//main.cpp

```
#include "Header.h"

int main(int argc, char*argv[]) {
    BootSector bootSector{};
    ULONG sectorSize;
    ULONG clusterSize;
    ULONG recordSize;
    LONGLONG totalClusters;
    LONGLONG recordsNumber;
    HANDLE handle = nullptr;
    std::string volume = R"(\\.\_:)";
    std::vector<Run> MFTRunList, BitMapRunList;
    LONGLONG MFTSize = 0LL, BitMapSize = 0LL;
    std::atomic<LONGLONG> progress(0);
    std::atomic<bool> analysisCompleted(false);
    try {
        if (argc == 3 && strlen(argv[1]) == 1 && strlen(argv[2])
== 2 && argv[2][0] == '-') {
            volume[4] = argv[1][0];
            handle = CreateFile(volume.c_str(), GENERIC_READ,
FILE_SHARE_READ | FILE_SHARE_WRITE,
                                nullptr, OPEN_EXISTING, 0,
nullptr);
            if (handle == INVALID_HANDLE_VALUE)
                throw std::runtime_error("Failed to open a
drive.");
            switch (argv[2][1]) {
                case 'i': {
                    readBootSector(handle, bootSector,
sectorSize, clusterSize, recordSize, totalClusters);
                    MFTRunList.assign(1,
Run(bootSector.MFTCluster, 24 * recordSize / clusterSize));
                    readMFTMain(handle, MFTRunList, MFTSize,
BitMapRunList, BitMapSize, sectorSize, clusterSize,
                                recordSize);
                    recordsNumber = MFTSize / recordSize;

                    basicInformation(bootSector, sectorSize,
clusterSize, recordSize,
                                     totalClusters, MFTSize,
recordsNumber, volume);
                    system("pause");
                    break;
                }
                case 'f': {
                    readBootSector(handle, bootSector,
sectorSize, clusterSize, recordSize, totalClusters);
```

```
                    MFTRunList.assign(1,
Run(bootSector.MFTCluster, 24 * recordSize / clusterSize));
                    readMFTMain(handle, MFTRunList, MFTSize,
BitMapRunList, BitMapSize, sectorSize, clusterSize,
                                recordSize);
                    recordsNumber = MFTSize / recordSize;

                    mainAnalysis(handle, MFTRunList,
recordsNumber, BitMapRunList, BitMapSize, sectorSize,
                                 clusterSize, recordSize,
volume, progress, analysisCompleted);
                    system("pause");
                    break;
                }
                default:
                    throw std::runtime_error(
                            "Invalid parameters.\nFormat:
file.exe disk_letter(C,D,F) parameter(-i,-f)\n -i - general
information about a volume\n -f - checking integrity\n");
            }
        } else {
            std::cout << "Format: ntfs_checker
disk_letter(C,D,F) parameter(-i, -f)" << std::endl <<
                      "-i - general information about a volume"
<< std::endl << "-f - checking integrity" << std::endl;
            system("pause");
        }
    }
    catch (std::exception &obj) {
        printf("Error: %s\n", obj.what());
        system("pause");
    }
    catch (...) {
        printf("Error: Unknown\n");
        system("pause");
    }

    if (handle != nullptr)
        CloseHandle(handle);

    return 0;
}
```

//header.h

```
#pragma once

#include <Windows.h>
#include <iostream>
```

```
#include <cstdio>
#include <winioctl.h>
#include <shellapi.h>
#include <vector>
#include <fstream>
#include <ctime>
#include <bitset>
