Document number: P0645R0
Audience: Library Evolution

Victor Zverovich
Lee Howes
2017-05-22

Text Formatting

Introduction

Even with proliferation of graphical and voice user interfaces, text remains one of the main ways for humans to interact with computer programs and programming languages provide a variety of methods to perform text formatting. The first thing we do when learning a new programming language is often writing a "Hello, World!" program that performs simple formatted output.

C++ has not one but two standard APIs for doing formatted output, the `printf` family of functions inherited from C and the I/O streams library (iostreams). While iostreams are usually the recommended way of doing formatted output in C++ for safety and extensibility reasons, `printf` offers some advantages, such as arguably more natural function call API, separation of formatted message and arguments, possibly with argument reordering as a POSIX extension, and often more compact code, both source and binary.

This paper proposes a new text formatting library that can be used as a safe and extensible alternative to the `printf` family of functions. It is intended to complement the existing C++ I/O streams library and reuse some of its infrastructure such as overloaded insertion operators for user-defined types.

Example:

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

Design

Format string syntax

Variations of the printf format string syntax are arguably the most popular among the programming languages and C++ itself inherits `printf` from C [1]. The advantage of the printf syntax is that many programmers are familiar with it. However, in its current form it has a number of issues:

- Many format specifiers like `hh`, `h`, `l`, `j`, etc. are used only to convey type information. They are redundant in type-safe formatting and would unnecessarily complicate specification and parsing.
- There is no standard way to extend the syntax for user-defined types.
- There are subtle differences between different implementations. For example, POSIX positional arguments [2] are not supported on some systems [6].
- Using `'%'` in a custom format specifier, e.g. for `put_time`-like time formatting, poses difficulties.

Although it is possible to address these issues while maintaining resemblance to the original printf format, this will still break compatibility and can potentially be more confusing to users than introducing a different syntax.

Therefore we propose a new syntax based on the ones used in Python [3], the .NET family of languages [4], and Rust [5]. This syntax employs `'{'` and `'}'` as replacement field delimiters instead of `'%'` and it is described in details in the syntax reference. Here are some of the advantages:

- Consistent and easy to parse mini-language focused on formatting rather than conveying type information
- Extensibility and support for custom format strings for user-defined types
- Positional arguments
- Support for both locale-specific and locale-independent formatting (see Locale support)
- Formatting improvements such as better alignment control, fill character, and binary format

The syntax is expressive enough to enable translation, possibly automated, of most printf format strings. The correspondence between `printf` and the new syntax is given in the following table.

printf	**new**
-	<
+	+
space	*space*
#	#
0	0
hh	unused
h	unused
l	unused
ll	unused
j	unused
z	unused
t	unused
L	unused
c	c (optional)
s	s (optional)
d	d (optional)
i	d (optional)
o	o
x	x
X	X
u	d (optional)
f	f
F	F
e	e
E	E
a	a
A	A
g	g (optional)
G	G
n	unused
p	p (optional)

Width and precision are represented similarly in `printf` and the proposed syntax with the only difference that runtime value is specified by `*` in the former and `{}` in the latter, possibly with the index of the argument inside the braces.

As can be seen from the table above, most of the specifiers remain the same which simplifies migration from `printf`. Notable difference is in the alignment specification. The proposed syntax allows left, center, and right alignment represented by `'<'`, `'^'`, and `'>'` respectively which is more expressive than the corresponding `printf` syntax. The latter only supports left and right (the default) alignment.

The following example uses center alignment and `'*'` as a fill character:

```
fmt::format("{:*^30}", "centered");
```

resulting in `"**********centered**********"`. The same formatting cannot be easily achieved with `printf`.

Extensibility

Both the format string syntax and the API are designed with extensibility in mind. The mini-language can be extended for user-defined types and users can provide functions that do parsing and formatting for such types.

The general syntax of a replacement field in a format string is

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

where `format-spec` is predefined for built-in types, but can be customized for user-defined types. For example, the syntax can be extended for `put_time`-like date and time formatting

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

by providing an overload of `format_value` for `tm`:

```
void format_value(fmt::buffer& buf, const tm& tm, fmt::context& ctx);
```

The `format_value` function parses the portion of the format string corresponding to the current argument and formats the value into `buf` using `format_to` or the buffer API.

