The benchmark

This section introduces a benchmark that measures performance of from_chars and to_chars against other conversion methods.

WE'LL COVER THE FOLLOWING

- How does the benchmark work
- Code for from_chars/to_chars:

How does the benchmark work

- Generates a vector of random integers of the size VECSIZE.
- Each pair of conversion methods will transform the input vector of integers into a vector of strings and then back to another vector of integers. This round-trip will be verified so that the output vector is the same as the input vector.
- The conversion is performed ITER times.
- Errors from the conversion functions are not checked.
- The code tests:
  - `from_char` / `to_chars`
  - `to_string` / `stoi`
  - `sprintf` / `atoi`
  - `ostringstream` / `istringstream`

Code for from_chars/to_chars:

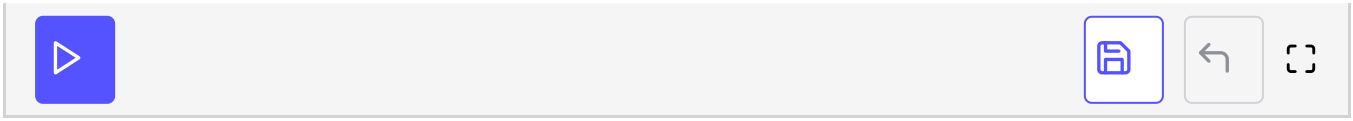
```
#include <algorithm>
#include <charconv>
#include <chrono>
#include <cstdio>
#include <iostream>
#include <random>
#include <sstream>
#include <string>
```

```
#include <string_view>
#include <vector>


/**
 * Call doNotOptimizeAway(var) against variables that you use for
 * benchmarking but otherwise are useless. The compiler tends to do a
 * good job at eliminating unused variables, and this function fools
 * it into thinking var is in fact needed.
 */
#ifdef _MSC_VER

#pragma optimize("", off)

template <class T>
void DoNotOptimizeAway(T&& datum) {
    datum = datum;
}

#pragma optimize("", on)

#elif defined(__clang__)

template <class T>
__attribute__((__optnone__)) void DoNotOptimizeAway(T&& /* datum */) {}

#else

template <class T>
void DoNotOptimizeAway(T&& datum) {
    asm volatile("" : "+r" (datum));
}

#endif

template <typename TFunc> void RunAndMeasure(const char* title, TFunc func)
{
    auto ret = func();
    ret = func();
    DoNotOptimizeAway(ret);
    const auto start = std::chrono::steady_clock::now();
    ret = func();
    const auto end = std::chrono::steady_clock::now();
    DoNotOptimizeAway(ret);
    std::cout << title << ": " << std::chrono::duration <double, std::milli>(end - start).cou
}

std::vector<int> GenRandVecOfNumbers(size_t count)
{
    std::vector<int> out(count);

    std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_int_distribution<int> dist(std::numeric_limits<int>::min(), std::numeric_lim

    std::generate_n(std::begin(out), count, [&dist, &rng] {return dist(rng); });

    return out;
}

void CheckVectors(const std::vector<int>& a, const std::vector<int>& b)
{
    if (a.size() != b.size())
```

```
    {
        std::cout << "wrong size!\n";
        return;
    }

    for (size_t i = 0; i < a.size(); ++i)
    {
        if (a[i] != b[i])
            std::cout << "error! " << i << " " << a[i] << " !=  " << b[i] << '\n';
    }
}

void Benchmark(size_t ITERS, size_t vecSize)
{
    const auto numIntVec = GenRandVecOfNumbers(vecSize);
    std::vector<std::string> numStrVec(numIntVec.size());
    std::vector<int> numBackIntVec(numIntVec.size());

    std::string strTemp(15, ' ');

    //
    // from_chars/to_chars
    //

    RunAndMeasure("to_chars", [&]() {
        for (size_t iter = 0; iter < ITERS; ++iter)
        {
            for (size_t i = 0; i < numIntVec.size(); ++i)
            {
                const auto res = std::to_chars(strTemp.data(), strTemp.data() + strTemp.size(
                numStrVec[i] = std::string_view(strTemp.data(), res.ptr - strTemp.data());
