Abstract

This paper is meant to propose an addition to the standard library. New compile time functions indicating how many characters can be produced when formatting any possible value of a given integer or floating point type.

Example question: How many characters can be produced when formatting a value of type `int`, with base `10`?

Possible answer: On an architecture that uses two's completion for signed integers, `sizeof(int) == 8`, `CHAR_BIT == 8` -> the answer is `20`. `19` digits and possibly a minus sign character.

Motivation

With `<char_conv>` it is easy to convert values to characters without dynamic memory allocation. E.g. create a buffer 'on the stack' and use it. But, it is not so easy to know how big the buffer has to be. If one wants to calculate the exact result, there are lots of factors to consider: `sizeof(T)`, `is_signed`, `CHAR_BIT`, signed integers representation, floating-point arithmetic and more.

It is common to create a big buffer, just to be sure the value fits there. This could be avoided with the help of the standard library and the compiler. All the necessary information is known at compile time and when you know it, it is fairly easy to calculate the result.

Available options for creating the buffer

First and the most obvious option is to create a big buffer and don't care. Again, how big the buffer has to be? `64`, `128`? What about floating points? `std::numeric_limits<>::lowest` of `float` takes `40` characters to format, `double` - `310` and of `long double` - `4934` (considering `std::numeric_limits<>::is_iec559 == true` and calling `std::to_chars()` with `std::chars_format::fixed`).

`<limits>` provides lots of information. There is `numeric_limits<>::digits10` which is not quite what we want and is bound to base `10`. There is `numeric_limits<>::digits` which can be used, but is not convenient - `is_signed` has to be used and representation of signed numbers has to be assumed. Floating-point numbers are complicated too.

`std::formatted_size("{}", val)` could be used, but it is not available at compile time and generally seems like an overkill for getting the result.

Example of proposed API

I propose API similar to the one from `std::to_chars()`. A constexpr function that instead of taking a value, takes the type as a template parameter.

```
// For integers
constexpr std::size_t std::max_format_size<T>(int base = 10);

// For floating-points
constexpr std::size_t
std::max_format_size<float/double/long double>(std::chars_format fmt);

constexpr std::size_t
std::max_format_size<float/double/long double>(std::chars_format fmt,
                                               int precision);
```

Example usage

Formatting an `int` value as hex characers.

```
int value = get_some_value();

constexpr std::size_t k_buf_size = std::max_format_size<int>(16);
char buffer[k_buf_size + 1u]; // 1u for '\0'

auto [ptr, _] = std::to_chars(buffer, buffer + k_buf_size, value, 16);
*ptr = '\0';

std::cout << buffer;
```

Where to put the functions

The `<limits>` header seems like a good candidate. But, I think it should be available in `<charconv>`, as the functions are going to be used along with `std::to_chars()` very often.