A blue and white logo

Description automatically generated

hỆ ĐIỀU HÀNH

PROJECT 1: Hệ thống quản lý tập tin trên Windows

**Thành viên:**

* Trần Đăng Tuấn – 22127438
* Nguyễn Phước Sang – 22127362
* Cao Minh Quang – 22127353
* Ngô Tấn Tài – 22127371

Mục lục

[**I.** **Các bước thực hiện đồ án** 2](#_Toc161103503)

[**II.** **Lấy thông tin từ RDET** 6](#_Toc161103504)

[**1.** **Giới thiệu sơ lược về cấu trúc của RDET** 6](#_Toc161103505)

[**2.** **Hàm lấy thông tin từ RDET và SDET** 8](#_Toc161103506)

[**2.1.** **getNameMainEntry - Hàm lấy tên file/folder** 9](#_Toc161103507)

[**2.2.** **Hàm lấy thông tin trạng thái** 11](#_Toc161103508)

[**2.3.** **getSize - Hàm lấy kích thước** 12](#_Toc161103509)

[**2.4.** **getNextSector - Hàm lấy địa chỉ dữ liệu** 12](#_Toc161103510)

[**2.5.** **ReadSector – Hàm đọc bytes biểu diễn thông tin Data** 13](#_Toc161103511)

[**2.6.** **readData – Hàm đọc dữ liệu** 14](#_Toc161103512)

[**III.** **Chạy demo sản phẩm FAT32 & RDET** 15](#_Toc161103513)

[**IV.** **Tổng quan về các hàm sử dụng cho việc đọc NTFS** 19](#_Toc161103514)

[**1.** **bool readSector(LPCWSTR drive, int readPoint, BYTE \*&sector)** 19](#_Toc161103515)

[**2.** **void readSect2(LPCWSTR disk, BYTE \*&DATA, unsigned int \_nsect)** 21](#_Toc161103516)

[**3.** **int64\_t getBytes(BYTE \*sector, int offset, int number)** 23](#_Toc161103517)

[**4.** **string numToString(BYTE \*DATA, int offset, int number)** 24](#_Toc161103518)

[**5.** **string decimalToBinary(int number)** 25](#_Toc161103519)

[**6.** **printTab(int tab)** 25](#_Toc161103520)

[**7.** **string getNameFile(int id)** 26](#_Toc161103521)

[**8.** **printSector(BYTE \*sector)** 26](#_Toc161103522)

[**9.** **printFolderTree(int id, int tab, int pos)** 30](#_Toc161103523)

[**10.** **readEntryInformation(BYTE \*Entry, int start)** 32](#_Toc161103524)

[**11.** **readEntryFileName(BYTE \*Entry, int start, int ID)** 34](#_Toc161103525)

[**12.** **readEntryData(BYTE \*Entry, int start)** 37](#_Toc161103526)

[**13.** **readBPB(BYTE \*sector, LPCWSTR disk)** 39](#_Toc161103527)

[**14.** **readMFT(unsigned int MFTStart, unsigned int sectors\_per\_cluster, LPCWSTR disk)** 40](#_Toc161103528)

[**15.** **folderTree(unsigned int len\_MFT, unsigned int MFTStart, LPCWSTR disk)** 43](#_Toc161103529)

[**16.** **Hàm main:** 46](#_Toc161103530)

[**V.** **Nguồn tham khảo:** 48](#_Toc161103531)

# **Các bước thực hiện đồ án**

**Bước 1:** Tìm hiểu về các thư viện, cũng như ngôn ngữ lập trình cần dùng

Thư viện Windows.h là một phần của Windows API, cung cấp các hàm và cấu trúc dữ liệu cần thiết để tương tác với hệ điều hành Windows. Trong ngữ cảnh đọc bit trong hệ thống FAT (File Allocation Table), Windows.h cung cấp các chức năng và cấu trúc dữ liệu để thực hiện các hoạt động như đọc và ghi dữ liệu trên ổ đĩa.

Để đọc bit trong hệ thống FAT, bạn có thể sử dụng các hàm và cấu trúc như **CreateFile**, **ReadFile**, **WriteFile** để mở, đọc và ghi vào các tệp tin hoặc thiết bị.

**Bước 2:** Xây dựng các **Class/Struct** phù hợp để sử dụng.

Trong đó, chúng em đã tạo ra các class như: **BootSector**, **Volume**, **Program** và **Item-File-Folder**, nhằm mục đích dễ quản lý các attribute và hàm cần thiết.

**BootSector** sẽ gồm các attributes trong **MBR (Master Boot Record)**, 512 Bytes đầu tiên trong hệ thống tập tin FAT. Vì vậy chúng em đã tạo đủ các biến tương đương với 512 Bytes. Tiếp đến là những biến có **DATA TYPE** là **INT**, để chuyển từ **BYTE (Unsigned char)**, thành số, để có thể dễ dàng tính toán và sử dụng sau này.

class BootSector

{

public:

    ~BootSector();

    void ReadBootSector(string path);

    void DisplayBootSector();

    std::vector<uint32\_t> ReadFAT();

    std::vector<uint8\_t> ReadRDET();

    int BytesPerSectorInt;

    int SectorsPerClusterInt;

    int SectorBeforeFatInt;

    int NumberOfFATsInt;

    int SectorsPerTrackInt;

    int NumberOfHeadsInt;

    int TotalSectorsInt;

    int RootClusterInt;

    int FSInfoInt;

    int BackupBootSectorInt;

    int SectorsPerFATInt;

    int PhysicalDriveNumberInt;

    HANDLE hDevice;

    DWORD dwBytesRead;

    BOOL bResult;

    BYTE bBootSector[512];

    BYTE jmpBoot[3];

    BYTE OEMName[8];

    BYTE BytesPerSector[2];

    BYTE SectorsPerCluster;

    BYTE SectorBeforeFat[2];

    BYTE NumberOfFATs;

    BYTE RootEntries[2];

    BYTE TotalSectors16[2];

    BYTE MediaDescriptor;

    BYTE SectorsPerFAT16[2];

    BYTE SectorsPerTrack[2];

    BYTE NumberOfHeads[2];

    BYTE Distance[4];

    BYTE TotalSectors32[4];

    BYTE SectorsPerFAT32[4];

    BYTE BootSig[2];

};

**Item-File-Folder** là 1 class dùng để lưu những thông tin của tập tin/thư mục, giúp chúng em có thể ouput ra cây thư mục theo thứ tự hợp lí.

class Item

{

protected:

    std::string m\_name;

    int size;

    int startSector;

    bool isSystem;

    bool isHidden;

public:

    Item(string name, uint32\_t \_size, uint32\_t \_startSector, bool \_isSystem, bool \_isHidden);

    std::string getName();

    virtual int getSize() = 0;

    virtual void print(int depth) = 0;

    virtual bool isFolder() = 0;

    virtual ~Item() = 0;

};

class File : public Item

{

private:

    string data;

public:

    File(string name, uint32\_t \_size, uint32\_t \_startSector, bool \_isSystem, bool \_isHidden, string \_data);

    int getSize();

    void print(int depth);

    bool isFolder();

};

class Folder : public Item

{

private:

    std::vector<Item \*> items;

public:

    Folder(string name, uint32\_t \_startSector, bool \_isSystem, bool \_isHidden);

    int getSize();

    void print(int depth);

    void addItem(Item \*item);

    Item \*removeByName(std::string name);

    Item \*findByName(std::string name);

    bool isFolder();

    ~Folder();

};

**Program** đơn giản chỉ là class chạy chương trình.

struct Program{

    Volume volume;

    void run();

};

**Volume** là class để lưu trữ các thông tin trong **MBR**, **Fat Table** và **RDET Data**.

struct Volume{

    BootSector bootSector;

    std::vector<uint32\_t> fatData;

    std::vector<uint8\_t> RDETData;

    vector<Item\*> TransRdet(vector<uint8\_t>& data, bool flag, uint32\_t startOffset);

    vector<uint8\_t> ReadSector(int sector , int numSector);

    void read\_fatData();

    void read\_RDETData();

};

**Bước 3:** Xây dựng những hàm cần thiết để chạy chương trình.

Chúng em đã tạo ra những hàm cơ bản cùng với sự kết hợp với những kiến thức đã học trong lớp (Đọc bảng Master Boot Record, Fat Table, RDET cũng như SDET) để thực hiện yêu cầu của đồ án như: **ReadBootSector**, **read**\_**fatData, read\_RDETData**, **TransRdet**,...

Với việc đọc **Master Boot Record**, ta sẽ đọc 512 Bytes đầu tiên của hệ thống, tiếp đến ta sẽ lưu vào 1 mảng các phần tử. Ta sẽ có những thông tin cần thiết như **số Bytes của Sector**, **số Sector trước bảng FAT**, **số lượng bảng FAT**,…

* SectorBeforeFatInt: số Sector trước bảng FAT
* BytesPerSectorInt: số Byte trong 1 Sector
* NumberOfFATsInt: số lượng bảng FAT
* SectorsPerFATInt: số Sector của bảng FAT
* RootClusterInt: chỉ số Cluster bắt đầu RDET

Dựa vào những thông tin đã có từ **Master Boot Record**, ta có thể tính ra được số **offset byte** bắt đầu của **bảng** **FAT** và **số lượng Bytes cần đọc** qua công thức:

fatOffset = SectorBeforeFatInt \* BytesPerSectorInt;

bytesToRead = NumberOfFATsInt \* SectorsPerFATInt \* BytesPerSectorInt;

Sau khi đọc xong bảng FAT, ta sẽ chuyển sang đọc bảng **RDET** nhờ vào những thông tin trong **Master Boot Record**. Ta cũng cần tính số **offset byte** bắt đầu của **RDET** qua công thức:

rdetOffset = (SectorBeforeFatInt + NumberOfFATsInt \* SectorsPerFATInt) \* BytesPerSectorInt;

Sau khi di chuyển tới **offset byte** bắt đầu của **RDET**, ta cần lấy thông **Cluster bắt đầu** của **RDET** từ **MBR**. Tại bảng RDET, cluster đầu tiên được tính là **2** thay vì là 0 hay 1. Và đa số, **Cluster bắt đầu** của **RDET** cũng là **2**. Nên ta không cần cộng thêm số byte cần thiết để tìm **offset byte** bắt đầu của **RDET**. Tuy nhiên, để đảm bảo sự đồng nhất, cũng như sự an toàn của chương trình, chúng ta cần thêm dòng lệnh **IF** nhằm mục đích cho các trường hợp mà **Cluster bắt đầu** của **RDET** là1 con số lớn hơn 2.

