GAMECENTRIC

Alberto Barbati
alberto@gamecentric.com

C++20 Text Formatting

Italian C++ Conference 2020

June 13th, ~~Rome~~ Internet

++it Italian C++ Community

www.italiancpp.org

Who am I?

C++ programmer since 1995

Works in videogame industry since 2000

Trainer of Game Programming at Digital Bros Game Academy

Follows the works of the C++ Committee since 2008

DIGITALBROS
GAMEACADEMY

REPLY
FORGE

UBISOFT

What will we see today?

- C++20 introduces the new <format> header that provides facilities for general purpose text formatting
  - A replacement for sprintf/stringstream
  - Type-safe
  - Extensible
  - Localization friendly
  - Faster

Nice, but… why?

sprintf and friends

- Well known, in the C library since the beginning
- General purpose
- Use a string with placeholders as a template

```
sprintf(buf, "The answer is %d", 42);
```

- Formats to a memory buffer, although interface is unsafe (possible buffer overrun). A "safe" version `sprintf_s` is available

Issue: type-unsafe, not extensible

- Arguments are passed as va_list using ellipsis
- Placeholders contain information about the type
- Bad things may happen if the type implied by the placeholder does not match the type of the argument (this is partly mitigated by smart compilers/static analysis)
- No support for arguments of user-defined types

Issue: mandatory global locale

- Number formatting is always locale-dependent and relies on global state
- Can be a nuisance if you need both "C" locale and user-locale output in the same program
- Can be a problem if program uses multi-threading

Issue: argument matching

- Matching between placeholders and arguments is always positional

```
sprintf("Today is %s %d", month, dayOfMonth);
// Today is June 13

sprintf("Oggi è il %d %s", dayOfMonth, month);
// Oggi è il 13 giugno
```

- Localization may not be data-driven

C++ iostreams

- Well known, in the C++ library since the beginning
- General purpose
- Use chains of operator <<

```
stream << "The answer is " << answer << "\n";
```

Advantage: type-safe and extensible

- Type-safe: The correct formatter function is selected at compile time, inferred from the argument type
- Extensible: it's easy to write formatters for user-defined types

Issue: code-driven

- No placeholders, data is interspersed with context text strings

```
stream << "Today is " << month << " " << dayOfMonth;
```

- Complex formatting is not easily localizable
- Code is often verbose and difficult to read

Issue: mandatory locale

- Formatting of numbers is always locale dependent
  - Relies on a per-stream locale object, defaulting to std::locale::global(), so use of global state is reduced
  - You can imbue a stream with a different locale object at any time
  - You can have both "C" locale and user-locale formatting, but it's cumbersome
  - There's an overhead cost even if you use only the "C" locale

Issue: slow and bloated

- iostreams implementations are known for less than stellar performances, large code size and memory consumption
- The design doesn’t allow much improvement, since it heavily relies on polymorphism in order to handle formatting, locale and buffering at the same time

Issue: mandatory std::string

- The only standard way to format to a memory buffer is to use a stringstream
- It has a safe interface, but forces you to use std::string, which may require a dynamic allocation
- You need an extra copy to fill a pre-allocated buffer

Alternative formatting libraries

- Boost.Format
- Folly.Format
- Loki.SafeFormat
- FastFormat
- QT QString
- Etc.

