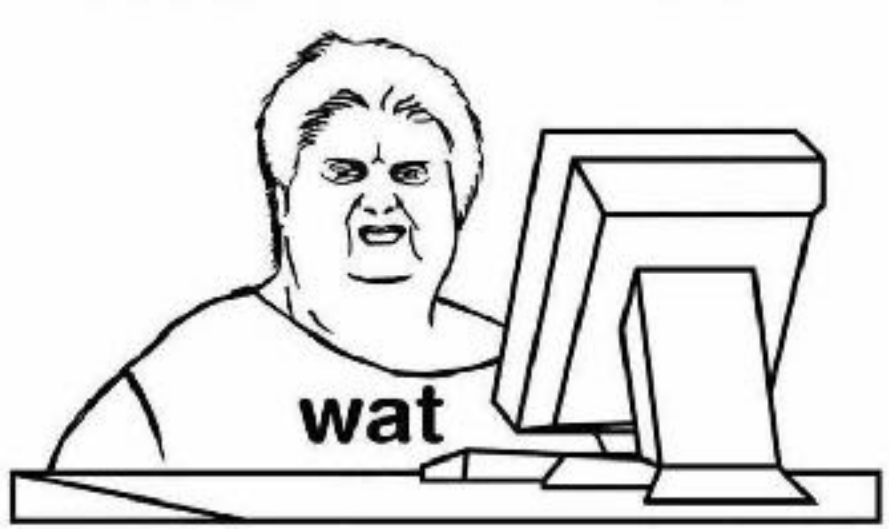
```
std::ofstream ofs("test.json");
ofs << "{'value': " << 4.2 << "}";
```

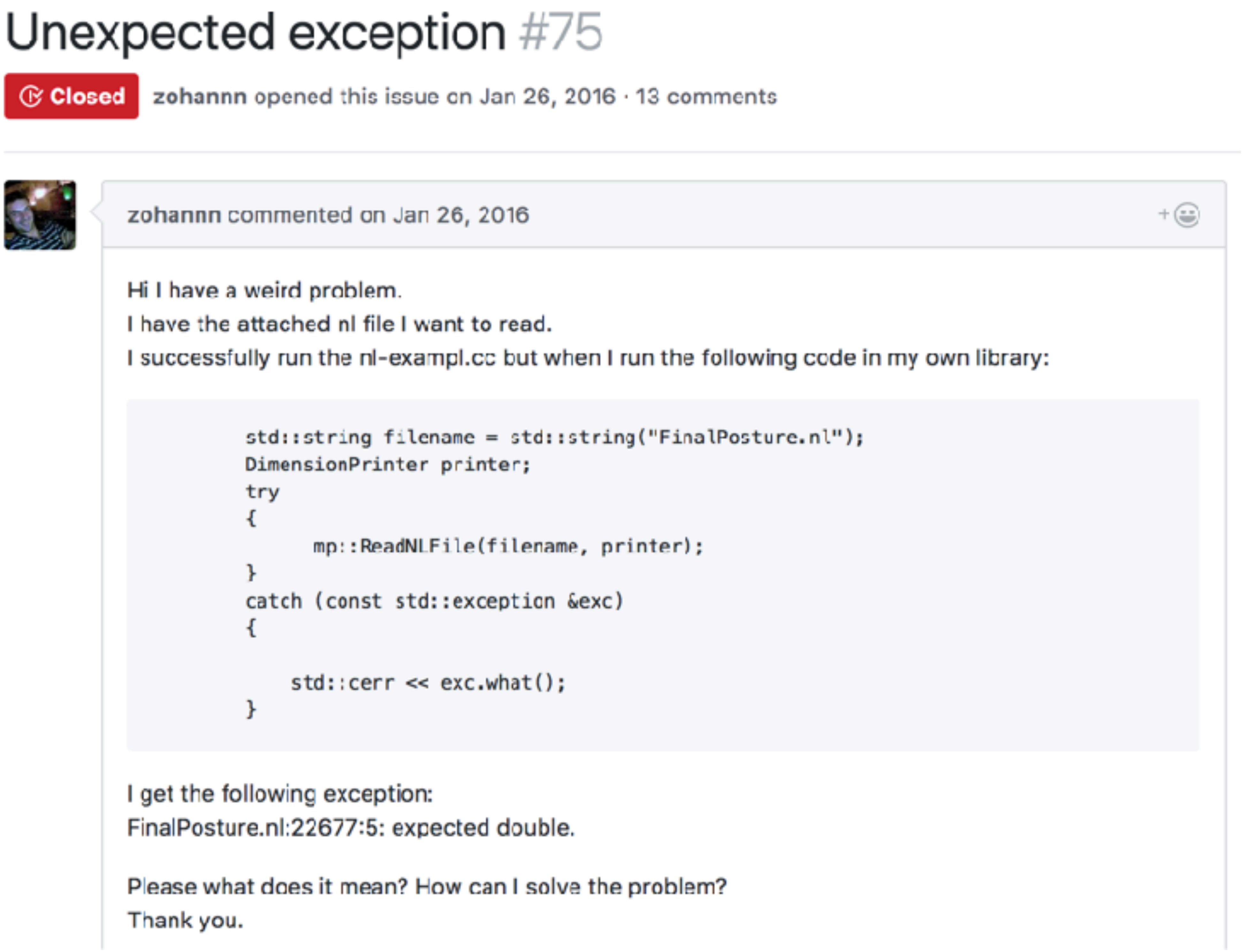
works fine:

```
{'value': 4.2}
```

until someone sets the global (!) locale to ru_RU.UTF-8:

```
{'value': 4,2}
```





Unexpected exception #75

Closed zohannn opened this issue on Jan 26, 2016 · 13 comments

zohannn commented on Jan 26, 2016

Hi I have a weird problem.
I have the attached nl file I want to read.
I successfully run the nl-exampl.cc but when I run the following code in my own library:

```
          std::string filename = std::string("FinalPosture.nl");
          DimensionPrinter printer;
          try
          {
                mp::ReadNLFile(filename, printer);
          }
          catch (const std::exception &exc)
          {

              std::cerr << exc.what();
          }
```

I get the following exception:
FinalPosture.nl:22677:5: expected double.

Please what does it mean? How can I solve the problem?
Thank you.

