Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и

системное программирование

ОТЧЁТ

к лабораторной работе №3

на тему

Управление памятью и вводом-выводом, расширенные возможности ввода-вывода Windows. Функции API подсистемы памяти Win 32. Организация и контроль асинхронных операций ввода-вывода. Отображение файлов в память.

Выполнил: студент группы 153503

Киселёва Елизавета Андреевна

Проверил: Гриценко Никита Юрьевич

Минск 2023

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc146728376)

[2 Краткие теоретические сведения 4](#_Toc146728377)

[3 Результаты выполнения лабораторной работы 7](#_Toc146728378)

[Выводы 7](#_Toc146728380)

[Список использованных исчтоников 8](#_Toc146728381)

[Приложение А (обязательное) Листинг кода 13](#_Toc146728382)

## 1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения лабораторной работы является создание приложения с управлением памятью и вводом-выводом, использованием расширенных возможностей ввода-вывоа *Windows*, функций *API* подсистемы памяти *Win*32, организация и контроль асинхронных операций ввода-вывода, отображение файлов в память. Для выполнения цели лабораторной работы будет создано приложение для сжатия и распаковки файлов, использующее *API* для работы.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## 2.1 Управление памятью в *Win*32 *API,* функции API подсистемы памяти Win32

В *Win*32 *API* существует концепция кучи (*heap*), которая представляет собой область памяти, из которой можно выделять блоки переменного размера. *HeapAlloc* используется для выделения блока памяти из кучи, а *HeapFree* – для освобождения выделенной памяти.

*GlobalAlloc* и *GlobalFree* также выделяют и освобождают память, но из глобальной кучи. Они были представлены в более ранних версиях *Windows* и могут использоваться для совместимости.

Разделяемая память (*Shared* *Memory*) – это механизм в операционных системах, который позволяет нескольким процессам разделять общую область памяти, что обеспечивает им механизм взаимодействия. Разделяемую память можно использовать для обмена данными между процессами.

Стандартные функции *memcpy* и *memset* в *C/C*++ предназначены для копирования и заполнения блоков памяти соответственно. Они могут использоваться для манипуляции данными в памяти. *Memcpy* просто копирует, *memmove* может обрабатывать перекрывающиеся области, а *memset* устанавливает значения байтов блока. *RtlCopyMemory* – эффективный вариант memcpy, предоставленный *API* системы (*Run*-*Time* *Library*). Он предназначен для копирования блоков памяти и может быть использован вместо *memcpy*.

*VirtualAlloc*, *VirtualFree* позволяют выделять и освобождать виртуальную память. Это основной механизм для работы с виртуальной памятью в *Windows*, включая выделение резервной памяти, изменение защиты страниц и другие операции.

*MEMORY*\_*BASIC*\_*INFORMATION* – структура, предоставляющая информацию о регионе виртуальной памяти. Используется с функцией *VirtualQuery*(предоставляет информацию о регионе виртуальной памяти, позволяет программе получать сведения о состоянии и атрибутах блока виртуальной памяти по указанному адресу.). [1]

## 2.2 Ввод-вывод в *Windows*

Ввод-вывод в операционной системе *Windows* включает в себя множество методов обмена данными между приложением и внешними устройствами или другими частями программы. Он охватывает работу с файлами, устройствами ввода-вывода, сетевыми соединениями и другими ресурсами.

Ввод-вывод в Windows реализуется через механизмы, такие как *CreateFile*, *ReadFile*, *WriteFile*, и другие. Эти функции предоставляют интерфейс для работы с файлами, а также другими объектами, такими как устройства. [2]

## 2.3 Асинхронные операции ввода-вывода

Для обеспечения эффективного использования ресурсов и улучшения производительности, в *Win*32 *API* реализованы асинхронные операции ввода-вывода. Функции, такие как *ReadFileEx* и *WriteFileEx*, позволяют выполнять операции ввода-вывода в фоновом режиме, не блокируя выполнение программы.

Асинхронные операции ввода-вывода в Win32 API предоставляют механизм для выполнения операций ввода-вывода без блокировки основного потока выполнения программы. Это особенно полезно в многозадачных приложениях и там, где производительность является важным фактором.

