МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Комп'ютерний практикум

Навчальний посібник

Укладач: В. М. Порєв

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем»
спеціальності 121 «Інженерія програмного забезпечення»

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Об'єктно-орієнтоване програмування: комп'ютерний практикум [Електронний ресурс]: навч. посіб. для студ. освітньої програми «Інженерія програмного забезпечення комп'ютерних систем» спеціальності 121 «Інженерія програмного забезпечення» / КПІ ім. Ігоря Сікорського; уклад. Порєв В.М. – Електронні текстові дані (1 файл: 2,5 МБайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 105 с. – Назва з екрана.

Гриф надано Методичною радою КПІ м.. Ігоря Сікорського
(протокол № 1 від 02.09.2022 р.)
за поданням Вченої ради факультету інформатики та обчислювальної техніки
(протокол № 11 від 11.07.2022 р.)

Електронне мережне навчальне видання

Укладач: Порєв Віктор Миколайович, к.т.н.

Рецензент: Лісовиченко О.І., к.т.н., доцент, КПІ ім. Ігоря Сікорського, факультет інформатики та обчислювальної техніки, кафедра інформатики та програмної інженерії

Відповідальний редактор Волокита А.М., к.т.н., доцент, КПІ ім. Ігоря Сікорського, факультет інформатики та обчислювальної техніки, кафедра обчислювальної техніки

У навчальному посібнику викладені теоретичні положення та практичні завдання для виконання комп'ютерного лабораторного практикуму при вивченні дисципліни «Об'єктно-орієнтоване програмування». Базовою мовою програмування для виконання завдань обрано C++. Приділено увагу основним аспектам розроблення програмних додатків для операційної системи Windows. Наведено методичні рекомендації щодо роботи з проєктами C++ у середовищі Microsoft Visual Studio.

Навчальний посібник призначений для здобувачів ступеня бакалавра за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем»» спеціальності 121 «Інженерія програмного забезпечення». Може бути також корисним студентам, які вивчають програмування у курсах споріднених дисциплін відповідних спеціальностей.

Реєстр. № НП 22/23-013.Обсяг 3,5 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© КПІ ім. Ігоря Сікорського, 2022

ЗМІСТ

ВСТУП 5

Лабораторна робота №1. Знайомство із середовищем розробки програм Microsoft Visual Studio та складання модульних проєктів програм на С++ 6

Завдання 6

Теоретичні положення 6

Методичні рекомендації 7

Варіанти завдань та основні вимоги 34

Контрольні запитання 36

Лабораторна робота №2. Розробка графічного редактора об’єктів на С++ 37

Завдання 37

Теоретичні положення 37

Методичні рекомендації 40

Варіанти завдань та основні вимоги 52

Контрольні запитання 54

Лабораторна робота №3. Розробка інтерфейсу користувача на С++ 55

Завдання 55

Теоретичні положення 55

Методичні рекомендації 56

Варіанти завдань та основні вимоги 66

Контрольні запитання 68

Лабораторна робота №4. Вдосконалення структури коду графічного редактора об’єктів на С++ 69

Завдання 69

Теоретичні положення 69

Методичні рекомендації 72

Варіанти завдань та основні вимоги 78

Контрольні запитання 79

Лабораторна робота №5. Розробка багатовіконного інтерфейсу користувача для графічного редактора об’єктів 80

Завдання 80

Теоретичні положення 80

Методичні рекомендації 84

Варіанти завдань та основні вимоги 88

Контрольні запитання 89

Лабораторна робота №6. Побудування програмної системи з множини об’єктів, керованих повідомленнями 90

Завдання 90

Теоретичні положення 90

Методичні рекомендації 92

Вимоги щодо організації системи 102

Вибір варіанту завдання 103

Контрольні запитання 104

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ 105

ВСТУП

Дисципліна «Об'єктно-орієнтоване програмування» (ООП) відіграє важливу роль у галузі інженерії програмного забезпечення і є актуальною, у першу чергу, для фахівців-розробників програмних систем. Впродовж останніх двох десятиріч об'єктно-орієнтований підхід є домінуючим у сфері технологій програмування. Більшість відомих найпопулярніших мов програмування є об'єктно-орієнтованими. Головним призначенням даного навчального посібника є підтримка процесу формування та закріплення умінь та практичних навичок роботи з інструментальними засобами технологій розроблення програм та програмних систем.

Базовою мовою програмування обрано С++. Чому сам її? Як здається, у мові С++ найбільш повно підтримується ООП на відміну деяких інших мов програмування, у яких реалізація певних положень об'єктно-орієнтованого підходу може бути частковою. Другим аргументом на користь С++ є те, що для вивчення рекомендуються, у першу чергу, ті технології, які створюють виконувані файли програм у певному форматі машинного коду для процесорів. Це може сприяти закріпленню здобувачами знань з основ побудови комп’ютерних систем та набуттю умінь щодо розроблення програмного забезпечення до них на основі відповідних програмних інтерфейсів (API) та бібліотек.

Для виконання циклу робіт комп’ютерного лабораторного практикуму рекомендовано технологію Microsoft Visual Studio. У комп'ютерному класі вона розгорнута у операційному середовищі Windows. Але для виконання завдань та проведення експериментів студенти можуть використовувати власні комп'ютери вдома. Особливо це стосується організації занять дистанційно. За погодженням з викладачем студенти можуть використовувати й інші технологічні засоби створення програм. Так, зокрема, для вирішення завдань (з певною адаптацією цих завдань) можливо використати Qt C++, Swift, Android Studio тощо.

Для заочної форми навчання перелік обов’язкових до виконання робіт може бути скороченим до № 3, 4 та 5.

Навчальний посібник призначений для здобувачів ступеня бакалавра за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем»» спеціальності 121 «Інженерія програмного забезпечення». Може бути також корисним студентам, які вивчають програмування у курсах споріднених дисциплін відповідних спеціальностей.

Лабораторна робота №1. Знайомство із середовищем розробки програм Microsoft Visual Studio та складання модульних проєктів програм на C++

Мета: отримати перші навички створення програм для Windows на основі проєктів для Visual C++ з використанням Windows API і навчитися модульному програмуванню на C++

Завдання

1. Створити у середовищі MS Visual Studio C++ проєкт з ім'ям **Lab1**.
2. Написати вихідний текст програми згідно варіанту завдання.
3. Скомпілювати вихідний текст і отримати виконуваний файл програми.
4. Перевірити роботу програми. Налагодити програму.
5. Проаналізувати та прокоментувати результати та вихідний текст програми.

Теоретичні положення

Операційна система (ОС) Windows належіть до класу багатозадачних операційних середовищ, яка керує виконанням багатьох завантажених на виконання програмних застосунків з графічним інтерфейсом користувача (GUI – *graphical user interface*). Для взаємодії з програмними застосунками ця ОС активно використовує систему повідомлень.

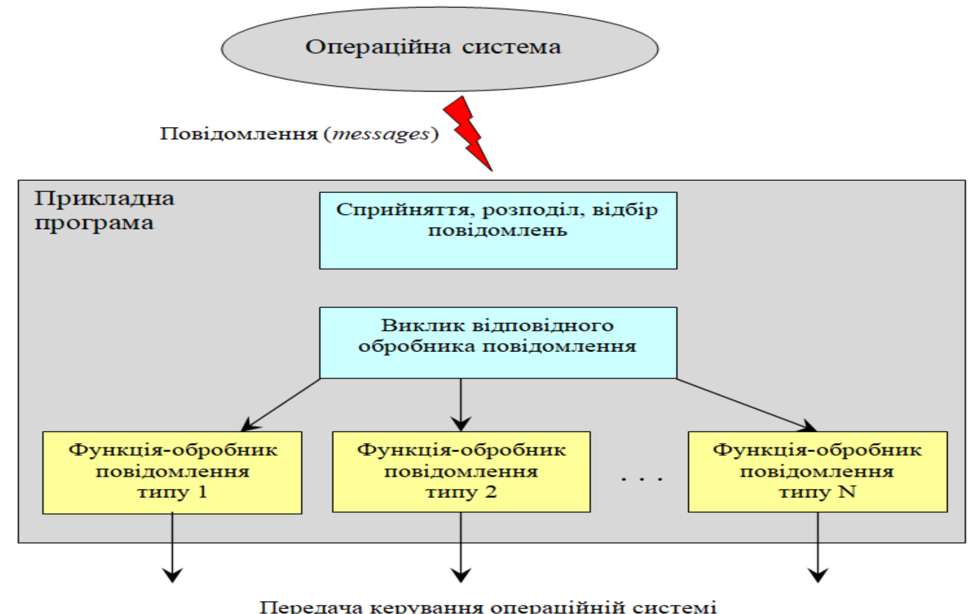

Рис. 1.1. Взаємодія ОС та програмного застосунку

Таким чином, програмний застосунок, керований повідомленнями (*message driven application*) повинен мати у себе програмний код обробників повідомлень.

Повідомлення від операційної системи можуть бути:

- про дії користувача;
- про події у системі;
- про завершення

Що таке повідомлення? В ОС Windows це безпосередній, або взятий з черги повідомлень виклик операційною системою функції вікна прикладної програми (CallBack-функції)

Функціональні можливості прикладної програми визначаються програмним кодом відповідних функцій-обробників повідомлень.

Розроблення програмних застосунків може бути на основі різноманітних бібліотек (фреймворків). Одним з різновидів є написання програмного коду на основі безпосереднього виклику функцій програмного інтерфейсу розробки застосунків операційної системи – API (*application program interface*).

Програмний код програмних застосунків зазвичай описується значним за обсягом вихідним текстом. Для того, щоб зменшити зусилля на розробку та виконати її якомога продуктивніше використовуються підходи щодо структурованості та модульності коду із забезпеченням можливостей повторного використання програмного коду. Конкретні рекомендації щодо цього наведені нижче.

Методичні рекомендації

Інсталяція Microsoft Visual Studio. Для виконання лабораторних робіт можна використовувати середовище розробки Microsoft Visual Studio починаючи з версії 2010. Але краще знайомитися з найсучаснішими версіями, якщо це дозволяють ресурси комп’ютера, який використовується для навчання.

Встановити на власному комп’ютері Microsoft Visual Studio можна з сайту корпорації Microsoft https://visualstudio.microsoft.com/

Створення в середовищі Microsoft Visual Studio нового проєкту C++

Далі розглянемо роботу із середовищем Microsoft Visual Studio на прикладі версії Visual Studio 2019 Community.

Для того, щоб розпочати створення нової програми, потрібно створити проєкт. Для цього на стартовій панелі обрати "Create a new project" (для англомовної версії Visual Studio)

Далі на панелі шаблонів проєктів C++ необхідно вибрати шаблон Windosw Desktop Application – Класичний застосунок Windows. Примітка. Класичним застосунком для операційної системи Windows вважається програмний додаток, який має головне вікно із заголовком та меню, які забезпечують графічний інтерфейс користувача.

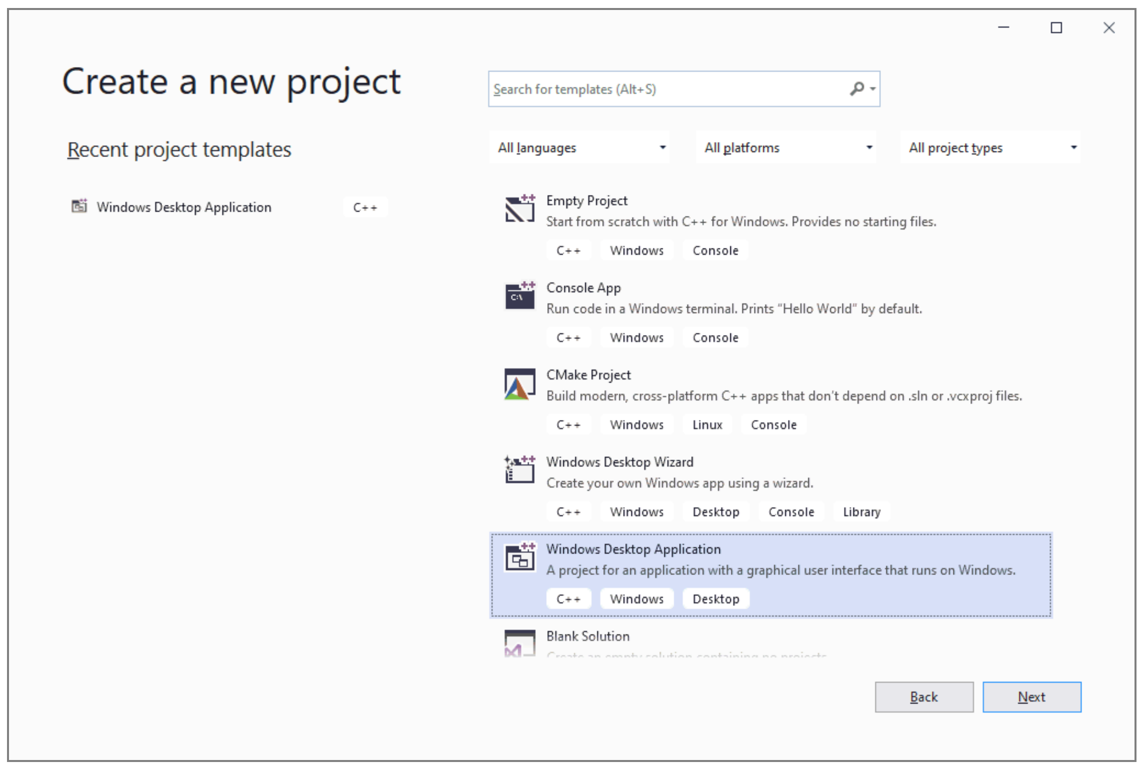

Рис. 1.1. Початок створення проєкту Visual Studio

Далі визначаємо назву проєкту, наприклад, MyProg і вказуємо його розташування на диску. Рекомендація: не записуйте проєкти у папки, які пропонуються за умовчанням (ProgramFiles), створіть для своїх проєктів окрему папку, наприклад, D:\Work.

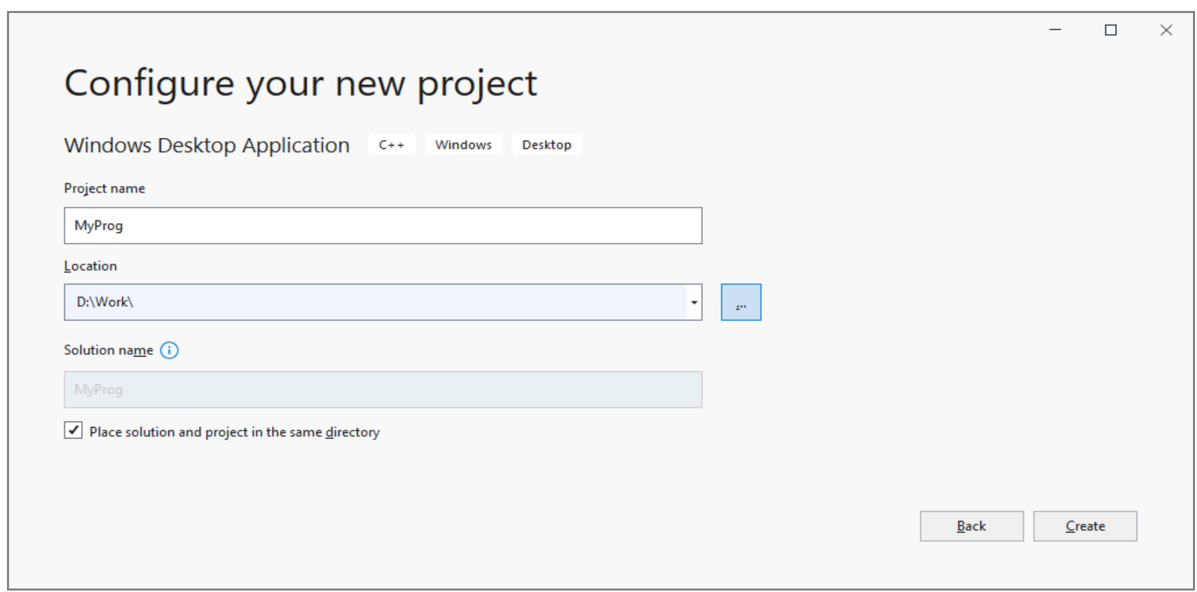

Рис. 1.2. Ім’я нового проєкту та його розташування

Натискуємо кнопку [Create]. Через декілька миттєвостей проєкт буде створений і середовище Visual Studio прийме наступний вигляд

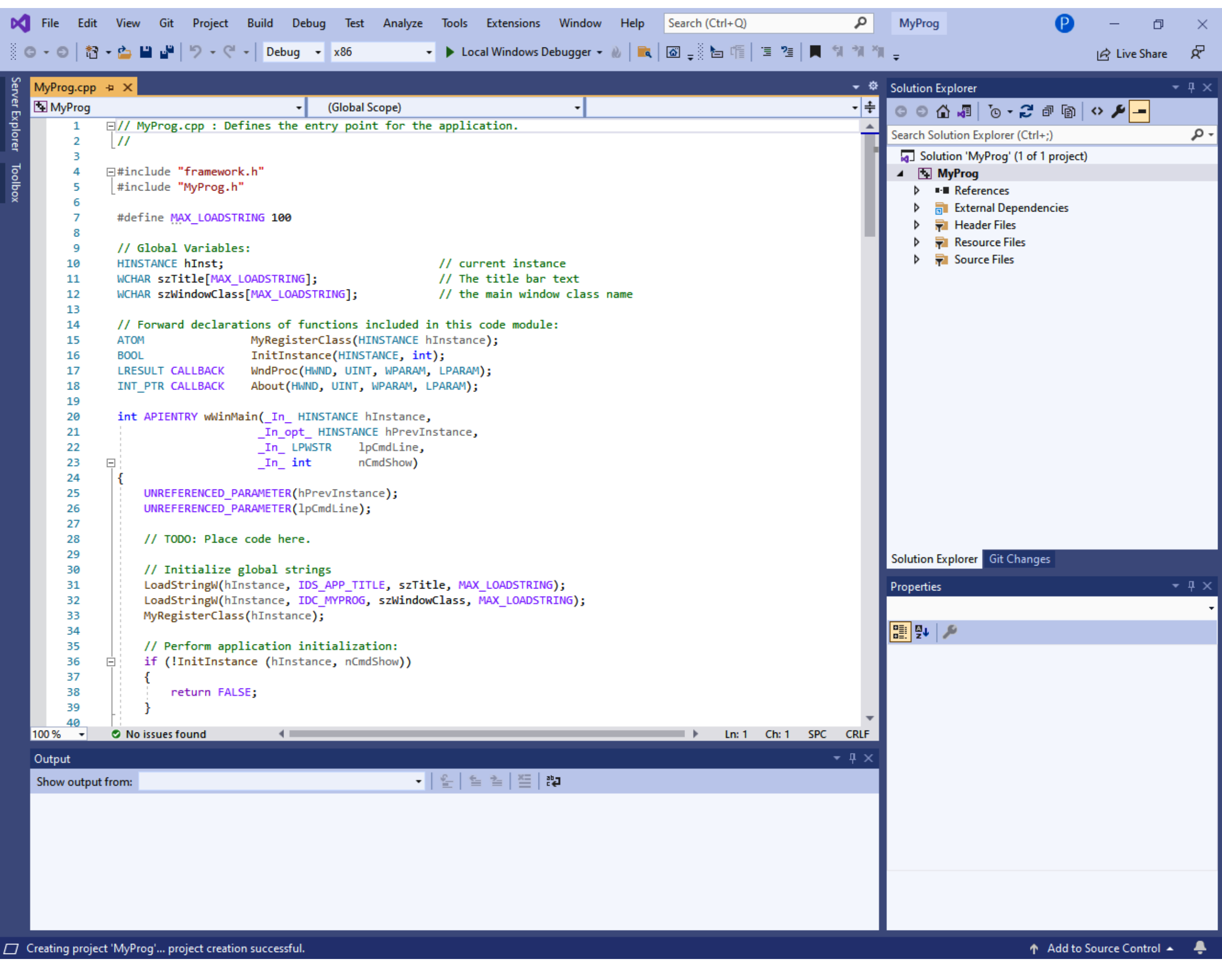

Рис. 1.3. Вигляд середовища розробки програми

У вікні праворуч "Solution Explorer" показуються назви файлів, які включені до проєкту. Головним файлом проєкту будемо вважати MyProg.cpp. Примітка. Якщо проєкт буде з ім’ям Lab1, то головний файл відповідно буде мати ім’я Lab1.cpp.

Тепер можна перевірити проєкт програми, який створюється від початку. Можна скомпілювати проєкт та викликати програму на виконання. Усе це можна зробити одним кроком. Для цього виберіть меню "Debug – Start Debugging" (або просто натиснути клавішу F5, або кнопку у вигляді трикутника). Розпочнеться компіляція, і далі, якщо помилок немає – лінкування. У разі успіху програма почне виконуватися – на екрані на тлі середовища Microsoft Visual Studio з’явиться вікно програми MyProg

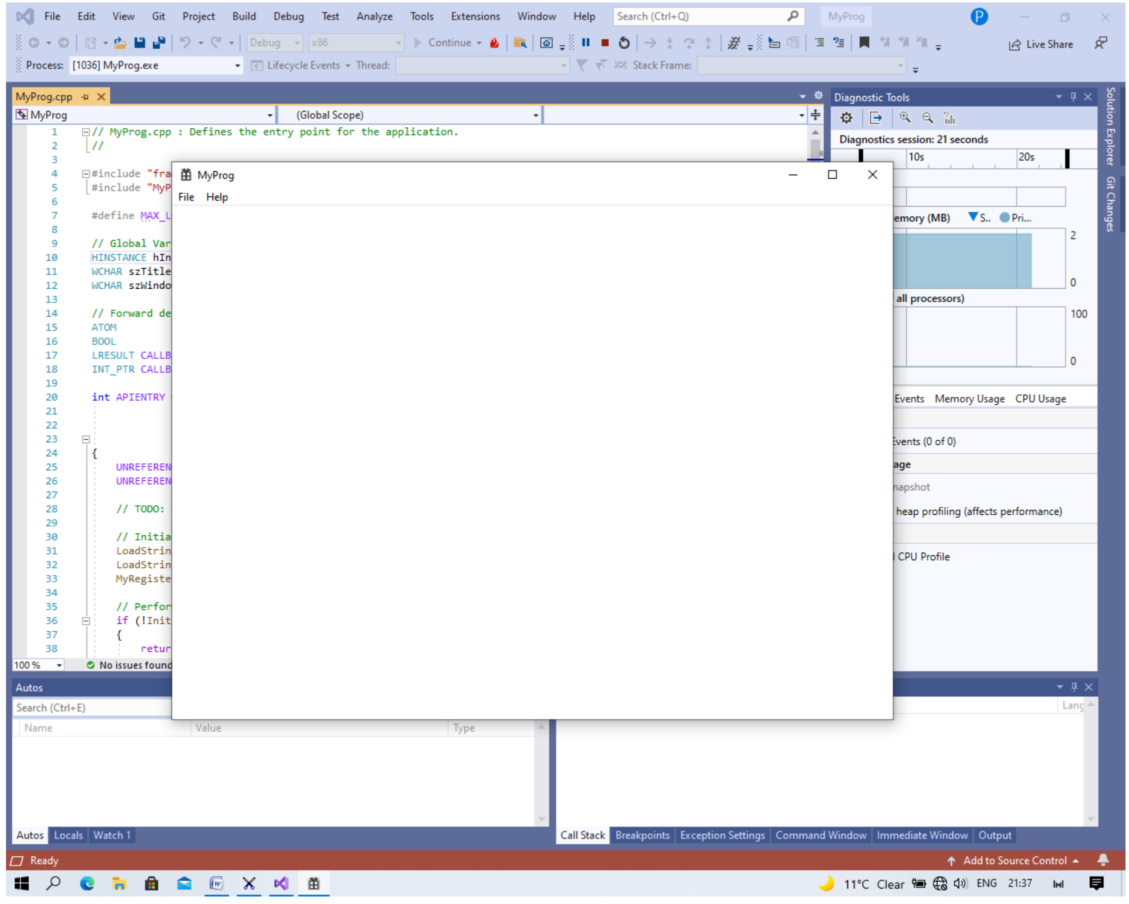

Рис. 1.4. Вигляд вікна програми MyProg на тлі середовища MS Visual Studio

У результаті отримуємо працюючу найпростішу програму, яка була автоматично запрограмована середовищем Visual Studio. Закриємо програму і перейдемо до наступного етапу роботи з проєктом.

