|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **STT** | **Phân hệ** | **Chức năng cơ bản** | **Chức năng nâng cao** | **Kết quả** | **Ghi chú** |
|  | URL | Liệt kê danh sách web truy cập | - Giám sát các đường link web đen  - Giám sát các đường link web truyện đen  - Giám sát các đường link web phim online  - Giám sát các đường link cờ bạc, cá độ  - Giám sát các đường link chơi game | Đã xong chức năng cơ bản | Hiệp làm phần nâng cao trên Web |
|  | Liệt kê danh sách link truy cập email |  | Đã xong |  |
|  | Danh sách youtube đã xem |  | Đã xong |  |
|  | Danh sách các từ đã tìm kiếm trên youtube và google | Danh sách các từ khóa tìm kiếm nhạy cảm | Đã xong |  |
|  | Video Audio | Liệt kê danh sách video đã xem | Tổng thời gian xem video theo ngày |  |  |
|  | Liệt kê danh sách file âm thanh đã nghe | Tổng thời gian nghe nhạc theo ngày |  |  |
|  | Tiến trình, phần mềm cài đặt | Danh sách các tiến trình, phần mềm đã chạy | Giám sát các tiến trình game đã chơi và cài đặt | Đã liệt kê được các tiến trình và phần mềm |  |
|  | Các thiết bị ngoại vi | Thời gian cắm các thiết bị ngoại vi và tên các văn bản trao đổi | Giám sát các file chuyển nhận qua USB hoặc vật mang tin | Đã nhận diện được sự kiện cắm, rút USB.  Xuất được thời gian cắm, rút, tổng thời gian cắm USB. | Chưa liệt kê giám sát các file truyền nhận |
|  | Keylogger |  | Keylogger (lưu trữ 1 đến 2 ngày gần nhất theo kiểu first in - first out) | Đã xong chức năng keylog |  |

C:\Users\HuynhDuc\AppData\Roaming\Microsoft\Windows\Recent

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

+ Default cho tài khoản mặc định:

C:\Users\HuynhDuc\AppData\Local\Google\Chrome\User Data\Default

+ Guest Profile cho tài khoản guest

C:\Users\HuynhDuc\AppData\Local\Google\Chrome\User Data\Guest Profile

A screenshot of a computer

Description automatically generated #include <iostream>

#include <fstream>

#include <sqlite3.h>

#include <ctime>

#include <chrono>

void handleError(sqlite3\* db, const char\* message) {

std::cerr << message << ": " << sqlite3\_errmsg(db) << std::endl;

}

void outputToConsoleAndFile(const char\* url, const char\* title, const std::string& timeString, int visitCount, int isHidden, std::ofstream& outFile) {

// Output to console

std::cout << "URL: " << url << ", Title: ";

// Check if the title is not null before using it

if (title != nullptr) {

std::cout << title;

}

else {

std::cout << "N/A";

}

std::cout << ", Visit Time: " << timeString

<< ", Visit Count: " << visitCount << ", Hidden: " << isHidden << std::endl;

// Write to file

outFile << "URL: " << url << ", Title: ";

// Check if the title is not null before using it

if (title != nullptr) {

outFile << title;

}

else {

outFile << "N/A";

}

outFile << ", Visit Time: " << timeString

<< ", Visit Count: " << visitCount << ", Hidden: " << isHidden << std::endl;

}

void convertAndOutputTime(int64\_t visitTimeInt64, const char\* url, const char\* title, int visitCount, int isHidden, std::ofstream& outFile, sqlite3\* db) {

// Convert from microseconds since an unspecified epoch

auto epochStart = std::chrono::system\_clock::from\_time\_t(0);

auto visitTimePoint = epochStart + std::chrono::microseconds(visitTimeInt64);

// Format time string

auto timeString = std::chrono::system\_clock::to\_time\_t(visitTimePoint);

// Use localtime\_s for Windows

struct tm timeInfo;

#ifdef \_WIN32

if (localtime\_s(&timeInfo, &timeString) == 0) {

#else

if (localtime\_r(&timeString, &timeInfo) != nullptr) {

#endif

char buffer[80];

strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", &timeInfo);

outputToConsoleAndFile(url, title, buffer, visitCount, isHidden, outFile);