The default implementation of `format_value` calls ostream insertion `operator<<` if available.

Safety

Formatting functions rely on variadic templates instead of the mechanism provided by `<cstdarg>`. The type information is captured automatically and passed to formatters guaranteeing type safety and making many of the `printf` specifiers redundant (see Format String Syntax). Buffer management is automatic to prevent buffer overflow errors common to `printf`.

Locale support

As pointed out in P0067R1: Elementary string conversions there is a number of use cases that do not require internationalization support, but do require high throughput when produced by a server. These include various text-based interchange formats such as JSON or XML. The need for locale-independent functions for conversions between integers and strings and between floating-point numbers and strings has also been highlighted in N4412: Shortcomings of iostreams. Therefore a user should be able to easily control whether to use locales or not during formatting.

We follow Python's approach [3] and designate a separate format specifier `'n'` for locale-aware numeric formatting. It applies to all integral and floating-point types. All other specifiers produce output unaffected by locale settings. This can also have positive effect on performance because locale-independent formatting can be implemented more efficiently.

Positional arguments

An important feature for localization is the ability to rearrange formatting arguments because the word order may vary in different languages [7]. For example:

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

A possible German translation of the format string might be:

```
"%2$d Zeichen lang ist die Zeichenkette `%1$s'\n"
```

using POSIX positional arguments [2]. Unfortunately these positional specifiers are not portable [6]. The C++ I/O streams don't support such rearranging of arguments by design because they are interleaved with the portions of the literal string:

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

The current proposal allows both positional and automatically numbered arguments, for example:

```
fmt::format("String `{}' has {} characters\n", string, length(string)));
```

with the German translation of the format string:

```
"{1} Zeichen lang ist die Zeichenkette `{0}'\n"
```

Performance

The formatting library has been designed with performance in mind. It tries to minimize the number of virtual function calls and dynamic memory allocations done per a formatting operation. In particular, if formatting output can fit into a fixed-size buffer allocated on stack, it should be possible to avoid them altogether by using a suitable API.

To this end, a buffer abstraction represented by the `fmt::basic_buffer` template is introduced. A buffer is a contiguous block of memory that can be accessed directly and can optionally grow. Only one virtual function, `grow`, needs to be called during formatting and only when the buffer is not large enough.

The locale-independent formatting can also be implemented more efficiently than the locale-aware one. However, the main goal for the former is to support specific use cases (see Locale support) rather than to improve performance.

Binary footprint

In order to minimize binary code size, each formatting function that uses variadic templates is a small inline wrapper around its non-variadic counterpart. This wrapper creates an object representing an array of argument references with `fmt::make_args` and calls the non-variadic function to do the actual work. For example, the `format` variadic function calls `vformat`.

Multiple argument type codes can be combined and passed into a formatting function as a single integer if the number of arguments is small. Since argument types are known at compile time this can be an integer constant and there will be no code generated to compute it, only to store according to calling conventions.

Given a reasonable optimizing compiler, this will result in a compact per-call binary code, effectively consisting of placing argument pointers (or, possibly, copies for primitive types) and packed argument type codes on stack and calling a formatting function.

Null-terminated string view

The formatting library uses a null-terminated string view `basic_cstring_view` instead of `basic_string_view`. This results in somewhat smaller and faster code because the string size, which is not used, doesn't have to be computed and passed. Also having a termination character makes parsing easier.

Impact on existing code

The proposed formatting API is defined in the new header `<format>` and should have no impact on existing code.

Proposed wording

Insert a new section in 27.7 [iostream.format].

