**Код программы**

**scanner\_exe - exe**

**testTask.cpp**

#include <iostream>

#include "Application.h"

#include <chrono>

int main(int argc, char\* argv[]) {

setlocale(LC\_ALL, "ru\_RU");

std::string hash\_base\_path = "hash\_base\_path";

std::string log\_file = "log\_file";

std::string scan\_dir\_path = "scan\_dir\_path";

for (int i = 1; i < argc; i++) {

std::string arg = argv[i]; // Преобразуем в std::string

if (arg == "--base") {

//std::cout << arg << std::endl;

if (i + 1 < argc) {

hash\_base\_path = argv[i + 1];

i++;

}

}

else if (arg == "--log") {

//std::cout << arg << std::endl;

if (i + 1 < argc) {

log\_file = argv[i + 1];

i++;

}

}

else if (arg == "--path") {

//std::cout << arg << std::endl;

if (i + 1 < argc) {

scan\_dir\_path = argv[i + 1];

i++;

}

}

}

//std::cout << hash\_base\_path << '\n';

//std::cout << log\_file << '\n';

//std::cout << scan\_dir\_path << '\n';

try {

auto start = std::chrono::steady\_clock::now();

std::cout << "Utility started working..." << std::endl;

Application app;

Report report = app.run(hash\_base\_path, log\_file, scan\_dir\_path);

auto end = std::chrono::steady\_clock::now();

std::chrono::duration<double> elapsed = end - start;

double duration\_sec = elapsed.count();

report.setElapsedTime(duration\_sec);

report.printReport();

return 0;

}

catch (const std::exception& e) {

std::cerr << "Fatal error: " << e.what() << '\n';

return 1;

}

}

**Application.h**

#ifndef APPLICATION\_H

#define APPLICATION\_H

#include <string>

#include "Report.h"

// Макрос для экспорта/импорта

#ifdef SCANNER\_LIBRARY\_EXPORTS

#define SCANNER\_API \_\_declspec(dllexport)

#else

#define SCANNER\_API \_\_declspec(dllimport)

#endif

class SCANNER\_API Application {

public:

Report run(const std::string& base, const std::string& log, const std::string& path);

};

#endif // !APPLICATION\_H

**scanner\_** **library - DLL**

**Application.cpp**

#include "Application.h"

#include "Database.h"

#include "HashCalculator.h"

#include "Logger.h"

#include <iostream>

#include "DirectoryScanner.h"

#include <fstream>

#include <chrono>

#include <thread>

Report Application::run(const std::string& base, const std::string& log, const std::string& path)

{

Report report;

Database database(base);

Logger logger(log);

DirectoryScanner scanner(database, logger, report);

scanner.scan(path);

return report;

}

**Database.h**

#ifndef DATABASE\_H

#define DATABASE\_H

#include <unordered\_map>

#include <string>

class Database{

public:

Database(const std::string& csvFilePath);

std::string getVerdict(std::string hash);

private:

void load(const std::string& csvFilePath);

std::unordered\_map<std::string, std::string> maliciousHashes;

};

#endif // !DATABASE\_H

**Database.cpp**

#include "Database.h"

#include <fstream>

#include <filesystem>

#include <iostream>

#include "StringUtils.h"

#include <regex>

Database::Database(const std::string& csvFilePath)

{

load(csvFilePath);

}

void Database::load(const std::string& csvFilePath)

{

if (!std::filesystem::exists(csvFilePath)) {

throw std::runtime\_error("CSV файл не существует: " + csvFilePath);

}

// Проверка что это обычный файл

if (!std::filesystem::is\_regular\_file(csvFilePath)) {

throw std::runtime\_error("Путь не является файлом: " + csvFilePath);

}

std::ifstream inf(csvFilePath);

if (!inf.is\_open()) {

// Проверяем код ошибки для более точного сообщения

if (errno == EACCES) {

throw std::runtime\_error("Недостаточно прав для чтения файла: " + csvFilePath);

}

else {

throw std::runtime\_error("Ошибка открытия файла: " + csvFilePath);

}

}

std::string line;

size\_t delimiterPos;

std::string hash;

std::string verdict;

while (std::getline(inf, line)) {

if (line.empty()) continue; // Пропускаем пустые строки

delimiterPos = line.find(';');

if (delimiterPos == std::string::npos) {

throw std::runtime\_error("Некорректный формат CSV файла: " + line);

