

backend _code challenge



```
1  #ifndef CSV_PROCESSOR_H
2  #define CSV_PROCESSOR_H
3
4  #ifdef __cplusplus
5  extern "C" {
6  #endif
7
8  void processCsv(const char* csv,
9  void processCsvFile(const char* c
10
11  #ifdef __cplusplus
12  }
13  #endif
14
15  #endif
```

//created by: **matheus henrique**

Sumário

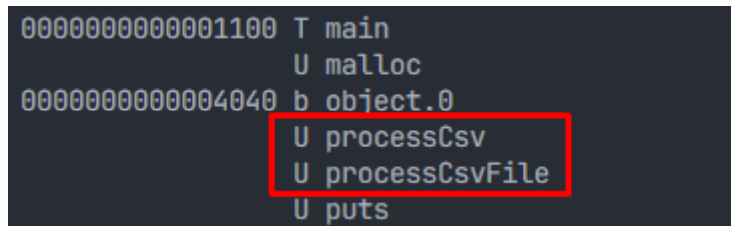
1.	Explorando o problema	3
1.1.	Compreensão inicial de um shared object e entendimento da implementação	3
1.2.	Criação da biblioteca base.....	3
1.3.	Arquivo “build.sh”	4
1.4.	Build do Dockerfile	4
2.	Detalhes do Código fonte.....	5
2.1.	Estrutura de pastas	5
2.2.	Arquivo csv-processor.cpp	5
	Função processCsv	6
	Função preprocessFilters	9
	Continuação do fluxo em processCsv	11
	Função satisfiesFilters	12
	Finalizando a execução de processCsv	14
	Função processFileCsv.....	14
2.3.	Testes unitários	15
3.	Execução da lib com os testes	17
	build_libcsv.sh	17
	build_tests.sh	18

1. Explorando o problema

1.1. Compreensão inicial de um shared object e entendimento da implementação

O objetivo do desafio consiste em desenvolver uma biblioteca compartilhada (`_shared object_` (.so) que será utilizada por uma aplicação C.

A partir disso, houve um estudo sobre o que é um *shared object* e como criá-lo. Então, após as primeiras análises com o conhecimento obtido, percebi que o arquivo “test_libcsv” era um Executable and Linking Format (ELF) e iria executar o que foi especificado no arquivo Header “libcsv.h”, ou seja, os métodos “**void processCsv(const char[], const char[], const char[])**” e “**void processCsvFile(const char[], const char[], const char[])**”. Para confirmar que o executável implementava os métodos quer iriam ser criados na biblioteca csv, usei o comando “nm” no executável “test_libcsv” para ter informações dos símbolos usados no arquivo e constatei que existia dois “Unresolved symbols” referente aos métodos.



```
00000000000001100 T main
                  U malloc
00000000000004040 b object.0
                  U processCsv
                  U processCsvFile
                  U puts
```

Portanto, concluí que o arquivo executável já existente executava os métodos “processCsv” e “processCsvFile” que, posteriormente, viriam ser implementados pela biblioteca compartilhada.

1.2. Criação da biblioteca base

Após o entendimento base de como funciona um *shared object*, foi feito um rascunho da biblioteca que consistia em apenas imprimir o conteúdo dos parâmetros que estavam sendo passados para os métodos “processCsv” e “processCsvFile”. Como a biblioteca inicial foi criada com o propósito de apenas exibir o conteúdo, e, eu já tinha em mente em usar uma linguagem compilada para facilitar a criação da biblioteca, criei ela inicialmente em C dada a facilidade de implementar os “printf” com o conhecimento em linguagens compiladas que tenho. Por tanto, foi criado um diretório csv com um arquivo csv.c apenas para implementar os métodos imprimindo os parâmetros passados.

1.3. Arquivo “build.sh”

Após notar que no Dockerfile não é especificada a instalação do GCC para podermos compilar a nossa biblioteca csv, criamos o comando no “build.sh” para instalar toda a build-base compatível com o Alpine Linux especificado no Dockerfile. O comando foi retirado da especificação presente em: <https://wiki.alpinelinux.org/wiki/GCC>.

```
3  # Checking whether GCC is installed or not
4  if ! command -v gcc &> /dev/null; then
5      echo "Instaling gcc, musl-dev and binutils packages..."
6      apk add build-base
7  fi
```

Por fim, ainda no “build.sh”, utilizamos o seguinte comando para compilar a biblioteca criada em C:

```
# Compiling the csv library
gcc -o libcsv.so -fpic -shared csv/csv.c
```

Vale ressaltar que o comando para a compilação posteriormente mudará.

