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 №4

на тему

**УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПОТОКАМИ (WINDOWS).**

Студент Е. С. Багровец

Преподаватель Н. Ю. Гриценко

Минск 2023

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_6127s8b6t8wy)

[2 Краткие теоретические сведения 4](#_r4y5108cx8yu)

[3 Результат выполнения лабораторной работы](#_jnfz1qbvv5j4) 5

[Выводы](#_ldsbh3e5c0xh) 6

[Список использованных источников 7](#_qto16fco0nf2)

[Приложение А (обязательное) Листинг кода](#_54xlhz7mfhe) 8

# 1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является разработка графического приложения с использованием управления процессами и потоками (Windows) их порождение, завершение, изменение приоритетов и исследование эффективности. Приложение должно использовать управление потоками и процессами, уметь запускать в отдельном потоке, отдельные процессы для компиляции и запуска проекта, отслеживать потребление ресурсов потоков и вывод графиков.

# 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ ТРЕБОВАНИЯ

Вычислительный процесс (упрощенно) – работающая программа. Как правило, в системе определен тип объекта процесс.

Вычислительный поток (thread) – последовательность инструкций, выполняемая независимо и асинхронно по отношению к другим аналогичным последовательностей. Отражение в иерархии объектов – объект поток (thread).

Сообщения (message) – более информативны: обладают дополнительными параметрами, что позволяет использовать их для передачи более сложной информации о событиях и в качестве опосредованных системой вызовов к другим процессам.

В Windows – оконные сообщения (window messages), но с оговорками: window messages могут передаваться не только синхронно, но и через очередь (асинхронно). В отличие от одиночных синхронных сообщений, очереди соо бщений (message queue) предполагают асинхронное взаимодействие и пригодны для передачи потока данных, поэтому правильнее рассматривать их как средство межпроцессного обмена.

Для выполнения данной лабораторной работы, были использованы следующие теоретические сведения и концепции:

1. Функция CreateThread создает для процесса новый поток. Созданный поток должен определить начальный адрес кода, с которого новый поток должен исполняться. Как правило, начальный адрес – это название функции, определенной в коде программы. Эта функция получает единственный параметр и возвращает значение типа DWORD. Процесс может иметь одновременно несколько потоков, выполняющих ту же самую функцию.[1]
2. Функция [CreateProcess](https://learn.microsoft.com/ru-ru/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa) создает новый процесс, который выполняется независимо от процесса создания. Если [CreateProcess](https://learn.microsoft.com/ru-ru/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa) завершается успешно, он возвращает [структуру PROCESS\_INFORMATION](https://learn.microsoft.com/ru-ru/windows/win32/api/processthreadsapi/ns-processthreadsapi-process_information), содержащую дескрипторы и идентификаторы для нового процесса и его основного потока. Дескрипторы потоков и процессов создаются с правами полного доступа, хотя вы можете ограничить доступ, если указать дескрипторы безопасности. Если эти дескрипторы больше не нужны, закройте их с помощью функции [CloseHandle](https://learn.microsoft.com/ru-ru/windows/desktop/api/handleapi/nf-handleapi-closehandle).[2]
3. WaitForSingleObject ждет, когда объект, хэндл которого был передан первым аргументом, перейдет в сигнальное состояние (signaled state). Кроме того, в зависимости от типа объекта, процедура может менять его состояние. WaitForSingleObject может быть применен к мьютексам, семафорам, потокам, процессам и не только. Если hHandle представляет собой хэндл мьютекса, процедура ждет, когда мьютекс освободится, а затем лочит его. Если же hHandle является хэндлом потока, то процедура просто ждет его завершения. Второй аргумент задает время ожидания в миллисекундах. Можно ждать вечно, передав специальное значение INFINITE.[3]

# 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы было реализовано графическое приложение, позволяющие компилировать и запускать .pro файлы используя отдельные потоки, а также отображать ресурсы, потребляемые потоком (Рисунок 1).