            }
            DoNotOptimizeAway(numStrVec.data());
        }
        return numStrVec.size();
    });

    RunAndMeasure("from_chars", [&]() {
        for (size_t iter = 0; iter < ITERS; ++iter)
        {
            for (size_t i = 0; i < numStrVec.size(); ++i)
            {
                const auto &str{ numStrVec[i] };
                std::from_chars(str.data(), str.data() + str.size(), numBackIntVec[i]);
            }
            DoNotOptimizeAway(numBackIntVec.data());
        }
        return numBackIntVec.size();
    });

    CheckVectors(numIntVec, numBackIntVec);

    //
    // to_string / stoi
    //

    RunAndMeasure("to_string", [&]() {
        for (size_t iter = 0; iter < ITERS; ++iter)
        {
            for (size_t i = 0; i < numStrVec.size(); ++i)
                numStrVec[i] = std::to_string(numIntVec[i]);
```

```
        DoNotOptimizeAway(numStrVec.data());
    }
    return numStrVec.size();
});

RunAndMeasure("stoi", [&]() {
    for (size_t iter = 0; iter < ITERS; ++iter)
    {
        for (size_t i = 0; i < numStrVec.size(); ++i)
            numBackIntVec[i] = std::stoi(numStrVec[i]);

        DoNotOptimizeAway(numBackIntVec.data());
    }
    return numBackIntVec.size();
});

CheckVectors(numIntVec, numBackIntVec);

//
// sprintf / atoi
//

RunAndMeasure("sprintf", [&]() {
    for (size_t iter = 0; iter < ITERS; ++iter)
    {
        for (size_t i = 0; i < numIntVec.size(); ++i)
        {
            auto res = snprintf(strTemp.data(), 15, "%d", numIntVec[i]);
            numStrVec[i] = std::string_view(strTemp.data(), (strTemp.data() + res) - strT
        }
        DoNotOptimizeAway(numStrVec.data());
    }
    return numStrVec.size();
});

RunAndMeasure("atoi", [&]() {
    for (size_t iter = 0; iter < ITERS; ++iter)
    {
        for (size_t i = 0; i < numStrVec.size(); ++i)
            numBackIntVec[i] = atoi(numStrVec[i].c_str());

        DoNotOptimizeAway(numBackIntVec.data());
    }
    return numBackIntVec.size();
});

// ostringstream / istringstream

RunAndMeasure("otringstream", [&]() {
    for (size_t iter = 0; iter < ITERS; ++iter)
    {
        for (size_t i = 0; i < numStrVec.size(); ++i)
        {
            std::ostringstream ss;
            ss << numIntVec[i];
            numStrVec[i] = ss.str();
        }
        DoNotOptimizeAway(numStrVec.data());
    }
    return numStrVec.size();
});
```

```
    RunAndMeasure("stringstream", [&]() {
        for (size_t iter = 0; iter < ITERS; ++iter)
        {
            for (size_t i = 0; i < numStrVec.size(); ++i)
            {
                std::istringstream ss(numStrVec[i]);
                ss >> numBackIntVec[i];
            }
            DoNotOptimizeAway(numBackIntVec.data());
        }
        return numBackIntVec.size();
    });
}

int main(int argc, const char** argv)
{
    const size_t ITERS = argc > 1 ? atoi(argv[1]) : 1000;
    std::cout << "test iterations: " << ITERS << '\n';
    const size_t VECSIZE = argc > 2 ? atoi(argv[2]) : 1000;
    std::cout << "vector size: " << VECSIZE << '\n';

    Benchmark(ITERS, VECSIZE);
}
```



`CheckVectors` - checks if the two input vectors of integers contain the same values, and prints mismatches on error.

> The benchmark converts `vector<int>` into `vector<string>` and we measure the whole conversion process which also includes the string object creation.

