Файл back_sfc\src\main.cpp

```
#include <iostream>
#include <thread>

#include "configuration/chek_conf_file.h"
#include "configuration/config.h"
#include "daemon/deamon_works.h"

flag_menu flagMenu;

int main() {
    setup_logger();
    BOOST_LOG_TRIVIAL(info) << "Application start";

    flagMenu.password  = generate_random_string(20);
    BOOST_LOG_TRIVIAL(info) << "Generated random password";

    daemonize();
    BOOST_LOG_TRIVIAL(info) << "Daemonized the process";

    std::string home = getenv("HOME");
    flagMenu.directory = home + "/control/";
    BOOST_LOG_TRIVIAL(info) << "Set control directory to: " <<
flagMenu.directory;

    std::string config_dir = flagMenu.directory +
"/config/config.json";
    BOOST_LOG_TRIVIAL(info) << "Set config directory to: " <<
config_dir;

    BOOST_LOG_TRIVIAL(info) << "Starting config file watcher
thread";
    std::thread watcher_thread(watch_config_file, config_dir);

    watcher_thread.join();
    BOOST_LOG_TRIVIAL(info) << "Config file watcher thread
joined";

    BOOST_LOG_TRIVIAL(info) << "Application end";
    return 0;
}
```

Файл back_sfc\src\configuration\chek_conf_file.cpp

```
#include "chek_conf_file.h"

#include <chrono>
#include <thread>
#include <sys/stat.h>


void watch_config_file(const std::string& config_path) {
    BOOST_LOG_TRIVIAL(info) << "start watch_config_file";

    int fd = inotify_init();
    if (fd < 0) {
        BOOST_LOG_TRIVIAL(error) << "Failed to initialize
inotify";
    }



    int wd = inotify_add_watch(fd, config_path.c_str(),
IN_MODIFY);
    while (wd < 0) {
        BOOST_LOG_TRIVIAL(error) << "Failed to add watch for " +
config_path;
        BOOST_LOG_TRIVIAL(info) << "Retrying in 3 seconds...";
        std::this_thread::sleep_for(std::chrono::seconds(3)); //
Задержка на 3 секунды
        wd = inotify_add_watch(fd, config_path.c_str(),
IN_MODIFY);
    }

    char buffer[sizeof(struct inotify_event) + NAME_MAX + 1];

    // Parse the config file and execute main_proc at the start
    if (parse_config()) {
        BOOST_LOG_TRIVIAL(info) << "Config file first loaded";
    }

    time_t last_modification_time = 0;
    while (true) {
        ssize_t len = read(fd, buffer, sizeof(buffer));
        if (len < 0) {
            BOOST_LOG_TRIVIAL(error) << "Failed to read inotify
events";
            break;
        }
        auto* event = reinterpret_cast<struct
inotify_event*>(buffer);
        if (event->mask & IN_MODIFY) {
            struct stat attr;
            stat(config_path.c_str(), &attr);
            time_t new_modification_time = attr.st_mtime;

            if (new_modification_time != last_modification_time)
{
                last_modification_time = new_modification_time;

                BOOST_LOG_TRIVIAL(error) << "Config file
modified, reloading...";

std::this_thread::sleep_for(std::chrono::seconds(1)); // Задержка
на 1 секунду

                if (parse_config()) {
                    BOOST_LOG_TRIVIAL(info) << "Config file
reloaded, mabye";

                } else {
                    BOOST_LOG_TRIVIAL(error) << "Failed to parse
config file";
                }
            }
        }
    }

    inotify_rm_watch(fd, wd);
    close(fd);
}
```

Файл back_sfc\src\configuration\config.cpp

```
#include "config.h"

bool parse_config() {
    std::string home_dir = getenv("HOME");
    std::string config_dir = home_dir +
"/control/config/config.json";

    std::ifstream config_file(config_dir, std::ifstream::binary);
    if (!config_file.is_open()) {
        BOOST_LOG_TRIVIAL(error) << "Failed to open config file";
        return false;
    }

    auto buffer_menu = flagMenu;
    flag_menu_changes changes;

    Json::Value root;
    Json::Reader reader;
    bool parsingSuccessful = reader.parse(config_file, root);
    if (!parsingSuccessful) {
        BOOST_LOG_TRIVIAL(error) << "Failed to parse config
file";
        return false;
    }

    flagMenu.protection = root["protection"].asBool();
    flagMenu.force_restore = root["force_restore"].asBool();
    flagMenu.path = root["path"].asString();
    flagMenu.interval = root["interval"].asInt();
    flagMenu.backup_type = root["backup_type"].asString();
    flagMenu.hash_algorithm = root["hash_algorithm"].asString();
    flagMenu.notification_channel =
root["notification_channel"].asString();


    if (buffer_menu.protection != flagMenu.protection) {
        changes.protection_changed = true;
        BOOST_LOG_TRIVIAL(info) << "Protection: " <<
(flagMenu.protection ? "Enabled" : "Disabled");
    }
    if (buffer_menu.force_restore != flagMenu.force_restore) {
        changes.force_restore_changed = true;
        BOOST_LOG_TRIVIAL(info) << "Force Restore: " <<
(flagMenu.force_restore ? "Enabled" : "Disabled");
    }
    if (buffer_menu.path != flagMenu.path) {
        changes.path_changed = true;
        BOOST_LOG_TRIVIAL(info) << "Path: " << flagMenu.path;
    }
    if (buffer_menu.interval != flagMenu.interval) {
        changes.interval_changed = true;
        BOOST_LOG_TRIVIAL(info) << "Interval: " <<
flagMenu.interval;
    }
    if (buffer_menu.backup_type != flagMenu.backup_type) {
        changes.backup_type_changed = true;
        BOOST_LOG_TRIVIAL(info) << "Backup Type: " <<
flagMenu.backup_type;
    }
    if (buffer_menu.hash_algorithm != flagMenu.hash_algorithm) {
        changes.hash_algorithm_changed = true;
        BOOST_LOG_TRIVIAL(info) << "Hash Algorithm: " <<
flagMenu.hash_algorithm;
    }
    if (buffer_menu.notification_channel !=
flagMenu.notification_channel) {
        changes.notification_channel_changed = true;
        BOOST_LOG_TRIVIAL(info) << "Notification Channel: " <<
flagMenu.notification_channel;
    }
    file_info();

    if (changes.path_changed or changes.protection_changed or
changes.interval_changed or changes.backup_type_changed or
changes.hash_algorithm_changed or
changes.notification_channel_changed) {
        send_notification(flagMenu.notification_channel, "Config
file reloaded");
    }

    main_proc(changes);

    return true;
}

void file_info() {
    // Вывод информации о файлах
    BOOST_LOG_TRIVIAL(info) << "||-------------------------------
---------||";
    BOOST_LOG_TRIVIAL(info) << "||-------------- File Information
--------||";
    BOOST_LOG_TRIVIAL(info) << "||-------------------------------
---------||";
    BOOST_LOG_TRIVIAL(info) << "|| Path: " << flagMenu.path;
    BOOST_LOG_TRIVIAL(info) << "|| Force Restore: " <<
(flagMenu.force_restore ? "Enabled" : "Disabled");

    BOOST_LOG_TRIVIAL(info) << "||-------------------------------
---------||";
    BOOST_LOG_TRIVIAL(info) << "|| Protection: " <<
(flagMenu.protection ? "Enabled" : "Disabled");
    BOOST_LOG_TRIVIAL(info) << "|| Interval: " <<
flagMenu.interval;
    BOOST_LOG_TRIVIAL(info) << "|| Backup Type: " <<
flagMenu.backup_type;
    BOOST_LOG_TRIVIAL(info) << "|| Hash Algorithm: " <<
flagMenu.hash_algorithm;
```

```
    BOOST_LOG_TRIVIAL(info) << "|| Notification Channel: " <<
flagMenu.notification_channel;
    BOOST_LOG_TRIVIAL(info) << "||--------------------------------
---------||";
}
```