}

hash = StringUtils::trim(line.substr(0, delimiterPos));

verdict = StringUtils::trim(line.substr(delimiterPos + 1));

maliciousHashes[hash] = verdict;

}

}

std::string Database::getVerdict(std::string hash)

{

std::string result = "";

std::unordered\_map<std::string, std::string>::const\_iterator it = maliciousHashes.find(hash);

if (it != maliciousHashes.end()) {

result = it->second;

}

return result;

}

**DirectoryScanner.h**

#ifndef DIRECTORY\_SCANNER\_H

#define DIRECTORY\_SCANNER\_H

#include "Database.h"

#include "Logger.h"

#include "Report.h"

#include <string>

#include "fileHandler.h"

class DirectoryScanner {

public:

DirectoryScanner(Database& base, Logger& logger, Report& report);

void scan(std::string path);

private:

Database& database;

Logger& logger;

Report& report;

bool canAccessDirectory(const std::filesystem::path& dir);

void scanDirectory(const std::filesystem::path& directory);

};

#endif // !DIRECTORY\_SCANNER\_H

**DirectoryScanner.cpp**

#include "DirectoryScanner.h"

#include <iostream>

#include <filesystem>

#include <thread>

DirectoryScanner::DirectoryScanner(Database& base, Logger& logger, Report& report) : database(base), logger(logger), report(report) {}//в списке инициализации потому что ссылки

void DirectoryScanner::scan(std::string path)

{

try {

std::filesystem::path rootPath(path);

if (!std::filesystem::exists(rootPath)) {

throw std::runtime\_error("Directory does not exist: " + path);

}

if (!std::filesystem::is\_directory(rootPath)) {

throw std::runtime\_error("Path is not a directory: " + path);

}

scanDirectory(rootPath);

}

catch (const std::runtime\_error& e) {

logger.logError(std::string(e.what()));

throw;//это ловит main

}

}

bool DirectoryScanner::canAccessDirectory(const std::filesystem::path& dir)

{

std::error\_code ec;

std::filesystem::directory\_iterator it(dir, ec);

if (ec) {

return false;

}

return true;

}

void DirectoryScanner::scanDirectory(const std::filesystem::path& directory)

{

FileHandler fileHandler(database, logger, report);

auto options = std::filesystem::directory\_options::skip\_permission\_denied;

std::vector<std::thread> threads;

const size\_t maxThreads = 2;

std::atomic<size\_t> activeThreads{ 0 };

std::mutex cvMutex;

std::condition\_variable cv;

for (const auto& entry : std::filesystem::recursive\_directory\_iterator(directory, options)) {

if (entry.is\_regular\_file()) {

std::unique\_lock<std::mutex> lock(cvMutex);

cv.wait(lock, [&]() { return activeThreads < maxThreads; });

activeThreads++;

threads.emplace\_back([this, path = entry.path().string(), &activeThreads, &cv]() {//emplace\_back - прямое создание , а не перемещение, копирование как в push\_back

try {

FileHandler fileHandler(database, logger, report);//

fileHandler.processFile(path);//

}

catch (const std::exception& e) {

// Логируем ошибку, но не прерываем другие потоки

logger.logError("Ошибка обработки файла " + path + ": " + e.what());

}

activeThreads--;

cv.notify\_one(); // Разблокируем ожидающие потоки

});

}

else {

if (!canAccessDirectory(entry.path())) {

logger.logError("Нет доступа к папке " + entry.path().string());

}

}

}

for (auto& t : threads) {

t.join();

}

}

**FileHandler.h**

#ifndef FILE\_HANDLER\_H

#define FILE\_HANDLER\_H

#include "Database.h"

#include "Logger.h"

#include "Report.h"

#include <string>

class FileHandler {

public:

FileHandler(Database& base, Logger& logger, Report& report);

void processFile(const std::string& filePath);

private:

Database& database;

Logger& logger;

Report& report;

};

#endif // !FILE\_HANDLER\_H

**FileHandler.cpp**

#include "fileHandler.h"

#include "ios"

#include "fstream"

#include <iostream>

#include "HashCalculator.h"

