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

**ОТЧЕТ**

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 гр 153504

Рогачёв А.А.

Проверил:

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc146720283)

[2 Теоретические сведения 4](#_Toc146720284)

[3 Реализация программного продукта 5](#_Toc146720285)

[4 Результат выполнения программы 6](#_Toc146720286)

[Список использованных источников 7](#_Toc146720287)

[Приложение А 8](#_Toc146720288)

# **1 ЦЕЛЬ РАБОТЫ**

Задачей этой лабораторной работы является исследование функционала Win 32 API для работы с памятью и операций ввода-вывода. Для выполнения этого проекта были освоены запись и чтение из файла, использование полей ввода для получения данных и их отображения.

# **2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Win32 API это набор функций и структур, предоставляемых операционной системой Windows для разработки приложений под эту платформу.[1] Win32 API предоставляет доступ к различным системным ресурсам и функциям, включая работу с окнами и оконными сообщениями. [2]

Windows предоставляет разработчикам множество инструментов и API для эффективного управления памятью. Это включает в себя выделение и освобождение динамической памяти с помощью функций, таких как malloc и free, а также использование различных специализированных функций для работы с памятью, таких как VirtualAlloc и VirtualFree. Подсистема памяти Win32 обеспечивает контроль над адресным пространством приложения и управление виртуальной памятью.[3]

Для ввода и вывода данных Windows предоставляет API, такие как ReadFile и WriteFile, которые позволяют выполнять синхронные операции ввода-вывода. Однако для улучшения производительности и отзывчивости приложений, также доступны асинхронные операции ввода-вывода через механизмы, такие как I/O Completion Ports и функции, позволяющие организовывать асинхронные вызовы, например ReadFileEx и WriteFileEx.[4]

# **3 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ПРОДУКТА**

Код прописанный в Main Interface.cpp представляет собой простую реализацию игры "Сапёр" на платформе Windows с использованием библиотеки WinAPI. Он создает графический интерфейс для игры и обрабатывает пользовательский ввод для управления игровым процессом. В дополнение к этому имеется меню с утилитами, которое содержит инструменты для создания своих уровней сложности.

В функции WinMain создается и отображается "Utility Window". Пользователь может его скрыть и показать. "Utility Window" содержит следующие элементы, отвечающие за сложность:

Комбо-бокс с выбором сложности: Позволяет пользователю выбрать уровень сложности игры, включая созданные пользователем.

Поля ввода для названия сложности и количества мин: Позволяют пользователю задать название новой сложности и количество мин выставляемое ей.

Кнопка “Save”: Используется для сохранения новой сложности.

В "Utility Window" обрабатываются связанные с созданием новых сложностей события.

Комбо-бокс с выбором сложности: При выборе уровня сложности обновляет параметры игры и перезапускает ее с новыми параметрами.

Кнопка “Save”: При нажатии записывает новую сложность в файл и добавляет её в комбо-бокс.

# **4 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ**

В данном приложении имеется меню, содержащее кнопку сохранения Save и два поля ввода для создания сложности (рисунок 4.1).



Рисунок 4.1 – Инструменты для создания сложности

При нажатии на кнопку, выводится диалог, который показывает было ли сохранение новой сложности успешным или нет (рисунок 4.2).

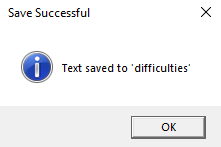


Рисунок 4.2 – Диалог при удачном сохранении

При нажатии на комбо-бокс пользователю предлагается выбрать сложность (рисунок 4.3).