Here are the results (time in milliseconds) of running 1000 iterations on a vector with 1000 elements:

Method	GCC 8.2	Clang 7.0 Win	VS 2017 15.8 x64
`to_chars`	21.94	18.15	24.81
`from_chars`	15.96	12.74	13.43
`to_string`	61.84	16.62	20.91

`stoi`	70.81	45.75	42.40
`sprintf`	56.85	124.72	131.03
`atoi`	35.90	34.81	32.50
`ostringstream`	264.29	681.29	575.95
`stringstream`	306.17	789.04	664.90

The machine: Windows 10 x64, i7 8700 3.2 GHz base frequency, 6 cores/12 threads (although the benchmark uses only one thread for processing).

- GCC 8.2 - compiled with `-O2 -Wall -pedantic`, MinGW Distro
- Clang 7.0 - compiled with `-O2 -Wall -pedantic`, Clang For Windows
- Visual Studio 2017 15.8 - Release mode, x64

Some notes:

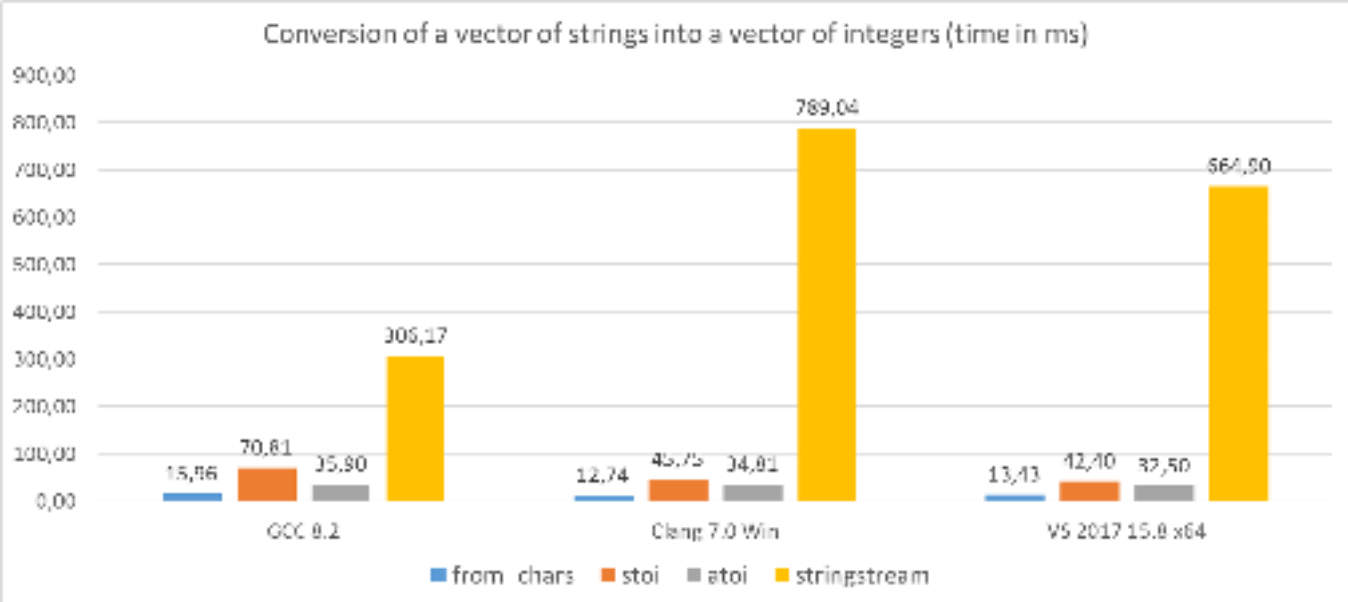
- On GCC `to_chars` is almost 3x faster than `to_string`, 2.6x faster than `sprintf` and 12x faster than `ostringstream`!
- On Clang `to_chars` is a bit slower than `to_string`, but ~7x faster than `sprintf` and surprisingly almost 40x faster than `ostringstream`!
- MSVC also has slower performance in comparison with `to_string`, but then `to_chars` is ~5x faster than `sprintf` and ~23x faster than `ostringstream`.