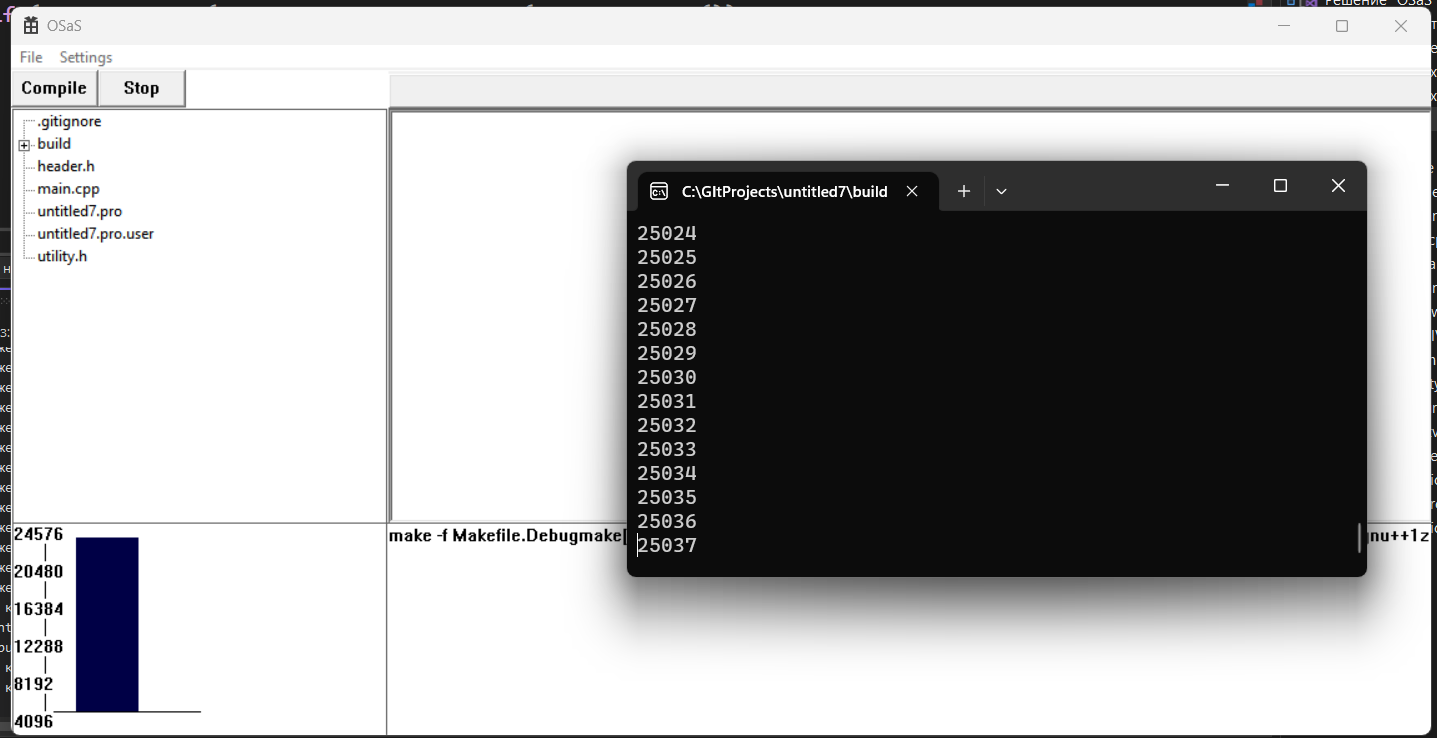


Рисунок 1 – Результат работы программы

# ВЫВОДЫ

В ходе выполнения данной лабораторной работы было разработано графическое приложение, способное эффективно управлять процессами и потоками в операционной системе Windows. Приложение успешно осуществляет порождение, завершение и изменение приоритетов процессов и потоков. Кроме того, оно демонстрирует функциональность по запуску компиляции и запуска проекта в отдельных процессах и отслеживанию потребления ресурсов потоков, что позволяет визуализировать данную информацию в виде графиков. Это подчеркивает важность эффективного управления процессами и потоками для оптимизации работы приложений, повышения производительности и обеспечения лучшего пользовательского опыта.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Thread creation [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/procthread/creating-threads
2. Process creation [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/procthread/creating-processes
3. WaitForSingleObject [Электронный ресурс]. – Режим доступа: https://eax.me/winapi-threads/

# ПРИЛОЖЕНИЕ А

# (обязательное)

# Листинг кода

Листинг 1 – Файл main.cpp

// OSaS.cpp : Определяет точку входа для приложения.

//

#include "framework.h"

#include "prototypes.h"

#include "fileHandlers.h"

#include "globalVariables.h"

#include "main.h"

int WINAPI wWinMain(HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPWSTR lpCmdLine,

int nCmdShow)

{

UNREFERENCED\_PARAMETER(hPrevInstance);

UNREFERENCED\_PARAMETER(lpCmdLine);

// Инициализация глобальных строк

LoadStringW(hInstance, IDS\_APP\_TITLE, szTitle, MAX\_LOADSTRING);

LoadStringW(hInstance, IDC\_OSAS, szWindowClass, MAX\_LOADSTRING);

MyRegisterClass(hInstance);

// Выполнить инициализацию приложения:

if (!InitInstance(hInstance, nCmdShow))

{

return FALSE;

}

HACCEL hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC\_OSAS));

hKeyboardHook = SetWindowsHookEx(WH\_KEYBOARD\_LL, KeyboardProc, hInstance, 0);

MSG msg;

// Цикл основного сообщения:

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

{

if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

UnhookWindowsHookEx(hKeyboardHook);

return (int)msg.wParam;

}

void MonitorDirectoryChanges() {

HANDLE directoryHandle = CreateFileW(

currentDirectory.c\_str(),

FILE\_LIST\_DIRECTORY,

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

NULL,

OPEN\_EXISTING,

FILE\_FLAG\_BACKUP\_SEMANTICS | FILE\_FLAG\_OVERLAPPED,

NULL

);

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

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

return;

}

DWORD bufferSize = 1024;

BYTE buffer[1024];

DWORD bytesReturned;

OVERLAPPED overlapped = { 0 };

while (true) {

ZeroMemory(buffer, bufferSize);

BOOL result = ReadDirectoryChangesW(

directoryHandle,

&buffer,

bufferSize,

TRUE,

FILE\_NOTIFY\_CHANGE\_LAST\_WRITE | FILE\_NOTIFY\_CHANGE\_FILE\_NAME,

&bytesReturned,

&overlapped,

NULL

);

if (!result) {

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

break;

}

// Wait for the completion of the overlapped operation

DWORD waitResult = WaitForSingleObject(directoryHandle, INFINITE);

if (waitResult == WAIT\_OBJECT\_0) {

std::wcout << L"Directory changed." << std::endl;

// TODO: Handle the directory change as needed

}

else {

std::cerr << "WaitForSingleObject failed: " << GetLastError() << std::endl;

break;

}

}

CloseHandle(directoryHandle);

}

//

// ФУНКЦИЯ: MyRegisterClass()

//

// ЦЕЛЬ: Регистрирует класс окна.

//

ATOM MyRegisterClass(HINSTANCE hInstance)

{

WNDCLASSEXW wcex;

wcex.cbSize = sizeof(WNDCLASSEX);

wcex.style = CS\_HREDRAW | CS\_VREDRAW;

wcex.lpfnWndProc = WndProc;

wcex.cbClsExtra = 0;

wcex.cbWndExtra = 0;

wcex.hInstance = hInstance;

wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI\_OSAS));

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

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

wcex.lpszMenuName = MAKEINTRESOURCEW(IDC\_OSAS);

wcex.lpszClassName = szWindowClass;

wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI\_SMALL));

return RegisterClassExW(&wcex);

}

//

// ФУНКЦИЯ: InitInstance(HINSTANCE, int)

//

// ЦЕЛЬ: Сохраняет маркер экземпляра и создает главное окно

//

// КОММЕНТАРИИ:

//

// В этой функции маркер экземпляра сохраняется в глобальной переменной, а также

// создается и выводится главное окно программы.

//

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)

{

hInst = hInstance; // Сохранить маркер экземпляра в глобальной переменной

hMainWindow = CreateWindowW(szWindowClass, szTitle, WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, 0, CW\_USEDEFAULT, 0, nullptr, nullptr, hInstance, nullptr);

if (!hMainWindow)

{

return FALSE;

}

ShowWindow(hMainWindow, nCmdShow);

UpdateWindow(hMainWindow);

return TRUE;

}

//

// ФУНКЦИЯ: WndProc(HWND, UINT, WPARAM, LPARAM)

//

// ЦЕЛЬ: Обрабатывает сообщения в главном окне.

//

// WM\_COMMAND - обработать меню приложения

// WM\_PAINT - Отрисовка главного окна

// WM\_DESTROY - отправить сообщение о выходе и вернуться

//

//

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

{

switch (message)

{

case WM\_COMMAND:

{

int wmId = LOWORD(wParam);

std::wstring messageNum = std::to\_wstring(wmId);

// OutputDebugString(messageNum.c\_str());

switch (wmId)

{

case IDM\_EXIT:

DestroyWindow(hWnd);

break;

case HIGHTLIGHT\_TEXT:

if (editWidget) {

HighLightKeyWords();

}

break;

case NEW\_FILE\_COMMAND:

NewFileCommand();

break;

case OPEN\_FILE\_COMMAND:

OpenFileCommand();

break;

case SAVE\_FILE\_COMMAND:

SaveFileCommand();

break;

case CLOSE\_FILE\_COMMAND:

CloseFileCommand();

break;

case CLOSE\_TAB\_COMMAND:

CloseFileCommand();

break;

case CHANGE\_BG\_COLOR:

OpenColorDialog();

break;

case CHANGE\_FONT:

ChangeFont();

break;

case CHANGE\_FONT\_COLOR:

ChangeFontColor();

break;

case OPEN\_FOLDER\_COMMAND:

OpenFolder(hWnd);

break;

default:

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

}

}

break;

case WM\_NOTIFY:

{

NMHDR\* nmhdr = (NMHDR\*)lParam;

if (nmhdr->code == TCN\_SELCHANGE) {

SwitchTab();

}

else if (nmhdr->idFrom == IDC\_TREEVIEW && nmhdr->code == NM\_DBLCLK) {

OutputDebugString((currentDirectory + L"\\" + GetPathRelativeToRoot(treeView,

TreeView\_GetSelection((HWND)nmhdr->hwndFrom))).c\_str());

OpenFile((currentDirectory + L"\\" + GetPathRelativeToRoot(treeView,

TreeView\_GetSelection((HWND)nmhdr->hwndFrom))).c\_str());

}

}

break;

case WM\_SIZE:

modifiedWidth = LOWORD(lParam);

modifiedHeight = HIWORD(lParam);

MoveWindow(tabControlWidget, 302, 1, modifiedWidth - 300, 30, TRUE);

MoveWindow(editWidget, 302, 31, modifiedWidth - 300, modifiedHeight, TRUE);

MoveWindow(treeView, 1, 31, 300, modifiedHeight, TRUE);

break;

case WM\_USER + 1:

TreeView\_DeleteAllItems(treeView);

PopulateTreeViewWithFiles(treeView,TreeView\_GetRoot(treeView), currentDirectory, 0 );

break;

case WM\_PAINT:

{

PAINTSTRUCT ps;

HDC hdc = BeginPaint(hWnd, &ps);

EndPaint(hWnd, &ps);

}

break;

case WM\_CREATE:

WinWidgetsCreation(hWnd);

WinMenuCreation(hWnd);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

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

}

return 0;

}

LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam)

{

if (nCode >= 0 && wParam == WM\_KEYDOWN)

{

KBDLLHOOKSTRUCT\* pKeyInfo = reinterpret\_cast<KBDLLHOOKSTRUCT\*>(lParam);

if (GetAsyncKeyState(VK\_CONTROL) & 0x8000)

{

switch (pKeyInfo->vkCode)

{

case 'S':

SaveFileCommand();

break;

case 'O':

OpenFileCommand();

break;

case 'N':

NewFileCommand();

break;

default:

break;

}

}

}

return CallNextHookEx(hKeyboardHook, nCode, wParam, lParam);

}

void WinWidgetsCreation(HWND hWnd) {

tabControlWidget = CreateWindowEx(0, WC\_TABCONTROL, L"", WS\_CHILD | WS\_VISIBLE | TCS\_FIXEDWIDTH,

1, 0, 500, 30, hWnd, (HMENU)IDC\_TABCTRL, GetModuleHandle(NULL), NULL);

LoadLibrary(TEXT("Msftedit.dll"));

editWidget = CreateWindowEx(

0,

MSFTEDIT\_CLASS, L"",

WS\_BORDER | WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | WS\_VSCROLL | WS\_TABSTOP,

302, 30, 400, 600,

hWnd, (HMENU)IDC\_RICHEDIT, hInst, NULL

);

treeView = CreateWindowEx(

0,

WC\_TREEVIEW,

L"",

WS\_VISIBLE | WS\_CHILD | WS\_BORDER | TVS\_HASBUTTONS | TVS\_LINESATROOT | TVS\_SHOWSELALWAYS | TVS\_HASLINES,

1, 30, 300, 600,

hWnd,

(HMENU)IDC\_TREEVIEW,

hInst,

NULL);

wchar\_t text[] = L"Root";

TVITEM tvItem = { 0 };

tvItem.mask = TVIF\_TEXT;

tvItem.pszText = text;

TVINSERTSTRUCT tvInsert = { 0 };

tvInsert.hParent = TVI\_ROOT;

tvInsert.item = tvItem;

MonitorDirectoryChanges();

SendMessage(editWidget, EM\_SETEVENTMASK, 0, ENM\_CHANGE);

}

void WinMenuCreation(HWND hWnd) {

HMENU fileMenu = CreateMenu();

HMENU fileSettings = CreateMenu();

HMENU fileSubMenu = CreateMenu();

HMENU settingsSubMenu = CreateMenu();

AppendMenu(fileSubMenu, MF\_STRING, OPEN\_FOLDER\_COMMAND, L"Open folder");

AppendMenu(fileSubMenu, MF\_STRING, NEW\_FILE\_COMMAND, L"New File");

AppendMenu(fileSubMenu, MF\_STRING, OPEN\_FILE\_COMMAND, L"Open File");

AppendMenu(fileSubMenu, MF\_STRING, SAVE\_FILE\_COMMAND, L"Save File");

AppendMenu(fileSubMenu, MF\_STRING, CLOSE\_FILE\_COMMAND, L"Close");

AppendMenu(settingsSubMenu, MF\_STRING, CHANGE\_BG\_COLOR, L"Bg");

AppendMenu(settingsSubMenu, MF\_STRING, CHANGE\_FONT, L"Font");

AppendMenu(settingsSubMenu, MF\_STRING, CHANGE\_FONT\_COLOR, L"Font color");

AppendMenu(settingsSubMenu, MF\_STRING, HIGHTLIGHT\_TEXT, L"Highlight text");

AppendMenu(fileMenu, MF\_POPUP, (UINT\_PTR)fileSubMenu, L"File");

AppendMenu(fileMenu, MF\_POPUP, (UINT\_PTR)settingsSubMenu, L"Settings");

SetMenu(hWnd, fileMenu);

}

void CloseFileCommand()

{

if (TabCtrl\_GetItemCount(tabControlWidget) == 0)

{

return;

}

int result = MessageBox(NULL, L"Do you want to save the file?", L"Save File", MB\_YESNOCANCEL | MB\_ICONQUESTION);

if (result == IDCANCEL) {

return;

}

else if (result == IDYES)

{

SaveFileCommand();

}

auto name = fileNames[tabIndex];

filePathes.erase(name);

fileData.erase(name);

fileNames.erase(fileNames.begin() + tabIndex);

TabCtrl\_DeleteItem(tabControlWidget, tabIndex);

SwitchTab();

}

void SaveFileCommand()

{

if (!TabCtrl\_GetItemCount(tabControlWidget))

{

return;

}

tabIndex = TabCtrl\_GetCurSel(tabControlWidget);

int length = GetWindowTextLength(editWidget) + 1;

LPSTR data = new CHAR[length];

length = GetWindowTextA(editWidget, data, length);

auto it = filePathes.find(fileNames[tabIndex]);

if (it != filePathes.end())

{

LPWSTR path = const\_cast<LPWSTR>((\*it).second.c\_str());

SaveFile(path, data, length);