Header `<format>` synopsis

```
namespace std {
namespace fmt {

template <class Context>
class basic_arg;

template <class Visitor, class Context>
see below visit(Visitor&& vis, basic_arg<Context> arg);

template <class Context, class ...Args>
class arg_store;

template <class Context>
class basic_args;

template <class Char>
class basic_context;

typedef basic_context<char> context;
typedef basic_args<context> args;

template <class ...Args>
arg_store<context, Args...> make_args(const Args&... args);

template <class Context, class ...Args>
arg_store<Context, Args...> make_args(const Args&... args);

template <class Char>
class basic_buffer;

typedef basic_buffer<char> buffer;

template <class Char>
class basic_cstring_view;

typedef basic_cstring_view<char> cstring_view;

class format_error;

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

string vformat(cstring_view format_str, args args);

template <class ...Args>
string format_to(buffer& buf, cstring_view format_str, const Args&... args);

string vformat_to(buffer& buf, cstring_view format_str, args args);

template <class Char, class T>
void format_value(basic_buffer<Char>& buf, const T& value, basic_context<Char>& ctx);
```

```
}  // namespace fmt
}  // namespace std
```

Format string syntax

Format strings contain *replacement fields* surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. A brace character can be included in the literal text by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement-field ::=  '{' [arg-id] [':' format-spec] '}'
arg-id            ::=  integer
integer           ::=  digit+
digit             ::=  '0'...'9'
```

In less formal terms, the replacement field can start with an `arg-id` that specifies the argument whose value is to be formatted and inserted into the output instead of the replacement field. The `arg-id` is optionally followed by a `format-spec`, which is preceded by a colon `':'`. These specify a non-default format for the replacement value.

See also the Format specification mini-language section.

If the numerical `arg-ids` in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order.

Some simple format string examples:

```
"First, thou shalt count to {0}" // References the first argument
"Bring me a {}"                  // Implicitly references the first argument
"From {} to {}"                  // Same as "From {0} to {1}"
```

The `format-spec` field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own *formatting mini-language* or interpretation of the `format-spec`.

Most built-in types support a common formatting mini-language, which is described in the next section.

A `format-spec` field can also include nested replacement fields in certain position within it. These nested replacement fields can contain only an argument index; format specifications are not allowed. This allows the formatting of a value to be dynamically specified.

Format specification mini-language

Format specifications are used within replacement fields contained within a format string to define how individual values are presented (see Format string syntax). Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

The general form of a *standard format specifier* is:

```
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' | 'f' | 'F' | 'g' | 'G' | 'p' | 's'
int-type    ::=  'b' | 'B' | 'd' | 'o' | 'x' | 'X'
```

The `fill` character can be any character other than `'{'` or `'}'`. The presence of a fill character is signaled by the character following it, which must be one of the alignment options. If the second character of `format-spec` is not a valid alignment option, then it is assumed that both the fill character and the alignment option are absent.

The meaning of the various alignment options is as follows:

Option	Meaning
`'<'`	Forces the field to be left-aligned within the available space (this is the default for most objects).
`'>'`	Forces the field to be right-aligned within the available space (this is the default for numbers).
`'='`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form `+000000120`. This alignment option is only valid for numeric types.
`'^'`	Forces the field to be centered within the available space.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

The `sign` option is only valid for number types, and can be one of the following:

Option	Meaning
`'+'`	Indicates that a sign should be used for both positive as well as negative numbers.
`'-'`	Indicates that a sign should be used only for negative numbers (this is the default behavior).
space	Indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

The `'#'` option causes the *alternate form* to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer and floating-point types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective `"0b"` (`"0B"`), `"0"`, or `"0x"` (`"0X"`) to the output value. Whether the prefix is lower-case or upper-case is determined by the case of the type specifier, for example, the prefix `"0x"` is used for the type `'x'` and `"0X"` is used for `'X'`. For floating-point numbers the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for `'g'` and `'G'` conversions, trailing zeros are not removed from the result.

`width` is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

Preceding the `width` field by a zero (`'0'`) character enables sign-aware zero-padding for numeric types. This is equivalent to a `fill` character of `'0'` with an `alignment` type of `'='`.

The `precision` is a decimal number indicating how many digits should be displayed after the decimal point for a floating-point value formatted with `'f'` and `'F'`, or before and after the decimal point for a floating-point value formatted with `'g'` or `'G'`. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The `precision` is not allowed for integer, character, Boolean, and pointer values.

Finally, the `type` determines how the data should be presented.

The available string presentation types are:

Type	Meaning

Type	Meaning
`'s'`	String format. This is the default type for strings and may be omitted.
none	The same as `'s'`.

The available character presentation types are:

Type	Meaning
`'c'`	Character format. This is the default type for characters and may be omitted.
none	The same as `'c'`.

The available integer presentation types are:

Type	Meaning
`'b'`	Binary format. Outputs the number in base 2. Using the `'#'` option with this type adds the prefix `"0b"` to the output value.
`'B'`	Binary format. Outputs the number in base 2. Using the `'#'` option with this type adds the prefix `"0B"` to the output value.
`'d'`	Decimal integer. Outputs the number in base 10.
`'o'`	Octal format. Outputs the number in base 8.
`'x'`	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9. Using the `'#'` option with this type adds the prefix `"0x"` to the output value.
`'X'`	Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9. Using the `'#'` option with this type adds the prefix `"0X"` to the output value.
`'n'`	Number. This is the same as `'d'`, except that it uses the buffer's locale to insert the appropriate number separator characters.
none	The same as `'d'`.

Integer presentation types can also be used with character and Boolean values. Boolean values are formatted using textual representation, either true or false, if the presentation type is not specified.

The available presentation types for floating-point values are:

Type	Meaning
`'a'`	Hexadecimal floating point format. Prints the number in base 16 with prefix `"0x"` and lower-case letters for digits above 9. Uses `'p'` to indicate the exponent.
`'A'`	Same as `'a'` except it uses upper-case letters for the prefix, digits above 9 and to indicate the exponent.
`'e'`	Exponent notation. Prints the number in scientific notation using the letter `'e'` to indicate the exponent.
`'E'`	Exponent notation. Same as `'e'` except it uses an upper-case `'E'` as the separator character.
`'f'`	Fixed point. Displays the number as a fixed-point number.
`'F'`	Fixed point. Same as `'f'`, but converts `nan` to `NAN` and `inf` to `INF`.
`'g'`	General format. For a given precision `p >= 1`, this rounds the number to `p` significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of `0` is treated as equivalent to a precision of `1`.
`'n'`	Number. This is the same as `'g'`, except that it uses the buffer's locale to insert the appropriate number separator characters.
none	The same as `'g'`.

The available presentation types for pointers are:

Type	Meaning
`'p'`	Pointer format. This is the default type for pointers and may be omitted.

Type	Meaning
none	The same as `'p'`.

Formatting functions

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

Effects: The function returns a `string` object constructed from the format string argument `format_str` with each replacement field substituted with the character representation of the argument it refers to, formatted according to the specification given in the field.

Returns: The formatted string.

Throws: `format_error` if `format_str` is not a valid format string.

```
string vformat(cstring_view format_str, fmt::args args);
```

Effects: The function returns a `string` object constructed from the format string argument `format_str` with each replacement field substituted with the character representation of the argument it refers to, formatted according to the specification given in the field.

Returns: The formatted string.

Throws: `format_error` if `format_str` is not a valid format string.

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

Effects: The function appends to `buf` the format string `format_str` with each replacement field substituted with the character representation of the argument it refers to, formatted according to the specification given in the field.

Throws: `format_error` if `format_str` is not a valid format string.

```
template <class ...Args>
  void vformat_to(buffer& buf, cstring_view format_str, fmt::args args);
```

Effects: The function appends to `buf` the format string `format_str` with each replacement field substituted with the character representation of the argument it refers to, formatted according to the specification given in the field.

Throws: `format_error` if `format_str` is not a valid format string.

Formatting argument