*I/O Completion Ports* (Порты завершения *I/O* – это объекты, которые предоставляют механизм для отслеживания завершения асинхронных операций ввода-вывода. Они позволяют приложению эффективно обрабатывать завершение множества операций одновременно.

Для запуска асинхронных операций ввода-вывода используется структура *OVERLAPPED*. Она содержит информацию о состоянии операции, а также может включать указатель на функцию обратного вызова, которая будет вызвана по завершении операции.

К функциям поддержки асинхронных операций относится множество функций *Win*32 *API*, таких как *ReadFile*, *WriteFile*, *DeviceIoControl*. Они могут быть использованы для асинхронного чтения или записи данных.

Для ожидания завершения асинхронных операций используются события, являющиеся механизмом сигнализации. После завершения операции событие сигнализирует о готовности данных для обработки.

Асинхронные операции могут использовать *callback*-функции, которые вызываются после завершения операции. Это предоставляет удобный способ обработки результата операции. [3]

## 2.4 Отображение файлов в память

Отображение файла в память – это механизм, предоставляемый операционной системой, который позволяет приложению работать с содержимым файла, как если бы это была область системной памяти, а не непосредственно файл на диске.

*CreateFileMapping* создает отображение файла в память – файл синхронизации (*File* *Mapping* *Object*), а *MapViewOfFile* позволяет процессу получить доступ к этому отображению. Это дает возможность нескольким процессам обмениваться информацией через общую память.

Отображение файлов в память предоставляет эффективный способ работать с содержимым файла, обеспечивая прямой доступ к данным. Функции *API*, такие как *CreateFileMapping* и *MapViewOfFile*, позволяют отобразить содержимое файла в адресное пространство процесса, что упрощает чтение и запись данных. Этот подход особенно полезен при работе с большими файлами, так как позволяет избежать необходимости частого чтения или записи в файл. [3]

## 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы было разработано приложение для сжатия и распаковки файлов. Приложение позволяет пользователю выбрать файл, который тот хочет сжать, сохранить его сжатый вариант под новым или уже имеющимся именем, а также распаковать уже имеющийся сжатый файл и сохранить распакованный под новым именем. Также пользователь может видеть название приложения и его иконку (Рисунок 1).

****

Рисунок 1 – Главное окно

Пользователь может выбрать файл для сжатия (Рисунок 2).

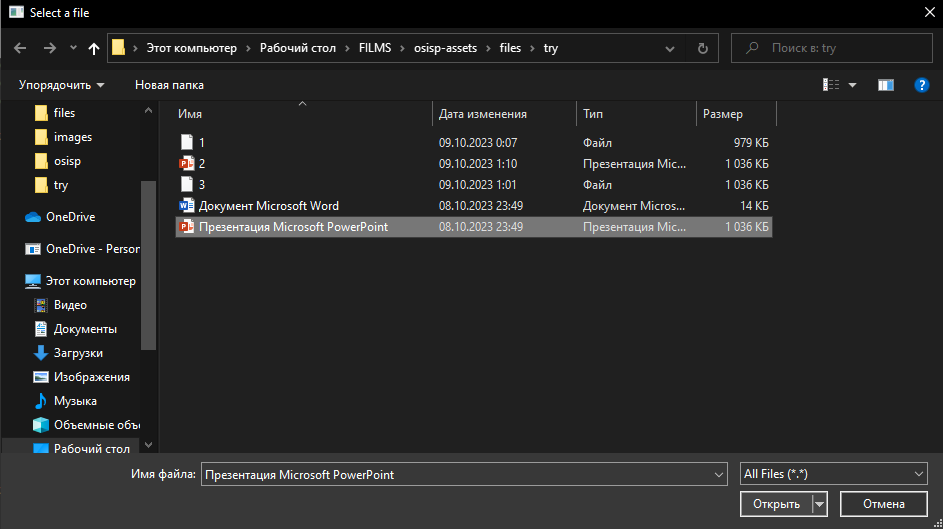


Рисунок 2 – Выбор файла для сжатия

Пользователь может сохранить сжатый файл (Рисунок 3).