    if (RootClusterInt != 0)

    {

        uint64\_t rootClusterOffset = (RootClusterInt - 2) \* SectorsPerClusterInt \* BytesPerSectorInt;

        rdetOffset += rootClusterOffset;

    }

Sau khi đã có được số **offset byte** bắt đầu của **RDET**, ta cần tính toán số lượng Byte cần phải đọc. Trong hệ thống tệp **FAT32**, kích thước của **RDET** **(Root Directory Entry Table)** được động hóa để phù hợp với số lượng tệp tin và thư mục trong thư mục gốc. **RDET** chứa một bản ghi cho mỗi tệp tin và thư mục trong thư mục gốc. Và hàm **ReadFile** của thư viện Windows.h chỉ có thể đọc số Byte là bội số của 512. Nhóm chúng em sử dụng 1 mảng để đọc 512 bytes mỗi lần, sau khi đọc xong 512 bytes thì sẽ kiểm tra mỗi byte đầu tiên của mỗi 32 bytes, kiểm tra xem có phải **0x00** để tính số entry có trong **RDET** và dừng lại, sau đó tạo 1 vector **RDET** để lưu tất cả giá trị trong **RDET**

# **Lấy thông tin từ RDET**

## **Giới thiệu sơ lược về cấu trúc của RDET**

RDET (Root Dỉrectory Entry Table: Cây thư mục gốc), nằm trên vùng dữ liệu (đối với FAT32), là một kiểu dữ liệu dùng để thể hiện cây thư mục. RDET là một dãy bytes lớn được chia ra thành những mảng con 32 bytes gọi là entry, mỗi entry sẽ chứa những bytes được quy ước sẵn và biểu diễn được thông tin của dữ liệu trong cây thư mục.

Ngoài ra các cây thư mục con của thư mục gốc cũng được lưu trữ theo kiểu dữ liệu giống với RDET – hay còn được gọi là SDET (Sub Directory Entry Table). Quy tắc lưu trữ thông tin trên SDET được quy định giống với RDET, chỉ khác một chỗ vì SDET có 64 bytes đầu chứa cha của folder/file đó, sự khác biệt k nhiều nên ta sẽ chỉ khảo sát cách truy xuất thông tin từ RDET.

RDET là một cấu trúc dữ liệu được xây dựng theo cấu trúc của linked-list. Các phần tử của RDET (entry) biểu diễn một phần thông tin của dữ liệu kèm địa chỉ của entry kế tiếp lưu trữ thông tin để truy cập vào.

Entry được chia ra làm 2 loại, entry chính và entry phụ. Entry chính sẽ lưu địa chỉ của entry kế tiếp (nơi chứa dữ liệu nếu entry đang xét biểu diễn một file, hoặc SDET nếu entry đang xét biểu diễn một folder). Các entry phụ sẽ lưu các thông tin của tập tin đó như tên, kích thước, trạng thái, … Các entry sẽ được sấp theo thứ tự ngược, các entry phụ xa nhất đầu tiên, sau đó đến entry chính

A screenshot of a computer

Description automatically generated

A table with text and numbers

Description automatically generatedChúng ta có thể nhận biết entry phụ bằng cách đọc 1 byte ở vị trí 0B, nếu byte ở đó có giá trị 0x0F thì entry đó là entry phụ, ở chương trình này, chúng ta chỉ lấy thông tin về tên được lưu ở entry phụ.

Ngược lại, nếu ta lấy 1 bytes ở vị trí 0B và nó có giá trị khác 0x0F, vậy entry ta đang xét chính là entry chính và có quy tắc lưu trữ thông tin khác với entry phụ.

A table with text and numbers

Description automatically generated with medium confidence

## **Hàm lấy thông tin từ RDET và SDET**

*Vì RDET và SDET có cấu trúc tương tự nhau nên ta chỉ khảo sát cách truy xuất thông tin từ RDET.*

vector<Item\* > Volume::TransRdet(vector<uint8\_t>& data, bool flag, uint32\_t startSector, uint32\_t& startCluster)

Hàm TransRdet sẽ được xây dựng dựa trên yêu cầu có khả năng đệ quy vì tính chất của việc truy cập vào dữ liệu bên trong folder sẽ yêu cầu nhiều lần gọi hàm, vậy nên ta sẽ có một số tham số sau để phục vụ cho mục đích đệ quy một cách hợp lí và trả được cho ta những kết quả mong muốn:

* + ***data*:** một vecto gồm các giá trị được lưu ở dạng decimal biểu diễn các bytes ở trong RDET (hoặc SDET)
  + ***flag*:** một biến kiểu boolean để check xem ra đang đọc RDET hay SDET để có xử lí dịch bytes đúng đắn.
  + ***startSector*:** để lưu trữ sector bắt đầu của đoạn bytes mình đang xét và tìm ra sector chính xác của các phần tử khi thực hiện phép truy cập tuần tự lấy thông tin từ đối tượng đó. Giá trị mặc định được truyền vào lần đầu gọi hàm chính là sector của RDET.
  + ***startCluster*:** được dùng để kiểu tra phần tử của FAT xem đối tượng hiện tại có thông tin nằm trong bao nhiêu cluster. Giá trị mặc định được truyền vào lần đầu gọi hàm chính là số cluster đầu trống của bảng FAT.

Hàm trả về một *vector<Item\*>* là 1 class đã được chúng em định nghĩa sẵn để lưu trữ thông tin lấy ra được.

Trong mảng dữ liệu đó, sẽ có nhiều entry là entry trống (có bytes đầu tiên là 0x00) hoặc tập tin được lưu bởi entry này đã bị xóa (có bytes đầu tiên là 0xE5). Vì cách lưu trữ ngược đặc biệt của FAT, nhóm em chọn phương pháp lặp qua từng entry (gồm 32 bytes) trong data và tìm ra entry chính, sau đó truy ngược về entry phụ để lấy được những thông tin cần thiết.

int startIdx = 0;

    if (!flag)

        startIdx = 64;

    for (int i = startIdx; i < data.size(); i += 32){

        if (data[i] == 0xE5 || data[i + 11] == 0x0F)   // if this entry is deleted or sub entry

            continue;

        if (data[i] == 0x00) // go through all entry

            break;

### **getNameMainEntry - Hàm lấy tên file/folder**

Thông tin đầu tiên chúng ta cần lấy là tên của folder/file đó: dựa trên quy tắc lưu trữ tên của entry, ta có 2 trường hợp:

* Nếu folder/file ta đang xét chỉ có một entry chính và không có entry phụ:
  + Tên sẽ được lưu bởi 8 bytes, bắt đầu từ vị trí 00, và có phần tên mở rộng (extension) nếu có sẽ được lưu bởi 3 bytes ở vị trí 08. Ta chỉ cần tạo vòng lặp và lấy dữ liệu từ những vị trí này sau đó chuyển từ decimal sang char là chúng ta sẽ lấy được tên.

string getNameMainEntry(vector<uint8\_t>& data, uint32\_t idx, bool isFolder){

    string result;

    vector<uint8\_t> name;

    // get the values of bytes that represent name

    for (int i = 0; i < 8; ++i){

        if (data[idx + i] == 0x20)

            continue;

        name.push\_back(data[idx+i]);

    }

    // convert it to string

    string Name = "";

    for (int i  = 0; i < name.size(); ++i)

        Name += (char)name[i];

    // check its extension

    if (!isFolder){

        vector<uint8\_t> extension;

        for (int i = 8; i < 11; ++i){

            extension.push\_back(data[idx + i]);

        }

        string Extension = "";

        for (int i  = 0; i < extension.size(); ++i)

            Extension += (char)extension[i];

        result = Name + "." + Extension;

        return result;

    }

    return Name;

}

* Mặc khác ,nếu tập tin đang xét có ít nhất một entry phụ. Cách để lấy được tên đó là sẽ có vòng lặp ngược lại để truy cập về những entry phụ trước đó, khi đến được entry phụ cuối cùng, ta sẽ lần lượt lấy bytes đại diện cho tên từ những vị trí được quy ước sẵn:

    while (tmp - 32 >= 0 && data[tmp - 32 + 11] == 0x0F)

        tmp -= 32;

1. Lấy 10 bytes từ vị trí 01, sau đó lấy tiếp 12 bytes từ vị trí 0E, sau đó là 4 bytes từ vị trí 1C, gộp tất cả lại thành 1 vecto “nameZone” và chuyển kiểu dữ liệu thành char. Ta sẽ dùng 3 vòng lặp để lấy được các giá trị cần thiết trong mảng data.

*Lưu ý:* Các kí tự không được lưu liên tiếp nhau mà được lưu cách nhau bởi 1 byte 0x00, cũng như khi tên quá ngắn, nó sẽ được kết thúc sớm bằng giá trị 0xFF. Vì vậy khi lấy giá trị ta phải liên tục check xem nếu giá trị đó là 0x00 thì ta sẽ bỏ qua hoặc khi gặp giá trị 0xFF thì ta sẽ dừng lấy giá trị và tới bước tiếp theo. Tiếp theo tuy rằng FAT32 được tổ chức theo little edian (byte có giá trị thấp nhất được ghi đầu tiên), tuy nhiên tên ở dạng character được lưu dưới bảng mã ASCII với mỗi kí tự được biểu diễn bằng 1 bytes, vậy nên ta có thể bỏ qua việc phải đảo ngược lại thứ tự theo tư tưởng của little edian

string s = "";

        if (data[i] == 0x00 || data[i] == 0xE5)

            return s;

        bool flag = true;

        vector <uint8\_t> nameZone;

        // the next 3 for loop: get name

        for (int j = 0; j < 10; ++j){

            if (data[i + j + 1] == 0xFF){

                flag = false;

                break;

            } else if (data[i + j + 1] != 0x00){

                nameZone.push\_back(data[i + j + 1]);

            }

        }

        for (int j = 0; flag && j < 12; ++j){

            if (data[i + j + 14] == 0xFF){

                flag = false;

                break;

            } else if (data[i + j + 14] == 0x00){

                continue;

            } else {

                nameZone.push\_back(data[i + j + 14]);

            }

        }

        for (int j = 0; flag && j < 4; ++j){

            if (data[i + j + 28] == 0xFF){

                flag = false;

                break;

            } else if (data[i + j + 28] != 0x00){

                nameZone.push\_back(data[i + j + 28]);

            }

        }

// turn decimal to character

        for (auto j : nameZone){

            s += (char)j;

        }

        nameZone.clear();

        result.push\_back(s);

1. Push các string nameZone vào một vecto<string> để ta có thể dễ dàng chuyển ngược lại tên (vì tên được xếp theo thứ tự entry 1, entry 2, … entry N).

for (int j = result.size() - 1 ; j >=0; --j){

s += result[j];

}

### **Hàm lấy thông tin trạng thái**

A diagram of a diagram

Description automatically generatedThông tin tiếp theo chúng ta lấy sẽ là trạng thái của đối tượng (là tệp tin hay thư mục, thuộc về hệ thống hay là của người dùng, …). Các thông tin này sẽ được thể hiện qua byte tại vị trí 0B, khi chúng ta đổi giá trị này thành chuỗi nhị phân. Dựa theo hình kế bên, giá trị tại bit nào đang bật lên 1 tức là tệp tin đang có trạng thái đó.