1.4. Build do Dockerfile

Com nosso arquivo “build.sh” pronto, basta fazer o build da aplicação com “**docker build -t lib .**”. Com a imagem buildada, agora podemos rodar o container e ver que tivemos o resultado esperado:

```
> docker run lib
processCsv output:

CSV Data: col1,col2,col3,col4,col5,col6,col7
l1c1,l1c2,l1c3,l1c4,l1c5,l1c6,l1c7
l1c1,l1c2,l1c3,l1c4,l1c5,l1c6,l1c7
l2c1,l2c2,l2c3,l2c4,l2c5,l2c6,l2c7
l3c1,l3c2,l3c3,l3c4,l3c5,l3c6,l3c7

Selected Columns: col1,col3,col4,col7
Row Filter Definitions: col1>l1c1
col3>l1c3

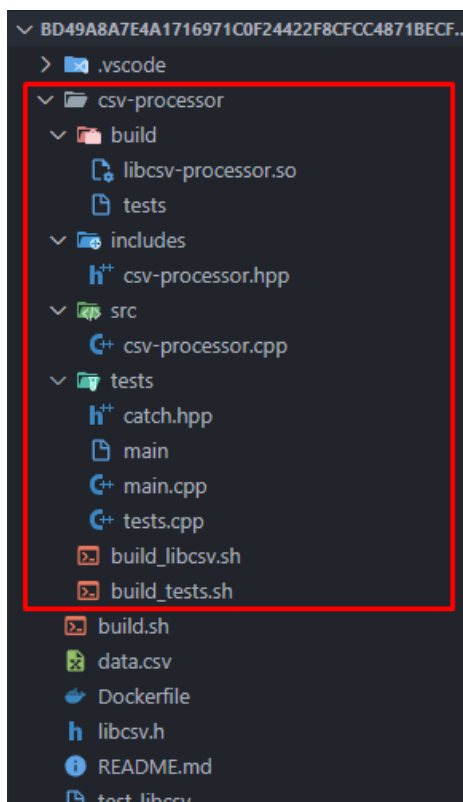
processCsvFile output:

CSV File Path: ./data.csv
Selected Columns: col1,col3,col4,col7
Row Filter Definitions: col1>l1c1
col3>l1c3
col1,col2,col3,col4,col5,col6,col7
l1c1,l1c2,l1c3,l1c4,l1c5,l1c6,l1c7
l1c1,l1c2,l1c3,l1c4,l1c5,l1c6,l1c7
l2c1,l2c2,l2c3,l2c4,l2c5,l2c6,l2c7
l3c1,l3c2,l3c3,l3c4,l3c5,l3c6,l3c7
```

2. Detalhes do Código fonte

2.1. Estrutura de pastas

Como podemos observar pela imagem abaixo, foi criada uma pasta “csv-processor” para a nossa lib, a seguir, iremos comentar de forma resumida sobre o conteúdo dela. A pasta **build** comporta todos os executáveis do projeto. Na **includes**, temos o arquivo .hpp, o nosso arquivo cabeçalho da lib. Na **src**, temos o arquivo .cpp, o principal com toda a lógica do shared object e que implementa as definições do arquivo .hpp. Na pasta **tests**, temos os testes unitários da aplicação. E, por fim, temos arquivos .sh na raiz do diretório csv-processor, sendo eles, o **build_libcsv.sh** e o **build_tests.sh**. É importante ressaltar que, posteriormente, os arquivos mais importantes vão ter uma explicação mais detalhada.



2.2. Arquivo csv-processor.cpp

Logo de início, vamos explicar detalhadamente o nosso principal arquivo desenvolvido que possui toda a implementação do nosso *shared object*, o **csv-processor.cpp**. É importante ressaltar que a explicação de cada trecho de código será atrelada à maioria das funcionalidades e requisitos especificados, tanto os obrigatórios, quanto os bônus. Também é importante ressaltar que a linguagem agora passa a ser c++, dada a quantidade extra de recursos que facilita a manipulação de strings e estruturas.

Função processCsv

Vamos começar com a início do fluxo de execução do código da função **processCsv**. No trecho abaixo, estamos utilizando o “**istream**” do c++ para auxiliar a extrair os dados da string **csv** que foi passada como parâmetro da função. Então, já que de acordo com os requisitos temos que **“A primeira linha do CSV sempre será um header”**, estamos usando um **getline** logo na primeira linha do csv lido e armazenando em uma string chamada **headerColumnsLine**.

```
119 void processCsv(const char csv[], const char selectedColumns[], const char rowFilterDefinitions[]) {
120     // Taking the first line of the csvData (headers columns line)
121     std::stringstream streamCsv(csv);
122     std::string headerColumnsLine;
123     std::getline(streamCsv, headerColumnsLine);
124     // Splitting the headerColumnsLine by commas into a vector of strings
```

Após isso, utilizamos outro stream para extrair dados, agora, sendo em cima da string **headerColumnsLine** recém-criada. Após isso, criamos um vector de string chamado **headerColumns**, que será responsável por armazenar cada header/coluna do csv. Então, iteramos sob a nossa stream baseada em **headerColumnsLine**, splitando por vírgula e armazenando no array. O split é feito respeitando o requisito **“Vírgulas sempre delimitam uma coluna, aspas não têm nenhuma interpretação especial”**.

```
123     std::getline(streamCsv, headerColumnsLine);
124     // Splitting the headerColumnsLine by commas into a vector of strings
125     std::stringstream headerColumnsStream(headerColumnsLine);
126     std::vector<std::string> headerColumns;
127     std::string column;
128     while (std::getline(headerColumnsStream, column, ',')) {
129         headerColumns.push_back(column);
130     }
131     }
```

Vale ressaltar que, até o ponto especificado, como estamos utilizando tipos de dados bem flexíveis e definidos do c++, como **string** e **vector**, estamos cumprindo os seguintes requisitos: **“A sua implementação deve tratar CSVs com quantidades arbitrárias de caracteres”** e **“A sua implementação deve tratar CSVs onde as colunas têm quantidades arbitrárias de caracteres”**.