Файл back_sfc\src\daemon\deamon_works.cpp

```
#include "deamon_works.h"

void daemonize() {
    pid_t pid;

    /* Создаем дочерний процесс */
    pid = fork();

    /* Если не удалось создать дочерний процесс */
    if (pid < 0)
        exit(EXIT_FAILURE);

    /* Если мы получили положительный PID, это означает, что мы
являемся родительским процессом */
    if (pid > 0)
        exit(EXIT_SUCCESS);

    /* Изменяем маску файла, чтобы записи могли быть прочитаны и
записаны правильно */
    umask(0);

    /* Создаем новый сеанс, делая текущий процесс его лидером */
    if (setsid() < 0)
        exit(EXIT_FAILURE);

    /* Изменяем рабочий каталог на корневой каталог */
    if (chdir("/") < 0)
        exit(EXIT_FAILURE);

    /* Закрываем стандартные файловые дескрипторы */
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);
}

bool is_process_running(const char* process_name) {
    char buffer[128];
    std::string command = std::string("pgrep ") + process_name;
    std::string result = "";
    FILE* pipe = popen(command.c_str(), "r");

    if (!pipe) throw std::runtime_error("popen() failed!");

    try {
        while (fgets(buffer, sizeof buffer, pipe) != NULL) {
            result += buffer;
        }
    } catch (...) {
        pclose(pipe);
        throw;
    }

    pclose(pipe);


    return !result.empty();
}
```

Файл back_sfc\src\other\logger.cpp

```
#include "logger.h"


std::ostream& operator<<(std::ostream& strm, const
boost::posix_time::ptime& pt)
{
    if(!pt.is_not_a_date_time())
    {
        boost::posix_time::time_facet* facet = new
boost::posix_time::time_facet();
        facet->format("%Y-%m-%d %H:%M"); // Измените формат
времени здесь
        strm.imbue(std::locale(strm.getloc(), facet));
        strm << pt;
    }
    return strm;
}

void setup_logger() {
    auto sink_backend =
boost::make_shared<sinks::text_file_backend>(
            keywords::file_name = "logs.txt",
            keywords::auto_flush = true  // Enable auto flushing
after each log record
```

```
    );

    auto sink =
boost::make_shared<sinks::synchronous_sink<sinks::text_file_backe
nd>>(sink_backend);

    sink->set_formatter(
            expr::format("[%1%] [%2%] %3%")
            % expr::attr< boost::posix_time::ptime >("TimeStamp")
            % expr::attr< unsigned int >("LineID")
            % expr::smessage
    );

    logging::core::get()->add_sink(sink);

    logging::add_common_attributes();  // Add common attributes
such as TimeStamp and LineID

    // Now all logs will be flushed immediately
    BOOST_LOG_TRIVIAL(info) << "information";
}
```

Файл back_sfc\src\other\notification.cpp

```
#include "notification.h"


void send_notification(std::string &notification_channel, const
std::string &message) {
    if (notification_channel == "system") {
        std::string cmd = "notify-send 'System File Control' '" +
message + "'";
        system(cmd.c_str());
    }
}

void send_critical_urgency_notification(std::string
&notification_channel, const std::string &message) {
    if (notification_channel == "system") {
        std::string cmd = "notify-send -u critical 'System File
Control' '" + message + "'";
        system(cmd.c_str());
    }
}
```

