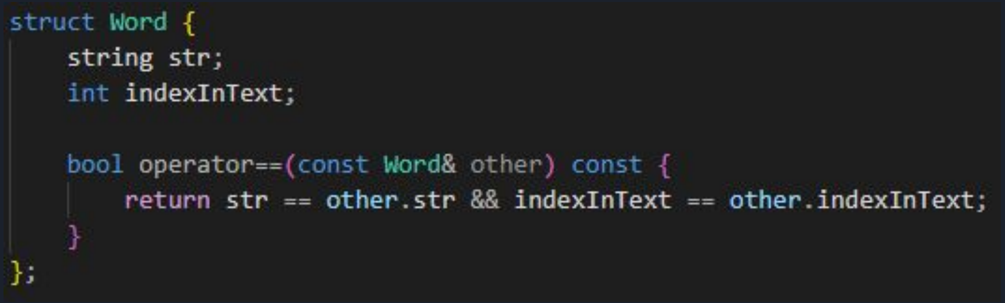
FPROG Project

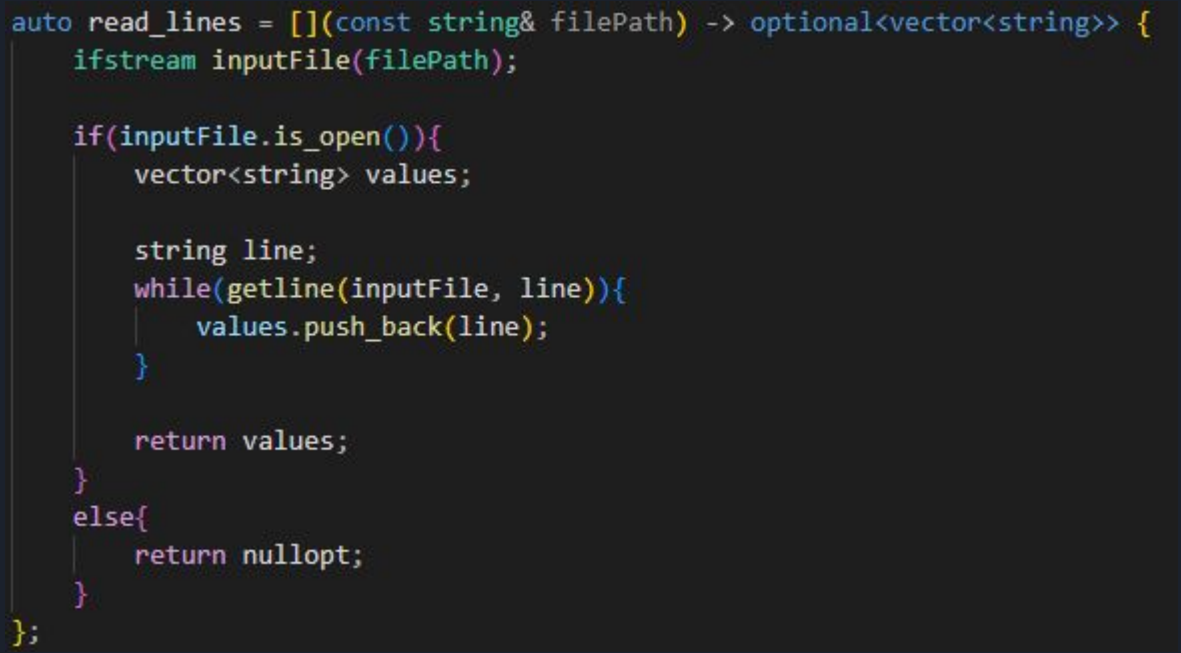
TextAnalytics

David Höchtl, Maximilian Mörth, Armin Islamovic

Read files

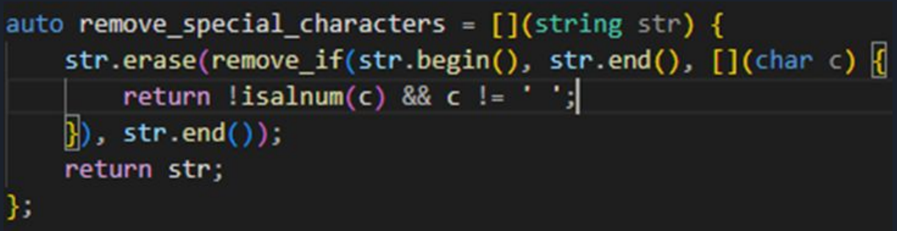


```
struct Word {
    string str;
    int indexInText;

    bool operator==(const Word& other) const {
        return str == other.str && indexInText == other.indexInText;
    }
};
```