}

else {

handleError(db, "Error converting time.");

}

}

void collectFirefoxHistory(const char\* profilePath) {

std::string dbPath = std::string(profilePath) + "\\places.sqlite";

sqlite3\* db;

int rc = sqlite3\_open\_v2(dbPath.c\_str(), &db, SQLITE\_OPEN\_READONLY | SQLITE\_OPEN\_SHAREDCACHE, nullptr);

if (rc != SQLITE\_OK) {

handleError(db, "Cannot open Firefox database");

return;

}

const char\* query = "SELECT id, url, title, last\_visit\_date, visit\_count, hidden FROM moz\_places";

sqlite3\_stmt\* stmt;

rc = sqlite3\_prepare\_v2(db, query, -1, &stmt, nullptr);

if (rc != SQLITE\_OK) {

handleError(db, "Cannot prepare statement");

sqlite3\_close(db);

return;

}

// Open the file for writing

std::ofstream outFile("firefox\_output.csv");

while ((rc = sqlite3\_step(stmt)) == SQLITE\_ROW) {

int64\_t id = sqlite3\_column\_int64(stmt, 0);

const char\* url = reinterpret\_cast<const char\*>(sqlite3\_column\_text(stmt, 1));

const char\* title = reinterpret\_cast<const char\*>(sqlite3\_column\_text(stmt, 2));

int64\_t visitTimeInt64 = sqlite3\_column\_int64(stmt, 3);

int visitCount = sqlite3\_column\_int(stmt, 4);

int isHidden = sqlite3\_column\_int(stmt, 5);

convertAndOutputTime(visitTimeInt64, url, title, visitCount, isHidden, outFile, db);

}

// Finalize the statement before closing the file and database

sqlite3\_finalize(stmt);

if (rc != SQLITE\_DONE) {

handleError(db, "Error reading Firefox data");

}

// Close the file

outFile.close();

// Close the database

sqlite3\_close(db);

}

int main() {

const char\* firefoxProfilePath = "C:\\Users\\HuynhDuc\\AppData\\Roaming\\Mozilla\\Firefox\\Profiles\\qn2bivag.default-release";

collectFirefoxHistory(firefoxProfilePath);

// Add code to collect history from other browsers (e.g., Coc Coc, Microsoft Edge) similarly

return 0;

}

CREATE TABLE "moz\_places" (

"id" INTEGER,

"url" LONGVARCHAR,

"title" LONGVARCHAR,

"rev\_host" LONGVARCHAR,

"visit\_count" INTEGER DEFAULT 0,

"hidden" INTEGER NOT NULL DEFAULT 0,

"typed" INTEGER NOT NULL DEFAULT 0,

"frecency" INTEGER NOT NULL DEFAULT -1,

"last\_visit\_date" INTEGER,

"guid" TEXT,

"foreign\_count" INTEGER NOT NULL DEFAULT 0,

"url\_hash" INTEGER NOT NULL DEFAULT 0,

"description" TEXT,

"preview\_image\_url" TEXT,

"site\_name" TEXT,

"origin\_id" INTEGER,

"recalc\_frecency" INTEGER NOT NULL DEFAULT 0,

"alt\_frecency" INTEGER,

"recalc\_alt\_frecency" INTEGER NOT NULL DEFAULT 0,

FOREIGN KEY("origin\_id") REFERENCES "moz\_origins"("id"),

PRIMARY KEY("id")

);

//==================================================================================

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <Windows.h>

#include <Knownfolders.h>

#include <Shlobj.h>

// Link against the Shell32.lib library

#pragma comment(lib, "Shell32.lib")

#include <locale>

#include <codecvt>

#include "sqlite3.h"

#include <ctime>

#include "chromeUrl.h"

wstring getChromeHistoryPath() {

// Get the local app data folder

PWSTR localAppData;

if (SHGetKnownFolderPath(FOLDERID\_LocalAppData, 0, nullptr, &localAppData) != S\_OK) {

std::cerr << "Error getting Local AppData folder." << std::endl;

return L"";