And…

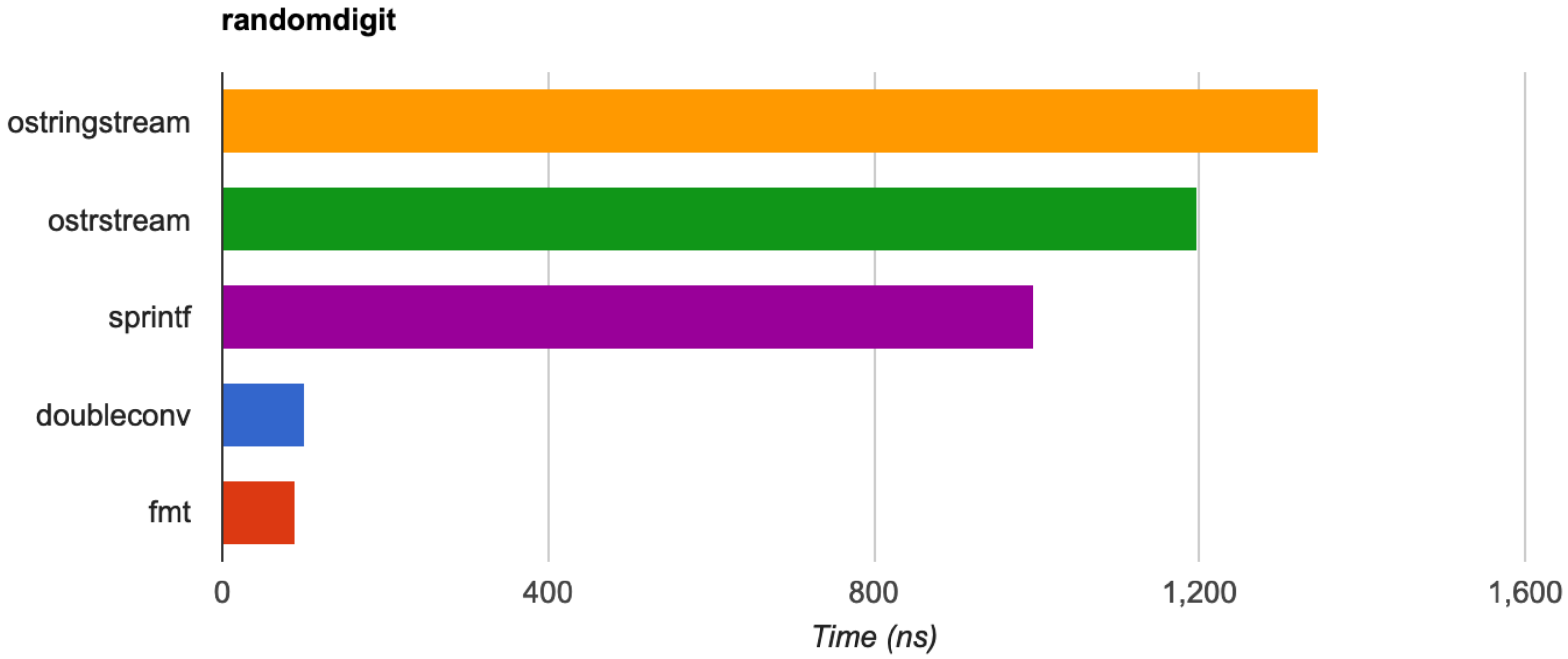
{fmt}

{fmt}

- {fmt} is the reference library that lead to the <format> proposal
- An open source library maintained by Victor Zverovich
- Available at https://fmt.dev
- MIT License
- Portable (GCC 4.8, Clang 3.0 and MSVC 19.0)
- Latest version requires C++11, but there’s also a version for C++98

{fmt} speed benchmark

(floating point formatting)



shorter is faster (source https://github.com/fmtlib/fmt)

Differences between <format> and {fmt}

- Only a subset of {fmt} has been included in C++20
- More than enough for everyday use
- A few extra {fmt} features might be considered for C++23

Overview

```
std::format("The answer is {}", 42);
```

- Uses format strings with {}-bracketed placeholders
- Clearly inspired by Python and other languages

Type safety

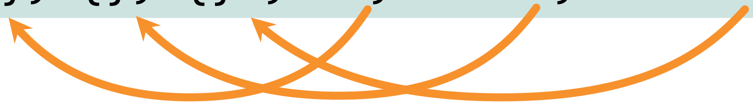
- Formatting is determined by the static type of the argument

std::format(…)	output
`"An integer: {}", 42`	`An integer: 42`
`"An ulonglong: {}", 42ull`	`An ulonglong: 42`
`"An fp number: {}", 1.234`	`An fp number: 1.234`
`"A string: {}", "hello"`	`A string: hello`

Arguments matching

- By default, placeholders are matched with arguments positionally

std::format(…)	output
`"{}, {}, {}", 42, 1.234, "hello"`	`42, 1.234, hello`



Arguments matching

- The default *automatic* indexing can be overridden by manually specifying indices

std::format(…)	output
"Today is {0}, {1}", month, day	Today is June, 13
"Oggi è il {1} {0}" , month, day	Oggi è il 13 Giugno