Файл back_sfc\src\proc\backup.cpp

```
#include "main_proc.h"

#include <boost/process.hpp>
#include <boost/filesystem.hpp>

void begin_backup() {
    full_backup(flagMenu.path, flagMenu.directory);
}

// Создание полного бэкапа. +
void full_backup(const std::string &source_directory, const
std::string &backup_directory) {
    boost::filesystem::path backup_dir(backup_directory);
    if (!boost::filesystem::exists(backup_dir)) {
        boost::filesystem::create_directories(backup_dir);
    }

    std::string source = source_directory;
    std::string destination = backup_directory;
    std::string backup_file = destination + "backup.tar";
    std::string compressed_file = backup_file + ".gz";
    std::string encrypted_file = compressed_file + ".enc";

    //архивация директории
    std::string tar_command = "tar -cf " + backup_file + " -C " +
source + " .";
    boost::process::system(tar_command);


    //сжатие
    std::string gzip_command = "gzip " + backup_file;
    boost::process::system(gzip_command);

    //шифрование
    if (boost::filesystem::exists(encrypted_file)) {
        boost::filesystem::remove(encrypted_file);
    }
    to_encrypt_file(compressed_file, flagMenu.password);

    //удаление сжатого архива
    boost::filesystem::remove(compressed_file);

    //ток чтение владельцем
    boost::filesystem::permissions(encrypted_file,
boost::filesystem::perms::owner_read);
```

```
    send_notification(flagMenu.notification_channel, "Full backup
created");
}

// Восстановление бэкапа
void restore_backup() {
    if (flagMenu.backup_type == "full") {
        BOOST_LOG_TRIVIAL(info) << "Performing full backup
restore";
        full_restore_backup(flagMenu.path, flagMenu.directory);
    } else if (flagMenu.backup_type == "differential") {
        differential_restore_backup(flagMenu.path,
flagMenu.directory);
    }
}

// Восстановление полного бэкапа. +
void full_restore_backup(const std::string &source_directory,
const std::string &backup_directory) {
    BOOST_LOG_TRIVIAL(info) << "Restoring full backup";
    std::string destination = backup_directory;
    std::string encrypted_file = destination +
"backup.tar.gz.enc";
    std::string decrypted_file = destination + "backup.tar.gz";
    std::string extracted_file = destination + "backup.tar";

    if (!boost::filesystem::exists(encrypted_file)) {
        BOOST_LOG_TRIVIAL(error) << "Encrypted backup file does
not exist: " << encrypted_file;

    }
    boost::filesystem::permissions(encrypted_file,
boost::filesystem::perms::all_all);

    to_decrypt_file(encrypted_file, flagMenu.password);

    boost::filesystem::remove(encrypted_file);

    if (!boost::filesystem::exists(decrypted_file)) {
        BOOST_LOG_TRIVIAL(error) << "Decrypted backup file does
not exist: " << decrypted_file;

    }

    std::string gunzip_command = "gunzip " + decrypted_file;
    int gunzip_status = boost::process::system(gunzip_command);

    if (gunzip_status != 0) {
        BOOST_LOG_TRIVIAL(error) << "Failed to extract the backup
file: " << decrypted_file;

    }

    if (!boost::filesystem::exists(source_directory)) {
        boost::filesystem::create_directories(source_directory);
    } else {
        boost::filesystem::remove_all(source_directory);
        boost::filesystem::create_directories(source_directory);
    }

    std::string tar_command = "tar -oxf " + extracted_file + " -C
\"" + source_directory + "\"";
    int tar_status = boost::process::system(tar_command);

    if (tar_status != 0) {
        BOOST_LOG_TRIVIAL(error) << "Failed to extract the backup
file to the source directory: " << source_directory;

    }

    boost::filesystem::remove(decrypted_file);
    boost::filesystem::remove(extracted_file);

    send_notification(flagMenu.notification_channel, "Full backup
restored");
}

// Восстановление дифференциального бэкапа.
void differential_restore_backup(const std::string
&source_directory, const std::string &backup_directory) {
    BOOST_LOG_TRIVIAL(error) << "differential_restore_backup
start";
    std::string destination = backup_directory;
    std::string encrypted_file = destination +
"backup.tar.gz.enc";
    std::string decrypted_file = destination + "backup.tar.gz";
    std::string extracted_file = destination + "backup.tar";

    // Проверка существования зашифрованного файла перед его
расшифровкой
    if (!boost::filesystem::exists(encrypted_file)) {
        BOOST_LOG_TRIVIAL(error) << "Encrypted backup file does
not exist: " << encrypted_file;
    }

    boost::filesystem::permissions(encrypted_file,
boost::filesystem::perms::all_all);

    to_decrypt_file(encrypted_file, flagMenu.password);

    std::string gunzip_command = "gunzip " + decrypted_file;
    boost::process::system(gunzip_command);

    // Создание исходного каталога, если он не существует
    if (!boost::filesystem::exists(source_directory)) {
        boost::filesystem::create_directories(source_directory);
    }
    std::string tar_command = "tar -oxf " + extracted_file + " -C
\"" + source_directory + "\"";
    boost::process::system(tar_command);

    // Удаление расшифрованных и извлеченных файлов после
восстановления
    boost::filesystem::remove(decrypted_file);
    boost::filesystem::remove(extracted_file);

    send_notification(flagMenu.notification_channel,
"Differential backup restored");
}

// Принудительное восстановление, если требуется
void force_restore_if_needed() {
    if (flagMenu.protection) {
        if (flagMenu.force_restore) {
            if (flagMenu.backup_type == "full") {
                force_full_restore(flagMenu.path,
flagMenu.directory);
            } else if (flagMenu.backup_type == "differential") {
                force_differential_restore(flagMenu.path,
flagMenu.directory);
            }
            flagMenu.force_restore = false;
        }
    } else {
        send_notification(flagMenu.notification_channel, "You
cannot recover files without protection enabled");
    }
}


// Принудительное восстановление полного бэкапа
void force_full_restore(const std::string &source_directory,
const std::string &backup_directory) {
    BOOST_LOG_TRIVIAL(info) << "Forcing full backup restore";
    std::string destination = backup_directory;
    std::string encrypted_file = destination +
"backup.tar.gz.enc";
    std::string decrypted_file = destination + "backup.tar.gz";
    std::string extracted_file = destination + "backup.tar";

    // Проверка существования зашифрованного файла перед его
расшифровкой
    if (!boost::filesystem::exists(encrypted_file)) {
        BOOST_LOG_TRIVIAL(error) << "Encrypted backup file does
not exist: " << encrypted_file;
    }

    boost::filesystem::permissions(encrypted_file,
boost::filesystem::perms::all_all);

    to_decrypt_file(encrypted_file, flagMenu.password);


    // Проверка существования расшифрованного файла перед его
извлечением
    if (!boost::filesystem::exists(decrypted_file)) {
        BOOST_LOG_TRIVIAL(error) << "Decrypted backup file does
not exist: " << decrypted_file;
    }

    std::string gunzip_command = "gunzip " + decrypted_file;
    boost::process::system(gunzip_command);

    // Создание исходного каталога, если он не существует
    if (!boost::filesystem::exists(source_directory)) {
        boost::filesystem::create_directories(source_directory);
    } else {
        // Удаление всех файлов и подкаталогов в исходном
каталоге
        boost::filesystem::remove_all(source_directory);
        boost::filesystem::create_directories(source_directory);
    }

    std::string tar_command = "tar -oxf " + extracted_file + " -C
\"" + source_directory + "\"";
    boost::process::system(tar_command);

    //ток чтение владельцем
    boost::filesystem::permissions(encrypted_file,
boost::filesystem::perms::owner_read);

    // Удаление расшифрованных и извлеченных файлов после
```

```
восстановления
    boost::filesystem::remove(decrypted_file);
    boost::filesystem::remove(extracted_file);

    send_notification(flagMenu.notification_channel, "Full backup
restored");
}

// Принудительное восстановление дифференциального бэкапа
void force_differential_restore(const std::string
&source_directory, const std::string &backup_directory) {
    BOOST_LOG_TRIVIAL(info) << "Forcing differential backup
restore";
    std::string destination = backup_directory;
    std::string encrypted_file = destination +
"backup.tar.gz.enc";
    std::string decrypted_file = destination + "backup.tar.gz";
    std::string extracted_file = destination + "backup.tar";

    // Проверка существования зашифрованного файла перед его
расшифровкой
    if (!boost::filesystem::exists(encrypted_file)) {
        BOOST_LOG_TRIVIAL(error) << "Encrypted backup file does
not exist: " << encrypted_file;
    }

    boost::filesystem::permissions(encrypted_file,
boost::filesystem::perms::all_all);


    to_decrypt_file(encrypted_file, flagMenu.password);

    // Проверка существования расшифрованного файла перед его
извлечением
    if (!boost::filesystem::exists(decrypted_file)) {
        BOOST_LOG_TRIVIAL(error) << "Decrypted backup file does
not exist: " << decrypted_file;
    }

    std::string gunzip_command = "gunzip " + decrypted_file;
    boost::process::system(gunzip_command);

    // Создание исходного каталога, если он не существует
    if (!boost::filesystem::exists(source_directory)) {
        boost::filesystem::create_directories(source_directory);
    }
    std::string tar_command = "tar -oxf " + extracted_file + " -C
\"" + source_directory + "\"";
```

```
    boost::process::system(tar_command);

    //ток чтение владельцем
    boost::filesystem::permissions(encrypted_file,
boost::filesystem::perms::owner_read);

    // Удаление расшифрованных и извлеченных файлов после
восстановления
    boost::filesystem::remove(decrypted_file);
    boost::filesystem::remove(extracted_file);

    send_notification(flagMenu.notification_channel,
"Differential backup restored");
}
```