Vd: chúng ta có một dãy bit 00010110

Ta thấy bit thứ 3, 5 và 6 đang bật lên 1 vậy nên đây là một thư mục hệ thống và bị ẩn khỏi người dùng.

Với lý thuyết đó, ta có thể dễ dàng lấy được giá trị bit tại vị trí ta muốn và kiểm tra bằng cách dịch phải và dùng phép & (and) với 1

int status = data[i + 11];

        bool isFile = (1 & (status >> 5));

        bool isFolder = (1 & (status >> 4));        //true: folder, false: file

        bool isSystem = (1 & (status >> 2));

        bool isHidden = (1 & (status >> 1));

### **getSize - Hàm lấy kích thước**

Thông tin tiếp theo chúng ta lấy được chính là kich thước của đối tượng. Ở dạng lưu trữ FAT32, chỉ có kích thước của tệp tin được lưu lại, còn kích thước của thư mục ta cần phải ấy từ tổng kích thước các tệp tin ở trong thư mục đó – điều ta sẽ làm ở *vector<\*Item>* sau này. Kích thước của tệp tin sẽ được biểu hiện bởi 4 bytes từ vị trí 1C.

*Lưu ý:* FAT32 được lưu trữ dưới dạng little-edian ( bytes có giá trị nhỏ nhất sẽ được ghi đầu tiên), vậy nên sau khi thấy được dãy bytes đó, ta cần đổi ngược lại rồi chuyển thành decimal, hoặc ta có thể lấy trực tiếp giá trị nhân với hệ số tương ứng để ra thẳng kết quả.

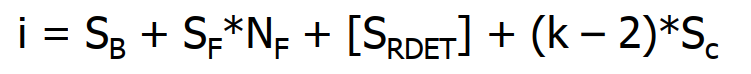
int getSize(vector<uint8\_t>& data, int idx){

    return data[idx + 28] + data[idx + 29] \* pow(16,2) + data[idx + 30] \* pow(16,4) + data[idx + 31] \* pow(16,6);}

### **getNextSector - Hàm lấy địa chỉ dữ liệu**

Đến đây, ta đã lấy được tất cả thông tin có thể nhận được từ đối tượng này, tiếp theo ta sẽ cần lấy được dữ liệu (đối với tệp tin) hoặc SDET (đối với thư mục), địa chỉ của chúng sẽ được lưu bởi 4 bytes tính từ vị trí 1C của entry chính biểu diễn đối tượng đang xét, giá trị này chính là chỉ số của cluster trên bảng FAT.

*Lưu ý:* ta cũng sẽ chuyển đổi giá trị được lưu theo định dạng little edian thành decimal để áp dụng vào công thức tương quan giữa cluster thứ k và sector thứ i trên vùng dữ liệu theo công thức:



Từ đó ta có đoạn code chuyển đổi sau:

int getNextSector(vector<uint8\_t>& data, int idx, int SectorBeforeFat, int SectorPerFat, int NumberOfFats, int SectorPerCluster, int RootCluster, uint32\_t& cluster){

    cluster = data[idx + 26] + data[idx + 27] \* pow(16,2) + data[idx + 21] \* pow(16,6) + data[idx + 20] \* pow(16,4) ;

    int result = SectorBeforeFat + SectorPerFat \* NumberOfFats + (cluster - RootCluster) \* SectorPerCluster;

    return result;

}

Từ đây, ta sẽ có được địa chỉ sector của SDET hoặc data cần lấy, việc ta cần làm là đếm trong bảng FAT xem dữ liệu này sẽ nằm trong bao nhiêu cluster, từ đó đếm được số sector sẽ đọc khi lấy dữ liệu tại địa chỉ đó. Giá trị của phần tử cluster của FAT có giá trị 0x0FFFFFFF có ý nghĩa là End of File (đã lấy được hết dữ liệu của đối tượng)

int numbCluster = 1;

        int tmp = startCluster;

        while (fatData[tmp++] != 0x0FFFFFFF)        // count number of cluster in that folder/file

            numbCluster++;

Vậy ta đã có số sector cần đọc sẽ là (số cluster) \* (Số sector trong 1 cluster)

### **ReadSector – Hàm đọc bytes biểu diễn thông tin Data**

Tiếp theo là đọc các dữ liệu có được ở sector đó trên vùng dữ liệu và chuyển thành vector số nguyên để đồng nhất với các hàm chúng ta đã phân tích ở trên, việc này sẽ được hỗ trợ phần lớn bởi các hàm có sẵn trong Windows API

vector<uint8\_t> Volume::ReadSector(int sector , int numSector){

    uint64\_t sectorOffset = sector \* bootSector.BytesPerSectorInt;

    // Seek to the beginning of the sector

    LARGE\_INTEGER liOffset;

    liOffset.QuadPart = sectorOffset;

    if (!SetFilePointerEx(bootSector.hDevice, liOffset, NULL, FILE\_BEGIN)){

        cerr << "Failed to seek to sector" << endl;

        exit(1);}

    // Read the sector

    std::vector<uint8\_t> sectorData(bootSector.BytesPerSectorInt  \* numSector);

    DWORD bytesRead;

    if (!ReadFile(bootSector.hDevice, sectorData.data(), bootSector.BytesPerSectorInt \* numSector , &bytesRead, NULL) || bytesRead != bootSector.BytesPerSectorInt \* numSector){

        cerr << "Failed to read sector" << endl;

        exit(1);}

    for (int i = 0; i < sectorData.size(); ++i)

        sectorData[i] = static\_cast<unsigned int> (sectorData[i]);

    return sectorData;}

### **readData – Hàm đọc dữ liệu**

* Bắt đầu từ việc đơn giản, ta sẽ đọc thông tin của tệp tin, nếu đối tượng đang xét là tập tin, ta chỉ việc lặp qua các phần tử trong mảng data và chuyển nó thành char, lưu ý việc bỏ qua những giá trị 0x00 được hệ thống thêm vào để thể hiện dữ liệu một cách chính xác.

string readData(vector<uint8\_t>& data){

    string result = "";

    int tmp = 0;

    while (data[tmp] != 0x00){

        result += (char)data[tmp++];

    }

    return result;

}

Ngoài ra đồ án này yêu cầu nếu tập tin đang xét là file txt thì mới thể hiện thông tin nên ta sẽ có thêm hàm riêng để check file txt thì print ra hoặc báo người dùng sẽ sử dụng phần mềm phù hợp để xem.

Sau khi lấy được các thông tin của đối tượng là file, ta sẽ tạo một đối tượng File để push vào vector<Item\*> phục vụ cho việc truy xuất sau này.

* Tiếp theo là nếu đối tượng đang xét là 1 folder, ta sẽ cần thấy thông tin SDET và đối xử với nó như là RDET – tức là ta sẽ dùng lại hàm TransRDET với những tham số truyền vào hợp lí để trả về kết quả ta mong muốn. vector <item\*> sau khi được trả về sẽ được nhập vào trong một đối tượng class Folder để lưu vào vector kết quả cuối cùng.

 vector<Item\*> data = TransRdet(sdet, false, sectorOfChild, startCluster);

            Folder\* folder = new Folder(Name, sector, isSystem, isHidden);

            for (auto i : data)

                folder->addItem(i);

            result.push\_back(folder);

Như vậy chúng ta đã lấy được hết tất cả các thông tin có được từ RDET và SDET, sau đó lưu vào một vector<Item\*> phục vụ cho việc truy xuất sau này.

# **Chạy demo sản phẩm FAT32 & RDET**

USB được format dưới dạng FAT32, chứa các folder và file như sau

A screenshot of a computer

Description automatically generated

Trong các file và folder này còn có rất nhiều folder và file con chứa nhiều nội dung khác nhau. Cụ thể sau khi chạy chương trình:

Đầu tiên ta có các thông tin cơ bản về FAT32, sau đó là các nội dung tập tin thư mục tương ứng

Vì việc lưu trữ dưới FAT32 đã biến các kí tự tên ở entry chính thành in hoa nên có một vài file ngắn sẽ được chỉnh thành in hoa, yêu cầu người dùng nhập đúng tên xuất hiện trên cây thư mục khi chương trình chạy để tìm đúng file / thư mục cần tìm

A screenshot of a computer

Description automatically generated

A computer screen shot of a program code

Description automatically generated

A computer screen shot of a computer program

Description automatically generated

A computer screen with white text

Description automatically generated

A black background with colorful text

Description automatically generated

# **Tổng quan về các hàm sử dụng cho việc đọc NTFS**

## **bool readSector(LPCWSTR drive, int readPoint, BYTE \*&sector)**

* Hàm nhận vào *`drive`* với kiểu dữ liệu *`LPCWSTR`* là 1 con trỏ đến chuỗi kí tự rộng (Long Pointer to Const Wide String). Trong chương trình Window, các chuỗi kí tự dài được sử dụng để xử lý văn bản Unicode. *`LPCWSTR`* cơ bản là một const *`wchar\_t\*`.* Tham số *`sector*` đại diện cho bộ nhớ tạm nơi dữ liệu được đọc từ sector được lưu trữ. Nó được xem như tham chiếu đến vùng nhớ nơi dữ liệu được ghi với kiểu dữ liệu *`BYTE`*. Tham số *`readPoint`* là 1 số nguyên thể hiện vị trí để đọc sector. Cụ thể trong trường hợp này tham số truyền vào từ hàm main là 0.