Розгляд вихідного тексту програми. Вихідний текст складається з декількох файлів. Головний файл MyProg.cpp містить:

- перша частина – дві директиви #include та оголошення перемінних і функцій
- друга частина – визначення головної функції
- третя частина – Callback-функція головного вікна
- четверта частина – інші функції

Нижче наведені фрагменти тексту файлу MyProg.cpp

Перша частина головного файлу:

```
// MyProg.cpp : Defines the entry point for the application.
//

#include "framework.h"
#include "MyProg.h"

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                                // current instance
WCHAR szTitle[MAX_LOADSTRING];                  // The title bar text
WCHAR szWindowClass[MAX_LOADSTRING];            // the main window class name

// Forward declarations of functions included in this code module:
ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);
```

Друга частина. Варто звернути увагу на те, що у даному проєкті C++ головна функція зветься не **main** а **_tWinMain**. Ця головна функція є точкою входу і початок роботи програми повністю описується кодом цієї функції. Для будь-якої Windows-програми зазвичай спочатку реєструється клас вікна, потім це вікно створюється і далі програма входить у цикл очікування повідомлень. Подальша робота залежіть вже від того, які повідомлення будуть надходити на адресу головного вікна програми.

```
int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                     _In_opt_ HINSTANCE hPrevInstance,
                     _In_ LPWSTR    lpCmdLine,
                     _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    // TODO: Place code here.

    // Initialize global strings
    LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadStringW(hInstance, IDC_MYPROG, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);
```

```
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    HACCEL hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_MYPROG));
    MSG msg;

    // Main message loop:
    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return (int) msg.wParam;
}
```

Далі записаний текст функцій **MyRegisterClass** та **InitInstance**

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{
. . .
    wcex.lpfnWndProc  = WndProc;    //зверніть на це увагу – так передаємо адресу
                                    //функції вікна, щоб потім Windows її викликала
. . .

    return RegisterClassExW(&wcex);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
. . .
   HWND hWnd = CreateWindowW(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, nullptr, nullptr, hInstance, nullptr);
. . .
   return TRUE;
}
```

Третя частина тексту – визначення Callback-функції головного вікна програми. Коментарі, які автоматично записуються самим Visual Studio при створенні проєкту вельми доречні:

```
//  FUNCTION: WndProc(HWND, UINT, WPARAM, LPARAM)
//
//  PURPOSE: Processes messages for the main window.
//
//  WM_COMMAND  - process the application menu
//  WM_PAINT    - Paint the main window
//  WM_DESTROY  - post a quit message and return
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
```

```
    case WM_COMMAND:
        {
            int wmId = LOWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
            case IDM_ABOUT:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                break;
            case IDM_EXIT:
                DestroyWindow(hWnd);
                break;
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
            }
        }
        break;
    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code that uses hdc here...
            EndPaint(hWnd, &ps);
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Четверта частина тексту головного файлу – функція вікна ABOUT.

```
// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
    case WM_INITDIALOG:
        return (INT_PTR)TRUE;

    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR)TRUE;
        }
        break;
    }
    return (INT_PTR)FALSE;
}
```

Як і належіть функції вікна, вона має чотири аргументи і має модифікатор типу CALLBACK. Це означає, що цю функцію викликає Windows – надсилає повідомлення про події, які стосуються цього вікна. Повідомлення WM_INITDIALOG надсилається при створенні вікна – можна

запрограмувати обробник для ініціалізації елементів вікна (тут цього немає). Повідомлення WM_COMMAND для цього вікна означає реакцію на натискування кнопки OK, а також закриття вікна. Закривається вікно діалогу після виклику функції EndDialog(hDlg, . . .);

Додавання пунктів меню. Меню є елементом графічного інтерфейсу користувача (GUI) програм. Меню програми описується у файлі ресурсів MyProg.rc наступним чином:

```
IDC_MYPROG MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                IDM_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About ...",           IDM_ABOUT
    END
END
```

Щоб редагувати меню, зокрема, додати пункт меню, можна викликати редактор ресурсів. Для цього треба двічі клікнути у вікні "Solution Explorer” на файлі MyProg.rc

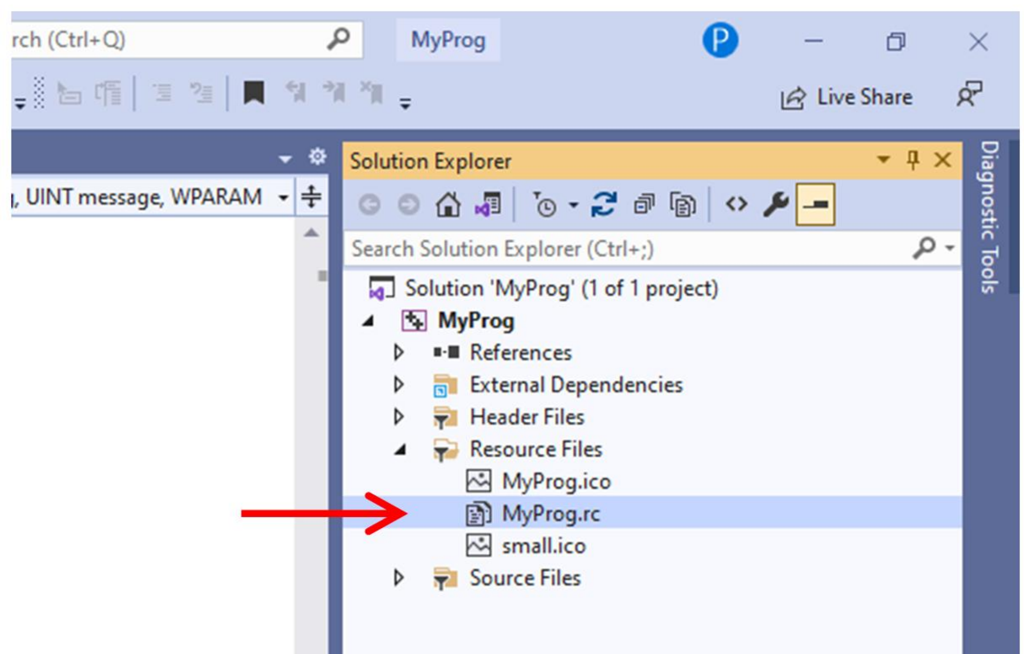

Рис. 1.5. Файл MyProg.rc у списку файлів проєкту

Після кліку на файлі MyProg.rc з'являється редактор ресурсів. У вікні списку ресурсів виберіть Menu і двічі клікніть – відкриється меню для редагування

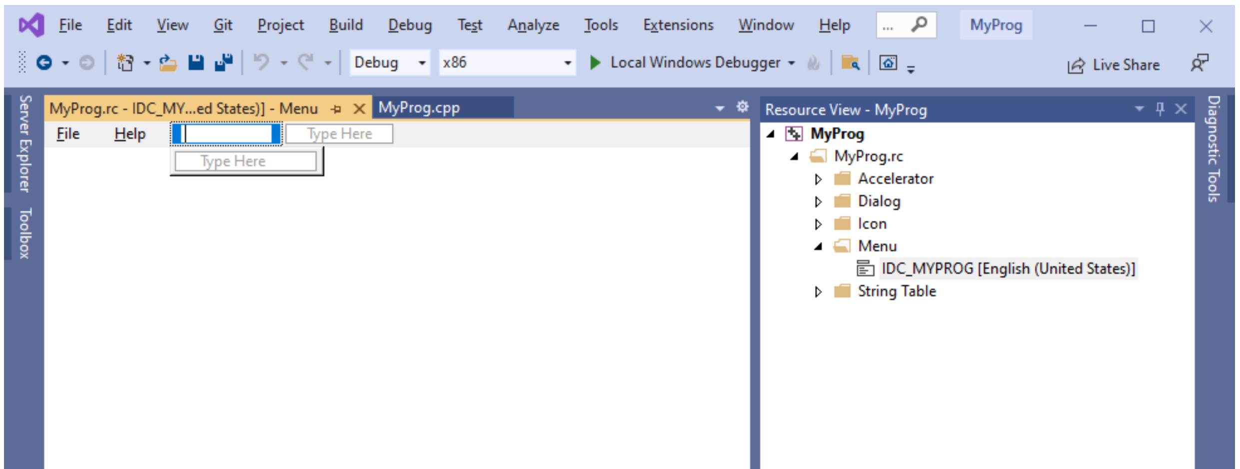

Рис. 1.6. Редактор ресурсів

Введіть назву нової групи пунктів меню (Popup menu), наприклад, "Actions", перетягніть її ліворуч пункту "Help". Далі в групі "Actions" можна внести пункти меню, наприклад, Work1 та Work2.

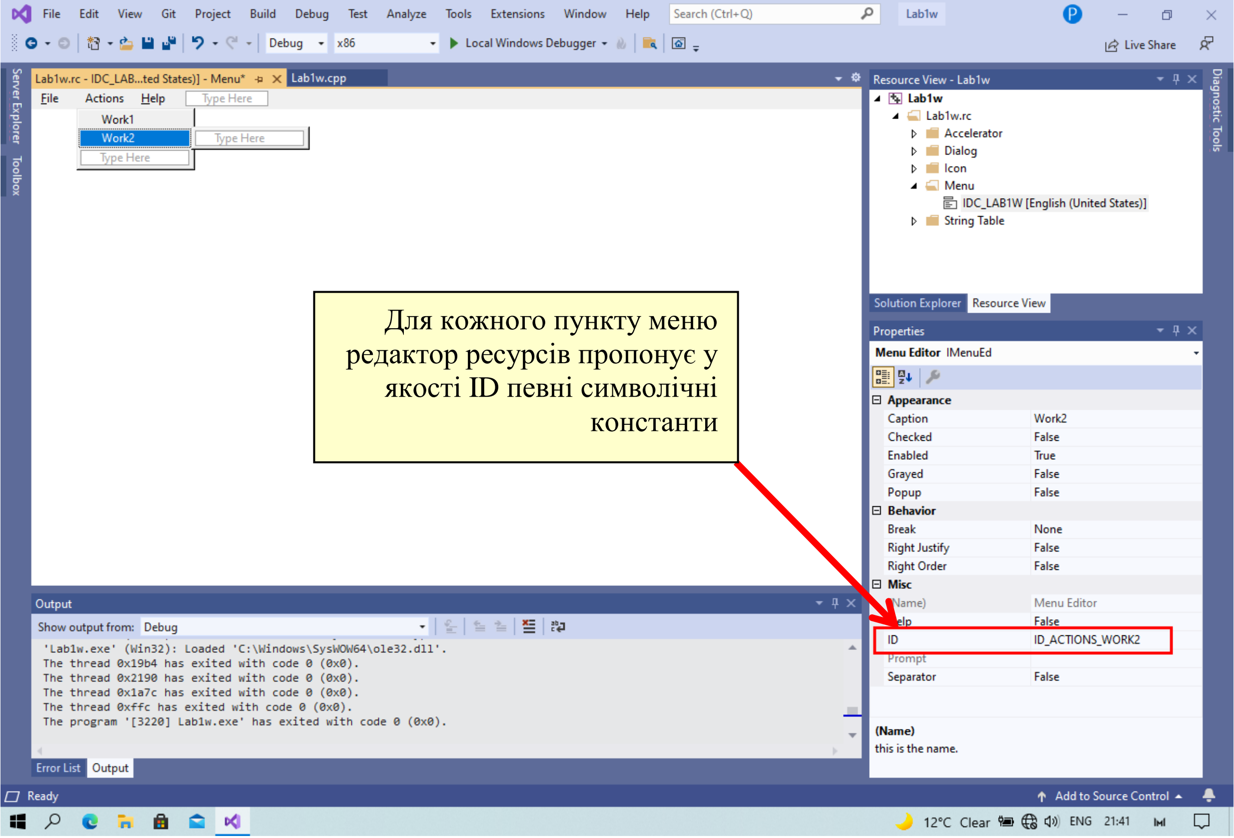

Рис. 1.7. Редактор ресурсів – створення пунктів меню

Символічні константи, які редактор ресурсів пропонує для кожного пункту меню, ми будемо використовувати у “case” обробників повідомлень у функції головного вікна. Визначення таких символічних констант можна

дозволити робити автоматично, але можна їхні імена та числові значення заповнювати власноруч. До відома – ці символічні константи Visual Studio автоматично записує у файл Resource.h.

Після того, як у редакторі ресурсів ми змінили меню далі перевіримо результати такої роботи – перекомпілюємо проєкт. Для цього достатньо натиснути F5. Після успішної компіляції, лінкування та запису exe-файлу наша програма почне виконуватися і у головному вікні програми мають відображатися потрібні пункти меню.

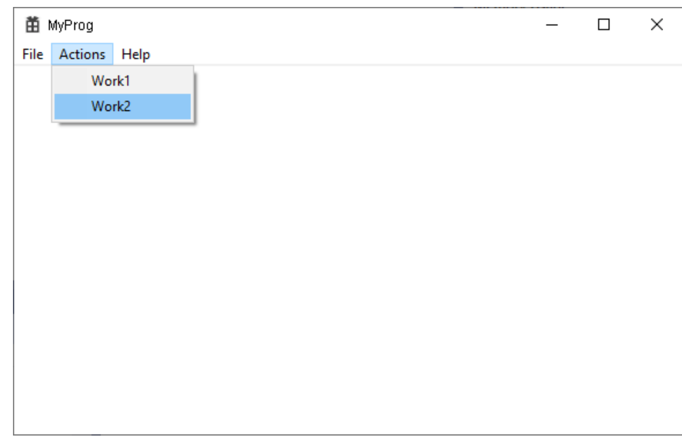

Рис. 1.8. Перевірка наявності нових пунктів меню

Новостворені пункти меню "Work1" або “Work2” можна клікнути, проте нічого не буде. Для того щоб при виборі меню щось виконувалося, потрібно це "щось" запрограмувати. Для цього треба додати у функцію WndProc для обробника повідомлення WM_COMMAND блоки case ID_ACTIONS_WORK1 та case ID_ACTIONS_WORK2:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_COMMAND:
        {
            int wmId = LOWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
            case ID_ACTIONS_WORK1:
                Work1(hWnd);
                break;
            case ID_ACTIONS_WORK2:
                Work2(hWnd);
                break;
            case IDM_ABOUT:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                break;
            case IDM_EXIT:
```

```
                DestroyWindow(hWnd);
                break;
```

Далі потрібно оголосити наші нові функції MyWork1 та MyWork2 у верхніх рядках тексту MyProg.cpp в списку оголошень функцій

```
ATOM   MyRegisterClass(HINSTANCE hInstance);
BOOL   InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

void MyWork1(HWND hWnd);      // оголошення наших функцій
void MyWork2(HWND hWnd);
```

Далі треба визначити функції MyWork1, MyWork2 – такі визначення найкраще записати наприкінці тексту MyProg.cpp

```
//функція-обробник пункту меню "Work1"
void MyWork1(HWND hWnd)
{
      //потрібний програмний код
}

//функція-обробник пункту меню "Work2"
void MyWork2(HWND hWnd)
{
      //потрібний програмний код робитися
}
```

У фігурних дужках для кожної функції запрограмуйте потрібний код згідно варіанту завдання.

Меню з пунктами у вигляді тексту можна вважати одним з найпростіших для програмістів елементів графічного інтерфейсу користувача. Значно складнішим для початківців може виглядати програмування вікон діалогу.

Як створити нове вікно діалогу? Для цього треба виконати наступні дії:

- у редакторі ресурсів додати нове вікно (Dialog) у файл ресурсів *.rc
- запрограмувати Callback-функцію вікна діалогу
- запрограмувати виклик вікна діалогу у функції MyWork1()

Візьміть до уваги, що у нашій програмі вже є одне вікно діалогу – ABOUT. Скористайтеся його текстом. Виклик вікна діалогу ABOUT робиться наступним чином

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
```

Для нового вікна зробіть так само, проте замість IDD_ABOUTBOX запишіть відповідний ID вікна діалогу, а замість About – ім'я Callback-функції для цього вікна діалогу.

Рекомендації щодо розробки вікон діалогу. Якщо необхідно запрограмувати вікно діалогу у окремому модулі, то спочатку потрібно скласти новий файл ресурсів, наприклад, **module1.rc**. Оскільки це текстовий файл, то теоретично, можливо це зробити у текстовому редакторі. Проте, набагато зручніше це робити у редакторів ресурсів. Як вже вказувалося вище, цей редактор ресурсів автоматично викликається після клику на файлі зі списку файлів ресурсів у вікні Solution Explorer.

У редакторі ресурсів можна розпочати створення нового вікна діалогу, визначити розміри вікна, накласти на поверхню цього вікна елементи діалогу – кнопки, стрічки вводу тексту, списки тощо.

Потім потрібно запрограмувати функції підтримки вікна діалогу у відповідному файлі модуля, наприклад, **module1.cpp**. Зокрема, треба запрограмувати Callback-функцію – обробник повідомлень для цього вікна.

Далі розглянемо деякі варіанти вікон діалогу.

Вікно діалогу зі стрічкою вводу тексту Edit Control. Нехай у редакторі ресурсів ми створили вікно діалогу зі стрічкою вводу тексту. Ця стрічка – елемент Edit Control, може мати ідентифікатор, наприклад, IDC_EDIT1.

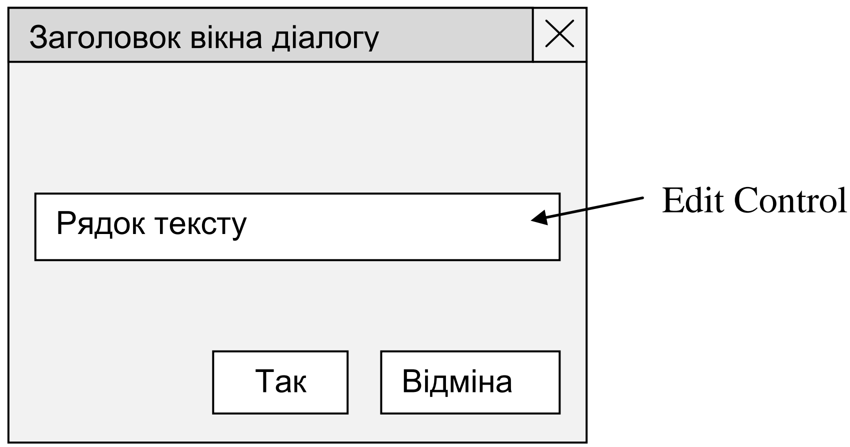

Рис. 1.9. Вікно діалогу зі стрічкою вводу тексту Edit Control

Потрібно запрограмувати, щоб після натискування кнопки Так (Ok) програма вихоплювала текст з цієї стрічки і записувала в якийсь буфер. Потім, після закриття вікна діалогу вміст цього буферу може бути переданий куди завгодно, у тому числі і зовні модуля **module1**.

Щоб вихопити текст з Edit Control можна скористатися функцією GetDlgItemText. Ця функція належіть до складу Windows API. Докладно про неї можна прочитати у довідці.

Фрагмент функції вікна діалогу:

```
//функція вікна діалогу
static BOOL CALLBACK Work1(HWND hDlg,UINT iMessage,WPARAM
wParam,LPARAM)
{
switch (iMessage)
  {
  case WM_COMMAND:
    if (LOWORD(wParam) == IDOK)      //якщо натиснуто кнопку Ok
       {
       //... можливо, щось ще
       //зчитуємо вміст елементу Edit Control у буфер
       GetDlgItemText(hDlg,IDC_EDIT1,буфер,кількість символів);
       //... можливо, щось ще
       EndDialog(hDlg, 1);
       break;
       }
    if (LOWORD(wParam) == IDCANCEL) EndDialog(hDlg, 0);
    break;
  default : break;
  }
return FALSE;
}
```

Вікно діалогу з окремим повзуном скролінгу – Horizontal Scroll Bar. У редакторі ресурсів на поверхню вікна додаємо такі елементи:

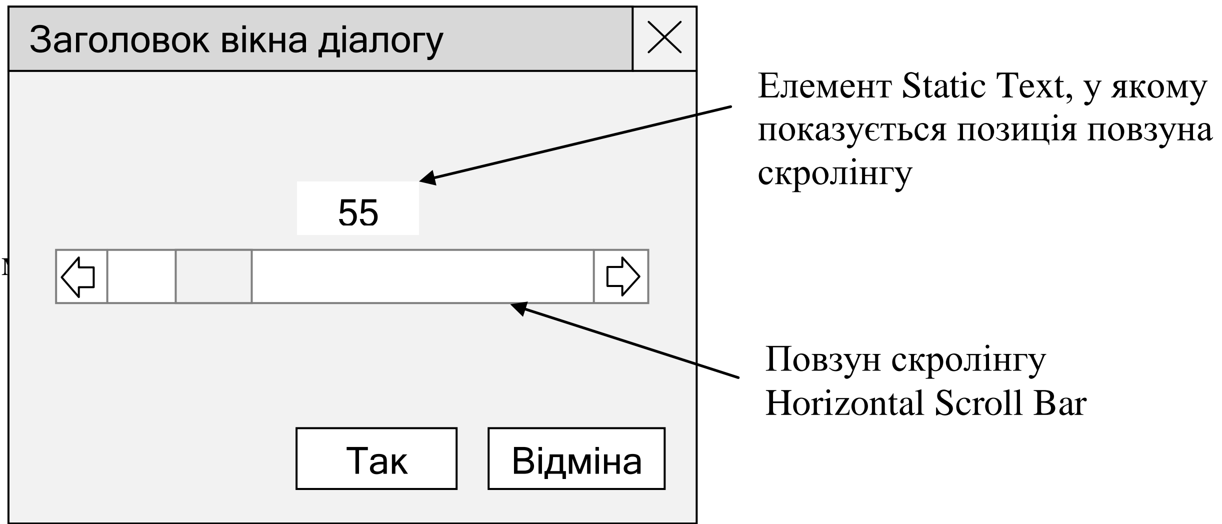

Рис. 1.9. Вікно діалогу з окремим повзуном скролінгу

Над повзуном скролінгу розташовуємо елемент для виводу тексту – Static Text, у який будемо записувати поточну позицію повзуна скролінгу.

Для того, щоб запрограмувати повзун скролінгу, у функції вікна потрібно передбачити обробник повідомлення WM_HSCROLL Спочатку потрібно задати потрібний діапазон значень. Для цього є функція SetScrollRange. Її краще викликати при обробці повідомлення WM_INITDIALOG, яке надсилається нашій програмі одразу після створення вікна діалогу. Щоб узнати поточну позицію повзуна, можна скористатися функцією GetScrollPos

```
pos = GetScrollPos(hWndScroll, SB_CTL);
```

Для того, щоб узнати handle повзуна скролінгу, можна використати функцію GetDlgItem

```
pos = GetScrollPos(GetDlgItem(hDlg, IDC_SCROLLBAR), SB_CTL);
```

Якщо користувач торкається повзуна скролінгу, то Windows надсилає на адресу функції вікна діалогу повідомлення WM_HSCROLL. Кожне повідомлення містить параметри. Для повідомлення WM_HSCROLL у параметр wParam записується код дії а також позиція повзуна. Нижче фрагмент функції вікна діалогу:

```
//Callback-функція вікна
static BOOL CALLBACK Work1(HWND hDlg,UINT iMessage,WPARAM
wParam,LPARAM)
{
switch (iMessage)
  {
  //...
  case WM_HSCROLL:
      pos = GetScrollPos(GetDlgItem(hDlg,IDC_SCROLLBAR),SB_CTL);
      switch (LOWORD(wParam))
        {
        case SB_LINELEFT:      //натиснуто кнопку ліворуч
            pos--;
            break;
        case SB_LINELEFT:      //натиснуто кнопку праворуч
            pos++;
            break;
        case SB_THUMBPOSITION: //фіксована позиція повзуна
        case SB_THUMBTRACK:    //поточна позиція повзуна
            pos = HIWORD(wParam);
            break;
        default : break;
        }
      //... потрібний код
```

```
      SetScrollPos(hWndScroll,SB_CTL,pos,TRUE);  //фіксація
                                               // позиції повзуна
      //... потрібний код
      break;
  //...
  default : break;
  }
return FALSE;
}
```

Коли користувач рухає повзун скролінгу, або натискує кнопки ліворуч або праворуч, то Windows показує відповідні дії. Але потрібну позицію повзуна треба ще зафіксувати – для цього призначена функція **SetScrollPos**.