#include <thread>
#include <atomic>
#include <iomanip>

#define NTFS_IDENTIFIER 0x202020205346544E
static_assert(true);

#pragma pack(push, 1)
struct BootSector
{
    UCHAR       jump[3];
    ULARGE_INTEGER       oemID;
    WORD        bytesPerSector;
    UCHAR       sectorsPerCluster;
    USHORT      zero0;
    UCHAR       zero1;
    USHORT      zero2;
    USHORT      zero3;
    UCHAR       mediaDescriptor; //0xf8 = hard disk
    USHORT      zero4;
    WORD        sectorsPerTrack; //unused
    WORD        heads; //unused
    DWORD       hiddenSectors;  //unused
    ULONG       zero5;
    UCHAR       unused0[4];
    LONGLONG    totalSectors;
    LONGLONG    MFTCluster;
    LONGLONG    MFTMirrCluster;
    CHAR        clustersPerRecord;
    UCHAR       unused1[3];
    CHAR        clustersPerIndex;
    UCHAR       unused2[3];
    LONGLONG    serialNumber;
    DWORD       checkSum;
    UCHAR       bootCode[0x1aa];
    UCHAR       endMarker[2];
};

struct RecordHeader
{
    UCHAR       signature[4];
```

```
    USHORT      updateSeqOffset;
    USHORT      updateSeqNumber;
    LONGLONG    lsn;
    USHORT      sequenceNumber;
    USHORT      hardLinkCount;
    USHORT      attributeOffset;
    USHORT      flag;
    ULONG       usedSize;
    ULONG       allocatedSize;
    LONGLONG    baseRecord;
    USHORT      nextAttributeID;
    USHORT      unused;
    ULONG       MFTRecord;
};

struct AttributeHeaderR
{
    ULONG       typeID;
    USHORT      length;
    USHORT      reserved;
    UCHAR       formCode;
    UCHAR       nameLength;
    USHORT      nameOffset;
    USHORT      flag;
    USHORT      attributeID;
    ULONG       contentLength;
    USHORT      contentOffset;
    WORD        unused;
};

struct AttributeHeaderNR
{
    ULONG       typeID;
    USHORT      length;
    USHORT      reserved;
    UCHAR       formCode;
    UCHAR       nameLength;
    USHORT      nameOffset;
    USHORT      flag;
    USHORT      attributeID;
    LONGLONG    startVCN;
    LONGLONG    endVCN;
    USHORT      runListOffset;
    USHORT      compressionUnit;
    UCHAR       unused[4];
    LONGLONG    allocatedContentSize;
    LONGLONG    contentSize;
    LONGLONG    initializedContentSize;
```

```
};

struct FileName
{
    LONGLONG    parentDirectory;
    LONGLONG    dateCreated;
    LONGLONG    dateModified;
    LONGLONG    dateMFTModified;
    LONGLONG    dateAccessed;
    LONGLONG    allocatedSize;
    LONGLONG    usedSize;
    ULONG       flag;
    ULONG       reparseValue;
    UCHAR       nameLength;
    UCHAR       nameType;
    UCHAR       name[];
};

#pragma pack(pop)

struct Run
{
    LONGLONG    offset;
    LONGLONG    length;
    Run() : offset(0LL), length(0LL) {}
    Run(LONGLONG offset, LONGLONG length) : offset(offset),
length(length) {}
};

typedef enum {
    MFT_RECORD_NOT_USED      = 0,
    MFT_RECORD_IN_USE         = 1,
    MFT_RECORD_IS_DIRECTORY   = 2,
} MFT_RECORD_FLAGS;

typedef enum
{
    AT_FILE_NAME                    = 0x30,
    AT_DATA                              = 0x80,
    AT_BITMAP                       = 0xb0,
    AT_END                               = 0xffffffff
} ATTR_TYPES;

//runList.cpp
void readRunList(HANDLE handle, LONGLONG recordIndex, ULONG
typeID, const std::vector<Run>& MFTRunList, ULONG sectorSize,
                 ULONG clusterSize, ULONG recordSize,
std::vector<Run>* runList, LONGLONG* contentSize);
```

```
std::vector<Run> parseRunList(LPBYTE runList);