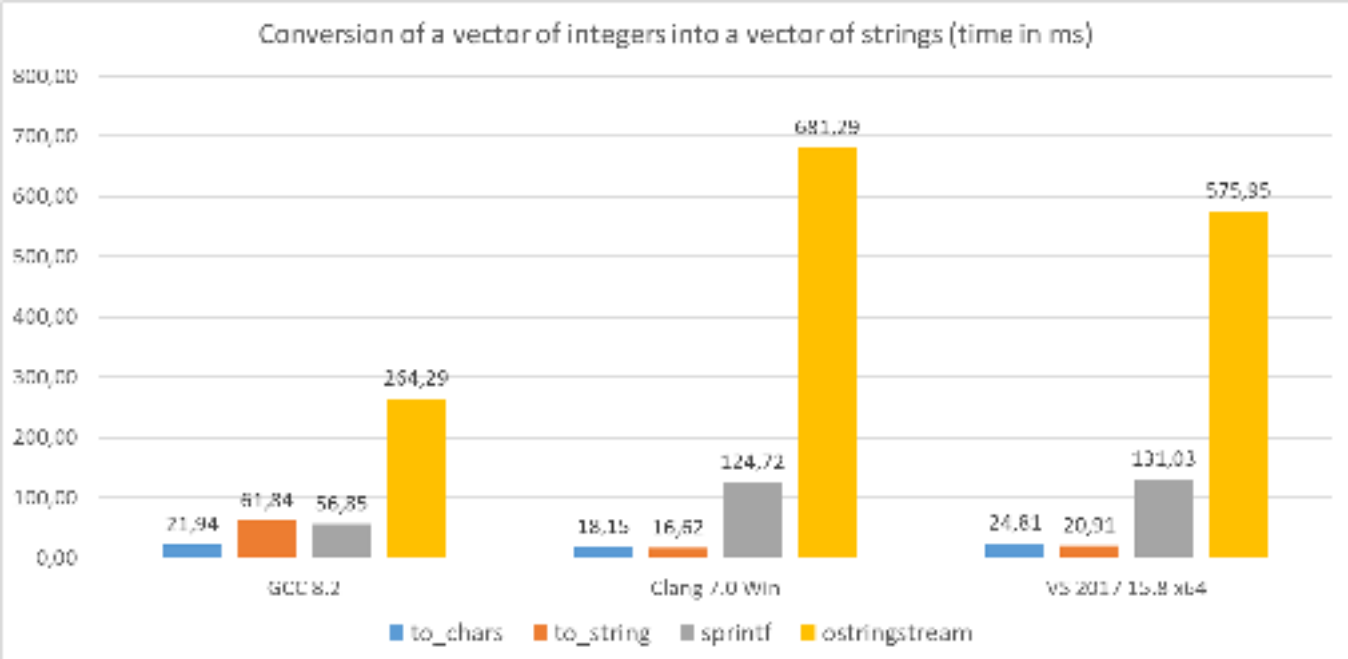
Looking now at `from_chars`:

- On GCC it's ~4,5x faster than `stoi`, 2,2x faster than `atoi` and almost 20x faster than `istringstream`.
- On Clang it's ~3,5x faster than `stoi`, 2.7x faster than `atoi` and 60x faster than `istringstream`!
- MSVC performs ~3x faster than `stoi`, ~2,5x faster than `atoi` and almost 50x faster than `istringstream`!

As mentioned earlier, the benchmark also includes the cost of string object creation. That's why `to_string` (optimized for strings) might perform a bit better than `to_chars`. If you already have a char buffer, and you don't need to create a string object, then `to_chars` should be faster.

Here are the two charts built from the table above:





> It's encouraged to run the benchmarks on your own before you make the final judgment. You might get different results in your environment, where maybe a different compiler or STL library implementation is available.

Now that you're done learning the last of the concepts this chapter

introduced. The next lesson will provide you with a small summary to refresh all your concepts.