P1729R1
Text Parsing

Published Proposal, 2019-10-06

This version:
http://wg21.link/P1729R1

Authors:
Victor Zverovich
Elias Kosunen

Project:
ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Audience:
LEWGI

Abstract

This paper discusses a new text parsing facility to complement the text formatting functionality of [P0645].

§ 1. Introduction

[P0645] has proposed a text formatting facility that provides a safe and extensible alternative to the `printf` family of functions. This paper explores the possibility of adding a symmetric parsing facility which is based on the same design principles and shares many features with [P0645], namely

- Safety
- Extensibility
- Performance
- Locale control
- Small binary footprint
- Integration with chrono

According to [CODESEARCH], a C and C++ codesearch engine based on the ACTCD19 dataset, there are 389,848 calls to `sprintf` and 87,815 calls to `sscanf` at the time of writing. So although formatted input functions are less popular than their output counterparts, they are still widely used.

Lack of a general-purpose parsing facility based on format strings has been raised in [P1361] in the context of formatting and parsing of dates and times.

Although having a symmetric parsing facility seems beneficial, not all languages provide it out-of-the-box. For example, Python doesn't have a `scanf` equivalent in the standard library but there is a separate `parse` package ([PARSE]).

Example:

```
std::string key;
int value;
std::scan("answer = 42", "{} = {}", key, value);
//        ~~~~~~~~~~~~~  ~~~~~~~~~  ~~~~~~~~~~
//            input        format    arguments
//
// Result: key == "answer", value == 42
```

§ 2. Design

The new parsing facility is intended to complement the existing C++ I/O streams library, integrate well with the chrono library, and provide an API similar to `std::format`. This section discusses major features of its design.

§ 2.1. Format strings

As with `printf`, the `scanf` syntax has the advantage of being familiar to many programmers. However, it has similar limitations:

- Many format specifiers like `hh`, `h`, `l`, `j`, etc. are used only to convey type information. They are redundant in type-safe parsing and would unnecessarily complicate specification and parsing.
- There is no standard way to extend the syntax for user-defined types.
- Using `'%'` in a custom format specifier poses difficulties, e.g. for `get_time`-like time parsing.

Therefore we propose a syntax based on [PARSE] and [P0645]. This syntax employs `'{'` and `'}'` as replacement field delimiters instead of `'%'`. It will provide the following advantages:

- An easy to parse mini-language focused on the data format rather than conveying the type information
- Extensibility for user-defined types
- Positional arguments
- Support for both locale-specific and locale-independent parsing (see §2.5 Locales)
- Consistency with `std::format` proposed by [P0645].

At the same time most of the specifiers will remain the same as in `scanf` which can simplify, possibly automated, migration.

§ 2.2. Safety

scanf is arguably more unsafe than printf because __attribute__((format(scanf, ...))) ([ATTR]) implemented by GCC and Clang doesn't catch the whole class of buffer overflow bugs, e.g.

```
char s[10];
std::sscanf(input, "%s", s); // s may overflow.
```

Specifying the maximum length in the format string above solves the issue but is error-prone especially since one has to account for the terminating null.

Unlike scanf, the proposed facility relies on variadic templates instead of the mechanism provided by <cstdarg>. The type information is captured automatically and passed to scanners guaranteeing type safety and making many of the scanf specifiers redundant (see §2.1 Format strings). Memory management is automatic to prevent buffer overflow errors.

§ 2.3. Extensibility

We propose an extension API for user-defined types similar to the one of [P0645]. It separates format string processing and parsing enabling compile-time format string checks and allows extending the format specification language for user types.