And then you get bug reports like this

Alt history: Boost Format, Fast Format

Boost Format

- Established C++ formatting library
- Pros:
  - Widely available as part of Boost
  - Compatible with older compilers (doesn't need C++11)
  - Good interoperability with iostreams
- Cons:
  - Performance & code bloat issues
  - Long compile times
  - Complicated format string syntax
  - Unorthodox use of operator%

Boost Format Syntax

Simple style:

```
cout << boost::format("%1% %2% %3% %2% %1% \n")
          % "11" % "22" % "333";
// prints "11 22 333 22 11 "
```

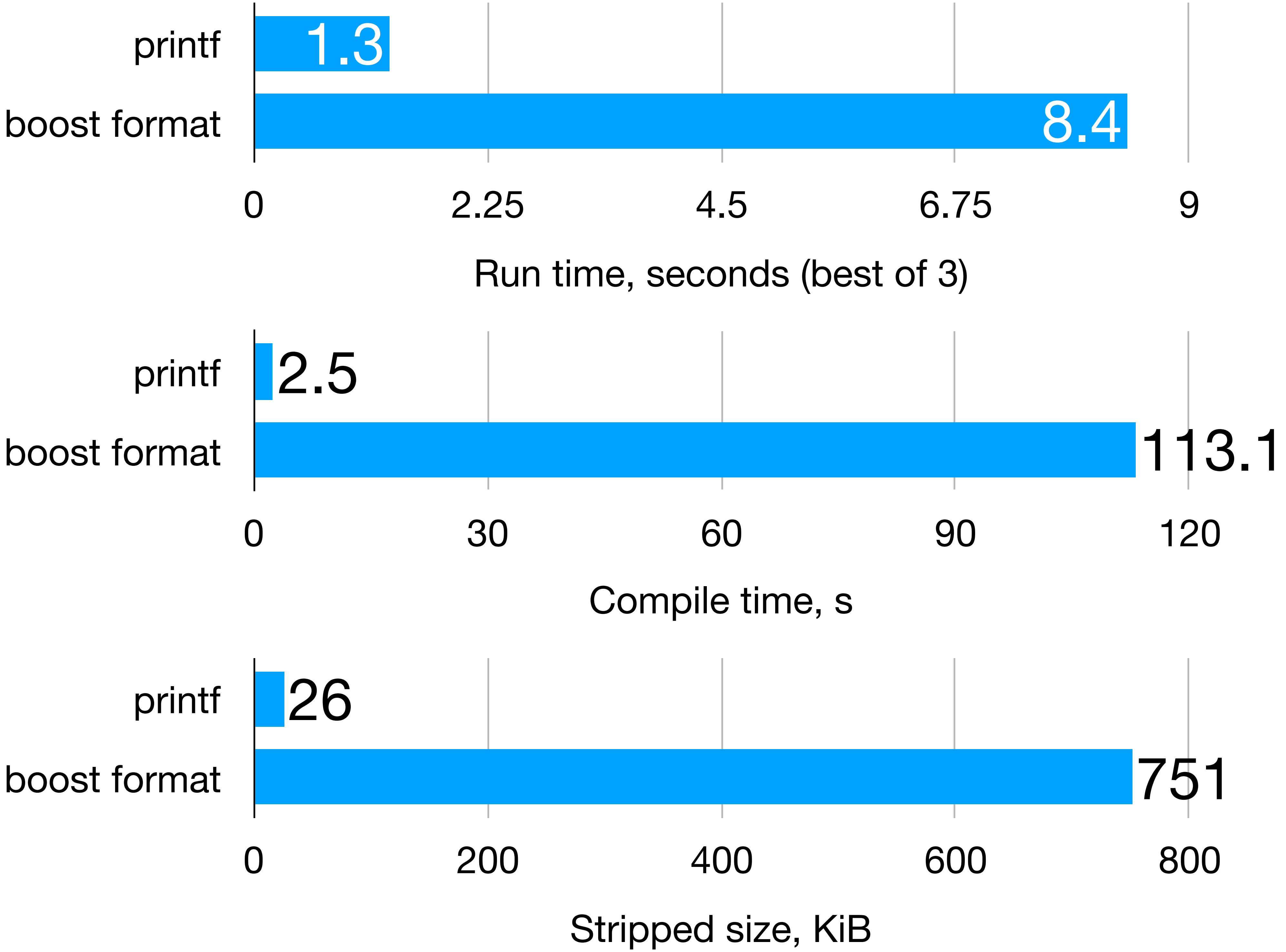
printf-like style

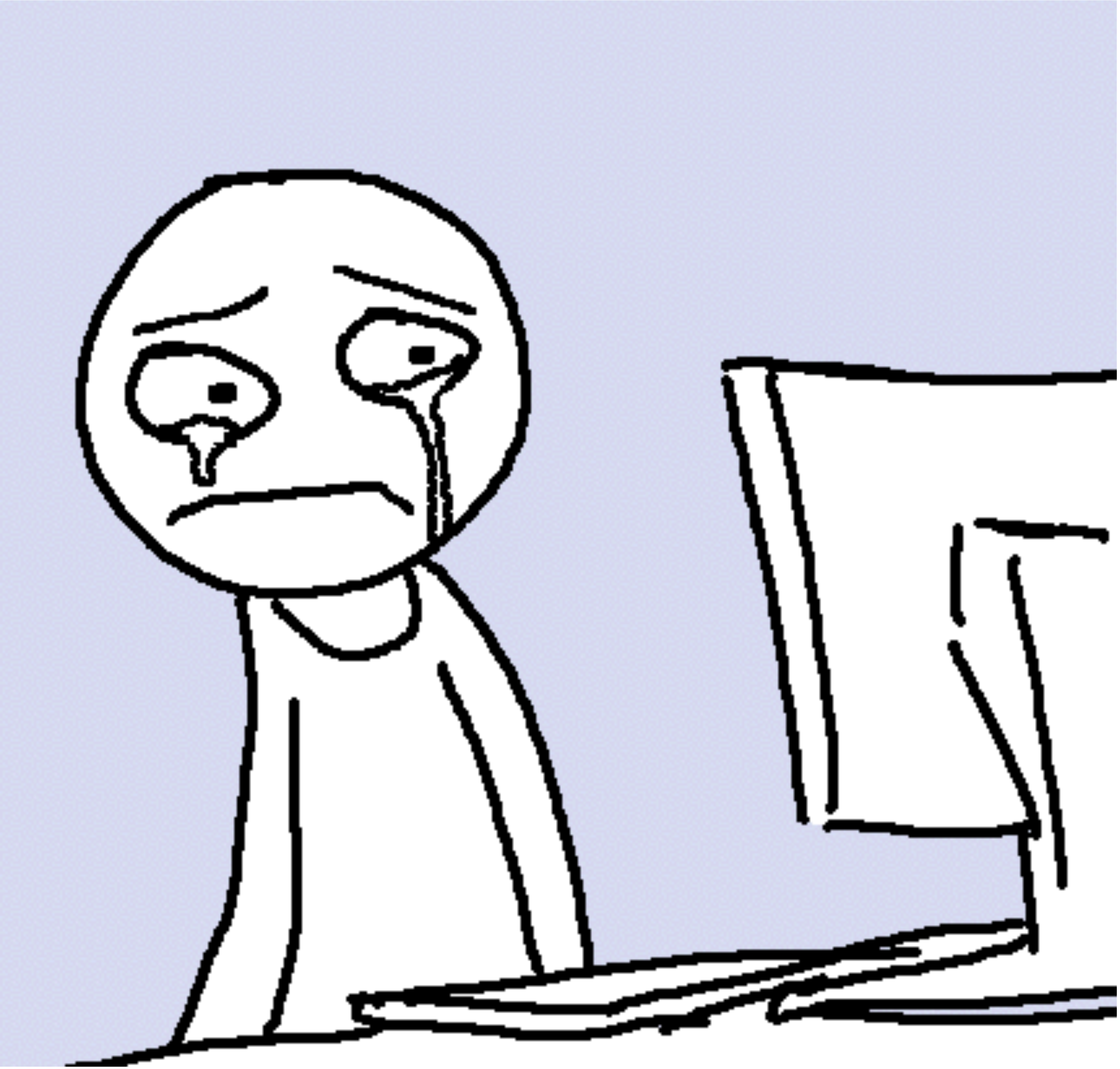
```
cout << boost::format("(x,y) = (%1$+5d,%2$+5d)\n")
          % -23 % 35;
// prints "(x,y) = (  -23,  +35)"
```

Expressive, but complicated syntax

Not fully printf compatible

Boost Format





Fast Format

- One of the most well-known C++ formatting libraries
- Pros:
  - Reportedly fast
  - Type and memory safe
  - Extensible
- Cons:
  - Limited formatting syntax
  - Heavy dependencies (STLSoft library)

Fast Format

Three features that have no hope of being accommodated within the current design are:

- Leading zeros (or any other non-space padding)
- Octal/hexadecimal encoding
- Runtime width/alignment specification

Matthew Wilson, An Introduction to Fast Format, Overload Journal #89.