Вікно діалогу зі списком ListBox. У редакторі ресурсів на поверхню вікна додаємо такі елементи:

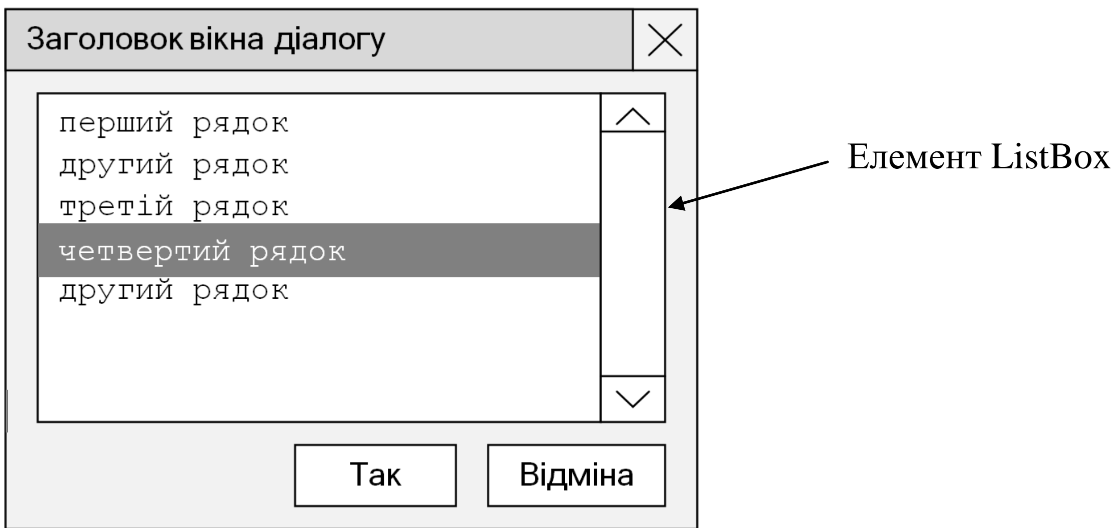

Рис. 1.10. Вікно діалогу зі списком ListBox

Вікно списку (ListBox) містить декілька рядків. Для того, щоб заповнити список потрібними рядками, можна викликати функцію

```
SendDlgItemMessage(hDlg, id, LB_ADDSTRING, 0, (LPARAM)text);
```

У параметр LPARAM записується вказівник на рядок тексту у вигляді або імені буфера тексту, або як символьна константа у подвійних лапках – "символи рядка" Ініціалізувати список, заповнювати його потрібними рядками зручно при обробці повідомлення WM_INITDIALOG, коли вікно діалогу починає активізуватися одразу після створення.

Для читання вибраного елемента списку у буфер можна використати функцію SendDlgItemMessage

```
SendDlgItemMessage(hDlg, id, LB_GETTEXT, indx, (long)buf);
```

Необхідно вказувати індекс потрібного елемента списку. Для того, щоб узнати індекс елемента, який вибрав користувач, можна викликати SendDlgItemMessage з параметром LB_GETCURSEL. Таким чином

```
indx = SendDlgItemMessage(hDlg, id, LB_GETCURSEL, 0, 0);
SendDlgItemMessage(hDlg, id, LB_GETTEXT, indx, (long)buf);
```

Щоб це виконувалося після натискування кнопки "Так", то необхідно запрограмувати відповідний обробник повідомлення WM_COMMAND у функції вікна. Нижче наведені фрагменти функції вікна діалогу, як приклад реалізації списку

```
//Callback-функція вікна діалогу
static BOOL CALLBACK Work1(HWND hDlg,UINT iMessage,WPARAM
wParam,LPARAM)
{
switch (iMessage)
  {
  case WM_INITDIALOG:
      SendDlgItemMessage(hDlg,id,LB_ADDSTRING,0, (LPARAM)str1);
      SendDlgItemMessage(hDlg,id,LB_ADDSTRING,0, (LPARAM)str2);
      SendDlgItemMessage(hDlg,id,LB_ADDSTRING,0, (LPARAM)str3);
      //...
      return (INT_PTR)TRUE;

  case WM_COMMAND:
      if (LOWORD(wParam) == IDOK)
         {
         //... можливо, щось ще
         //зчитуємо вибраний елементу списку у буфер
         indx = SendDlgItemMessage(hDlg, id, LB_GETCURSEL,0,0);
         SendDlgItemMessage(hDlg,id,LB_GETTEXT,indx,(long)buf);
         //... можливо, щось ще
         EndDialog(hDlg, 1);
         break;
         }
      if (LOWORD(wParam) == IDCANCEL) EndDialog(hDlg, 0);
      break;
  default : break;
  }
return FALSE;
}
```

Рекомендації щодо структурованості програмного коду

Навіть для достатньо простої програми вихідний текст може бути записаний занадто громіздким і заплутаним. Одним з факторів складносі сприйняття людиною вихідного тексту може стати текст Callback-функції вікна – як головного, так і вікна діалогу. Якщо вписувати програмний код усіх обробників повідомлень поміж case та break головного switch цієї функції, то це – поганий шлях. Для кожного з обробників можуть знадобитися локальні перемінні, масиви тощо. Функція вікна стає завеликою, буде мати сотні а то й тисячі рядків дещо хаотичного вихідного тексту.

```
//Callback-функція вікна
BOOL CALLBACK WndX(HWND hDlg,UINT iMessage,WPARAM wParam,LPARAM)
{
switch (iMessage)
  {
  case WM_INITDIALOG:
      //... програмний код з багатьох рядків
      return (INT_PTR)TRUE;
  case WM_HSCROLL:
      //... програмний код з багатьох рядків
      break;
  case WM_COMMAND:
      if (LOWORD(wParam) == IDOK)
         {
         //... програмний код з багатьох рядків
         break;
         }
      if (LOWORD(wParam) == IDCANCEL) EndDialog(hDlg, 0);
      break;
  default : break;
  }
return FALSE;
}
```

Приблизно такий самий вигляд має й функція головного вікна, де програмісти-початківці намагаються вписати багато рядків також, зокрема, в обробник повідомлення WM_PAINT.

Порада. Програмний код функції вікна треба розділити, структурувати шляхом оформлення обробника кожного повідомлення у вигляді окремої функції. Нижче наведений скелет структурованого програмного коду функції вікна діалогу

```
//Callback-функція вікна. Структурована версія коду
BOOL CALLBACK WndX(HWND hDlg,UINT iMessage,WPARAM wParam,LPARAM)
{
```

```
switch (iMessage)
  {
  case WM_INITDIALOG:
      return onInit(hDlg);

  case WM_HSCROLL:
      onHScroll(hDlg, wParam);
      break;

  case WM_COMMAND:
      if (LOWORD(wParam) == IDOK) onClickOk(hDlg);
      if (LOWORD(wParam) == IDCANCEL) EndDialog(hDlg, 0);
      break;
  default : break;
  }
return FALSE;
}
```

Тут кожний обробник повідомлення – це окрема функція **onXXX(..).** Програмний код визначення цих функцій міститься в окремій частині вихідного тексту

```
//Визначення функцій-обробників повідомлень
//Обробник повідомлення WM_INITDIALOG
BOOL onInit(HWND hDlg)
{
//... програмний код з багатьох рядків
return TRUE;
}

//Обробник повідомлення WM_HSCROLL
void onHScroll(HWND hDlg, WPARAM wParam)
{
//... програмний код з багатьох рядків
}

//Обробник натискування кнопки Ok
void onClickOk(HWND hDlg)
{
//... програмний код з багатьох рядків
EndDialog(hDlg, 1);
}
```

У подібний спосіб також рекомендується структурувати програмний код Callback-функції головного вікна – у тексті оператора switch для кожного “case“ повідомлень записувати виклик відповідної функції-обробника **onXXX(..).**

Потужним засобом структурованості є модульність, яка розглядається нижче.

Розробка модульного проєкту програми у середовищі Microsoft Visual Studio

Після того, як був створений проєкт C++ Visual Studio та перевірено роботу першої програми, можна продовжувати подальшу розробку. Далі розглянемо наступні теоретичні положення та методичні рекомендації.

Модульне програмування в C++

Загалом, весь вихідний текст програми може бути записаний в один файл *.cpp (за винятком файлу *.rc для проєктів Desktop Application). Таке можна уявити тільки для невеличких простих програм. При вдосконаленні, додаванні нових можливостей вихідний текст, як правило, зростатиме. Починаючи з деякої кількості рядків, у вихідному тексті буде важко розбиратися, аналізувати, робити виправлення.

Модульність є ефективним шляхом розробки складних програм. Вихідний текст розподіляється по модулям, наприклад, відповідно основним функціям. Засоби розробки програм на C++, такі як Visual Studio, дозволяють легко створювати програми з модулів, які можуть окремо компілюватися. Перевагою модульності є зменшення складності об'єктів компіляції, можливість прискорити розробку шляхом командної розробки – кожний модуль може розроблятися та налагоджуватися окремо різними програмістами.

Структура вихідного тексту модуля C++. Вихідний текст кожного модуля, наприклад, з ім'ям module, складається, як мінімум, з двох файлів:

1. Основний файл **module1.cpp**
2. Файл заголовку **module1.h**

Можна сказати, що це традиційно для мови C++

Для проєктів Visual Studio C++ Desktop Application додатково використовуються також й інші файли. Зокрема:

3. Файл **framework.h**
4. Файл ресурсів *.rc, наприклад, **module1.rc** (це необов'язково, тільки у випадку, коли модуль окремо репрезентує якийсь елемент інтерфейсу користувача, наприклад, вікно діалогу тощо)
5. Інші файли

1. Основний файл вихідного тексту модуля (**module1.cpp**) може містити оголошення-визначення перемінних, констант, визначення окремих функцій, визначення функцій-членів класів тощо. Приклад вмісту файлу **module1.cpp**:

```
#include "framework.h"
#include <math.h>
#include "module1.h"

//--оголошення-визначення глобальної перемінної--
int iResult_MOD1  = 0;

//--оголошення внутрішніх перемінних модуля--
static int i,n,size,counter;

//--оголошення внутрішніх функцій модуля—
static INT_PTR CALLBACK DlgProc_MOD1(HWND,UINT,WPARAM,LPARAM);
static void SomeFunc_MOD1(void);


//--визначення функції, яка експортується--
void MyFunc_MOD1(void)
{
. . .
}

//--визначення внутрішньої функції – функції вікна діалогу--
INT_PTR CALLBACK DlgProc_MOD1(HWND hDlg, UINT message,
                              WPARAM wParam, LPARAM lParam)
{
. . .
return 0;
}

//--визначення внутрішньої службової функції--
void SomeFunc_MOD1(void)
{
. . .
}
```

У наведеному вище прикладі перемінна **iResult_MOD1** та функція **MyFunc_MOD1()** будуть використовуватися в інших модулях. Це буде інтерфейс модуля. Крім того, у модулі є члени – перемінні та функції, які будуть використовуватися тільки для внутрішніх потреб модуля. Вони зовні невидимі, їхні імена недоступні для інших модулів програми – для цього вони оголошенні зі словом **static** попереду.

Файл module1.cpp потрібно включити у проєкт – він повинен бути у списку файлів Source Files вікна Solution Explorer. Тільки тоді цей файл буде компілюватися.

Примітка. У проєктах Visual Studio C++ Desktop Application на початку тексту файлів *.cpp повинен бути рядок

```
#include "framework.h"
```

2. Файл заголовку модуля (**module1.h**) містить оголошення елементів, які будуть загальнодоступними – для використання у інших модулях програми. Іншими словами, описують інтерфейс модуля. Такі елементи позначаються словом **`extern`**.

Приклад вмісту файлу **module1.h**

```
#ifndef _MODULE1_H_
#define _MODULE1_H_

#define STATUS_ERROR_MOD1 -1
#define STATUS_GOOD_MOD1 1

extern int iResult_MOD1;

extern void MyFunc_MOD1(void);

#endif
```

або

```
#pragma once

#define STATUS_ERROR_MOD1 -1
#define STATUS_GOOD_MOD1 1

extern int iResult_MOD1;

extern void MyFunc_MOD1(void);
```

Для того, щоб при включенні заголовочного файлу директивою **`#include`** уникнути помилок компіляції повторного визначення, зокрема символічних констант **`STATUS_ERROR_MOD1`** та **`STATUS_GOOD_MOD1`**, у файлі записано рядки:

```
#ifndef _MODULE1_H_
#define _MODULE1_H_
...

#endif
```

або

```
#pragma once

...
```

Рекомендується подібним чином оформлювати усі заголовочні файли *.h.

Файл module1.h включати у проєкт не обов'язково (він буде автоматично завантажуватися директивою **`#include`**), проте бажано – тоді він буде у списку файлів Header Files вікна Solution Explorer і його буде легко викликати для огляду.

Рекомендація. Щоб не заплутатися у іменах ідентифікаторів і не використовувати щось на кшалт **namespace**, рекомендується імена **extern** членів модулів робити унікальним. Наприклад, до кожного імені члена

модуля **module1** додавати наприкінці "_MOD1", для членів модуля **module2** додавати наприкінці "_MOD2" і так далі.

3. Файл ресурсів (**module1.rc**) містить опис елементів графічного інтерфейсу, наприклад, меню, вікон діалогу, та інших ресурсів. Структура файлу **module1.rc**:

```
#include "framework.h"
 . . .

IDD_MYDIALOG_MOD1 DIALOGEX . . .   //вікно діалогу
. . .  //опис елементів вікна діалогу

//інші ресурси – меню, курсори, іконки, рядки тексту тощо
```

Файл module1.rc потрібно включити у проєкт – він повинен бути у списку файлів Resource Files вікна Solution Explorer.

Примітка. Ідентифікатори елементів ресурсів, наприклад ід. вікон діалогу, є константами. Важливо те, що значення таких ід. повинні бути унікальними – не може співпадати з ід. інших відповідних ресурсів проєкту. Краще робити ід. у вигляді символічних констант, на кшалт **IDD_MYDIALOG_MOD1**, проте, тоді їх потрібно десь визначати. Визначення таких символічних констант можна зробити у окремому файлі *.h, а можливо, у файлі з іншим розширенням. Наприклад, якщо визначення **IDD_MYDIALOG_MOD1** зробити у файлі **module1.rh**

```
#pragma once

#define IDD_MYDIALOG_MOD1 1001
```

тоді у файлі **module1.rc** повинно бути #include "module1.rh"

```
#include "framework.h"
#include "module1.rh"

 . . .

IDD_MYDIALOG_MOD1 DIALOGEX . . .
. . .
```

а якщо у файлі **module1.cpp** також використовується ім’я **IDD_MYDIALOG_MOD1**, то у цьому файлі також повинно бути #include "module1.rh":

```
#include "framework.h"
#include <math.h>
#include "module1.h"
#include "module1.rh"
```

Як додати модуль в проєкт? Будемо вважати, що тексту модуля поки що немає і ми плануємо його писати безпосередньо у середовищі Visual Studio. Для того, щоб додати файл *.cpp модуля, вкажіть курсором розділ "Source Files", клацніть правою кнопкою миші, і потім – ”Add" ”New Item"

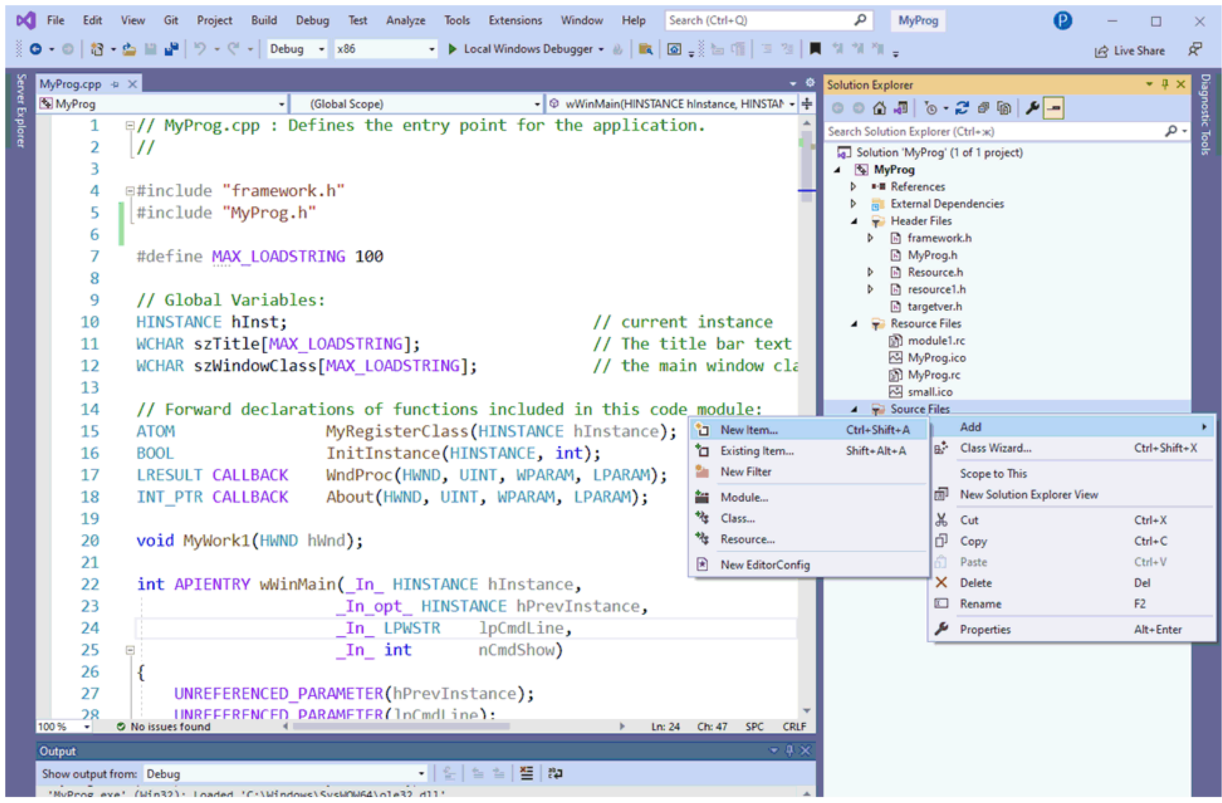

Рис. 1.11. Додавання файлу модуля у проєкт

Поняття #include – ієрархії файлів

Схема **#include – ієрархії файлів** описує відношення включення файлів проєкту директивами #include. У такій схемі вказуються файли, які містять директиви #include, а також файли, які включаються цими директивами.

На найнижчому рівні ієрархії розташовуються файли, які не містять #include. Наступний рівень посідають файли, у текстах яких записані #include файлів найнижчого рівня. І так далі. Така схема ілюструє залежності елементів проєкту (рис. 1.12).

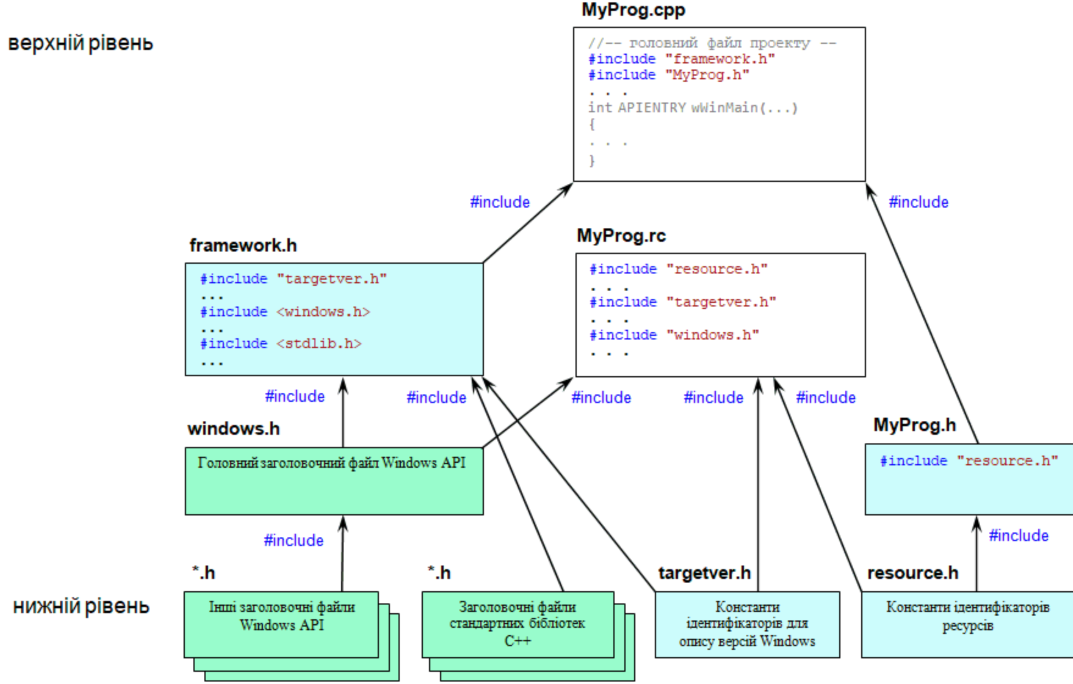

Рис. 1.12.Приклад схеми #include-ієрархії файлів проєкту

Поняття залежності модулів програми

Початковий проєкт C++, який автоматично створює Microsoft Visual Studio, не є модульним у контексті даної лабораторної роботи. І це незважаючи на те, що проєкт містить багато пов'язаних між собою файлів, як наведено вище. Тепер уявимо, що у проєкті є три модуля та головний файл програми. Відобразимо залежності модулів відповідно #include наступним чином:

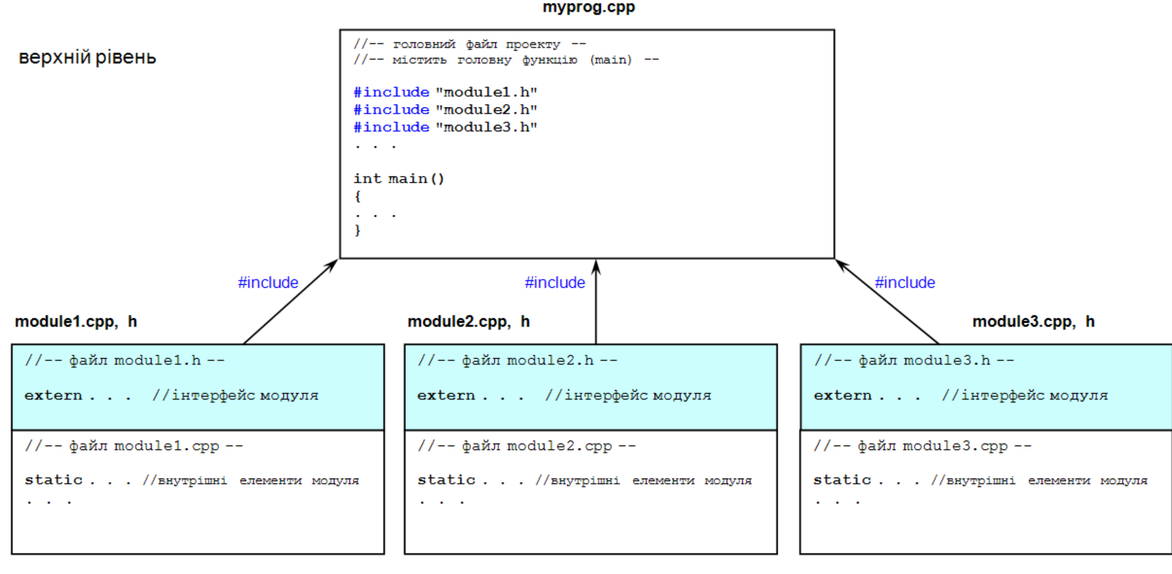

Рис. 1.13.Приклад схеми залежностей модулів

З цієї схеми видно, що три модуля – **module1, module2** та **module3** є незалежними один від одного. Елементи інтерфейсу кожного з модулів використовуються у головному файлі програми **myprog.cpp**. У якості елементів інтерфейсу зазвичай записуються функції – вони оперують схованими у модулях функціями та даними. Інтерфейсні функції призначені для виклику у інших модулях. Таким чином, у даному проєкті головний файл залежіть від вказаних модулів. Модулі можуть роздільно компілюватися – відповідно створюються об'єктні файли **module1.obj, module2.obj** та **module3.obj.**

У лабораторній роботі потрібно розробити декілька модулів, врахувавши специфіку програмування на основі Windows API.