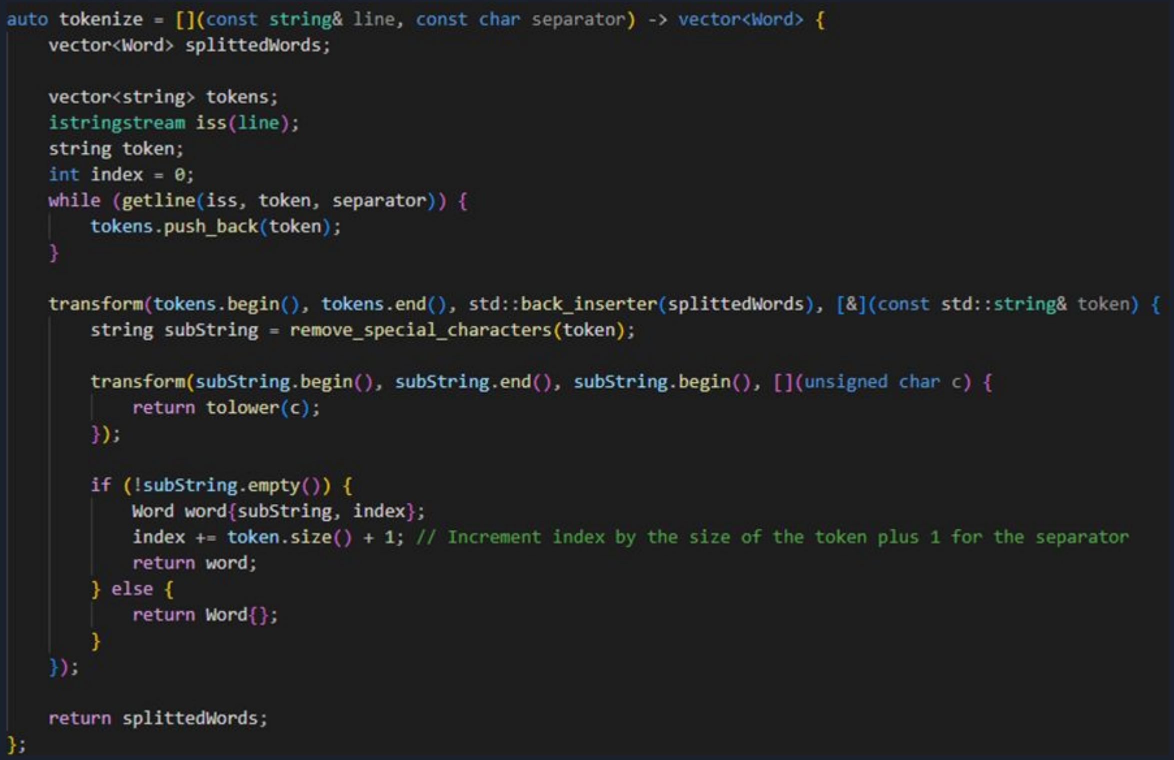


```
auto read_lines = [](const string& filePath) -> optional<vector<string>> {
    ifstream inputFile(filePath);

    if(inputFile.is_open()){
        vector<string> values;

        string line;
        while(getline(inputFile, line)){
            values.push_back(line);
        }

        return values;
    }
    else{
        return nullopt;
    }
};
```