Fast Format

Solution:

```
ff::fmtln(std::cout, "{0}",
          pan::integer(10, 8, pan::fmt::fullHex));
```

Now how it is better than

```
std::cout << std::hex << std::setw(8) << 10;
```

Non-sticky but even more verbose than iostreams.

The (proposed) future: P0645Rx Text Formatting

Motivation

Alternative to (s)printf

Safe

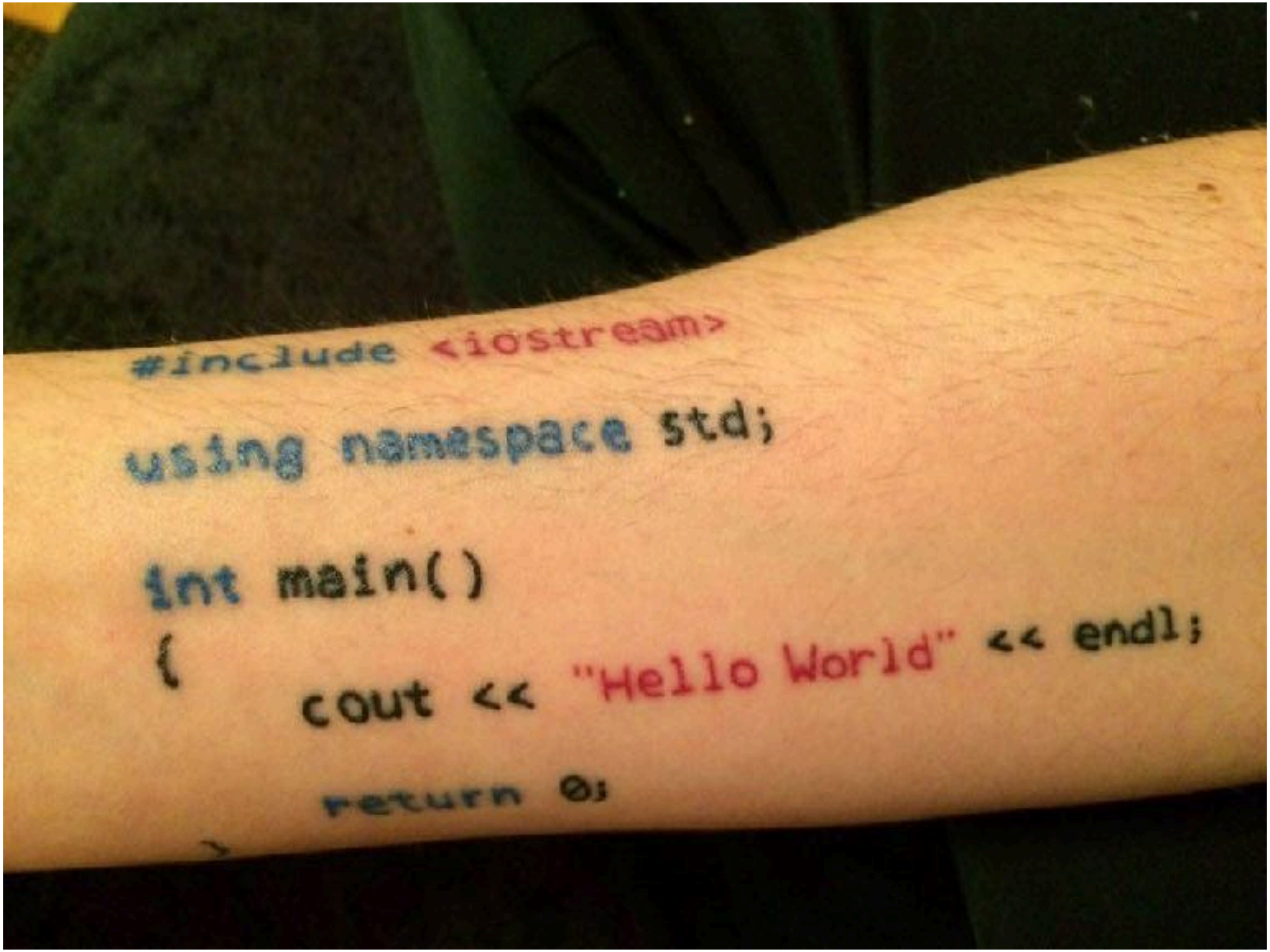
Extensible

Fast

Interoperable with IOStreams

Small code size and reasonable compile times

Expressive syntax and locale control



```
#include <iostream>

using namespace std;

int main()
{
     cout << "Hello World" << endl;
     return 0;
}
```

Not an iostream replacement

Syntax

Python-like

```
string message = format("The answer is {}.", 42);
```

Positional arguments

```
format("I'd rather be {1} than {0}.", "right", "happy");
```

Expressive

```
format("{:^20}", "centered"); // "      centered      "

format("{:05.2f}", 1.234);    // "01.23"
```

Syntax

Simple grammar

```
format-spec ::=  [[fill] align] [sign] ['#'] ['0']
                 [width] ['.' precision] [type]
fill        ::=  <a character other than '{' or '}'>
align       ::=  '<' | '>' | '=' | '^'
sign        ::=  '+' | '-' | ' '
width       ::=  integer | '{' arg-id '}'
precision   ::=  integer | '{' arg-id '}'
type        ::=  int-type | 'a' | 'A' | 'c' | 'e' | 'E' | ... | 's'
int-type    ::=  'b' | 'B' | 'd' | 'o' | 'x' | 'X'
```

Easy to parse

Named arguments (not in P0645Rx)

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

Why new syntax?

Legacy-free:

```
printf("%d", my_int);
printf("%lld", my_long_long);
printf("%" PRIu64, my_int64);
```

↓

```
format("{}", my_int);
format("{}", my_long_long);
format("{}", my_int64);
```

Semantical: conveys formatting, not type info, e.g. `"d"` means "decimal formatting" not "decimal `int"`

BYOG: bring your own grammar

Extensibility

User-defined format specs

```
replacement-field ::= '{' [arg-id] [':' format-spec] '}'
```

Extension API

```
void format_value(buffer& buf, const tm& tm, context& ctx) {
  // Parse format spec and format tm.
}
```

