Elementary string conversions

This lessons discusses the shortcomings of the string conversion functions present in older versions of C++ and the new more efficient string manipulation functions introduced in C++ 17.

WE'LL COVER THE FOLLOWING

The growing number of data formats like JSON or XML require efficient string processing and manipulation. The maximum performance is especially crucial when such data formats are used to communicate over the network, where high throughput is the critical factor.

For example, you get the characters in a network packet, you deserialise it (convert strings into numbers), then process the data, and finally, it's serialised back to the same file format (numbers into strings) and sent over the network as a response.

The Standard Library had bad luck in those areas. It's usually perceived to be too slow for such advanced string processing. Often developers prefer custom solutions or third-party libraries.

The situation might change as with C++17 we get two sets of functions: `from_chars` and `to_chars` that allow for low-level string conversions.

Current Solutions Summaries Table

Facility	Shortcomings
`sprintf`	format string, locale, buffer overrun

	overrun
`snprintf`	format string, locale
`sscanf`	format string, locale
`atol`	locale, does not signal errors
`strtol`	locale, ignores whitespace and 0x prefix
`strstream`	locale, ignores whitespace
`stringstream`	locale, ignores whitespace, memory allocation
`num_put` / `num_get` facets	locale, virtual function
`to_string`	locale, memory allocation
`stoi` etc.	locale, memory allocation, ignores whitespace and 0x prefix, exceptions

Shortcomings

As you can see above, sometimes converting functions do too much work, which makes the whole processing slower. Moreover, often there's no need for the extra features.

First of all, all of them use **"locale"**. So even if you work with universal strings, you have to pay a little price for localization support. For example, if you parse numbers from XML or JSON, there's no need to apply current system language, as those formats are interchangeable.

The next issue is error reporting. Some functions might throw an exception, some of them return just a basic value. Exceptions might not only be costly (as throwing might involve extra memory allocations), but often a parsing error

is not an exceptional situation - usually, an application is prepared for such

cases. Returning a simple value - for example, 0 for `atoi`, 0.0 for `atof` is also limited, as in that case you don't know if the parsing was successful or not.

The third topic, especially related to C-style API, is that you have to provide some form of the "format string". Parsing such string might involve some additional cost.

Another thing is "empty space" support. Functions like `strtol` or `stringstream` might skip empty spaces at the beginning of the string. That might be handy, but sometimes you don't want to pay for that extra feature.

There's also another critical factor: safety. Simple functions don't offer any buffer overrun solutions, and also they work only on null-terminated strings. In that case, you cannot use `string_view` to pass the data.

C++17 API and New Functions

> The new C++17 API addresses all of the above issues. Rather than providing many functionalities, they focus on giving very low-level support. That way, you can have the maximum speed and tailor them to your needs.

The new functions are guaranteed to be:

• non-throwing - in case of some error they won't throw any exception (as `stoi`)

• non-allocating - the whole processing is done in place, without any extra memory allocation

• no locale support - the string is parsed universally as if used with default ("C") locale

• memory safety - input and output range are specified to allow for buffer overrun checks

• no need to pass string formats of the numbers

• error reporting additional information about the conversion outcome

* error reporting additional information about the conversion outcome

All in all, with C++17, you have two sets of functions:

- `from_chars` - for conversion from strings into numbers, integer and floating points.
- `to_chars` - for converting numbers into string.

Let's have a look at the functions in a bit more detail.