The Serial Version

Let's take a look at the serial implementation of the CSV application from the previous lesson.

As the first step, we'll cover a serial version of the application.

Here is the implementation:

input.cpp

files/cpu.csv

files/book.csv

scope_timer.h

```
#include <algorithm>
#include <charconv>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <optional>
#include <string_view>
#include <string>
#include <utility>
#include <vector>

#include "scope_timer.h"

static const char* const CSV_EXTENSION = ".csv";
static constexpr char DEFAULT_DATE_DELIM = '-';
static constexpr char CSV_DELIM = ';';

namespace fs = std::filesystem;

[[nodiscard]] std::string GetFileContents(const fs::path& filePath)
{
    std::ifstream inFile{ filePath, std::ios::in | std::ios::binary };
    if (!inFile)
        throw std::runtime_error("Cannot open " + filePath.filename().string());

    std::string str(static_cast<size_t>(fs::file_size(filePath)), 0);

    inFile.read(str.data(), str.size());
    if (!inFile)
```

```
        throw std::runtime_error("Could not read the full contents from " + filePath.filenar

    return str;

}

[[nodiscard]] std::vector<std::string_view> SplitString(std::string_view str, char delim)
{
    std::vector<std::string_view> output;

    const auto last = str.end();
    for (auto first = str.begin(), second = str.begin(); second != last && first != last; fi
    {
        second = std::find(first, last, delim);

        // we might get empty string views here, but that's ok in the case of CSV reader
        //output.emplace_back(str.substr(std::distance(str.begin(), first), std::distance(fi
        output.emplace_back(&*first, std::distance(first, second));

        if (second == last)
            break;
    }

    return output;
}

[[nodiscard]] std::vector<std::string_view> SplitLines(std::string_view str)
{
    auto lines = SplitString(str, '\n');

    if (!lines.empty() && lines[0].back() == '\r') // Windows CR conversion
    {
        for (auto &&line : lines)
        {
            if (line.back() == '\r')
                line.remove_suffix(1);
        }
    }

    return lines;
}

template<typename T>
[[nodiscard]] std::optional<T> TryConvert(std::string_view sv) noexcept
{
    T value{ };
    const auto last = sv.data() + sv.size();
    const auto res = std::from_chars(sv.data(), last, value);
    if (res.ec == std::errc{} && res.ptr == last)
        return value;

    return std::nullopt;
}

// as of Dec 2018 only MAVC supports floating point conversion
// this code might be removed when GCC/Clang handles it as well
#ifndef _MSC_VER
template<>
[[nodiscard]] std::optional<double> TryConvert(std::string_view sv) noexcept
{
    std::string str{ sv };
    return ::atof(str.c_str());
}
```

```
#endif

class Date
{
public:
    enum class Format { DayMonthYear, YearMonthDay };
public:
    Date() = default;
    Date(uint8_t day, uint8_t month, uint16_t year) noexcept : mDay(day), mMonth(month), mYe
    explicit Date(std::string_view sv, char delim = DEFAULT_DATE_DELIM, Format fmt = Format

    bool IsInvalid() const noexcept {
        return mDay == 0 || mMonth == 0 || mYear == 0 || mDay > 31 || mMonth > 12;
    }

    friend bool operator < (const Date& lhs, const Date& rhs) noexcept {
        return std::tie(lhs.mYear, lhs.mMonth, lhs.mDay) < std::tie(rhs.mYear, rhs.mMonth, r
    }

    friend bool operator <= (const Date& lhs, const Date& rhs) noexcept {
        return std::tie(lhs.mYear, lhs.mMonth, lhs.mDay) <= std::tie(rhs.mYear, rhs.mMonth,
    }

    friend std::ostream& operator<<(std::ostream &os, const Date &d) noexcept {
        return os << d.mDay << DEFAULT_DATE_DELIM << d.mMonth << DEFAULT_DATE_DELIM << d.mYe
    }

private:
    uint8_t mDay{ 0 }; // 1...31, 0 is invalid
    uint8_t mMonth{ 0 }; // 1...12, 0 is invalid
    uint16_t mYear{ 0 }; // 0 means invalid
};

Date::Date(std::string_view sv, char delim, Date::Format fmt)
{
    const auto columns = SplitString(sv, delim);
    if (columns.size() == 3)
    {
        mDay = TryConvert<uint8_t>(columns[fmt == Format::DayMonthYear ? 0 : 2]).value_or(0)
        mMonth = TryConvert<uint8_t>(columns[1]).value_or(0);
        mYear = TryConvert<uint16_t>(columns[fmt == Format::DayMonthYear ? 2 : 0]).value_or(

        if (IsInvalid())
            throw std::runtime_error("Cannot convert date from " + std::string(sv));
    }
    else
        throw std::runtime_error("Cannot convert date, wrong element count/format, " + std::
}


class OrderRecord
{
public:
    OrderRecord() = default;
    OrderRecord(Date date, std::string coupon, double unitPrice, double discount, unsigned i
        : mDate(date)
        , mCouponCode(std::move(coupon))
        , mUnitPrice(unitPrice)
        , mDiscount(discount)
        , mQuantity(quantity)
    { }
```

```
    double CalcRecordPrice() const noexcept { return mQuantity * (mUnitPrice*(1.0 - mDiscour
    bool CheckDate(const Date& startDate, const Date& endDate) const noexcept {
        return (startDate.IsInvalid() || startDate <= mDate) && (endDate.IsInvalid() || mDat