Файл back_sfc\src\proc\encrypted.cpp

```
#include "main_proc.h"

#include <openssl/evp.h>
#include <boost/filesystem.hpp>

void to_encrypt_file(const std::string& filename, const
std::string& password) {

    std::string encrypted_file = filename + ".enc";
    if (boost::filesystem::exists(encrypted_file)) {
        boost::filesystem::remove(encrypted_file);
    }

    std::ifstream inFile(filename, std::ios::binary);
    std::ofstream outFile(filename + ".enc", std::ios::binary);

    const EVP_CIPHER* cipherType = EVP_aes_256_cbc();
    const EVP_MD* digestType = EVP_sha256();
    unsigned char key[EVP_MAX_KEY_LENGTH];
    unsigned char iv[EVP_MAX_IV_LENGTH];

    EVP_BytesToKey(cipherType, digestType, nullptr,
                   reinterpret_cast<const unsigned
char*>(password.c_str()), password.size(), 1, key, iv);

    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    EVP_EncryptInit_ex(ctx, cipherType, nullptr, key, iv);

    char buffer[4096];
    unsigned char bufferOut[4096 + EVP_MAX_BLOCK_LENGTH];
    int numRead, numCrypted;

    while (inFile.read(buffer, sizeof(buffer)), numRead =
inFile.gcount()) {
        EVP_EncryptUpdate(ctx, bufferOut, &numCrypted,
reinterpret_cast<unsigned char*>(buffer), numRead);
        outFile.write(reinterpret_cast<char*>(bufferOut),
numCrypted);
    }

    EVP_EncryptFinal_ex(ctx, bufferOut, &numCrypted);
    outFile.write(reinterpret_cast<char*>(bufferOut),
numCrypted);

    EVP_CIPHER_CTX_free(ctx);
}

void to_decrypt_file(const std::string& filename, const
std::string& password) {
    std::ifstream inFile(filename, std::ios::binary);
    std::ofstream outFile(filename.substr(0, filename.size() -
4), std::ios::binary);

    const EVP_CIPHER* cipherType = EVP_aes_256_cbc();
    const EVP_MD* digestType = EVP_sha256();
    unsigned char key[EVP_MAX_KEY_LENGTH];
    unsigned char iv[EVP_MAX_IV_LENGTH];

    EVP_BytesToKey(cipherType, digestType, nullptr,
                   reinterpret_cast<const unsigned
char*>(password.c_str()), password.size(), 1, key, iv);

    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    EVP_DecryptInit_ex(ctx, cipherType, nullptr, key, iv);

    char buffer[4096];
    unsigned char bufferOut[4096 + EVP_MAX_BLOCK_LENGTH];
    int numRead, numDecrypted;

    while (inFile.read(buffer, sizeof(buffer)), numRead =
inFile.gcount()) {
        EVP_DecryptUpdate(ctx, bufferOut, &numDecrypted,
reinterpret_cast<unsigned char*>(buffer), numRead);
        outFile.write(reinterpret_cast<char*>(bufferOut),
numDecrypted);
    }
    EVP_DecryptFinal_ex(ctx, bufferOut, &numDecrypted);
    outFile.write(reinterpret_cast<char*>(bufferOut),
numDecrypted);

    EVP_CIPHER_CTX_free(ctx);
}
```

Файл back_sfc\src\proc\hash.cpp

```
#include "main_proc.h"

#include <filesystem>
#include <unordered_map>
#include <iostream>
#include <thread>
#include <chrono>

std::unordered_map<std::string, std::pair<std::string,
std::string>> file_hashes;
std::unordered_map<std::string, std::pair<std::string,
std::string>> new_file_hashes;
std::atomic<bool> check_interval(true);

void begin_hash() {
    file_hashes.clear();
    calculate_and_store_hashes(flagMenu.path, file_hashes);
//первичное
}

// обход директории и сохранение хешей в мапу
void calculate_and_store_hashes(const std::string &directory,
std::unordered_map<std::string, std::pair<std::string,
std::string>> &map_hashes) {
    for (const auto &entry:
std::filesystem::recursive_directory_iterator(directory)) {
        if (entry.is_regular_file()) {
            std::string path = entry.path().string();
            std::pair<std::string, std::string> hashes =
calculate_hashes(path);
            map_hashes[path] = hashes; // assign the pair of
hashes
        }
    }
}
```

```
std::string exec(const char* cmd) {
    std::array<char, 128> buffer;
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd,
"r"), pclose);
    if (!pipe) {
        throw std::runtime_error("popen() failed!");
    }
    while (fgets(buffer.data(), buffer.size(), pipe.get()) !=
nullptr) {
        result += buffer.data();
    }
    return result;
}

std::string calculate_md5_hash(const std::string &path) {
    std::string command = "md5sum \"" + path + "\"";
    std::string result = exec(command.c_str());
    return result.substr(0, result.find(' ')); // md5sum returns
the hash followed by the filename
}

std::string calculate_sha256_hash(const std::string &path) {
    std::string command = "sha256sum \"" + path + "\"";
    std::string result = exec(command.c_str());
    return result.substr(0, result.find(' ')); // sha256sum
returns the hash followed by the filename
}

std::pair<std::string, std::string> calculate_hashes(const
std::string &path) {
    std::string md5_hash = calculate_md5_hash(path);
    std::string sha256_hash = calculate_sha256_hash(path);
    return std::make_pair(md5_hash, sha256_hash);
}

void check_file(const std::string &directory) {
    check_interval = true;

    std::thread([directory]() {
        sleep(5);
        while (check_interval) {
            new_file_hashes.clear();
            calculate_and_store_hashes(directory,
new_file_hashes);

            for (const auto &entry : new_file_hashes) {
                if (file_hashes.find(entry.first) ==
file_hashes.end()) {
                    BOOST_LOG_TRIVIAL(error) << "Новый файл
обнаружен: " << entry.first << "\n";

send_critical_urgency_notification(flagMenu.notification_channel,
"Attention! New file found: " + entry.first);

                } else {
                    if (flagMenu.hash_algorithm == "MD5") {
                        if (file_hashes[entry.first].first !=
entry.second.first) {
                            BOOST_LOG_TRIVIAL(error) <<
"Несоответствие MD5 хеша для файла: " << entry.first << "\n";

send_critical_urgency_notification(flagMenu.notification_channel,
"Attention! File: " + entry.first + " changed (hash MD5
mismatch)");
                        }
                    } else {
                        if (file_hashes[entry.first].second !=
entry.second.second) {
                            BOOST_LOG_TRIVIAL(error) <<
"Несоответствие SHA256 хеша для файла: " << entry.first << "\n";

send_critical_urgency_notification(flagMenu.notification_channel,
"Attention! File: " + entry.first + " changed (hash SHA256
mismatch)");
                        }
                    }
                }
            }

            for (const auto &entry : file_hashes) {
                if (new_file_hashes.find(entry.first) ==
new_file_hashes.end()) {
                    std::cout << "Файл не найден при новом
сканировании: " << entry.first << "\n";

send_critical_urgency_notification(flagMenu.notification_channel,
"Attention! File not found: " + entry.first);
                }
            }

            std::this_thread::sleep_for(std::chrono::minutes
(flagMenu.interval)); // Check every minute
        }
    }).detach();
}
```

```
void stop_check_file() {
    std::cout << "Остановка проверки файлов\n";
    send_notification(flagMenu.notification_channel, "File check
stopped");
    check_interval = false;
}
```