//processRecord.cpp
void readRecord(HANDLE h, LONGLONG recordIndex, const
std::vector<Run>& MFTRunList,ULONG recordSize, ULONG
clusterSize,
                ULONG sectorSize, LPBYTE buffer);
void seek(HANDLE h, ULONGLONG position);
LPBYTE findAttribute(RecordHeader* record, ULONG recordSize,
ULONG typeID);


//analysis.cpp
void readBootSector(HANDLE handle, BootSector &bootSector, ULONG
&sectorSize,ULONG &clusterSize,
                    ULONG &recordSize, LONGLONG &totalClusters);
void readMFTMain(HANDLE handle, std::vector<Run>& MFTRunList,
LONGLONG &MFTSize,
                 std::vector<Run>& BitMapRunList, LONGLONG
&BitMapSize, ULONG sectorSize, ULONG clusterSize, ULONG
recordSize);
void basicInformation(BootSector bootSector, ULONG
sectorSize,ULONG clusterSize, ULONG recordSize,
                      LONGLONG totalClusters, LONGLONG MFTSize,
LONGLONG recordsNumber, const std::string& volume);
void mainAnalysis(HANDLE handle, const std::vector<Run>&
MFTRunList, LONGLONG recordsNumber, const std::vector<Run>&
BitMapRunList,
                  LONGLONG BitMapSize, ULONG sectorSize, ULONG
clusterSize, ULONG recordSize, const std::string& volume,
                  std::atomic<LONGLONG>& progress,
std::atomic<bool>& analysisCompleted);
void processRecord(HANDLE handle, const std::vector<Run>&
MFTRunList, LONGLONG recordIndex, ULONG recordSize, ULONG
clusterSize, ULONG sectorSize,
                   const std::vector<Run>& BitMapRunList,
LONGLONG BitMapSize, std::fstream& log, std::ofstream& analysis,
bool& errorOccur);
void readBitmap(HANDLE handle, const std::vector<Run>&
BitMapRunList, ULONG clusterSize, std::vector<UCHAR>& blocks);
void writeToLog(std::fstream& log, const std::string& volume);
void writeToAnalysis(std::ofstream& analysis, const std::string&
volume);
void progressBar(std::atomic<LONGLONG>& progress, LONGLONG
recordsNumber, std::atomic<bool>& analysisCompleted);
```

```
//runList.cpp
#include "header.h"

void readRunList(HANDLE handle, LONGLONG recordIndex, ULONG
typeID, const std::vector<Run>& MFTRunList, ULONG sectorSize,
                 ULONG clusterSize, ULONG recordSize,
std::vector<Run>* runList, LONGLONG* contentSize)
{
    std::vector<UCHAR> record(recordSize);
    readRecord(handle, recordIndex, MFTRunList, recordSize,
clusterSize, sectorSize, &record[0]);
    auto* recordHeader = (RecordHeader*)&record[0];
    auto* headerNR =
(AttributeHeaderNR*)findAttribute(recordHeader, recordSize,
typeID);

    if (headerNR == nullptr)
        return ;
    if (headerNR->formCode == 1)
    {
        std::vector<Run> runListTmp =
parseRunList(LPBYTE(headerNR) + headerNR->runListOffset);

        runList->resize(runListTmp.size());
        for (size_t i = 0; i < runListTmp.size(); i++)
            (*runList)[i] = runListTmp[i];

        if (contentSize != nullptr)
            *contentSize = headerNR->contentSize;
    }
}

std::vector<Run> parseRunList(LPBYTE runList)
{
    std::vector<Run> result;

    LONGLONG offset = 0LL;

    LPBYTE p = runList;
    while (*p != 0x00)
    {
        int lenLength = *p & 0xf;
        int lenOffset = *p >> 4;
        p++;

        ULONGLONG length = 0;
        for (int i = 0; i < lenLength; i++)
            length |= *p++ << (i * 8);
```

```
        LONGLONG offsetDiff = 0;
        for (int i = 0; i < lenOffset; i++)
            offsetDiff |= *p++ << (i * 8);
        if (offsetDiff >= (1LL << ((lenOffset * 8) - 1)))