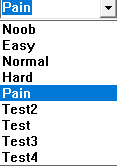


Рисунок 4.3 – Выбор сложности, с успешно добавленной новой сложностью

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Get Started with Win32 and C++ [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/learnwin32/learn-to-program-for-windows

[2] Введение в Win32 API [Электронный ресурс]. – Режим доступа: http://cppstudio.com/post/9384/

[3] Функции управления памятью [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/memory/memory-management-functions

[4] Открытие файла для чтения или записи [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/fileio/opening-a-file-for-reading-or-writing

# **ПРИЛОЖЕНИЕ А**

**(обязательное)**

#include <windows.h>

#include <windowsx.h>

#include <vector>

#include <ctime>

#include <string>

#include <fstream>

#include <sstream>

#include <thread>

// Define grid size and mine count

const int GRID\_SIZE = 10;

int NUM\_MINES = 10; // You can adjust the number of mines as needed

const int CELL\_SIZE = 30; // Define cell size

int windowWidth = CELL\_SIZE \* GRID\_SIZE + 30;

int windowHeight = CELL\_SIZE \* GRID\_SIZE + 160;

bool GameOver = false;

int remainingMines = NUM\_MINES;

int flaggedMines = 0;

std::vector<bool> isMine(GRID\_SIZE\* GRID\_SIZE, false);

std::vector<bool> isFlagged(GRID\_SIZE\* GRID\_SIZE, false);

std::vector<bool> isChecked(GRID\_SIZE\* GRID\_SIZE, false);

std::vector<bool> isRightClicked(GRID\_SIZE\* GRID\_SIZE, false); // Track right-clicks

HWND hLabel = NULL; // Global variable to store the label control handle

HWND hMineLabel = NULL;

HWND hRestartButton = NULL;

HWND hExitButton = NULL;

HWND hUtilityWnd = NULL;

HWND hDialogButton = NULL;

HWND hComboBox = NULL;

// Define constants for difficulty levels

/\*enum Difficulty {

Noob = 1,

Easy = 5,

Normal = 10,

Hard = 15,

Pain = 20

};\*/

std::vector<std::pair<std::wstring, int>> Difficulty;

int currentDifficulty = 3; // Default difficulty

void SaveTextToFile(const std::wstring& textCharacters, const std::wstring& textIntegers) {

// Open a file for writing using WinAPI

HANDLE hFile = CreateFile(L"difficulties", GENERIC\_WRITE, 0, NULL, OPEN\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

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

std::wstring fileData = L"\n" + textCharacters + L"\n" + textIntegers;

DWORD bytesWritten;

// Move to the end of the file

SetFilePointer(hFile, 0, NULL, FILE\_END);

if (WriteFile(hFile, fileData.c\_str(), fileData.size() \* sizeof(wchar\_t), &bytesWritten, NULL)) {

MessageBox(NULL, L"Text saved to 'difficulties'", L"Save Successful", MB\_OK | MB\_ICONINFORMATION);

}

else {

MessageBox(NULL, L"Failed to write to the file!", L"Save Error", MB\_OK | MB\_ICONERROR);

}

CloseHandle(hFile);

}

else {

MessageBox(NULL, L"Failed to create or open the file for writing!", L"Save Error", MB\_OK | MB\_ICONERROR);

}

}

void LoadDifficultiesFromFile() {

Difficulty.push\_back(std::pair<std::wstring, int>(L"Noob", 1));

Difficulty.push\_back(std::pair<std::wstring, int>(L"Easy", 5));

Difficulty.push\_back(std::pair<std::wstring, int>(L"Normal", 10));

Difficulty.push\_back(std::pair<std::wstring, int>(L"Hard", 15));

Difficulty.push\_back(std::pair<std::wstring, int>(L"Pain", 20));

HANDLE hFile = CreateFile(L"difficulties", GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

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

std::wstring fileData;

DWORD bytesRead;

fileData.resize(GetFileSize(hFile, NULL));

if (ReadFile(hFile, &fileData[0], GetFileSize(hFile, NULL), &bytesRead, NULL)) {

std::wstringstream ss(fileData);

std::wstring line;

while (std::getline(ss, line)) {

if (line == L"[" || line == L"]") {

continue;

}

else {

std::wstring namebuffer = line;

std::getline(ss, line);

std::wstring numbuffer = line;

Difficulty.push\_back(std::pair<std::wstring, int>(namebuffer, stoi(numbuffer)));

}

}

}

else {

MessageBox(NULL, L"Failed to read from the file!", L"Load Error", MB\_OK | MB\_ICONERROR);

}

CloseHandle(hFile);

}

else {

CreateFile(L"difficulties", GENERIC\_READ, 0, NULL, CREATE\_NEW, FILE\_ATTRIBUTE\_NORMAL, NULL);

// Handle the case where the file does not exist

// You can create a default difficulty file here if needed.