// Đọc sector 0 từ ổ đĩa "drive" và lưu vào "sector"

        int flag = readSector(drive, 0, sector);

hàm được gọi ở main

* Trong hàm khai báo 2 biến bytesRead với kiểu dữ liệu DWORD (Double Word) thể hiện một số nguyên 32 bits không dấu trong chương trình Window. Nó được sử dụng để lưu kích thước tập tin, kích thước bộ nhớ tạm, và lỗi code. Trong trường hợp này, bytesRead dùng để lưu số bytes đọc được từ ổ đĩa. Biến device với kiểu dữ liệu HANDLE được xem như một sự điều khiển cho đối tượng. Trong trường hợp này, nó thể hiện sự điều khiển với ổ đĩa cần đọc.

    DWORD bytesRead;      // lưu số bytes đọc được từ ổ đĩa

    HANDLE device = NULL; // lưu handle của ổ đĩa cần đọc

* Tiếp theo, hàm *`CreateFileW`*, một hàm Windows API dùng để xử lí tệp tin được sử dụng để mở một drive cụ thể với các tham số:
* *`drive`:* Ổ đĩa cần mở.
* *`GENERIC\_READ`:* Chế độ truy cập chỉ đọc vào tệp.
* *`FILE\_SHARE\_READ | FILE\_SHARE\_WRITE*`: Chế độ chia sẻ cho phép các tiến trình khác đọc và ghi vào tệp trong khi nó đang mở.
* *`NULL`:* Mô tả bảo mật mặc định.
* *`OPEN\_EXISTING`:* Chỉ định rằng tệp phải tồn tại để được mở.
* *`0`:* Thuộc tính tệp (không có thuộc tính nào được chỉ định trong trường hợp này).
* *`NULL`:* Bảng mẫu tệp (không được sử dụng).
* Sau đó là bước kiểm tra hàm đã mở thành công drive hay không

// Mở ổ đĩa

    device = CreateFileW(drive,                              // Drive

                         GENERIC\_READ,                       // Trạng thái truy cập

                         FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, // Share Mode

                         NULL,                               // Security Descriptor (default)

                         OPEN\_EXISTING,                      // cách tạo file

                         0,                                  // thuộc tính file

                         NULL);                              // xử lý file mẫu

    if (device == INVALID\_HANDLE\_VALUE) // Nếu không mở được file

    {

        cerr << "CreateFile : " << GetLastError() << endl; // In ra lỗi và thoát chương trình

        return 0;

    }

* Hàm *`SetFilePointer`* dùng để đặt con trỏ tệp tin vào vị trí cụ thể trong ổ đĩa. Hàm *`readFile`* truyền vào *`device`* để thực hiện đọc, *`sector`* trỏ đến 1 BYTE, hay nói cách khác trỏ đến vùng dữ liệu được viết, `512` là số byte để đọc từ tệp tin hoặc thiết bị. Trong trường hợp này được đặt tĩnh là 512, con số thường thấy về kích của 1 sector trên nhiều nhiều ổ đĩa drive. `byteRead` là con trỏ đến biến nhận số lượng byte được đọc thực tế bởi hàm `ReadFile`. Sau khi gọi hàm, `byteRead` chứa số lượng byte được đọc thành công từ tệp tin hoặc thiết bị. `NULL` là tham số tùy chọn và đậc thù cho địa chỉ của một cấu trúc overlapped trong sự vận hành nhập xuất không đồng bộ. Trong trường hợp này, sự vận hành được sử dụng, nên tham số được đặt là `NULL`.
* Hàm `ReadFile` trả về giá trị boolean. `TRUE` nếu như thành công và `FALSE nếu lỗi. Trong trường hợp này, lỗi trả về dạng tin nhắn và trả về 0 sau bước kiểm tra, hoặc thông báo thành công về trả về 1 cùng dữ liệu sau khi đã đọc được lưu vào sector trong main.

// Đặt vị trí con trỏ file tại vị trí "readPoint" truyền vào bằng 0

    SetFilePointer(device, readPoint, NULL, FILE\_BEGIN);

    // Đọc sector

    if (!ReadFile(device, sector, 512, &bytesRead, NULL))

    {

        cerr << "ReadFile : " << GetLastError() << endl;

        return 0;

    }

    // Đóng file

    else

    {

        cout << "Thanh cong doc sector" << endl;

        cout << endl;

        return 1;

    }

## **void readSect2(LPCWSTR disk, BYTE \*&DATA, unsigned int \_nsect)**

* Hàm nhận vào 3 tham số: Chuỗi ký tự rộng `disk` đại diện cho ổ đĩa từ đó sẽ đọc các sector. Tham chiếu `DATA` đến con trỏ BYTE, sẽ chứa dữ liệu được đọc từ các sector. `\_nsect` là một số nguyên không dấu, chỉ ra số lượng sector muốn đọc.
* Biến dwBytesRead có kiểu dữ liệu là DWORD, được sử dụng để lưu trữ số byte thực sự được đọc bởi hàm ReadFile. Ban đầu, nó được khởi tạo với giá trị 0.

A white text on a blue background

Description automatically generated

* Biến hDisk được khai bảo để xử lí ổ đĩa, tương tự như ở hàm `ReadFile`. Tiếp theo tiến hành mở ổ đĩa bằng cách sử dụng hàm CreateFileW, với mục đích tạo hoặc mở thiết bị ổ đĩa cụ thể để đọc. Tham số `disk` là một chuỗi ký tự rộng, đại diện cho ổ đĩa sẽ được mở. Chế độ truy cập `GENERIC\_READ` được sử dụng để chỉ định quyền đọc cho ổ đĩa. Chế độ chia sẻ `FILE\_SHARE\_READ | FILE\_SHARE\_WRITE` cho phép các tiến trình khác có thể đọc và ghi vào ổ đĩa trong khi nó đang mở. Tham số `NULL` được sử dụng cho mô tả bảo mật mặc định. `OPEN\_EXISTING` chỉ định rằng ổ đĩa phải tồn tại trước khi nó có thể được mở. Giá trị `0` được sử dụng cho các thuộc tính tệp (trong trường hợp này không có thuộc tính được chỉ định). Tham số cuối cùng, `NULL`, không được sử dụng và đại diện cho cán mẫu tệp.

HANDLE hDisk = NULL;

    hDisk = CreateFileW(disk, // Floppy drive cần mở

                        GENERIC\_READ,

                        FILE\_SHARE\_READ | FILE\_SHARE\_WRITE,

                        NULL,

                        OPEN\_EXISTING,

                        0,

                        NULL);

    if (hDisk != NULL)

    {

        LARGE\_INTEGER li;

        li.QuadPart = \_nsect \* 512;

        // Đặt vị trí con trỏ file tại vị trí "li"

        SetFilePointerEx(hDisk, li, 0, FILE\_BEGIN);

        // Đọc boot sector

        if (!ReadFile(hDisk, DATA, 512, &dwBytesRead, NULL))

        {

            cerr << "Xay ra loi khi doc sector" << endl;

        }

        CloseHandle(hDisk);

* Tiếp theo kiểm tra việc mở tệp tin thành công hay không. Nếu thành công, khai báo biến li với kiểu dữ liệu `LARGE\_INTEGER`. `LARGE\_INTEGER` là một cấu trúc được định nghĩa trong API Windows. Thường được sử dụng để biểu diễn một giá trị số nguyên 64-bit có dấu. Cấu trúc LARGE\_INTEGER thường được sử dụng để xử lý kích thước tệp lớn, độ lệch đĩa, hoặc các giá trị số nguyên lớn khác yêu cầu nhiều hơn 32-bit để biểu diễn chính xác. Trong trường hợp này, `li` được khai báo và đặt `QuadPart`, 1 thành viên trong Windows API, thể hiện 1 giá trị số nguyên 64 bit có dấu thành \_nsect \* 512. Điều này tính toán offset byte tuyệt đối từ đầu đĩa nơi việc đọc bắt đầu. Tiếp theo hàm `SetFilePointerEx`, một hàm API Windows được sử dụng để thiết lập vị trí con trỏ tệp cho một tệp hoặc thiết bị cho phép tìm kiếm, như một tệp đĩa, được gọi là `SetFilePointer`. Hàm này cho phép di chuyển con trỏ tệp đến một vị trí cụ thể bên trong tệp hoặc thiết bị. Trong trường hợp này, con trỏ tệp của ổ đĩa vào vị trí được chỉ định bởi biến li. Điều này di chuyển con trỏ tệp đến vị trí được chỉ định bên trong ổ đĩa. Hàm kiểm tra `ReadFile` được sử dụng với ý tưởng tương tự đọc sector ở trên. Cuối cùng, sử dụng `CloseHandle`, một hàm API Windows được sử dụng để đóng một handle mở đến một tệp, thiết bị hoặc tài nguyên hệ thống khác, để giải phóng các tài nguyên liên kết với nó và tránh rò rỉ tài nguyên trong trường hợp trên.

## **int64\_t getBytes(BYTE \*sector, int offset, int number)**

* Hàm nhận vào Con trỏ `BYTE \*sector` trỏ tới dữ liệu sector từ đó muốn trích xuất byte. Tham số `int offset` đại diện cho độ lệch trong sector từ nơi bắt đầu trích xuất byte. Và cuối cùng, tham số `int number` chỉ ra số lượng byte cần trích xuất từ sector này.

// Lấy number bytes từ vị trí offset

int64\_t getBytes(BYTE \*sector, int offset, int number)

{

    int64\_t extractedBytesToInteger = 0;

    memcpy(&extractedBytesToInteger, sector + offset, number);

    return extractedBytesToInteger;

}

* Đầu tiên, hàm khai báo `extractedBytesToInteger` với kiểu `int64\_t` và gán bằng 0 sau đó sử dụng hàm memcpy để sao chép số lượng byte đã chỉ định từ dữ liệu sector bắt đầu từ offset vào vị trí bộ nhớ được trỏ tới bởi extractedBytesToInteger. Cuối cùng trả về giá trị extractedBytesToInteger.