```
template <class Context>
class basic_arg {
public:
  basic_arg();

  explicit operator bool() const noexcept;

  bool is_integral() const;
  bool is_numeric() const;
```

```
  bool is_pointer() const;
};
```

An object of type `basic_arg<Context>` represents a reference to a formatting argument parameterized on a formatting context (see 3.9). It can hold a value of one of the following types:

- `int`
- `unsigned int`
- integral types larger than `int`
- `bool`
- `Char`
- `double`
- floating-point types larger than `double`
- `const char*`
- `string_view`
- `basic_string_view<Char>` if different from `string_view`
- `const void*`
- reference to an object of a user-defined type (implementation-defined)
- `monostate` representing an empty state, i.e. when the object doesn't refer to an argument

where `Char` is `typename Context::char_type`. The value can be accessed via the visitation interface defined in the next section.

`basic_arg();`

> *Effects*: Constructs a `basic_arg` object that doesn't refer to an argument.
>
> *Postcondition*: `!(*this)`.

`explicit operator bool() const noexcept;`

> *Returns*: `true` if `*this` refers to an argument, otherwise `false`.

`bool is_integral() const;`

> *Returns*: `true` if `*this` represents an argument of an integral type.

`bool is_numeric() const;`

> *Returns*: `true` if `*this` represents an argument of a numeric type.

`bool is_pointer() const;`

> *Returns*: `true` if `*this` represents an argument of type `const void*`.

Complexity: The invocation of `is_integral`, `is_numeric`, and `is_pointer` does not depend on the number of possible value types of a formatting argument.

Formatting argument visitation

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

> *Requires:* The expression in the Effects section shall be a valid expression of the same type, for all alternative value types of a formatting argument. Otherwise, the program is ill-formed.

Effects: Let `value` be the value stored in the formatting argument `arg`. Returns `INVOKE(forward(vis), value);`.

Remarks: The return type is the common type of all possible `INVOKE` expressions in the Effects section. Since exact value types are implementation-defined, visitors should use type traits to handle multiple types.

Complexity: The invocation of the callable object does not depend on the number of possible values types of a formatting argument.

Example:

```
auto uint_value = visit([](auto value) {
  if constexpr (is_unsigned_v<delctype(value)>)
    return value;
  return 0;
}, arg);
```

Class template `arg_store`

```
template <class Context, class ...Args>
class arg_store;
```

An object of type `arg_store` stores formatting arguments or references to them.

Class template `basic_args`

```
template <class Context>
class basic_args {
public:
  typedef implementation-defined size_type;

  basic_args() noexcept;

  template <class ...Args>
  basic_args(const arg_store<Context, Args...>& store);

  basic_arg<Context> operator[](size_type i) const;
};
```

An object of type `basic_args` provides access to formatting arguments. Copying a `basic_args` object does not copy the arguments.

```
basic_args() noexcept;
```

Effects: Constructs an empty `basic_args` object.

Postcondition: `!(*this)[0]`.

```
template <class ...Args>
basic_args(const arg_store<Context, Args...>& store);
```

Effects: Constructs a `basic_args` object that provides access to the agruments in `store`.

```
basic_arg<Context> operator[](size_type i) const;
```

Requires: `i <=` the number of formatting arguments represented by the `basic_args` object.

Returns: A `basic_arg` object that represents an argument at index `i` if `i <` the number of arguments. Otherwise, returns an empty `basic_arg` object.

Function template `make_args`

```
template <class ...Args>
arg_store<context, Args...> make_args(const Args&... args);
```

Effects: The function returns an `arg_store` object that stores references to or copies of formatting arguments `args`.

Returns: The `arg_store` object constructed from formatting arguments.

Formatting context

```
template <class Char>
class basic_context {
public:
  typedef Char char_type;
  typedef basic_args<basic_context> args_type;

  basic_context(const Char* format_str, args_type args);

  const Char*& ptr();

  args_type args() const;
};
```

The class `basic_context` represents a formatting context for user-defined types. It provides access to formatting arguments and the current position in the format string being parsed.

```
basic_context(const Char* format_str, args_type args);
```

Effects: Constructs an object of class `basic_context` storing references to the formatting arguments and the format string in it.

Postcondition: `ptr() == format_str`.

```
const Char*& ptr();
```

Effects: Returns a pointer to the current position in the format string being parsed.