}

else

{

OPENFILENAME ofn;

wchar\_t szFileName[1024] = L"";

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn);

ofn.hwndOwner = NULL;

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

ofn.lpstrFile = szFileName;

ofn.nMaxFile = MAX\_PATH;

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

if (GetOpenFileName(&ofn) == TRUE)

{

SaveFile(szFileName, data, length);

std::wstring name = szFileName;

name = name.substr(name.find\_last\_of('\\') + 1);

fileData[name] = fileData[fileNames[tabIndex]];

fileNames[tabIndex] = name;

filePathes[name] = szFileName;

TCITEM tcItem = { 0 };

tcItem.mask = TCIF\_TEXT;

tcItem.pszText = const\_cast<LPWSTR>(name.data());

TabCtrl\_SetItem(tabControlWidget, tabIndex, &tcItem);

}

}

}

void NewFileCommand()

{

int num = 1;

auto it = fileData.find(L"File" + std::to\_wstring(num));

while (it != fileData.end())

{

num++;

it = fileData.find(L"File" + std::to\_wstring(num));

}

std::wstring fileName = L"File" + std::to\_wstring(num);

CreateTab(const\_cast<LPWSTR>(fileName.c\_str()), L"", nullptr);

}

void OpenFileCommand()

{

OPENFILENAME ofn;

wchar\_t szFileName[1024] = L"";

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn);

ofn.hwndOwner = NULL;

ofn.lpstrFilter = L"Text Files\0\*.cpp;\*.h\0";;

ofn.lpstrFile = szFileName;

ofn.nMaxFile = MAX\_PATH;

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

if (GetOpenFileName(&ofn) == TRUE)

{

LPCWSTR path = szFileName;

if (OpenFile(path)) {

currentDirectory = path;

}

}

}

void ReplaceCloseButton(int currentTabIndex)

{

/\* RECT tabRect;

TabCtrl\_GetItemRect(tabControlWidget, currentTabIndex, &tabRect);

if (closeButton)

{

MoveWindow(closeButton, tabRect.right - 20, tabRect.top + 2, 15, 15, TRUE);

}

else

{

closeButton = CreateWindowA("button", "X", WS\_CHILD | WS\_VISIBLE,

tabRect.right - 20, tabRect.top + 2, 15, 15, tabControlWidget,

(HMENU)CLOSE\_TAB\_COMMAND, NULL, NULL);

}\*/

}

void HighLightKeyWords()

{

DWORD cursorPos;

SendMessage(editWidget, EM\_GETSEL, reinterpret\_cast<WPARAM>(&cursorPos), NULL);

const std::unordered\_map<std::wstring, COLORREF> wordColors = {

{ L"class", RGB(0, 255, 0) },

{ L"struct", RGB(0, 255, 0) },

{ L"static", RGB(0, 0, 255) },

{ L"int", RGB(0, 0, 255) },

{ L"auto", RGB(0, 0, 255) },

{ L"float", RGB(0, 0, 255) },

{ L"void", RGB(0, 0, 255) },

{ L"string", RGB(0, 0, 255) },

{ L"double", RGB(0, 0, 255) },

{ L"static", RGB(0, 0, 255) },

{ L"while", RGB(255, 0, 255) },

{ L"if", RGB(255, 0, 255) } ,

{ L"switch", RGB(255, 0, 255) } ,

{ L"return", RGB(255, 0, 255) } ,

};

CHARFORMAT2 charFormat;

memset(&charFormat, 0, sizeof(CHARFORMAT2));

charFormat.cbSize = sizeof(CHARFORMAT2);

charFormat.dwMask = CFM\_COLOR;

charFormat.crTextColor = RGB(0, 0, 0);

int textLength = GetWindowTextLength(editWidget);

std::wstring text(textLength + 1, L'\0');

GetWindowText(editWidget, &text[0], textLength + 1);

for (auto& entry : wordColors)

{

FINDTEXTEXW findText;

findText.chrg.cpMin = 0;

findText.chrg.cpMax = -1;

findText.lpstrText = entry.first.c\_str();

while (SendMessage(editWidget, EM\_FINDTEXTEX, FR\_DOWN, (LPARAM)&findText) != -1)

{

CHARRANGE selRange;

selRange.cpMin = findText.chrgText.cpMin;

selRange.cpMax = findText.chrgText.cpMax;

SendMessage(editWidget, EM\_EXSETSEL, 0, (LPARAM)&selRange);

charFormat.crTextColor = entry.second;

SendMessage(editWidget, EM\_SETCHARFORMAT, SCF\_SELECTION, (LPARAM)&charFormat);

findText.chrg.cpMin = findText.chrgText.cpMax;

findText.chrg.cpMax = -1;

}

}

}

void SwitchTab()

{

if (TabCtrl\_GetItemCount(tabControlWidget) == 0)

{

DestroyWindow(closeButton);

closeButton = nullptr;

SetWindowTextW(editWidget, L"");

return;

}

int length = GetWindowTextLengthW(editWidget) + 1;

LPWSTR data = new WCHAR[length];

length = GetWindowTextW(editWidget, data, length);

fileData[fileNames[tabIndex]] = data;

tabIndex = TabCtrl\_GetCurSel(tabControlWidget);

SendMessage(editWidget, EM\_SETSEL, 0, -1);

SendMessage(editWidget, EM\_REPLACESEL, TRUE,

reinterpret\_cast<LPARAM>(fileData[fileNames[tabIndex]].c\_str()));

ReplaceCloseButton(tabIndex);

}

void CreateTab(LPWSTR name, LPCWSTR data, LPWSTR path)

{

if (TabCtrl\_GetItemCount(tabControlWidget) != 0)

{

int length = GetWindowTextLengthW(editWidget) + 1;

LPWSTR newData = new WCHAR[length];

length = GetWindowTextW(editWidget, newData, length);

fileData[fileNames[tabIndex]] = newData;

}

TCITEM tie;

tie.mask = TCIF\_TEXT | TCIS\_BUTTONPRESSED;

tie.pszText = name;

TabCtrl\_InsertItem(tabControlWidget, fileNames.size(), &tie);

fileNames.emplace\_back(name);

fileData.insert(std::make\_pair(fileNames.back(), std::wstring(data)));

if (path)

{

filePathes.insert(std::make\_pair(fileNames.back(), std::wstring(path)));

}

tabIndex = fileNames.size() - 1;

SetWindowTextW(editWidget, data);

ReplaceCloseButton(fileNames.size() - 1);

TabCtrl\_SetCurSel(tabControlWidget, fileNames.size() - 1);

}

void OpenColorDialog()

{

CHOOSECOLOR cc = { sizeof(CHOOSECOLOR) };

static COLORREF custColors[16] = { 0 };

cc.rgbResult = RGB(255, 255, 255);

cc.lpCustColors = custColors;

cc.Flags = CC\_FULLOPEN | CC\_RGBINIT;

if (ChooseColor(&cc))