## **string numToString(BYTE \*DATA, int offset, int number)**

* Hàm nhận vào con trỏ `BYTE \*DATA` trỏ tới mảng dữ liệu mà muốn trích xuất ký tự từ đó. Tham số `int offset` đại diện cho độ lệch trong mảng dữ liệu từ nơi bắt đầu trích xuất ký tự. Cuối cùng, tham số `int number` chỉ ra số lượng ký tự cần trích xuất.

// Chuyển số lượng bytes Data từ vị trí offset thành string

string numToString(BYTE \*DATA, int offset, int number)

{

    // tạo mảng temp để lưu kí tự

    char \*temp = new char[number + 1];

    memcpy(temp, DATA + offset, number);

    string result = "";

    // chuyển mảng temp thành string

    for (int i = 0; i < number; ++i)

        if (temp[i] != 0x00)

            result += temp[i];

    delete[] temp;

    return result;

}

* Đầu tiên cấp phát bộ nhớ động cho một mảng ký tự `temp` có kích thước là number + 1. Khoảng trắng bổ sung là cho null terminator ('\0') cần thiết cho chuỗi kiểu C/C++. Tiếp theosử dụng hàm `memcpy` để sao chép số lượng ký tự được chỉ định từ mảng dữ liệu bắt đầu từ offset vào mảng ký tự `temp`. Vòng lặp duyệt mỗi ký tự trong mảng temp và thêm các ký tự không null vào chuỗi kết quả. Điều kiện đảm bảo rằng chỉ có các ký tự không null được thêm vào chuỗi kết quả. Cuối cùng giải phóng bộ nhớ temp và trả về chuỗi kết quả.

## **string decimalToBinary(int number)**

* Hàm nhận vào tham số `number` là số hệ 10 cần chuyển sang hệ 2 dưới dạng string.

// Chuyển hệ 10 sang hệ 2

string decimalToBinary(int number)

{

    string result = "";

    while (number > 0)

    {

        result = to\_string(number % 2) + result;

        number /= 2;

    }

    return result;

}

* Bắt đầu vòng lặp tiếp tục cho đến khi biến number lớn hơn 0. Trong vòng lặp, phép chia lấy phần dư `number % 2` tính toán phần dư khi số number được chia cho 2. Phần dư này có thể là 0 hoặc 1, đại diện cho chữ số nhị phân tiếp theo. Hàm `to\_string()` chuyển đổi phần dư này (hoặc 0 hoặc 1) thành một chuỗi. Chuỗi này sau đó được nối vào đầu chuỗi kết quả. Bước này đảm bảo rằng các chữ số nhị phân được thêm vào theo đúng thứ tự từ phải sang trái. Cuối cùng trả về chuỗi nhị phân kết quả.

## **printTab(int tab)**

* Hàm dùng để in ‘\t’. Bắt đầu một vòng lặp, in ra một ký tự tab (\t) kèm theo một dấu gạch ngang (-).

// Hàm in "tab" lần /t

void printTab(int tab)

{

    for (int i = 0; i < tab; ++i)

        cout << "\t-";

}

## **string getNameFile(int id)**

* Hàm nhận một tham số là số nguyên `id` đại diện cho ID của tệp cần lấy và trả về một chuỗi chứa tên tệp tương ứng. Hàm này tìm kiếm thông qua mảng `fileID` để tìm chỉ mục của id được cho, và nếu tìm thấy, nó sẽ lấy tên tệp tương ứng từ mảng `nameFile` và trả về nó. Ngược lại, trả về chuỗi rỗng. Các `getNameFile` cung cấp một cơ chế để lấy tên tệp dựa trên ID của chúng.

// lấy tên file trong mảng nameFile có ID file là "id"

string getNameFile(int id)

{

    string res = ""; // Khởi tạo res là chuỗi rỗng để lưu trữ tên file

    // Xác định index id trong fileID

    int pos = -1;                           // Khởi tạo pos là -1, đại diện cho trường hợp không tìm thấy id trong fileID

    for (int i = 0; i < fileID.size(); ++i) // Duyệt qua mảng fileID để tìm id

        if (fileID[i] == id)                // Nếu tìm thấy id trong mảng fileID

        {

            pos = i; // Lưu lại vị trí của id trong fileID

            break;   // Thoát khỏi vòng lặp

        }

    if (pos != -1)           // Nếu pos khác -1 (tức là tìm thấy id trong fileID)

        res = nameFile[pos]; // Gán tên file từ mảng nameFile tại vị trí pos vào res

    return res;              // Trả về tên file

}

## **printSector(BYTE \*sector)**

* Hàm nhận vào biến `sector` kiểu dữ liệu BYTE. Gán các giá trị ban đầu cho các biến `count`, `num`, và `flag` theo dõi số lượng byte được in trên mỗi dòng, số dòng hiện tại đang được xử lý và xác định liệu hàm đang in nửa đầu hoặc nửa sau của sector, tương ứng. Sau khi khởi tạo, hàm tiếp tục in ra header chỉ định vị trí thập lục phân của các byte, cung cấp bối cảnh quan trọng để giải mã dữ liệu byte sau đó. Để bắt đầu đầu ra, hàm in ra số dòng 0x00, thiết lập điểm bắt đầu cho biểu diễn dữ liệu byte.

// In bảng sector

void printSector(BYTE \*sector)

{

    int count = 0;

    int num = 0;

    bool flag = false;

    cout << "         0  1  2  3  4  5  6  7    8  9  A  B  C  D  E  F" << endl;

    cout << "0x0" << num << "0  ";

* Trong quá trình khởi tạo vòng lặp, chương trình bắt đầu duyệt qua từng byte trong sector để xử. Biến `count` được tăng lên mỗi khi một byte được in ra, để theo dõi số lượng byte đã được in. Hai dấu cách được thêm vào mỗi khi đã in ra 8 byte để tạo ra định dạng cho kết quả đầu ra,. Mỗi byte được in ra dưới dạng thập lục phân, cung cấp thông tin chi tiết về nội dung của sector.

    for (int i = 0; i < 512; ++i)

    {

        count++;

        if (i % 8 == 0)

            cout << "  ";

        printf("%02X ", sector[i]);

* Khi đạt đến byte thứ 255, biến `flag` được thiết lập thành `true` để biểu thị rằng nửa thứ hai của `sector` đang được in ra, và số dòng được đặt lại về `0`. Vòng lặp kết thúc khi tất cả `512 byte` đã được in ra. Mỗi khi in ra `16 byte` (cuối dòng), hàm bắt đầu một dòng mới và in ra `số dòng` dưới dạng thập lục phân. Biến `flag` được sử dụng để xác định xem nửa đầu hay nửa sau của `sector` đang được in ra, hàm kiểm tra số dòng `num` có nhỏ hơn 10 không. Nếu có, nó in ra `0x1` tiếp theo là số dòng (num) và sau đó là 0 để duy trì định dạng thập lục phân. Nếu `num` lớn hơn hoặc bằng 10, nó chuyển đổi số dòng thành ký tự thập lục phân (A-F) và in ra thay vì giá trị số học và biến `num` được tăng lên tương ứng. Sau đó, biến `count` được thiết lập lại cho dòng mới. Cuối cùng, một dòng mới được in ra để hoàn thiện định dạng của đầu ra.

        if (i == 255)

        {

            flag = true;

            num = 0;

        }

        if (i == 511)

            break;

        if (count == 16)

        {

            int index = i;

            cout << endl;

            if (!flag)

            {

                ++num;

                if (num < 10)

                    cout << "0x0" << num << "0  ";

                else

                {

                    char hexa = char(num - 10 + 'A');

                    cout << "0x0" << hexa << "0  ";

                }

            }

            else

            {

                if (num < 10)

                    cout << "0x1" << num << "0  ";

                else

                {

                    char hexa = char(num - 10 + 'A');

                    cout << "0x1" << hexa << "0  ";

                }

                ++num;

            }

            count = 0;

        }

    }

    cout << endl;

}

* Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, thực đơn

  Mô tả được tạo tự độngTest case khi chạy chương trình:

## **printFolderTree(int id, int tab, int pos)**

* -Hàm nhận vào `id`: Đại diện cho ID của thư mục hoặc tệp hiện tại trong cấu trúc phân cấp. `tab`: Chỉ ra mức hiện tại của lùi vào để định dạng đúng cấu trúc cây. `pos`: Đại diện cho vị trí của thư mục hoặc tệp hiện tại trong các vector `fileID` và `parentID`.
* -Đầu tiên, tăng biến tab để tăng mức độ lùi vào cho việc in ra tiếp theo, sau đó gọi hàm `printTab` để in ra các khoảng trống lùi vào tương ứng với mức độ lùi hiện tại. Tiếp theo in ra tên của thư mục hoặc tệp tương ứng với `id` đã cho bằng cách sử dụng hàm `getNameFile`. Cuối cùng, khởi tạo hai vector `child` và `POS` để lưu trữ các ID và vị trí của các con tương ứng.

    tab++;

    printTab(tab);

    cout << getNameFile(id) << endl;

    // Cho ID với parents = -1

    fileID[pos] = -1;

    parentID[pos] = -1;

    vector<int> child;

    vector<int> POS;

* Đến vòng lặp, tất cả các phần tử trong `fileID` để tìm các con của thư mục hoặc tệp hiện tại được duyệt (được xác định bởi `id`) để đẩy vào các vector `child` và `POS` với các ID và vị trí của các con này. Nếu không tìm thấy con nào, hàm trả về, vì không cần phải liệt kệ cấu trúc cây từ nút này nữa. Trong trường hợp tìm thấy các con, hàm gọi đệ quy chính nó cho mỗi con để in ra cấu trúc cây của chúng. Điều này cho phép duyệt toàn bộ các thư mục/tệp trong hệ thống phân cấp.