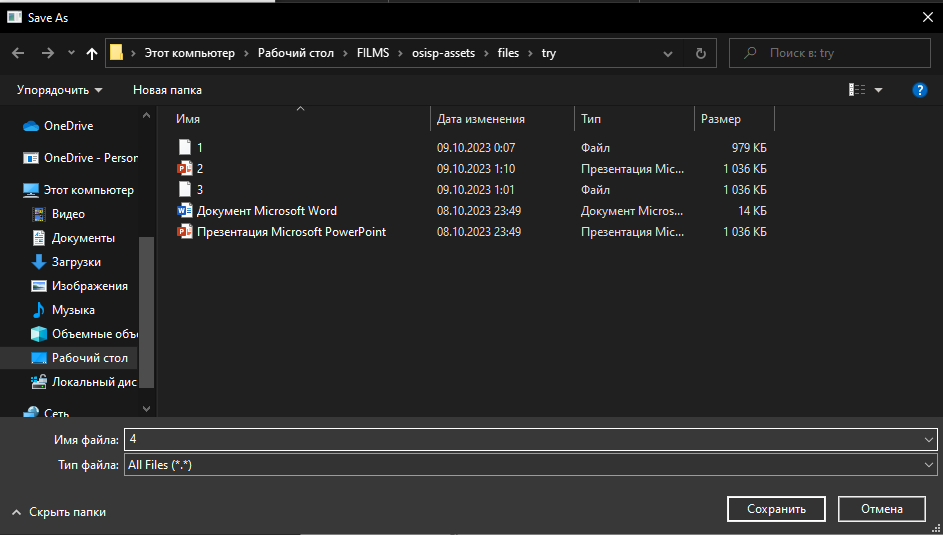


Рисунок 3 – Сохранение сжатого файла

Пользователь видит информацию об успешном завершении сжатия и о том, какой размер у входного файла, какой – у выходного (Рисунок 4).

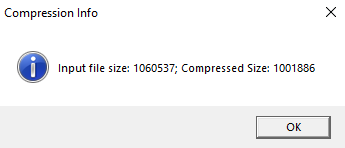


Рисунок 4 – Информация о сжатии

Пользователь видит информацию о том, сколько времени ушло на сжатие файла (Рисунок 5).

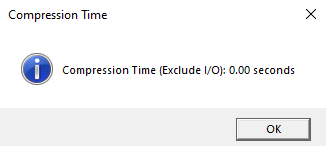


Рисунок 5 – Информация о времени сжатия

Пользователь может выбрать файл для распаковки (Рисунок 6).

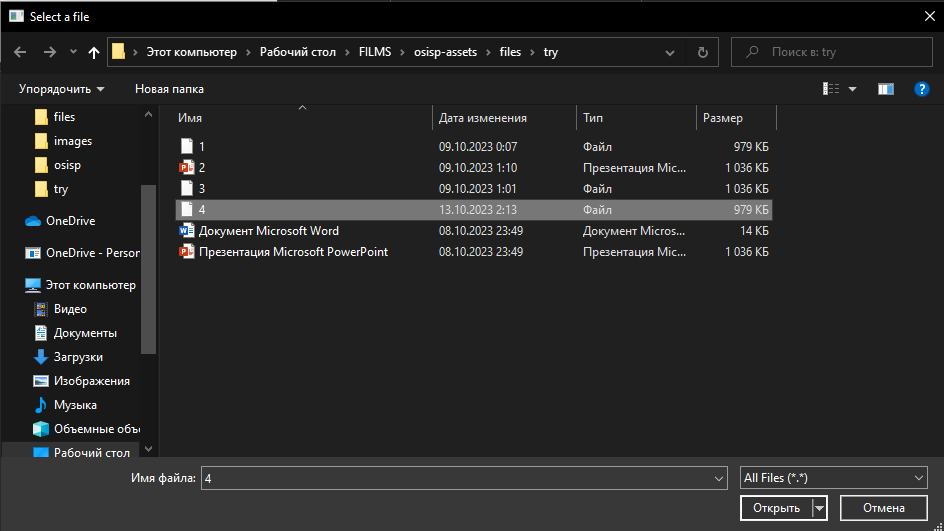


Рисунок 6 – Выбор файла для распаковки

Пользователь может сохранить распакованный файл (Рисунок 7).