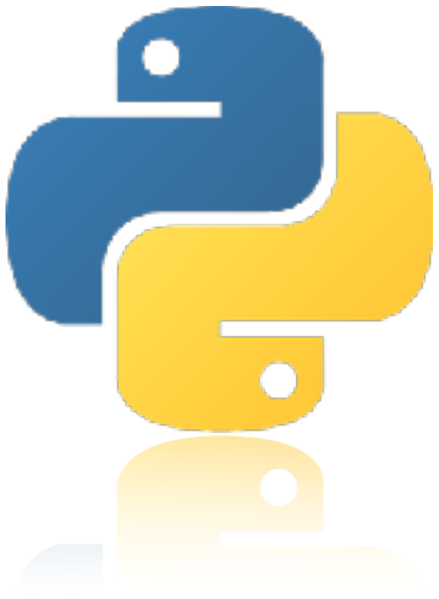
Usage

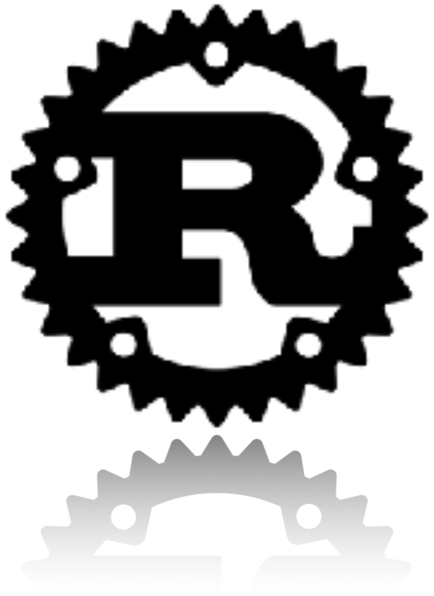
```
time_t t = time(nullptr);
string date = format("The date is {0:%Y-%m-%d}.", *localtime(&t));
```

Falls back on ostream `operator<<`.

Why this syntax?

Proven to work





Has popular C++ implementations:

- fmt - basis of this proposal
- Facebook Folly

Safety

Type safe - variadic templates instead of varargs

```
template <typename... Args>
std::string format(string_view format_str,
                   const Args&... args);
```

Memory safe - automatic buffer management

```
template <typename... Args>
void format_to(buffer& buf, string_view format_str,
               const Args&... args);
```

Memory management

```
template <typename T>
class basic_buffer {
public:
  std::size_t size() const;
  const T *data() const;
  virtual locale locale() const;
  void resize(std::size_t new_size);
protected:
  virtual void grow(size_type n) = 0;
};
```

- Contiguous memory range
- Efficient access, virtual call only to grow
- Can be limited (including fixed) size and report an error on growth
- Has an associated locale

Going deeper

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

template <typename... Args>
inline std::string format(string_view format_str,
                          const Args&... args) {
  return vformat(format_str, make_args(args...));
}
```

`arg_store` class - argument list storage (simplified):

```
template <typename... Args>
arg_store<Args...> make_args(const Args&... args);
```

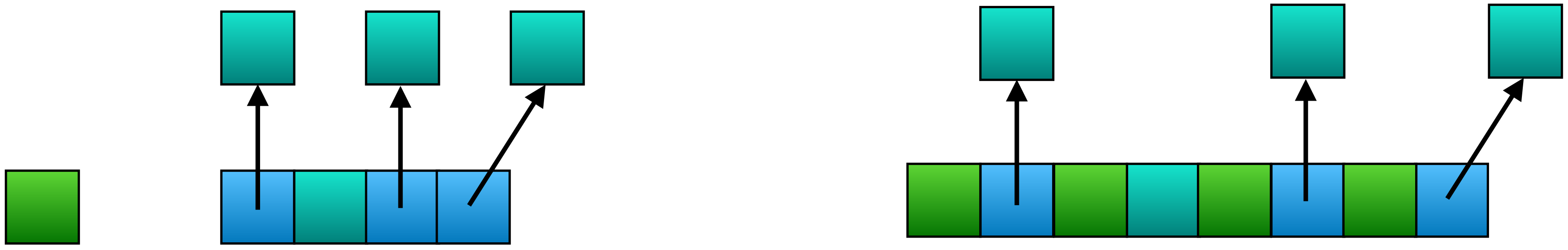
`args` class - argument list view, implicitly convertible from arg_store (simplified):