No trecho abaixo, temos como objetivo guardar os headers que foram selecionados e é passado como parâmetro na função **processCsv**. Então, criamos outro vector do tipo **HeaderColumn** com o nome de **HeaderColumnsToSelect** que vai guardar os headers selecionados.

```
// We'll store the header columns name and its index just if it's in the selectedColumns
// If selectedColumns is empty, we'll store all headers
std::vector<HeaderColumn> headerColumnsToSelect; // Array to store the header columns name and its index
```

O tipo **HeaderColumn** consiste em uma string com o nome da coluna e o seu índice real baseado nas colunas do csv. Essa estrutura será bem útil em algumas

operações, tal como a do requisito bônus que consiste na aparição em ordem arbitrária das colunas selecionadas.

```
// Struct to store the header column name and its index
You, 4 days ago | 1 author (You)
struct HeaderComponent
{
    std::string name;
    int index;
};
```

Continuando o fluxo, de acordo com o requisito **“Uma string de seleção de colunas vazia é equivalente a selecionar todas as colunas”**, temos uma condição que avalia se o parâmetro **“selectedColumns”** é uma string vazia, se for, iteramos sob nosso array **headerColumns**, armazenando todas as colunas como colunas selecionadas em **headerColumnsToSelect**, guardando o nome e a posição/índice dessa coluna.

```
if (selectedColumns[0] == '\\0') { // selectedColumns is empty
    for(int i = 0; i < headerColumns.size(); i++){
        headerColumnsToSelect.push_back({headerColumns[i], i});
    }
} else {
```

Caso ao contrário, significa que temos alguma coluna selecionada. Antes de armazenar em **headerColumnsToSelect** as colunas que foram selecionadas, pensando em performance computacional, criamos um **unordered_map** do c++, que é basicamente uma estrutura hash e vai permitir que, posteriormente, façamos uma verificação da existência de uma coluna selecionada em todas as colunas do csv em complexidade $O(1)$.

```
} else {
    // Creating an unordered_map to store the header name and its index.
    // It will be used to find the index of the selectedColumns with complexity O(1)
    std::unordered_map<std::string, int> headerColumnIndexMap;
    for (int i = 0; i < headerColumns.size(); ++i) {
        headerColumnIndexMap[headerColumns[i]] = i;
    }
}
```

Então, como podemos ver na figura abaixo, iremos iterar sob as colunas que foram selecionadas pegando cada uma delas e verificando se ela existe nas colunas que foram fornecidas no csv. Então, iremos splitar cada coluna do parâmetro **“selectedColumns”** pela vírgula e fazer a verificação gastando $O(n)$ no total, onde **n** é a quantidade de colunas selecionadas existentes. Como temos uma estrutura em hash com todas as colunas do csv, basta procurar nessa estrutura se há algo com o nome da coluna passada, com isso, temos uma operação instantânea, gastando $O(1)$. Note que, se percorrêssemos um array com todas as colunas para fazer a busca de um elemento, no pior caso, iríamos percorrer todo o array, já que o elemento a se buscar poderia estar na última posição ou nem existir.

Por fim, a fim de cumprir outro requisito bônus que consistem em **“Colunas que não existem podem aparecer na seleção de colunas e nos filtros”**, temos um tratamento para essa ocasião, onde, se não acharmos a coluna na estrutura hash, lançamos um erro falando que a coluna não foi achada no CSV file/string e parando a execução do programa com um early return. Caso achemos a coluna na estrutura hash, armazenamos o seu nome e o índice no nosso array **HeaderColumnsToSelect** do tipo **HeaderColumn**.

```
std::istringstream selectedColumnsStream(selectedColumns);
std::string column;
// We'll iterate over the selectedColumns with complexity O(n) where n is the number of selectedColumns
// The complexity time of this block is O(n) * O(1) = O(n)
while(std::getline(selectedColumnsStream, column, ',')){
    // Checking if the column is in the headerColumnIndexMap
    auto it = headerColumnIndexMap.find(column); // find in an unordered_map has complexity O(1)
    if (it != headerColumnIndexMap.end()) {
        headerColumnsToSelect.push_back({it->first, it->second});
    } else {
        std::cerr << "Header '" << column << "' not found in CSV file/string" << std::endl;
        return;
    }
}
```

A fim de cumprir outro requisito, que consiste em **“As colunas que aparecem na string de colunas selecionadas podem estar em ordem arbitrária”**, vamos fazer um sort em **headerColumnsToSelect** que custará apenas $O(n \log n)$, já que estamos utilizando o sort otimizado do c++. Esse sort só foi possível por guardarmos a referência do índice inicial de todas as colunas do csv no array do tipo **HeaderColumn**.

```
// Before any processing, we need to sort the headerColumnsToSelect by the index
// Therefore, we'll have the selected columns in the correct order
// The std::sort has complexity O(n log n) where n is the number of selected columns
std::sort(headerColumnsToSelect.begin(), headerColumnsToSelect.end(), [](const HeaderColumn& a, const HeaderColumn& b) {
    return a.index < b.index;
});
```

Em conformidade com os requisitos **“O CSV processado deve ser escrito no stdout”** e **“O CSV processado deverá incluir o header do CSV considerando a seleção de coluna”**, a esse ponto, já poderíamos imprimir as colunas que foram selecionadas, porém, como existem outros requisitos que podem lançar erros na aplicação, para só imprimirmos algo quando tivermos a certeza de que tudo executará certo, armazenamos essa informação em um buffer para imprimir apenas no final do fluxo de execução.

```
// Storing the selected columns in a buffer output to print them at the end
std::stringstream bufferHeaderColumnsToSelect;
for (int i = 0; i < headerColumnsToSelect.size(); ++i) {
    bufferHeaderColumnsToSelect << headerColumnsToSelect[i].name;
    if (i < headerColumnsToSelect.size() - 1) bufferHeaderColumnsToSelect << ",";
}
bufferHeaderColumnsToSelect << "\n";
```