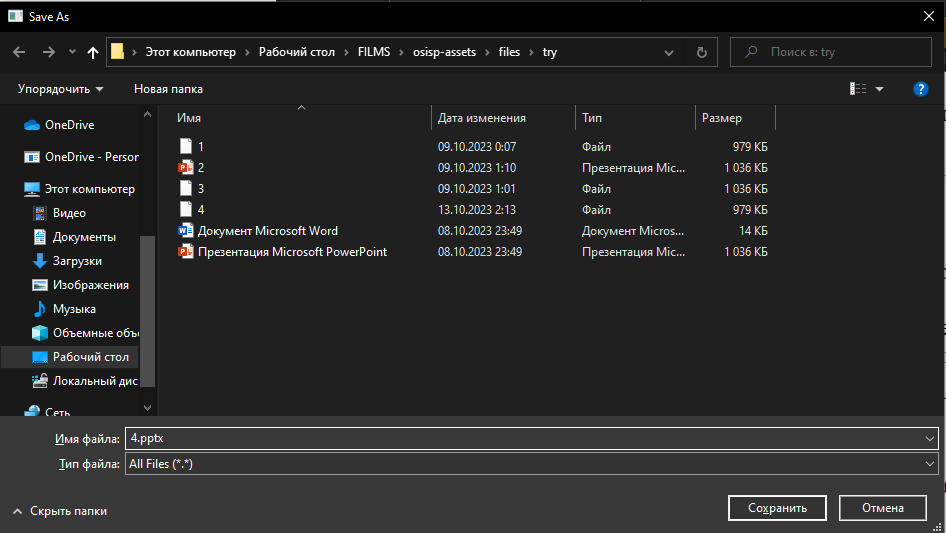


Рисунок 7 – Сохранение распакованного файла

Пользователь видит информацию об успешном завершении распаковки и о том, какой размер у входного файла, какой – у выходного (Рисунок 8).

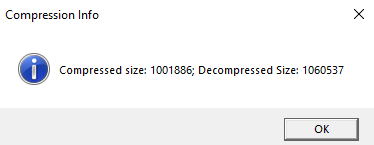


Рисунок 8 – Информация о распаковке

Пользователь видит информацию о том, сколько времени ушло на распаковку файла (Рисунок 9).

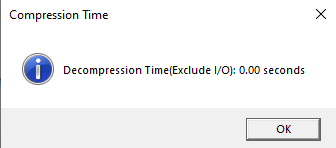


Рисунок 9 – Информация о времени распаковки

## ВЫВОДЫ

В ходе выполнения данной лабораторной работы было разработано приложение для сжатия и распаковки файлов, использующее *API* для работы. Проект включает в себя создание графического окна приложения, обработку оконных сообщений, пользовательский интерфейс с кнопками, работу с файлами, использование асинхронных операций ввода-вывода, управление памятью в *Win*32 *API*, функции *API* подсистемы памяти *Win*32, ввода-вывода в *Windows*. Результатом стало рабочее приложение для сжатия и распаковки файлов, имеющее простое управление даже для начинающего пользователя и способное сжимать файлы, превращая их в файлы меньшего размера, а после распаковывать их, возвращая в исходное состояние.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Архитектура памяти в *Win*32 *API*[Электронный ресурс]. – Режим доступа: https://studfile.net/preview/1175379/page:4/ – Дата доступа 10.10.2023

[2] Щупак Ю. *Win*32 *API*. Разработка приложений для *Windows*. – СПб: Питер, 2008. – 592 с.: ип.

[3] *Microsoft* *Learn*[Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/fileio/i-o-concepts – Дата доступа 08.10.2023

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг кода

**main.cpp**

#pragma once

#include <windows.h>

#include <iostream>

#include <string>

#include <compressapi.h>

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

const wchar\_t g\_szClassName[] = L"FileCompressionAppClass";

const int IDC\_COMPRESS\_BUTTON = 101;

const int IDC\_DECOMPRESS\_BUTTON = 102;

std::wstring GetOpenFileName(HWND hwnd) {

OPENFILENAME ofn;

wchar\_t szFileName[MAX\_PATH] = L"";