Arguments matching

- Manual indices can be omitted and/or repeated

std::format(...)	output
`"Omission {0} {2}", 0, 1, 2`	`Omission 0 2`
`"Repetition {0} {0} {0}", 42`	`Repetition 42 42 42`

Arguments matching

- You can't mix automatic and manual indexing in the same string

std::format(…)	output
`"Looking for troubles {} {1}"`	`throws std::format_error`

Types supported by the library

- The library supports, out of the box:
  - Arithmetic types: all integers and floating-point types, plus bool, char and wchar_t, but excluding char8_t, char16_t and char32_t
  - String types: null-terminated (const char*), char arrays, std::string and std::string_view
  - Pointer types: void* and nullptr
  - Time/date types defined in <chrono>

Format specifiers

- To customize formatting, you can provide optional format specifiers after the index, if present
- Format specifiers are always introduced by a colon

std::format("The answer is {:*specifiers*}", 42);

std::format("Today is {0:*specifiers*}, {1: *specifiers*}", month, day);

Format specifiers

- The interpretation of the format specifiers is determined by the type of the matching argument
- If a type doesn’t support a specific format specifier, a `std::format_error` exception is thrown
- The library design is open to the possibility to have the format string parsed and checked at compile time, but C++20 still lacks language support to have this performed automatically
- {fmt} implements compile time checks, but requires explicit opt-in

Format specifiers for common types

- The general format specifier for the standard-supported types is:

fill-and-align sign* # 0 *width precision* L *type

Also have a look to https://fmt.dev/latest/syntax.html

- Every piece is optional
- A few pieces only make sense for specific types
- Refer to [format.string.std] for details

Common specifiers

- A few pieces work as they do in sprintf (or similar):
  - *sign* controls the presence of a + sign before positive numbers
  - *#* switches to an "alternate" formatting
  - *0* produces 0-padding for numbers
  - *width* specifies the minimum output width
  - *precision* specifies the maximum output width (for strings) or controls the number of digits (for floating point numbers)

Alignment: examples

<table>
<tr><th>std::format(…)</th><th>output</th></tr>
<tr><td>"|{:8}|", 42</td><td>| 42|</td></tr>
<tr><td>"|{:*>8}|", 42</td><td>|******42|</td></tr>
<tr><td>"|{:*<8}|", 42</td><td>|42******|</td></tr>
<tr><td>"|{:*^8}|", 42</td><td>|***42***|</td></tr>
<tr><td>"|{:8}|", "hello"</td><td>|hello |</td></tr>
<tr><td>"|{:*>8}|", "hello"</td><td>|***hello|</td></tr>
<tr><td>"|{:*<8}|", "hello"</td><td>|hello***|</td></tr>
<tr><td>"|{:*^8}|", "hello"</td><td>|*hello**|</td></tr>
</table>

Opt-in locale support

- By default, numbers are formatted in the "C" locale, but you can opt-in for a localized formatting using the L specifier

std::format(…)	Output (in the IT_it locale)
`"{0}", 1234.56`	`1235.56`
`"{0:L}", 1234.56`	`1234,56`

- Formatting functions can either use the global locale or have the locale passed as an argument

The fmt library before 6.2.1
uses n instead of L

Integer formatting examples

std::format(...)	output
`"{:d}", 42`	`42`
`"{:b}", 42`	`101010`
`"{:o}", 42`	`52`
`"{:x}", 42`	`2a`
`"{:#x}", 42`	`0x2a`
`"{:08x}", 42`	`0000002a`
`"{:#08x}", 42`	`0x00002a`

Floating point formatting examples

- Floating point formatting rely upon C++17 function `to_chars`:

std::format(…)	to_chars equivalent	output
`"{}",    3.141592654`	`(shortest round-trip)`	`3.141592654`
`"{:f}", 3.141592654`	`chars_format::fixed`	`3.141593`
`"{:g}", 3.141592654`	`chars_format::general`	`3.14159`
`"{:e}", 3.141592654`	`chars_format::scientific`	`3.141593e+00`
`"{:a}", 3.141592654`	`chars_format::hex`	`0x1.921fb54524550p+1`