Agora, como temos todas as informações necessárias sobre as colunas selecionadas, iremos nos atentar aos filtros passados para a função. Para isso, criaremos um array de filtros com ajuda de outra estrutura, um tipo `Filtro`.

```
// Preprocessing the rowFilterDefinitions based on all columns.  
// It's throw a error if a filter has a non-existent column or there's an invalid filter  
std::vector<Filter> filters;
```

O tipo filtro vai facilitar a análise de cada linha posteriormente. Ele consiste no índice de qual coluna devemos analisar, o comparador que iremos utilizar no filtro e o valor a se comparar.

```
// Struct to store the filter definition  
You, 6 days ago | 1 author (You)  
struct Filter  
{  
    int columnIndex;  
    std::string comparator;  
    std::string value;  
};  
You, 6 days ago • implementação base em c++ de processCsv na lib ...
```

Já que na função `preprocessFilters` validamos o filtro, utilizaremos um try-catch na chamada da função. Explicaremos mais abaixo dentro da função como será feita a validação do filtro.

```
// Preprocessing the rowFilterDefinitions based on all columns.  
// It's throw a error if a filter has a non-existent column or there's an invalid filter  
std::vector<Filter> filters;  
try {  
    filters = preprocessFilters(headerColumns, rowFilterDefinitions);  
} catch(const std::runtime_error& e){  
    std::cerr << e.what() << std::endl;  
    return;  
}
```

Função `preprocessFilters`

O objetivo da função é fazer um pré-processamento dos filtros e armazená-los no nosso array **Filters** do tipo **Filter**. Essa função recebe todos os headers do csv e a definição dos filtros presentes no parâmetro **rowFilterDefinition**. Vale ressaltar que, estamos passando todos os headers e não só os selecionados como de acordo com o requisito e o exemplo que é passado no README e que consiste em “Apenas linhas que condizem com todos os filtros devem ser selecionadas”, com o exemplo: “Aplicando os filtros `header1=4\nheader2>3` e selecionando as colunas header1 e header3. Somente a linha 4,5,6 `(header1 = 4 AND header2 > 3)` deve ser selecionada, pois todas as condições dos filtros devem ser atendidas”. Ou seja, como no exemplo é passado um filtro com a coluna **header2** que não está entre as colunas selecionadas, consideramos que os

filtros que serão aplicados podem ser baseados em filtros que não foram selecionados.

```
// Preprocess the filters based on the header columns and the rowFilterDefinitions and store them in a vector of Filters
std::vector<Filter> preprocessFilters(const std::vector<std::string>& headerColumns, const std::string& rowFilterDefinitions){
```

Logo no início da função, para entrar em conformidade com o requisito bônus que consiste em **“Tratamento de erro para filtros inválidos”**, temos uma verificação que, se o parâmetro **rowFilterDefinitions** estiver vazio, lançamos uma exceção de erro falando que o não há filtros que vai ser capturada pelo try-catch na chamada da função.

```
// Preprocess the filters based on the header columns and the rowFilterDefinitions and store them in a vector of Filters
std::vector<Filter> preprocessFilters(const std::vector<std::string>& headerColumns, const std::string& rowFilterDefinitions){
    // Checking if the rowFilterDefinitions is empty
    if(rowFilterDefinitions.empty()){
        throw std::runtime_error("Invalid filter: There is no filter, rowFilterDefinitions is empty");
    }
}
```

Após isso, iremos percorrer os filtros presentes em **rowFilterDefinition** com o delimitador de newline (\n). Para cada filtro, iremos aplicar uma regex. A regex consiste em: Pegar um primeiro grupo em que não haja nenhum símbolo referentes aos nossos operadores usados (<=>!=>=<=), um segundo grupo de um ou dois dígitos referente aos nossos operadores e um terceiro grupo que tem a mesma especificação do primeiro. Vale ressaltar que, tive uma dificuldade para analisar com perfeição o grupo do meio e já lançar um erro se fosse passado um comparador semelhante a “>>”, então, tenho uma outra verificação em outra parte do código que valida o operador.

Após isso, vamos verificar se a regex é válida, então, caso **“Tratamento de erro para filtros inválidos”**, ela não seja válida, lançamos uma exceção de erro falando que o filtro não é válido e qual é o filtro, estando assim, em conformidade com o requisito bônus que consiste em **“Tratamento de erro para filtros inválidos”**.

Após isso, pegamos os grupos que foram especificados na regex e guardamos nas variáveis **headerColumnName** (com o nome da coluna do filtro), **comparador** (com o comparador referente a operação que deve ser aplicada) e **value** (valor a ser comparado).

Como para nós é interessante ter o índice da coluna que representa o nome, iremos procurar em **headerColumns** qual é o índice da coluna com o nome que passamos. Achando o índice da coluna, adicionaremos esse novo filtro ao nosso array **Filters** do tipo **Filter**. Caso não encontremos algum índice referente ao nome da coluna, significa que não há uma coluna com esse nome no CSV, com isso, lançaremos um erro que mostra que não existe uma coluna com o nome passado. Lançando essa outra exceção de erro, estamos em conformidade com o requisito bônus que consiste em **“Colunas que não existem podem aparecer na seleção de colunas e nos filtros”**.