Tokenize the text

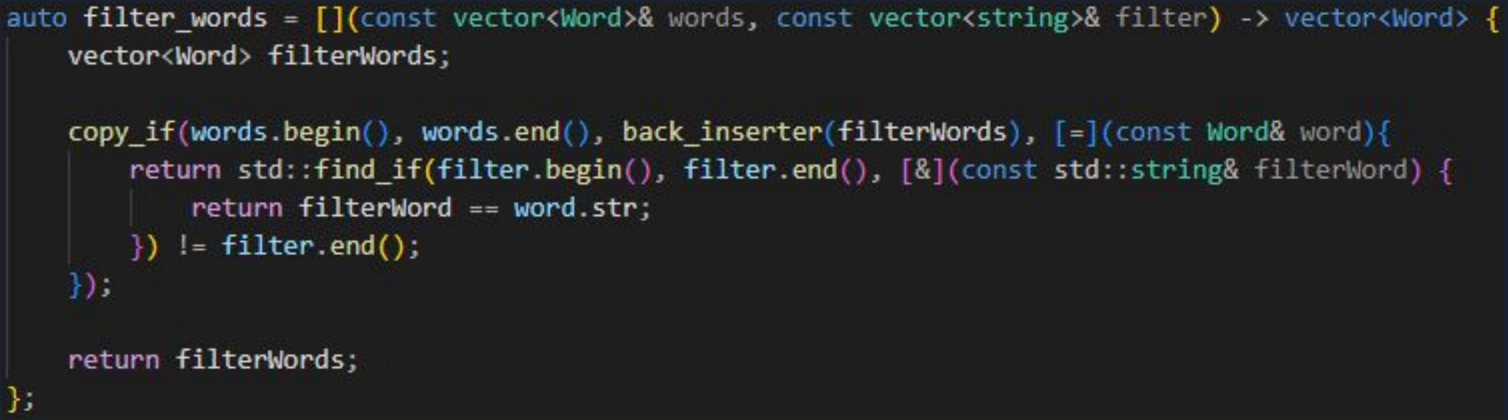


```
auto remove_special_characters = [](string str) {
    str.erase(remove_if(str.begin(), str.end(), [](char c) {
        return !isalnum(c) && c != ' ';
    }), str.end());
    return str;
};
```



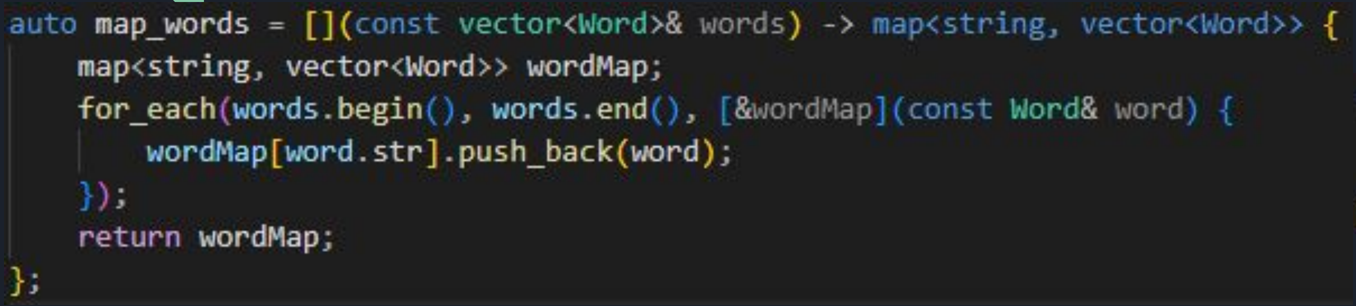
```
auto tokenize = [](const string& line, const char separator) -> vector<Word> {
    vector<Word> splittedWords;

    vector<string> tokens;
    istringstream iss(line);
    string token;
    int index = 0;
    while (getline(iss, token, separator)) {
        tokens.push_back(token);
    }

    transform(tokens.begin(), tokens.end(), std::back_inserter(splittedWords), [&](const std::string& token) {
        string subString = remove_special_characters(token);

        transform(subString.begin(), subString.end(), subString.begin(), [](unsigned char c) {
            return tolower(c);
        });

        if (!subString.empty()) {
            Word word{subString, index};
            index += token.size() + 1; // Increment index by the size of the token plus 1 for the separator
            return word;
        } else {
            return Word{};
        }
    });

    return splittedWords;
};
```

Filter words

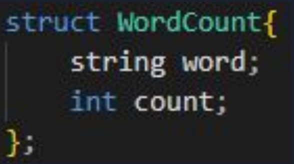


```
auto filter_words = [](const vector<Word>& words, const vector<string>& filter) -> vector<Word> {
    vector<Word> filterWords;

    copy_if(words.begin(), words.end(), back_inserter(filterWords), [=](const Word& word){
        return std::find_if(filter.begin(), filter.end(), [&](const std::string& filterWord) {
            return filterWord == word.str;
        }) != filter.end();
    });

    return filterWords;
};
```