```
args_type args() const;
```

Effects: Returns a copy of the `args` object that was passed to the constructor of `basic_context`.

Formatting buffer

```
template <class Char>
class basic_buffer {
public:
  typedef implementation-defined size_type;

  virtual ~basic_buffer();

  size_type size() const noexcept;
  size_type capacity() const noexcept;
```

```
  void resize(size_type sz);
  void reserve(size_type n);

  Char* data() noexcept;
  const Char* data() const noexcept;

  void push_back(const Char& x);

  template <class InputIterator>
  void append(InputIterator first, InputIterator last);

  virtual locale locale() const;

protected:
  basic_buffer() noexcept;

  void set(Char* s, size_type n) noexcept;

  virtual void grow(size_type n) = 0;
};
```

The class `basic_buffer<T>` represents a contiguous memory buffer with an optional growing ability. An instance of `basic_buffer<T>` stores elements of type `T`. The elements of a `basic_buffer` are stored contiguously, meaning that if `b` is a `basic_buffer<T>` then it obeys the identity `&b.data()[n] == &b.data()[0] + n` for all `0 <= n < d.size()`.

`basic_buffer() noexcept;`

Effects: Constructs an empty buffer.

Postcondition: `size() == 0`.

`set(Char* s, size_type n) noexcept;`

Effects: Sets data and capacity.

Postcondition: `data() == s` and `capacity() == n`.

`size_type size() const noexcept;`

Returns: The buffer size.

`size_type capacity() const noexcept;`

Returns: The total number of elements that the buffer can hold without requiring reallocation.

`void resize(size_type sz);`

Effects: If `sz <= size()`, removes last `size() - sz` elements from the buffer. If `size() < sz`, appends `sz - size()` default-initialized elements to the buffer.

Postcondition: `size() == sz`.

`void reserve(size_type n);`

Requires: `T` shall be `MoveInsertable` into `*this`.

Effects: A directive that informs a buffer of a planned change in size, so that it can manage the storage allocation accordingly. After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if

reallocation happens; and equal to the previous value of `capacity()` otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`, and it is performed by calling `grow(n)`. If an exception is thrown other than by the move constructor of a non-`CopyInsertable` type, there are no effects.

```
void grow(size_type n);
```

Requires: `n > capacity()`.

Effects: Reallocates the buffer to increase its capacity to at least `n` in a way that is defined separately for each class derived from `basic_buffer`.

Throws: `length_error` if `n` exceeds a limit defined by a derived class, in particular, if the latter has a fixed capacity.

```
Char* data() noexcept;
const Char* data() const noexcept;
```

Returns: A pointer such that `[data(), data() + size())` is a valid range.

```
void push_back(const Char& x);
```

Effects: Equivalent to `append(&x, &x + 1)`.

```
template <class InputIterator>
void append(InputIterator first, InputIterator last);
```

Let `n` be the number of elements in the range `[first, last)`.

Requires: `[first, last)` is a valid range.

Throws: `length_error` if `size() + n > capacity()` and exceeds the capacity limit defined by a derived class.

Effects: The function replaces the buffer controlled by `*this` with a buffer of length `size() + n` whose first `size()` elements are a copy of the original buffer controlled by `*this` and whose remaining elements are a copy of the elements in the range.

```
locale locale() const;
```

Returns: The locale to be used for locale-specific formatting. The default implementation returns a copy of the global C++ locale but derived classes may return different locales.

Format string

```
template <class Char>
class basic_cstring_view {
public:
  basic_cstring_view(const basic_string<Char>& str);
  basic_cstring_view(const Char* str);

  const Char* c_str() const noexcept;
};
```

The class `basic_cstring_view` represents a null-terminated string view.

```
basic_cstring_view(const basic_string<Char>& str);
```

Effects: Constructs an object of class `basic_cstring_view`.

Postcondition: `c_str() == str.c_str()`.

```
basic_cstring_view(const Char* str);
```

Effects: Constructs an object of class `basic_cstring_view`.

Postcondition: `c_str() == str`.