    // Tìm số lượng con

    for (int j = 0; j < fileID.size(); ++j)

        if (parentID[j] == id)

        {

            child.push\_back(fileID[j]);

            POS.push\_back(j);

        }

    if (child.size() == 0)

        return;

    // In con của từng phần tử

    for (int i = 0; i < child.size(); ++i)

        printFolderTree(child[i], tab, POS[i]);

* Test case và thực tế trong File Explorer:

Ảnh có chứa ảnh chụp màn hình

Mô tả được tạo tự động

A screenshot of a computer

Description automatically generated

## **readEntryInformation(BYTE \*Entry, int start)**

* Hàm nhận vào tham số `\*Entry` là một con trỏ đến một khối bộ nhớ chứa thông tin về mục nhập. Hàm sử dụng con trỏ này để đọc các byte từ mục nhập và lấy thông tin về loại thuộc tính và kích thước của nó. Trong khi đó, tham số `start` chỉ ra vị trí bắt đầu trong mục nhập mà hàm nên bắt đầu đọc thông tin. Tham số này được sử dụng như một độ lệch để xác định các byte cụ thể trong bộ đệm mục nhập, giúp hàm truy cập vào thông tin cần thiết để xử lý. Hàm `getBytes` được gọi để lấy các byte này từ bộ đệm `Entry`.Hàm chuyển đổi biến trạng thái (loại thuộc tính) sang định dạng nhị phân bằng cách sử dụng hàm `decimalToBinary` giúp kiểm tra các bit cụ thể trong biểu diễn nhị phân.

    // Byte thứ 0 đến 3, Type của attribute

    int status = getBytes(Entry, start + 56, 4);

    // chuyển status sang hệ 2

    string bin = decimalToBinary(status);

* Tiếp theo, hàm lặp qua từng bit của biểu diễn nhị phân của biến trạng thái. Nếu bit cuối cùng được đặt thành 1, nó kiểm tra các bit khác để tìm các `cờ`. Chỉ Đọc (if (i == n - 2)): Nếu bit là bit thứ hai từ cuối (n - 2), điều này chỉ ra rằng hệ thống tệp không thể được đọc. Trong trường hợp này, hàm trả về -1 để chỉ ra một lỗi hoặc không thể đọc được hệ thống tệp.Tệp Ẩn (if (i == n - 3)): Nếu bit là bit thứ ba từ cuối (n - 3), có thể chỉ ra rằng tệp được ẩn hoặc có các thuộc tính cụ thể, trả về -1.

    // Kiểm tra các bits ở vị trí cuối cùng của status

    for (int i = bin.length() - 1; i >= 0; --i)

    {

        int n = bin.length();

        // Nếu bit cuối cùng = 1 thì kiểm tra các bit khác

        if (bin[i] == '1')

        {

            // Read Only

            if (i == n - 2)

            {

                // Hệ thống tập tin không thể đọc

                return -1;

            }

            // Hidden file

            if (i == n - 3)

            {

                // Hệ thống tập tin không thể ghi

                return -1;

            }

        }

    }

* Hàm in ra thông tin về thuộc tính (Thuộc Tính $STANDARD\_INFORMATION), kích thước của nó (được lấy từ byte 4 đến 7 của entry), và trạng thái của nó (biểu diễn nhị phân).

    cout << "Attribute $STANDARD\_INFORMATION" << endl;

    // Byte thứ 4 đến 7, Kích thước của attribute

    int size = getBytes(Entry, start + 4, 4);

    cout << "\t- Do dai attribute (bao gom header): " << size << endl;

    cout << "\t- Trang thai attribute: " << bin << endl;

* In hàm này tiếp tục kiểm tra các bit cụ thể của biến trạng thái và in thêm thông tin dựa trên giá trị của chúng. Ví dụ, nếu bit cuối cùng được đặt, điều này chỉ ra rằng thuộc tính là `Read Only`. Các bit khác cũng được giải mã tương tự để xác định các cờ như `Vollabel`, `Directory` và `Archive`. Cuối cùng, hàm trả về kích thước của thuộc tính, đã được lấy trước đó.

// Duyệt qua các bit của status

    for (int i = bin.length() - 1; i >= 0; --i)

    {

        // Kiểm tra các bit cuối cùng của status

        int n = bin.length();

        if (bin[i] == '1')

        {

            if (i == n - 1)

                cout << "\t\t => Read Only" << endl;

            if (i == n - 4)

                cout << "\t\t => Vollabel" << endl;

            if (i == n - 5)

                cout << "\t\t => Directory" << endl;

            if (i == n - 6)

                cout << "\t\t => Archive" << endl;

        }

    }

    cout << endl;

    // Trả về size của attribute

    return size;

}

## **readEntryFileName(BYTE \*Entry, int start, int ID)**

* Hàm nhận vào con trỏ `\*Entry` trỏ đến một khối bộ nhớ chứa thông tin về mục nhập, được sử dụng để đọc các byte từ mục nhập để thu thập thông tin liên quan đến tên tệp. Tham số `start` xác định vị trí bắt đầu trong entry mà hàm nên bắt đầu đọc thông tin, là một offset để xác định các byte cụ thể trong bộ đệm entry. In tên của thuộc tính đang được xử lý là "$FILE\_NAME". Tiếp theo, hàm thu thập kích thước của thuộc tính (bao gồm cả tiêu đề) bằng cách gọi hàm `getBytes`. Nó đọc 4 bytes bắt đầu từ vị trí `start + 4` trong bộ đệm `Entry`. Sau đó, nó in ra kích thước của thuộc tính. Sau đó, hàm thu thập ID của tệp cha bằng cách đọc 6 bytes bắt đầu từ vị trí `start + 24` trong bộ đệm `Entry`. Nó in ra ID của tệp cha và thêm nó vào vector `parentID`.

    cout << "Attribute $FILE\_NAME" << endl;

    int size = getBytes(Entry, start + 4, 4);

    cout << "\t- Do dai attribute (bao gom header): " << size << endl;

    int parent\_file = getBytes(Entry, start + 24, 6);

    cout << "\t- File cha: " << parent\_file << endl;

    parentID.push\_back(parent\_file);

* Hàm trích xuất và in ra tên tệp bằng cách gọi hàm `numToString`, chuyển đổi một chuỗi byte thành một chuỗi. Nó đọc `lengthName \* 2` bắt đầu từ vị trí start + 90 trong bộ đệm `Entry`. Sau đó in ra tên tệp. Phần này của mã trích xuất phần mở rộng từ tên tệp. Nó lặp lại ngược qua các ký tự của tên tệp, dừng lại khi gặp '.' hoặc sau khi kiểm tra 5 ký tự cuối cùng. Xây dựng chuỗi phần mở rộng tệp (extensions) theo thứ tự ngược lại và sau đó đảo ngược để có được phần mở rộng đúng.

    // Lấy đuôi mở rộng

    string extensions = "";

    for (int i = name.length() - 1; i >= name.length() - 5; i--)

    {

        if (name[i] == '.')

            break;

        extensions += name[i];

    }

    reverse(extensions.begin(), extensions.end());

* Tiếp theo, hàm kiểm tra và in định dạng tệp được hỗ trợ: Tùy thuộc vào phần mở rộng tệp (extensions), hàm kiểm tra các định dạng tệp cụ thể được hỗ trợ và in các thông báo tương ứng chỉ ra phần mềm nào có thể được sử dụng để mở tệp đó. Nếu phần mở rộng tệp là "txt", biến `check` toàn cục thành true. Sau đó, nó thêm tên tệp vào vector nameFile và in ra một dòng trống. Cuối cùng, hàm trả về kích thước của thuộc tính (bao gồm cả tiêu đề).

    // Hỗ trợ đọc file

    cout << "\t- Phan mo rong: " << extensions << endl;

    if (extensions == "doc" || extensions == "docx")

        cout << "\t\t Xin vui long su dung Microsoft Office Word de mo.\n";

    if (extensions == "ppt" || extensions == "pptx")

        cout << "\t\t Xin vui long Su dung Microsoft Office PowerPoint de mo\n";

    if (extensions == "xls" || extensions == "xlsx")

        cout << "\t\t Xin vui long Su dung Microsoft Office Excel de mo\n";

    if (extensions == "sln" || extensions == "cpp" || extensions == "java" || extensions == "html" || extensions == "css")

        cout << "\t\t Xin vui long Su dung Microsoft Visual Studio de mo\n";

    if (extensions == "pdf")

        cout << "\t\t Xin vui long Su dung Foxit PDF Reader hoac Trinh duyet Web (Edge, Chrome, ...) de mo\n";

    if (extensions == "txt")

        check = true;

    nameFile.push\_back(name);

    cout << endl;

    return size;

}

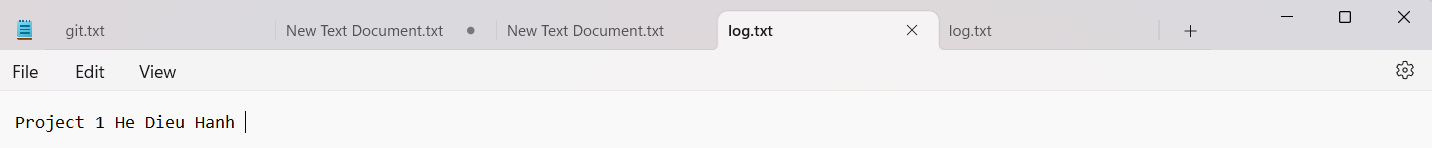
Ảnh có chứa văn bản, ảnh chụp màn hình

Mô tả được tạo tự độngẢnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động



Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

Ảnh có chứa văn bản, ảnh chụp màn hình

Mô tả được tạo tự động

## **readEntryData(BYTE \*Entry, int start)**

* Hàm nhận vào con trỏ `\*Entry` trỏ đến một khối bộ nhớ chứa thông tin về entry, được sử dụng để đọc các bytes từ entry để lấy thông tin liên quan đến dữ liệu. Tham số `start` xác định vị trí bắt đầu trong entry mà hàm nên bắt đầu đọc thông tin và được sử dụng như một offset để xác định các byte cụ thể trong buffer entry. Đầu tiên, tên của thuộc tính đang được xử lý được in ra là "$DATA". Hàm truy xuất kích thước của thuộc tính (bao gồm cả tiêu đề) bằng cách gọi hàm `getBytes`. Nó đọc 4 bytes bắt đầu từ vị trí `start + 4` trong bộ đệm `Entry`. Sau đó, nó in ra kích thước của thuộc tính. Hàm truy xuất kích thước của tệp liên quan đến thuộc tính dữ liệu. Nó đọc 4 bytes bắt đầu từ vị trí `start + 16` trong bộ đệm `Entry`. Sau đó, nó in ra kích thước của tệp.

