The Core Parts

Let's examine the core parts of our CSV Reader application.

The Main

Let’s start with the `main()` function.

```
int main(int argc, const char** argv) {
    if (argc <= 1)
        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(startDate, endDate, results);
    }
    catch (const std::filesystem::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;
}
```

Once we're sure that there are enough arguments in the command line we enter the main scope where all the processing happens:

- **line 6** - gather all the files to process in function `CollectPaths`
- **line 16** - convert data from the files into record data and calculate the results - in `CalcResults`
- **line 18** - show the results on the output - in `ShowResults`

The code relies on exceptions across the whole application.

IsCSVFile

The paths are collected using `directory_iterator` from the `std::filesystem` library:

```
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;
}
```

As in other filesystem examples, the namespace `fs` is an alias for `std::filesystem`.

With `directory_iterator` we can easily iterate over a given directory. By using `copy_if` we can filter out unwanted files and select only those with CSV extension. Notice how easy it is to get the elements of the path and check properties of a file.

Going back to `main` code in the beginning of the lesson, we check if there are any files to process (**line 8**).

Then, in lines **13** and **14**, we parse optional dates: `startDate` and `endDate` are read from `argv[2]` and `argv[3]`.

The dates are stored in a helper class `Date` that allows converting from strings with a simple format of `Day-Month-Year` or `Year-Month-Day`. The class also supports comparison of dates. This will help to filter only the orders from the given date span.

Now, all of the computations and printouts are contained in lines:

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

CalcResults

`CalcResults` implements the core requirements of the application:

- converts data from the file into a list of records to process
- calculates a sum of records between given dates

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

[[nodiscard]] std::vector<Result>
CalcResults(const std::vector<fs::path>& paths, Date startDate, Date endDate)
{
    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;
}
```

The code loads records from each CSV file, then calculates the sum of those records. The results (along with the name of the file) are stored in the output vector.

We can now reveal the code behind the two essential methods `LoadRecords` and `CalcTotalOrder`.