Рекомендації щодо модуля, який показує вікно діалогу

Інтерфейс модуля (описується у файлі ***.h**) потрібно зробити у вигляді єдиної функції, наприклад

```
extern int Func_MOD1(HINSTANCE hInst, HWND hWnd);
```

Згідно завдань нашої лабораторної роботи, ця функція буде викликати єдине вікно діалогу, створення та підтримка якого буде запрограмоване у файлі ***.cpp** модуля. Наприклад, для першого модуля у файлі **module1.cpp** можна так:

```
//функція-оболонка створення вікна діалогу
int Func_MOD1(HINSTANCE hInst, HWND hWnd)
{
return DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, Work1);
}

//Callback-функція вікна
static BOOL CALLBACK Work1(HWND hDlg,UINT iMessage,WPARAM
wParam,LPARAM)
{
switch (iMessage)
  {
  case WM_INITDIALOG:
      //. . . ініціалізуємо елементи вікна діалогу (якщо потрібно)
      break;
  case WM_COMMAND:
      if (LOWORD(wParam) == IDOK)
         {
         //. . . зчитуємо вміст елеменітів вікна (якщо потрібно)
         EndDialog(hDlg, 1);
         break;
         }
```

```
      if (LOWORD(wParam) == IDCANCEL) EndDialog(hDlg, 0);
      break;
  default : break;
  }
return FALSE;
}
```

Функція **DialogBox** повертає значення, яке вказується другим параметром функції **EndDialog.** Наприклад, для того, щоб результат роботи **Func_MOD1** при натискуванні кнопки Cancel був 0, то відповідно у функції вікна записується **EndDialog(hDlg, 0)**.

Де взяти константу **IDD_DIALOG1**? Її можна визначити, наприклад, так

```
#define IDD_DIALOG1 1301
```

і записати це у окремому файлі **module1.rh**. Потім посилання на цей файл записати у файлах **module1.cpp** та **module1.rc** директивою #uinclude

```
#include "module1.rh"
```

Для того, щоб в інтерфейсній функції модуля здихатися параметра **hInst**, який потрібен для DialogBox, можна скористатися функцією **GetWindowLongPtr( )**, яка при параметрі GWLP_HINSTANCE повертає значення HINSTANCE.

Таким чином, визначення функції **Func_MOD1** може бути таким:

```
int Func_MOD1(HWND hWnd)
{
return DialogBox(GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
                 MAKEINTRESOURCE(IDD_DIALOG1),
                 hWnd,
                 Work1);
}
```

Можна уявити собі, що кожний модуль, який реалізує якесь вікно діалогу, буде запрограмований вже чотирма файлами: ***.cpp**, ***.h**, ***.rc**, ***.rh**. Тоді загальна структура проєкту, який складатиметься з двох модулів, буде такою:

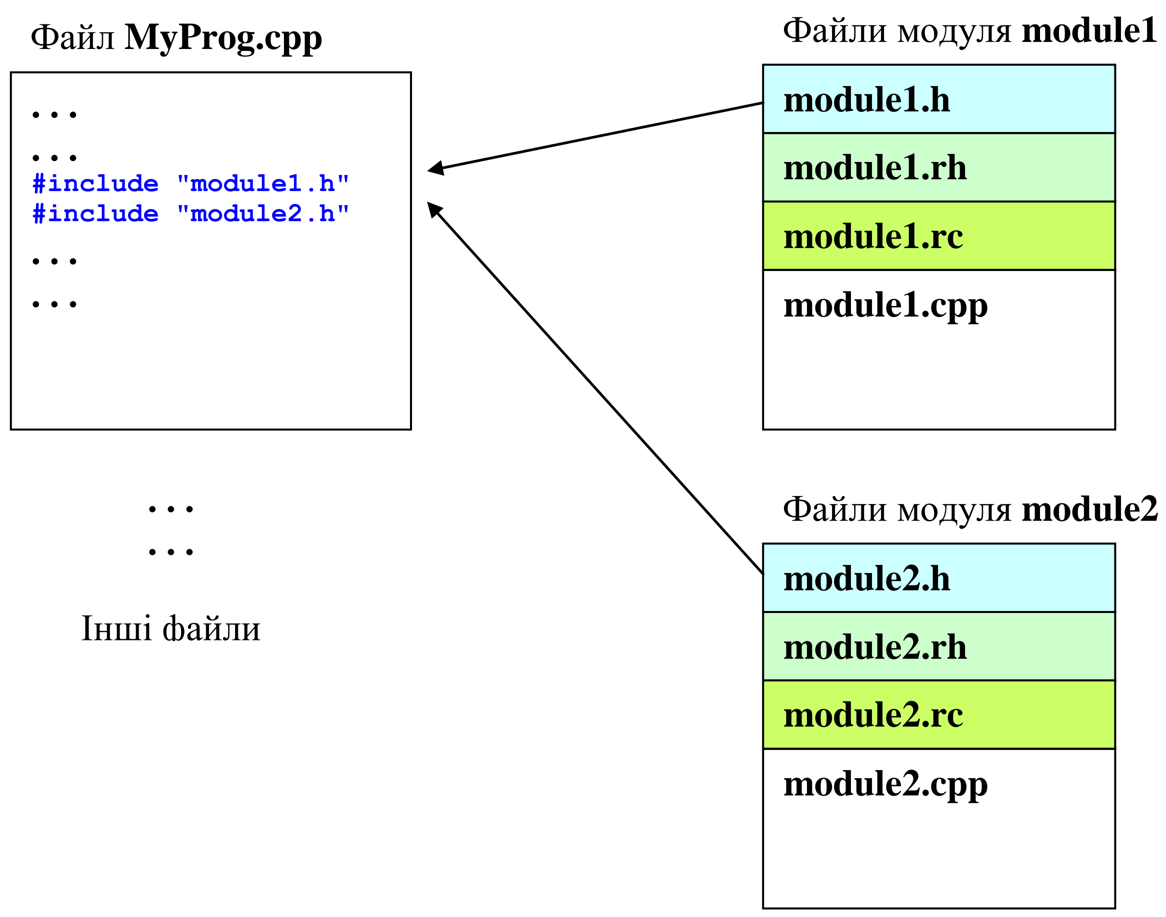

Рис. 1.14. Структура проєкту з двох модулів

Рекомендації щодо джерел інформації

Серед джерел інформації щодо розробки програм на основі інструментів Microsoft у першу чергу можна порекомендувати довідково-навчальну службу Microsoft, зокрема:

- Щодо інсталяції Visual Studio: https://learn.microsoft.com/en-us/cpp/build/vscpp-step-0-installation?view=msvc-170
- Щодо створення проектів C++: https://learn.microsoft.com/en-us/windows/win32/learnwin32/learn-to-program-for-windows?view=msvc-170
- Огляд можливостей програмування на C++ додатків для Windows: Overview of Windows Programming in C++ https://learn.microsoft.com/en-us/cpp/windows/overview-of-windows-programming-in-cpp?view=msvc-170

Варіанти завдань та основні вимоги

Для усіх варіантів необхідно створити **два пункти меню** – "Робота1" та "Робота2".

1. Потрібно, щоб при виборі пункту меню "Робота1" виконувалося щось згідно варіанту B_1, причому B_1 обчислюється за формулою

$$B_1 = Ж \bmod 4,$$

де Ж – номер студента у журналі, mod – залишок від ділення

2. Запрограмувати також, щоб при виборі пункту меню "Робота2" виконувалося щось згідно варіанту B_2:

$$B_2 = (Ж+1) \bmod 4$$

3. Вихідний текст функції B_1 запрограмувати в окремому модулі, а вихідний текст для функції B_2 – у іншому окремому модулі. Таким чином, проєкт повинен складатися з головного файлу Lab2.cpp та ще, як мінімум, двох файлів модулів з іменами module1.cpp та module2.cpp. Примітка: **кожне вікно діалогу повинно бути в окремому незалежному модулі**, тому якщо $B_1 = 2$, або $B_2 = 2$ то має бути, як мінімум, три модуля.

4. **Callback-функції вікон діалогу не мають бути видимими зовні модулів** – це внутрішні подробиці модулів, треба сховати їх, оголошуючи словом **static**. Також не можна виносити за межі модулів ресурси – опис вікон діалогу. Це забезпечується тим, що для кожного модуля **moduleX.cpp** має бути власний файл ресурсів **moduleX.rc**.

5. Інтерфейс модулів. Інтерфейс кожного модуля, наприклад, модуля **module1.cpp**, записується у відповідний файл заголовку **module1.h**. Інтерфейс модуля повинен бути у вигляді тільки одної функції

```
extern int Func_MOD1(HWND hWnd, . . . );
```

Ця функція має, як мінімум, один аргумент, через який у модуль буде передаватися handle головного вікна. Можуть бути ще інші аргументи – за необхідності. Інтерфейсна функція повинна повертати 0, якщо користувач натиснув у вікні діалогу кнопку "Відміна" ("Cancel"), або "×" у правому верхньому куті вікна. **Вимога: інших функції, перемінних у інтерфейсі модуля (у файлі .h) не повинно бути.**

6. Якщо виводиться наступне вікно діалогу, то попереднє вікно повинно закриватися

7. Намалювати повну #include-ієрархію усіх наявних файлів проєкту (враховувати тільки ті файли, які відображаються у вікні Solution Explorer). Неприпустимі перехресні #include-зв'язки між модулями.

8. Варіанти B1 та B2 вибираються згідно таблиці 1.1

Таблиця 1.1

Варіант	Що треба зробити	Підказка
0	Вікно діалогу для вводу тексту, яке має стрічку вводу (Edit Control) та дві кнопки: [Так] і [Відміна]. Якщо ввести рядок тексту і натиснути [Так], то у головному вікні повинен відображатися текст, що був введений.	Взяти від стрічки вводу рядок тексту – функція **GetDlgItemText** Вивести текст у головному вікні – функція **TextOut** в обробнику повідомлення WM_PAINT
1	Вікно діалогу з повзуном горизонтального скролінгу (Horizontal scroll Bar) та дві кнопки: [Так] і [Відміна]. Рухаючи повзунок скролінгу користувач вводить число у діапазоні від 1 до 100. Після натискування кнопки [Так] вибране число буде відображатися у головному вікні.	Обробка руху повзуна скролінгу – обробка повідомлення **WM_HSCROLL** у Callback-функції вікна діалогу. Вивести число у головному вікні – функція **TextOut** в обробнику повідомлення WM_PAINT. Щоб число вивести як текст – функція **itoa** або **ltoa** або інша функція, яка перетворює ціле число у рядок тексту.
2	Два вікна діалогу. Спочатку з'являється перше, яке має дві кнопки: [Далі >] і [Відміна]. Якщо натиснути кнопку [Далі >], то воно закриється і з'явиться друге длг вікно, яке має кнопки: [< Назад], [Так] і [Відміна]. Якщо натиснути кнопку [<Назад], вікно закриється і перехід до першого вікна.	Аналізувати результат роботи длг вікна можна по значенню, яке повертає DialogBox. Значення визначається другим аргументом функції EndDialog, яка розташовується у функції вікна. Можна, наприклад: - для кнопки [Далі >] робити виклик функції EndDialog(hDlg, 1) - для кнопки [Відміна] – EndDialog(hDlg, 0) - для кнопки [< Назад] – EndDialog(hDlg, -1)
3	Вікно діалогу з елементом списку (List Box) та двома кнопками: [Так] і [Відміна]. У список автоматично записуються назви груп нашого факультету. Якщо вибрати потрібний рядок списку і натиснути [Так], то у головному вікні повинен відображатися текст вибраного рядка списку.	Занесення рядка у список: **SendDlgItemMessage**(hDlg, id, LB_ADDSTRING, 0, (LPARAM)text); Читання елементу списку. Спочатку індекс рядка, що вибрано: indx = **SendDlgItemMessage**(hDlg, id, LB_GETCURSEL, 0, 0); а потім взяття indx-го рядка тексту: **SendDlgItemMessage**(hDlg, id, LB_GETTEXT, indx, (long)buf); Вивести текст у головному вікні – функцією **TextOut** в обробнику для WM_PAINT

Примітка. Незважаючі на однакові для деяких студентів варіанти завдань, їхні рішення повинні бути індивідуальними.

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний текст головного файлу .cpp (фрагменти, що ілюструють власний код),
4. та вихідні тексти модулів – файли .cpp, .h.
5. Ілюстрації (скріншоти)
6. Висновки

Контрольні запитання

1. Що виконує головна функція програми?
2. Чим відрізняється реєстрація вікна від його створення?
3. Що робить у Callback-функції вікна оператор switch?
4. Як створюється новий пункт меню?
5. Як викликається вікно діалогу?
6. Як створити нове вікно діалогу?
7. Де запрограмований цикл очікування повідомлень головного вікна?
8. Що таке WM_COMMAND?
9. Які файли потрібні для проєкту Visual Studio Desktop Application?
10. Що таке #include – ієрархія?
11. Що робить директива #include?
12. Чим відрізняється Callback-функція головного вікна від функції вікна діалогу?
13. Що означає роздільна компіляція модулів і як її досягти?

Лабораторна робота №2. Розробка графічного редактора об’єктів на С++

Мета: отримати вміння та навички використовувати інкапсуляцію, абстракцію типів, успадкування та поліморфізм на основі класів С++, запрограмувавши простий графічний редактор в об’єктно-орієнтованому стилі.

Завдання

1. Створити у середовищі MS Visual Studio C++ проект типу Windows Desktop Application з ім’ям **Lab2**.
2. Скомпілювати проект і отримати виконуваний файл програми.
3. Перевірити роботу програми. Налагодити програму.
4. Проаналізувати та прокоментувати результати та вихідний текст програми.
5. Оформити звіт.

Теоретичні положення

Об'єктно-орієнтована парадигма побудови систем передбачає визначення потрібних класів та реалізацію взаємодії об'єктів. Важливими моментами є інкапсуляція, абстракція типів, спадкування, поліморфізм.

Програмісту необхідно реалізувати

- оголошення класу та визначення його членів
- створення екземплярів об'єктів даного класу
- виклик функцій-членів та використання інших членів

Оголошення вводить новий тип із вказаним ім’ям. Цей тип створюється на основі вже відомих типів даних – елементи класу можуть мати стандартні типи, або типи, які створені програмістом до цього. Крім того, в оголошенні визначається обмеженість доступу до членів класу. Оголошення окремого класу мовою С++ має наступний формат

```
class Ім’я
{
private:
. . .           //список членів
protected:
. . .           //список членів
public:
. . .           //список членів
};
```

Три різновиди обмежень доступу до членів класу

- private означає, що члени класу доступні тільки усередині цього класу.
- protected: члени класу доступні у цьому класі та у похідних класах.
- public: члени класу доступні будь-де в області видимості даного класу. Такі члени зазвичай використовуються як інтерфейс до об’єктів.

Клас може містити наступні елементи (члени)

1. Дані-члени. Це набір елементів, можливо різних типів. Це можуть бути, наприклад, перемінні.

2. Функції-члени. Це функції, що маніпулюють даними-членами класу. Мають доступ до усіх даних-членів або функцій-членів, оголошеним усередині класу. Можуть бути віртуальними. Функції-члени часто звуть методами.

3. Конструктор. Конструктор автоматично викликається при створенні об’єкту типу клас. Якщо створюється динамічний (тимчасовий) об’єкт за допомогою оператора new, то конструктор викликається при виконанні цього оператора.

Конструктор має те саме ім'я, що і клас, для якого він визначений. Конструктор може мати аргументи. Конструктор не має типу – його не можна оголошувати навіть як void.

Конструктор може бути в private, protected або public частини класу.

Може бути визначений усередині або поза оголошенням класу (аналогічно функціям-членам).

Зазвичай конструктор використовується для початкової ініціалізації даних при створенні нового об'єкта.

4. Деструктор. Деструктор автоматично викликається при знищенні (або виході з області видимості) цього об’єкта. Якщо об’єкт був створений оператором new, то деструктор викликається при виконанні операції delete при знищенні цього динамічного об’єкта.

Має те ж ім'я, що і клас, але зі знаком ~ (тильда) попереду.

Деструктор зазвичай використовується для дій, що супроводжують знищення об'єкта.

Успадкування класів

Від класів C++ можна створювати класи-спадкоємці. Можна сказати, що об’єкт А, властивості якого описуються класом ClassA, може мати спадкоємця – об’єкт В, який має властивості об’єкта А, плюс деякі інші, індивідуальні властивості, які описуються класом-спадкоємцем ClassB

```
class ClassA
{
. . .  //члени класу – властивості об’єкта А
};

//--оголошення класу-спадкоємця--
class ClassB : public ClassA
{
. . .  //члени цього класу – індивідуальні властивості об’єкта В
};
```

У класі ClassB будуть міститися усі члени класу ClassA плюс члени класу ClassB. Клас ClassA зветься базовим класом, а класи-спадкоємці звуть ще похідними класами.

Поліморфізм

Поліморфізм – це властивість, яка дозволяє одне ім’я використовувати для виконання різних дій. В загальному сенсі, основна ідея поліморфізму – "один інтерфейс, багато методів". Це означає, що базовий клас визначає загальний інтерфейс для будь-якого об’єкту, але дозволяє у похідних класах конкретизувати роботу методів цього інтерфейсу.

Використовуючи одне ім’я метода базового класу ми можемо викликати потрібний однойменний метод будь-якого похідного класу, в залежності від об’єкту, який створено. Вибір метода робиться на етапі виконання програми. Селектором є значення вказівника на клас. Поліморфний метод повинен бути запрограмований у вигляді віртуальної функції-члена базового класу.

```
ClassBase *p;        //вказівник на базовий клас
p = new DerivedClass;
p->func( );          //поліморфний виклик функції-члена
```

Важливим аспектом поліморфізму є те, що він сприяє зменшенню складності програм.

Методичні рекомендації

Загальна структура програми

Програма повинна бути модульною. Вихідний текст складається з головного файлу Lab2.cpp, модуля shape_editor та інших модулів та файлів проекту. Модулі, зокрема, shape_editor, повинні роздільно компілюватися у проекті.

У модулі **shape_editor** міститься клас **ShapeObjectsEditor**. Інтерфейс модуля у вигляді оголошення класу **ShapeObjectsEditor** повинен міститися у файлі **shape_editor.h**

```
class ShapeObjectsEditor
{
private:
  . . .
public:
   ShapeObjectsEditor(void);
  ~ShapeObjectsEditor();
   void StartPointEditor(...);
   void StartLineEditor(...);
   void StartRectEditor(...);
   void StartEllipseEditor(...);
   void OnLBdown(HWND);
   void OnLBup(HWND);
   void OnMouseMove(HWND);
   void OnPaint(HWND);
   void OnInitMenuPopup(HWND,WPARAM);  //відповідно варіанту
};                                     //завдання
```

Примітки. Позначкою ". . ." вказане те, що програміст запише для вирішення завдання. Метод `OnInitMenuPopup` потрібен лише для окремих варіантів завдань.

Визначення методів (функцій-членів) класу має бути у файлі **shape_editor.cpp.** Методи класу `ShapeObjectsEditor` є обробниками основних повідомлень, важливих для редагування. Їхні виклики у тілі функції головного вікна програми – у файлі Lab2.cpp

```
//---файл Lab2.cpp---
#include "shape_editor.h"

ShapeObjectsEditor Ім’я;    //для варіанту статичного екземпляру
                            //об'єкта редактора

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam)
{
int wmId, wmEvent;
```

```
switch (message)
  {
  case WM_LBUTTONDOWN:      //натиснуто ліву кнопку миші
    Ім'я.OnLBdown(hWnd);
    break;
  case WM_LBUTTONUP:        //відпущено ліву кнопку миші
    Ім'я.OnLBup(hWnd);
    break;
  case WM_MOUSEMOVE:        //пересунуто мишу у клієнтській частині вікна
    Ім'я.OnMouseMove(hWnd);
    break;
  case WM_PAINT:            //потрібно оновлення зображення вікна
    Ім'я.OnPaint(hWnd);
    break;
  case WM_INITMENUPOPUP:   //позначка пунктів меню - якщо потрібно
    Ім'я.OnInitMenuPopup(hWnd,wParam);
    break;

  case WM_COMMAND:
    wmId    = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    // Parse the menu selections:
    switch (wmId)
      {
      case IDM_POINT:
        Ім'я.StartPointEditor(...);  //початок вводу точкових об'єктів
        break;
      case IDM_LINE:
        Ім'я.StartLineEditor(...);   //початок вводу об'єктів-ліній
        break;
      case IDM_RECT:
        Ім'я.StartRectEditor(...);   //початок вводу прямокутників
        break;
      case IDM_ELLIPSE:
        Ім'я.StartEllipseEditor(...);      //початок вводу еліпсів
        break;
```

Нижче вигляд головного вікна програми

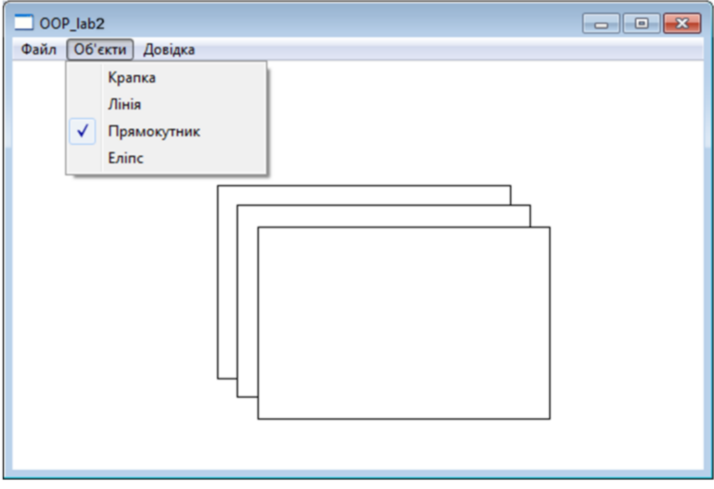

Рис. 2.1 . Вигляд головного вікна програми

Загальну структуру програми можна відобразити наступним чином:

Рис. 2.2. Діаграма залежностей файлів та модулів (include-ієрархія)

Базовий клас Shape та його спадкоємці

Ієрархія графічних об’єктів у цій програмі починається з базового абстрактного класу **Shape**:

```
//---оголошення класу---
class Shape
{
protected:
   long xs1,ys1,xs2,ys2;
public:
   void Set(long x1,long y1,long x2,long y2);
   virtual void Show(HDC) = 0;
};

//---визначення методу Set---
void Shape::Set(long x1,long y1,long x2,long y2)
{
xs1 = x1;
ys1 = y1;
xs2 = x2;
ys2 = y2;
}
```

Від класу **Shape** утворюються похідні класи: **PointShape**, **LineShape**, **RectShape**, **EllipseShape.** У кожному похідному класі визначається метод показу **Show** відповідно до геометрії об’єкта.