Por fim, retornamos os nossos filtros para a variável que chamou a função **preprocessFilters** em **processCsv**.

```
std::vector<Filter> filters;
std::istringstream filterStream(rowFilterDefinitions);
std::string filterDefinition;

while (std::getline(filterStream, filterDefinition, '\n')) {
    // Using a regex to find the headerColumnName, the comparator and the value
    // If we don't find any valid comparator, we throw an error
    std::regex re(R"([^\<=>!=>=<=]+)([<=>!=>=<=]{1,2})([^\<=>!=>=<=]+)"); // Allows anything before and after the comparator
    std::smatch match;
    if (!std::regex_match(filterDefinition, match, re)) {
        throw std::runtime_error("Invalid filter: '" + filterDefinition + "'");
    }
    std::string headerColumnName = match[1];
    std::string comparator = match[2];
    std::string value = match[3];

    // Finding the index of the headerColumnName in the headerColumns and storing the filter in the filters vector
    auto it = std::find(headerColumns.begin(), headerColumns.end(), headerColumnName);
    if (it != headerColumns.end()) {
        int columnIndex = std::distance(headerColumns.begin(), it);
        filters.push_back({columnIndex, comparator, value});
    } else {
        throw std::runtime_error("Header '" + headerColumnName + "' not found in CSV file/string");
    }
}

return filters;
```

Continuação do fluxo em processCsv

Possuindo nossos filtros, iremos percorrer todas as linhas do csv aplicando nossos filtros a cada linha (row). A linha em questão, para ser analisada, vai ter seus valores guardados em um array de strings. Então, para cada linha, vamos aplicar uma função **satisfiesFilters** que valida ela, passando o array de rows e os nossos filtros. Se a linha for válida, iremos armazenar em outro buffer de dados apenas os índices da linha que estão presentes nas colunas que foram selecionadas. No trecho que checka se uma linha satisfaz o filtro, também temos um try-catch para verificar possíveis erros.

```
// Processing the rows based on the selected columns and the filters
std::string line;
std::stringstream bufferDataOutput; // Buffer to store the data output
while (std::getline(streamCsv, line)) {
    std::istringstream lineStream(line);
    std::string field;
    std::vector<std::string> row;

    while (std::getline(lineStream, field, ',')) {
        row.push_back(field);
    }

    // Checking if the row satisfies the filters
    try {
        if (satisfiesFilters(row, filters)) {
            // Storing valid lines in an output buffer to print them at the end
            for (int i = 0; i < headerColumnsToSelect.size(); ++i) {
                bufferDataOutput << row[headerColumnsToSelect[i].index];
                if (i < headerColumnsToSelect.size() - 1) bufferDataOutput << ",";
            }
            bufferDataOutput << "\n";
        }
    } catch (const std::runtime_error& e) {
        std::cerr << e.what() << '\n';
        return;
    }
}
```

Função satisfiesFilters

Agora, mostraremos o que acontece em cada linha em que é aplicada essa função. Antes de começar a explicar, vale ressaltar alguns pontos ligados aos requisitos. Para o requisito **“No mínimo, o candidato deve implementar filtros para maior (>), menor (<) e igual (=)”**, desde o início, o código contempla esses 3 operadores. No requisito **“As colunas na string de filtros podem aparecer em ordem arbitrária”**, com a implementação que temos, não precisávamos nos preocupar, visto que a ordem do filtro não importaria, já que ele seria implementado na linha de qualquer forma e o que importaria no final é se, para cada linha, foi retornado um true ou false. Agora, ao explicarmos a função, iremos resgatar a conformidade com outros requisitos que foram especificados.

Logo de início, vemos que há a declaração de outro unordered_map, antecipando parte da explicação, ele serve para verificar se foi armazenado um valor false em um dos headers. Essa implementação visa cumprir o requisito que consiste em **“Aceitar mais de 1 filtro por header”**, onde, em uma situação em que, considerando que no array de filters há dois filtros que são aplicados no mesmo header, se algum desses filtros resultar em true, podemos considerar que o header é válido na linha.

```
// Check if the row satisfies the filters.  
// Receives a row and a vector of filters, so it iterates over the filters and checks if the row satisfies them  
bool satisfiesFilters(const std::vector<std::string>& row, const std::vector<Filter>& filters) {  
    // Map to store whether a header satisfies any of its filters  
    std::unordered_map<int, bool> headerSatisfied;
```

Então, a partir disso, iremos iterar no vetor de filtros, onde cada filtro tem o **índice** onde está a coluna que devemos verificar, o **comparador** que devemos usar e o **valor** para a comparação. Após isso, armazenamos em **“field”** o valor que deve ser comparado com **valor**. Note que o valor de **field** é o valor que está na row no índice especificado.

```
// Iterating over the filters and checking if the row satisfies them  
for (const auto& filter : filters) {  
    int columnIndex = filter.columnIndex;  
    std::string comparator = filter.comparator;  
    std::string value = filter.value;  
  
    // Obtaining the field of the row based on the columnIndex  
    const std::string& field = row[columnIndex];
```