Formatting functions

- The simplest formatting function is `std::format`, which returns the formatted string as a `std::string` object:

```
template <class Args...>
string format(string_view fmt, const Args&... args);
```

Example

```
string s = format("The answer is {}", 42);
cout << format("Today is {}, {}\n", "June", 13);
```

Formatting functions

- Actually we have four overloads of format:

```
string format(string_view fmt, const Args&... args);
string format(const locale& loc, string_view fmt, const Args&... args);
wstring format(wstring_view fmt, const Args&... args);
wstring format(const locale& loc, wstring_view fmt, const Args&... args);
```

- Similarly for all other formatting functions. For the sake of brevity, I will omit the overloads for wstring and with the explicit locale object

Formatting into containers

- You can format directly into a container, without allocating a string:

```
template<class Out, class... Args>
Out format_to(Out out,
              string_view fmt,
              const Args&... args);
```

- This function takes an output iterator and returns an iterator past the end of the formatted text
- No null termination is appended

Example

```
std::vector<char> v;
format_to(back_inserter(v), "The answer is {}", 42);

format_to(ostreambuf_iterator(cout),
            "Today is {0}, {1}\n", "June", 13);
```

Formatting into fixed size buffers

- To make it safer to work with pre-allocated buffers of fixed size, there's a variant that takes an explicit maximum size:

```
template <class Out, class... Args>
format_to_n_result<Out>
format_to_n(Out out,
            iter_difference_t<Out> n,
            string_view fmt,
            const Args&... args);
```

Formatting into fixed size buffers

- The return type contains both the iterator and the full size of the formatted string

```
template <class Out>
struct format_to_n_result
{
    Out out;
    iter_difference_t<Out> size;
};
```

Example (buffer big enough)

```
char buffer[20];
auto [out, size] = format_to_n(buffer, sizeof(buffer),
                               "The answer is {}", 42);

// buffer = "The answer is 42□□□□"
// out == buffer + 16
// size == 16
```

uninitialized

Example (buffer too small)

```
char buffer[10];
auto [out, size] = format_to_n(buffer, sizeof(buffer),
                               "The answer is {}", 42);

// buffer = "The answer"    no null-termination
// out == buffer + 10       past-the-output iterator
// size == 16               required buffer size
```

Formatted size

- If you just need to know the size of the formatted string without actually formatting it, you can use

```
template<class... Args>
size_t formatted_size(string_view fmt,
                      const Args&... args);
```

Fighting code bloat

- Since all format functions we have seen so far are templates, it's important to avoid code bloat
- The library actively fights that by:
  - Packing all the variadic arguments in a type-erased struct with the function **`make_format_args()`**
  - Doing the actual formatting in a non-template function

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

Example of {fmt} generated code

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

int main()
{
    fmt::print(
       "The answer is {}.", 42);
}
```

```asm
main: # @main
  sub rsp, 24
  mov qword ptr [rsp], 42
  mov rcx, rsp
  mov edi, offset .L.str
  mov esi, 17
  mov edx, 2
  call fmt::v5::vprint(...
  xor eax, eax
  add rsp, 24
  ret
.L.str:
  .asciz "The answer is {}."
```

Custom formatting example: the type

```
struct Conference
{
    std::string title;
    std::string location;
    std::chrono::sys_days date;
};

Conference italianCpp20 { "Italian C++ Conference",
                          "Internet", // previously Rome
                          2020y/June/13 };
```

Custom formatting example: desired output

```
format("Event: {}", italianCpp20);

// output:
// Event: Italian C++ Conference – 2020-06-13, Internet
```

Custom formatters

- Formatting can be extended to support user-defined types by specializing the `std::formatter` template
- The formatter needs to implement both these steps:
  - Parsing the format specifier
  - Do the formatting

Custom formatting example: the plan

```
namespace std
{
    template<>
    struct formatter<Conference>
    {
        template <class ParseCtx>
        constexpr typename ParseCtx::iterator parse(ParseCtx& ctx)
        {
            // TODO: parsing
        }

        template <class FormatCtx>
        typename FormatCtx::iterator format(const Conference& c, FormatCtx& ctx)
        {
            // TODO: formatting
        }
    };
}
```

Custom formatting example: the plan (C++20 syntax)