У класі **PointShape** кожна точка малюється викликом функції
```
SetPixel(hdc,x,y,колір);
```

У класі **LineShape** відрізок прямої лінії малюється викликом функцій
```
MoveToEx(hdc,x1,y1,NULL);
LineTo(hdc,x2,y2);
```

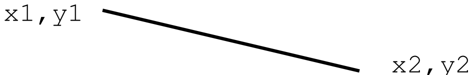

Рис. 2.3. Відрізок лінії та його координати

У класі **RectShape** прямокутник малюється викликом функції
`Rectangle(hdc,x1,y1,x2,y2);` (прямокутник із заповненням)
або чотирма лініями (якщо потрібен тільки контур).

x1,y1

x2,y2

Рис. 2.4. Прямокутник визначено координатами двох кутів

У класі **EllipseShape** еліпс малюється викликом функції
`Ellipse(hdc,x1,y1,x2,y2);` (еліпс із заповненням)
або `Arc(hdc, x1,y1, x2,y2, 0,0,0,0);` (якщо потрібен тільки контур).

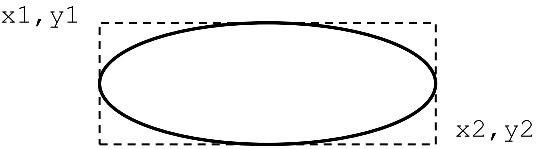

Рис. 2.5. Еліпс визначено кооординатами охопиючого прямокутника

Малювання фігур із заповненням

Функції `Rectangle` та `Ellipse` малюють фігури із заповненням. За умовчанням контур – тонка чорна лінія, заповненя – суцільне білого кольору. Визначення кольору заповнення можна запрограмувати у такий спосіб. Спочатку потрібно створити пензль та помістити його у контекст пристрою (hdc). Далі виклик функцій малювання потрібних одної або декількох фігур.

Після того, як такий пензль вже більше не потрібен, або треба створити новий, то спочатку вибирається попередній пензль, а потім створений знищується (інакше будуть витоки пам'ті).

```
BRUSH hBrush,hBrushOld;

hBrush = (HBRUSH)CreateSolidBrush(colorfill);  //новий пензль
hBrushOld = (HBRUSH)SelectObject(hdc, hBrush); //пензль -> у hdc

Ellipse(hdc,x1,y1,x2,y2);       //або Rectangle

SelectObject(hdc, hBrushOld);  //відновлюється пензль-попередник
DeleteObject(hBrush);    //створений пензль знищується
```

Аргументом функції **CreateSolidBrush** є ціле число, яке визначає колір заповнення. У якості аргументу можна рекомендувати макрос RGB, наприклад, для жовтого кольру заповнення:

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

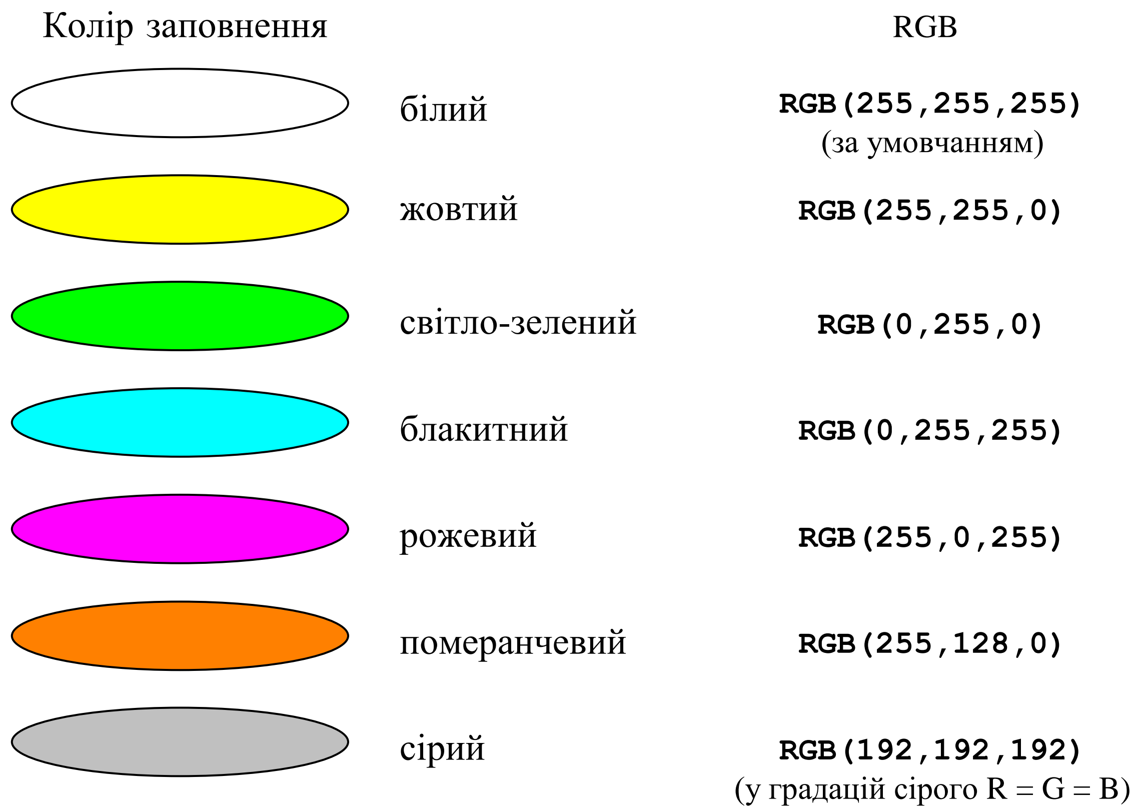

Рис. 2.6. Приклади визначення кольорів

Діаграма класів. Основні поняття

Діаграма класів показує наявні класи та їхню ієрархію спадкування. Кожний клас відображається прямокутником, у якому записане ім'я класу. Також у цьому прямокутнику записуються імена членів класу. Можна позначати окремо дані-члени та функції-члени (методи). Також можна якось позначати різновиди оголошення членів – чи то private, чи то protected або public.

Рис. 2.7. Клас та його члени на діаграмі класів

У найпростішому випадку, коли клас усього один, у діаграмі особливого сенсу немає. Діаграма корисна для показу двох та більше класів – тоді вона відображатиме взаємини класів. Наприклад, так

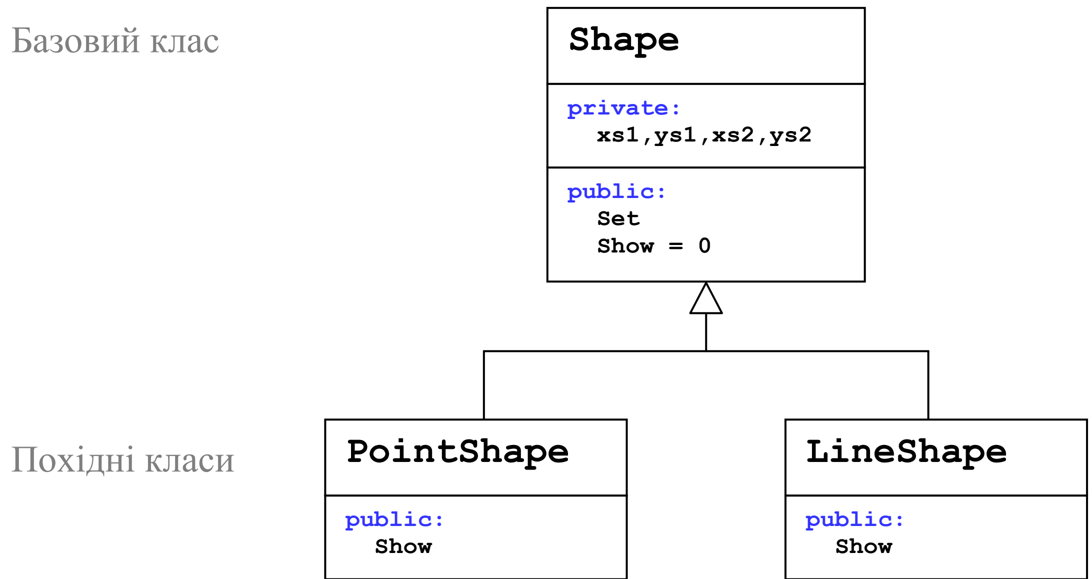

Рис. 2.8. Відображення залежностей класів на діаграмі класів

Існують багато способів відображення таких діаграм. Дуже популярною є нотація UML. Дещо інший, але подібний вигляд мають діаграми класів, які можна створити засобами Microsoft Visual Studio. У середовищі Visual Studio C++ діаграма створюється напівавтоматично. Спочатку потрібно створити і додати у проект новий елемент – "Схема класів" або "Class Diagram". А потім за допомогою панелі інструментів

(toolbox) вставити потрібні елементи на діаграму. Як саме - це може відрізнятися для різних версій Visual Studio. Зазвичай це достатньо легко. Розберіться у цьому самостійно.

Базовий абстрактний клас Editor та його спадкоємці

Ієрархія класів редагування об'єктів різних типів у Лаб2 повинна розпочинатися з абстрактного базового класу **Editor**.

```
class Editor
{
public:          //а, можливо, ще потрібен віртуальний деструктор?
  virtual void OnLBdown(HWND) = 0;
  virtual void OnLBup(HWND) = 0;
  virtual void OnMouseMove(HWND) = 0;
  virtual void OnPaint(HWND) = 0;
};
```

Навіщо потрібен цей базовий абстрактний клас?

По-перше, у ньому фіксуються імена та склад інтерфейсних методів, які необхідно реалізувати. Наприклад, метод **OnLBdown** буде викликатися для обробки повідомлення **WM_LBUTTONDOWN**, метод **OnPaint –** для обробки повідомлення **WM_PAINT**, і тому подібне.

По-друге, від базового класу буде утворена ієрархія класів-спадкоємців, які будуть описувати конкретні дії редагування для кожного типу об'єктів.

По-третє, можна запрограмувати поліморфізм виклику наведених вище інтерфейсних методів класу **Editor** по вказівнику на цей клас.

Примітка. Оскільки в деяких варіантах завдань потрібно ще й обробляти повідомлення **WM_INITMENUPOPUP**, то можливо додати відповідний інтерфейсний метод **OnInitMenuPopup** – проте тут **вимога: базовий клас Editor не змінювати,** а цей метод додати у похідний клас **ShapeEditor** і зробити вказівник вже на похідний клас **ShapeEditor**, вважаючи його базовим для забезпечення поліморфізму. Крім того, мабуть корисно такий похідний клас **ShapeEditor** зробити базовим і тому, що в ньому можна зосередити корисні члени, які враховують специфіку Windows-програм.

```
class ShapeEditor : public Editor
{
protected:
   . . .    //корисні члени, які враховують специфіку Windows-програм
public:
```

```
   ShapeEditor(void);
   void OnLBdown(HWND);
   void OnLBup(HWND);
   void OnMouseMove(HWND);
   void OnPaint(HWND);
   void OnInitMenuPopup(HWND,WPARAM);  //додатковий інтерфейсний метод
};
```

Від класу **ShapeEditor** треба утворити ієрархію похідних класів для усіх потрібних типів об'єктів. Ці класи своїми методами будуть описувати дії, які потрібно виконувати при вводі об'єктів відповідних типів. Так, наприклад, для вводу точкового об’єкту достатньо клікнути лівою кнопкою миші у потрібній точці вікна. А при вводі прямокутника спочатку визначаємо один кут, натискуючи кнопку миші (**OnLBdown**), далі пересуваємо курсор в точку іншого кута (**OnMouseMove**) і відпускаємо кнопку (**OnLBup**). Враховуючи це, створимо похідні класи для усіх потрібних типів об'єктів

```
class PointEditor : public ShapeEditor
{
. . .
};

class LineEditor : public ShapeEditor
{
. . .
};

class RectEditor : public ShapeEditor
{
. . .
};

class EllipseEditor : public ShapeEditor
{
. . .
};
```

Оголосимо вказівник на базовий клас **ShapeEditor**

```
ShapeEditor  *pse = NULL;
```

Тоді обробники потрібних повідомлень можна реалізовувати як поліморфні виклики методів **OnLBdown, OnPaint** та інших:

```
switch (message)
   {
   case WM_LBUTTONDOWN:
     if (pse) pse->OnLBdown(hWnd);  //поліморфний виклик метода onLBDown
     break;
   case WM_PAINT:
     if (pse) pse->OnPaint(hWnd);   //поліморфний виклик метода onPaint
```

```
     break;
   . . .                              //інші
   case WM_COMMAND:
     wmId    = LOWORD(wParam);
     wmEvent = HIWORD(wParam);
     switch (wmId)
       {
       case IDM_POINT:
         if (pse) delete pse;
         pse = new PointEditor;
         break;
       case IDM_LINE:
         if (pse) delete pse;
         pse = new LineEditor;
         break;
       . . .
```

Можна дещо структурувати програмний код. Замість рядків на кшалт

```
       case IDM_POINT:
         if (pse) delete pse;
         pse = new PointEditor;
         break;
```

для обробки основних повідомлень записати виклики методів класу **ShapeObjectsEditor**, який згідно вимогам є інтерфейсом для модуля редагування,

```
ShapeObjectsEditor Ім'я;
. . .
       case IDM_POINT:
         Ім'я.StartPointEditor(...);
         break;
```

як і наведено на сторінці 2 цього документу.

Цикл відображення об'єктів

Для циклу відображення об'єктів треба використати поліморфізм. Якщо оголосити масив вказівників на базовий клас Shape, наприклад, так:

```
Shape *pcshape[MY_SHAPE_ARRAY_SIZE];    //статичний масив
                                        //вказівників на об'єкти
```

то вивід усіх об'єктів обробником повідомлень WM_PAINT можна зробити наступним чином:

```
void ShapeEditor::OnPaint(HWND hWnd)
{
```

```
PAINTSTRUCT ps;
HDC hdc;

hdc = BeginPaint(hWnd, &ps);
for (int i=0; i<MY_SHAPE_ARRAY_SIZE; i++)
    if (pcshape[i])
        pcshape[i]->Show(hdc);
EndPaint(hWnd, &ps);
}
```

"Гумовий" слід

При вводі об’єктів інколи замість їхніх реальних зображень використовують спрощені образи, які показують лише контури об’єктів. Цим також підкреслюється, що зараз відбувається саме введення об’єкту. Можна побачити, наприклад, пунктирний "гумовий" слід, який розтягується від початкової точки до поточної точки курсору.

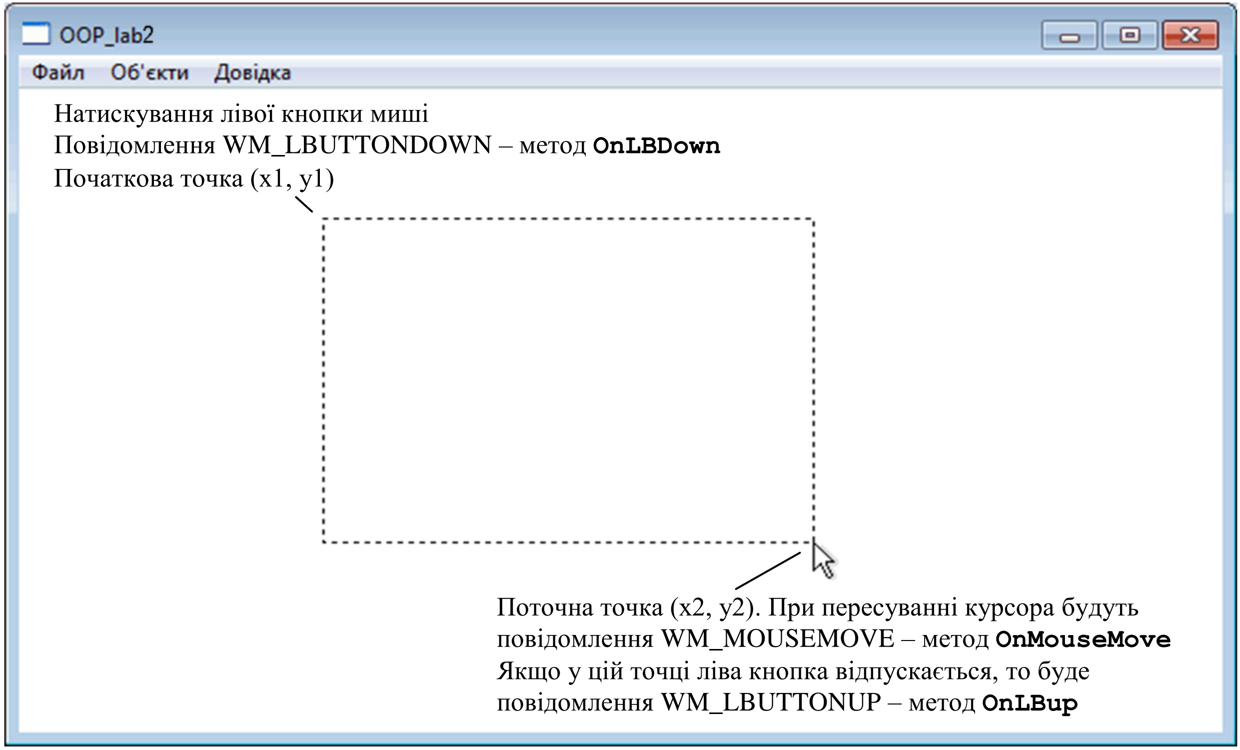

Рис. 2.9. Увід прямокутника по його протилежним кутам

Як запрограмувати рухомий "гумовий" слід? Розглянемо це на прикладі об’єкта-відрізка лінії. Уявимо, що курсор миші пересунуто у початкову точку лінії. Натискується ліва кнопка миші (і далі тримається натиснутою). Нашій програмі надсилається повідомлення WM_LBUTTONDOWN. Обробником цього повідомлення є наш метод OnLBdown. Цей метод повинен записати у якісь структури даних координати початкової точки. Зробити це можна, наприклад, так:

```
POINT pt;
GetCursorPos(&pt);
ScreenToClient(hWnd, &pt);
xstart = xend = pt.x;  //зберігаємо координати початкової точки
ystart = yend = pt.y;
```

Далі пересувається курсор миші і після кожного руху нашій програмі надсилається повідомлення WM_MOUSEMOVE, яке обробляється методом OnMouseMove. Тут потрібно малювати "гумовий" слід. Рекомендується така послідовність дій

- спочатку стирається слід для попереднього розташування курсору
- потім обчислюються нові координати курсору і для них малюється слід

```
POINT pt;

Стирається відрізок(xstart,ystart, xend,yend);

GetCursorPos(&pt);
ScreenToClient(hWnd, &pt);
xend = pt.x;      //координати поточної точки курсору
yend = pt.y;

Малюється відрізок(xstart,ystart, xend,yend);
```

Як стерти попереднє зображення "гумового" сліду, щоб не пошкодити зображення інших об'єктів? Найпростіше зробити це для лінійного сліду. Для цього лінію можна малювати у режимі комбінування `R2_NOTXORPEN` з попереднім зображенням.

Як зробити "гумовий" слід пунктирним потрібного кольору? За умовчанням усі лінії чорні, суцільні. Щоб змінити стиль, потрібно створити перо, намалювати щось, а коли перо стає непотрібним, його обов'язково треба знищити:

```
HPEN hPenOld,hPen;

hPen = CreatePen(PS_DOT, 1, Колір);
hPenOld = (HPEN)SelectObject(hdc, hPen);

Малюється лінія (лінії)

SelectObject(hdc, hPenOld);
DeleteObject(hPen);
```

Якщо об'єднати усі відомості, то можна порекомендувати наступну послідовність обробки повідомлення WM_MOUSEMOVE:

```
POINT pt;
HPEN hPenOld,hPen;
HDC hdc;

hdc = GetDC(hWnd);          //отримуємо контекст вікна для малювання
SetROP2(hdc,R2_NOTXORPEN);
hPen = CreatePen(PS_DOT, 1, 0);
hPenOld = (HPEN)SelectObject(hdc, hPen);

Малюються лінії "гумового" сліду попереднього розташування курсору

GetCursorPos(&pt);
ScreenToClient(hWnd, &pt);
xend = pt.x;      //координати поточної точки курсору
yend = pt.y;

Малюються лінії "гумового" сліду для поточного розташування курсору

SelectObject(hdc, hPenOld);
DeleteObject(hPen);
ReleaseDC(hWnd,hdc);    //закриваємо контекст вікна
```

Запис об'єкта у масив Shape

Після того, як ми перемістили курсор миші у потрібну кінцеву точку, можна відпускати ліву кнопку миші. Нашій програмі надсилається повідомлення WM_LBUTTONUP, яке оброблятиме метод **OnLBup**. Цей метод обчислить координати кінцевої точки і запише об'єкт у масив вказівників на об'єкти типу Shape.

Після запису об'єкту можна викликом **InvalidateRect(hWnd,NULL,TRUE)** спонукати нашу програму перемалювати зображення у головному вікні – Windows надішле повідомлення WM_PAINT, яке буде оброблено нашим методом **OnPaint**.

Позначка поточного режиму редагування

Потрібно, щоб користувач бачив, який тип об'єктів зараз вибрано для вводу. Позначку потрібного пункту меню можна запрограмувати обробником повідомлення WM_INITMENUPOPUP – **OnInitMenuPopup**. Позначка пункту меню робиться викликом функції `CheckMenuItem`. Наприклад, якщо потрібно позначити пункт меню "Еліпс" і зняти позначку з іншого пункту, то це можна запрограмувати так:

```
HMENU hMenu,hSubMenu;

hMenu=GetMenu(hWnd);
hSubMenu=GetSubMenu(hMenu,1);     //POPUP-меню Об'єкти
```

```
if ((HMENU)wParam == hSubMenu)
   {
   CheckMenuItem(hSubMenu, IDM_POINT, MF_UNCHECKED);  //зняти позначку
   CheckMenuItem(hSubMenu, IDM_LINE, MF_UNCHECKED);
   CheckMenuItem(hSubMenu, IDM_RECT, MF_UNCHECKED);
   CheckMenuItem(hSubMenu, IDM_ELLIPSE, MF_CHECKED); //позначити пункт
   }
```

Як саме запрограмувати такі позначки, студент вирішує самостійно

Позначку поточного типу об'єкту, що вводиться, можна зробити по-іншому. Для деяких варіантів завдань цієї лаб. роботи пропонується виводити текст у заголовку вікна програми. Це можна зробити викликом функції SetWindowText, наприклад

```
SetWindowText(hWnd, "Режим вводу еліпсів");
```

При використанні подібних функцій варто враховувати налаштування проекту щодо Unicode або multibyte символів.

Варіанти завдань та основні вимоги

1. Для усіх варіантів завдань необхідно дотримуватися вимог та положень, викладених вище у порядку виконання роботи та методичних рекомендаціях.

2. У звіті повинна бути схема успадкування класів – **діаграма класів**

3. Для вибору типу об'єкта в графічному редакторі Lab2 повинно бути меню "Об'єкти" з чотирма підпунктами. Меню "Об'єкти" повинно бути праворуч меню "Файл" та ліворуч меню "Довідка". Підпункти меню "Об'єкти" містять назви українською мовою геометричних форм – так, як наведено вище у порядку виконання роботи та методичних рекомендаціях. Геометричні форми згідно варіанту завдання.

4. Для вибору варіанту використовується Ж – номер студента в журналі.

5. Масив вказівників для динамічних об'єктів типу Shape

- динамічний масив Shape **pcshape;
- статичний масив Shape *pcshape[N];

причому, кількість елементів масиву вказівників як для статичного, так і динамічного має бути N = Ж+100.

Динамічний масив обирають студенти, у яких варіант (Ж mod 3 = 0). Решта студентів – статичний масив. Позначка mod означає залишок від ділення.