}

}

HHOOK g\_hHook = NULL;

HWND g\_hMainWindow = NULL;

bool g\_bUtilityWindowVisible = false;

void ToggleUtilityWindowVisibility() {

g\_bUtilityWindowVisible = !g\_bUtilityWindowVisible;

ShowWindow(hUtilityWnd, g\_bUtilityWindowVisible ? SW\_SHOW : SW\_HIDE);

}

// Function to randomly place mines on the grid

void PlaceMines() {

srand(static\_cast<unsigned>(time(nullptr)));

int minesPlaced = 0;

while (minesPlaced < NUM\_MINES) {

int randomPosition = rand() % (GRID\_SIZE \* GRID\_SIZE);

if (!isMine[randomPosition]) {

isMine[randomPosition] = true;

minesPlaced++;

}

}

}

// Function to draw the grid cells

void DrawGridCells(HDC hdc) {

for (int row = 0; row < GRID\_SIZE; ++row) {

for (int col = 0; col < GRID\_SIZE; ++col) {

// Calculate cell position

int x = col \* CELL\_SIZE;

int y = row \* CELL\_SIZE;

int dr[] = { -1, -1, -1, 0, 0, 1, 1, 1 };

int dc[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

int mineCount = 0;

for (int i = 0; i < 8; ++i) {

int newRow = row + dr[i];

int newCol = col + dc[i];

// Check if the neighboring cell is within bounds

if (newRow >= 0 && newRow < GRID\_SIZE && newCol >= 0 && newCol < GRID\_SIZE) {

int cellId = newRow \* GRID\_SIZE + newCol;

if (isMine[cellId]) {

mineCount++;

}

}

}

// Create a white brush for non-flagged cells

HBRUSH hBrush = CreateSolidBrush(RGB(255, 255, 255));

// Check if the cell is flagged

int cellId = row \* GRID\_SIZE + col;

if (isFlagged[cellId]) {

// Change the brush to red for flagged cells

DeleteObject(hBrush); // Delete the previous white brush

hBrush = CreateSolidBrush(RGB(255, 0, 0));

}

else if (isChecked[cellId])

{

DeleteObject(hBrush); // Delete the previous white brush

hBrush = CreateSolidBrush(RGB(192, 192, 192));

}

RECT cellRect = { x, y, x + CELL\_SIZE, y + CELL\_SIZE }; // Create a RECT structure

// Fill the cell with the selected brush color

FillRect(hdc, &cellRect, hBrush);

if (mineCount > 0 && isChecked[cellId]) {

// Display mine count in the cell for cells with nearby mines

SetBkMode(hdc, TRANSPARENT);

wchar\_t mineCountStr[2];

swprintf\_s(mineCountStr, L"%d", mineCount);

DrawText(hdc, mineCountStr, -1, &cellRect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

}

// Draw an outline around the cell

FrameRect(hdc, &cellRect, CreateSolidBrush(RGB(0, 0, 0)));

// Delete the brush

DeleteObject(hBrush);

}

}

}

bool CheckWinCondition() {

for (int row = 0; row < GRID\_SIZE; ++row) {

for (int col = 0; col < GRID\_SIZE; ++col) {

int cellId = row \* GRID\_SIZE + col;

// If any empty cell is not revealed or any mine is not flagged, the game is not won

if ((!isMine[cellId] && isFlagged[cellId]) || (isMine[cellId] && !isFlagged[cellId])) {

return false;

}

}

}

return true;