Файл back_sfc\src\proc\main_proc.cpp

```
#include "main_proc.h"

void main_proc(flag_menu_changes flagMenuChanges) {
    if (flagMenuChanges.force_restore_changed){
        if (flagMenu.force_restore){
            force_restore_if_needed();
        }
    }
    if (flagMenuChanges.protection_changed or
flagMenuChanges.path_changed){
        if (flagMenu.protection){
            begin_hash();
            begin_backup();
            check_file(flagMenu.path);
        } else{
            restore_backup();
            stop_check_file();
        }
        if (flagMenuChanges.path_changed){
            restore_backup();
            stop_check_file();

            begin_hash();
            begin_backup();
            check_file(flagMenu.path);
        }
    }

}


std::string generate_random_string(size_t length) {
    auto randchar = []() -> char {
        const char charset[] =
                "0123456789"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                "abcdefghijklmnopqrstuvwxyz";
        const size_t max_index = (sizeof(charset) - 1);
        return charset[rand() % max_index];
    };
    std::string str(length, 0);
    std::generate_n(str.begin(), length, randchar);
    return str;
}
```

Файл front\front_sfc\config.cpp

```
#include "config.h"

bool file_exists(const std::string& name) {
    struct stat buffer;
    return (stat(name.c_str(), &buffer) == 0);
}

void update_config_file() {
    // Создание директории
    std::string home_dir = getenv("HOME");
    std::string config_dir = home_dir + "/control/config";
    std::string mkdir_command = "mkdir -p " + config_dir;
    system(mkdir_command.c_str());

    // Создание JSON объекта
    Json::Value root;

    // Проверка существования файла и чтение его, если он 
существует
    std::string config_file_path = config_dir + "/config.json";
    if (!file_exists(config_file_path)) {
        std::string touch_command = "touch " + config_file_path;
        system(touch_command.c_str());
    }
    if (file_exists(config_file_path)) {
        std::ifstream config_file_in(config_file_path, 
std::ios::in);
        if (config_file_in && config_file_in.peek() != 
std::ifstream::traits_type::eof()) {
            config_file_in >> root;
            config_file_in.close();
        } else{
            root["protection"] = false;
            root["path"] = home_dir;
            root["interval"] = 3;
            root["backup_type"] = "full";
            root["hash_algorithm"] = "MD5";
            root["notification_channel"] = "system";
            root["force_restore"] = false;
        }
    }
    // Добавление информации в конфигурационный файл, если флаг
установлен
    if (flagMenu.flag_protection) root["protection"] =
flagMenu.protection;
    if (flagMenu.flag_path) root["path"] = flagMenu.path;
    if (flagMenu.flag_interval) root["interval"] =
flagMenu.interval;
    if (flagMenu.flag_backup_type) root["backup_type"] =
flagMenu.backup_type;
    if (flagMenu.flag_hash_algorithm) root["hash_algorithm"] =
flagMenu.hash_algorithm;
    if (flagMenu.flag_notification_channel)
root["notification_channel"] = flagMenu.notification_channel;
    if (flagMenu.flag_force_restore) root["force_restore"] =
flagMenu.force_restore;
    // Запись обновленного JSON объекта обратно в файл
    std::ofstream config_file_out(config_file_path, std::ios::out
| std::ios::trunc);
    if (!config_file_out) {
        std::cerr << "I/O error while opening file: " <<
config_file_path << std::endl;
        return;
    }

    config_file_out << root;
    config_file_out.close();
}

void read_config_file(){
    std::string home_dir = getenv("HOME");
    std::string config_dir = home_dir +
"/control/config/config.json";

    std::ifstream config_file(config_dir, std::ifstream::binary);
    if (!config_file.is_open()) {
        return;
    }

    Json::Value root;
    Json::Reader reader;
    bool parsingSuccessful = reader.parse(config_file, root);
    if (!parsingSuccessful) {
        return;
    }

    bufferMenu.protection = root["protection"].asBool();
    bufferMenu.force_restore = root["force_restore"].asBool();
    bufferMenu.path = root["path"].asString();
    bufferMenu.interval = root["interval"].asInt();
    bufferMenu.backup_type = root["backup_type"].asString();
    bufferMenu.hash_algorithm =
root["hash_algorithm"].asString();
    bufferMenu.notification_channel =
root["notification_channel"].asString();


}
```

Файл front\front_sfc\main.cpp

```
#include <iostream>
#include "menu.h"

/** Тэкс
* Что может пойти не так:
* - безопасность при использованиии файла конфигурации
* - не может создать директорию в HOME
* - немного стремный дизайн
*
* Что работает:
* - проверочки, выводы всякие, сохранение - все норм
* */

 int main(int argc, char* argv[]) {
        int help_me;
        help_me = menu_arg_main(argc, argv);
    return 0;
}
```