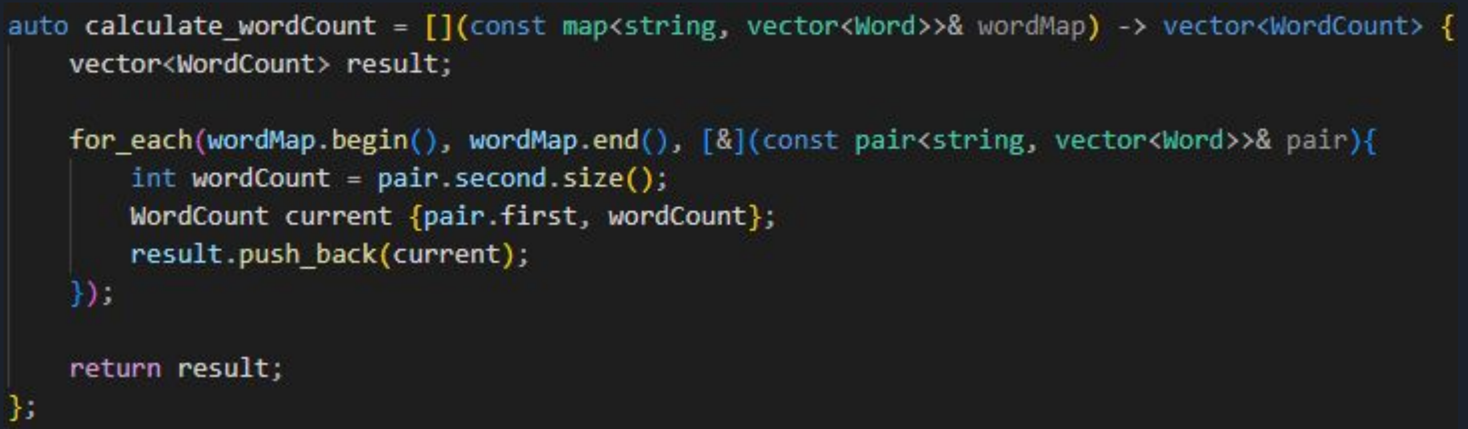
Count word occurrences



```cpp
auto map_words = [](const vector<Word>& words) -> map<string, vector<Word>> {
    map<string, vector<Word>> wordMap;
    for_each(words.begin(), words.end(), [&wordMap](const Word& word) {
        wordMap[word.str].push_back(word);
    });
    return wordMap;
};
```

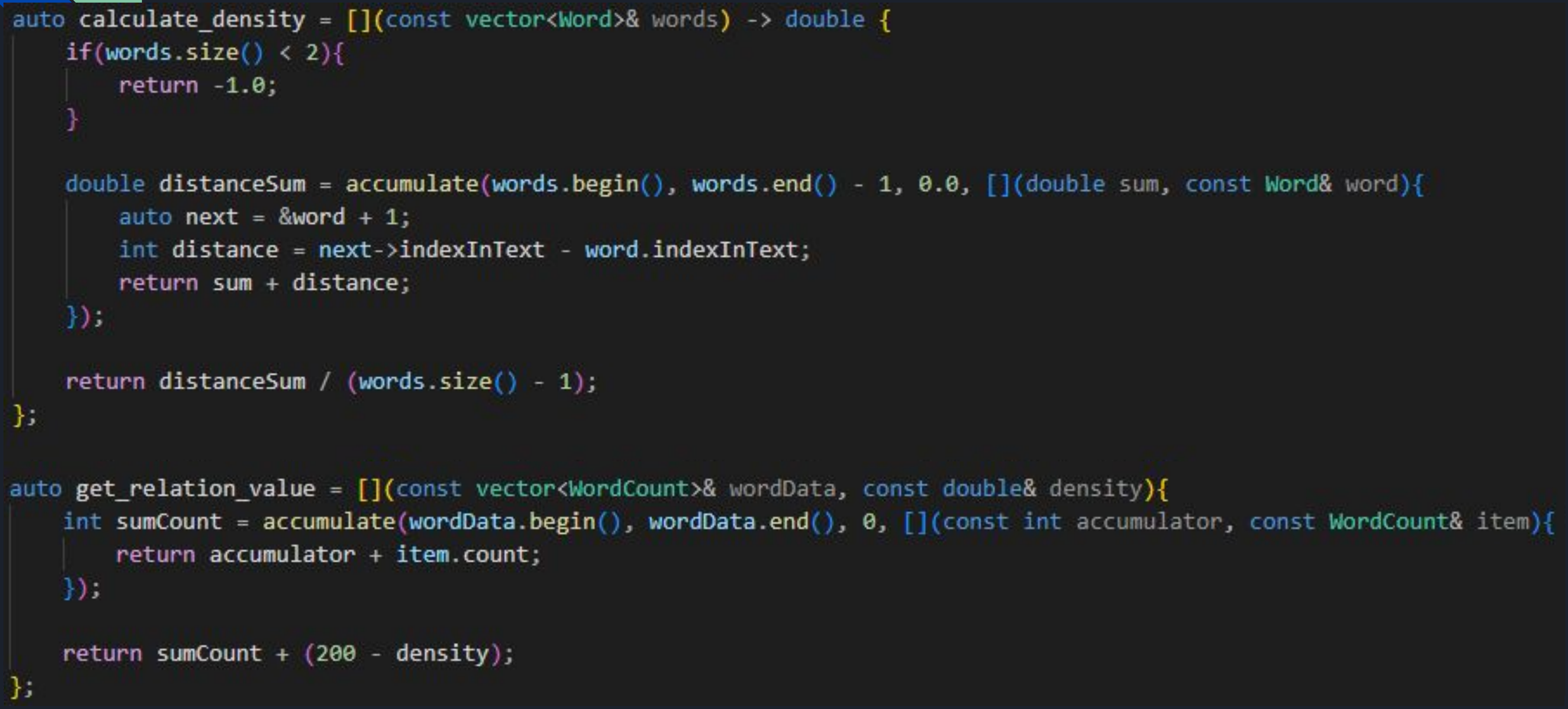


```cpp
struct WordCount{
    string word;
    int count;
};
```