```
template <typename... Args>
args(const arg_store<Args...>& store);
```

Handling arguments

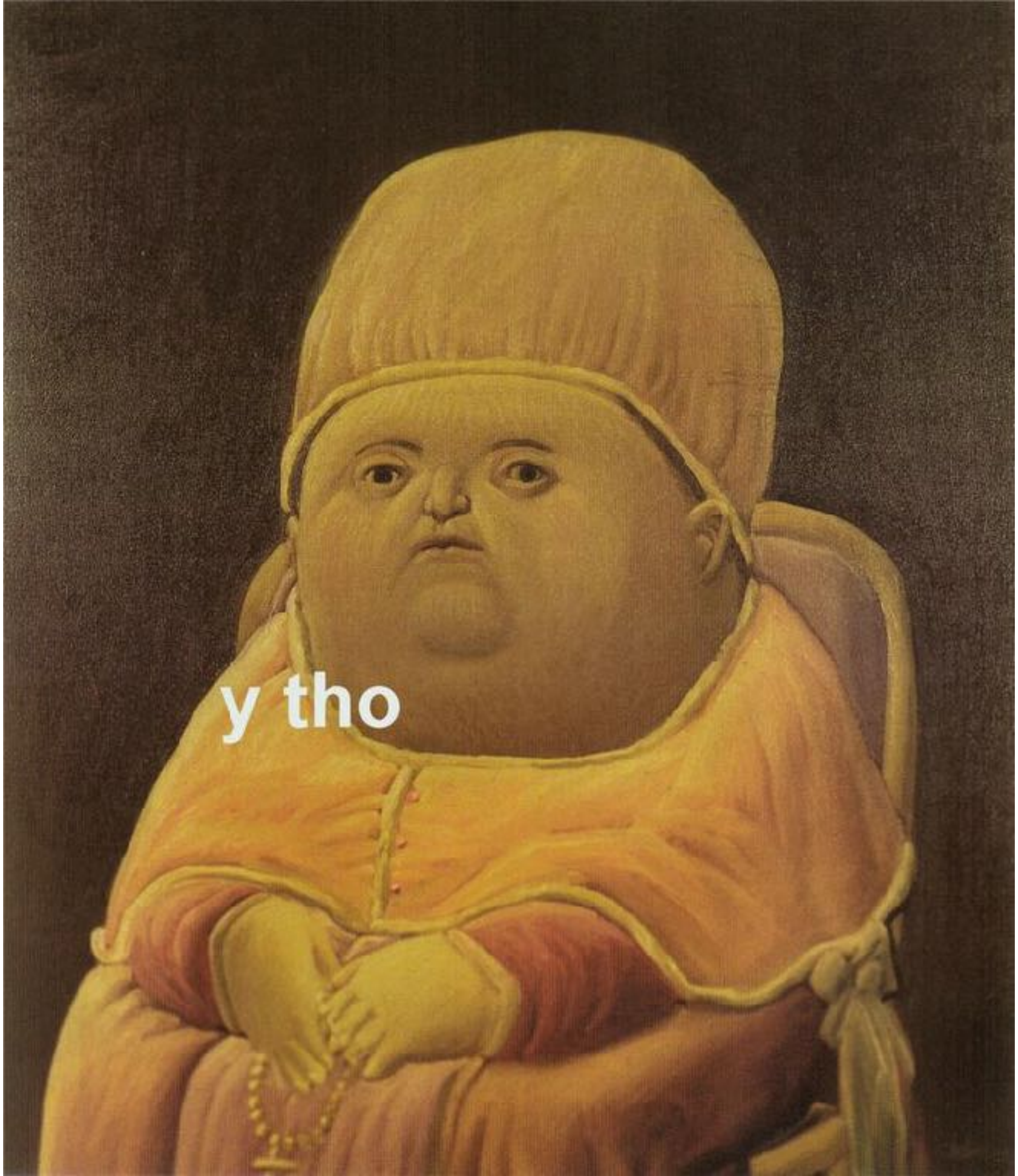
`arg_store` - **efficient** argument list storage à la array<variant>

small (< N args) large



`args` - **unparameterized** argument list view

"Type erasure" - preventing code bloat |{T1, ..., Tn}| -> 1

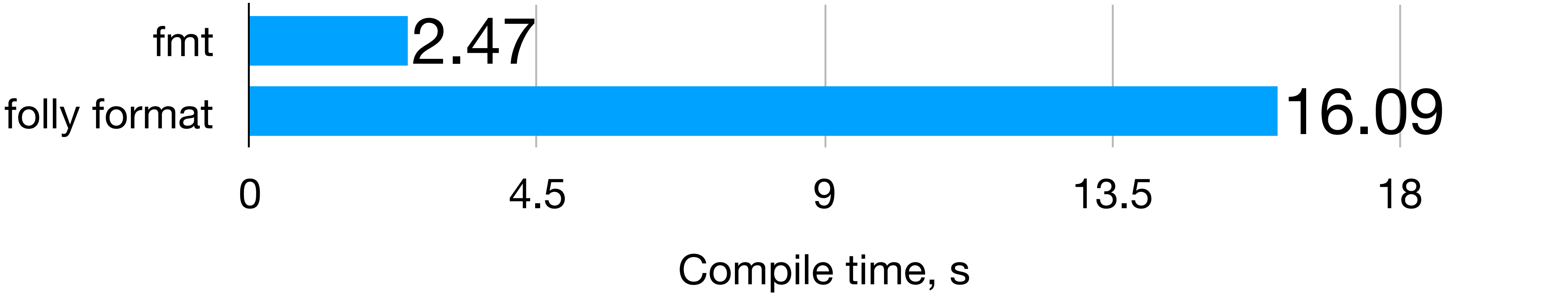

y tho

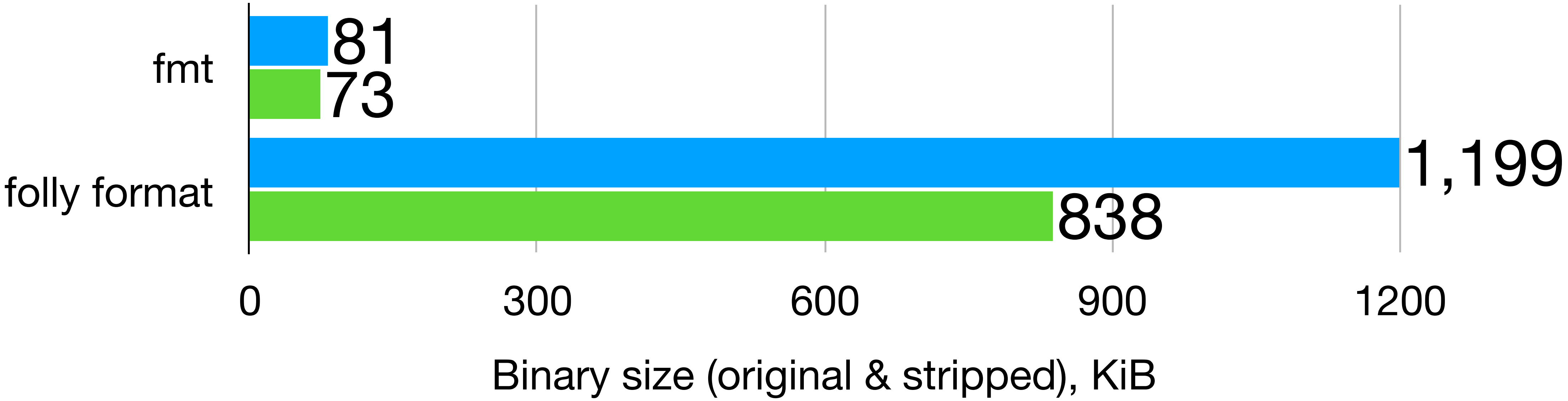
Let's benchmark

```
template <typename F>
void gen_args(F f) {
  f('x');
  f(42);
  f(4.2);
  f("foo");
  f(static_cast<void*>(0));
}

template <size_t N, typename F, typename... Args>
void gen_args(F f, Args... args) {
  if constexpr (N > 0)
    gen_args([=](auto value) { gen_args<N - 1>(f, args..., value); });
  else
    f(args...);
}