```
const Char* c_str() const noexcept;
```

Returns: A pointer to the string.

User-defined types

If a format string refers to an object of a user-defined type as in

```
X x;
string s = format("{}", x);
```

the formatting function will call `format_value(buf, x, ctx)`, where `buf` is a reference to the formatting buffer, `x` is a const reference to the argument and `ctx` is a reference to the formatting context. `ctx.ptr()` will point to one of the following positions in the format string being parsed:

- `':'` preceding `format-spec` for the current argument,
- `'}'` if there is no `format-spec`.

The `format_value` function should parse `format-spec`, format the argument and advance `ctx.ptr()` to point to `'}'` that ends `replacement-field` for the current argument.

The default implementation of `format_value` calls ostream insertion `operator<<` to format the value.

```
template <class Char, class T>
void format_value(basic_buffer<Char>& buf, const T& value, basic_context<Char>& ctx);
```

Effects: The function calls `os << value`, where `os` is an instance of `std::basic_ostream<Char>` with a stream buffer writing to `buf`.

Throws: `format_error` if `*ctx.ptr() != '}'`.

Error reporting

```
class format_error : public runtime_error {
public:
  explicit format_error(const string& what_arg);
  explicit format_error(const char* what_arg);
};
```

The class `format_error` defines the type of objects thrown as exceptions to report errors from the formatting library.

```
format_error(const string& what_arg);
```

Effects: Constructs an object of class `format_error`.

Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

`format_error(const char* what_arg);`

Effects: Constructs an object of class `format_error`.

Postcondition: `strcmp(what(), what_arg) == 0`.

Related work

The Boost Format library [8] is an established formatting library that uses printf-like format string syntax with extensions. The main differences between this library and the current proposal are:

- Syntax: for the reasons descibed in section Format String Syntax this proposal uses a new syntax instead of extending the printf one. This allows much simpler and easier to parse grammar, not burdened by legacy specifiers used to convey type information. For example, Boost Format has two ways to refer to an argument by index and allows but ignores some format specifiers.
- API: Boost Format uses `operator%` to pass formatting arguments while this proposal uses variadic function templates.
- Performance: The implementation of this proposal is several times faster that the implementation of Boost Format on tinyformat benchmarks [9], generates smaller binary code and is faster to compile.

A printf-like Interface for the Streams Library [10] is similar to the Boost Format library but uses variadic templates instead of `operator%`. Unfortunately it hasn't been updated since 2013 and the same arguments about format string syntax apply to it.

The FastFormat library [11] is another well-known formatting library. Similarly to this proposal, FastFormat uses brace-delimited format specifiers, but otherwise the format string syntax is different and the library has significant limitations [12]:

> 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

Formatting facilities of the Folly library [13] are the closest to the current proposal. Folly also uses Python-like format string syntax nearly identical to the one described here. However, the API details are quite different. The current proposal tries to address performance and code bloat issues that are largely ignored by Folly Format. For instance formatting functions in Folly Format are parameterized on all argument types while in this proposal, only the inlined wrapper functions are, which results in much smaller binary code and better compile times.

Implementation

An implementation of this proposal is available in the `std` branch of the open-source fmt library [14].

Acknowledgements

The Format String Syntax section is based on the Python documentation [3].

References

[1] *The `fprintf` function. ISO/IEC 9899:2011. 7.21.6.1.*
[2] *fprintf, printf, snprintf, sprintf - print formatted output. The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition.*
[3] *6.1.3. Format String Syntax. Python 3.5.2 documentation.*
[4] *String.Format Method. .NET Framework Class Library.*
[5] *Module `std::fmt`. The Rust Standard Library.*
[6] *Format Specification Syntax: printf and wprintf Functions. C++ Language and Standard Libraries.*
[7] *10.4.2 Rearranging printf Arguments. The GNU Awk User's Guide.*
[8] *Boost Format library. Boost 1.63 documentation.*
[9] *Speed Test. The fmt library repository.*
[10] *A printf-like Interface for the Streams Library (revision 1).*
[11] *The FastFormat library website.*
[12] *An Introduction to Fast Format (Part 1): The State of the Art. Overload Journal #89 - February 2009*
[13] *The folly library repository.*
[14] *The fmt library repository.*