Com isso, estando em conformidade com o requisito que consiste em **“Os comparadores nos filtros devem seguir a ordem lexicográfica conforme a implementação da [strcmp](https://www.man7.org/linux/man-pages/man3/strcmp.3.html) da stdlibc”**, fazemos a comparação baseado na ordem lexicográfica utilizando a própria strcmp.

```
// Lexicographical comparison using std::strcmp
int comparison = std::strcmp(field.c_str(), value.c_str());
```

Com o valor em **comparison**, ainda com as especificações de como prosseguir com o valor que foi resultado, temos a verificação do resultado baseado no comparador que estamos vendo no filtro. Se de fato temos o resultado esperado para esse comparador, alteramos a variável **satisfied** para true, que, inicialmente, é declarada em false. É importante observar que, caso o comparador não seja nenhum do que foi especificado, entramos no else da verificação, lançando um erro dizendo que o filtro é inválido, assim estando em conformidade com o requisito que consiste em “**Tratamento de erro para filtros inválidos**”. No fim do trecho abaixo, temos a definição de que, no header verificado da linha, existe algum filtro que resulta em true. Vale observar ainda que temos os novos comparadores exigidos no requisito bônus que consiste em “**Implementar os operadores diferente (!=), Maior ou igual que (>=), e Menor ou igual que (<=)**”.

```
bool satisfied = false;

// Checking if the comparison satisfies the filter
if (comparator == ">") {
    satisfied = (comparison > 0);
} else if (comparator == "<") {
    satisfied = (comparison < 0);
} else if (comparator == "=") {
    satisfied = (comparison == 0);
} else if (comparator == "!=") {
    satisfied = (comparison != 0);
} else if (comparator == ">=") {
    satisfied = (comparison >= 0);
} else if (comparator == "<=") {
    satisfied = (comparison <= 0);
} else { // Checking again if the comparator is valid
    throw std::runtime_error("Invalid filter: '" + field + comparator + value + "'");
}

// If the header satisfies the filter, we store as true in the headerSatisfied map
if(satisfied){
    headerSatisfied[columnIndex] = true;
}
```

Após iterar por todos os filtros, vamos verificar se, baseado na **columnIndex** (índice da coluna do filtro) de cada filtro, no nosso unordered_map de **headerSatisfied** com esse índice, há um valor que não é true, ou seja, se acharmos um valor de header que deu falso, significa que a linha (row) não é válida, assim, já retornamos false em **satisfiedFilters**. Caso todos os valores sejam true, retornamos true.

```

// Checking if all headers satisfy the filters
for (const auto& filter : filters) {
    int columnIndex = filter.columnIndex;
    // If any header doesn't satisfy the filter, we return false
    if(!headerSatisfied[columnIndex]){
        return false;
    }
}

return true;

```

Finalizando a execução de processCsv

Após verificarmos todas as linhas do csv e armazenar todas as linhas que são válidas, por fim, exibimos todos os dados que foram armazenados nos buffers **bufferHeaderColumnsToSelect** (com todas as colunas selecionadas) e **bufferDataOutput** (com os dados das linhas válidas).

```

// Showing all buffers content in the console
std::cout << bufferHeaderColumnsToSelect.str() << bufferDataOutput.str();

```

Função processFileCsv

A função `processFileCsv` é nada mais que um intermediador que vai processar os dados direto do csv e chamar a função `processCsv`. Nessa função, lemos o csv, e convertemos para o formato recebido por `processCsv`, concatenando cada linha do csv com um newline (`\n`) e mandando como uma string para **processCsv**.

```

void processCsvFile(const char csvFilePath[], const char selectedColumns[], const char rowFilterDefinitions[]) {
    // Reading the CSV file
    std::ifstream file(csvFilePath);
    if (!file.is_open()) {
        std::cerr << "Error opening CSV file" << std::endl;
        return;
    }

    // We'll store the csv data in a string
    std::string csvString;
    std::getline(file, csvString); // Getting the first line

    // Adding the first newline in the string
    csvString += "\n";

    // Concatenating with the others lines
    std::string line;
    while (std::getline(file, line)) {
        csvString += line;
        csvString += "\n";
    }

    // Converting the std::string to const char*
    const char* csv = csvString.c_str();

    // Calling the processCsv function with the new csv
    processCsv(csv, selectedColumns, rowFilterDefinitions);

    file.close();
}

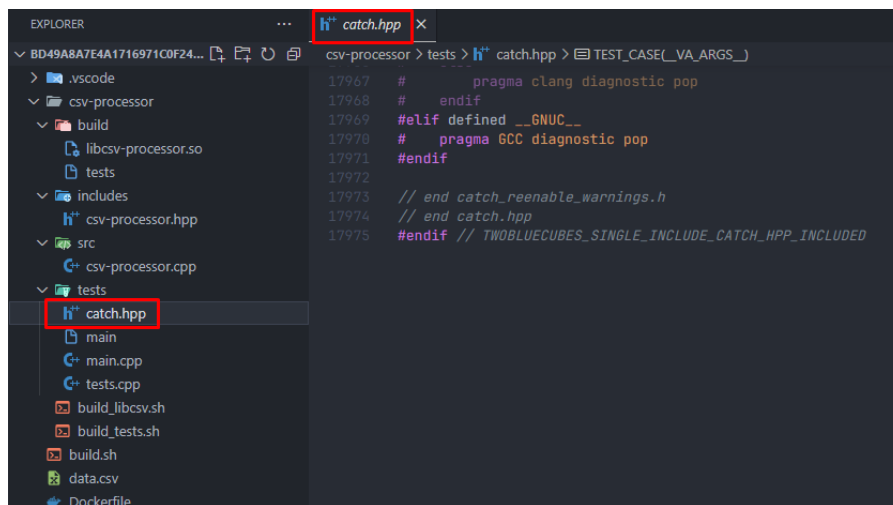
```