#include <thread>

FileHandler::FileHandler(Database& base, Logger& logger, Report& report) : database(base), logger(logger), report(report) {}

void FileHandler::processFile(const std::string& filePath)

{

//Если не получится открыть то и хеш не посчитаем

report.addFileProcessed();//добавляем "всего обработнных файлов"

std::ifstream file(filePath, std::ios::binary);

if (!file.is\_open()) {

report.addError();//добоавляем "ошибка доступа"

logger.logError("Нет прав на чтение файла: " + filePath);

return;

}

std::string hash = HashCalculator::calcHash(file);

/\*std::cout << std::this\_thread::get\_id() << std::endl;

std::cout << filePath + "\n" + hash + "\n" + "-----------------------------------------------------------------------------------------------"<<std::endl;\*/

std::string verdict = database.getVerdict(hash);

if (verdict != "") {

report.addInfected();

logger.logDetection(filePath, hash, verdict);

}else{

report.addHealthy();

}

}

**HashCalculator.h**

#ifndef HASH\_CALCULATOR\_H

#define HASH\_CALCULATOR\_H

#include <string>

class HashCalculator {

public:

std::string static calcHash(std::ifstream& file);

};

#endif // !HASH\_CALCULATOR\_H

**HashCalculator.cpp**

#include "HashCalculator.h"

#include <openssl/md5.h>

#include <fstream>

#include <sstream>

#include <iomanip>

#include <stdexcept>

std::string HashCalculator::calcHash(std::ifstream& file) {

unsigned char c[MD5\_DIGEST\_LENGTH];

MD5\_CTX mdContext;

MD5\_Init(&mdContext);

// Читаем файл блоками по 4KB

const size\_t bufferSize = 4096;

char buffer[bufferSize];

while (file.good()) {

file.read(buffer, bufferSize);

MD5\_Update(&mdContext, buffer, file.gcount());

}

MD5\_Final(c, &mdContext);

// Переводим байты в hex-строку

std::ostringstream oss;

oss << std::hex << std::setfill('0');

for (int i = 0; i < MD5\_DIGEST\_LENGTH; ++i) {

oss << std::setw(2) << static\_cast<int>(c[i]);

}

return oss.str();

}

**Logger.h**

#ifndef LOGGER\_H

#define LOGGER\_H

#include <string>

#include <fstream>

#include <chrono>

class Logger {

private:

std::ofstream log\_file;

public:

Logger(const std::string& filename);

void logDetection(const std::string& file\_path, const std::string& hash,

const std::string& verdict);

void logError(const std::string& error);

void logInfo(const std::string& message);

~Logger();

};

#endif // !LOGGER\_H

**Logger.cpp**

#include "Logger.h"

#include <stdexcept>

#include <iostream>

#include <iomanip>

Logger::Logger(const std::string& filename)

{

log\_file.open(filename);

if (!log\_file.is\_open()) {

throw std::runtime\_error("Cannot open log file: " + filename);

}

}

void Logger::logDetection(const std::string& file\_path, const std::string& hash, const std::string& verdict)

{

auto now = std::chrono::system\_clock::now();

auto time\_t = std::chrono::system\_clock::to\_time\_t(now);

char buffer[80];

std::strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", std::localtime(&time\_t));

log\_file << "[DETECTED] "

<< buffer << " | "

<< "File: " << file\_path << " | "

<< "Hash: " << hash << " | "

<< "Verdict: " << verdict

<< std::endl;

}

void Logger::logError(const std::string& error)

{

auto now = std::chrono::system\_clock::now();

auto time\_t = std::chrono::system\_clock::to\_time\_t(now);

char buffer[80];

std::strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", std::localtime(&time\_t));

log\_file << "[ERROR] "

<< buffer << " | "

<< "Error: " << error

<< std::endl;

}

void Logger::logInfo(const std::string& message)

{

auto now = std::chrono::system\_clock::now();

auto time\_t = std::chrono::system\_clock::to\_time\_t(now);

char buffer[80];

std::strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", std::localtime(&time\_t));

log\_file << "[INFO] "

<< buffer << " | "

<< message

<< std::endl;

}

Logger::~Logger()

{

if (log\_file.is\_open()) {

log\_file.close();

}

}

#endif // !REPORT\_H

**Report.h**

#ifndef REPORT\_H

#define REPORT\_H

#ifdef SCANNER\_LIBRARY\_EXPORTS

#define SCANNER\_API \_\_declspec(dllexport)