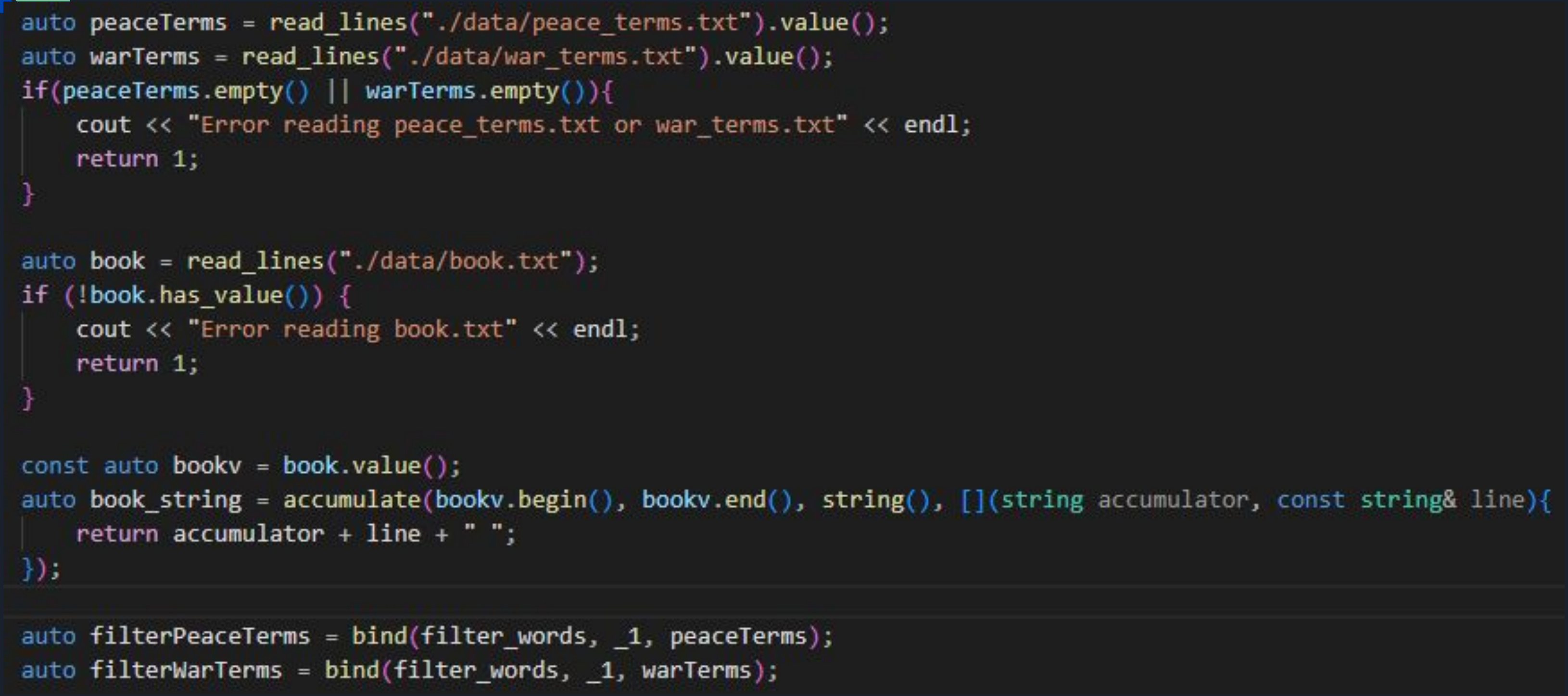
```cpp
auto calculate_wordCount = [](const map<string, vector<Word>>& wordMap) -> vector<WordCount> {
    vector<WordCount> result;

    for_each(wordMap.begin(), wordMap.end(), [&](const pair<string, vector<Word>>& pair){
        int wordCount = pair.second.size();
        WordCount current {pair.first, wordCount};
        result.push_back(current);
    });

    return result;
};
```