            offsetDiff -= 1LL << (lenOffset * 8);

        offset += offsetDiff;

        result.emplace_back(offset, length);
    }

    return result;
}
```

//processRecord.cpp

```
#include "header.h"

void readRecord(HANDLE h, LONGLONG recordIndex, const
std::vector<Run>& MFTRunList,
                ULONG recordSize, ULONG clusterSize, ULONG
sectorSize, LPBYTE buffer)
{
    LONGLONG sectorOffset = recordIndex * recordSize /
sectorSize;
    ULONG sectorNumber = recordSize / sectorSize;

    for (ULONG sector = 0; sector < sectorNumber; sector++)
    {
        LONGLONG cluster = (sectorOffset + sector) /
(clusterSize / sectorSize);
        LONGLONG vcn = 0LL; //виртуальный номер кластера внутри
MFT
        LONGLONG offset = -1LL;

        //пробегаемся по списку отрезков MFT
        for (const Run& run : MFTRunList)
        {
            if (cluster < vcn + run.length)
            {
                //смещение относительно начала MFT в байтах
                offset = (run.offset + cluster - vcn) *
clusterSize + (sectorOffset + sector) * sectorSize %
clusterSize;
                break;
            }
            vcn += run.length;
        }
```

```
        if (offset == -1LL)
            throw std::runtime_error("Failed to read file
record");

        seek(h, offset);
        ULONG read;
        if (!ReadFile(h, buffer + sector * sectorSize,
sectorSize,
                      &read, nullptr) || read != sectorSize)
            throw std::runtime_error("Failed to read file
record");
    }
}

void seek(HANDLE h, ULONGLONG position)
{
    LARGE_INTEGER pos;
    pos.QuadPart = (LONGLONG)position;
    LARGE_INTEGER result;
    if (!SetFilePointerEx(h, pos, &result, SEEK_SET) ||
        pos.QuadPart != result.QuadPart)
        throw std::runtime_error("Failed to seek");
}

LPBYTE findAttribute(RecordHeader* record, ULONG recordSize,
ULONG typeID)
{
    LPBYTE p = LPBYTE(record) + record->attributeOffset;
    while (true)
    {
        if (p + sizeof(AttributeHeaderR) > LPBYTE(record) +
recordSize)
            break;

        auto* header = (AttributeHeaderR*)p;
        if (header->typeID == AT_END)
            break;

        if (header->typeID == typeID && p + header->length <=
LPBYTE(record) + recordSize)
            return p;

        p += header->length;
    }
    return nullptr;
}
```

//analysis.cpp

```
#include "header.h"

void readBootSector(HANDLE handle, BootSector &bootSector, ULONG
&sectorSize,ULONG &clusterSize,
                    ULONG &recordSize, LONGLONG &totalClusters){

    ULONG read;
    if (!ReadFile(handle, &bootSector, sizeof(BootSector),
&read,
                  nullptr) || read != sizeof(BootSector)) {
        throw std::runtime_error("Failed to read boot sector");
    }

    if (bootSector.oemID.QuadPart != NTFS_IDENTIFIER) {
        printf("\nVolume is not NTFS. OEM ID: %llu\n",
bootSector.oemID.QuadPart);
        throw std::runtime_error("Volume is not NTFS");
    }

    printf("Volume is NTFS. OEM ID: \"%llu\"\n\n",
bootSector.oemID.QuadPart);

    if(bootSector.zero0!=0 || bootSector.zero1 !=0 ||
bootSector.zero2 !=0 || bootSector.zero3 !=0 ||
    bootSector.zero4 !=0 || bootSector.zero5 !=0)
        throw std::runtime_error("Particular bits are not
0.\n");

    sectorSize = bootSector.bytesPerSector;
    clusterSize = sectorSize * bootSector.sectorsPerCluster;
    recordSize = bootSector.clustersPerRecord >= 0 ?