{

SendMessage(editWidget, EM\_SETBKGNDCOLOR, FALSE, cc.rgbResult);

}

}

void ChangeFont()

{

CHARFORMAT2 cf;

cf.cbSize = sizeof(CHARFORMAT2);

CHOOSEFONT cfDialogParams = { 0 };

LOGFONT lf = { 0 };

cfDialogParams.lStructSize = sizeof(CHOOSEFONT);

cfDialogParams.hwndOwner = editWidget;

cfDialogParams.lpLogFont = &lf;

cfDialogParams.Flags = CF\_INITTOLOGFONTSTRUCT | CF\_EFFECTS | CF\_SCREENFONTS;

if (ChooseFont(&cfDialogParams)) {

lstrcpy(cf.szFaceName, lf.lfFaceName);

cf.yHeight = lf.lfHeight \* 20;

cf.dwEffects = 0;

cf.dwMask = CFM\_FACE | CFM\_SIZE;

SendMessage(editWidget, EM\_SETCHARFORMAT, SCF\_ALL, (LPARAM)&cf);

}

}

void ChangeFontColor() {

CHOOSECOLOR cc = { 0 };

ZeroMemory(&cc, sizeof(cc));

cc.lStructSize = sizeof(cc);

cc.hwndOwner = editWidget;

cc.lpCustColors = new COLORREF[16]{ 0 };

cc.Flags = CC\_FULLOPEN | CC\_RGBINIT;

if (ChooseColor(&cc))

{

COLORREF chosenColor = cc.rgbResult;

CHARFORMAT2 cf = { };

cf.cbSize = sizeof(CHARFORMAT2);

cf.dwMask = CFM\_COLOR;

cf.crTextColor = chosenColor;

SendMessage(editWidget, EM\_SETCHARFORMAT, SCF\_ALL, reinterpret\_cast<LPARAM>(&cf));

}

}

bool OpenFile(LPCWSTR path)

{

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

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

OutputDebugString(L"Unable to open file");

return false;

}

DWORD dwHighFileSize;

DWORD dwFileSize = GetFileSize(hFile, &dwHighFileSize);

if (dwFileSize == INVALID\_FILE\_SIZE) {

std::cerr << "GetFileSize failed";

CloseHandle(hFile);

return false;

}

HANDLE hMapFile = CreateFileMapping(hFile, NULL, PAGE\_READONLY, dwHighFileSize, dwFileSize, NULL);

if (hMapFile == NULL) {

OutputDebugString(L"Unable to create file mapping");

CloseHandle(hFile);

return false;

}

LPVOID lpMapAddress = MapViewOfFile(hMapFile, FILE\_MAP\_READ, 0, 0, dwFileSize);

if (lpMapAddress == NULL) {

OutputDebugString(L"Unable to map view of file");

CloseHandle(hMapFile);

CloseHandle(hFile);

return false;

}

std::wstring name = path;

name = name.substr(name.find\_last\_of('\\') + 1);

std::wstring data = std::wstring((char\*)lpMapAddress, (char\*)lpMapAddress + dwFileSize);

LPWSTR fileName = const\_cast<LPWSTR>(name.data());

LPCWSTR fileData = data.c\_str();

LPWSTR filePath = const\_cast<LPWSTR>(path);

CreateTab(fileName, fileData, filePath);

UnmapViewOfFile(lpMapAddress);

CloseHandle(hMapFile);

CloseHandle(hFile);

return true;

}

bool SaveFile(LPWSTR path, LPCSTR data, int length)

{

HANDLE hFile = CreateFile(path, GENERIC\_WRITE, 0, NULL,

CREATE\_ALWAYS, FILE\_FLAG\_OVERLAPPED, NULL);

if (hFile == INVALID\_HANDLE\_VALUE)

{

OutputDebugString(L"Unable to open file");

return 1;

}

OVERLAPPED overlapped = { 0 };

overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (!WriteFileEx(hFile, data, length, &overlapped, WriteCompletionRoutine))

{

OutputDebugString(L"Unable to async write");

CloseHandle(hFile);

return 1;

}

// Ожидание завершения асинхронной операции

SleepEx(INFINITE, TRUE);

CloseHandle(overlapped.hEvent);

CloseHandle(hFile);

return true;

}

void OpenFolder(HWND hWnd)

{

BROWSEINFO bi = { 0 };

bi.hwndOwner = hWnd;

bi.lpszTitle = L"Choose folder:";

bi.ulFlags = BIF\_NEWDIALOGSTYLE | BIF\_RETURNONLYFSDIRS;

LPITEMIDLIST pidl = SHBrowseForFolder(&bi);

if (pidl != NULL)

{

TreeView\_DeleteAllItems(treeView);

wchar\_t folderPath[MAX\_PATH];

SHGetPathFromIDList(pidl, folderPath);

currentDirectory = folderPath;

PopulateTreeViewWithFiles(treeView, TreeView\_GetRoot(treeView), folderPath, 0);

// Освобождаем память, выделенную для pidl

CoTaskMemFree(pidl);

}

}

void PopulateTreeViewWithFiles(HWND hTreeView, HTREEITEM hParent, const std::wstring& folderPath, const int& indent)

{

WIN32\_FIND\_DATA findFileData;

HANDLE hFind = FindFirstFile((folderPath + L"\\\*").c\_str(), &findFileData);

if (hFind == INVALID\_HANDLE\_VALUE)

return;

do

{

if (findFileData.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)

{

// Пропускаем папки "." и ".."

if (wcscmp(findFileData.cFileName, L".") != 0 && wcscmp(findFileData.cFileName, L"..") != 0)

{

TreeView\_SetIndent(hTreeView, indent);

// Добавляем папку в TreeView

TVITEM tvItem = { 0 };

tvItem.mask = TVIF\_TEXT | TVIF\_CHILDREN | TVIF\_PARAM;

tvItem.pszText = findFileData.cFileName;

tvItem.cChildren = 1;

TVINSERTSTRUCT tvInsert = { 0 };

tvInsert.hParent = hParent;

tvInsert.item = tvItem;

HTREEITEM hItem = TreeView\_InsertItem(hTreeView, &tvInsert);

PopulateTreeViewWithFiles(hTreeView, hItem, folderPath + L"\\" + findFileData.cFileName, indent + 30 );

}

}

else

{

TreeView\_SetIndent(hTreeView, indent);

// Добавляем файлы в TreeView

TVITEM tvItem = { 0 };

tvItem.mask = TVIF\_TEXT | TVIF\_PARAM;

tvItem.pszText = findFileData.cFileName;

TVINSERTSTRUCT tvInsert = { 0 };

tvInsert.hParent = hParent;

tvInsert.item = tvItem;

TreeView\_InsertItem(treeView, &tvInsert);

}

} while (FindNextFile(hFind, &findFileData) != 0);

FindClose(hFind);