int main() {
  gen_args<3>([](auto... args) { format("{}{}{}\n", args...); });
}
```

Let's benchmark

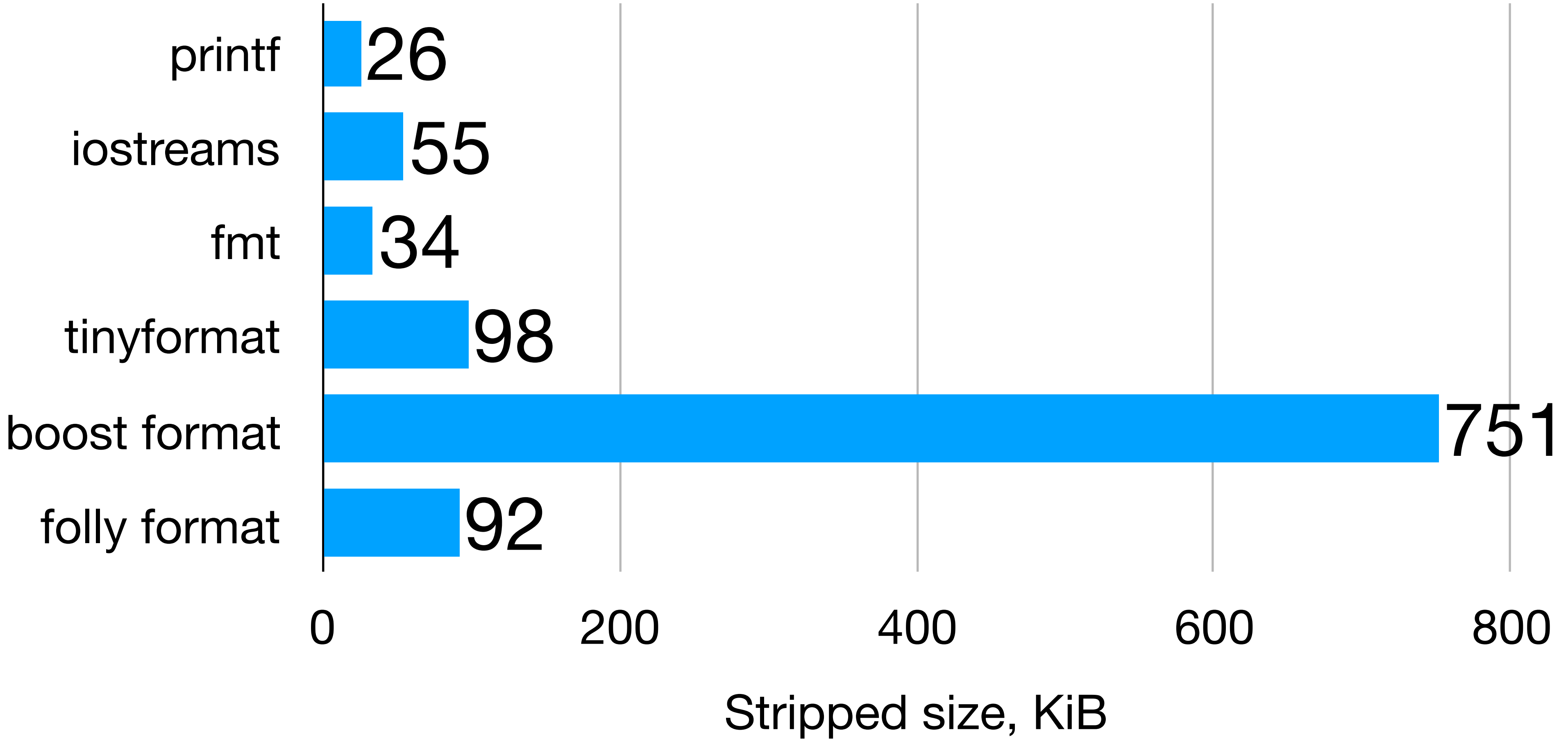
Compare with Folly Format where everything is parameterized on argument types.





Use variadic templates judiciously

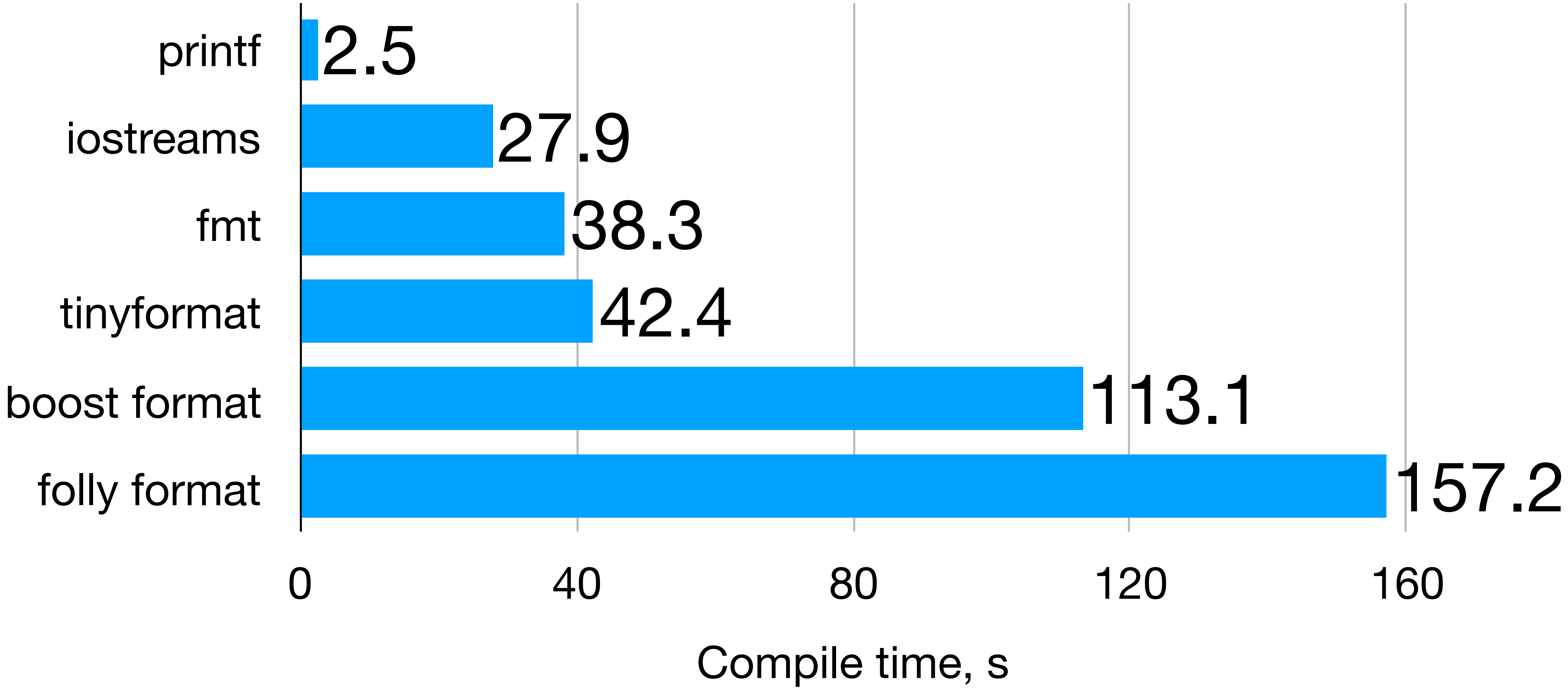
Code bloat



tinyformat benchmark: 100-TU project with 5 formatting calls per TU

Optimized build

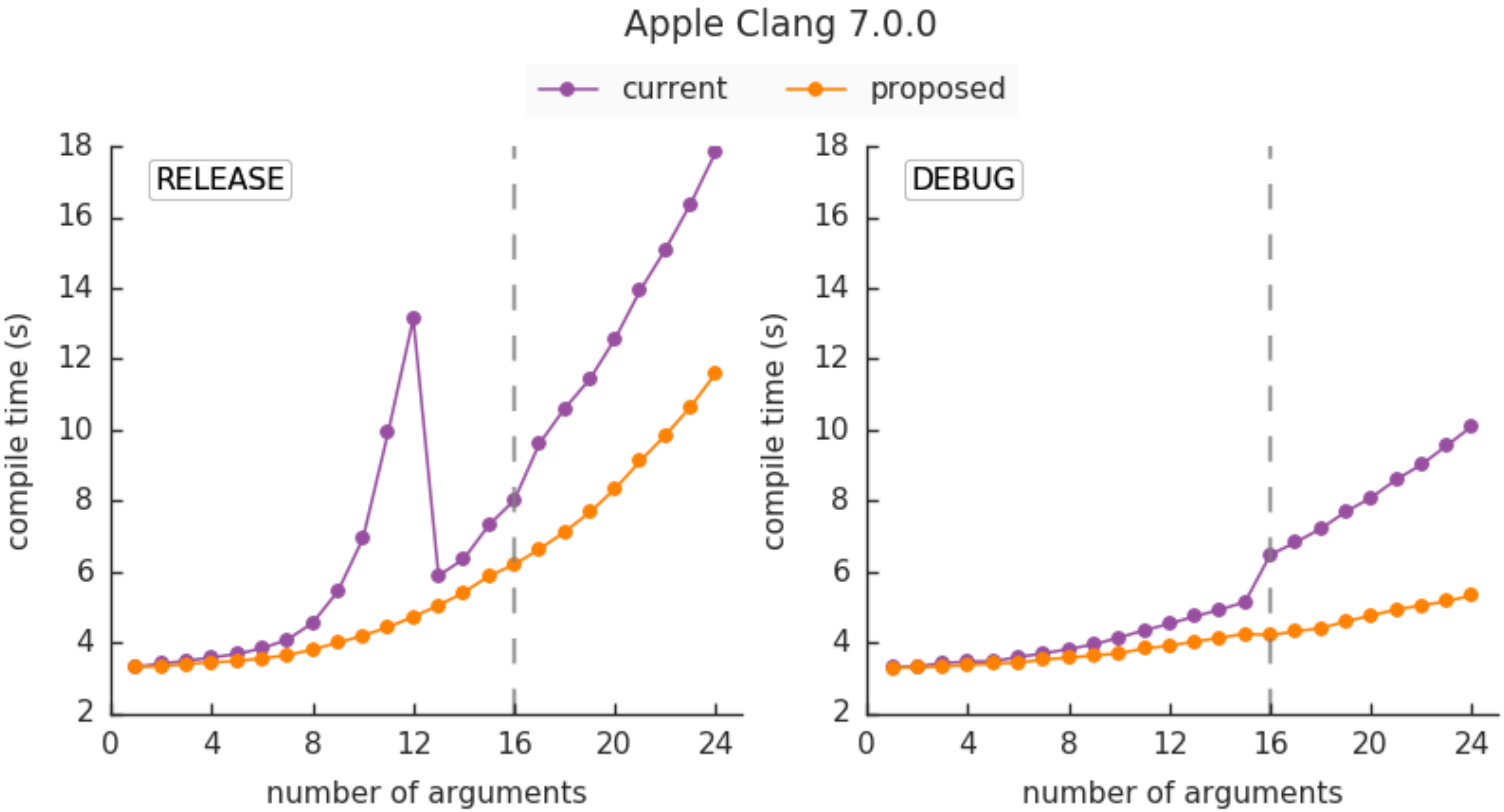
Compile time



tinyformat benchmark: 100-TU project with 5 formatting calls per TU

