A More Composable `from_chars`

Document #: D2584R1
Date: 2022-08-09
Programming Language C++
Audience: LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose an easier way to convert a sequence of characters to a number using `std::from_chars`. This paper is a follow-up to P2007R0 [2].

Tony table

Before

```
std::string s = "1.2.3.4";

auto ints =
s | std::views::split('.')
  | std::views::transform([](const auto & v){
    int i = 0;
    std::from_chars(std::to_address(v.begin()),
               std::to_address(v.end(), i);
    return i;
});
```

After

```
std::string s = "1.2.3.4";

auto ints =
s | std::views::split('.')
  | std::views::transform([](const auto & v) {
      return std::from_chars<int>(v).value_or(0);
});
```

This example was taken from Barry's Revzin blog post on the deficiencies of the old split view.

Revisions

R1

- Modify the `expected` base interface to inherit from `expected` such that the unparsed information is always preserved.
- Present the `expected` interface as the primary option preferred by the author.
- Add wording for the `expected` based option
- Fix typos and wording issues

Example

Motivation and design

We propose to add new `from_chars` overloads with the aim of simplifying the use of the interface and making it more composable.

Design using `std::expected`

We propose an interface returning an object inheriting from expected to make it easier to access the value, and to check for errors:

```
template <typename T>
struct from_chars_result_range : std::expected<T, std::errc> {
    std::span<const char> unparsed = {};
};

template <std::integral T>
constexpr from_chars_result_range<T> from_chars(span<const char> rng, int base = 10);

template <std::floating_point T>
from_chars_result_range<T>
from_chars(std::span<const char> rng, std::chars_format fmt = std::chars_format::general);
```

This interface is hard to misuse and would encourage checking for errors. It turns out to be pretty nice to use too. The one drawback is the reliance on the `expected` header.

Contrary to R0 and what was claimed previously, if one or more characters are matched, but the value is outside of the type bound, there is both an error and some characters parsed. So we need to inherit from `expected` to add the unparsed member in both the value and error case.

```
int main() {
    assert(from_chars<int>("123").value_or(0) == 123);
    assert(from_chars<int>("cafe", 16).value_or(0) == 0xcafe);
    assert(from_chars<int>("cafe").value_or(42) == 42);
```

```
        if(auto parsed = std::from_chars<int>("123!!"); parsed) {
            assert(*parsed == 123);
            assert(std::ranges::equal(parsed.unparsed, "!!"));
        }
    }
```

`from_chars` should take a range rather than a pair of pointers

As explained in P2007R0 [2], a correct use of `from_chars` with any kind of range call for

```
std::from_chars(std::to_address(std::ranges::begin(rng)), std::to_address(std::ranges::end(rng)), out);
```

This is because:

- The iterators may not be pointers
- The range may be contiguous but not sized (so `data(), data()+size()` isn't an option).

It's a lot of subtleties and verbosity for a relatively common interface.

`from_chars` should return its result by value

Having the converted value as part of the return type gives more opportunity for composition. For example, it allows patterns such as:

```
if(auto [value, ec, _]  = std::from_chars<int>(range); ec == std::errc()) {}
```

To achieve that, the proposed `from_chars` overloads take the desired output type as a template parameter and return a `from_chars_result_range` object.

`span` vs `string_view` vs `contiguous_range`

This proposal uses `span<const char>`. This is because P2499R0 [4], by making `string_view`'s string_view range constructor explicit, makes using it in contexts where we want to accept any range of char more tedious than it needs to be and less composable.

Ultimately, whether we choose `span<const char>` or `string_view` depends on whether we think the range case is more commone than the `const char*` use case.

Using `contiguous_range` over `span` has very little benefits. The proposed design uses span in its returned object anyway (to store the remaining range), so it would not save on headers inclusion, and `<span>` is a very small header anyway,

Header

During previous discussions, there were some concerns that this would impact compile times. In the meantime we:

- Made `from_chars` constexpr, leading to potentially bigger header

- Standardized header units and a `std` module.

`from_chars_result_range` is not comparable

The rationale to make `from_chars_result` (P1191R0 [3]) comparable is unclear, and it has been regarded as a bad move. Indeed, it is unclear what the invariant of `from_chars_result` is. We do, therefore, not propose to make the new `from_chars_result_range` type comparable, especially in the absence of good rationale.

But `from_chars` is intended as a low level interface!