6. "Гумовий" слід при вводі об'єктів

- суцільна лінія чорного кольору для варіантів (Ж mod 4 = 0)
- суцільна лінія червоного кольору для варіантів (Ж mod 4 = 1)
- суцільна лінія синього кольору для варіантів (Ж mod 4 = 2)
- пунктирна лінія чорного кольору для варіантів (Ж mod 4 =3)

7. Чотири геометричні форми (крапка, лінія, прямокутник, еліпс) можуть мати наступні різновиди вводу та відображення.

7.1. Прямокутник

Увід прямокутника:
- по двом протилежним кутам для варіантів (Ж mod 2 = 0)
- від центру до одного з кутів для варіантів (Ж mod 2 = 1)

Відображення прямокутника:
- чорний контур з білим заповненням для (Ж mod 5 = 0)
- чорний контур з кольоровим заповненням для (Ж mod 5 = 1 або 2)
- чорний контур прямокутника без заповнення для (Ж mod 5 = 3 або 4)

Кольори заповнення прямокутника:
- жовтий для (Ж mod 6 = 0)
- світло-зелений для (Ж mod 6 = 1)
- блакитний для (Ж mod 6 = 2)
- рожевий для (Ж mod 6 = 3)
- померанчевий для (Ж mod 6 = 4)
- сірий для (Ж mod 6 = 5)

7.2. Еліпс

Увід еліпсу:
- по двом протилежним кутам охоплюючого прямокутника для варіантів (Ж mod 2 = 1)
- від центру до одного з кутів охоплюючого прямокутника для варіантів (Ж mod 2 = 0)

Відображення еліпсу:
- чорний контур з білим заповненням для (Ж mod 5 = 1)
- чорний контур з кольоровим заповненням для (Ж mod 5 = 3 або 4)
- чорний контур еліпсу без заповнення для (Ж mod 5 = 0 або 2)

Кольори заповнення еліпсу:
- жовтий для (Ж mod 6 = 1)
- світло-зелений для (Ж mod 6 = 2)
- блакитний для (Ж mod 6 = 3)
- рожевий для (Ж mod 6 = 4)
- померанчевий для (Ж mod 6 = 5)
- сірий для (Ж mod 6 = 0)

8. Позначка поточного типу об'єкту, що вводиться
- в меню (метод OnInitMenuPopup) для варіантів (Ж mod 2 = 0)
- в заголовку вікна для (Ж mod 2 = 1)
9. Приклад вибору варіанту. Для 9-го студента у списку (Ж = 9) буде:
- динамічний масив для Shape (9 mod 3 = 0) обсягом 109 об'єктів
- "гумовий" слід (9 mod 4 = 1) – суцільна лінія червоного кольору
- прямокутник:
  - ввід від центру до одного з кутів (9 mod 2 = 1)
  - чорний контур прямокутника без заповнення (9 mod 5 = 4)
- еліпс:
  - по двом протилежним кутам охопл. прямокутника(9 mod 2 = 1)
  - чорний контур з кольоровим заповненням (9 mod 5 = 4)
  - колір заповнення: блакитний (9 mod 6 = 3)
- позначка поточного типу об'єкту: в заголовку вікна (9 mod 2 = 1)

Примітка. Визначення кольорів та інші параметри варіантів можуть бути змінені викладачем шляхом оголошення студентам відповідного повідомлення завчасно перед постановкою завдань.

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний текст головного файлу .cpp та вихідні тексти власних модулів
4. Схеми, діаграми згідно завданню
5. Ілюстрації роботи програми (скріншоти)
6. Висновки

Контрольні запитання

1. Що таке поліморфізм?
2. Обробку яких повідомлень потрібно виконувати для вводу об'єктів?
3. Що таке абстрактний клас і скільки їх у цій програмі?
4. Як намалювати лінії та фігури потрібного кольору та стилю?
5. Як відобразити об'єкти у вікні програми?

Лабораторна робота №3. Розробка інтерфейсу користувача на C++

Мета: Мета роботи – отримати вміння та навички використовувати інкапсуляцію, абстракцію типів, успадкування та поліморфізм на основі класів C++, запрограмувавши графічний інтерфейс користувача.

Завдання

1. Створити у середовищі MS Visual Studio C++ проект Win32 з ім'ям **Lab3**.
2. Написати вихідний текст програми згідно варіанту завдання.
3. Скомпілювати вихідний текст і отримати виконуваний файл програми.
4. Перевірити роботу програми. Налагодити програму.
5. Проаналізувати та прокоментувати результати та вихідний текст програми.
6. Оформити звіт.

Теоретичні положення

Об'єктна орієнтованість ОС Windows. Для розробника програмних застосунків цю ОС цілком доречно вважати об'єктно-орієнтованою. З кожним застосунком зазвичай зв'язується головне вікно. Перед створенням вікна спочатку реєструється класс цього вікна у структурах даних ОС. Програма може окрім головного вікна створювати ще багато вікон, наприклад, вікон діалогу. Кожному вікну Windows надсилає повідомлення і отримує також повідомлення у відповідь.

На поверхні вікон можуть встановлюватися різноманітні елементи: кнопки, стрічки вводу тексту, радіокнопки-перемикачі, вікна списків тощо. Важливим моментом є те, що розробники Windows кожний такий елемент керуваня визначили теж як вікно. Це дає можливість оперувати елементами вікон і самими вікнами у єдиному стилі. Так, зокрема, кожному елементу також надсилаються повідомлення. Усі конкретні типи вікон мають деякі спільнівластивості, оскільки вони є похідними від абстракції вікна.

Існує ієрархія вікон. Зокрема, якщо на поверхню головного вікна встановити якійсь елемент, наприклад, кнопку, то головне вікно буде батьківським, а елементи на поверхні вікна будуть його дочірніми вікнами.

Методичні рекомендації

Створення панелі інструментів (Toolbar)

Для створення Toolbar зручно використати бібліотеку **COMCTL32** елементів керування загального користування (*Common Control Library*). Для того, щоб використати якійсь елемент з цієї бібліотеки, потрібно включити заголовочний файл

```
#include <commctrl.h>
```

Крім того, лінкеру потрібно посилання на статичну бібліотеку **COMCTL32.** Це можна зробити так:

```
#pragma comment(lib, "comctl32.lib")
```

При роботі програми потрібно спочатку викликати функцію

```
InitCommonControls();
```

а потім вже можна створювати Toolbar потрібного вигляду.

Створити дочірнє вікно Toolbar з 4 кнопками можна шляхом використання функції **CreateToolbarEx**, наприклад, так:

```
TBBUTTON tbb[4];                          //масив опису кнопок вікна
Toolbar

ZeroMemory(tbb, sizeof(tbb));
tbb[0].iBitmap = STD_FILENEW;             //стандартне зображення
tbb[0].fsState = TBSTATE_ENABLED;
tbb[0].fsStyle = TBSTYLE_BUTTON;          //тип елементу - кнопка
tbb[0].idCommand = ID_TOOL_FILE_NEW;      //цей ID буде у повідомленні
WM_COMMAND

tbb[1].iBitmap = STD_FILEOPEN;
tbb[1].fsState = TBSTATE_ENABLED;
tbb[1].fsStyle = TBSTYLE_BUTTON;
tbb[1].idCommand = ID_TOOL_FILE_OPEN;

tbb[2].iBitmap = STD_FILESAVE;
tbb[2].fsState = TBSTATE_ENABLED;
tbb[2].fsStyle = TBSTYLE_BUTTON;
tbb[2].idCommand = ID_TOOL_FILE_SAVEAS;

tbb[3].iBitmap = STD_PRINT;
tbb[3].fsState = TBSTATE_ENABLED;
tbb[3].fsStyle = TBSTYLE_BUTTON;
tbb[3].idCommand = ID_TOOL_FILE_PRINT;
```

```
hwndToolBar = CreateToolbarEx(hWndParent,        //батьківське вікно
                              WS_CHILD | WS_VISIBLE | WS_BORDER
                              | WS_CLIPSIBLINGS | CCS_TOP,
                              IDC_MY_TOOLBAR,    //ID дочірнього вікна
                                                 //Toolbar
                              1, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                              tbb,
                              4,             //кількість кнопок
                              0,0,0,0,       //розташування та розміри
                              sizeof(TBBUTTON));
```

Дочірнє вікно Toolbar розташовується на поверхні якогось вікна, HWND якого вказується параметром **hWndParent.** Зазвичай Toolbar у якості дочірнього вікна розташовується на головному вікні програми. Створити дочірнє вікно Toolbar можна також і викликом функції **CreateWindow**

```
hwndToolBar = CreateWindow(TOOLBARCLASSNAME,         //ім'я класу вікна
                           NULL,
                           WS_CHILD | WS_VISIBLE | WS_BORDER
                           | WS_CLIPSIBLINGS | CCS_TOP,
                           0,0,0,0,        //розташування та розміри
                           hWndParent,           //батьківське вікно
                           (HMENU)IDC_MY_TOOLBAR,//ID вікна toolbar
                           hInst,
                           0);
```

або, використавши **CreateWindowEx**

```
hwndToolBar = CreateWindowEx(0,
                             TOOLBARCLASSNAME,      //ім'я класу вікна
                             NULL,
                             WS_CHILD | WS_VISIBLE | WS_BORDER
                             | WS_CLIPSIBLINGS | CCS_TOP,
                             0,0,0,0,         //розташування та розміри
                             hWndParent,           //батьківське вікно
                             (HMENU)IDC_MY_TOOLBAR, //ID вікна toolbar
                             hInst,
                             0);
```

Примітка. У якості імені класу використана символічна константа **TOOLBARCLASSNAME**, замість якої автоматично підставляється **"ToolbarWindow32"**

Якщо Toolbar створювався викликом функції **CreateWindow** або **CreateWindowEx,** то спочатку буде порожній Toolbar. Далі потрібно у нього вставити потрібні елементи – це можна зробити так:

```
hwndToolBar = CreateWindow...

SendMessage(hwndToolBar, TB_BUTTONSTRUCTSIZE,
(WPARAM)sizeof(TBBUTTON), 0);

TBBUTTON tbb[4];
```

```
TBADDBITMAP tbab;
tbab.hInst = HINST_COMMCTRL;
tbab.nID = IDB_STD_SMALL_COLOR;
SendMessage(hwndToolBar, TB_ADDBITMAP, 0, (LPARAM)&tbab);

ZeroMemory(tbb, sizeof(tbb));
tbb[0].iBitmap = STD_FILENEW;          //стандартне зображення
tbb[0].fsState = TBSTATE_ENABLED;
tbb[0].fsStyle = TBSTYLE_BUTTON;       //тип елементу - кнопка
tbb[0].idCommand = ID_TOOL_FILE_NEW;   //цей ID буде у WM_COMMAND

tbb[1].iBitmap = STD_FILEOPEN;
tbb[1].fsState = TBSTATE_ENABLED;
tbb[1].fsStyle = TBSTYLE_BUTTON;
tbb[1].idCommand = ID_TOOL_FILE_OPEN;

tbb[2].iBitmap = STD_FILESAVE;
tbb[2].fsState = TBSTATE_ENABLED;
tbb[2].fsStyle = TBSTYLE_BUTTON;
tbb[2].idCommand = ID_TOOL_FILE_SAVEAS;

tbb[3].iBitmap = STD_PRINT;
tbb[3].fsState = TBSTATE_ENABLED;
tbb[3].fsStyle = TBSTYLE_BUTTON;
tbb[3].idCommand = ID_TOOL_FILE_PRINT;

SendMessage(hwndToolBar, TB_ADDBUTTONS, 4, (LPARAM)&tbb);
```

У результаті отримаємо такий Toolbar з чотирма кнопками

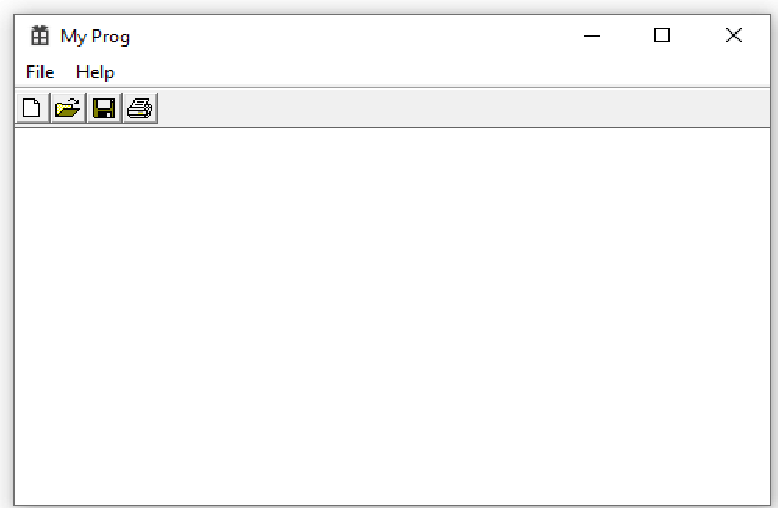

Рис. 3.1. Вигляд вікна toolbar

Кнопки можуть дублювати деякі важливі пункти меню, які часто використовуються впродовж роботи програми. Це зазвичай полегшує роботу користувачів програм.

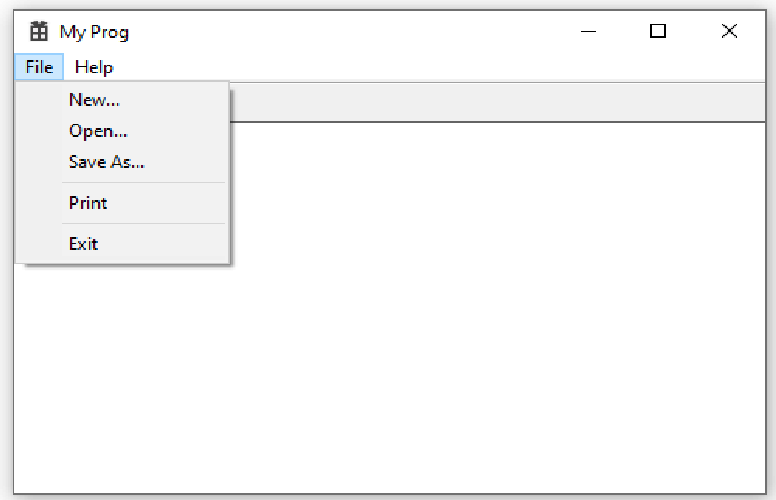

Рис. 3.2. Деякі важливі пункти меню

При натискуванні на кнопку повинна викликатися відповідна функція-обробник такої події (а точніше кажучі, обробник відповідного повідомлення)

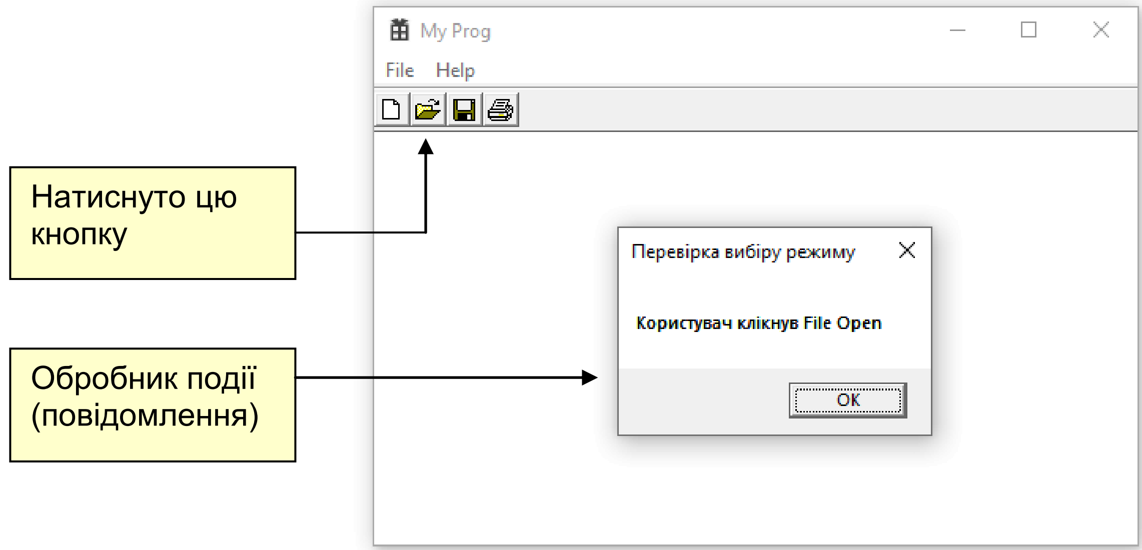

Рис. 3.3. Приклад реагування на натискування кнопки toolbar

Структуру програмного коду (скелет) для такої програми можна надати наступним чином

```
HWND hwndToolBar = NULL;      //ідентифікатор вікна Toolbar
. . .
```

```
int APIENTRY _tWinMain(. . .)
{
InitCommonControls();
. . .
}
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
   {
   case WM_CREATE:
      OnCreate(hWnd);          //тут створимо Toolbar
      break;

   case WM_COMMAND:
      switch (wmId)
         {
         case IDM_NEW:
         case ID_TOOL_FILE_NEW:
            OnFileNew(hWnd);
            break;

         case IDM_OPEN:                //ID пункту меню
         case ID_TOOL_FILE_OPEN:       //ID кнопки Toolbar
            OnFileOpen(hWnd);         //функція-обробник
            break;

         case IDM_SAVEAS:
         case ID_TOOL_FILE_SAVEAS:
            OnFileSaveAs(hWnd);
            break;

         . . .
```

```
void OnCreate(HWND hWnd)
{
TBBUTTON tbb[4];       //для Toolbar з чотирма кнопками
ZeroMemory(tbb, sizeof(tbb));
tbb[0].iBitmap = ...
. . .
hwndToolBar = CreateToolbarEx(hWnd,              //батьківське вікно
                              WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_CLIPSIBLINGS | CCS_TOP,
                              IDC_MY_TOOLBAR,    //ID дочірнього вікна Toolbar
                              1, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                              tbb,               //масив опису кнопок
                              4,                 //кількість кнопок
                              0,0,0,0,           //розташування та розміри
                              sizeof(TBBUTTON));
}
```

```
void OnFileNew(HWND hWnd)
{
   . . .
}
```

```
void OnFileOpen(HWND hWnd)
{
   MessageBox(hWnd, L"Користувач клікнув File Open", L"Перевірка вибіру режиму",
MB_OK);   //тест
}
```

Необхідно зазначити, що для підтримки такого Toolbar можна записати на початку головного файлу *.cpp наступні рядки

```
#include <commctrl.h>
#pragma comment(lib, "comctl32.lib")

//--toolbar support--
HWND hwndToolBar = NULL;
#define IDC_MY_TOOLBAR 1
#define ID_TOOL_FILE_NEW 1
#define ID_TOOL_FILE_OPEN 2
#define ID_TOOL_FILE_SAVEAS 3
#define ID_TOOL_FILE_PRINT 4
```

Врахування змін розмірів батьківського вікна

Якщо батьківське вікно раптом змінить розміри, то дочірнє вікно саме по собі не змінюється. Тоді виникне невідповідність ширини Toolbar ширині головного вікна. Один з варіантів вирішення такої проблеми – належна обробка повідомлення **WM_SIZE** батьківського вікна. Нижче наведений код обробника, який змінює ширину вікна Toolbar відповідно ширині клієнтської частини батьківського вікна викликом функції **MoveWindow**.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam)
{
switch (message)
   {
   case WM_CREATE:
      OnCreate(hWnd);
      break;

   case WM_SIZE:     //це повідомлення надсилається, якщо вікно
змінить розмір
      OnSize(hWnd);
      break;
. . .
}

//---обробник повідомлення WM_SIZE---
void OnSize(HWND hWnd)
{
RECT rc,rw;

if (hwndToolBar)
   {
   GetClientRect(hWnd, &rc);          //нові розміри головного вікна
   GetWindowRect(hwndToolBar, &rw);   //потрібно знати висоту Toolbar
   MoveWindow(hwndToolBar, 0,0,
              rc.right-rc.left,  //ширина Toolbar як у головного вікна
              rw.bottom-rw.top, FALSE);
   }
}
```

Власні зображення на кнопках Toolbar

Для цього можна створити **ресурс-bitmap**, який буде містити потрібні зображення, які потім можна відобразити на кнопках. Щоб створити такий ресурс, треба у вікні Solution Explorer клікнути на файлі ресурсів, потім додати у ресурси Bitmap. Потрібно вказати розміри цього BITMAP у пікселах. Тут потрібно розуміти, що картинки кнопок у растрі будуть розташовані по горизонталі, тому то для 5 кнопок 24×24 потрібен BITMAP розмірами 120×24. Цей BITMAP буде зберігатися у папці проекту у файлі формату BMP.

Після створення нового BITMAPу треба намалювати потрібні зображення кнопок – засобами редактора ресурсів, або ще якось.

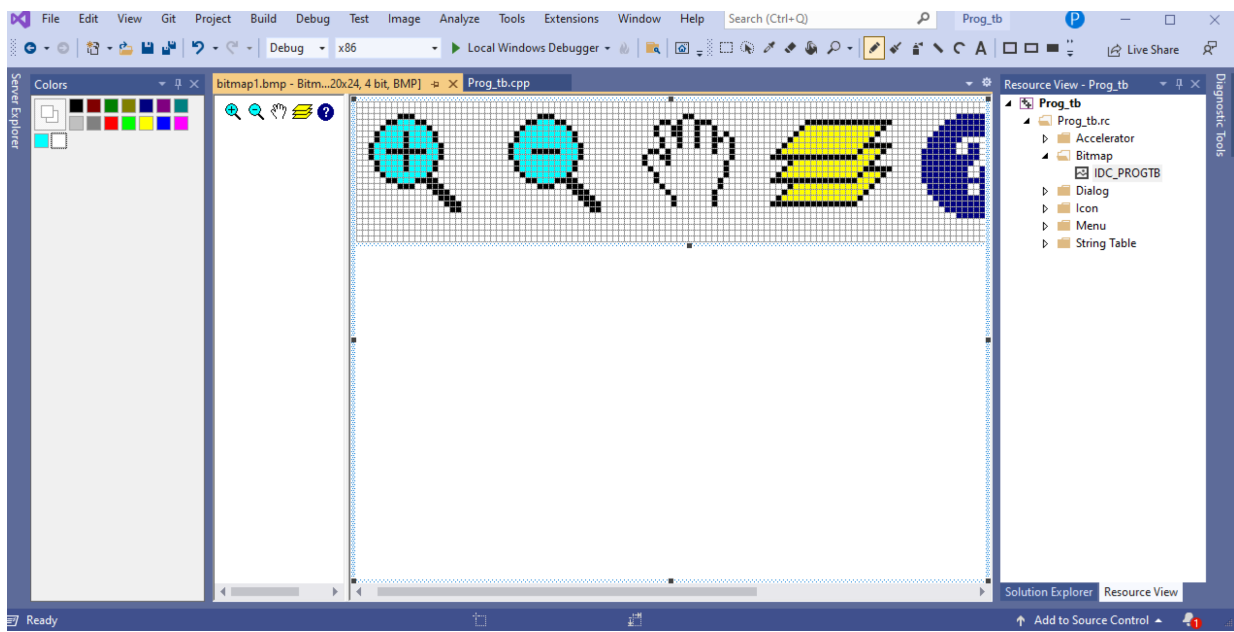

Рис. 3.4. Створення та редагування бітмапу для кнопок Toolbar у середовищі Visual Studio

Запрограмувати прикріплення власних зображень до кнопок Toolbar можна так:

```
TBBUTTON tbb[6];

ZeroMemory(tbb, sizeof(tbb));
tbb[0].iBitmap = 0;
tbb[0].fsState = TBSTATE_ENABLED;
tbb[0].fsStyle = TBSTYLE_BUTTON;
tbb[0].idCommand = ID_TOOL_ZOOMPLUS;

tbb[1].iBitmap = 1;
tbb[1].fsState = TBSTATE_ENABLED;
tbb[1].fsStyle = TBSTYLE_BUTTON;
tbb[1].idCommand = ID_TOOL_ZOOMINUS;

tbb[2].iBitmap = 2;              //індекс зображення у BITMAP
tbb[2].fsState = TBSTATE_ENABLED;
```

