Add Constexpr Modifiers to Functions to_chars and from_chars for Integral Types in <charconv> Header

Contents

1 Add Constexpr Modifiers to Functions to_chars and from_chars for Integral Types in <charconv> Header 2
  1.1 Introduction and Motivation ................................................. 2
  1.1.1 constexpr std::format and reflection .................................... 2
  1.1.2 No standard way to parse integer from string at compile-time .......... 2
  1.2 Design Decisions ................................................................. 3
  1.2.1 Testing ................................................................. 3
  1.2.2 Floating-point ......................................................... 3
  1.2.3 Other implementations ............................................... 3
  1.3 Conclusions ........................................................................... 3
  1.4 Proposed Changes relative to N4868 ....................................... 4
  1.4.1 Modifications to “20.19.1 Header <charconv> synopsis” [charconv.syn] ... 4
  1.4.2 Modifications to “20.19.2 Primitive numeric output conversion” [charconv.to.chars] ... 4
  1.4.3 Modifications to “20.19.3 Primitive numeric input conversion” [charconv.from.chars] ... 5
  1.4.4 Modifications to “17.3.2 Header <version> synopsis” [version.syn] ........ 5
  1.5 Revision History ................................................................. 5
  1.6 Acknowledgements .................................................................. 5
  1.7 References ............................................................................. 5
1 Add Constexpr Modifiers to Functions to_chars and from_chars for Integral Types in <charconv> Header

1.1 Introduction and Motivation

There is currently no standard way to make conversion between numbers and strings at compile time. std::to_chars and std::from_chars are fundamental blocks for parsing and formatting being locale-independent and non-throwing without memory allocation, so they look like natural candidates for constexpr string conversions. The paper proposes to make std::to_chars and std::from_chars functions for integral types usable in constexpr context.

Consider the simple example:

```cpp
constexpr std::optional<int> to_int(std::string_view s) {
    int value;
    if (auto [p, err] = std::from_chars(s.begin(), s.end(), value); err == std::errc()) {
        return value;
    } else {
        return std::nullopt;
    }
}
static_assert(to_int("42") == 42);
static_assert(to_int("foo") == std::nullopt);
```

We do not propose constexpr for floating-point overloads, see design choices below.

1.1.1 constexpr std::format and reflection

In C++20 constexpr std::string was adopted, so we can already build strings at compile-time:

```cpp
static_assert(std::string(Hello, "") + "world" + "!", world) == "Hello, world";
```

In addition, std::format was also adopted in C++20 and now its original author actively proposes various improvements like P2216 for compile-time format string checking. The current proposal is another step towards fully constexpr std::format which implies not only format string checking but also compile-time formatting (the only non-constexpr dependency of std::format is <charconv>):

```cpp
static_assert(std::format("Hello, C++{}!", 23) == "Hello, C++23!");
```

This can be very useful in context of reflection, i.e. to generate unique member names:

```cpp
// constexpr function
for (std::size_t i = 0; i < sizeof...(Ts); i++) {
    std::string member_name = std::format("member_{}", i);
}
```

1.1.2 No standard way to parse integer from string at compile-time

There are too many ways to convert string-like object to number - atol, sscanf, stoi, strtol, istream and the best C++17 alternative - from_chars. However, none of them are constexpr. This leads to numerous hand-made constexpr int parse_int(const char* str) or template <char...> constexpr int operator"" _foo() in various libraries:

- boost::multiprecision and similar examples with constexpr user-defined literals for my-big-integer-type construction at compile-time.
- boost::metaparse — yet another template <> struct digit_to_int_c<0> : boost::mpl::int_<0> {};

2
— lexy — parser combinator library with manually written constexpr std::from_chars equivalent for integers (any radix, overflow checks).
— ctre (compile time regular expressions) — number parsing is an important part of regex pattern processing (ctre::pcre_actions::hexdec).