    }

public:
    // not enum class so that we can easily use it as array index
    enum Indices { DATE, COUPON, UNIT_PRICE, DISCOUNT, QUANTITY, ENUM_LENGTH };

private:
    Date mDate;
    std::string mCouponCode;
    double mUnitPrice{ 0.0 };
    double mDiscount{ 0.0 }; // 0... 1.0
    unsigned int mQuantity{ 0 };
};

[[nodiscard]] OrderRecord LineToRecord(std::string_view sv)
{
    const auto columns = SplitString(sv, CSV_DELIM);
    if (columns.size() == static_cast<size_t>(OrderRecord::ENUM_LENGTH)) // assuming we also
    {
        const auto unitPrice = TryConvert<double>(columns[OrderRecord::UNIT_PRICE]);
        const auto discount = TryConvert<double>(columns[OrderRecord::DISCOUNT]);
        const auto quantity = TryConvert<unsigned int>(columns[OrderRecord::QUANTITY]);

        if (unitPrice && discount && quantity)
        {
            return { Date(columns[OrderRecord::DATE]),
                     std::string(columns[OrderRecord::COUPON]),
                     *unitPrice,
                     *discount,
                     *quantity };
        }
    }
    throw std::runtime_error("Cannot convert Record from " + std::string(sv));
}

[[nodiscard]] std::vector<OrderRecord> LinesToRecords(const std::vector<std::string_view>& l
{
    //ScopeTimer _t(__func__);

    std::vector<OrderRecord> outRecords;
    std::transform(lines.begin(), lines.end(), std::back_inserter(outRecords), LineToRecord);

    return outRecords;
}

[[nodiscard]] std::vector<OrderRecord> LoadRecords(const fs::path& filename)
{
    //ScopeTimer _t(__func__);
    const auto content = GetFileContents(filename);

    ScopeTimer _t("Parsing Strings", /*store*/true);
    const auto lines = SplitLines(content);

    return LinesToRecords(lines);
}

[[nodiscard]] double CalcTotalOrder(const std::vector<OrderRecord>& records, const Date& sta
{
    ScopeTimer _t(__func__, /*store*/true);
```

```
    return std::accumulate(std::begin(records), std::end(records), 0.0,
        [&startDate, &endDate](double val, const OrderRecord& rec) {
            if (rec.CheckDate(startDate, endDate))
                return val + rec.CalcRecordPrice();
            else
                return val;
        }
    );
}

bool IsCSVFile(const fs::path &p)
{
    return fs::is_regular_file(p) && p.extension() == CSV_EXTENSION;
}

[[nodiscard]] std::vector<fs::path> CollectPaths(const fs::path& startPath)
{
    std::vector<fs::path> paths;
    fs::directory_iterator dirpos{ startPath };
    std::copy_if(fs::begin(dirpos), fs::end(dirpos), std::back_inserter(paths), IsCSVFile);
    return paths;
}

struct Result
{
    std::string mFilename;
    double mSum{ 0.0 };
};

[[nodiscard]] std::vector<Result>
CalcResults(const std::vector<fs::path>& paths, Date startDate, Date endDate)
{
    ScopeTimer _t(__func__, /*store*/true);
    std::vector<Result> results;
    for (const auto& p : paths)
    {
        const auto records = LoadRecords(p);

        const auto totalValue = CalcTotalOrder(records, startDate, endDate);
        results.push_back({ p.string(), totalValue });
    }
    return results;
}

void ShowResults(const std::vector<Result>& results, Date startDate, Date endDate)
{
    const size_t maxStringLen = std::accumulate(std::cbegin(results), std::cend(results), 0,
        [](size_t l, const auto &result) { return std::max(l, result.mFilename.length()); }
    );

    std::cout << std::setw(maxStringLen + 1) << std::left << "Name Of File";
    if (!startDate.IsInvalid() && !endDate.IsInvalid())
        std::cout << " | Total Orders Value between " << startDate << " and " << endDate <<
    else if (!startDate.IsInvalid())
        std::cout << " | Total Orders Value since " << startDate << '\n';
    else
        std::cout << " | Total Orders Value\n";

    for (const auto& [fileName, sum] : results)
        std::cout << std::setw(maxStringLen + 1) << std::left << fileName << " | " << std::f
```

```
    ScopeTimer::ShowStoredResults();
}

int main(int argc, const char** argv)
{
    if (argc <= 1)
    {
        std::cerr << "path (startDate) (endDate)\n";
        return 1;
    }

    try
    {
        const auto paths = CollectPaths(argv[1]);

        if (paths.empty())
        {
            std::cout << "No files to process...\n";
            return 0;
        }

        const auto startDate = argc > 2 ? Date(argv[2]) : Date();
        const auto endDate = argc > 3 ? Date(argv[3]) : Date();

        const auto results = CalcResults(paths, startDate, endDate);

        ShowResults(results, startDate, endDate);
    }
    catch (const fs::filesystem_error& err)
    {
        std::cerr << "filesystem error! " << err.what() << '\n';
    }
    catch (const std::runtime_error& err)
    {
        std::cerr << "runtime error! " << err.what() << '\n';
    }

    return 0;
}
```

The code is being compiled using the following `G++` command:

```
g++ -std=c++17 -O2 -Wall csv_reader.cpp -lstdc++fs;
./a.out ./files
```

You can download this code and pass a directory containing CSV files.

In the next lessons, we'll explore the core parts of the application.