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn);

ofn.hwndOwner = hwnd;

ofn.lpstrFilter = L"All Files (\*.\*)\0\*.\*\0";

ofn.lpstrFile = szFileName;

ofn.nMaxFile = sizeof(szFileName) / sizeof(\*szFileName);

ofn.lpstrTitle = L"Select a file";

ofn.Flags = OFN\_DONTADDTORECENT | OFN\_FILEMUSTEXIST;

if (GetOpenFileName(&ofn)) {

return szFileName;

}

return L"";

}

std::wstring GetSaveFileName(HWND hwnd) {

OPENFILENAME ofn;

wchar\_t szFileName[MAX\_PATH] = L"";

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn);

ofn.hwndOwner = hwnd;

ofn.lpstrFilter = L"All Files (\*.\*)\0\*.\*\0";

ofn.lpstrFile = szFileName;

ofn.nMaxFile = sizeof(szFileName) / sizeof(\*szFileName);

ofn.lpstrTitle = L"Save As";

ofn.Flags = OFN\_DONTADDTORECENT | OFN\_OVERWRITEPROMPT;

if (GetSaveFileName(&ofn)) {

return szFileName;

}

return L"";

}

void CompressFile(HWND hwnd) {

std::wstring sourcePath = GetOpenFileName(hwnd);

if (sourcePath.empty()) {

return; // Пользователь отменил выбор файла

}

std::wstring destPath = GetSaveFileName(hwnd);

if (destPath.empty()) {

return; // Пользователь отменил выбор места сохранения файла

}

COMPRESSOR\_HANDLE Compressor = NULL;

PBYTE CompressedBuffer = NULL;

PBYTE InputBuffer = NULL;

HANDLE InputFile = INVALID\_HANDLE\_VALUE;

HANDLE CompressedFile = INVALID\_HANDLE\_VALUE;

BOOL DeleteTargetFile = TRUE;

BOOL Success;

SIZE\_T CompressedDataSize, CompressedBufferSize;

DWORD InputFileSize, ByteRead, ByteWritten;

LARGE\_INTEGER FileSize;

ULONGLONG StartTime, EndTime;

double TimeDuration;

// Открыть входной файл для чтения, только существующий файл.

InputFile = CreateFile(

sourcePath.c\_str(),

GENERIC\_READ,

FILE\_SHARE\_READ, // Разрешено совместное чтение

NULL, // Атрибуты безопасности, стандартная безопасность

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

NULL); // Атрибуты шаблона не используются

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

MessageBox(hwnd, L"Cannot open source file.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

// Получить размер входного файла.

Success = GetFileSizeEx(InputFile, &FileSize);

if ((!Success) || (FileSize.QuadPart > 0xFFFFFFFF)) {

MessageBox(hwnd, L"Cannot get input file size or file is larger than 4GB.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

InputFileSize = FileSize.LowPart;

// Выделить память для содержимого файла.

InputBuffer = (PBYTE)malloc(InputFileSize);

if (!InputBuffer) {

MessageBox(hwnd, L"Cannot allocate memory for uncompressed buffer.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

// Прочесть входной файл.

Success = ReadFile(InputFile, InputBuffer, InputFileSize, &ByteRead, NULL);

if ((!Success) || (ByteRead != InputFileSize)) {

MessageBox(hwnd, L"Cannot read from source file.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

// Открыть файл для записи, если существует - перезаписать

CompressedFile = CreateFile(

destPath.c\_str(),

GENERIC\_WRITE | DELETE, // Открыть для чтения; удалить, если нельзя сжать

0,

NULL,

CREATE\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

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

MessageBox(hwnd, L"Cannot create compressed file.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

// Создать XpressHuff compressor.

Success = CreateCompressor(

COMPRESS\_ALGORITHM\_XPRESS\_HUFF, // алгоритм сжатия

NULL, // параметр для опционального аллокатора памяти

&Compressor);

if (!Success) {

MessageBox(hwnd, L"Cannot create a compressor.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

// Запросить размер сжатого буфера.