}

void CALLBACK WriteCompletionRoutine(DWORD dwErrorCode, DWORD dwNumberOfBytesTransfered, LPOVERLAPPED lpOverlapped)

{

// Обработка завершения асинхронной операции записи

if (dwErrorCode == 0)

{

std::cout << "Асинхронная запись завершена. Байт записано: " << dwNumberOfBytesTransfered << std::endl;

}

else

{

std::cerr << "Ошибка асинхронной записи: " << dwErrorCode << std::endl;

}

}

std::wstring GetPathRelativeToRoot(HWND hTreeView, HTREEITEM hItem)

{

std::wstring path;

TVITEM item = { 0 };

item.mask = TVIF\_TEXT;

item.hItem = hItem;

item.cchTextMax = 255;

wchar\_t itemName[256];

item.pszText = itemName;

// Получаем имя текущего элемента

if (TreeView\_GetItem(hTreeView, &item)) {

path = itemName;

}

// Получаем родительский элемент

HTREEITEM hParent = TreeView\_GetParent(hTreeView, hItem);

// Если у элемента есть родитель, рекурсивно добавляем его к пути

if (hParent != nullptr) {

std::wstring parentPath = GetPathRelativeToRoot(hTreeView, hParent);

path = parentPath + L"\\" + path;

}

return path;

}

ativeToRoot(HWND hTreeView, HTREEITEM hItem)

{

std::wstring path;

TVITEM item = { 0 };

item.mask = TVIF\_TEXT;

item.hItem = hItem;

item.cchTextMax = 255;

wchar\_t itemName[256];

item.pszText = itemName;

if (TreeView\_GetItem(hTreeView, &item)) {

path = itemName;

}

HTREEITEM hParent = TreeView\_GetParent(hTreeView, hItem);

if (hParent != nullptr) {

std::wstring parentPath = GetPathRelativeToRoot(hTreeView, hParent);

path = parentPath + L"\\" + path;

}

return path;

}

void OpenProject(HWND hWnd) {

OPENFILENAME ofn;

wchar\_t szFileName[1024] = L"";

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn);

ofn.hwndOwner = NULL;

ofn.lpstrFilter = L"Text Files\0\*.pro\0";

ofn.lpstrFile = szFileName;

ofn.nMaxFile = MAX\_PATH;

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

if (GetOpenFileName(&ofn) == TRUE)

{

TreeView\_DeleteAllItems(treeView);

std::wstring path = szFileName;

std::wstring filePath, fileName;

size\_t found = path.find\_last\_of(L"\\/");

if (found != std::string::npos) {

filePath = path.substr(0, found);

fileName = path.substr(found + 1, path.size() - 1);

}

PopulateTreeViewWithFiles(treeView, TreeView\_GetRoot(treeView), filePath, 0);

currentDirectory = filePath;

currentProjectPath = path;

found = fileName.find\_last\_of(L".");

if (found != std::string::npos) {

fileName = fileName.substr(0, found);

}

currentProjectName = fileName;

}

}

DWORD WINAPI CompileProject(LPVOID lpParam) {

STARTUPINFO startupInfo = { sizeof(STARTUPINFO) };

std::wstring directory(currentDirectory.begin(), currentDirectory.end());

directory += L"\\build";

WIN32\_FIND\_DATA findFileData;

HANDLE hFind = FindFirstFile(directory.c\_str(), &findFileData);

if (hFind == INVALID\_HANDLE\_VALUE)

{

CreateDirectoryW(directory.c\_str(), NULL);

}

else

{

std::string directoryRemove(directory.begin(), directory.end());

directoryRemove = "rd /s /q " + directoryRemove;

OutputDebugString(std::wstring(directoryRemove.begin(), directoryRemove.end()).c\_str());

auto result = std::to\_wstring(system(directoryRemove.c\_str()));

result = L'\n' + result + L'\n';

SendMessage(consoleWidget, EM\_SETSEL, (WPARAM)result.data(), (LPARAM)result.size() \* sizeof(char));

SendMessage(consoleWidget, EM\_REPLACESEL, 0, (LPARAM)result.data());

CreateDirectoryW(directory.c\_str(), NULL);

}

FindClose(hFind);

directory = L" cd " + directory;

std::wstring complierQMake = directory + L" && C:/Qt/6.2.4/mingw\_64/bin/qmake.exe ";

std::wstring params = L" -spec win32-g++ \"CONFIG+=debug\" \"CONFIG+=qml\_debug\" && C:/Qt/Tools/mingw1120\_64/bin/mingw32-make.exe qmake &&";

std::wstring makeStrW = L" make | tee output.txt";

std::wstring compileStrW = complierQMake + currentProjectPath + params + makeStrW;

std::string compileStr(compileStrW.begin(), compileStrW.end());

OutputDebugString((compileStrW + L"\n").c\_str());

auto resultCompilation = std::to\_wstring(system(compileStr.c\_str()));

resultCompilation = L'\n' + resultCompilation + L'\n';

SendMessage(consoleWidget, EM\_SETSEL, (WPARAM)resultCompilation.c\_str(), (LPARAM)resultCompilation.size() \* sizeof(char));

SendMessage(consoleWidget, EM\_REPLACESEL, 0, (LPARAM)resultCompilation.c\_str());

ReadLogFile();

std::wstring exeStrW = currentDirectory + L"\\build\\debug\\" + currentProjectName + L".exe";

OutputDebugString((exeStrW + L"\n").c\_str());

isProcessCreated = true;

if (CreateProcess(NULL, const\_cast<LPWSTR>(exeStrW.c\_str()), NULL, NULL, FALSE, 0, NULL, NULL, &startupInfo, &processInfo)) {

WaitForSingleObject(processInfo.hProcess, INFINITE);

ShowWindow(stopButton, SW\_HIDE);

CloseHandle(processInfo.hProcess);

CloseHandle(processInfo.hThread);

PAINTSTRUCT ps;

HDC hdc = BeginPaint(resourcesGraphicWidget, &ps);

ClearGraph(hdc);

}

else {

OutputDebugString(L"exe error: " + GetLastError());

}

PostMessage(hMainWindow, WM\_USER + 2, 0, 0);

return 0;

}

DWORD WINAPI SaveFileThread(LPVOID param) {

SaveFileParams\* params = reinterpret\_cast<SaveFileParams\*>(param);

SaveFile(params->filePath, params->fileContent, params->fileSize);

delete params;

return 0;

}

void CompileProjectCommand() {

if (currentProjectPath.empty()) {

return;

}

WIN32\_FIND\_DATA findFileData;

HANDLE hFind = FindFirstFile((currentDirectory + L"\\\*").c\_str(), &findFileData);

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

OutputDebugString(L"Failed to list files with error: " + GetLastError());

return;

}

std::vector<std::wstring> fileNames;

do {

if (findFileData.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY) {

continue;

}

fileNames.push\_back(findFileData.cFileName);

} while (FindNextFile(hFind, &findFileData) != 0);