The general syntax of a replacement field in a format string is the same as in [P0645]:

```
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 get_time-like date and time formatting

```
auto t = tm();
scan(input, "Date: {0:%Y-%m-%d}", t);
```

by providing a specialization of scanner for tm:

```
template <>
struct scanner<tm> {
  constexpr scan_parse_context::iterator parse(scan_parse_context& ctx);

  template <class ScanContext>
  typename ScanContext::iterator scan(tm& t, ScanContext& ctx);
};
```

The `scanner<tm>::parse` function parses the `format-spec` portion of the format string corresponding to the current argument and `scanner<tm>::scan` parses the input range `[ctx.begin(), ctx.end())` and stores the result in `t`.

An implementation of `scanner<T>::scan` can potentially use ostream extraction `operator>>` for user-defined type `T` if available.

§ 2.4. Iterator and range support

Currently, this paper proposes taking a `string_view` (and possibly a `wstring_view`) as the first parameter to `std::scan`. While convenient for the common case of parsing a value from a string, it's less so for a lot of other cases.

Consider reading an integer from a file. Because this proposed facility doesn't concern itself with I/O, it's not possible without reaching to other standard APIs, which are not designed for this purpose. There are two conceivable approaches to this:

Reading the file character-by-character with C stdio:

```
auto f = std::fopen(...);

std::string buf;
for (int ch = 0; (ch = std::fgetc(f)) != EOF;) {
  if (std::isspace(ch)) break;
  buf.push_back(ch);
}

// buf now contains the input
int i{};
auto ret = std::scan(buf, "{}", i);

// Input has some leftovers and wasn’t exhausted.
// Unused parts need to be put back into the file stream
// This can happen when, for example, the input is "4.2":
// '4' is read into the integer, ".2" is left over, and must be put back.
// This is consistent with iostreams and scanf.
if (!ret.empty()) {
  for (auto it = ret.rbegin(); it != ret.rend(); ++it) {
    if (std::ungetc(*it, f) == EOF) {
      // Putback failed; file stream unusable
      throw ...;
    }
  }
}
```

Reading a "word" from the file with `fstream`:

```
auto f = std::fstream(...);

std::string buf{};
f >> buf;

int i{};
auto ret = std::scan(buf, "{}", i);

// See above for rationale
if (!ret.empty()) {
  for (auto it = ret.rbegin(); it != ret.rend(); ++it) {
    if (!f.putback(*it)) {
      // Putback failed; file stream unusable
      throw ...;
    }
  }
}
```

Both of these approaches are flawed. The first one has a significant amount of bookkeeping that's required of the user, and so does the second one, although not as much. Both examples are difficult to use and are easy to get wrong. In the second example, it'd probably be better to skip the hassle, and just do `f >> my_int` directly, making this proposal obsolete.

One could argue, that solving this problem is out-of-scope for this proposal. [P0645] doesn't deal with I/O, and maybe this paper shouldn't either. Comparisons to [P0645] in this case may be unwarranted, however. Scanning without I/O is significantly less powerful than formatting without I/O; scanning is, inherently, a different process from formatting.

With formatting, you can just dump all your values into a string. With scanning, that just isn't possible, as seen earlier. The source has to be read lazily, and might grow mid-call.

The solution to this is to make `std::scan` do I/O and take a `range`:

```
// exposition only
template <typename Range>
concept scan-range =
    std::ranges::bidirectional_range<Range> &&
    std::ranges::view<Range> &&
    std::ranges::pair-reconstructible-range<Range>;
```

Although not a part of this proposal, a range type wrapping a file could be created (bidirectional `std::istreambuf_iterator`? range-based I/O?), making the above use cases trivial.

```
// Hypothetical
auto f = std::io::ifile(...);

// All the buffer management is dealt with by the range and std::scan
// Range iterator operator++ would read a character,
// operator* would return the last character read,
// and operator-- would putback a character
int i{};
f = std::scan(std::io::file_view(f), "{}", i);
```

Why `bidirectional_range`?

The reason is error recovery. In the case that reading an argument fails, the range needs to be reset to the state it was in before starting to read that value. This behavior is consistent with scanf.

```
auto input = "42 foo"sv;
int i, j;
// Reading of j failed: foo is not an integer
input = std::scan(input, "{} {}", i, j);
// input == "foo"
// i == 42
// j is uninitialized (not written to)
```

In fact, this is main the reason why `std::ungetc` and `std::istream::unget` exist.

See more discussion on error handling in §3.1 Error handling and partial successes.

Why `view`?

It makes ownership semantics clearer, and avoids potentially expensive copying.