Success = Compress(

Compressor, // Обработчик компрессора

InputBuffer, // Буфер ввода, несжатые данные

InputFileSize, // Размер несжатых данных

NULL, // Буфер сжатых данных

0, // Размер буфера для сжатых данных

&CompressedBufferSize); // Размер сжатых данных

// Выделить память для сжатого буфера.

if (!Success) {

DWORD ErrorCode = GetLastError();

if (ErrorCode != ERROR\_INSUFFICIENT\_BUFFER) {

MessageBox(hwnd, L"Cannot compress data.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

CompressedBuffer = (PBYTE)malloc(CompressedBufferSize);

if (!CompressedBuffer) {

MessageBox(hwnd, L"Cannot allocate memory for compressed buffer.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

}

StartTime = GetTickCount64();

// Вызовите Compress() еще раз для выполнения реального сжатия и вывода сжатых

// данных в CompressedBuffer.

Success = Compress(

Compressor, // Обработчик компрессора

InputBuffer, // Буфер ввода, несжатые данные

InputFileSize, // Размер несжатых данных

CompressedBuffer, // Буфер сжатых данных

CompressedBufferSize, // Размер буфера для сжатых данных

&CompressedDataSize); // Размер сжатых данных

if (!Success) {

MessageBox(hwnd, L"Cannot compress data.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

EndTime = GetTickCount64();

// Получить время сжатия.

TimeDuration = (EndTime - StartTime) / 1000.0;

// Записать сжатые данные в выходной файл.

Success = WriteFile(

CompressedFile, // Дескриптор файла

CompressedBuffer, // Начало данных для записи

CompressedDataSize, // Количество байт для записи

&ByteWritten, // Количество записанных байт

NULL); // Нет структуры перекрытия

if ((ByteWritten != CompressedDataSize) || (!Success)) {

MessageBox(hwnd, L"Cannot write compressed data to file.", L"Error", MB\_ICONERROR | MB\_OK);

goto done;

}

wchar\_t infoMessage[256];

swprintf\_s(infoMessage, L"Input file size: %d; Compressed Size: %d\n", InputFileSize, CompressedDataSize);

MessageBox(hwnd, infoMessage, L"Compression Info", MB\_OK | MB\_ICONINFORMATION);

swprintf\_s(infoMessage, L"Compression Time (Exclude I/O): %.2f seconds\n", TimeDuration);

MessageBox(hwnd, infoMessage, L"Compression Time", MB\_OK | MB\_ICONINFORMATION);

MessageBox(hwnd, L"File Compressed.", L"Success", MB\_OK | MB\_ICONINFORMATION);

DeleteTargetFile = FALSE;

done:

if (Compressor != NULL) {

CloseCompressor(Compressor);

}

if (CompressedBuffer) {

free(CompressedBuffer);

}

if (InputBuffer) {

free(InputBuffer);

}

if (InputFile != INVALID\_HANDLE\_VALUE) {

CloseHandle(InputFile);

}

if (CompressedFile != INVALID\_HANDLE\_VALUE) {

// Сжатие не удалось, удалить сжатый файл.

if (DeleteTargetFile) {

FILE\_DISPOSITION\_INFO fdi;

fdi.DeleteFile = TRUE; // Пометка для удаления

Success = SetFileInformationByHandle(

CompressedFile,

FileDispositionInfo,

&fdi,

sizeof(FILE\_DISPOSITION\_INFO));

if (!Success) {

MessageBox(hwnd, L"Cannot delete corrupted compressed file.", L"Error", MB\_ICONERROR | MB\_OK);

}

}

CloseHandle(CompressedFile);

}

}

void DecompressFile(HWND hwnd) {

std::wstring sourcePath = GetOpenFileName(hwnd);

if (sourcePath.empty()) {

return; // Пользователь отменил выбор файла

}

std::wstring destPath = GetSaveFileName(hwnd);

if (destPath.empty()) {

return; // Пользователь отменил выбор места сохранения файла

}

DECOMPRESSOR\_HANDLE Decompressor = NULL;

PBYTE CompressedBuffer = NULL;

PBYTE DecompressedBuffer = NULL;

HANDLE InputFile = INVALID\_HANDLE\_VALUE;

HANDLE DecompressedFile = INVALID\_HANDLE\_VALUE;

BOOL DeleteTargetFile = TRUE;

BOOL Success;

SIZE\_T DecompressedBufferSize, DecompressedDataSize;

DWORD InputFileSize, ByteRead, ByteWritten;

ULONGLONG StartTime, EndTime;

LARGE\_INTEGER FileSize;

double TimeDuration;

// Открыть для чтения только существующий файл

InputFile = CreateFile(

sourcePath.c\_str(),

GENERIC\_READ,

FILE\_SHARE\_READ,

NULL,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

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

MessageBox(hwnd, L"Cannot open source file!", L"Error", MB\_ICONERROR);

goto done;

}

// Получить размер сжатого файла.

