Examples

Below we take a look at the floating and integral type string conversions!

WE'LL COVER THE FOLLOWING ^

Here are two examples of how to convert a string into a number using `from_chars`. The first one will convert into `int` and the second one converts into a floating-point number.

Integral types

```cpp
#include <iostream>
#include <charconv> // from_chars, to_chars
#include <string>

int main()
{
    const std::string str { "12345678901234" };
    int value = 0;
    const auto res = std::from_chars(str.data(),
                                     str.data() + str.size(),
                                     value);

    if (res.ec == std::errc())
    {
        std::cout << "value: " << value << ", distance: " << res.ptr - str.data() << '\n';
    }
    else if (res.ec == std::errc::invalid_argument)
    {
        std::cout << "invalid argument!\n";
    }
    else if (res.ec == std::errc::result_out_of_range)
    {
        std::cout << "out of range! res.ptr distance: " << res.ptr - str.data() << '\n';
    }
}
```

The example is straightforward, it passes a string `str` into `from_chars` and then displays the result with additional information if possible.

Below you can find an output for various `str` value.

`str` value	Output
12345	value: 12345, distance 5
12345678901234	out of range! res.ptr distance: 14
hfhfyt	invalid argument

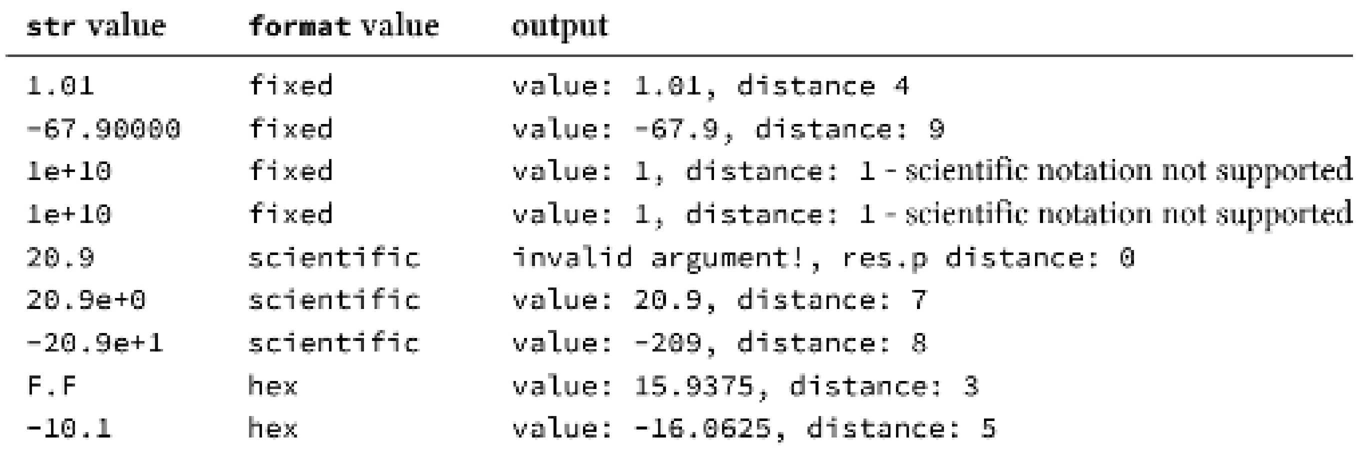
In the case of `12345678901234`, the conversion routine could parse the number (all 14 characters were checked), but it's too large to fit in `int` thus we got `out_of_range`.

Floating Point

To get the floating-point test, we can replace the top lines of the previous example with the highlighted code below:

```
const std::string str {
    "12345678901234"
};
double value = 0.0;
const auto format = std::chars_format::general;
const auto res = std::from_chars(str.data(),
                                 str.data() + str.size(),
                                 value,
                                 format);
```

Example output:



str value	format value	output
1.01	fixed	value: 1.01, distance 4
-67.90000	fixed	value: -67.9, distance: 9
1e+10	fixed	value: 1, distance: 1 - scientific notation not supported
1e+10	fixed	value: 1, distance: 1 - scientific notation not supported
20.9	scientific	invalid argument!, res.p distance: 0
20.9e+0	scientific	value: 20.9, distance: 7
-20.9e+1	scientific	value: -209, distance: 8
F.F	hex	value: 15.9375, distance: 3
-10.1	hex	value: -16.0625, distance: 5

The main difference is the last parameter: `format`.

Here’s the example output that we get:

`str` value	`format` value	output
`1.01`	`fixed`	`value: 1.01, distance 4`
`-67.90000`	`fixed`	`value: -67.9, distance: 9`
`1e+10`	`fixed`	`value: 1, distance: 1` - scientific notation not supported
`1e+10`	`fixed`	`value: 1, distance: 1` - scientific notation not supported
`20.9`	`scientific`	`invalid argument!, res.p distance: 0`
`20.9e+0`	`scientific`	`value: 20.9, distance: 7`

`-20.9e+1`	`scientific`	`value: -209,` `distance: 8`
`F.F`	`hex`	`value: 15.9375,` `distance: 3`
`-10.1`	`hex`	`value: -16.0625,` `distance: 5`

The `general` format is a combination of `fixed` and `scientific` so it handles regular floating-point string with the additional support for `e+num` syntax.

You have a basic understanding of converting from strings to numbers, so let’s have a look at how to do the opposite way in the next lesson.