```
std::string str = "verylongstring";
str = std::scan(str, ...);
// str would have to be reallocated and its contents moved
```

Why `pair-reconstructible-range`?

(Depends on [P1664].)

Should be pretty obvious. While parsing the input range, `std::scan` advances an iterator into that range. Once the parsing is complete, the range needs to be returned starting from the iterator.

Parsing operations could be optimized further for more refined range concepts. For example, for `contiguous_range`s, `string_view`s could be read, eliminating an extra allocation and a copy into a `string`.

A problem with taking a range as input would be a potential increase in generated code size, as the internals would need to be instantiated for every used range type. This could be mitigated by:

- Generating non-inline `vscan` overloads for different range types -> increase of library compile time and binary size (first one a non-issue for stdlib, second one maybe not so much)
  - `string_view` is a prime candidate for this
- Type-erasing the range alongside the arguments -> virtual function call with every `*it` and `++it` (significant performance degradation)

§ 2.5. Locales

As pointed out in [N4412]:

> There are a number of communications protocol frameworks in use that employ text-based representations of data, for example XML and JSON. The text is machine-generated and machine-read and should not depend on or consider the locales at either end.

To address this [P0645] provided control over the use of locales. We propose doing the same for the current facility by performing locale-independent parsing by default and designating separate format specifiers for locale-specific one.

§ 2.6. Performance

The API allows efficient implementation that minimizes virtual function calls and dynamic memory allocations, and avoids unnecessary copies. In particular, since it doesn’t need to guarantee the lifetime of the input across multiple function calls, `scan` can take `string_view` avoiding an extra string copy compared to `std::istringstream`.

We can also avoid unnecessary copies required by `scanf` when parsing string, e.g.

```
std::string_view key;
int value;
std::scan("answer = 42", "{} = {}", key, value);
```

This has lifetime implications similar to returning match objects in [P1433] and iterator or subranges in the ranges library and can be mitigated in the same way.

§ 2.7. Binary footprint

We propose using a type erasure technique to reduce per-call binary code size. The scanning function that uses variadic templates can be implemented as a small inline wrapper around its non-variadic counterpart:

```
string_view vscan(string_view input, string_view fmt, scan_args args);

template <typename... Args>
inline auto scan(string_view input, string_view fmt, const Args&... args) {
  return vscan(input, fmt, make_scan_args(args...));
}
```

As shown in [P0645] this dramatically reduces binary code size which will make `scan` comparable to `scanf` on this metric.

§ 2.8. Integration with chrono

The proposed facility can be integrated with `std::chrono::parse` ([P0355]) via the extension mechanism similarly to integration between chrono and text formatting proposed in [P1361]. This will improve consistency between parsing and formatting, make parsing multiple objects easier, and allow avoiding dynamic memory allocations without resolving to deprecated `strstream`.

Before:

```
std::istringstream is("start = 10:30");
std::string key;
char sep;
std::chrono::seconds time;
is >> key >> sep >> std::chrono::parse("%H:%M", time);
```

After:

```
std::string key;
std::chrono::seconds time;
std::scan("start = 10:30", "{0} = {1:%H:%M}", key, time);
```

Note that the `scan` version additionally validates the separator.

§ 2.9. Impact on existing code

The proposed API is defined in a new header and should have no impact on existing code.

§ 3. Open design questions

§ 3.1. Error handling and partial successes

This paper deliberately avoids dealing with errors at this point. To be consistent with [P0645], exceptions would be the proper way to deal with errors. The problem with this approach is partial successes.