}

std::wstring chromeProfilePath = std::wstring(localAppData) + L"\\Google\\Chrome\\User Data\\Default\\History";

CoTaskMemFree(localAppData);

return chromeProfilePath;

}

void processChromeHistory() {

wstring chromeHistoryPath = getChromeHistoryPath();

wstring\_convert<codecvt\_utf8<wchar\_t>> converter;

string HistoryPathChrome = converter.to\_bytes(chromeHistoryPath);

const char\* outputFilePath = "outputChrome.txt";

sqlite3\* db;

int rc = sqlite3\_open\_v2(HistoryPathChrome.c\_str(), &db, SQLITE\_OPEN\_READONLY | SQLITE\_OPEN\_SHAREDCACHE, nullptr);

if (rc != SQLITE\_OK) {

std::cerr << "Cannot open database: " << sqlite3\_errmsg(db) << std::endl;

return;

}

const char\* query = "SELECT url, last\_visit\_time FROM urls";

sqlite3\_stmt\* stmt;

rc = sqlite3\_prepare\_v2(db, query, -1, &stmt, nullptr);

if (rc != SQLITE\_OK) {

std::cerr << "Cannot prepare statement: " << sqlite3\_errmsg(db) << std::endl;

sqlite3\_close(db);

return;

}

std::ofstream outFile(outputFilePath);

while ((rc = sqlite3\_step(stmt)) == SQLITE\_ROW) {

const char\* url = reinterpret\_cast<const char\*>(sqlite3\_column\_text(stmt, 0));

int64\_t visitTimeInt64 = sqlite3\_column\_int64(stmt, 1);

time\_t visitTime = static\_cast<time\_t>(visitTimeInt64 / 1000000 - 11644473600LL);

char buffer[80];

struct tm timeInfo;

if (localtime\_s(&timeInfo, &visitTime) == 0) {

strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", &timeInfo);

std::cout << "URL: " << url << ", Visit Time: " << buffer << std::endl;

outFile << "URL: " << url << ", Visit Time: " << buffer << std::endl;

}

else {

std::cerr << "Error converting time." << std::endl;

}

}

if (rc != SQLITE\_DONE) {

std::cerr << "Error reading data: " << sqlite3\_errmsg(db) << std::endl;

}

outFile.close();

sqlite3\_finalize(stmt);

sqlite3\_close(db);

}

**CREATE TABLE "urls"** (

"id" INTEGER,

"url" LONGVARCHAR,

"title" LONGVARCHAR,

"visit\_count" INTEGER NOT NULL DEFAULT 0,

"typed\_count" INTEGER NOT NULL DEFAULT 0,

"last\_visit\_time" INTEGER NOT NULL,

"hidden" INTEGER NOT NULL DEFAULT 0,

PRIMARY KEY("id" AUTOINCREMENT)

);

USB detect

usbmon.h:

#pragma once

#include <Windows.h>

#ifndef USBMON\_H

#define USBMON\_H

class DeviceNotification {

public:

DeviceNotification();

~DeviceNotification();

void run\_from\_thread();

private:

HWND hWnd;

};

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

#endif // USBMON\_H

Usbmon.cpp:  
#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <Dbt.h>

#include <chrono>

#include <ctime>

#include "usbMon.h"

using namespace std;

using namespace std::chrono;

high\_resolution\_clock::time\_point plugInTime; // Define plugInTime and plugOutTime globally

high\_resolution\_clock::time\_point plugOutTime;

DeviceNotification::DeviceNotification() {

// Register window class

WNDCLASS wc = {};

wc.lpfnWndProc = WndProc;

wc.hInstance = GetModuleHandle(nullptr);

wc.lpszClassName = L"USBEventWindowClass";

RegisterClass(&wc);

// Create a window

hWnd = CreateWindow(wc.lpszClassName, nullptr, 0, 0, 0, 0, 0, nullptr, nullptr, wc.hInstance, nullptr);

if (!hWnd) {

cerr << "Failed to create window. Error: " << GetLastError() << endl;

}

else {

cout << "Window created successfully" << endl;