Calculate term density



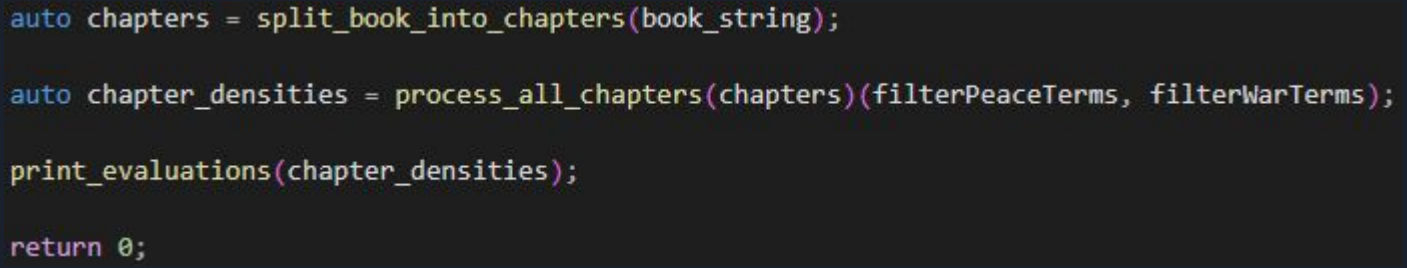
```cpp
auto calculate_density = [](const vector<Word>& words) -> double {
    if(words.size() < 2){
        return -1.0;
    }

    double distanceSum = accumulate(words.begin(), words.end() - 1, 0.0, [](double sum, const Word& word){
        auto next = &word + 1;
        int distance = next->indexInText - word.indexInText;
        return sum + distance;
    });

    return distanceSum / (words.size() - 1);
};

auto get_relation_value = [](const vector<WordCount>& wordData, const double& density){
    int sumCount = accumulate(wordData.begin(), wordData.end(), 0, [](const int accumulator, const WordCount& item){
        return accumulator + item.count;
    });

    return sumCount + (200 - density);
};
```

Main Methode



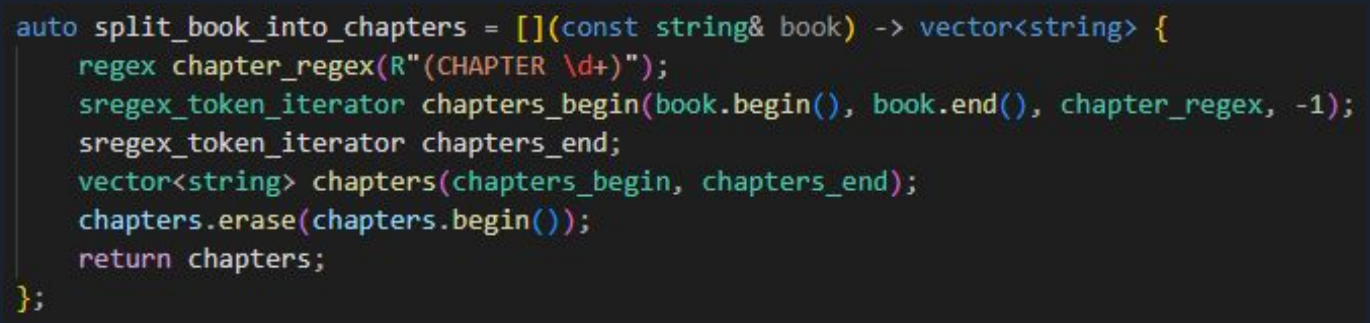
```
auto peaceTerms = read_lines("./data/peace_terms.txt").value();
auto warTerms = read_lines("./data/war_terms.txt").value();
if(peaceTerms.empty() || warTerms.empty()){
    cout << "Error reading peace_terms.txt or war_terms.txt" << endl;
    return 1;
}

auto book = read_lines("./data/book.txt");
if (!book.has_value()) {
    cout << "Error reading book.txt" << endl;
    return 1;
}

const auto bookv = book.value();
auto book_string = accumulate(bookv.begin(), bookv.end(), string(), [](string accumulator, const string& line){
    return accumulator + line + " ";
});

auto filterPeaceTerms = bind(filter_words, _1, peaceTerms);
auto filterWarTerms = bind(filter_words, _1, warTerms);
```