`from_chars` is efficient, correct and usable portably. That doesn't mean it should be hard to use. The proposed interface doesn't make `from_chars` less usable, quite the contrary, and that's a good thing. It's not because a facility is ”low-level” that it should be gratuitously expert-friendly.

Alternative interface

The following design, which was presented as the primary option in R0 does not use expected and can be more easily used with structured binding. But, it encourages ignoring errors, and does not offer the same ergonomic benefits as the monadic interfaces of `expected`.

```
template <typename T>
struct from_chars_result_range {
    T value;
    std::errc ec;
    std::span<const char> unparsed;
};
template <integral T>
requires (!std::same_as<bool, T>)
constexpr from_chars_result_range<T> from_chars(std::span<const char> rng, int base = 10);

template <floating_point T>
from_chars_result_range<T> from_chars(std::span<const char> rng, chars_format fmt = chars_format::general);
```

Question for LEWG

- Do we like the general direction?
- Do we prefer the version with expected or the one without?

Implementation experience

The new overloads are specified to wrap the existing one, so this proposal presents no particular implementation complexity. The design using `std::expected` is demoed here.

The alternative design (without expected) is also on Compiler Explorer.

Wording (for the design with expected)

� Header <charconv> synopsis [charconv.syn]

```
namespace std {
    // ??, primitive numerical output conversion
    struct to_chars_result {
        char* ptr;
        errc ec;
        friend bool operator==(const to_chars_result&, const to_chars_result&) = default;
    };


    // ??, primitive numerical input conversion
    struct from_chars_result {
        const char* ptr;
        errc ec;
        friend bool operator==(const from_chars_result&, const from_chars_result&) = default;
    };

    template <typename T>
    struct from_chars_result_range : expected<T, errc> {
        std::span<const char> unparsed = {};

        constexpr from_chars_result_range(T value, span<const char> unparsed)  // exposition only
        noexcept
            : expected<T, errc>(value), unparsed(unparsed) {};
        constexpr from_chars_result_range(errc err, span<const char> unparsed)  // exposition only
        noexcept
            : expected<T, errc>(std::unexpect, err), unparsed(unparsed) {};

        template <typename U>
        bool operator==(const from_chars_result_range<U>&) = delete;
    };



    from_chars_result from_chars(const char* first, const char* last,
    see below& value, int base = 10);
    template <typename T>
    constexpr from_chars_result_range<T> from_chars(span<const char> rng, int base = 10)

    from_chars_result from_chars(const char* first, const char* last, float& value,
    chars_format fmt = chars_format::general);
    from_chars_result from_chars(const char* first, const char* last, double& value,
    chars_format fmt = chars_format::general);
    from_chars_result from_chars(const char* first, const char* last, long double& value,
    chars_format fmt = chars_format::general);
    template <typename T>
    from_chars_result_range<T> from_chars(span<const char> rng,
     chars_format fmt = chars_format::general);
}
```

The type `chars_format` is a bitmask type with elements `scientific`, `fixed`, and `hex`.

The types `to_chars_result`, `from_chars_result_range`, and `from_chars_result` have the data members and special members specified above. They have no base classes or members other than those specified.

� Primitive numeric input conversion [charconv.from.chars]

All functions named `from_chars` analyze the string `[first, last)` for a pattern, where `[first, last)` is required to be a valid range. If no characters match the pattern, `value` is unmodified, the member `ptr` of the return value is `first` and the member `ec` is equal to `errc::invalid_argument`. [*Note:* If the pattern allows for an optional sign, but the string has no digit characters following the sign, no characters match the pattern. — *end note*] Otherwise, the characters matching the pattern are interpreted as a representation of a value of the type of `value`. The member `ptr` of the return value points to the first character not matching the pattern, or has the value `last` if all characters match. If the parsed value is not in the range representable by the type of `value`, `value` is unmodified and the member `ec` of the return value is equal to `errc::result_out_of_range`. Otherwise, `value` is set to the parsed value, after rounding according to `round_to_nearest`, and the member `ec` is value-initialized.

```
from_chars_result from_chars(const char* first, const char* last,
see below& value, int base = 10);
```

Preconditions: `base` has a value between 2 and 36 (inclusive).

Effects: The pattern is the expected form of the subject sequence in the `"C"` locale for the given nonzero base, as described for `strtol`, except that no `"0x"` or `"0X"` prefix shall appear if the value of `base` is 16, and except that `'-'` is the only sign that may appear, and only if `value` has a signed type.

Throws: Nothing.