}

void YouWin() {

SetWindowText(hLabel, L"YOU WIN");

GameOver = true;

}

// Function to reveal empty cells and count mines in the vicinity

void RevealEmptyCells(int row, int col, HDC hdc) {

// Array of neighboring offsets (8 positions)

int dr[] = { -1, -1, -1, 0, 0, 1, 1, 1 };

int dc[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

int mineCount = 0;

for (int i = 0; i < 8; ++i) {

int newRow = row + dr[i];

int newCol = col + dc[i];

// Check if the neighboring cell is within bounds

if (newRow >= 0 && newRow < GRID\_SIZE && newCol >= 0 && newCol < GRID\_SIZE) {

int cellId = newRow \* GRID\_SIZE + newCol;

if (isMine[cellId]) {

mineCount++;

}

}

}

// Calculate cell position

int x = col \* CELL\_SIZE;

int y = row \* CELL\_SIZE;

// Create a white brush for non-flagged cells

HBRUSH hBrush = CreateSolidBrush(RGB(255, 255, 255));

RECT cellRect = { x, y, x + CELL\_SIZE, y + CELL\_SIZE };

// Gray out revealed cells

hBrush = CreateSolidBrush(RGB(192, 192, 192));

// Fill the cell with the selected brush color

FillRect(hdc, &cellRect, hBrush);

if (mineCount > 0) {

// Display mine count in the cell for cells with nearby mines

SetBkMode(hdc, TRANSPARENT);

wchar\_t mineCountStr[2];

swprintf\_s(mineCountStr, L"%d", mineCount);

DrawText(hdc, mineCountStr, -1, &cellRect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

}

// Draw an outline around the cell

FrameRect(hdc, &cellRect, CreateSolidBrush(RGB(0, 0, 0)));

// Delete the brush

DeleteObject(hBrush);

// Recursively reveal neighboring empty cells if mine count is 0

if (mineCount == 0) {

for (int i = 0; i < 8; ++i) {

int newRow = row + dr[i];

int newCol = col + dc[i];

// Check if the neighboring cell is within bounds

if (newRow >= 0 && newRow < GRID\_SIZE && newCol >= 0 && newCol < GRID\_SIZE) {

int cellId = newRow \* GRID\_SIZE + newCol;

if (!isFlagged[cellId] && !isChecked[cellId] && !isMine[cellId]) {

isChecked[cellId] = true;

RevealEmptyCells(newRow, newCol, hdc);

}

}

}

}

}

// Function to show "YOU DIED" on the label and lock gameplay

void YouDied() {

SetWindowText(hLabel, L"YOU DIED");

GameOver = true;

}

void RestartGame() {

// Reset game variables and redraw the grid

GameOver = false;

isMine.assign(GRID\_SIZE \* GRID\_SIZE, false);

isFlagged.assign(GRID\_SIZE \* GRID\_SIZE, false);

isChecked.assign(GRID\_SIZE \* GRID\_SIZE, false);

NUM\_MINES = Difficulty[currentDifficulty].second;

PlaceMines();

InvalidateRect(g\_hMainWindow, NULL, FALSE);

SetWindowText(hLabel, L""); // Clear the label

remainingMines = NUM\_MINES;

flaggedMines = 0;

wchar\_t mineCountStr[50];

swprintf\_s(mineCountStr, L"Mines Remaining: %d", NUM\_MINES - flaggedMines);

SetWindowText(hMineLabel, mineCountStr);

}

void LaunchNewInstance() {

std::thread newInstanceThread([]() {

HINSTANCE hInstance = GetModuleHandle(NULL);

// Reset game variables and redraw the grid

GameOver = false;

isMine.assign(GRID\_SIZE \* GRID\_SIZE, false);

isFlagged.assign(GRID\_SIZE \* GRID\_SIZE, false);

isChecked.assign(GRID\_SIZE \* GRID\_SIZE, false);

NUM\_MINES = Difficulty[currentDifficulty].second;