}

// Register for USB device change notifications

DEV\_BROADCAST\_DEVICEINTERFACE dbdi = {};

dbdi.dbcc\_size = sizeof(DEV\_BROADCAST\_DEVICEINTERFACE);

dbdi.dbcc\_devicetype = DBT\_DEVTYP\_DEVICEINTERFACE;

RegisterDeviceNotification(hWnd, &dbdi, DEVICE\_NOTIFY\_WINDOW\_HANDLE);

}

DeviceNotification::~DeviceNotification() {

// Clean up

UnregisterDeviceNotification(hWnd);

}

void DeviceNotification::run\_from\_thread() {

MSG msg;

while (GetMessage(&msg, nullptr, 0, 0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

void printTime(const string& label, const high\_resolution\_clock::time\_point& timePoint) {

auto system\_now = system\_clock::now();

auto duration\_since\_epoch = timePoint.time\_since\_epoch() - high\_resolution\_clock::now().time\_since\_epoch() + system\_now.time\_since\_epoch();

auto millis = duration\_cast<milliseconds>(duration\_since\_epoch).count() % 1000;

time\_t tt = system\_clock::to\_time\_t(system\_clock::time\_point(duration\_cast<system\_clock::duration>(duration\_since\_epoch)));

tm tm\_time;

localtime\_s(&tm\_time, &tt);

cout << label << ": " << put\_time(&tm\_time, "%Y-%m-%d %H:%M:%S") << "." << setfill('0') << setw(3) << millis << " milliseconds" << endl;

}

void convertAndOutputTime(const steady\_clock::time\_point& timePoint) {

// Convert from microseconds since an unspecified epoch

auto epochStart = system\_clock::from\_time\_t(0);

auto visitTimePoint = epochStart + duration\_cast<system\_clock::duration>(timePoint.time\_since\_epoch());

// Format time string

auto timeString = system\_clock::to\_time\_t(visitTimePoint);

struct tm timeInfo;

localtime\_s(&timeInfo, &timeString);

char buffer[80];

strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", &timeInfo);

std::cout << "Time: " << buffer << std::endl;

}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {

PDEV\_BROADCAST\_HDR lpdb = nullptr;

switch (message) {

case WM\_CREATE:

cout << "WM\_CREATE received" << endl;

break;

case WM\_DEVICECHANGE:

cout << "WM\_DEVICECHANGE received. wParam: " << wParam << endl;

lpdb = reinterpret\_cast<PDEV\_BROADCAST\_HDR>(lParam);

switch (wParam) {

case DBT\_DEVICEARRIVAL:

cout << "DBT\_DEVICEARRIVAL received" << endl;

if (lpdb->dbch\_devicetype == DBT\_DEVTYP\_VOLUME) {

plugInTime = high\_resolution\_clock::now(); // Record plug in time

PDEV\_BROADCAST\_VOLUME lpdbv = reinterpret\_cast<PDEV\_BROADCAST\_VOLUME>(lpdb);

DWORD unitMask = lpdbv->dbcv\_unitmask;

for (int drive = 0; drive < 26; ++drive) {

if (unitMask & (1 << drive)) {

cout << "USB Device Inserted. Drive Letter: " << char('A' + drive) << endl;

}

}

}

break;

case DBT\_DEVICEREMOVECOMPLETE:

cout << "DBT\_DEVICEREMOVECOMPLETE received" << endl;

if (lpdb->dbch\_devicetype == DBT\_DEVTYP\_VOLUME) {

plugOutTime = high\_resolution\_clock::now(); // Record plug out time

PDEV\_BROADCAST\_VOLUME lpdbv = reinterpret\_cast<PDEV\_BROADCAST\_VOLUME>(lpdb);

cout << "USB Device Removed. Drive Letter: " << char('A' + lpdbv->dbcv\_unitmask) << endl;

// Print plug-in and plug-out times

printTime("Plug-in Time", plugInTime);

printTime("Plug-out Time", plugOutTime);

// Calculate and print duration

auto duration = duration\_cast<milliseconds>(plugOutTime - plugInTime).count();

cout << "Duration: " << duration << " milliseconds" << endl;

}

break;

}

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

//////////===========================================================================