Файл front\front_sfc\menu.cpp

```
#include "menu.h"
```

```
flag_menu flagMenu;
buffer_menu bufferMenu;

int menu_arg_main(int argc, char* argv[]) {
    if (argc > 1) {
        process_command_line_options(argc, argv);
    } else {
        process_menu_options();
    }
    return 0;
}

void process_command_line_options(int argc, char* argv[]) {
    po::options_description desc("Options");
    desc.add_options()
            ("enable", "Enable protection")
            ("disable", "Disable protection")
            ("file-info", "Get information about protected
files")
            ("force-restore", "Force file restoration")
            ("help", "Show help")
            ("exit", "Exit the program")
            ("path", po::value<std::string>(), "Specify the file
path")
            ("interval", po::value<int>(), "Specify the time file
check interval (1, 2, ...180 , minute)")
            ("backup-type", po::value<std::string>(), "Specify
the backup type (full/differential")
            ("hash-algorithm", po::value<std::string>(), "Specify
the hashing algorithm (MD5/SHA256)")
            ("notification-channel", po::value<std::string>(),
"Specify the notification channel (no/system)");

    po::variables_map vm;
    try {
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);

    if (vm.count("enable")) {
        enable_protection();
    } else if (vm.count("disable")) {
        disable_protection();
    } else if (vm.count("file-info")) {
        get_file_info();
    } else if (vm.count("force-restore")) {
        force_file_restore();
    } else if (vm.count("help")) {
        show_help_info(desc);
    } else if (vm.count("exit")) {
        exit_program();
    } else if (vm.count("path")) {
        set_file_path(vm);
    } else if (vm.count("interval")) {
        set_file_check_interval(vm);
    } else if (vm.count("backup-type")) {
        set_backup_type(vm);
    } else if (vm.count("hash-algorithm")) {
        set_hash_algorithm(vm);
    } else if (vm.count("notification-channel")) {
        set_notification_channel(vm);
    } else {
        std::cout << "Unrecognized command. Use --help to see
available commands.\n";
    }

        update_config_file();

    } catch (boost::program_options::unknown_option& e) {
        std::cout << "Error: " << e.what() << "\n";
        std::cout << "Use --help to see available commands.\n";
        return;
    }
}

void process_menu_options() {
    int choice;
    do {
        system("clear");
        std::cout << "=====================================\n";
        std::cout << "|| 1. Enable protection           ||\n";
        std::cout << "|| 2. Disable protection          ||\n";
        std::cout << "|| 3. Get information about files ||\n";
        std::cout << "|| 4. Force file restoration      ||\n";
        std::cout << "|| 5. Configure protection        ||\n";
        std::cout << "|| 6. Show help                   ||\n";
        std::cout << "|| 7. Exit the program            ||\n";
        std::cout << "=====================================\n";
        std::cout << "Enter your choice: ";
        std::cin >> choice;

        if (std::cin.fail()) {
            std::cin.clear();

std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
            continue;
        }

        switch (choice) {
            case 1:
                enable_protection();
                break;
            case 2:
                disable_protection();
                break;
            case 3:
                get_file_info();
                break;
            case 4:
                force_file_restore();
                break;
            case 5:
                configure_protection();
                break;
            case 6:
                show_help_menu();
                break;
            case 7:
                std::cout << "Exit the program\n";
                update_config_file();
                std::exit(0);
            default:
                std::cout << "Invalid choice. Please enter a
number between 1 and 7.\n";
                break;
        }
    } while (choice != 7);
}

void configure_protection() {
    int config_choice;
    do {
        system("clear");
        std::cout << "=======================================\n";
        std::cout << "|| 1. Specify the file path
||\n";
        std::cout << "|| 2. Specify the file check interval
||\n";
        std::cout << "|| 3. Specify the backup type
||\n";
        std::cout << "|| 4. Specify the hashing algorithm
||\n";
        std::cout << "|| 5. Specify the notification
channel||\n";
```

```
        std::cout << "|| 6. Return to main menu
||\n";
        std::cout << "=========================================\n";
        std::cout << "Enter your choice: ";
        std::cin >> config_choice;

        // Проверка на ввод
        if (std::cin.fail()) {
            std::cin.clear(); // очистка состояния ошибки

std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n'); // пропуск некорректного ввода
            continue; // возврат к началу цикла
        }

        switch (config_choice) {
            case 1: {
                std::cout << "Enter the file path: ";
                std::string buffer;
                std::cin >> buffer;
                if (std::filesystem::exists(buffer)) {
                    flagMenu.path = buffer;
                    flagMenu.flag_path = 1;
                } else {
                    std::cout << "The file or directory does not
exist.\n";
                }
                ignore_cin();
                break;
            }
            case 2: {
                std::cout << "Enter the file check interval, 1 <=
{} <= 180 minute: ";
                int buffer;
                std::cin >> buffer;
                if (buffer >= 1 && buffer <= 180) {
                    flagMenu.interval = buffer;
                    flagMenu.flag_interval = 1;
                } else {
                    std::cout << "Invalid interval. Please enter
a number between 1 and 180.\n";
                }
                ignore_cin();
                break;
            }
            case 3: {
                std::cout << "Enter the backup type
(full/differential): ";
```

```
                std::string buffer;
                std::cin >> buffer;
                if (buffer == "full" || buffer == "differential")
{
                    flagMenu.backup_type = buffer;
                    flagMenu.flag_backup_type = 1;
                } else {
                    std::cout << "Invalid input. Please enter
'full' or 'differential'.\n";
                }
                ignore_cin();
                break;
            }
            case 4:{
                std::cout << "Enter the hashing algorithm
(MD5/SHA256): ";
                std::string buffer;
                std::cin >> buffer;
                if (buffer == "MD5" || buffer == "SHA256") {
                    flagMenu.hash_algorithm = buffer;
                    flagMenu.flag_hash_algorithm = 1;
                } else {
                    std::cout << "Invalid input. Please enter
'MD5' or 'SHA256'.\n";
                }
                ignore_cin();
                break;
            }
            case 5:{
                std::cout << "Enter the notification channel
(no/system): ";
                std::string buffer;
                std::cin >> buffer;
                if (buffer == "no" || buffer == "system") {
                    flagMenu.notification_channel = buffer;
                    flagMenu.flag_notification_channel = 1;
                } else {
                    std::cout << "Invalid input. Please enter
'no' or 'system'.\n";
                }
                ignore_cin();
                break;
            }
            case 6:
                break;
            default:
                std::cout << "Invalid choice. Please enter a
number between 1 and 6.\n";
```

```
                ignore_cin();
                break;
        }
    } while (config_choice != 6);
}
```