PlaceMines();

InvalidateRect(g\_hMainWindow, NULL, FALSE);

SetWindowText(hLabel, L""); // Clear the label

remainingMines = NUM\_MINES;

flaggedMines = 0;

wchar\_t mineCountStr[50];

swprintf\_s(mineCountStr, L"Mines Remaining: %d", NUM\_MINES - flaggedMines);

SetWindowText(hMineLabel, mineCountStr);

// Create a new main window instance with default options

WinMain(hInstance, NULL, NULL, SW\_SHOWNORMAL);

});

// Detach the thread to allow it to run independently

newInstanceThread.detach();

}

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

switch (message) {

case WM\_CREATE:

// Create the label control

// Create the "Restart" button

hRestartButton = CreateWindow(L"BUTTON", L"Restart", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

10, windowHeight - 130, 80, 30, hWnd, (HMENU)1001, NULL, NULL);

// Create the "Exit" button

hExitButton = CreateWindow(L"BUTTON", L"Exit", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

100, windowHeight - 130, 80, 30, hWnd, (HMENU)1002, NULL, NULL);

hLabel = CreateWindow(L"STATIC", L"", WS\_CHILD | WS\_VISIBLE | SS\_CENTER,

0, windowHeight - 160, windowWidth, 30, hWnd, NULL, NULL, NULL);

hMineLabel = CreateWindow(L"STATIC", L"", WS\_CHILD | WS\_VISIBLE | SS\_CENTER,

0, windowHeight - 70, windowWidth, 30, hWnd, NULL, NULL, NULL);

SetWindowText(hLabel, L""); // Initially empty text

wchar\_t mineCountStr[50];

swprintf\_s(mineCountStr, L"Mines Remaining: %d", NUM\_MINES - flaggedMines);

SetWindowText(hMineLabel, mineCountStr);

PlaceMines(); // Randomly place mines

break;

case WM\_COMMAND:

switch (LOWORD(wParam)) {

case 1001: // Restart button

// Reset game variables and redraw the grid

GameOver = false;

isMine.assign(GRID\_SIZE \* GRID\_SIZE, false);

isFlagged.assign(GRID\_SIZE \* GRID\_SIZE, false);

isChecked.assign(GRID\_SIZE \* GRID\_SIZE, false);

PlaceMines();

InvalidateRect(hWnd, NULL, FALSE);

SetWindowText(hLabel, L""); // Clear the label

remainingMines = NUM\_MINES;

flaggedMines = 0;

wchar\_t mineCountStr[50];

swprintf\_s(mineCountStr, L"Mines Remaining: %d", NUM\_MINES - flaggedMines);

SetWindowText(hMineLabel, mineCountStr);

LoadDifficultiesFromFile();

break;

case 1002: // Exit button

// Close the application

PostQuitMessage(0);

break;

}

break;

case WM\_PAINT:

{

PAINTSTRUCT ps;

HDC hdc = BeginPaint(hWnd, &ps);

DrawGridCells(hdc); // Draw the grid cells

EndPaint(hWnd, &ps);

}

break;

case WM\_LBUTTONDOWN:

// Handle left mouse button down

if (!GameOver) {

int xPos = LOWORD(lParam) / CELL\_SIZE;

int yPos = HIWORD(lParam) / CELL\_SIZE;

int cellId = yPos \* GRID\_SIZE + xPos;

if (xPos >= GRID\_SIZE || yPos >= GRID\_SIZE)

break;

if (!isFlagged[cellId] && !isChecked[cellId]) {

if (!isMine[cellId]) {

// Reveal the clicked empty cell and its neighbors

isChecked[cellId] = true;

HDC hdc = GetDC(hWnd);

RevealEmptyCells(yPos, xPos, hdc);

ReleaseDC(hWnd, hdc);

}

else {

YouDied();

}

}

if (CheckWinCondition()) {

YouWin(); // Check for win condition after each move

}

}

break;

case WM\_RBUTTONDOWN:

// Handle right mouse button down

if (!GameOver) {

int xPos = LOWORD(lParam) / CELL\_SIZE;

int yPos = HIWORD(lParam) / CELL\_SIZE;

int cellId = yPos \* GRID\_SIZE + xPos;

if (!isFlagged[cellId] && !isChecked[cellId]) {

// Mark the cell as a mine

isFlagged[cellId] = true;

flaggedMines++;

wchar\_t mineCountStr[50];

swprintf\_s(mineCountStr, L"Mines Remaining: %d", NUM\_MINES - flaggedMines);

SetWindowText(hMineLabel, mineCountStr);

// Redraw the cell to indicate flagging

InvalidateRect(hWnd, NULL, FALSE);

}

else

{

isFlagged[cellId] = false;

flaggedMines--;

wchar\_t mineCountStr[50];

swprintf\_s(mineCountStr, L"Mines Remaining: %d", NUM\_MINES - flaggedMines);

SetWindowText(hMineLabel, mineCountStr);

InvalidateRect(hWnd, NULL, FALSE);

}

}

if (CheckWinCondition()) {

YouWin(); // Check for win condition after each move

}

break;

case WM\_DESTROY:

// Post a quit message to exit the application

PostQuitMessage(0);

break;

default:

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

}

return 0;

}

LRESULT CALLBACK KeyboardHook(int nCode, WPARAM wParam, LPARAM lParam) {

if (nCode == HC\_ACTION && (wParam == WM\_KEYDOWN || wParam == WM\_SYSKEYDOWN)) {

KBDLLHOOKSTRUCT\* pKeyInfo = (KBDLLHOOKSTRUCT\*)lParam;

if (pKeyInfo->vkCode == 'H') { // Check for the "H" key

ToggleUtilityWindowVisibility(); // Toggle the Utility Window's visibility

}

}

return CallNextHookEx(g\_hHook, nCode, wParam, lParam);

}

// Utility Window procedure

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

static HWND hEdit = NULL; // Input for characters

static HWND hIntEdit = NULL; // Input for integers

static std::wstring textCharacters;

static std::wstring textIntegers;

switch (message) {

case WM\_CREATE:

// Create the "Show Dialog" button

hDialogButton = CreateWindow(L"BUTTON", L"Show Dialog", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

10, 10, 100, 30, hWnd, (HMENU)2001, NULL, NULL);

// Create the edit control for the first input (characters)

hEdit = CreateWindow(L"EDIT", L"", WS\_CHILD | WS\_VISIBLE | WS\_BORDER | ES\_AUTOHSCROLL,

10, 50, 200, 30, hWnd, (HMENU)2003, NULL, NULL);

// Create another edit control for the second input (integers)

hIntEdit = CreateWindow(L"EDIT", L"", WS\_CHILD | WS\_VISIBLE | WS\_BORDER | ES\_AUTOHSCROLL | ES\_NUMBER,

10, 90, 80, 30, hWnd, (HMENU)2004, NULL, NULL);

// Create a "Save" button

CreateWindow(L"BUTTON", L"Save", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

10, 130, 80, 30, hWnd, (HMENU)2005, NULL, NULL);

hComboBox = CreateWindow(L"COMBOBOX", NULL, WS\_CHILD | WS\_VISIBLE | CBS\_DROPDOWN,

120, 10, 120, 200, hWnd, (HMENU)2002, NULL, NULL);

// Create the "Launch new instance" button

CreateWindow(L"BUTTON", L"Launch new instance", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

10, 170, 100, 30, hWnd, (HMENU)2006, NULL, NULL);

// Add difficulty levels to the combo box

for (size\_t i = 0; i < Difficulty.size(); ++i) {

SendMessage(hComboBox, CB\_ADDSTRING, 0, (LPARAM)Difficulty[i].first.c\_str());