2.3. Testes unitários

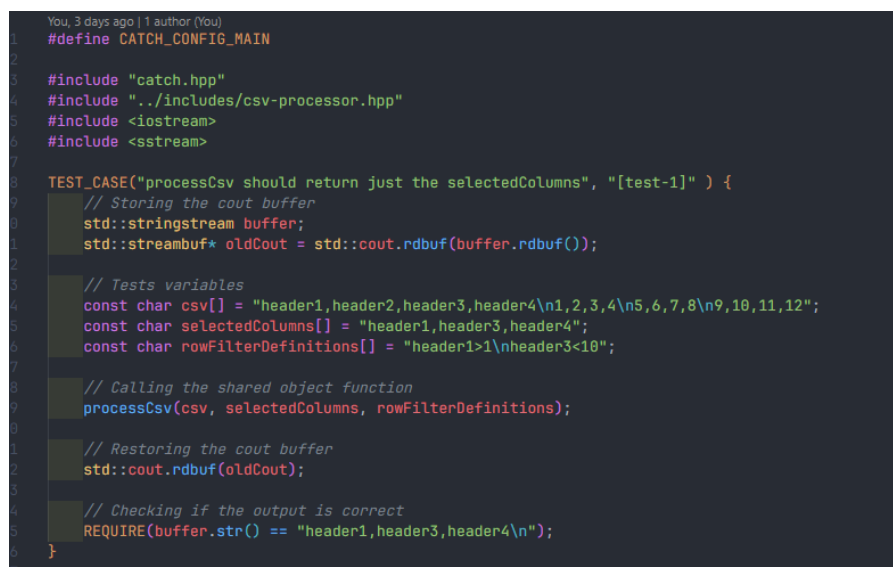
De forma a estar em conformidade com o requisito “**Desenvolver testes unitários**”, foram desenvolvidos testes unitários relacionados às funcionalidades e requisitos para o shared object, utilizando a lib Catch2 (<https://github.com/catchorg/Catch2>).

Após um breve estudo, a lib Catch2 foi escolhida dada a sua simplicidade de uso comparada as outras, e, após os primeiros testes, ela se mostrou suficiente.

Para o uso da lib, basta copiar o arquivo cabeçalho da lib (catch.hpp) para o diretório relativo aos testes da aplicação.



Com isso, no arquivo com os testes “tests.cpp”, devemos importar o cabeçalho, a lib e executar os testes. Para executar um teste, basta usarmos o método **TEST_CASE**, fornecendo uma breve descrição e uma tag. Em cada **TEST_CASE**, guardamos em um buffer o que o nosso shared object deu de resposta em seu stdout, para no fim, compararmos com o resultado esperado usando um **REQUIRE**.



Nos testes que queremos verificar erros, utilizamos um buffer com **cerr**.

```
TEST_CASE("processCsv shouldn't allow non-existent columns in selectedColumns", "[test-9]" ) {
    // Redirect cerr buffer
    std::stringstream errStream;
    std::streambuf* oldCerr = std::cerr.rdbuf(errStream.rdbuf());

    // Tests variables
    const char csv[] = "header1,header2,header3\n1,2,3\n4,5,6\n7,8,9";
    const char selectedColumns[] = "header1,header2,header4";
    const char rowFilterDefinitions[] = "header2>3\nheader1=4";

    // Calling the shared object function
    processCsv(csv, selectedColumns, rowFilterDefinitions);

    // Restore cerr
    std::cerr.rdbuf(oldCerr);

    // Checking if the output is correct
    REQUIRE(errStream.str() == "Header 'header4' not found in CSV file/string\n");
}
```

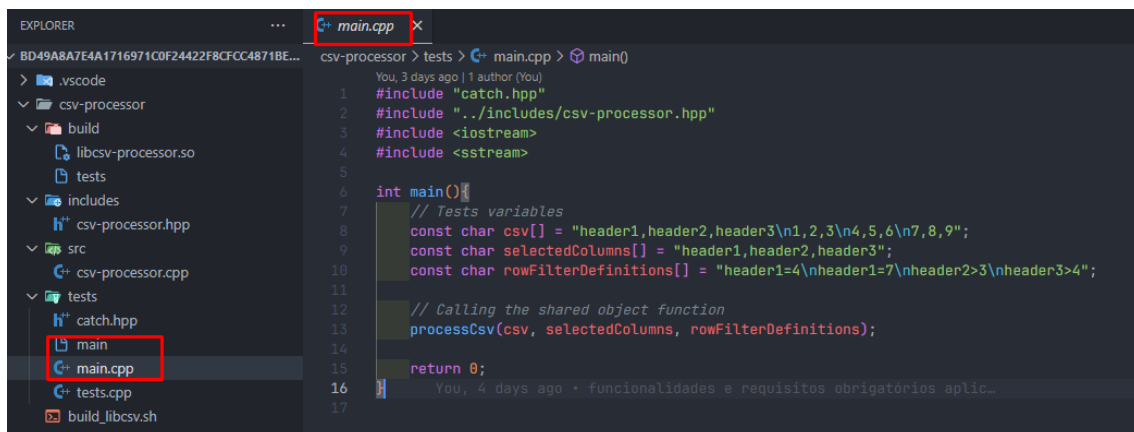
Ao todo, foram feitos 14 testes, sendo que em dentro de dois desses testes, existem **SECTIONS**, que consistem em subtestes, assim, totalizando em 18 testes.