```
template<>
struct std::formatter<Conference>
{
    template <class ParseCtx>
    constexpr ParseCtx::iterator parse(ParseCtx& ctx)
    {
        // TODO: parsing
    }

    template <class FormatCtx>
    FormatCtx::iterator format(const Conference& c, FormatCtx& ctx)
    {
        // TODO: formatting
    }
};
```

Custom formatting example: parse part

```
template <class ParseCtx>
constexpr ParseCtx::iterator parse(ParseCtx& ctx)
{
    // parse the range [ctx.begin(), ctx.end())
    // return the iterator to the first unparsed char
    // throw std::format_error on error
}
```

Custom formatting example: the range

```
"Lorem {:specifiers} ipsum"
         ↑               ↑
         ctx.begin()     ctx.end()
```

Custom formatting example: parse part (no specifiers)

```
template <class ParseCtx>
constexpr ParseCtx::iterator parse(ParseCtx& ctx)
{
    auto it = ctx.begin();
    if (it == ctx.end() || *it != '}')
        throw std::format_error();
    return it;
}
```

Custom formatting example: format part

```
template <class FormatCtx>
FormatCtx::iterator format(const Conference& c, FormatCtx& ctx)
{
    // format text to ctx.out()
    // return iterator past the formatted text
}
```

Custom formatting example: format part

```
template <class FormatCtx>
FormatCtx::iterator format(const Conference& c, FormatCtx& ctx)
{
    return format_to(ctx.out(),
                     "{} - {:%F}, {}",
                     c.title, c.date, c.location);
}
```

%F formats a date in ISO format
YYYY-MM-DD

A slightly more complex example

```
format("Event: {}", italianCpp20);
// Event: Italian C++ Conference – 2020-06-13, Internet

format("Event: {:L}", italianCpp20);
// Event: Italian C++ Conference – June 13 2020, Internet

format(locale("IT_it"), "Event: {:L}", italianCpp20);
// Event: Italian C++ Conference – 13 Giugno 2020, Internet
```

A slightly more complex example

```
template<>
struct std::formatter<Conference>
{
    bool localizedFormat = false;

    template <class ParseCtx>
    constexpr ParseCtx::iterator parse(ParseCtx& ctx);

    template <class FormatCtx>
    FormatCtx::iterator format(const Conference& c, FormatCtx& ctx);
};
```

A slightly more complex example: parsing

```
template <class ParseCtx>
constexpr ParseCtx::iterator parse(ParseCtx& ctx)
{
    auto it = ctx.begin();
    if (it != ctx.end() && *it == 'L')
    {
        localizedFormat = true;
        ++it;
    }
    if (it == ctx.end() || *it != '}')
        throw std::format_error();
    return it;
}
```

A slightly more complex example: formatting

```
template <class FormatCtx>
FormatCtx::iterator format(const Conference& c, FormatCtx& ctx)
{
    if (localizedFormat)
        return format_to(ctx.locale(),
                         ctx.out(),
                         "{} - {:%x}, {}",
                         c.title, c.date, c.location);
    else
        // as before
}
```

%x produces the localized date format

Another example: enumerations

```
enum class Color { Red, Green, Blue };

format("Color: {}", Color::Red);
// Color: Red
```

Another example: enumerations

```
const char* colors[] = { "Red", "Green", "Blue" };

template<>
struct std::formatter<Color> : std::formatter<const char*>
{
    // parse is inherited from base class

    auto format(Color c, auto& ctx)
    {
       // format is delegated to base class
       return formatter<const char*>::format(colors[(int)c], ctx);
    }
};
```

Recap

- The C++20 <format> header provides facilities for general purpose text formatting, addressing major issues of sprintf/stringstream:
  - Type-safe
  - Extensible
  - Localization friendly
  - Performances and code bloat
- You can try it right now by using the {fmt} library

Thanks for your attention

Questions?

Alberto Barbati
alberto@gamecentric.com
Twitter @gamecentric