1.2 Design Decisions

The discussion is based on the implementation of to_chars and from_chars from Microsoft/STL, because it has full support of <charconv>.

During testing, the following changes were made to the original algorithm to make the implementation possible:

— Add constexpr modifiers to all functions
— Replace internal assert-like macro with simple assert (_Adl_verify_range, _STL_ASSERT, _STL_INTERNAL_CHECK)
— Replace static constexpr variables inside function scope with constexpr
— Replace std::memcpy, std::memmove, std::memset with constexpr equivalents: third_party::trivial_copy, third_party::trivial_move, third_party::trivial_fill. To keep performance in a real implementation, one should use std::is_constant_evaluated

1.2.1 Testing

All the corresponding tests were constexprified and checked at compile-time and run-time. The modified version passes full set tests from Microsoft/STL test.

1.2.2 Floating-point

std::from_chars/std::to_chars are probably the most difficult to implement parts of a standard library. As of January 2021, only one of the three major implementations has full support of P0067R5:

<table>
<thead>
<tr>
<th>Vendor</th>
<th>&lt;charconv&gt; support (according to cppreference.com)</th>
</tr>
</thead>
<tbody>
<tr>
<td>libstdc++</td>
<td>no floating-point std::to_chars</td>
</tr>
<tr>
<td>libc++</td>
<td>no floating-point std::from_chars/std::to_chars</td>
</tr>
<tr>
<td>MS STL</td>
<td>full support</td>
</tr>
</tbody>
</table>

So at least for now we don’t propose constexpr for floating-point overloads.

1.2.3 Other implementations

Check of implementation libc++, the following changes were made to the original algorithm to make the implementation possible:

— Move utils functions from charconv.cpp to charconv header
— Replace std::memcpy, std::memmove with constexpr equivalents: third_party::trivial_copy, third_party::trivial_move or bit_cast
— Replace std::log2f with constexpr equivalents

Quick check of implementation libstdc++, showed that there are no blocking changes for implementation either.

1.3 Conclusions

to_chars and from_chars are basic building blocks for string conversions, so marking them constexpr provides a standard way for compile-time parsing and formatting. An implementation might prefer to provide this functionality via a compiler intrinsic, for performance and to reduce header bloat.
1.4 Proposed Changes relative to N4868

All the additions to the Standard are marked with green.

1.4.1 Modifications to “20.19.1 Header <charconv> synopsis” [charconv.syn]

```cpp
constexpr 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);
```

1.4.2 Modifications to “20.19.2 Primitive numeric output conversion” [charconv.to.chars]

```cpp
constexpr from_chars_result from_chars(const char* first, const char* last, see below & value, 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);
```
1.4.3 Modifications to “20.19.3 Primitive numeric input conversion” [charconv.from.chars]

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

1.4.4 Modifications to “17.3.2 Header <version> synopsis” [version.syn]

```cpp
+ #define __cpp_lib_constexpr_charconv _DATE OF ADOPTION_ // also in <charconv>
```

1.5 Revision History

Revision 3:
— Add a variant of implementation via compiler intrinsic.

Revision 2:
— Add missing modifications to [charconv.to.chars]/[charconv.from.chars]
— Add missing comment to feature-test macro

Revision 1:
— Update the wording relative to [N4868]
— Used __cpp_lib_constexpr_charconv as feature macro

Revision 0:
— Initial proposal
— Mailing list review Summary
   — No implementation concerns for libstdc++, should be possible for libc++ too
   — Please put the wording in code font
   — Use __cpp_lib_constexpr_charconv as feature macro

1.6 Acknowledgements

Thanks to Antony Polukhin for reviewing the paper and providing valuable feedback.

1.7 References

— Microsoft’s C++ Standard Library https://github.com/microsoft/STL, commit 2b4cf99c044176637497518294281046439a1bcc
— Proof of concept for to_chars and from_chars functions for integral types https://github.com/Neargye/charconv-constexpr-proposal/tree/integral