Файл front\front_sfc\menu_func.cpp

```
#include "menu_func.h"

void enable_protection() {
    flagMenu.protection = true;
    flagMenu.flag_protection = 1;
    std::cout << "Enable protection \n";
    ignore_cin();
}

void disable_protection() {
    flagMenu.protection = false;
    flagMenu.flag_protection = 1;
    std::cout << "Disable protection \n";
    ignore_cin();
}

void get_file_info() {
    std::cout << "Get information about protected files \n";
    file_info();
}

void force_file_restore() {
    std::cout << "Force file restoration.\n";
    flagMenu.force_restore = true;
    flagMenu.flag_force_restore = 1;
    ignore_cin();
}

void show_help_info(po::options_description& desc) {
    std::cout << desc << std::endl;
    ignore_cin();
}

void exit_program() {
    std::cout << "Exit the program\n";
    std::exit(0);
}
```

```
void set_file_path(po::variables_map& vm) {
    std::string buffer = vm["path"].as<std::string>();
    if (std::filesystem::exists(buffer)) {
        flagMenu.path = buffer;
        flagMenu.flag_path = 1;
        std::cout << "Path: " << flagMenu.path << "\n";
    } else {
        std::cout << "The file or directory does not exist.\n";
    }
    ignore_cin();
}

void set_file_check_interval(po::variables_map& vm) {
    int buffer = vm["interval"].as<int>();
    if (buffer >= 1 && buffer <= 180) {
        flagMenu.interval = buffer;
        flagMenu.flag_interval = 1;
        std::cout << "Interval: " << flagMenu.interval << "
minute \n";
    } else {
        std::cout << "Invalid interval. Please enter a number
between 1 and 180.\n";
    }
    ignore_cin();
}

void set_backup_type(po::variables_map& vm) {
    std::string buffer = vm["backup-type"].as<std::string>();
    if (buffer == "full" || buffer == "differential") {
        flagMenu.backup_type = buffer;
        flagMenu.flag_backup_type = 1;
        std::cout << "Backup Type: " << flagMenu.backup_type <<
"\n";
    } else {
        std::cout << "Invalid input. Please enter 'full' or
'differential'.\n";
    }
    ignore_cin();
}

void set_hash_algorithm(po::variables_map& vm) {
    std::string buffer = vm["hash-algorithm"].as<std::string>();
    if (buffer == "MD5" || buffer == "SHA256") {
        flagMenu.hash_algorithm = buffer;
        flagMenu.flag_hash_algorithm = 1;
        std::cout << "Hash Algorithm: " <<
flagMenu.hash_algorithm << "\n";
    } else {
        std::cout << "Invalid input. Please enter 'MD5' or
'SHA256'.\n";
    }
    ignore_cin();
}

void set_notification_channel(po::variables_map& vm) {
    std::string buffer = vm["notification-
channel"].as<std::string>();
    if (buffer == "no" || buffer == "system") {
        flagMenu.notification_channel = buffer;
        flagMenu.flag_notification_channel = 1;
        std::cout << "Notification Channel: " <<
flagMenu.notification_channel << "\n";
    } else {
        std::cout << "Invalid input. Please enter 'no' or
'system'.\n";
    }
    ignore_cin();
}

void show_help_menu() {
    system("clear");
    show_help();
    ignore_cin();
}



void file_info() {
    read_config_file();
    // Вывод информации о файлах
std::cout << "\n************************************\n";
std::cout << "* File information:                *\n";
std::cout << "* Path: " << bufferMenu.path << std::endl;

// Вывод состояний из структуры flagMenu
std::cout << "* Protection: " << (bufferMenu.protection ?
"Enabled" : "Disabled") << std::endl;
std::cout << "* Interval: " << bufferMenu.interval << std::endl;
std::cout << "* Backup Type: " << bufferMenu.backup_type <<
std::endl;
std::cout << "* Hash Algorithm: " << bufferMenu.hash_algorithm <<
std::endl;
std::cout << "* Notification Channel: " <<
bufferMenu.notification_channel << std::endl;
std::cout << "************************************\n";
```

```
    ignore_cin();
}

void show_help() {

    std::cout <<
"****************************************************************
***********************\n";
    std::cout << "*                                 Help
Information:                                    *\n";
    std::cout << "* Enable protection: This will enable the file
protection.                              *\n";
    std::cout << "* Disable protection: This will disable the
file protection.                            *\n";
    std::cout << "* Get information about protected files: This
will display information about the        *\n";
    std::cout << "*
protected files.                             *\n";
    std::cout << "* Force file restoration: This will force the
restoration of the protected files.       *\n";
    std::cout << "* Configure protection:
*\n";
    std::cout << "** Specify the file path: Specify the path to
the file or directory to be protected.    *\n";
    std::cout << "** Specify the file check interval: Specify the
interval in minute (1-180) at which the *\n";
    std::cout << "**                                 file or
directory should be checked.              *\n";
    std::cout << "** Specify the backup type: The type of backup
to be performed (full/differential).     *\n";
    std::cout << "** Specify the hashing algorithm: Specify the
hashing algorithm to be used (MD5/SHA256).*\n";
    std::cout << "** Specify the notification channel: Specify
the channel through which notifications    *\n";
    std::cout << "**                                  should be
sent (no/system).                         *\n";
    std::cout <<
"****************************************************************
***********************\n";
    ignore_cin();
}


void ignore_cin(){
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
```

```
    getchar();
}
```

Файл back_sfc\src\configuration\chek_conf_file.h

```
#ifndef BACK_CHEK_CONF_FILE_H
#define BACK_CHEK_CONF_FILE_H

#include <sys/inotify.h>
#include <unistd.h>
#include <iostream>
#include <climits>

#include "config.h"

void watch_config_file(const std::string& config_path)

#endif //BACK_CHEK_CONF_FILE_H
```

Файл back_sfc\src\configuration\config.h

```
#ifndef BACK_CONFIG_H
#define BACK_CONFIG_H


#include <string>
#include <fstream>
#include <json/json.h>
#include <ncurses.h>
#include <fstream>
#include <iostream>
#include <random>

#include "../other/logger.h"
#include "../other/notification.h"

struct flag_menu {
    bool protection; //+
    bool force_restore;
    std::string path; //
    int interval; //+
    std::string backup_type; //+
    std::string hash_algorithm; //+
    std::string notification_channel; //

    std::string password;
    std::string directory;
};
extern flag_menu flagMenu;

struct flag_menu_changes {
    bool protection_changed = false;
    bool force_restore_changed = false;
    bool path_changed = false;
    bool interval_changed = false;
    bool backup_type_changed = false;
    bool hash_algorithm_changed = false;
    bool notification_channel_changed = false;
};

bool parse_config();
void file_info();
void main_proc(flag_menu_changes flagMenuChanges);

std::string generate_random_string(size_t length);

#endif //BACK_CONFIG_H
```

Файл back_sfc\src\daemon\deamon_works.h

```
#ifndef BACK_DEAMON_WORKS_H
#define BACK_DEAMON_WORKS_H

#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <csignal>
#include <cstdlib>
#include <cstdlib>
#include <cstring>
#include <cstdio>
#include <string>
#include <stdexcept>
#include <sys/prctl.h>


void daemonize();
```

```
#endif //BACK_DEAMON_WORKS_H
```

Файл back_sfc\src\other\logger.h

```
#ifndef BACK_LOGGER_H
#define BACK_LOGGER_H


#include <boost/log/trivial.hpp>
#include <boost/log/core.hpp>
#include <boost/log/trivial.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/sinks/text_file_backend.hpp>
#include <boost/log/utility/setup/file.hpp>
#include <boost/log/utility/setup/common_attributes.hpp>
#include <boost/log/attributes/current_thread_id.hpp>
#include <boost/log/attributes/named_scope.hpp>
#include <boost/date_time/posix_time/posix_time_io.hpp>

namespace logging = boost::log;
namespace sinks = boost::log::sinks;
namespace keywords = boost::log::keywords;
namespace expr = boost::log::expressions;

void setup_logger();

#define LOG BOOST_LOG_TRIVIAL

#endif //BACK_LOGGER_H
```