    // In thông tin của attribute $DATA

    cout << "Attribute $DATA" << endl;

    int size = getBytes(Entry, start + 4, 4);

    cout << "\t- Do dai attribute (bao gom header): " << size << endl;

    int sizeFile = getBytes(Entry, start + 16, 4);

    cout << "\t- Kich thuoc File: " << sizeFile << endl;

* Tiếp theo, hàm `getBytes` truy xuấy loại dữ liệu (resident data hoặc hoặc non-resident data) liên quan đến thuộc tính. Nó đọc 1 bytes bắt đầu từ vị trí `start + 8` trong bộ đệm `Entry`. Nếu dữ liệu là resident data và là một tệp văn bản ("txt"), nó in ra " Resident data" và thực hiện xử lý thu thập và in ra nội dung của tệp. Nó đọc kích thước và vị trí bắt đầu của nội dung trong bộ đệm `Entry`, chuyển đổi thành một chuỗi, và sau đó in ra nội dung. Ngược lại in ra "Non-resident”.

    // lấy kiểu của attribute

    int type = getBytes(Entry, start + 8, 1);

    // Kiểm tra type của attribute

    if (type == 0 && check == true)

    {

        cout << "\t\t => Resident data:" << endl;

        int cont\_Size = getBytes(Entry, start + 16, 4);

        int cont\_Start = getBytes(Entry, start + 20, 2);

        string content = numToString(Entry, start + cont\_Start, cont\_Size);

        cout << endl;

        cout << "Noi dung: " << content << endl;

    }

    else

        cout << "\t\t=> Non-resident data" << endl;

    cout << endl;

## **readBPB(BYTE \*sector, LPCWSTR disk)**

* Hàm nhận vào con trỏ `\*sector` chỉ đến một khối bộ nhớ chứa dữ liệu sector BPB (BIOS Parameter Block). Hàm sử dụng con trỏ này để đọc các tham số khác nhau từ sector này và trích xuất thông tin về hệ thống tệp. Tham số `LPCWSTR disk` đại diện cho đường dẫn hoặc định danh của ổ đĩa đang được đọc. Điều này có thể được sử dụng cho các hoạt động tiếp theo hoặc để xác định rõ ổ đĩa đang phân tích. Tiếp theo, hàm đọc lần lượt để lấy số byte mỗi sector từ sector BPB. Nó đọc 2 bytes bắt đầu từ vị trí offset 0x0B trong bộ đệm sector sử dụng hàm `getBytes`. Tương tự cho lấy số sector mỗi cluster từ sector BPB. Nó đọc 1 byte bắt đầu từ vị trí offset 0x0D trong bộ đệm sector; Lấy số sector mỗi track từ sector BPB. Nó đọc 2 bytes bắt đầu từ vị trí offset 0x18 trong bộ đệm sector và lấy tổng số sector từ sector BPB. Nó đọc 8 bytes bắt đầu từ vị trí offset 0x28 trong bộ đệm sector. Tiếp theo, lấy khối bắt đầu của bảng tệp chính (MFT) từ sector BPB. Nó đọc 8 bytes bắt đầu từ vị trí offset 0x30 trong bộ đệm sector. Tương tự cho bảng tệp chính (MFT) phản chiếu từ sector BPB. Nó đọc 8 bytes bắt đầu từ vị trí offset 0x38 trong bộ đệm sector.

    unsigned int bytesPerSector = getBytes(sector, 0x0B, 2);    // Bytes Per Sector

    unsigned int sectorsPerCluster = getBytes(sector, 0x0D, 1); // Sectors Per Cluster

    unsigned int sectorsPerTrack = getBytes(sector, 0x18, 2);   // Sectors Per Track

    unsigned int totalSectors = getBytes(sector, 0x28, 8);      // Total Sectors

    unsigned int MFTStart = getBytes(sector, 0x30, 8);          // Cluster start of MFT

    unsigned int MFTMirrorStart = getBytes(sector, 0x38, 8);    // Cluster start of MFTMirror

* Sau đó, hàm thực hiện in lần lượt các thông số đã được đọc bao gồm số byte mỗi sector, số sector mỗi cluster, số sector mỗi track, tổng số sector, khối bắt đầu của bảng tệp chính (MFT), và khối bắt đầu của bảng tệp chính (MFT) phản chiếu, lên cửa sổ console. Cuối cùng, hàm `readMFT` được gọi truyền vào khối bắt đầu của MFT, số sector mỗi nhóm và định danh của ổ đĩa làm tham số để khởi tạo việc đọc các mục của Master File Table (MFT).

    cout << "\nSo Bytes moi Sector : " << bytesPerSector << endl;

    cout << "So Sectors moi Cluster : " << sectorsPerCluster << endl;

    cout << "So Sectors moi Track : " << sectorsPerTrack << endl;

    cout << "Tong so Sectors : " << totalSectors << endl;

    cout << "Cluster bat dau cua MFT : " << MFTStart << endl;

    cout << "Cluster bat dau cua MFTMirror : " << MFTMirrorStart << endl;

    cout << endl;

    // Đọc $MFT Entry

    readMFT(MFTStart, sectorsPerCluster, disk);

}

* Test case khi chạy chương trình:

Ảnh có chứa văn bản, Phông chữ, ảnh chụp màn hình

Mô tả được tạo tự động

## **readMFT(unsigned int MFTStart, unsigned int sectors\_per\_cluster, LPCWSTR disk)**

* Hàm nhận vào tham số `MFTStart` là chỉ số của cluster bắt đầu của Master File Table (MFT), được nhân với số sector mỗi nhóm để xác định sector bắt đầu của MFT. Biến `sectors\_per\_cluster` là số sector mỗi nhóm và được sử dụng để tính toán sector bắt đầu của MFT. Biến `disk` chỉ đường dẫn hoặc định danh của ổ đĩa đang được đọc, có thể được sử dụng cho các thao tác tiếp theo trên ổ đĩa. Đầu tiên, hàm cấp phát bộ nhớ động để lưu trữ các mục của MFT. Nó cấp phát bộ nhớ kích thước tiêu chuẩn 512 bytes. Tiếp theo, hàm tính toán sector bắt đầu của MFT dựa trên khối bắt đầu của MFT và số sector mỗi nhóm. Sau đó đọc dữ liệu sector vào bộ đệm MFT bằng cách sử dụng hàm `readSector2`.

    BYTE \*MFT = new BYTE[512];

    MFTStart \*= sectors\_per\_cluster;

    readSector2(disk, MFT, MFTStart);

* Tiếp theo, Dòng `int entryInformation = getBytes(MFT, 0x014, 2)` đọc 2 bytes từ bộ đệm MFT bắt đầu từ vị trí `0x014`, có thể chứa thông tin liên quan đến thuộc tính `$INFORMATION` trong MFT. Giá trị thu được được lưu trong biến `entryInformation`. Tương tự, `int informationLength = getBytes(MFT, 0x048, 4)` trích xuất 4 bytes bắt đầu từ vị trí `0x048` trong bộ đệm MFT, độ dài của thuộc tính `$INFORMATION`. Giá trị thu được được lưu trong biến `informationLength`. Tiếp theo, `int entryName = entryInformation + informationLength` tính toán số thứ tự bắt đầu của thuộc tính `File name` bằng cách cộng các giá trị `entryInformation` và `informationLength`. Kết quả được lưu trong `entryName`. Cuối cùng, `int nameLength = getBytes(MFT, 0x09C, 4)` đọc 4 bytes từ bộ đệm MFT bắt đầu từ vị trí `0x09C`, có thể trích xuất độ dài của mục thuộc tính `File name`, sau đó lưu trong `nameLength`.

    // INFORMATION

    int entryInformation = getBytes(MFT, 0x014, 2);

    cout << "Attribute $INFORMATION Entry bat dau tai: " << entryInformation << endl;

    int informationLength = getBytes(MFT, 0x048, 4);

    cout << "Do dai cua Infomation Entry: " << informationLength << endl;

    // FILE NAME

    int entryName = entryInformation + informationLength;

    cout << "Attribute $FILE NAME Entry bat dau tai: " << entryName << endl;

    int nameLength = getBytes(MFT, 0x09C, 4);

    cout << "Do dai cua $FILE NAME Entry: " << nameLength << endl;

* Sau đó, dòng `int temp = getBytes(MFT, 0x108, 4)` đọc 4 bytes bắt đầu từ vị trí `0x108` trong bộ đệm MFT, có thể trích xuất thông tin liên quan đến mục thuộc tính `$DATA`, và giá trị thu được được lưu trong biến `temp`. Sau đó kiểm tra giá trị temp để xác định loại dữ liệu `$DATA` là `resident` hay `non-resident`. Nếu `temp` bằng 64 mã tính toán số thứ tự bắt đầu của thuộc tính `$DATA` và trích xuất độ dài của nó. Nếu `temp` không bằng 64, tính toán kích thước của thuộc tính dữ liệu chính trong mục MFT và xử lý cấu trúc cây thư mục. Cuối cùng, bộ nhớ được giải phóng để tránh rò rỉ bộ nhớ.

