Text Formatting

P0645R2

Victor Zverovich (victor.zverovich@gmail.com)

Overview

- Python-like format string syntax:

```
string s = format("{}", 42);
s = format("dec: {0:d}, hex: {0:x} oct: {0:o}", 42);
s = format("{:4.2}", 1.2345);
```

- Formatting is extensible for user-defined types

- Natural API using variadic templates:

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

- Positional arguments:

```
s = format("{1}{0}", "foo", "bar");
```

- Control over the use of locales:

```
s = format("{}", 1.2);   // not using a locale
s = format("{:n}", 1.2); // using a locale
```

Jacksonville results

We like this format syntax (vs. printf syntax).

SF	F	N	A	SA
12	5	3	3	0

We like the user-extensibility of the format syntax.

SF	F	N	A	SA
11	7	4	0	1

We want only format_to (as is, with a single output iterator).

SF	F	N	A	SA
0	0	7	9	7

We want only format_to_n.

SF	F	N	A	SA
2	5	6	5	6

We want both format_to and format_to_n.

SF	F	N	A	SA
6	11	7	1	0

https://issues.isocpp.org/show_bug.cgi?id=322

Changes

Changes since R1

- Rename `count` to `formatted_size`.
- Add the `format_to_n` function taking an output iterator and a size.
- Drop nested namespace `fmt` and add `format` to some names to prevent potential collisions.
- Add a note that compile-time processing of format strings applies to user-defined types.
- Wording cleanup

Formatted output size

"`count` (and `size`) are bad names for this function."
– Walter Brown

Function to compute the formatted output size was renamed from `count` to `formatted_size`:

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

format_to_n

"We want both `format_to` and `format_to_n`."
– LEWG

Added `format_to_n` function template that takes an iterator and a size:

```
template <class OutputIterator,
          class Size,
          class... Args>
format_to_n_result<OutputIterator, Size>
  format_to_n(
    OutputIterator out,
    Size n,
    string_view format_str,
    const Args&... args);
```

format_to_n_result

```
template <class OutputIterator, class Size>
struct format_to_n_result {
  OutputIterator out;
  Size size;
};
```

Should `Size` be a template parameter in `format_n` (and therefore in `format_to_n_result`) as in `copy_n`?

Namespace

Should formatting functions go in `std` or a nested namespace such as `std::fmt`?

```
 std::string s = std::format("{}", 42);
```

vs

```
 std::string s = std::fmt::format("{}", 42);
```

Argument access

We need access to arguments for dynamic width, precision, e.g.:

```
string s = format("{0:{1}}", "foo", 10);
//                          ^dynamic width in arg 1
// s == "10         "
```

`format_args` is a lightweight proxy object that provides access to arguments (type-erased to limit code bloat):

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

template <class... Args>
string format(
    string_view format_str,
    const Args&... args) {
  return vformat(
    format_str, make_format_args(args...));
}
```

Argument access

Argument access API:

```
template <class Context, class... Args>
using format_arg_store = unspecified;

template <class Context>
class basic_format_args {
public:
  using size_type = size_t;

  basic_format_args() noexcept;

  template <class... Args>
  basic_format_args(const format_arg_store<Context, Args...>& store);

  basic_format_arg<Context> get(size_type i) const;
};
```

Capturing arguments:

```
template <class Context, class... Args>
format_arg_store<Context, Args...>
  make_format_args(const Args&... args);
```

Argument access

Argument visitation API:

```
template <class Visitor, class Context>
/* see below */ visit(
  Visitor&& vis,
  basic_format_arg<Context> arg);
```

Previously discussed

- Extensibility
- Compile-time format strings
- Output iterators
- Benchmarks

Extensibility

Replacement field syntax

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

where `format-spec` is predefined for built-in types, but can be customized for user-defined types, e.g. `put_time`-like formatting for `tm`:

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

by providing a specialization of `formatter` for `tm`:

```
template <>
struct formatter<tm> {
  constexpr parse_context::iterator
    parse(parse_context& ctx); // note constexpr

  template <class FormatContext>
  typename FormatContext::iterator
    format(const tm& tm, FormatContext& ctx);
};
```

Compile-time format strings

Extension API is constexpr-ready: parsing can be done at compile time.

Possible API (not proposed):

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

If P0732R1 "Class Types in Non-Type Template Parameters" goes in it will be possible to do one of:

```
s = format<"{}">(42);
s = format(fmt<"{}">, 42);
```

Demonstrated to work with all of the features of the current proposal in the reference implementation (compile-time strings emulated with macros).

Runtime format strings still need to be supported, compile-time can be added later.

Output iterators

"Look at using or explain why not to use an output iterator." - LEWG

Removed the buffer API which was in R0.

Changed `format_to` to the following:

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

Benchmarks

Format 1000 random integers.

```
Run on (4 X 3100 MHz CPU s)
2018-01-27 07:12:00
Benchmark                      Time           CPU Iterations
-------------------------------------------------------------
sprintf                   882311 ns     881076 ns          781
ostringstream            2892035 ns    2888975 ns          242
to_string                1167422 ns    1166831 ns          610
format                    675636 ns     674382 ns         1045
format_to                 499376 ns     498996 ns         1263
```

Compiled with clang (Apple LLVM version 9.0.0 clang-900.0.39.2) with -O3 -DNDEBUG and run on a macOS system.

`sprintf` and `format_to` use a stack-allocated array, the rest use `std::string`.

Binary code comparison

```
void consume(const char*);

// 84 bytes
void sprintf_test() {
  char buffer[100];
  sprintf(buffer, "The answer is %d.", 42);
  consume(buffer);
}
```

```
// 127 bytes
void format_test() {
  consume(format("The answer is {}.", 42).c_str());
}

// 607 bytes
void ostringstream_test() {
  std::ostringstream ss;
  ss << "The answer is " << 42 << ".";
  consume(ss.str().c_str());
}
```

C++ source #1

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

int main() {
  std::string s =
      fmt::format("The answer is {}.", 42);
}
```

x86-64 clang 4.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 4.0.0 | -std=c++14 -O2

```
main: # @main
  sub rsp, 56
  mov qword ptr [rsp], 42
  lea rdi, [rsp + 24]
  mov r8, rsp
  mov esi, .L.str
  mov edx, 17
  mov ecx, 2
  call fmt::vformat[abi:cxx11](fmt::
  mov rdi, qword ptr [rsp + 24]
  lea rax, [rsp + 40]
  cmp rdi, rax
  je .LBB0_2
  call operator delete(void*)
.LBB0_2:
  xor eax, eax
  add rsp, 56
  ret

.L.str:
  .asciz "The answer is {}."
```

Thanks

- Beman Dawes
- Bengt Gustafsson
- Eric Niebler
- Jason McKesson
- Jeffrey Yasskin
- Joël Lamotte
- Howard Hinnant
- Lee Howes
- Louis Dionne
- Michael Park
- Thiago Macieira
- Zach Laine
- LEWG