Acredito que não vale a pena detalhar todos os testes, porém tendo explicado a estrutura, o entendimento se torna fácil ao ver o arquivo **tests.cpp**.

Vale ressaltar que todos os testes passaram:

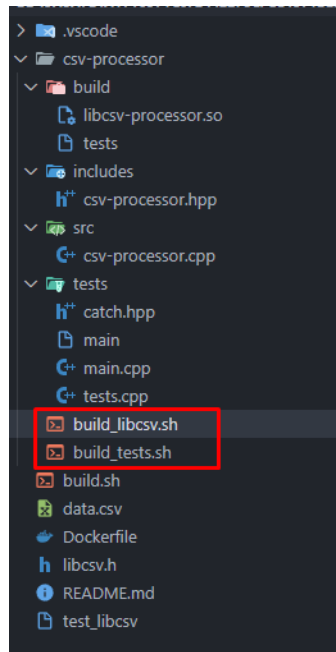
```
~/GIT/bd49a8a7e4a1716971c0f24422f8cfcc4871becf/csv-processor main
) ./build/tests
=====
All tests passed (18 assertions in 14 test cases)
```

Por fim, vale citar que os arquivos destacados abaixo são referentes a testes bem específicos para testar rapidamente quando uma mudança no *shared object* tinha sido feita sem a necessidade de compilação e inserção de novos testes em “**tests.cpp**”.



3. Execução da lib com os testes

Por mais que já existam os arquivos compilados na pasta **build**. Vamos explicar como geramos esses arquivos. Como podemos ver na imagem abaixo, na raiz da pasta “**csv-processor**”, temos os arquivos de script .sh **build_libcsv.sh** (para o build do *shared object*) e **build_tests.sh** (para o build dos tests).



build_libcsv.sh

O script abaixo verifica se já existe um diretório build, se não existir, ele cria. Após isso, temos um comando para compilar a lib e, por fim, exibe uma mensagem de término de execução do script.

```
build_libcsv.sh ×
csv-processor > build_libcsv.sh
You, 4 days ago | 1 author (You)
1  #!/bin/bash      You, 5 days ago • organizando a lib csv-processor em diretórios, ...
2
3  # Build directory
4  BUILD_DIR=build
5
6  # If the build directory does not exist, create it
7  if [ ! -d "$BUILD_DIR" ]; then
8      mkdir $BUILD_DIR
9  fi
10
11 # Compiling the shared object code
12 g++ -o $BUILD_DIR/libcsv-processor.so -fpic -shared src/csv-processor.cpp
13
14
15 # Finished message
16 echo "build_libcsv finished"
17
```

Para executar o script, basta estar dentro do diretório “**csv-processor**” e usar o comando “**./build_libcsv.sh**”.

```
~/GIT/bd49a8a7e4a1716971c0f24422f8cfcc4871becf/csv-processor main
) ./build_libcsv.sh
build_libcsv finished
```

build_tests.sh

O script abaixo também verifica se já existe um diretório build, se não existir, ele cria. Após isso, temos um comando para compilar os testes utilizando a lib e, por fim, exibe uma mensagem de término de execução do script.

```
build_tests.sh x
csv-processor > build_tests.sh
You, 4 days ago | 1 author (You)
1  #!/bin/bash      You, 5 days ago • organizando a lib csv-processor em diretórios, ...
2
3  # Build directory
4  BUILD_DIR=build
5
6  # If the build directory does not exist, create it
7  if [ ! -d "$BUILD_DIR" ]; then
8      mkdir $BUILD_DIR
9  fi
10
11 # Compiling the shared object code
12 g++ -o $BUILD_DIR/tests tests/tests.cpp -L. -l:build/libcsv-processor.so
13
14 # Finished message
15 echo "build_test finished"
16
```

Para executar o script, basta estar dentro do diretório “**csv-processor**” e usar o comando “**./build_tests.sh**”.

```
~/GIT/bd49a8a7e4a1716971c0f24422f8cfcc4871becf/csv-processor main
) ./build_tests.sh
build_test finished
```

Por fim, para rodar os testes, basta estar dentro do diretório “**csv-processor**” e usar o comando “**./build/tests**”.

```
~/GIT/bd49a8a7e4a1716971c0f24422f8cfcc4871becf/csv-processor main
) ./build/tests
=====
All tests passed (18 assertions in 14 test cases)
```

Se quiser ver detalhes dos testes, basta adicionar a diretiva “**—success**” ao comando `acima,` resultando `em:`

```
~/6IT/bd49a8a7e4a1716971c0f24422f8cfcc4871becf/csv-processor main
) ./build/tests --success

~~~~~
tests is a Catch v2.13.10 host application.
Run with -? for options

-----
processCsv should return just the selectedColumns
-----
tests/tests.cpp:8
.....

tests/tests.cpp:25: PASSED:
  REQUIRE( buffer.str() == "header1,header3,header4\n" )
with expansion:
  "header1,header3,header4
  "
  ==
  "header1,header3,header4
  "

-----
processCsvFile should return the specified result
```