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <Dbt.h>

#include <chrono>

#include <ctime>

#include "usbMon.h"

using namespace std;

using namespace std::chrono;

high\_resolution\_clock::time\_point plugInTime; // Define plugInTime and plugOutTime globally

high\_resolution\_clock::time\_point plugOutTime;

DeviceNotification::DeviceNotification() {

// Register window class

WNDCLASS wc = {};

wc.lpfnWndProc = WndProc;

wc.hInstance = GetModuleHandle(nullptr);

wc.lpszClassName = L"USBEventWindowClass";

RegisterClass(&wc);

// Create a window

hWnd = CreateWindow(wc.lpszClassName, nullptr, 0, 0, 0, 0, 0, nullptr, nullptr, wc.hInstance, nullptr);

if (!hWnd) {

cerr << "Failed to create window. Error: " << GetLastError() << endl;

}

else {

cout << "Window created successfully" << endl;

}

// Register for USB device change notifications

DEV\_BROADCAST\_DEVICEINTERFACE dbdi = {};

dbdi.dbcc\_size = sizeof(DEV\_BROADCAST\_DEVICEINTERFACE);

dbdi.dbcc\_devicetype = DBT\_DEVTYP\_DEVICEINTERFACE;

RegisterDeviceNotification(hWnd, &dbdi, DEVICE\_NOTIFY\_WINDOW\_HANDLE);

}

DeviceNotification::~DeviceNotification() {

// Clean up

UnregisterDeviceNotification(hWnd);

}

void DeviceNotification::run\_from\_thread() {

MSG msg;

while (GetMessage(&msg, nullptr, 0, 0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

void printTime(const string& label, const high\_resolution\_clock::time\_point& timePoint) {

auto system\_now = system\_clock::now();

auto duration\_since\_epoch = timePoint.time\_since\_epoch() - high\_resolution\_clock::now().time\_since\_epoch() + system\_now.time\_since\_epoch();

auto millis = duration\_cast<milliseconds>(duration\_since\_epoch).count() % 1000;

time\_t tt = system\_clock::to\_time\_t(system\_clock::time\_point(duration\_cast<system\_clock::duration>(duration\_since\_epoch)));

tm tm\_time;

localtime\_s(&tm\_time, &tt);

cout << label << ": " << put\_time(&tm\_time, "%Y-%m-%d %H:%M:%S") << "." << setfill('0') << setw(3) << millis << " milliseconds" << endl;

}

void convertAndOutputTime(const steady\_clock::time\_point& timePoint) {

// Convert from microseconds since an unspecified epoch

auto epochStart = system\_clock::from\_time\_t(0);

auto visitTimePoint = epochStart + duration\_cast<system\_clock::duration>(timePoint.time\_since\_epoch());

// Format time string

auto timeString = system\_clock::to\_time\_t(visitTimePoint);

struct tm timeInfo;

localtime\_s(&timeInfo, &timeString);

char buffer[80];

strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", &timeInfo);

std::cout << "Time: " << buffer << std::endl;

}

void MonitorDirectoryChanges(const wstring& directoryPath) {

// Create a handle to the directory to monitor

HANDLE hDir = CreateFile(

directoryPath.c\_str(),

FILE\_LIST\_DIRECTORY,

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

NULL,

OPEN\_EXISTING,

FILE\_FLAG\_BACKUP\_SEMANTICS,

NULL

);

if (hDir == INVALID\_HANDLE\_VALUE) {

cerr << "Failed to open directory: " << GetLastError() << endl;

return;

}

char buffer[1024];

DWORD bytesReturned;

FILE\_NOTIFY\_INFORMATION\* fileInfo;

BOOL result;

while (true) {

result = ReadDirectoryChangesW(

hDir,

buffer,

sizeof(buffer),

TRUE,

FILE\_NOTIFY\_CHANGE\_FILE\_NAME | FILE\_NOTIFY\_CHANGE\_DIR\_NAME | FILE\_NOTIFY\_CHANGE\_ATTRIBUTES |

FILE\_NOTIFY\_CHANGE\_SIZE | FILE\_NOTIFY\_CHANGE\_LAST\_WRITE | FILE\_NOTIFY\_CHANGE\_LAST\_ACCESS |

FILE\_NOTIFY\_CHANGE\_CREATION | FILE\_NOTIFY\_CHANGE\_SECURITY,

&bytesReturned,

NULL,

NULL

);

if (!result) {

cerr << "ReadDirectoryChangesW failed: " << GetLastError() << endl;