```
auto input = "42 foo"sv;
int i, j;
try {
    // Will throw:
    // foo is not an integer
    input = std::scan(input, "{} {}", i, j);

    // If this was ever reached,
    // i and j would both be usable here
} catch (const std::scan_error& e) {
    input = e.input;
    if (e.read == 1) {
        // 1 value read
        // Only i usable
    } else {
        // No values read
        // Neither i or j usable
    }
}
```

Problems with std::scan communicating partial successes by throwing:

- The read values need to be declared far away from their use site: declaration is outside the try-block, while using the value is after the call and in the catch-block.
- Possible code duplication: Say, for example, that the user wants to do something for the value i in the previous example. This code would have to be in two places: in the end of the try-block, and inside the if in the catch-block. Also, the input needs to be reassigned in two different places: in the try-block in the case of a success, and in the catch on failure.

An alternative would be to have an expected-like return type ([P0323]), except the success side should always be present, almost like:

```
namespace std {
template <typename Range>
struct scan_result {
    int read;
    Range input;
    optional<scan_error> error;
};
}

auto input = "42 foo"sv;
int i, j;
auto ret = std::scan(input, "{} {}", i, j);
input = ret.input;
if (ret.read >= 1) {
    // i is usable
} else {
    // Neither i or j is usable
}
```

This would, of course, be inconsistent with [P0645], which might be undesirable.

§ 3.2. Returning a tuple vs. output parameters

In Cologne, LEWGI encouraged to explore an alternative API returning a tuple, instead of scanf-like output parameters. We find the scanf approach to be superior, for the following reasons:

- The API would be clunky to allow for partial successes. The return type would have to be std::tuple<std::optional<T>, std::optional<U>...>, or even `std::scan_result<std::tuple<std::optional<T>, std::optional<U>...>>`. Using this is awkward, with all the destructuring the user would have to do.

- Tuples, in conjunction with optionals, have a measurable overhead compared to output parameters, at ~5-10%, depending on the use case, according to [SCNLIB] benchmarks.

§ 3.3. Naming

1. `scan`
2. `parse`
3. other

The name "parse" is a bit problematic because of ambiguity between format string parsing and input parsing.

"scan" as a name collides with some of the new C++17 `<numeric>` algorithms:

- `std::inclusive_scan`
- `std::exclusive_scan`
- `std::transform_inclusive_scan`
- `std::transform_exclusive_scan`

"scan" is the name used by [SCNLIB] and [FMT], and is the authors' preferred name, and would be consistent with existing `scanf`.

Main API	`format`	`scan`	`parse`
Extension point	`formatter`	`scanner`	`parser`
Parse format string	`formatter::parse`	`scanner::parse`	`parser::parse_format`?
Extension function	`formatter::format`	`scanner::scan`	`parser::parse`
Format string parse context	`format_parse_context`	`scan_parse_context`	`parse_parse_context`?
Context	`format_context`	`scan_context`	`parse_context`

§ 4. Existing work

[SCNLIB] is a C++ library that, among other things, provides a range-based `scan` interface similar to the one described in this paper. [FMT] has a prototype implementation of the proposal.

§ References

§ Informative References

[ATTR]
Common Function Attributes. URL: https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc/Common-Function-Attributes.html

[CODESEARCH]
Andrew Tomazos. Code search engine website. URL: https://codesearch.isocpp.org

[FMT]
Victor Zverovich et al. The fmt library. URL: https://github.com/fmtlib/fmt

[N4412]
Jens Maurer. N4412: Shortcomings of iostreams. URL: http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4412.html

[P0323]
JF Bastien; Vicente Botet. std::expected. URL: https://wg21.link/p0323

[P0355]
Howard E. Hinnant; Tomasz Kamiński. Extending <chrono> to Calendars and Time Zones. URL: https://wg21.link/p0355

[P0645]
Victor Zverovich. Text Formatting. URL: https://wg21.link/p0645

[P1361]
Victor Zverovich; Daniela Engert; Howard E. Hinnant. Integration of chrono with text formatting. URL: https://wg21.link/p1361

[P1433]
Hana Dusíková. Compile Time Regular Expressions. URL: https://wg21.link/p1433

[P1664]
JeanHeyd Meneide; Hannes Hauswedell. reconstructible_range - a concept for putting ranges back together. URL: https://wg21.link/p1664

[PARSE]
Python `parse` package. URL: https://pypi.org/project/parse/

[SCNLIB]
Elias Kosunen. scnlib: scanf for modern C++. URL: https://github.com/eliaskosunen/scnlib