Main Method

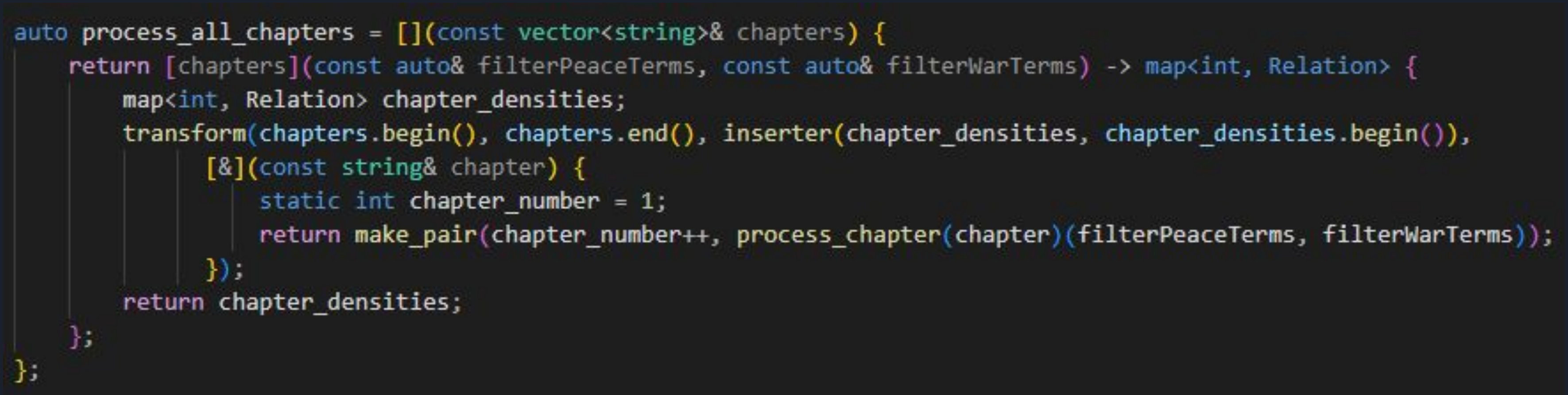


```
auto chapters = split_book_into_chapters(book_string);

auto chapter_densities = process_all_chapters(chapters)(filterPeaceTerms, filterWarTerms);

print_evaluations(chapter_densities);

return 0;
```