}

// Select the current difficulty in the combo box

SendMessage(hComboBox, CB\_SETCURSEL, static\_cast<WPARAM>(currentDifficulty) - 1, 0);

g\_hHook = SetWindowsHookEx(WH\_KEYBOARD\_LL, KeyboardHook, NULL, 0);

break;

case WM\_COMMAND:

switch (LOWORD(wParam)) {

case 2001: // Show Dialog button

MessageBox(hWnd, L"ЛКМ - открыть клетку, ПКМ - поставить флаг, H - открыть меню", L"Game Instructions", MB\_OK);

break;

case 2005: // Save button

{

const int bufferSize = 256; // Adjust the buffer size as needed

wchar\_t buffer[bufferSize];

// Get text from the character input field

GetWindowText(hEdit, buffer, bufferSize);

textCharacters = buffer;

// Get text from the integer input field

GetWindowText(hIntEdit, buffer, bufferSize);

textIntegers = buffer;

// Append the text to the difficultyEntries vector

Difficulty.push\_back(std::pair<std::wstring, int>(textCharacters, stoi(textIntegers)));

// Save the text to the file

SaveTextToFile(textCharacters, textIntegers);

// Update the combo box with the new difficulty entry

SendMessage(hComboBox, CB\_ADDSTRING, 0, (LPARAM)Difficulty.back().first.c\_str());

// Clear input fields

SetWindowText(hEdit, L"");

SetWindowText(hIntEdit, L"");

break;

}

case 2002: // Combo box selection

if (HIWORD(wParam) == CBN\_SELCHANGE) {

// Get the selected difficulty level

int selected = SendMessage(hComboBox, CB\_GETCURSEL, 0, 0);

if (selected != CB\_ERR) {

// Update the current difficulty

currentDifficulty = selected;

RestartGame(); // Restart with the new difficulty

}

}

break;

case 2006: // Launch new instance button

LaunchNewInstance();

break;

}

break;

case WM\_CLOSE:

// Hide the Utility Window instead of closing it

ShowWindow(hUtilityWnd, SW\_HIDE);

break;

case WM\_DESTROY:

// Post a quit message for the Utility Window

PostQuitMessage(0);

break;

default:

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

}

return 0;

}

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

// Define the window class for the main window

WNDCLASSEX wcex = { sizeof(WNDCLASSEX), CS\_HREDRAW | CS\_VREDRAW, WndProc, 0, 0, GetModuleHandle(NULL), NULL, NULL, NULL, NULL, L"GridWindowClass", NULL };

// Register the window class for the main window

RegisterClassEx(&wcex);

// Create the application window with a fixed size

HWND hWnd = CreateWindow(L"GridWindowClass", L"Grid of Buttons", WS\_OVERLAPPEDWINDOW & ~WS\_THICKFRAME & ~WS\_MAXIMIZEBOX,

CW\_USEDEFAULT, CW\_USEDEFAULT, windowWidth, windowHeight, NULL, NULL, hInstance, NULL);

if (!hWnd) {

return -1;

}

LoadDifficultiesFromFile();

// Show and update the window

ShowWindow(hWnd, nCmdShow);

UpdateWindow(hWnd);

// Set the global main window handle

g\_hMainWindow = hWnd;

// Register the window class for the Utility Window

WNDCLASSEX wcexUtility = { sizeof(WNDCLASSEX), CS\_HREDRAW | CS\_VREDRAW, UtilityWndProc, 0, 0, GetModuleHandle(NULL), NULL, NULL, NULL, NULL, L"UtilityWindowClass", NULL };

RegisterClassEx(&wcexUtility);

// Create the Utility Window

hUtilityWnd = CreateWindow(L"UtilityWindowClass", L"Utility Window", WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT, 500, 400, NULL, NULL, hInstance, NULL);

if (!hUtilityWnd) {

return -1;

}

// Show and update the Utility Window

ShowWindow(hUtilityWnd, SW\_HIDE); // Initially hide the Utility Window

UpdateWindow(hUtilityWnd);

MSG msg;

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

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return (int)msg.wParam;

}