Remarks: The implementation shall provide overloads for all signed and unsigned integer types and `char` as the referenced type of the parameter `value`.

```
template <typename T>
constexpr from_chars_result_range<T>
from_chars(span<const char> rng, int base = 10)
```

Constraints: `T` models `integral` and `same_as<T, bool>` is `false`.

Preconditions: `base` has a value between 2 and 36 (inclusive).

Effects: Equivalent to

```
T out;
auto [ptr, ec] = from_chars(to_address(begin(rng)),
                            to_address(end(rng)), out, base);
auto subspan = rng.subspan(ptr - rng.data());
return ec != errc()
    ? from_chars_result_range<T>{ec,  subspan}
    : from_chars_result_range<T>{out, subspan};
```

Throws: Nothing.

```
from_chars_result from_chars(const char* first, const char* last, float& value,
chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, double& value,
chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, long double& value,
chars_format fmt = chars_format::general);
```

Preconditions: `fmt` has the value of one of the enumerators of `chars_format`.

Effects: The pattern is the expected form of the subject sequence in the `"C"` locale, as described for `strtod`, except that

- the sign `'+'` may only appear in the exponent part;
- if `fmt` has `chars_format::scientific` set but not `chars_format::fixed`, the otherwise optional exponent part shall appear;
- if `fmt` has `chars_format::fixed` set but not `chars_format::scientific`, the optional exponent part shall not appear; and
- if `fmt` is `chars_format::hex`, the prefix `"0x"` or `"0X"` is assumed. [*Example:* The string `0x123` is parsed to have the value `0` with remaining characters `x123`. — *end example*]

In any case, the resulting `value` is one of at most two floating-point values closest to the value of the string matching the pattern.

Throws: Nothing.

```
template <typename T>
from_chars_result_range<T> from_chars(span<const char> rng, chars_format fmt = chars_format::general);
```

Constraints: `T` models `floating_point`.

Preconditions: `fmt` has the value of one of the enumerators of `chars_format`.

Effects: Equivalent to

```
T out;
auto [ptr, ec] = from_chars(to_address(begin(rng)),
                            to_address(end(rng)), out, fmt);
auto subspan = rng.subspan(ptr - rng.data());
return ec != errc()
    ? from_chars_result_range<T>{ec,  subspan}
    : from_chars_result_range<T>{out, subspan};
```

Wording (for the design without expected)

� Header `<charconv>` synopsis [charconv.syn]

```
namespace std {
    // floating-point format for primitive numerical conversion
    enum class chars_format {
        scientific = unspecified,
        fixed = unspecified,
        hex = unspecified,
        general = fixed | scientific\textbf{}
    };

    // ??, primitive numerical output conversion
    struct to_chars_result {
        char* ptr;
        errc ec;
        friend bool operator==(const to_chars_result&, const to_chars_result&) = default;
    };

    to_chars_result to_chars(char* first, char* last, see below value, int base = 10);
    to_chars_result to_chars(char* first, char* last, bool value, int base = 10) = delete;

    to_chars_result to_chars(char* first, char* last, float value);
    to_chars_result to_chars(char* first, char* last, double value);
    to_chars_result to_chars(char* first, char* last, long double value);

    to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
    to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
    to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);

    to_chars_result to_chars(char* first, char* last, float value,
    chars_format fmt, int precision);
    to_chars_result to_chars(char* first, char* last, double value,
    chars_format fmt, int precision);
    to_chars_result to_chars(char* first, char* last, long double value,
    chars_format fmt, int precision);

    // ??, primitive numerical input conversion
    struct from_chars_result {
        const char* ptr;
        errc ec;
        friend bool operator==(const from_chars_result&, const from_chars_result&) = default;
    };

    template <integral T>
    struct from_chars_result_range {
        T value;
        errc ec;
        span<const char> unparsed;
     };


    from_chars_result from_chars(const char* first, const char* last,
    see below& value, int base = 10);
    template <typename T>
    constexpr from_chars_result_range<T> from_chars(span<const char> rng, int base = 10)

    from_chars_result from_chars(const char* first, const char* last, float& value,
    chars_format fmt = chars_format::general);
    from_chars_result from_chars(const char* first, const char* last, double& value,
    chars_format fmt = chars_format::general);
    from_chars_result from_chars(const char* first, const char* last, long double& value,
    chars_format fmt = chars_format::general);
    template <typename T>
    from_chars_result_range<T> from_chars(span<const char> rng,
                                          chars_format fmt = chars_format::general);
}
```

The type `chars_format` is a bitmask type with elements `scientific`, `fixed`, and `hex`.