#else

#define SCANNER\_API \_\_declspec(dllimport)

#endif

#include <atomic>

class SCANNER\_API Report {

public:

void addFileProcessed();

void addInfected();

void addHealthy();

void addError();

void setElapsedTime(double seconds);

void printReport();

private:

std::atomic<int> totalFiles{ 0 };

std::atomic<int> infectedFiles{ 0 };

std::atomic<int> healthyFiles{ 0 };

std::atomic<int> errors{ 0 };

double elapsedTime = 0.0;

};

#endif // !REPORT\_H

**Report.cpp**

#include "Report.h"

#include <iostream>

#include <chrono>

#include <iomanip>

void Report::addFileProcessed()

{

totalFiles += 1;

}

void Report::addInfected()

{

infectedFiles += 1;

}

void Report::addHealthy()

{

healthyFiles += 1;

}

void Report::addError()

{

errors += 1;

}

void Report::setElapsedTime(double seconds)

{

elapsedTime = seconds;

}

void Report::printReport()

{

std::cout << "\n=== SCAN REPORT ===\n";

std::cout << "Total files processed: " << totalFiles.load() << "\n";//load явно чиатет атомарные значения

std::cout << "Infected files found: " << infectedFiles.load() << "\n";

std::cout << "Healthy files found: " << healthyFiles.load() << "\n";

std::cout << "Errors encountered: " << errors.load() << "\n";

std::cout << "Time elapsed: " << std::fixed << std::setprecision(2)

<< elapsedTime << " seconds\n";

}

**StringUtils.h**

#ifndef STRING\_UTILS

#define STRING\_UTILS

#include <string>

class StringUtils {

public:

static std::string trim(const std::string& str);

};

#endif // !STRING\_UTILS

**StringUtils.cpp**

#include "StringUtils.h"

std::string StringUtils::trim(const std::string& str)

{

const std::string whitespace = " \t\n\r\f\v";

// Находим первый непробельный символ

size\_t start = str.find\_first\_not\_of(whitespace);

if (start == std::string::npos) return ""; // Вся строка - пробелы

// Находим последний непробельный символ

size\_t end = str.find\_last\_not\_of(whitespace);

return str.substr(start, end - start + 1);

}

**CMAKE**

cmake\_minimum\_required(VERSION 4.1.1 FATAL\_ERROR)

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

if(WIN32)

message(STATUS "Building for Windows")

else()

message(STATUS "Building for Unix-like system")

endif()

project(scaner)#имя проекта

add\_executable(scanner\_exe src/testTask.cpp )#имя исполняемого файла

#set(CMAKE\_WINDOWS\_EXPORT\_ALL\_SYMBOLS ON)#делаю макросы в коде Application.h и Report.h

add\_library(scanner\_library SHARED src/Application.cpp src/Database.cpp

src/Report.cpp src/StringUtils.cpp src/Application.h

src/FileHandler.h

src/FileHandler.cpp

src/Database.h

src/HashCalculator.cpp

src/Logger.cpp

src/DirectoryScanner.cpp

src/HashCalculator.h

src/DirectoryScanner.h

src/Logger.h

src/Report.h

src/StringUtils.h)

source\_group("Header Files" FILES

src/Application.h

src/Database.h

src/HashCalculator.h

src/DirectoryScanner.h

src/FileHandler.h

src/Logger.h

src/Report.h

src/StringUtils.h

)

# Для библиотеки - определяем макрос экспорта

target\_compile\_definitions(scanner\_library PRIVATE SCANNER\_LIBRARY\_EXPORTS)

# Находим установленный OpenSSL

find\_package(OpenSSL REQUIRED)

# Линкуем OpenSSL с нашей библиотекой

target\_link\_libraries(scanner\_library OpenSSL::Crypto)

target\_link\_libraries(scanner\_exe PRIVATE scanner\_library)

add\_custom\_command(TARGET scanner\_exe

POST\_BUILD

COMMAND ${CMAKE\_COMMAND} -E copy

"${CMAKE\_CURRENT\_SOURCE\_DIR}/src/text.csv"

$<TARGET\_FILE\_DIR:scanner\_exe>/text.csv

COMMENT "Copying txt file")