FindClose(hFind);

SaveFileCommand();

std::vector<HANDLE> threadHandles;

for (const std::wstring& fileName : fileNames) {

auto it = filePathes.find(fileName);

if (it != filePathes.end()) {

auto dataIt = fileData.find(fileName)->second;

int length = dataIt.size();

wchar\_t\* wcharPtr = new wchar\_t[length + 1];

wcscpy\_s(wcharPtr, length + 1, dataIt.c\_str());

int bufferSize = WideCharToMultiByte(CP\_ACP, 0, wcharPtr, -1, NULL, 0, NULL, NULL);

char\* ansiStr = new char[bufferSize];

WideCharToMultiByte(CP\_ACP, 0, wcharPtr, -1, ansiStr, bufferSize, NULL, NULL);

LPWSTR path = const\_cast<LPWSTR>((\*it).second.c\_str());

SaveFileParams\* params = new SaveFileParams{ path, ansiStr, length + 1 };

HANDLE hThread = CreateThread(NULL, 0, SaveFileThread, params, 0, NULL);

if (hThread == NULL) {

OutputDebugString(L"Failed to list files with error: " + GetLastError());

}

threadHandles.push\_back(hThread);

}

}

WaitForMultipleObjects(threadHandles.size(), threadHandles.data(), TRUE, INFINITE);

isProcessCreated = false;

for (const auto& hThread : threadHandles) {

CloseHandle(hThread);

}

if (сompileThread != NULL)

{

StopProjectCommand();

}

сompileThread = CreateThread(NULL, 0, CompileProject, NULL, 0, NULL);

if (сompileThread == NULL) {

OutputDebugString(L"Failed to create thread for compilation. Error code: " + GetLastError());

return;

}

graphicThread = CreateThread(NULL, 0, UpdateAndDraw, (LPVOID)resourcesGraphicWidget, 0, NULL);

if (graphicThread == NULL) {

OutputDebugString(L"Failed to create thread for compilation. Error code: " + GetLastError());

return;

}

SetPriorityClass(сompileThread, THREAD\_PRIORITY\_TIME\_CRITICAL);

ShowWindow(stopButton, SW\_SHOW);

}

void ReadLogFile() {

HANDLE hFile = CreateFileW((currentDirectory + L"\\build\\output.txt").c\_str(), GENERIC\_READ, FILE\_SHARE\_READ, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

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

DWORD fileSize = GetFileSize(hFile, NULL);

if (fileSize == INVALID\_FILE\_SIZE && GetLastError() != NO\_ERROR) {

CloseHandle(hFile);

return;

}

content.resize(fileSize + 1);

DWORD bytesRead;

if (ReadFile(hFile, content.data(), fileSize, &bytesRead, NULL)) {

SetWindowTextA(consoleWidget, content.data());

}

else {

return;

}

CloseHandle(hFile);

return;

}

else {

return ;

}

}

void StopProjectCommand() {

TerminateThread(сompileThread, 0);

if (isProcessCreated) {

TerminateProcess(processInfo.hProcess, 0);

TerminateThread(processInfo.hThread, 0);

TerminateThread(graphicThread, 0);

}

PAINTSTRUCT ps;

HDC hdc = BeginPaint(resourcesGraphicWidget, &ps);

ClearGraph(hdc);

ShowWindow(stopButton, SW\_HIDE);

}

void ClearGraph(HDC hdc)

{

RECT rect;

GetClientRect(resourcesGraphicWidget, &rect);

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

FillRect(hdc, &rect, hBrush);

}

void DrawGraph(HDC hdc, int memoryUsage)

{

int height = 200 - memoryUsage;

if (height < 5)

{

graphicScale++;

return;

}

Rectangle(hdc, 50, 150, 100, height);

RECT rect = { 50, 150, 100, height };

HBRUSH hBrush = CreateSolidBrush(RGB(0, 0, 70));

FillRect(hdc, &rect, hBrush);

}

DWORD WINAPI UpdateAndDraw(LPVOID lpParam)

{

HWND hWnd = (HWND)lpParam;

while (true)

{

PROCESS\_MEMORY\_COUNTERS\_EX pmc;

if (processInfo.hProcess != NULL && GetProcessMemoryInfo(processInfo.hProcess, (PROCESS\_MEMORY\_COUNTERS\*)&pmc, sizeof(pmc)))

{

HDC hdc = GetDC(hWnd);

ClearGraph(hdc);

DrawAxes(hdc, 200, 200, 50, 50, 1024 \* 10 \* graphicScale);

if (200 - (pmc.WorkingSetSize / (1024 \* 10 \* graphicScale)) < 5)

{

graphicScale++;

continue;

}

DrawGraph(hdc, pmc.WorkingSetSize / (1024 \* 10 \* graphicScale));

ReleaseDC(hWnd, hdc);

}

Sleep(100);

}

}

void DrawAxes(HDC hdc, int width, int height, int maxX, int maxY, int axesMaxValue) {

MoveToEx(hdc, 25, 150, nullptr);

LineTo(hdc, 25, 5);

MoveToEx(hdc, 25, 150, nullptr);

LineTo(hdc, 150, 150);

int stepY = 150 / 5;

int axesValue = axesMaxValue / 5;

for (int i = 150; i >= 0; i -= stepY, axesValue += axesMaxValue / 5) {

MoveToEx(hdc, 22, i, nullptr);

LineTo(hdc, 28, i);

std::wstring label = std::to\_wstring(axesValue);

TextOut(hdc, 0, i, label.c\_str(), label.length());

}

}

Листинг 2 – Файл fileHandlers.cpp

#include "fileHandlers.h"

#include "Resource.h"

#include "framework.h"

bool OpenFile(LPCSTR path, std::vector<char>& content)

{

HANDLE hFile = CreateFileA(path, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

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

DWORD fileSize = GetFileSize(hFile, NULL);

if (fileSize == INVALID\_FILE\_SIZE && GetLastError() != NO\_ERROR) {

CloseHandle(hFile);

return false;

}

content.resize(fileSize + 1);

DWORD bytesRead;

if (ReadFile(hFile, content.data(), fileSize, &bytesRead, NULL)) {

content[bytesRead] = '\0';

}

else {

return false;

}

CloseHandle(hFile);

}

else {

return false;

}

}

bool SaveFile(LPWSTR path, LPCSTR data, int length)

{

HANDLE file = CreateFileW(path , GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

DWORD bytes;

WriteFile(file, data, length, &bytes, NULL);

CloseHandle(file);

delete[] data;

return true;

}

void HighLightKeyWords()