Optimized build

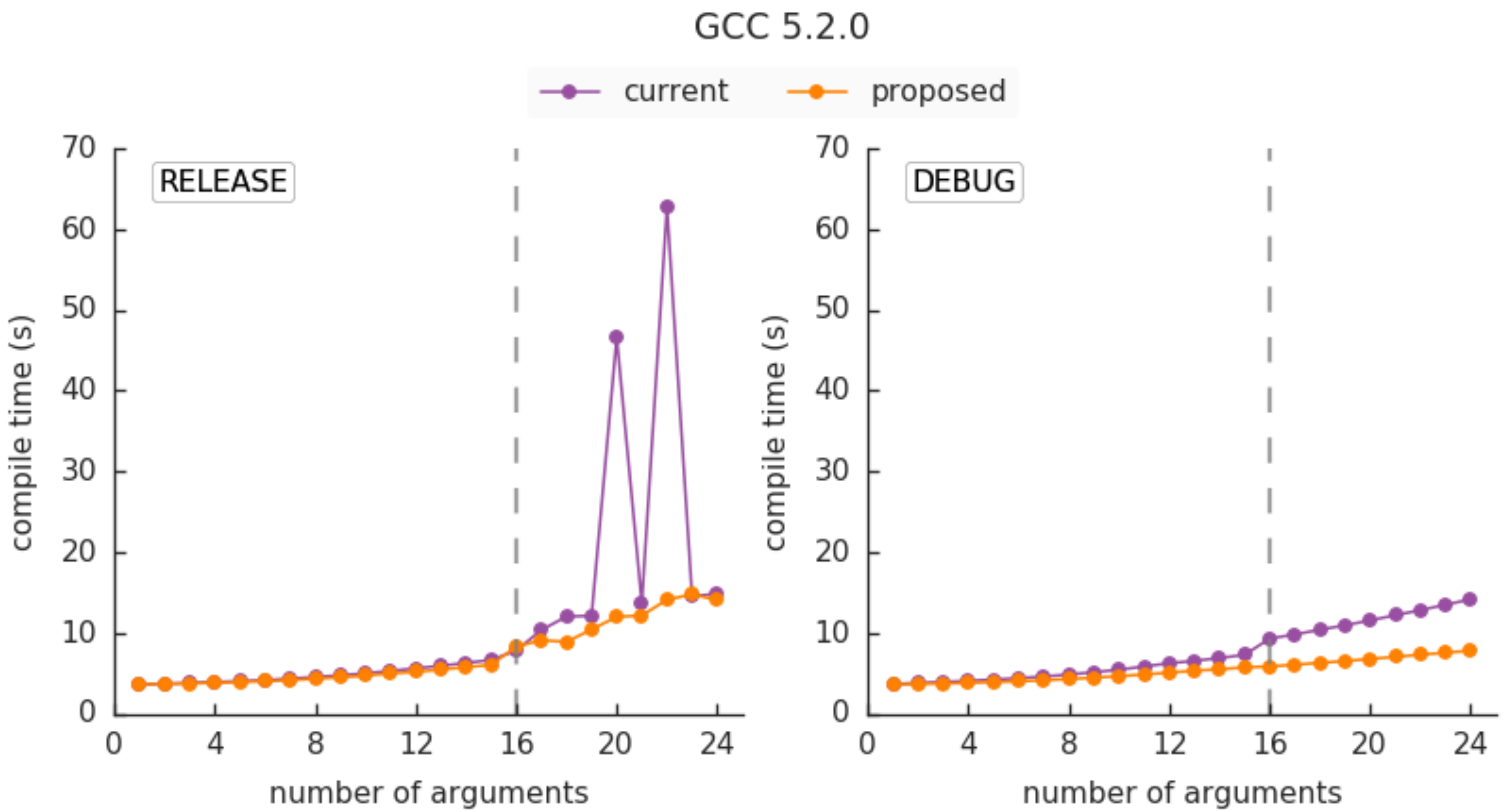
Compile time



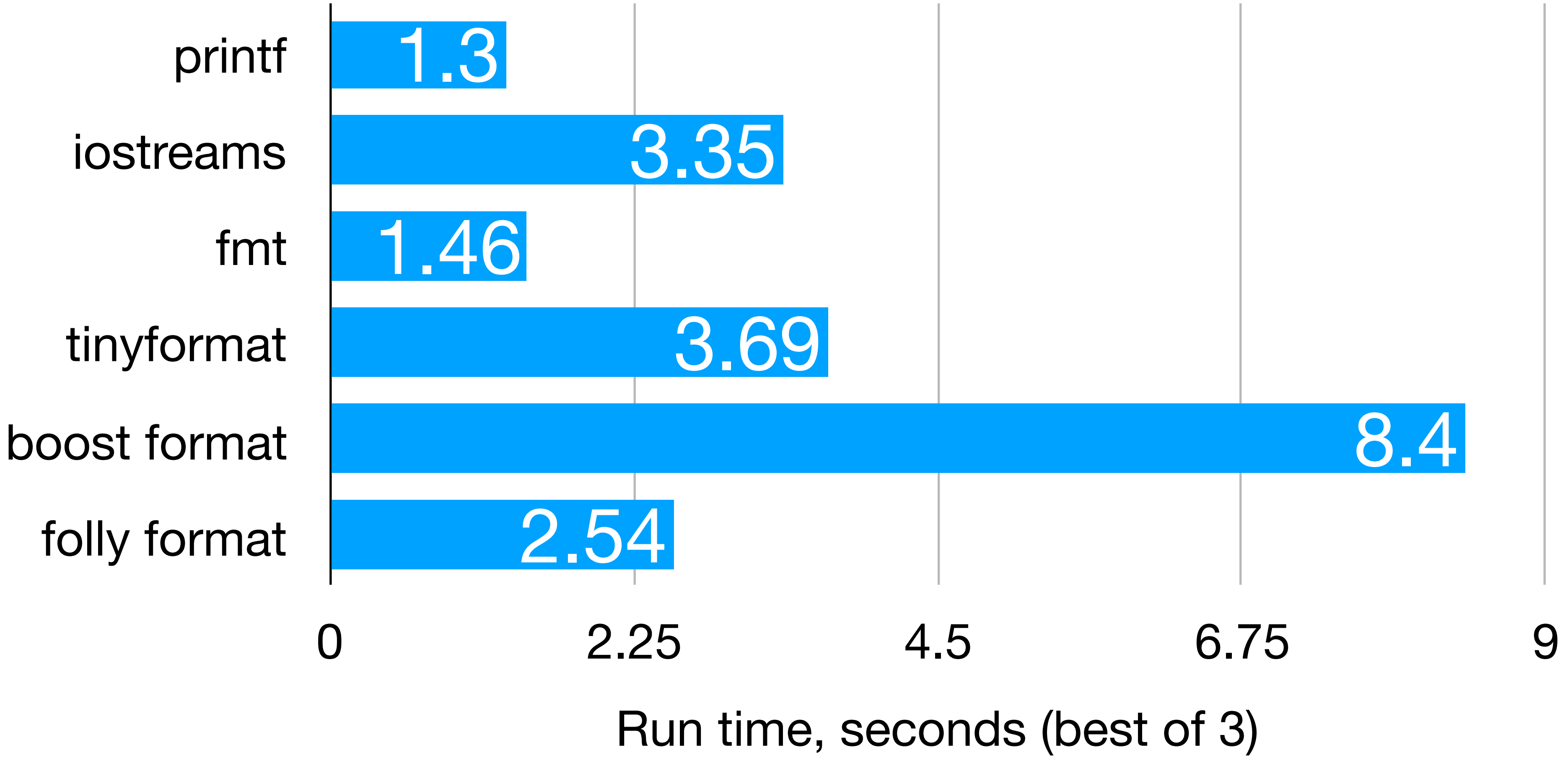
Compile time optimization work done by Dean Moldovan.

Replaced template recursion with variadic array initialization.

Compile time



Performance



tinyformat benchmark

Apple LLVM version 8.1.0 (clang-802.0.42)

macOS Sierra on Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz

format-like functions

Writing your own formatting functions