CloseHandle(hDir);

return;

}

fileInfo = reinterpret\_cast<FILE\_NOTIFY\_INFORMATION\*>(buffer);

while (true) {

if (fileInfo->Action == FILE\_ACTION\_ADDED) {

// Handle newly added file

// Print the file name or perform any other desired action

wcout << L"New file added: " << fileInfo->FileName << endl;

}

if (fileInfo->NextEntryOffset == 0) {

break;

}

fileInfo = reinterpret\_cast<FILE\_NOTIFY\_INFORMATION\*>(reinterpret\_cast<char\*>(fileInfo) + fileInfo->NextEntryOffset);

}

}

CloseHandle(hDir);

}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {

PDEV\_BROADCAST\_HDR lpdb = nullptr;

switch (message) {

case WM\_CREATE:

cout << "WM\_CREATE received" << endl;

break;

case WM\_DEVICECHANGE:

cout << "WM\_DEVICECHANGE received. wParam: " << wParam << endl;

lpdb = reinterpret\_cast<PDEV\_BROADCAST\_HDR>(lParam);

switch (wParam) {

case DBT\_DEVICEARRIVAL:

cout << "DBT\_DEVICEARRIVAL received" << endl;

if (lpdb->dbch\_devicetype == DBT\_DEVTYP\_VOLUME) {

plugInTime = high\_resolution\_clock::now(); // Record plug in time

PDEV\_BROADCAST\_VOLUME lpdbv = reinterpret\_cast<PDEV\_BROADCAST\_VOLUME>(lpdb);

DWORD unitMask = lpdbv->dbcv\_unitmask;

for (int drive = 0; drive < 26; ++drive) {

if (unitMask & (1 << drive)) {

cout << "USB Device Inserted. Drive Letter: " << char('A' + drive) << endl;

// Construct USB drive path

wstring usbDrivePath = L"";

usbDrivePath += wchar\_t('A' + drive);

usbDrivePath += L":\\";

// Monitor changes in the destination directory

MonitorDirectoryChanges(usbDrivePath);

}

}

}

break;

case DBT\_DEVICEREMOVECOMPLETE:

cout << "DBT\_DEVICEREMOVECOMPLETE received" << endl;

if (lpdb->dbch\_devicetype == DBT\_DEVTYP\_VOLUME) {

plugOutTime = high\_resolution\_clock::now(); // Record plug out time

PDEV\_BROADCAST\_VOLUME lpdbv = reinterpret\_cast<PDEV\_BROADCAST\_VOLUME>(lpdb);

cout << "USB Device Removed. Drive Letter: " << char('A' + lpdbv->dbcv\_unitmask) << endl;

// Print plug-in and plug-out times

printTime("Plug-in Time", plugInTime);

printTime("Plug-out Time", plugOutTime);

// Calculate and print duration

auto duration = duration\_cast<milliseconds>(plugOutTime - plugInTime).count();

cout << "Duration: " << duration << " milliseconds" << endl;

}

break;

}

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

///////////////////////////////////////////////////////////////////

#include <iostream>

#include <Windows.h>

#include <vector>

#include <string>

// Function to read registry values under a specified key

std::vector<std::wstring> ReadRegistryValues(const std::wstring& keyPath) {

std::vector<std::wstring> values;

HKEY hKey;

if (RegOpenKeyExW(HKEY\_CURRENT\_USER, keyPath.c\_str(), 0, KEY\_READ, &hKey) == ERROR\_SUCCESS) {

wchar\_t valueName[MAX\_PATH];

DWORD valueNameSize = MAX\_PATH;

DWORD valueType;

BYTE valueData[MAX\_PATH];

DWORD valueDataSize = MAX\_PATH;