Файл back_sfc\src\other\notification.h

```
#ifndef BACK_NOTIFICATION_H
#define BACK_NOTIFICATION_H

// не должен включать никакие файлы

#include <iostream>
#include <string>
#include <cstdlib>

void send_notification(std::string &notification_channel, const
std::string &message);
```

```
void send_critical_urgency_notification(std::string
&notification_channel, const std::string &message);

#endif //BACK_NOTIFICATION_H
```

Файл back_sfc\src\proc\main_proc.h

```
#ifndef BACK_MAIN_PROC_H
#define BACK_MAIN_PROC_H

#include "../configuration/config.h"

#include <filesystem>
#include <cstdlib>
#include <iostream>
#include <sys/stat.h>
#include <atomic>

void to_decrypt_file(const std::string& filename, const
std::string& password);
void to_encrypt_file(const std::string& filename, const
std::string& password);

void begin_backup();
void restore_backup();
void full_backup(const std::string& source_directory, const
std::string& backup_directory);
void full_restore_backup(const std::string& source_directory,
const std::string& backup_directory);
void differential_restore_backup(const std::string&
source_directory, const std::string& backup_directory);

void force_restore_if_needed();
void force_full_restore(const std::string& source_directory,
const std::string& backup_directory);
void force_differential_restore(const std::string&
source_directory, const std::string& backup_directory);



extern std::unordered_map<std::string, std::pair<std::string,
std::string>> file_hashes;
extern std::unordered_map<std::string, std::pair<std::string,
std::string>> new_file_hashes;
```

```
void begin_hash();
void calculate_and_store_hashes(const std::string &directory,
std::unordered_map<std::string, std::pair<std::string,
std::string>> &map_hashes);
std::pair<std::string, std::string> calculate_hashes(const
std::string& path);
std::string calculate_md5_hash(const std::string& path);
std::string calculate_sha256_hash(const std::string& path);


extern std::atomic<bool> check_interval;

void check_file(const std::string &directory);
void stop_check_file();




#endif //BACK_MAIN_PROC_H
```

Файл front\front_sfc\config.h

```
#ifndef FRONT_SFC_CONFIG_H
#define FRONT_SFC_CONFIG_H

#include <json/json.h>
#include <fstream>
#include <sys/stat.h>
#include <sys/types.h>

#include "menu.h"

void update_config_file();
bool file_exists(const std::string& name);

#endif //FRONT_SFC_CONFIG_H
```

Файл front\front_sfc\menu.h

```
#ifndef FRONT_SFC_MENU_H
#define FRONT_SFC_MENU_H

#include <cstdlib>
#include <libconfig.h>
```

```
#include "config.h"
#include "menu_func.h"




int menu_arg_main(int argc, char* argv[]);
void configure_protection();
void process_command_line_options(int argc, char* argv[]);
void process_menu_options();




#endif //FRONT_SFC_MENU_H
```

Файл front\front_sfc\menu_func.h

```
#ifndef FRONT_SFC_MENU_FUNC_H
#define FRONT_SFC_MENU_FUNC_H

#include <boost/program_options.hpp>
#include <iostream>
#include <filesystem>

struct flag_menu {
    bool protection;
    int flag_protection = 0;

    bool force_restore = false;
    int flag_force_restore = 0;

    std::string path;
    int flag_path = 0;

    int interval;
    int flag_interval = 0;

    std::string backup_type;
    int flag_backup_type = 0;

    std::string hash_algorithm;
    int flag_hash_algorithm = 0;

    std::string notification_channel;
    int flag_notification_channel = 0;
};
extern flag_menu flagMenu;

struct buffer_menu {
    bool protection;
    bool force_restore = false;
    std::string path;
    int interval;
    std::string backup_type;
    std::string hash_algorithm;
    std::string access_permissions;
    std::string notification_channel;
};
extern buffer_menu bufferMenu;

namespace po = boost::program_options;

void enable_protection();
void disable_protection();
void get_file_info();
void force_file_restore();
void show_help_info(po::options_description& desc);
void exit_program();
void set_file_path(po::variables_map& vm);
void set_file_check_interval(po::variables_map& vm);
void set_backup_type(po::variables_map& vm);
void set_hash_algorithm(po::variables_map& vm);
void set_notification_channel(po::variables_map& vm);
void file_info();
void show_help();
void show_help_menu();

void read_config_file();
void ignore_cin();

#endif //FRONT_SFC_MENU_FUNC_H
```

Файл back_sfc\makefile

```
cmake_minimum_required(VERSION 3.22)
project(back_sfc)

set(CMAKE_CXX_STANDARD 17)
```

```
# Find packages jsoncpp, Boost, and OpenSSL
find_package(jsoncpp REQUIRED)
find_package(Boost REQUIRED COMPONENTS log_setup log filesystem
regex thread date_time system chrono atomic filesystem)
find_package(OpenSSL REQUIRED)

add_executable(back_sfc src/main.cpp
        src/configuration/config.cpp
        src/configuration/config.h
        src/configuration/chek_conf_file.cpp
        src/configuration/chek_conf_file.h
        src/daemon/deamon_works.cpp
        src/daemon/deamon_works.h
        src/other/logger.h
        src/other/logger.cpp
        src/proc/main_proc.cpp
        src/proc/main_proc.h
        src/proc/backup.cpp
        src/proc/hash.cpp
        src/other/notification.cpp
        src/other/notification.h
        src/proc/hash.cpp
        src/proc/encrypted.cpp)

# Link libraries jsoncpp, Boost, and OpenSSL with your executable
target_link_libraries(back_sfc jsoncpp_lib Boost::log_setup
Boost::log Boost::filesystem Boost::regex Boost::thread
Boost::date_time Boost::system Boost::chrono Boost::atomic
OpenSSL::Crypto OpenSSL::SSL)
```

Файл front_sfc\makefile

```
cmake_minimum_required(VERSION 3.22)
project(front_sfc)

set(CMAKE_CXX_STANDARD 17)

# Find Boost and the program_options component
find_package(Boost REQUIRED COMPONENTS program_options)

# Use pkg-config to find Jsoncpp
find_package(PkgConfig REQUIRED)
pkg_check_modules(JSONCPP jsoncpp)

# Include Boost and Jsoncpp headers
```

```
include_directories(${Boost_INCLUDE_DIRS}
${JSONCPP_INCLUDE_DIRS})

# Add your executable target
add_executable(front_sfc main.cpp menu.cpp menu.h
        config.cpp
        config.h
        menu_func.cpp
        menu_func.h)

# Link your executable target with Boost.Program_options and
Jsoncpp
target_link_libraries(front_sfc ${Boost_LIBRARIES}
${JSONCPP_LIBRARIES})
```