Success = GetFileSizeEx(InputFile, &FileSize);

if ((!Success) || (FileSize.QuadPart > 0xFFFFFFFF))

{

MessageBox(hwnd, L"Cannot get file size!", L"Error", MB\_ICONERROR);

goto done;

}

InputFileSize = FileSize.LowPart;

// Выделение памяти под сжатые данные.

CompressedBuffer = (PBYTE)malloc(InputFileSize);

if (!CompressedBuffer)

{

MessageBox(hwnd, L"Memory allocation error!", L"Error", MB\_ICONERROR);

goto done;

}

// Прочитать сжатые данные в буфер.

Success = ReadFile(InputFile, CompressedBuffer, InputFileSize, &ByteRead, NULL);

if ((!Success) || (ByteRead != InputFileSize))

{

MessageBox(hwnd, L"Cannot read from source file!", L"Error", MB\_ICONERROR);

goto done;

}

// Открыть пустой файл для записи, если существует - перезапис

DecompressedFile = CreateFile(

destPath.c\_str(),

GENERIC\_WRITE | DELETE,

0,

NULL,

CREATE\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

if (DecompressedFile == INVALID\_HANDLE\_VALUE)

{

MessageBox(hwnd, L"Cannot create destination file!", L"Error", MB\_ICONERROR);

goto done;

}

// Создать XpressHuff распаковщик.

Success = CreateDecompressor(

COMPRESS\_ALGORITHM\_XPRESS\_HUFF, // алгоритм сжатия

NULL, // параметр для опционального аллокатора памяти

&Decompressor);

if (!Success)

{

MessageBox(hwnd, L"Cannot create decompressor!", L"Error", MB\_ICONERROR);

goto done;

}

// Запросить размер распаковочного буфера.

Success = Decompress(

Decompressor, // дескриптор компрессора

CompressedBuffer,

InputFileSize,

NULL, // Буфер установлен в NULL

0, // Buffer size в 0

&DecompressedBufferSize); // Размер данных распаковки

// Выделение памяти для распаковочного буфера.

if (!Success)

{

DWORD ErrorCode = GetLastError();

// Первоначальный размер, возвращаемый функцией,

// извлекается из самого буфера и должен рассматриваться как ненадежный и проверяться

// на разумные пределы.

if (ErrorCode != ERROR\_INSUFFICIENT\_BUFFER)

{

MessageBox(hwnd, L"Cannot decompress data", L"Error", MB\_ICONERROR);

goto done;

}

DecompressedBuffer = (PBYTE)malloc(DecompressedBufferSize);

if (!DecompressedBuffer)

{

MessageBox(hwnd, L"Cannot allocate memory for decompressed buffer.", L"Error", MB\_ICONERROR);

goto done;

}

}

StartTime = GetTickCount64();

// Распаковка данных и их запись в DecompressedBuffer.

Success = Decompress(

Decompressor, // дескриптор распаковщика

CompressedBuffer, // сжатые данные

InputFileSize, // размер сжатых данных

DecompressedBuffer, // буфер распаковки

DecompressedBufferSize, // размер буфера распаковки

&DecompressedDataSize); // размер распакованных данных

if (!Success)

{

MessageBox(hwnd, L"Cannot decompress data 2", L"Error", MB\_ICONERROR);

goto done;

}

EndTime = GetTickCount64();

// Получение времени распаковки.

TimeDuration = (EndTime - StartTime) / 1000.0;

// Записать распакованные данные в выходной файл.