bootSector.clustersPerRecord * clusterSize :
                1 << -bootSector.clustersPerRecord;
    totalClusters = bootSector.totalSectors /
bootSector.sectorsPerCluster;
}

void readMFTMain(HANDLE handle, std::vector<Run>& MFTRunList,
LONGLONG &MFTSize,
                 std::vector<Run>& BitMapRunList, LONGLONG
&BitMapSize, ULONG sectorSize, ULONG clusterSize, ULONG
recordSize){

    readRunList(handle, 0, AT_DATA, MFTRunList, sectorSize,
clusterSize, recordSize,
                &MFTRunList, &MFTSize);
```

```
    readRunList(handle, 0, AT_BITMAP, MFTRunList,sectorSize,
clusterSize, recordSize,
                &BitMapRunList, &BitMapSize);
}


void readBitmap(HANDLE handle, const std::vector<Run>&
BitMapRunList, ULONG clusterSize, std::vector<UCHAR>& blocks) {
    ULONG read;
    LONGLONG sumClusters = 0;
    for (const Run& run : BitMapRunList) {
        LONGLONG offset = run.offset * clusterSize;
        seek(handle, offset);
        if (!ReadFile(handle, &blocks[0] + sumClusters *
clusterSize,
                      run.length * clusterSize, &read,
                      nullptr) || read != run.length *
clusterSize)
            throw std::runtime_error("Failed to read BitMap");
        sumClusters += run.length;
    }
}

// Запись в лог
void writeToLog(std::fstream& log, const std::string& volume) {

    int choice;
    bool flag = true;
    std::cout << "Before running, do you want to clear Log.txt
file?\n1 - Yes\n2 - No" << std::endl;
    while (flag) {
        std::cin >> choice;
        if (choice == 1 ) {
            log.open("Log.txt", std::ios::out);
            flag = false;
        } else if (choice == 2) {
            log.open("Log.txt", std::ios::app);
            flag = false;
        } else {
            std::cout << "Incorrect input. Try again." <<
std::endl;
        }
    }

    std::time_t currentTime = std::time(nullptr);
    std::string dateTimeString = std::ctime(&currentTime);
    log << "Integrity checking of volume: " << volume[4] <<
"\nDate/time: " << dateTimeString << std::endl;
```

```
    log << "-----------------------------\n";
}

// Запись в файл анализа
void writeToAnalysis(std::ofstream& analysis, const std::string&
volume) {
    std::time_t currentTime = std::time(nullptr);
    std::string dateTimeString = std::ctime(&currentTime);
    analysis << "Integrity checking of volume: " << volume[4] <<
"://\nDate/time: " << dateTimeString;
    analysis << "-----------------------------\n";
    analysis << "File List:\n";
}

// Обработка каждой записи
void processRecord(HANDLE handle, const std::vector<Run>&
MFTRunList, LONGLONG recordIndex, ULONG recordSize, ULONG
clusterSize, ULONG sectorSize,
                   const std::vector<Run>& BitMapRunList,
LONGLONG BitMapSize, std::fstream& log, std::ofstream& analysis,
bool& errorOccur) {
    std::vector<UCHAR> record(recordSize);
    analysis << recordIndex;
    readRecord(handle, recordIndex, MFTRunList, recordSize,
clusterSize, sectorSize, &record[0]);
    auto* recordHeader = (RecordHeader*)&record[0];
    if (memcmp(recordHeader->signature, "FILE", 4) != 0) {
        analysis << "   -\n";
        return;
    }

    if (recordHeader->baseRecord != 0LL) {
        analysis << "   extension for " << recordHeader-
>baseRecord << std::endl;
        return;
    }

    auto* nameAttr =
(AttributeHeaderR*)findAttribute(recordHeader, recordSize,
AT_FILE_NAME);

    if (nameAttr == nullptr) {
        analysis << "   Failed to find $File_Name attribute\n";
        return;
    }

    auto* nameContent = (FileName*)((LPBYTE)nameAttr + nameAttr-
>contentOffset);
```

```
    switch (recordHeader->flag) {
        case MFT_RECORD_NOT_USED: {
            log << "File: ";
            for (int i = 0; i < nameContent->nameLength; i++) {
                log << nameContent->name + i * 2;
            }
            log << ". Error: record is not used.\n";
            analysis << "   file ";
            errorOccur = true;
            break;
        }
        case MFT_RECORD_NOT_USED | MFT_RECORD_IS_DIRECTORY: {
            log << "Directory: ";
            for (int i = 0; i < nameContent->nameLength; i++) {
                log << nameContent->name + i * 2;
            }
            log << ". Error: record is not used.\n";
            analysis << "   dir ";
            errorOccur = true;
            break;
        }
        case MFT_RECORD_IN_USE: analysis << "   file ";  break;
        case MFT_RECORD_IN_USE | MFT_RECORD_IS_DIRECTORY:
analysis << "   dir ";  break;
        default: analysis << "        "; break;
    }

    for (int i = 0; i < nameContent->nameLength; i++) {
        analysis << nameContent->name + i * 2;
    }

    std::vector<UCHAR> blocks((BitMapSize / clusterSize + 1) *
clusterSize);
    readBitmap(handle, BitMapRunList, clusterSize, blocks);

    LONGLONG cluster = recordHeader->MFTRecord * recordSize /
clusterSize;
    std::bitset<8> bits = blocks[cluster / 9];
    bool isClusterUsed = bits.test(cluster - 9 * cluster / 9);
    if (!isClusterUsed) {
        analysis << ", ERROR: BitMap error!";
        log << "File: ";
        for (int i = 0; i < nameContent->nameLength; i++) {
            log << nameContent->name + i * 2;
        }
        log << ". Error: Corresponding cluster is not
allocated.\n";
```

```
        errorOccur = true;
    }
    analysis << "\n";
}

void mainAnalysis(HANDLE handle, const std::vector<Run>&
MFTRunList, LONGLONG recordsNumber, const std::vector<Run>&
BitMapRunList,
                  LONGLONG BitMapSize, ULONG sectorSize, ULONG
clusterSize, ULONG recordSize, const std::string& volume,
                  std::atomic<LONGLONG>& progress,
std::atomic<bool>& analysisCompleted) {

    bool  errorOccur = false;
    std::fstream log;
    std::ofstream analysis("Analysis.txt");

    writeToLog(log, volume);
    writeToAnalysis(analysis, volume);

    std::cout<<"\nPlease waiting... Analysis is
started.\n"<<std::endl;
    std::thread progressBarThread(progressBar,
std::ref(progress), recordsNumber, std::ref(analysisCompleted));
    for (LONGLONG recordIndex = 0; recordIndex < recordsNumber;
recordIndex++) {
        progress = recordIndex + 1;
        processRecord(handle, MFTRunList, recordIndex,
recordSize, clusterSize, sectorSize, BitMapRunList, BitMapSize,
log, analysis,  errorOccur);
    }

    log << "------------------------------\n\n\n\n";
    analysis << "------------------------------\n\n\n\n";
    log.close();
    analysis.close();
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    analysisCompleted = true;

    progressBarThread.join();
    std::cout << "Analysis is finished.\n" << std::endl;
    if (errorOccur) {
        std::cout << "Some errors were found. Please check
Log.txt for more information" << std::endl;
    } else {
        std::cout << "There are no errors. Everything is OK." <<
std::endl;
    }
```

```
    std::cout << "Analysis process is located in Analysis.txt in
an utility directory.\n" << std::endl;
}

void progressBar(std::atomic<LONGLONG>& progress, LONGLONG
recordsNumber, std::atomic<bool>& analysisCompleted) {
    while (!analysisCompleted) {
        int barWidth = 70;
        float progressPercent = (float)progress /
(float)recordsNumber;
        std::cout << "[";
        int pos = barWidth * progressPercent;
        for (int i = 0; i < barWidth; ++i) {
            if (i < pos) std::cout << "=";
            else if (i == pos) std::cout << ">";
            else std::cout << " ";
        }
        int integerPart = (int)(progressPercent * 100.0);
        double fractionalPart = (progressPercent*100.0 -
(float)integerPart)*100.0;
        std::cout << "] " << integerPart <<"." <<
(int)fractionalPart << " %\r";
        std::cout.flush();

std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    std::cout << std::endl << std::endl;
}

void basicInformation(BootSector bootSector, ULONG
sectorSize,ULONG clusterSize, ULONG recordSize,
                      LONGLONG totalClusters, LONGLONG MFTSize,
LONGLONG recordsNumber, const std::string& volume) {
    std::fstream log;
    writeToLog(log, volume);

    printf("BASIC INFORMATION ABOUT A VOLUME\n\n");
    printf("Total clusters: %lld, %lld (Bytes)\n",
totalClusters, totalClusters * clusterSize);
    printf("Cluster size: %lu (Bytes)\n", clusterSize);
    printf("Total sectors: %llu, %llu (Bytes)\n",
bootSector.totalSectors,
           bootSector.totalSectors * sectorSize);
    printf("Sector size: %lu (Bytes)\n", sectorSize);
    printf("Sectors per cluster: %u\n",
bootSector.sectorsPerCluster);
    printf("Start MFT cluster: %llu\n", bootSector.MFTCluster);
```

```
    printf("MFT size: %llu (Bytes)\n", MFTSize);
    printf("Records number: %llu\n", recordsNumber);
    printf("Record size: %lu (Bytes)\n", recordSize);

    log << "BASIC INFORMATION ABOUT A VOLUME\n";
    log << "-------------------------------\n\n\n\n";
    log.close();
}
```