```
void vlog_error(error_code ec, string_view format,
                fmt::args args) {
  LOG(ERROR) << "error " << ec << ": "
             << fmt::vformat(format, args);
}

template <typename... Args>
void log_error(error_code ec, string_view format,
               const Args&... args) {
  vlog_error(ec, format, fmt::make_args(args...));
}
```

Usage

```
log_error(ec, "cannot open {}", filename);
```

Work in progress

- Separate parsing and formatting in extension API

```
template
struct formatter<MyType> {
  void parse(string_view format) {
    // Parse the format string.
  }

  void format(buffer& buf, const MyType& value, context& ctx) {
    // Format value.
  }
};
```

- Compile-time format string checks
- Range-based interface
- `cstring_view` -> `string_view` - **done**

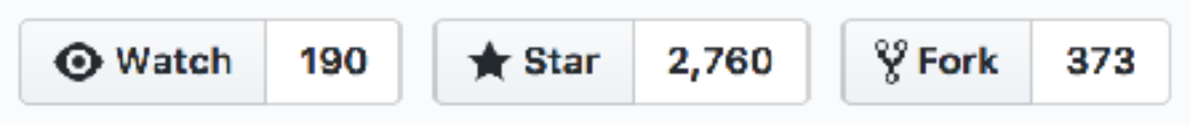
Migration path

How do we move away from printf?

- Easy mapping between printf and the new mini-language
- A compatibility library with printf-like semantics, particularly, error codes
- A tool like clang-tidy to automatically transform old code that uses literal format strings

P0645R0

The fmt library



https://github.com/fmtlib/fmt & http://fmtlib.net/

> 70 contributors:

https://github.com/fmtlib/fmt/graphs/contributors

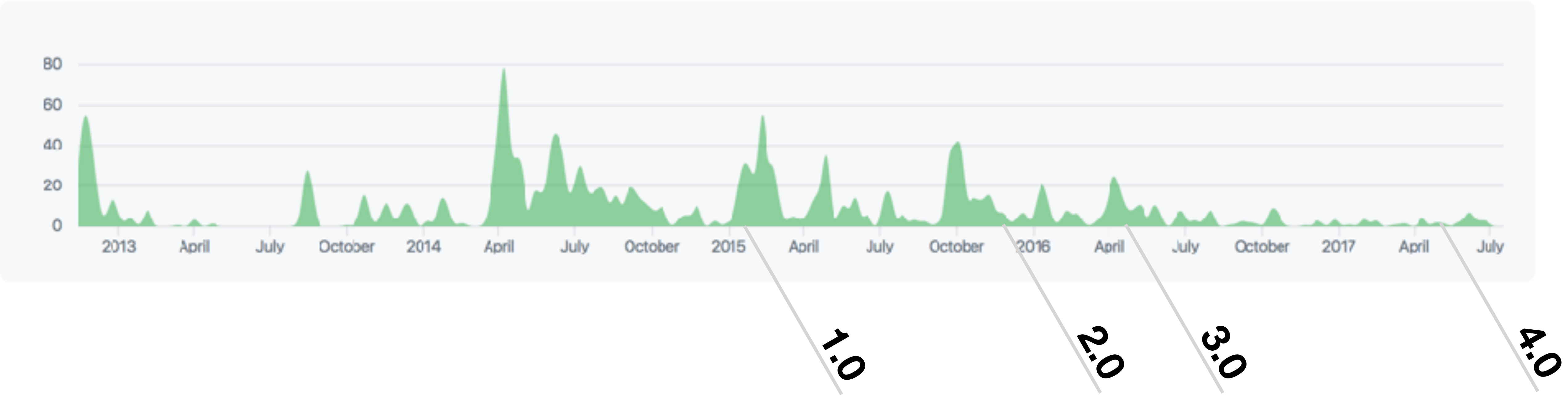
Available in package managers of major Linux distributions and in HomeBrew.

std branch - implementation of the proposal:

https://github.com/fmtlib/fmt/tree/std

Timeline

- Started in Dec 2012, originally called cppformat
- Inspired by formatting facilities in clang
- Since mid 2016 focus is on the standards proposal



Projects using fmt

- 0 A.D.: A free, open-source, cross-platform real-time strategy game
- AMPL/MP: An open-source library for mathematical programming
- CUAUV: Cornell University's autonomous underwater vehicle
- Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems (MIT)
- Envoy: C++ L7 proxy and communication bus (Lyft)
- Kodi (formerly xbmc): Home theater software
- quasardb: A distributed, high-performance, associative database
- Salesforce Analytics Cloud: Business intelligence software
- Scylla: A Cassandra-compatible NoSQL data store that can handle 1 million transactions per second on a single server
- Seastar: An advanced, open-source C++ framework for high-performance server applications on modern hardware
- spdlog: Super fast C++ logging library
- Stellar: Financial platform
- Touch Surgery: Surgery simulator
- TrinityCore: Open-source MMORPG framework

and more

Questions?