    // DATA

    int temp = getBytes(MFT, 0x108, 4);

    int entryData;

    if (temp == 64)

    {

        entryData = entryName + nameLength + getBytes(MFT, 0x10C, 4);

        cout << "Attribute $DATA Entry bat dau tai: " << entryData << endl;

        int dataLength = getBytes(MFT, 0x134, 4);

        cout << "Do dai cua Data Entry: " << dataLength << endl;

    }

    else

    {

        entryData = entryName + nameLength;

        cout << "Attribute $DATA Entry bat dau tai: " << entryData << endl;

        int dataLength = getBytes(MFT, 0x10C, 4);

        cout << "Do dai cua Data Entry: " << dataLength << endl;

    }

    // main DATA

    unsigned int MFTLength = MFTStart + (getBytes(MFT, entryData + 24, 8) + 1) \* 8;

    cout << "So luong sector trong MFT: " << MFTLength - MFTStart << endl;

    cout << endl;

    // xử lí cây thư mục

    folderTree(MFTLength, MFTStart, disk);

    delete[] MFT;

}

* Test case trong khi chạy chương trình:

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự độngĐ

## **folderTree(unsigned int len\_MFT, unsigned int MFTStart, LPCWSTR disk)**

* Hàm nhận vào biến `len\_MFT` là số nguyên không dấu biểu diễn độ dài của MFT chỉ định vị trí kết thúc của MFT trong các sector. `MFTStart` là số nguyên không dấu biểu diễn sector bắt đầu của MFT. Biến `disk` là `LPCWSTR` con trỏ đến một chuỗi rộng hằng số biểu diễn đường dẫn đến ổ đĩa. Bắt đầu một vòng lặp lặp qua các mục trong MFT, bắt đầu từ 2 và tăng lên 2 mỗi lần lặp, Mỗi mục trong MFT đại diện cho một tệp hoặc thư mục, và chúng thường được tổ chức theo cặp vì một số mục siêu dữ liệu được theo sau bởi các mục dữ liệu tương ứng. Bằng cách tăng lên 2, vòng lặp lặp qua chỉ các entry siêu dữ liệu, thường được lưu trữ ở các chỉ mục chẵn, trong khi bỏ qua các entry dữ liệu, được lưu trữ ở các chỉ mục lẻ. Việc bỏ qua này đảm bảo rằng mỗi lần lặp của vòng lặp chỉ xử lý các entry siêu dữ liệu giúp cải thiện hiệu suất. Sector hiện tại được đọc bằng cách thêm chỉ mục vòng lặp `i` vào sector bắt đầu của MFT. Bộ nhớ được cấp phát động để lưu trữ mục MFT hiện tại, tạo một bộ đệm có kích thước 512 bytes. Cuối cùng đọc sector tương ứng với mục MFT hiện tại từ đĩa và lưu dữ liệu vào bộ đệm `currentEntry`.

    for (int i = 2; i < len\_MFT - MFTStart; i += 2)

    {

        int currentSector = MFTStart + i;

        // Tạo mảng lưu sector

        BYTE \*currentEntry = new BYTE[512];

        // Đọc sector từ ổ đĩa disk từ vị trí currentSector lưu vào currentEntry

        readSector2(disk, currentEntry, currentSector);

* Tiếp theo, kiểm tra xem entry MFT hiện tại có là một tệp không bằng cách chuyển đổi 4 bytes đầu tiên của mục thành một chuỗi và so sánh nó với chuỗi "FILE" để đặt lại biến `check` trước khi vào hàm đọc entry để theo dõi điều kiện liên quan đến việc xử lý tệp. Sau đó lấy `ID` của tệp từ mục MFT hiện tại. Nó đọc 4 bytes bắt đầu từ offset `0x02C` trong mục. Dòng này kiểm tra xem ID tệp trích xuất có lớn hơn 38 cho việc xử lý tiếp theo sẽ được thực hiện cho tệp này. Tiếp theo, in ra hai dòng trống để định dạng và xuất ra ID tệp được trích xuất từ `currentEntry` của MFT. Sau đó, nó truy xuất vị trí bắt đầu của thuộc tính `$INFORMATION` và gọi hàm `readEntryInformation` để xử lý thông tin thuộc tính. Nếu kích thước của thuộc tính `$INFORMATION` được trả về là -1, điều này biểu thị một điều kiện lỗi và bộ nhớ được giải phóng trước khi tiếp tục với vòng lặp tiếp theo.

        // Kiểm tra xem có phải là FILE không

        if (numToString(currentEntry, 0x00, 4) == "FILE")

        {

            check = false;

            int ID = getBytes(currentEntry, 0x02C, 4);

            if (ID > 38) // ID > 38 là các file thực sự tồn tại trong ổ đĩa (có thể là file hoặc thư mục) còn lại là các file ảo (ví dụ: $MFT, $MFTMirr, ...)

            {

                cout << endl;

                cout << "ID File: " << ID << endl;

                int startInfor = getBytes(currentEntry, 0x014, 2);

                int sizeInfor = readEntryInformation(currentEntry, startInfor);

                if (sizeInfor == -1)

                {

                    delete[] currentEntry;

                    continue;

                }

* Sau đó, thuộc tính `$FILE\_NAME` trong `currentEntry` của MFT được tính toán vị trí bắt đầu. Đặt `startName` bằng `sizeInfor + 56`, trong đó `sizeInfor` là kích thước của thuộc tính `$INFORMATION`. Hàm `readEntryFileName` gọi để xử lý thuộc tính `File name`, chuyển đổi `currentEntry` buffer và vị trí bắt đầu tính toán `startName`. Giá trị trả về `sizeName`, đại diện cho kích thước của thuộc tính ` File name`. Tiếp theo tính toán vị trí bắt đầu của thuộc tính `$DATA` dựa trên kích thước của thuộc tính ` File name`. Đặt `startData` bằng `startName + sizeName`. Kiểm tra xem loại thuộc tính `$INFORMATION` có phải là `$OBJECT` không. Nếu đúng, tiến hành xử lý cho `$OBJECT`: truy xuất độ dài bằng cách sử dụng hàm `getBytes` với offset `startData + 4` và kích thước 4. Cập nhật vị trí bắt đầu `startData` dựa trên độ dài của dữ liệu. Gọi hàm `readEntryData` để xử lý dữ liệu, chuyển đổi `currentEntry` buffer và vị trí bắt đầu cập nhật `$INFORMATION`. Nếu loại thuộc tính `$INFORMATION` không phải là `$OBJECT`, tìm kiếm chỉ số sector của thuộc tính `$INFORMATION` bằng cách lặp đi lặp lại việc tăng `startData` lên 4 cho đến khi tìm thấy giá trị 128. Sau đó hàm `readEntryData` để xử lý thuộc tính `$INFORMATION`. Cuối cùng, nó thêm `ID` vào vector `fileID`. Giải phóng bộ nhớ được cấp phát cho `currentEntry` buffer bằng cách sử dụng `delete[] currentEntry`.

                // Đọc thông tin của FILE\_NAME

                int startName = sizeInfor + 56;

                int sizeName = readEntryFileName(currentEntry, startName);

                int startData = startName + sizeName;

                if (getBytes(currentEntry, startData, 4) == 64) // Nếu là $OBJECT

                {

                    int objectLength = getBytes(currentEntry, startData + 4, 4);

                    startData += objectLength;

                    readEntryData(currentEntry, startData);

                }

                else

                {

                    // Tìm sector dấu hiệu của DATA

                    while (getBytes(currentEntry, startData, 4) != 128)

                    {

                        startData += 4;

                    }

                    readEntryData(currentEntry, startData);

                }

                fileID.push\_back(ID);

            }

        }

        delete[] currentEntry;

* Công đoạn tiếp theo, hàm in ra cây thư mục bằng cách tạo định dạng và duyệt qua lần lượt từng phần từ trong `fileID`, xét điều kiện tồn tại của `fileID` và `parentID` để truyền chỉ số vào hàm `printFolderTree`.

    // in ra cây thư mục

    cout << "---------------------------------------------------------\n";

    cout << "\t \t \t CAY THU MUC: " << endl;

    for (int i = 0; i < fileID.size(); ++i)

        if (fileID[i] != -1 && parentID[i] != -1)

            printFolderTree(fileID[i], -1, i);

## **Hàm main:**

* Người dùng được nhắc nhập tên của ổ đĩa mà họ muốn đọc, có thể nhập -1 để thoát khỏi chương trình. Sau khi tên ổ đĩa được đọc và kiểm tra, chương trình sẽ tiến hành định dạng tên ổ đĩa để bao gồm tiền tố \\.\ cùng ký tự hai chấm, sau đó chuyển đổi sang kiểu LPCWSTR. Một mảng động được tạo để lưu trữ dữ liệu của sector với kích thước là 512 byte, và chương trình thử đọc sector 0 từ ổ đĩa đã được chỉ định. Nếu hoạt động đọc thành công, nội dung của sector 0 sẽ được in ra, và sau đó chương trình sẽ đọc và xử lý Khối Tham Số BIOS từ sector 0. Trong trường hợp ổ đĩa không hợp lệ hoặc không thể truy cập, một thông báo lỗi sẽ được hiển thị, và cuối cùng, mảng sector được giải phóng để dọn dẹp bộ nhớ.

int main()

{

    // Người dùng nhập vào tên địa chỉ ổ đĩa cần đọc (ví dụ: C, D, E, F, ...)

    while (1)

    {

        wstring disk\_name;

        cout << "Nhap ten o dia (-1 de thoat): ";

        wcin >> disk\_name;

        if (disk\_name == L"-1")

        {

            cout << "Thoat chuong trinh! Cam on ban da su dung chuong trinh!";

            break;

        }

        disk\_name = L"\\\\.\\" + disk\_name + L":";

        LPCWSTR drive = disk\_name.c\_str();

        // Tạo mảng lưu sector

        BYTE \*sector = new BYTE[512];

        // Đọc sector 0 từ ổ đĩa "drive" và lưu vào "sector"

        int flag = readSector(drive, 0, sector);

        // Kiểm tra xem có đọc được không

        if (flag)

        {

            // In ra sector 0

            printSector(sector);

            // Đọc Bios Parameter Block

            readBPB(sector, drive);

        }

        else

        {

            cout << "Dia chi nhap khong hop le hoac khong ton tai" << endl;

        }

        // Xóa mảng lưu sector

        delete[] sector;

    }

}

* Các trường hợp trong test case bao gồm in các thuộc tính `$INFORMATION`, `$DATA`, `resident` hoặc `non-resident` data, tên file, phần mở rộng, `$FILE\_NAME`, `$FILE\_DATA`, `$STANDARD INFORMATION` và hỗ trợ đọc file txt.

# **Nguồn tham khảo:**

[Hướng dẫn đọc file NTFS](http://ntfs.com/ntfs-partition-boot-sector.htm)

[Tổng quan NTFS](https://en.wikipedia.org/wiki/NTFS)

[Documentation C++ tham khảo](https://www.codeproject.com/Articles/81456/An-NTFS-Parser-Lib)

[Documentation C++ tham khảo (2)](https://cplusplus.com/forum/general/254030/)

[Hỏi đáp NTFS](https://www.codeproject.com/Questions/273746/Given-an-NTFS-File-ID-is-there-any-official-way-to)