{

const std::unordered\_map<std::wstring, COLORREF> wordColors = {

{ L"class", RGB(0, 255, 0) },

{ L"struct", RGB(0, 255, 0) },

{ L"static", RGB(0, 0, 255) },

{ L"int", RGB(0, 0, 255) },

{ L"auto", RGB(0, 0, 255) },

{ L"float", RGB(0, 0, 255) },

{ L"void", RGB(0, 0, 255) },

{ L"string", RGB(0, 0, 255) },

{ L"double", RGB(0, 0, 255) },

{ L"static", RGB(0, 0, 255) },

{ L"while", RGB(255, 0, 255) },

{ L"if", RGB(255, 0, 255) } ,

{ L"switch", RGB(255, 0, 255) } ,

{ L"return", RGB(255, 0, 255) } ,

};

CHARFORMAT2 charFormat;

memset(&charFormat, 0, sizeof(CHARFORMAT2));

charFormat.cbSize = sizeof(CHARFORMAT2);

charFormat.dwMask = CFM\_COLOR;

charFormt.crTextColor = RGB(0, 0, 0);

int textLength = GetWindowTextLength(editWidget);

std::wstring text(textLength + 1, L'\0');

GetWindowText(editWidget, &text[0], textLength + 1);

for (auto& entry : wordColors)

{

FINDTEXTEXW findText;

findText.chrg.cpMin = 0;

findText.chrg.cpMax = -1;

findText.lpstrText = entry.first.c\_str();

while (SendMessage(editWidget, EM\_FINDTEXTEX, FR\_DOWN, (LPARAM)&findText) != -1)

{

CHARRANGE selRange;

selRange.cpMin = findText.chrgText.cpMin;

selRange.cpMax = findText.chrgText.cpMax;

SendMessage(editWidget, EM\_EXSETSEL, 0, (LPARAM)&selRange);

charFormat.crTextColor = entry.second;

SendMessage(editWidget, EM\_SETCHARFORMAT, SCF\_SELECTION, (LPARAM)&charFormat);

findText.chrg.cpMin = findText.chrgText.cpMax;

findText.chrg.cpMax = -1;

}

}

}

void OpenColorDialog()

{

CHOOSECOLOR cc = { sizeof(CHOOSECOLOR) };

static COLORREF custColors[16] = { 0 };

cc.rgbResult = RGB(255, 255, 255);

cc.lpCustColors = custColors;

cc.Flags = CC\_FULLOPEN | CC\_RGBINIT;

if (ChooseColor(&cc))

{

SendMessage(editWidget, EM\_SETBKGNDCOLOR, FALSE, cc.rgbResult);

}

}

void ChangeFont()

{

CHARFORMAT2 cf;

cf.cbSize = sizeof(CHARFORMAT2);

CHOOSEFONT cfDialogParams = { 0 };

LOGFONT lf = { 0 };

cfDialogParams.lStructSize = sizeof(CHOOSEFONT);

cfDialogParams.hwndOwner = editWidget;

cfDialogParams.lpLogFont = &lf;

cfDialogParams.Flags = CF\_INITTOLOGFONTSTRUCT | CF\_EFFECTS | CF\_SCREENFONTS;

if (ChooseFont(&cfDialogParams)) {

lstrcpy(cf.szFaceName, lf.lfFaceName);

cf.yHeight = lf.lfHeight \* 20;

cf.dwEffects = 0;

cf.dwMask = CFM\_FACE | CFM\_SIZE;

SendMessage(editWidget, EM\_SETCHARFORMAT, SCF\_ALL, (LPARAM)&cf);

}

}

void ChangeFontColor() {

CHOOSECOLOR cc = {0};

ZeroMemory(&cc, sizeof(cc));

cc.lStructSize = sizeof(cc);

cc.hwndOwner = editWidget;

cc.lpCustColors = new COLORREF[16]{ 0 };

cc.Flags = CC\_FULLOPEN | CC\_RGBINIT;

if (ChooseColor(&cc))

{

COLORREF chosenColor = cc.rgbResult;

CHARFORMAT2 cf = { };

cf.cbSize = sizeof(CHARFORMAT2);

cf.dwMask = CFM\_COLOR;

cf.crTextColor = chosenColor;

SendMessage(editWidget, EM\_SETCHARFORMAT, SCF\_ALL, reinterpret\_cast<LPARAM>(&cf));

}

Листинг 3 – Файл prototypes.h

#pragma once

#include "framework.h"

ATOM MyRegisterClass(HINSTANCE );

BOOL InitInstance(HINSTANCE, int);

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

INT\_PTR CALLBACK About(HWND, UINT, WPARAM, LPARAM);

LRESULT CALLBACK KeyboardProc(int, WPARAM, LPARAM );

void WinMenuCreation(HWND);

void WinWidgetsCreation(HWND);

bool OpenFile(LPCSTR, std::vector<char>&);

bool SaveFile(LPWSTR, LPCSTR, int);

void NewFileCommand();

void OpenFileCommand();

void CloseFileCommand();

void SaveFileCommand();

void CreateTab(LPWSTR, LPCWSTR, LPWSTR);

void SwitchTab();

bool OpenFile(LPCSTR path, std::vector<char>& content);

bool SaveFile(LPWSTR path, LPCSTR data, int length);

void ReplaceCloseButton(int);

void HighLightKeyWords();

void OpenColorDialog();

void ChangeFont();

void ChangeFontColor();

void PopulateTreeViewWithFiles(HWND , HTREEITEM , const std::wstring& , const int& );

void CALLBACK WriteCompletionRoutine(DWORD , DWORD , LPOVERLAPPED );

std::wstring GetPathRelativeToRoot(HWND , HTREEITEM );

Листинг 4 – Файл Resources.h

#define IDS\_APP\_TITLE 103

#define IDC\_TABCTRL 120

#define IDR\_MAINFRAME 128

#define IDD\_OSAS\_DIALOG 102

#define IDD\_ABOUTBOX 103

#define IDM\_ABOUT 104

#define IDM\_EXIT 105

#define IDI\_OSAS 107

#define IDI\_SMALL 108

#define IDC\_OSAS 109

#define IDC\_MYICON 2

#ifndef IDC\_STATIC

#define IDC\_STATIC -1

#define IDC\_RICHEDIT 111

#endif

// Кастомные

#define NEW\_FILE\_COMMAND 10

#define OPEN\_FILE\_COMMAND 11

#define SAVE\_FILE\_COMMAND 12

#define CLOSE\_FILE\_COMMAND 13

#define CLOSE\_TAB\_COMMAND 14

#define CHANGE\_BG\_COLOR 15

#define CHANGE\_FONT 16

#define HIGHTLIGHT\_TEXT 17

#define CHANGE\_FONT\_COLOR 18

#ifdef APSTUDIO\_INVOKED

#ifndef APSTUDIO\_READONLY\_SYMBOLS

#define \_APS\_NO\_MFC 130

#define \_APS\_NEXT\_RESOURCE\_VALUE 129

#define \_APS\_NEXT\_COMMAND\_VALUE 32771

#define \_APS\_NEXT\_CONTROL\_VALUE 1000

#define \_APS\_NEXT\_SYMED\_VALUE 110

#endif

#endif