Success = WriteFile(

DecompressedFile,

DecompressedBuffer, // Начало данных для записи

DecompressedDataSize, // Количество байт для записи

&ByteWritten, // КОличество записанных байт

NULL); // Нет структуры перекрытия

if ((ByteWritten != DecompressedDataSize) || (!Success))

{

MessageBox(hwnd, L"Cannot write to destination file!", L"Error", MB\_ICONERROR);

}

wchar\_t infoMessage[256];

swprintf\_s(infoMessage, L"Compressed size: %d; Decompressed Size: %d\n", InputFileSize, DecompressedDataSize);

MessageBox(hwnd, infoMessage, L"Compression Info", MB\_OK | MB\_ICONINFORMATION);

swprintf\_s(infoMessage, L"Decompression Time(Exclude I/O): %.2f seconds\n", TimeDuration);

MessageBox(hwnd, infoMessage, L"Compression Time", MB\_OK | MB\_ICONINFORMATION);

MessageBox(hwnd, L"File decompressed.", L"Success", MB\_OK | MB\_ICONINFORMATION);

DeleteTargetFile = FALSE;

done:

if (Decompressor != NULL)

{

CloseDecompressor(Decompressor);

}

if (CompressedBuffer)

{

free(CompressedBuffer);

}

if (DecompressedBuffer)

{

free(DecompressedBuffer);

}

if (InputFile != INVALID\_HANDLE\_VALUE)

{

CloseHandle(InputFile);

}

if (DecompressedFile != INVALID\_HANDLE\_VALUE)

{

// Распаковка не удалась, удалить файл

if (DeleteTargetFile)

{

FILE\_DISPOSITION\_INFO fdi;

fdi.DeleteFile = TRUE; // Пометка для удаления

Success = SetFileInformationByHandle(

DecompressedFile,

FileDispositionInfo,

&fdi,

sizeof(FILE\_DISPOSITION\_INFO));

if (!Success) {

MessageBox(hwnd, L"Cannot delete corrupted decompressed file.", L"Error", MB\_ICONERROR | MB\_OK);

}

}

CloseHandle(DecompressedFile);

}

}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {

switch (msg) {

case WM\_CREATE: {

CreateWindowEx(0, L"BUTTON", L"Compress", WS\_CHILD | WS\_VISIBLE, 10, 10, 100, 30, hwnd, (HMENU)IDC\_COMPRESS\_BUTTON, GetModuleHandle(nullptr), nullptr);

CreateWindowEx(0, L"BUTTON", L"Decompress", WS\_CHILD | WS\_VISIBLE, 120, 10, 100, 30, hwnd, (HMENU)IDC\_DECOMPRESS\_BUTTON, GetModuleHandle(nullptr), nullptr);

break;

}

case WM\_COMMAND: {

switch (LOWORD(wParam)) {

case IDC\_COMPRESS\_BUTTON: {

CompressFile(hwnd);

break;

}

case IDC\_DECOMPRESS\_BUTTON: {

DecompressFile(hwnd);

break;

}

}

break;

}

case WM\_DESTROY: {

PostQuitMessage(0);

break;

}

default: {

return DefWindowProc(hwnd, msg, wParam, lParam);

}

}

return 0;

}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

WNDCLASSEX wc;

HWND hwnd;

MSG Msg;

// Регистрация класса окна

wc.cbSize = sizeof(WNDCLASSEX);

wc.style = 0;

wc.lpfnWndProc = WndProc;

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hInstance = hInstance;

wc.hIcon = LoadIcon(nullptr, IDI\_APPLICATION);

wc.hCursor = LoadCursor(nullptr, IDC\_ARROW);

wc.hbrBackground = (HBRUSH)(COLOR\_WINDOW + 1);

wc.lpszMenuName = nullptr;

wc.lpszClassName = g\_szClassName;

wc.hIconSm = LoadIcon(nullptr, IDI\_APPLICATION);

if (!RegisterClassEx(&wc)) {

MessageBox(nullptr, L"Window Registration Failed!", L"Error!", MB\_ICONEXCLAMATION | MB\_OK);

return 0;

}

// Создание окна

hwnd = CreateWindowEx(0, g\_szClassName, L"File Compression App", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, CW\_USEDEFAULT, 300, 100, nullptr, nullptr, hInstance, nullptr);

if (hwnd == nullptr) {

MessageBox(nullptr, L"Window Creation Failed!", L"Error!", MB\_ICONEXCLAMATION | MB\_OK);

return 0;

}

// Отображение окна

ShowWindow(hwnd, nCmdShow);

UpdateWindow(hwnd);

// Цикл обработки сообщений

while (GetMessage(&Msg, nullptr, 0, 0) > 0) {

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}

return Msg.wParam;

}