DWORD index = 0;

while (RegEnumValueW(hKey, index, valueName, &valueNameSize, NULL, &valueType, valueData, &valueDataSize) == ERROR\_SUCCESS) {

if (valueType == REG\_SZ) {

values.push\_back(std::wstring(valueName));

}

// Reset sizes for next iteration

valueNameSize = MAX\_PATH;

valueDataSize = MAX\_PATH;

index++;

}

RegCloseKey(hKey);

}

return values;

}

int main() {

// Đường dẫn trong registry

std::wstring keyPath = L"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\RecentDocs\\.mp4";

// Đọc các giá trị từ registry

std::vector<std::wstring> recentFiles = ReadRegistryValues(keyPath);

// Hiển thị danh sách các tệp đã mở gần đây

if (recentFiles.empty()) {

std::wcout << L"No recent opened files found." << std::endl;

}

else {

std::wcout << L"Recent opened files:" << std::endl;

for (const auto& file : recentFiles) {

std::wcout << file << std::endl;

}

}

return 0;

}

///////////////////////////////////////////////////////////////////

#include <iostream>

#include <Windows.h>

void PrintRecentFiles() {

const wchar\_t\* regPath = L"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\RecentDocs\\.mp4";

HKEY hKey;

LONG result = RegOpenKeyExW(HKEY\_CURRENT\_USER, regPath, 0, KEY\_READ, &hKey);

if (result != ERROR\_SUCCESS) {

LPWSTR errorMessage = NULL;

FormatMessageW(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM | FORMAT\_MESSAGE\_IGNORE\_INSERTS,

NULL, result, MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPWSTR)&errorMessage, 0, NULL);

if (errorMessage)

{

wprintf(L"Failed to open registry key: %s\n", errorMessage);

LocalFree(errorMessage);

}

return;

}

wchar\_t valueName[256];

DWORD valueNameSize = 256;

BYTE data[1024];

DWORD dataSize = 1024;

DWORD valueType;

DWORD index = 0;

while (RegEnumValueW(hKey, index++, valueName, &valueNameSize, NULL, &valueType, data, &dataSize) == ERROR\_SUCCESS)

{

wprintf(L"%s\n", (wchar\_t\*)data);

valueNameSize = 256;

dataSize = 1024;

}

RegCloseKey(hKey);

}

int main() {

PrintRecentFiles();

return 0;

}

///////////////////////////////////////////////////////////////////

#include <Windows.h>

#include <iostream>

#include <string>

void EnumerateInstalledSoftware(HKEY hKey, const std::wstring& subkey) {

HKEY hSubKey;

if (RegOpenKeyExW(hKey, subkey.c\_str(), 0, KEY\_READ, &hSubKey) == ERROR\_SUCCESS) {

DWORD index = 0;

WCHAR achKey[255];

DWORD cbName = sizeof(achKey);

while (RegEnumKeyExW(hSubKey, index++, achKey, &cbName, NULL, NULL, NULL, NULL) == ERROR\_SUCCESS) {

HKEY hSoftwareKey;

if (RegOpenKeyExW(hSubKey, achKey, 0, KEY\_READ, &hSoftwareKey) == ERROR\_SUCCESS) {

WCHAR achValue[1024];

DWORD cbValue = sizeof(achValue);

if (RegQueryValueExW(hSoftwareKey, L"DisplayName", NULL, NULL, (LPBYTE)achValue, &cbValue) == ERROR\_SUCCESS) {

std::wcout << achValue << std::endl;

}

RegCloseKey(hSoftwareKey);

}

cbName = sizeof(achKey);

}

RegCloseKey(hSubKey);

}

}

int main() {

std::wcout << L"Installed software:" << std::endl;

EnumerateInstalledSoftware(HKEY\_LOCAL\_MACHINE, L"Software\\Microsoft\\Windows\\CurrentVersion\\Uninstall");

EnumerateInstalledSoftware(HKEY\_LOCAL\_MACHINE, L"Software\\Wow6432Node\\Microsoft\\Windows\\CurrentVersion\\Uninstall"); // For 32-bit software on 64-bit Windows

return 0;

}

///////////////////////////////////////////////////////////////////