Process chapters

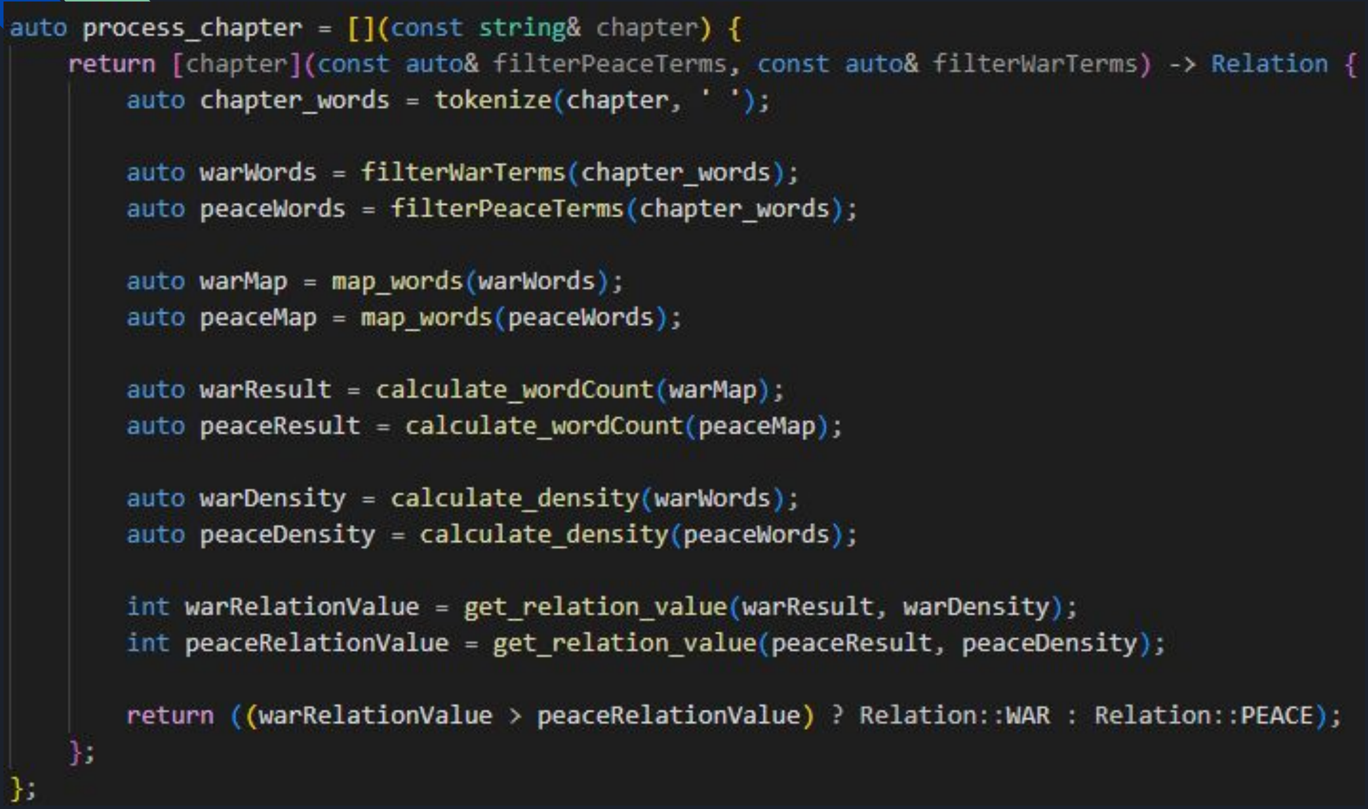


```cpp
auto split_book_into_chapters = [](const string& book) -> vector<string> {
    regex chapter_regex(R"(CHAPTER \d+)");
    sregex_token_iterator chapters_begin(book.begin(), book.end(), chapter_regex, -1);
    sregex_token_iterator chapters_end;
    vector<string> chapters(chapters_begin, chapters_end);
    chapters.erase(chapters.begin());
    return chapters;
};
```

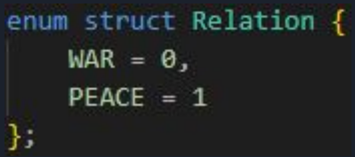


```cpp
auto process_all_chapters = [](const vector<string>& chapters) {
    return [chapters](const auto& filterPeaceTerms, const auto& filterWarTerms) -> map<int, Relation> {
        map<int, Relation> chapter_densities;
        transform(chapters.begin(), chapters.end(), inserter(chapter_densities, chapter_densities.begin()),
              [&](const string& chapter) {
                  static int chapter_number = 1;
                  return make_pair(chapter_number++, process_chapter(chapter)(filterPeaceTerms, filterWarTerms));
              });
        return chapter_densities;
    };
};
```

Process Chapter & categorize chapters



```cpp
auto process_chapter = [](const string& chapter) {
    return [chapter](const auto& filterPeaceTerms, const auto& filterWarTerms) -> Relation {
        auto chapter_words = tokenize(chapter, ' ');

        auto warWords = filterWarTerms(chapter_words);
        auto peaceWords = filterPeaceTerms(chapter_words);

        auto warMap = map_words(warWords);
        auto peaceMap = map_words(peaceWords);

        auto warResult = calculate_wordCount(warMap);
        auto peaceResult = calculate_wordCount(peaceMap);

        auto warDensity = calculate_density(warWords);
        auto peaceDensity = calculate_density(peaceWords);

        int warRelationValue = get_relation_value(warResult, warDensity);
        int peaceRelationValue = get_relation_value(peaceResult, peaceDensity);

        return ((warRelationValue > peaceRelationValue) ? Relation::WAR : Relation::PEACE);
    };
};
```



```cpp
enum struct Relation {
    WAR = 0,
    PEACE = 1
};
```

Danke für Eure Aufmerksamkeit