The types `to_chars_result`, `from_chars_result_range`, and `from_chars_result` have the data members and special members specified above. They have no base classes or members other than those specified.

� Primitive numeric input conversion [charconv.from.chars]

All functions named `from_chars` analyze the string `[first, last)` for a pattern, where `[first, last)` is required to be a valid range. If no characters match the pattern, `value` is unmodified, the member `ptr` of the return value is `first` and the member `ec` is equal to `errc::invalid_argument`. [*Note:* If the pattern allows for an optional sign, but the string has no digit characters following the sign, no characters match the pattern. — *end note*] Otherwise, the characters matching the pattern are interpreted as a representation of a value of the type of `value`. The member `ptr` of the return value points to the first character not matching the pattern, or has the value `last` if all characters match. If the parsed value is not in the range representable by the type of `value`, `value` is unmodified and the member `ec` of the return value is equal to `errc::result_out_of_range`. Otherwise, `value` is set to the parsed value, after rounding according to `round_to_nearest`, and the member `ec` is value-initialized.

```
from_chars_result from_chars(const char* first, const char* last,
see below& value, int base = 10);
```

Preconditions: `base` has a value between 2 and 36 (inclusive).

Effects: The pattern is the expected form of the subject sequence in the `"C"` locale for the given nonzero base, as described for `strtol`, except that no `"0x"` or `"0X"` prefix shall appear if the value of `base` is 16, and except that `'-'` is the only sign that may appear, and only if `value` has a signed type.

Throws: Nothing.

Remarks: The implementation shall provide overloads for all signed and unsigned integer types and `char` as the referenced type of the parameter `value`.

```
template <typename T>
constexpr from_chars_result_range<T> from_chars(span<const char> rng, int base = 10);
```

Constraints: T models integral and same_as<T, bool> is false.

Preconditions: base has a value between 2 and 36 (inclusive).

Effects: Equivalent to

```
T out;
auto res = from_chars(to_address(rng.begin()), to_address(rng.end()), out, base);
return {out, res.ec, rng.subspan(res.ptr - rng.data())};
```

Throws: Nothing.

```
from_chars_result from_chars(const char* first, const char* last, float& value,
chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, double& value,
chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, long double& value,
chars_format fmt = chars_format::general);
```

Preconditions: fmt has the value of one of the enumerators of chars_format.

Effects: The pattern is the expected form of the subject sequence in the "C" locale, as described for strtod, except that

- the sign '+' may only appear in the exponent part;
- if fmt has chars_format::scientific set but not chars_format::fixed, the otherwise optional exponent part shall appear;
- if fmt has chars_format::fixed set but not chars_format::scientific, the optional exponent part shall not appear; and
- if fmt is chars_format::hex, the prefix "0x" or "0X" is assumed. [*Example:* The string 0x123 is parsed to have the value 0 with remaining characters x123. — *end example*]

In any case, the resulting value is one of at most two floating-point values closest to the value of the string matching the pattern.

Throws: Nothing.

```
template <typename T>
from_chars_result_range<T> from_chars(span<const char> rng, chars_format fmt = chars_format::general);
```

Constraints: T models floating_point.

Preconditions: fmt has the value of one of the enumerators of chars_format.

Effects: Equivalent to

```
T res;
auto [ptr, ec] = from_chars(to_address(rng.begin()), to_address(rng.end()), res, base);
return {res, ec, rng.subspan(ptr - rng.data())};
```

Feature test macro

[Editor's note: Bump the value of `__cpp_lib_to_chars` to the date of adoption in `charconv` and `version`]

Acknowledgments

Thanks to Mateusz Pusz for writing P2007R0 [2] which this paper is derived from. Thanks to Zhihao Yuan, Jeff Garland and others for helping me brainstorm these interfaces.

References

[1] Marc Mutz. P2218R0: More flexible optional::value_or(). `https://wg21.link/p2218r0`, 9 2020.

[2] Mateusz Pusz. P2007R0: 'std::from_chars' should work with 'std::string_view'. `https://wg21.link/p2007r0`, 1 2020.

[3] David Stone. P1191R0: Adding operator<=> to types that are not currently comparable. `https://wg21.link/p1191r0`, 8 2018.

[4] James Touton. P2499R0: string_view range constructor should be explicit. `https://wg21.link/p2499r0`, 12 2021.