```
tbb[2].fsStyle = TBSTYLE_BUTTON;
tbb[2].idCommand = ID_TOOL_MOVE;

tbb[3].iBitmap = 3;
tbb[3].fsState = TBSTATE_ENABLED;
tbb[3].fsStyle = TBSTYLE_BUTTON;
tbb[3].idCommand = ID_TOOL_LAYERS;

tbb[4].iBitmap = 0;
tbb[4].fsState = TBSTATE_ENABLED;
tbb[4].fsStyle = TBSTYLE_SEP;      //роздільник груп кнопок
tbb[4].idCommand = 0;

tbb[5].iBitmap = 4;
tbb[5].fsState = TBSTATE_ENABLED;
tbb[5].fsStyle = TBSTYLE_BUTTON;
tbb[5].idCommand = IDM_ABOUT;

hwndToolBar = CreateToolbarEx(hWnd,
                              WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CLIPSIBLINGS |
CCS_TOP,
                              IDC_MY_TOOLBAR,
                              5,              //кількість зображень у BITMAP
                              hInst,
                              IDB_BITMAP1,    //ID ресурсу BITMAP
                              tbb,
                              6,              //кількість кнопок (разом з
роздільником)
                              24,24,24,24,    //розміри кнопок та зображень BITMAP
                              sizeof(TBBUTTON));
```

Результат:

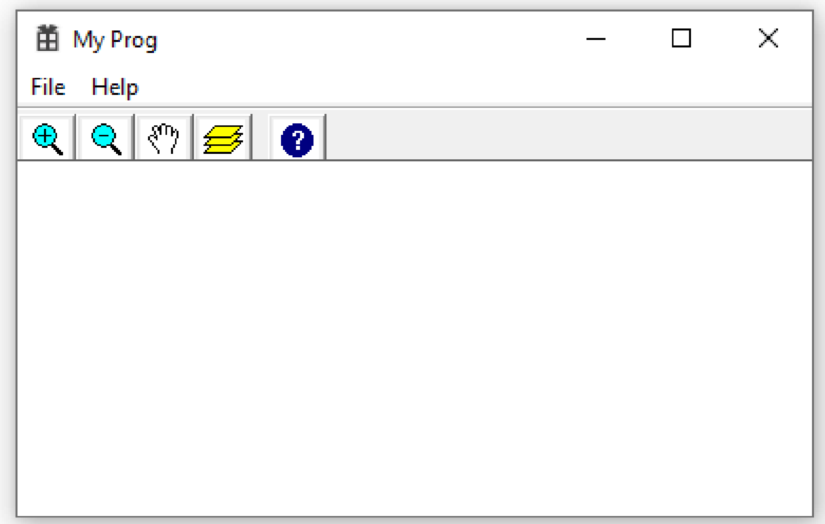

Рис. 3.5. Власні зображення кнопок

Керування станом кнопок

Вікну Toolbar і навіть кожній кнопці окремо можна надсилати повідомлення. Назви цих повідомлень починаються з **TB_**. Наприклад, якщо надіслати повідомлення **TB_PRESSBUTTON**, то можна "програмно" натиснути або віджати кнопку. Нижче наведено реалізацію кнопки, яка фіксується

```
int press = 0;
. . .

void OnToolMove(HWND hWnd)
{
press = !press;
SendMessage(hwndToolBar,TB_PRESSBUTTON, ID_TOOL_MOVE, press);
}
```

Результат на рисунку нижче.

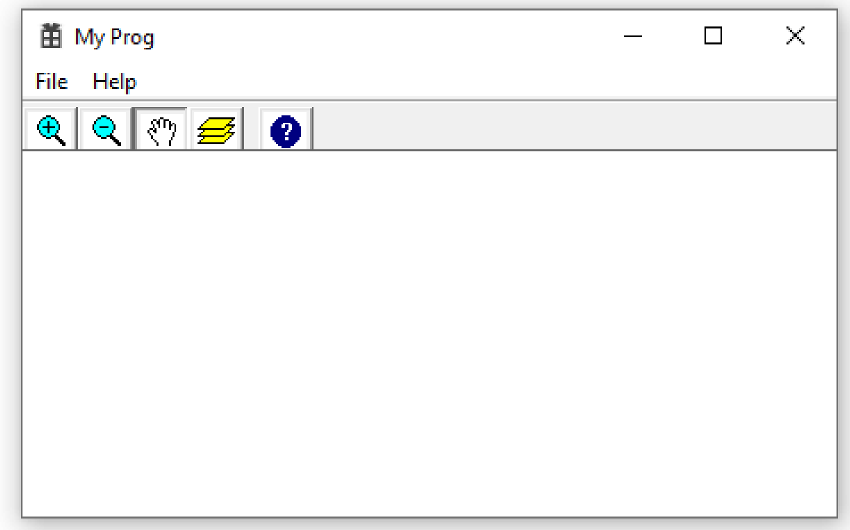

Рис. 3.6. Фіксація стану кнопки

Підказки для кнопок (tooltips)

Для цього потрібно при створенні Toolbar вказати стиль **TBSTYLE_TOOLTIPS**

```
hwndToolBar = CreateToolbarEx(hWnd,
                              WS_CHILD | WS_VISIBLE | WS_BORDER |
                              WS_CLIPSIBLINGS | CCS_TOP |
                              TBSTYLE_TOOLTIPS,
                              IDC_MY_TOOLBAR,
                              5,
                              hInst,
                              IDB_BITMAP1,
                              tbb,
                              6,
                              24,24,24,24,
                              sizeof(TBBUTTON));
```

Але це ще не все. Потрібно ще вставити обробник повідомлення **WM_NOTIFY** у тілі callback-функції вікна

```
LRESULT CALLBACK WndProc(HWND hWnd,UINT message,
                         WPARAM wParam,LPARAM lParam)
{
switch (message)
   {
   case WM_CREATE:
      OnCreate(hWnd);
      break;

   case WM_SIZE:
      OnSize(hWnd);
      break;

   case WM_NOTIFY:           //повідомлення від кнопок
      OnNotify(hWnd,wParam,lParam);
      break;

   case WM_COMMAND:
      . . .
```

Обробник повідомлення **WM_NOTIFY** для підказок можна запрограмувати так:

```
void OnNotify(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    LPNMHDR pnmh = (LPNMHDR)lParam;

    if (pnmh->code == TTN_NEEDTEXT)
    {
       LPTOOLTIPTEXT lpttt = (LPTOOLTIPTEXT)lParam;
       switch (lpttt->hdr.idFrom)
       {
       case ID_TOOL_ZOOMPLUS:
          lstrcpy(lpttt->szText, L"Збільшити");
          break;
       case ID_TOOL_ZOOMINUS:
          lstrcpy(lpttt->szText, L"Зменшити");
          break;
       case ID_TOOL_MOVE:
          lstrcpy(lpttt->szText, L"Пересунути");
          break;
       case ID_TOOL_LAYERS:
          lstrcpy(lpttt->szText, L"Вибрати");
          break;
       case IDM_ABOUT:
          lstrcpy(lpttt->szText, L"Довідка");
          break;
       default: lstrcpy(lpttt->szText, L"Щось невідоме");
       }
    }
}
```

Результат проілюстровано на рисунку нижче

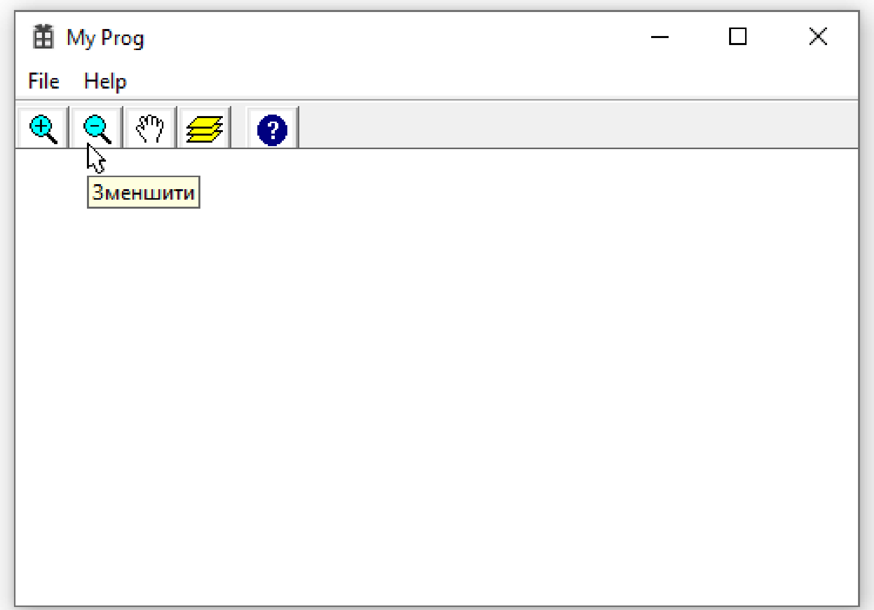

Рис. 3.7. Підказки у кнопок

Таким чином, наша програма повинна обробляти вже достатньо багато видів повідомлень, які надсилаються її головному вікну.

Варіанти завдань та основні вимоги

1. У звіті повинна бути схема успадкування класів – **діаграма класів**

2. Усі методи-обробники повідомлень, зокрема, і метод **OnNotify**, повинні бути функціями-членами деякого класу (класів).

3. Для вибору типу об'єкту в графічному редакторі Lab3 повинно бути **вікно Toolbar з кнопками відповідно типам об'єктів.** Кнопки дублюють підпункти меню "Об'єкти". Кнопки мають бути з підказками (tooltips). Меню "Об'єкти" повинно бути праворуч меню "Файл" та ліворуч меню "Довідка". Підпункти меню "Об'єкти" містять назви геометричних форм українською мовою. Геометричні форми згідно варіанту завдання.

4. Для вибору варіанту використовується значення $Ж = Ж_{лаб2} + 1$, де $Ж_{лаб2}$ – номер студента в журналі, який використовувався для попередньої лаб. роботи №2.

5. Масив вказівників для динамічних об'єктів типу Shape

- динамічний масив Shape **pcshape;
- статичний масив Shape *pcshape[N];

причому кількість елементів масиву вказівників як для статичного, так і динамічного має бути N = Ж+100.

Динамічний масив обирають студенти, у яких варіант (Ж mod 3 = 0). Решта студентів роблять статичний масив. Примітка. Позначка mod означає залишок від ділення.

6. "Гумовий" слід при вводі об’єктів

- суцільна лінія чорного кольору для варіантів (Ж mod 4 = 0)
- суцільна лінія червоного кольору для (Ж mod 4 = 1)
- суцільна лінія синього кольору для (Ж mod 4 = 2)
- пунктирна лінія чорного кольору для (Ж mod 4 =3)

7. Чотири геометричні форми (крапка, лінія, прямокутник, еліпс) можуть мати наступні різновиди вводу та відображення.

7.1. Прямокутник

Увід прямокутника:

- по двом протилежним кутам для варіантів (Ж mod 2 = 0)
- від центру до одного з кутів для (Ж mod 2 = 1)

Відображення прямокутника:

- чорний контур з білим заповненням для (Ж mod 5 = 0)
- чорний контур з кольоровим заповненням для (Ж mod 5 = 1 або 2)
- чорний контур прямокутника без заповнення для (Ж mod 5 = 3 або 4)

Кольори заповнення прямокутника:

- жовтий для (Ж mod 6 = 0)
- світло-зелений для (Ж mod 6 = 1)
- блакитний для (Ж mod 6 = 2)
- рожевий для (Ж mod 6 = 3)
- померанчевий для (Ж mod 6 = 4)
- сірий для (Ж mod 6 = 5)

7.2. Еліпс

Ввід еліпсу:

- по двом протилежним кутам охоплюючого прямокутника для варіантів (Ж mod 2 = 1)
- від центру до одного з кутів охоплюючого прямокутника для варіантів (Ж mod 2 = 0)

Відображення еліпсу:

- чорний контур з білим заповненням для (Ж mod 5 = 1)
- чорний контур з кольоровим заповненням для (Ж mod 5 = 3 або 4)
- чорний контур еліпсу без заповнення для (Ж mod 5 = 0 або 2)

Кольори заповнення еліпсу:
- жовтий для (Ж mod 6 = 1)
- світло-зелений для (Ж mod 6 = 2)
- блакитний для (Ж mod 6 = 3)
- рожевий для (Ж mod 6 = 4)
- померанчевий для (Ж mod 6 = 5)
- сірий для (Ж mod 6 = 0)

8. Позначка поточного типу об’єкту, що вводиться
   - в меню (метод OnInitMenuPopup) для варіантів (Ж mod 2 = 0)
   - в заголовку вікна для (Ж mod 2 = 1)

Примітка. Визначення кольорів та інші параметри варіантів можуть бути змінені викладачем шляхом оголошення студентам відповідного повідомлення завчасно перед постановкою завдань.

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний текст головного файлу .cpp (фрагменти, що ілюструють власний код), та вихідні тексти власних модулів
4. Схеми, діаграми згідно завданню
5. Ілюстрації (скріншоти)
6. Висновки

Контрольні запитання

1. У чому полягає об'єктна орієнтованість ОС Windows?
2. Обробку яких повідомлень потрібно виконувати у програмі Лаб3?
3. Що таке абстрактний клас і скільки їх у цій програмі?
4. Як забезпечити відповідність пунктів меню і кнопок Toolbar?
5. Як запрограмувати показ власних зображень на кнопках Toolbar?
6. Як створити власні зображення кнопок і де вони зберігаються?
7. Як запрограмувати текст підказок (tooltips)?

Лабораторна робота №4. Вдосконалення структури коду графічного редактора об’єктів на С++

Мета: Мета роботи – отримати вміння та навички проектування класів, виконавши модернізацію коду графічного редактора в об’єктно-орієнтованому стилі для забезпечення зручного додавання нових типів об'єктів.

Завдання

1. Створити у середовищі MS Visual Studio C++ проект Win32 з ім’ям **Lab4**.
2. Написати вихідний текст програми згідно варіанту завдання.
3. Скомпілювати вихідний текст і отримати виконуваний файл програми.
4. Перевірити роботу програми. Налагодити програму.
5. Проаналізувати та прокоментувати результати та вихідний текст програми.
6. Оформити звіт.

Теоретичні положення

Проблема небажаного зростання залежностей

В ході вдосконалення, розвитку системи зазвичай постає проблема зростання залежностей класів та модулів. Це може свідчити про неправильні рішення на етапі проєктування архітектури – без урахування можливостей нарощування системи в наступних версіях.

Якщо деякий клас A залежіть від декількох конкретних інших класів, то це може призвести до того, що при додаванні нових функціональних можливостей потрібно буде переписувати код цього класу A. Наприклад, якщо у класі A використовується композиція об'єктів якихось конкретних класів то це означає жорстку залежність класу A від цих конкретних класів.

Окремим випадком хибного архітектурного рішення може бути наявність у класі A декількох методів, кожний з яких призначений для роботи зі своїм конкретним класом – як це зроблено у класі ShapeObjectsEditor лабораторної роботи №2. Якщо розширювати систему, додаючи нові типи певних об'єктів, то треба додавати відповідно нові методи у клас A.

Можна сказати, що зазвичай треба проектувати так, щоб потім не треба було б переписувати існуючий програмний код. Безумовно, якщо у систему додаються нові функціональні можливості, то завжди потрібно якось додавати відповідний новий програмний код – але саме додавати, а не переробляти вже існуючий, починаючи з базового класу. Ця ідея, зокрема, задекларована у одному з принципів SOLID – Open/closed principle. Програмні сутності повинні бути відкриті для розширення, але закриті для змін.

З погляду на зазначену проблему, більш досконалим архітектурним рішенням у багатьох випадках буде залежність від абстракції замість залежності від конкретних класів. Одним з прийомів досягнення спрощення залежностей є інжекція, впровадження залежностей (*dependency injection*). Для цього у класі, який повинен використовувати інші конкретні класи, потрібно зробити член pB – вказівник на базовий клас цих конкретних класів, скажімо, клас B. А потім, впродовж роботи можна по мірі необхідності створювати об'єкти похідних класі від класу B класи і передавати адресу потрібного об'єкта в клас A, записуючи цю адресу у вказівник pB. Для передавання адреси потрібного об'єкт (або посилання на нього) у клас A можна використати спеціальний метод з параметром типу вказівника (або посилання) на базовий клас B. Завдяки поліморфізму через вказівник може викликатися однойменний метод будь-якого похідного класу від B без зміни коду класу A. Докладніше це викладено нижче щодо огляду загальної структури програми Lab4.

Рефакторинг

Зазвичай спочатку створюється дизайн системи, а потім пишеться код. Спочатку розробляється, як здається на той час, найоптимальніша структура системи, а потім виконується кодування. З часом код змінюється, і цілісність системи поступово погіршується.

Рефакторинг – це така зміна програмної системи, коли не змінюється зовнішня поведінка коду, проте покращується його внутрішня структура. Це спосіб систематичного наведення ладу, при якому вірогідність появи нових помилок мінімальна. Фактично, рефакторинг коду – це покращення його дизайну вже після того, як він написаний. За допомогою рефакторингу, як здається, можна взяти будь-який проект, навіть хаотичний, і переробити його в добре спроектований код.

Рефакторинг не змінює функціональність програми (системи), змінюється тільки код (можна сказати – дизайн коду). Рефакторинг

виконується як послідовність дрібних кроків, які робляться цілеспрямовано, і це дає у результаті значний сумарний ефект.

Методи та прийоми рефакторингу були вперше класифіковані та систематично викладені у 2000 році в книзі Мартіна Фаулера та ін. «Рефакторинг. Покращення існуючого коду»

Насправді, проектування безперервно відбувається впродовж розробки, а не завершується цілком на початку. Рефакторинг потрібно виконувати достатньо часто і регулярно впродовж роботи над проєктом. Потрібно знаходити час для систематичного рефакторингу, незважаючи на те, що робота над проектом уповільниться.

Необхідність рефакторингу є одною з рис практичного виконання усіх реальних проектів шляхом ООП. При розробці в стилі ООП зазвичай не вдається з першої спроби оптимально спроектувати архітектуру, структуру програми та її програмний код. Таким чином, розробку можна представити як послідовність кроків-ітерацій: первісний дизайн –> кодування –> тестування –> рефакторинг –> тестування –> додавання нового функціоналу –> тестування –> рефакторинг –> тестування і так далі.

Множинне спадкування класів

Множинне спадкування класів дозволяє створювати похідні класи одразу від декількох базових класів, наприклад

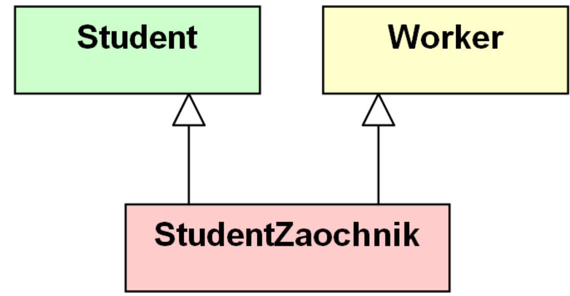

Рис. 4.1. Приклад множинного спадкування

Основна мета множинного спадкування – об'єкт похідного класу повинен об’єднувати властивості об’єктів декількох різних класів. Формат оголошення такого похідного класу наступний

```
class StudentZaochnik : public Student, public Worker
{
. . .
};
```

У даній лабораторній роботі буде запропоновано зробити нові типи фігур графічних об'єктів шляхом множинного спадкування класів.

Методичні рекомендації

Загальна структура програми

Програма повинна бути модульною. Потрібно використати модулі попередніх лабораторних робіт – №2 та №3. Вихідний текст складається з головного файлу **Lab4.cpp** та інших модулів і файлів проекту Win32. Усі модулі повинні роздільно компілюватися у проекті.

Зауваження по структурі попередніх рішень

Для лабораторної роботи №2 рекомендувалося створити клас **ShapeObjectsEditor** на кшалт цього:

```
class ShapeObjectsEditor
{
private:
  ...
public:
   ...
   void StartPointEditor(...);
   void StartLineEditor(...);
   void StartRectEditor(...);
   void StartEllipseEditor(...);
   void OnLBdown(HWND);
   void OnLBup(HWND);
   void OnMouseMove(HWND);
   void OnPaint(HWND);
};
```

Оголошення цього класу рекомендувалося записати у файлі **shape_editor.h**. Призначення інтерфейсних функцій-членів цього класу: щоб їх можна було викликати у функції головного вікна у якості обробників відповідних повідомлень, як наведено нижче

```
//---файл Lab2.cpp---
#include "shape_editor.h"

ShapeObjectsEditor ED;  //для варіанту статичного екземпляру редактора

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
int wmId, wmEvent;
```

```
switch (message)
  {
  case WM_LBUTTONDOWN:      //натиснуто ліву кнопку миші
    ED.OnLBdown(hWnd);
    break;
  case WM_LBUTTONUP:       //відпущено ліву кнопку миші
    ED.OnLBup(hWnd);
    break;
  case WM_MOUSEMOVE:        //пересунуто мишу у клієнтській частині вікна
    ED.OnMouseMove(hWnd);
    break;
  case WM_PAINT:  //потрібно оновлення зображення клієнтської частині вікна
    ED.OnPaint(hWnd);
    break;

  case WM_COMMAND:
    wmId    = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    // Parse the menu selections:
    switch (wmId)
      {
      case IDM_POINT:
        ED.StartPointEditor(...);     //початок вводу точкових об'єктів
        break;
      case IDM_LINE:
        ED.StartLineEditor(...);      //початок вводу об'єктів-ліній
        break;
      case IDM_RECT:
        ED.StartRectEditor(...);      //початок вводу прямокутників
        break;
      case IDM_ELLIPSE:
        ED.StartEllipseEditor(...);       //початок вводу еліпсів
        break;
. . .
```

Такий саме підхід був і для попередньої лабораторної роботи №3

Головний недолік такого рішення: у цьому класі фіксується склад функцій `StartТипEditor()`, окремих для кожного з типів об'єктів. Тобто, у класі вже фіксується певний список можливих типів об'єктів-фігур. Це погано. Як бути, якщо при розвитку можливостей редактора потрібно буде додати нові типи об'єктів – тоді змінювати, переписувати головний клас редактора?

Вимоги щодо головного класу графічного редактора

Назвемо цей клас інакше, аніж у лаб. №2 та 3, наприклад, **MyEditor**. Його оголошення записується у файлі-заголовку **my_editor.h**. Цей клас буде інкапсулювати масив об'єктів типу Shape, та містити інтерфейсні public

функції-члени, які повинні реалізовувати основні функції графічного редактора

```
class MyEditor
{
private:
   Shape **pshape;      //масив об'єктів типу Shape

   . . .                //інші члени

public:
   MyEditor(void);
  ~MyEditor();
   void Start(Shape *);
   void OnLBdown(HWND);
   void OnLBup(HWND);
   void OnMouseMove(HWND);
   void OnPaint(HWND);
   . . .                //інші члени (якщо необхідно)

};
```

Щодо конструктора. Зовсім не обов'язково, щоб створення робочого динамічного масиву об'єктів типу Shape відбувалося б саме у ньому. Також запрограмуйте, щоб відкриття динамічного массиву та його заповнення виконувалося би по мірі необхідності.

Щодо деструктора. Деструктор класу **MyEditor** має знищувати динамічного масиву об'єктів типу Shape.

Похідні класи та поліморфізм для класу **MyEditor** не передбачаються

Для початку редагування об'єктів рекомендується функція-член **Start( )** наприклад, так:

```
  case IDM_POINT:
     ED.Start(new Point);
     break;
  case IDM_LINE:
     ED.Start(new Line);
     break;
  case IDM_RECT:
     ED.Start(new Rect);
     break;
  case IDM_ELLIPSE:
     ED.Start(new Ellipse);
     break;

  case IDM_SOMETHING:
     ED.Start(new Something);       //ще якійсь тип фігур від Shape
     break;
```

Інші обробники повідомлень можна було б викликати так:

```
  case WM_LBUTTONDOWN:
     ED.OnLBdown(hWnd);
     break;
  case WM_LBUTTONUP:
     ED.OnLBup(hWnd);
     break;
  case WM_MOUSEMOVE:
     ED.OnMouseMove(hWnd);
     break;
  case WM_PAINT:
     ED.OnPaint(hWnd);
     break;
```

Для глобального об'єкта ED, який живе весь час роботи головного вікна програми, такий програмний код є достатньо коректним. Але, у випадку створення динамічного об'єкта для класу **MyEditor**, виклик обробників повідомлень може мати нюанси, наприклад:

```
MyEditor * ped = NULL;
  . . .
ped = new MyEditor;    //динамічне створення об'єкта-редактора
  . . .
  case WM_LBUTTONDOWN:
     if (ped) ped->OnLBdown(hWnd);
     break;
  case WM_LBUTTONUP:
     if (ped) ped->OnLBup(hWnd);
     break;
  case WM_MOUSEMOVE:
     if (ped) ped->OnMouseMove(hWnd);
     break;
  case WM_PAINT:
     if (ped) ped->OnPaint(hWnd);      //для WM_PAINT це не зовсім коректно
     break;
```

Якщо ми взялися обробляти повідомлення WM_PAINT, то пара функцій BeginPaint та EndPaint повинна викликатися завжди, незалежно від того, чи вже існує об'єкт, на який посилається вказівник ped, чи не існує. Тому, з огляду на особливості роботи Windows, надійніше буде так:

```
  case WM_PAINT:
     hdc = BeginPaint(hWnd, &ps);
     if (ped) ped->OnPaint(hWnd, hdc);
     EndPaint(hWnd, &ps);
     break;
```

При вводі об'єкта його "гумовий" слід повинен вимальовуватися поліморфним викликом функції Show класу з ієрархії базового класу Shape.

Базовий клас Shape та його спадкоємці

Так само, як в лабораторних роботах №2 та 3, ієрархія графічних об'єктів у цій лаб. роботі №4 починається з базового абстрактного класу **Shape**:

```
//---оголошення базового класу---
class Shape
{
protected:
   long xs1,ys1,xs2,ys2;
public:
   void Set(long x1,long y1,long x2,long y2);
   virtual void Show(HDC) = 0;     //визначається у похідних класах
};

//---визначення методу Set---
void Shape::Set(long x1,long y1,long x2,long y2)
{
xs1 = x1;
ys1 = y1;
xs2 = x2;
ys2 = y2;
}
```

Від класу **Shape** утворюються похідні класи: **PointShape**, **LineShape**, **RectShape**, **EllipseShape.** У кожному похідному класі визначається метод показу **Show** відповідно до геометрії об'єкта. Нагадаємо, як це пропонувалося для попередніх лабораторних робіт №2, 3.

У класі **PointShape** кожна точка малюється викликом функції

`SetPixel(hdc,x,y,колір);`

У класі **LineShape** відрізок прямої лінії малюється викликом функцій

`MoveToEx(hdc,x1,y1,NULL);`

`LineTo(hdc,x2,y2);`

У класі **RectShape** прямокутник малюється викликом функції

`Rectangle(hdc,x1,y1,x2,y2);` (прямокутник із заповненням)

або чотирма лініями (якщо потрібен тільки контур).

У класі **EllipseShape** еліпс малюється викликом функції

`Ellipse(hdc,x1,y1,x2,y2);` (еліпс із заповненням)

або **`Arc(hdc, x1,y1, x2,y2, 0,0,0,0);`** (якщо потрібен тільки контур).

Нові геометричні форми та множинне успадкування

Лінія з кружечками на кінцях – має бути об'єктом класу LineOOShape. Кружечки можна малювати методом Show класу EllipseShape

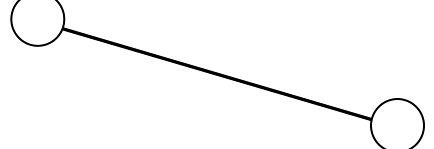

Рис. 4.2. Об'єкт класу LineOOShape

```
class LineOOShape : public LineShape, public EllipseShape
{
. . .
};
```

Каркас кубу – має бути об'єктом класу **CubeShape.**

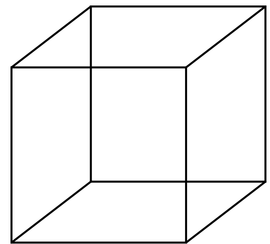

Рис. 4.3. Об'єкт класу **CubeShape**

Каркас кубу складатиметься з прямокутників та відрізків прямих ліній. Відповідно оголосимо його клас похідним від LineShape та RectShape

```
class CubeShape : public LineShape, public RectShape
{
. . .
};
```

Оголошення класів для усіх типів об'єктів треба робити у окремих заголовочних файлах *.h, а визначення функцій членів – у окремих файлах *.cpp. Таким чином, програмний код для усіх наявних типів об'єктів розподіляється по множині окремих модулів і файлів.

Ієрархія класів та побудова модулів повинні бути зручними для можливостей додавання нових типів об'єктів без переписування коду вже існуючих модулів. Необхідно забезпечити коректну підтримку поліморфізму для усіх типів об'єктів – як старих та нових.

Варіанти завдань та основні вимоги

1. Для усіх варіантів завдань необхідно дотримуватися вимог та положень, викладених вище у порядку виконання роботи та методичних рекомендаціях.

2. Номер варіанту завдання дорівнює номеру зі списку студентів у журналі.

Студенти з **непарним** номером (1, 3, 5, . . .) програмують глобальний статичний об'єкт класу MyEditor.

Студенти з **парним** номером (2, 4, 6, . . .) програмують **динамічний** обєкт класу MyEditor, забезпечивши коректне його створення та знищення.

3. Усі кольори та стилі (за винятком "гумового" сліду) геометричних форм – як у попередній лабораторній роботі №3. "Гумовий" слід при вводі усіх фігур малювати пунктирною лінією.

4. Окрім чотирьох типів фігур, які були у попередніх лаб. №2 та 3, запрограмувати ще введення та відображення двох нових фігур – лінія з кружечками та каркас куба.

Кольори ліній та заповнення цих нових фігур студент визначає на свій розсуд.

Для об’єктів типів лінії з кружечками та каркасу кубу відповідні класи запрограмувати саме **множинним успадкуванням**. У цій лабораторній роботі не дозволяється замінювати множинне спадкування, наприклад, композицією. У першу чергу це стосується метода Show для нових фігур – для відображення ліній треба використовувати виклики метода Show з класу **`LineShape`**, для відображення кружечків – виклики метода Show з класу **`EllipseShape`**, а для відображення прямокутників – виклики метода Show з класу **`RectShape.`**

5. Для усіх шости типів форм зробити кнопки Toolbar з підказками (tooltips)

6. У звіті повинна бути схема успадкування класів – діаграма класів. Потрібно побудувати діаграму класів засобами Visual Studio C++.

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний текст головного файлу .cpp (фрагменти, що ілюструють власний код), та вихідні тексти власних модулів
4. Схеми, діаграми згідно завданню

5. Ілюстрації (скріншоти)
6. Висновки

Контрольні запитання

1. Що таке поліморфізм і як він використаний у цій лабораторній роботі?
2. Обробку яких повідомлень потрібно виконувати для вводу об’єктів?
3. Що таке абстрактний клас і скільки їх у цій програмі?
4. Що таке множинне успадкування і як воно впливає на модульність?
5. Що таке ромбічне спадкування та які проблеми воно може спричинити?
6. Як побудувати діаграму класів засобами Visual Studio?
7. Як додати нові кнопки у Toolbar?

Лабораторна робота №5. Розробка багатовіконного інтерфейсу користувача для графічного редактора об'єктів

Мета: Мета роботи – отримати вміння та навички програмувати багатовіконний інтерфейс програми на C++ в об'єктно-орієнтованому стилі.

Завдання

1. Створити у середовищі MS Visual Studio C++ проект Desktop Application з ім'ям **Lab5**.
2. Написати вихідний текст програми згідно варіанту завдання.
3. Скомпілювати вихідний текст і отримати виконуваний файл програми.
4. Перевірити роботу програми. Налагодити програму.
5. Проаналізувати та прокоментувати результати та вихідний текст програми.
6. Оформити звіт.

Теоретичні положення

Патерн Singleton

Цей патерн гарантує, що для деякого класа можна створити тільки один екземпляр об'єкта. Для деяких типів об'єктів важливо, щоб впродовж роботи існував тільки один екземпляр об'єкта і забороняється навмисно чи помилково створювати інші екземпляри.

Класична реалізація класу Singleton. Для блокування створення інших об'єктів контролюється значення статичного члену – вказівника **p_instance.** Об'єкт створюється шляхом **new**.

Приклад оголошення класу у файлі singleton.h

```
class Singleton
{
private:
   static Singleton * p_instance;
   Singleton() {}
   Singleton( const Singleton& );
   Singleton& operator=( Singleton& );
public:
   static Singleton * getInstance();
   void Func();    //змістовна функція для клієнта
   . . .           //інші змістовні функції, потрібні для клієнта
};
```

Визначення функцій класу у файлі singleton.cpp

```
#include "singleton.h"

Singleton* Singleton::p_instance = 0;

Singleton* Singleton::getInstance()
{
if(!p_instance)
       p_instance = new Singleton();
return p_instance;
}

void Singleton::Func()
{
. . .    //щось виконується

}
```

Приклад створення об’єкта та виклику функції:

```
#include "singleton.h"
. . .

Singleton* s = Singleton::getInstance();    //створюємо екземпляр об'єкта

s->Func();              //викликаємо потрібну змістовну функцію
```

Інший варіант реалізації – **Singleton Меєрса**. Оголошення класу:

```
class Singleton
{
private:
    Singleton() {}
    Singleton( const Singleton&);
    Singleton& operator=( Singleton& );
public:
    static Singleton& getInstance()
    {
    static Singleton  instance;
    return instance;
    }
   void Func();   //змістовна функція для клієнта
   . . .          //інші змістовні функції, потрібні для клієнта
};
```

Змістовна функція цього класу визначає якійсь потрібний клієнту код

```
void Singleton::Func()
{
. . .    //щось виконується

}
```

Приклад створення об’єкта класу Singleton Меєрса:

```
Singleton& a = a.getInstance();    //створюємо екземпляр об'єкта
a.Func();   //викликаємо потрібну змістовну функцію
```

Примітка. Так само, як і для класичної реалізації, оголошення файлу можна зробити у окремому файлі-заголовку, а визначення функцій та створення-використання – у відповідних файлах *.cpp.

В даній лабораторній роботі пропонується зробити клас Singleton для головного об’єкта-редактора.

Немодальні вікна діалогу

В програмних застосунках для ОС Windows можна використовувати різноманітні вікна діалогу. Модальні (*modal*) вікна діалогу зазвичай використовуються для вводу якихось параметрів. Коли з’являється модальне вікно, фокус вводу зосереджується на ньому. Зазвичай модальні вікна для вводу якихось значень мають один чи декілька елементів вводу та дві кнопки: Ok та Cancel. Не можна переключитися на інші вікна доти не закрити модальне вікно натискуванням Ok чи Cancel. Модальне вікно діалогу ми вже створювали у попередніх роботах, зокрема, у лабораторній роботі №1 для цього ми використали функцію DialogBox зі складу Windows API.

Немодальне (*modeless*) вікно діалогу не вимагає, щоб його закрили для переключення фокусу вводу на інше вікно програми. Воно може містити різноманітні елементи для вводу (керування) а також кнопку Close. Такі вікна можуть достатньо самостійно існувати упродовж роботи програми, не блокуючи роботу інших вікон. Немодальні вікна варто робити у випадках, коли потрібно локалізувати у окремому вікні якісь поточні параметри вводу, наприклад, у такому вікні може бути палітра для вибіру кольорів малювання об’єктів у графічному редакторі. Іншій приклад – вікно пошуку тексту у стандартному блокноті Windows.

Як створити немодальне вікно діалогу? Для створення немодального вікна діалогу можна скористатися функцією CreateDialog, яка належіть до функцій Windows API. Якщо на поверхні такого вікна мають бути якісь елементи керування – кнопки, вікна списків, перемикачі, радіо кнопки тощо, то спочатку потрібно у редакторі ресурсів внести ці елементи на ресурс опису вікна, який потрібно зберегти у файлі ресурсів так само, як і для модального вікна. Тоді при створенні вікна можна передати функції CreateDialog у якості другого параметру передати через

MAKEINTRESOURCE Id ресурсу вікна діалогу. Нижче приклад створення немодального вікна у власній функції ActivateDlgWindow

```
HWND hWndDlg = NULL;      //хендл немодального вікна діалогу
. . .

void ActivateModeless(HWND hWnd)
{
if (hWndDlg)   //якщо це вікно діалогу вже є, то закриваємо його
   {
   PostMessage(hWndDlg, WM_SYSCOMMAND, SC_CLOSE, 0);
   return;
   }
hWndDlg = CreateDialog(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, DlgProc);
}
```

Приклад Callback-функції немодального вікна діалогу

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT message,
                      WPARAM wParam, LPARAM lParam)
{
switch (message)
   {
   case WM_SYSCOMMAND:
      if (wParam == SC_CLOSE) DestroyWindow(hDlg);
      break;
   case WM_COMMAND:
      if (LOWORD(wParam) == IDOK)   //натиснуто кнопку Apply
         {
         Apply(hDlg);    //щось робимо
         break;
         }
      if (LOWORD(wParam) == IDCANCEL)    //натиснуто кнопку Close
         DestroyWindow(hDlg);
      break;
   case WM_DESTROY:        //знищення вікна
      hWndDlg = NULL;
      break;
   default: break;
   }
return FALSE;
}
```

Для закриття такого вікна діалогу будемо використовувати окрему кнопку Close а також кнопку-хрестик у правому верхньому куті вікна – там розташовується системне меню. Ми запрограмували, щоб при натискуванні вказаних кнопок робився виклик функції DestroyWindow.

Методичні рекомендації

Загальна структура програми

Програма складається, як мінімум, з двох частин:

– частини підтримки головного вікна, зокрема, для вводу та відображення об'єктів геометричних форм;

– частина підтримки окремого **немодального діалогового вікна таблиці.** У якому відображаються список об'єктів. Таблиця має бути описана окремим класом **MyTable.** Ця частина повинна бути запрограмована у окремому модулі (файли *.cpp, *.h, *.rc)

Програма повинна бути модульною. Вихідний текст складається з головного файлу **Lab5.cpp** та інших модулів і файлів проекту Desktop Application. Усі модулі повинні роздільно компілюватися у проекті.

Вікно таблиці

Множину об'єктів, які вводяться та відображаються у головному вікні, утворюючи складний малюнок з багатьох форм, потрібно відображати у таблиці, яка представляється у окремому немодальному вікні діалогу.

Кожний рядок таблиці – це опис відповідного геометричного об'єкту, який уведено у масив форм та відображається у головному вікні. Рядки таблиці заповнюються у порядку відповідно вводу об'єктів. У таблиці повинен бути скролінг по вертикалі, щоб мати можливість переглядати список з багатьох об'єктів – геометричних форм.

Стовпчиків таблиці п'ять – назва об'єкту та координати розташування.

Таблиця 5.1

Назва	**x1**	**y1**	**x2**	**y2**
Точка	10	130	10	130
Лінія	45	23	400	327
Еліпс	123	304	257	412

Впродовж роботи редактора при кожному додаванні нового об'єкта у головному вікні, інформація про цей об'єкт повинна автоматично з'являтися у вікні таблиці у новому нижньому рядку

Вікно таблиці можна запрограмувати на основі стандартного вікна списка. Інший варіант – щоб програма сама малювала на поверхні

діалогового вікна лінії вкликом функцій Windows API: кожна лінія малюється MoveToEx( ), LineTo( ), а текст – TextOut( ).

Вимоги щодо класу таблиці

Вікно таблиці програмується у незалежному модулі **my_table**, який складається з файлів **my_table.cpp, h, rc**. У файлі my_table.h записується оголошення класу **MyTable**

```
class MyTable
{
   . . .          //якісь члени класу
public:
   void Add(..);  //функція додавання у таблицю нового рядка
                  //з описок об'єктів

   . . .          //інші функції

};
```

Після уводу у головному вікні нового об'єкту обробник повідомлення WM_LBUTTONUP викликає фанкцію MyTable::Add( ) щоб занести у список-таблицю назву та координати нового об'єкту. Після цього у вікні таблиці з'являється новий рядок.

Вимоги щодо головного класу графічного редактора

Назвемо цей клас так само, як у попередньої лаб. №4 – **MyEditor**. Його оголошення записується у файлі-заголовку **my_editor.h**. Цей клас буде інкапсулювати масив об'єктів типу Shape, та містити інтерфейсні public функції-члени, які повинні реалізовувати основні функції графічного редактора

Головний клас повинен бути **Singleton** певної ралізації відповідно варіанту завдання

Залежності модулів

Клієнт – головний файл (Lab5.cpp) використовує інтерфейсні функції модулів **my_editor, my_table** та інших модулів. Ієрархія залежностей модулів відображена нижче

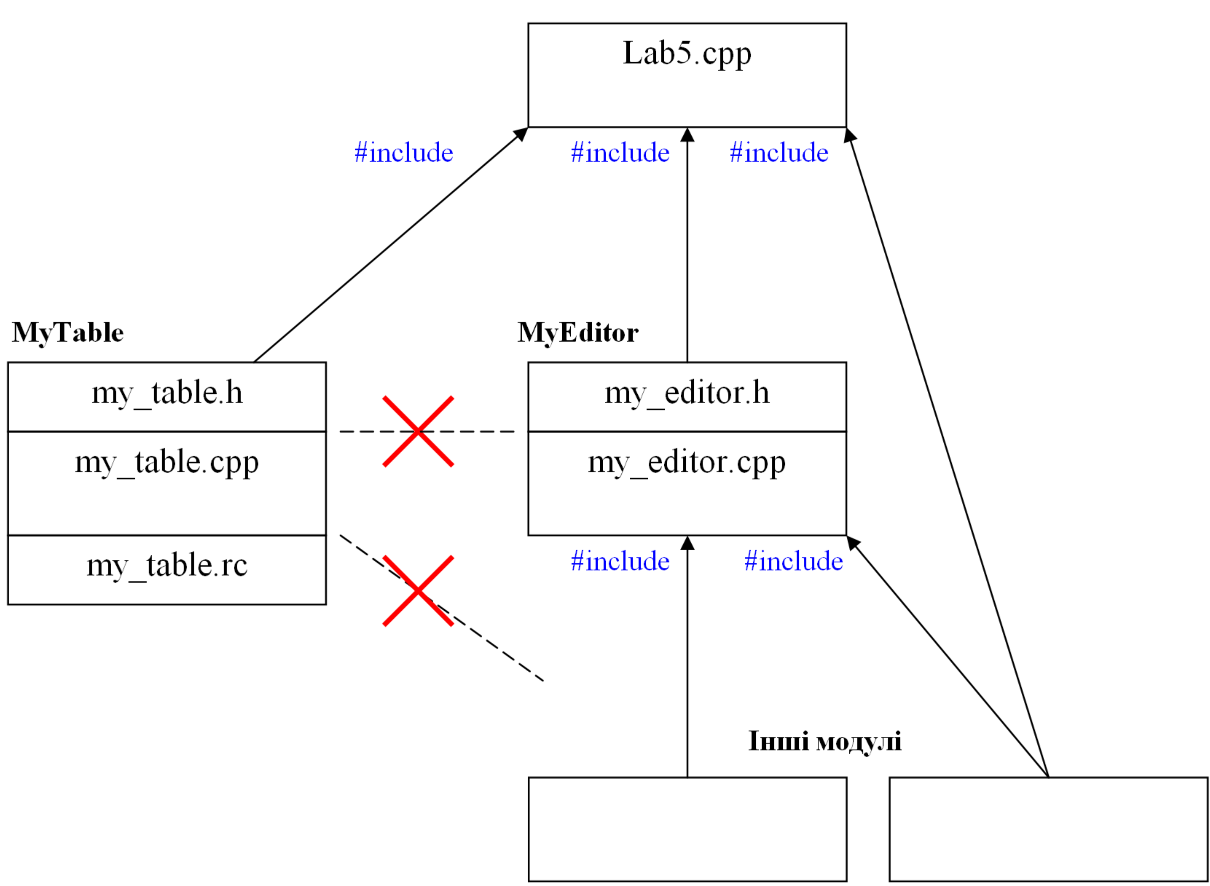

Рис. 5.1. Залежності модулів у проекті Lab5

Вимога: модуль **my_table** повинен бути самодостатнім та **незалежним від будь-якого з модулів програми**. Він не може використовувати по #include будь-який файл проекту (окрім хіба що файлу **framework.h**).

Можна сказати, що вікно таблиці є самодостатнім компонентом, який можливо буде використовувати у інших проектах, не тягнучи за собою залежності від модулів проекту Lab5.

Запис об'єктів у файл

Множину об'єктів, які вводяться у головному вікні та відображаються у головному вікні та вікні таблиці, потрібно записувати у файл.

Програма графічний редактор лаб. №5 повинна записувати кожний об'єкт при вводі у текстовий файл, який міститься у поточній директорії редактора.

Інформацію про поточний об'єкт, що введено, записувати у файл у вигляді одного рядка тексту. Цей рядок додається до вже записаного файлу.

Рядок файлу містить п’ять полів: назва класу об’єкта та чотири координати: x1, y1, x2, y2. Поля повинні відокремлюватися символами табуляції.

Для програмування запису у файл можна скористатися бібліотечними функціями, наприклад: **fopen_s( )**, **fprintf( )**, **fclose( )** або іншими.

Нижче приклад запису у файл 10 чисел, розділених символами табуляції ’\t’:

```
FILE* fout;
fopen_s(&fout, "myfile.txt", "wt");
if (fout)
{
    int num = 10;
    fprintf(fout, "%d numbers:\n", num);
    for (int i=1; i<num; i++)
        fprintf(fout, "%d\t", i);
    fprintf(fout, "%d\n", num);    //завершуємо рядок символом \n
    fclose(fout);

}
```

Результат:

```
10 numbers:
1    2    3    4    5    6    7    8    9    10
```

Приклад коду для читання усіх рядків текстового файлу:

```
FILE* fin;
fopen_s(&fin, "myfile.txt", "rt");
if (fin)
{
    char buf[256];
    while (fgets(buf, 256, fin))   //читаємо рядок тексту у буфер
    {
        //... щось робимо з рядком тексту
    }
    fclose(fin);
}
```

Варіанти завдань та основні вимоги

1. Для усіх варіантів завдань необхідно дотримуватися вимог та положень, викладених вище у порядку виконання роботи та методичних рекомендаціях.

2. Номер варіанту завдання дорівнює номеру зі списку студентів у журналі.

Студенти з **непарним** номером (1, 3, 5, . . .) програмують глобальний статичний об'єкт класу MyEditor у вигляді **Singleton Меєрса**.

Студенти з **парним** номером (2, 4, 6, . . .) програмують об’єкт класу MyEditor на основі **класичної реалізації Singleton**.

3. Усі кольори та стилі геометричних форм – як у попередньої лаб. роботі №4.

4. Запрограмувати вікно таблиці. Для його відкриття та закриття передбачити окремий пункт меню. Вікно таблиці повинно автоматично закриватися при виході з програми.

5. Вікно таблиці – немодальне вікно діалогу. Таблиця повинна бути запрограмована як клас у окремому модулі. Інтерфейс модуля у вигляді оголошення класу таблиці

6. Запрограмувати запис файлу множини об’єктів, що вводяться

7. Оголошення класів для усіх типів об'єктів робити у окремих заголовочних файлах *.h, а визначення функцій членів – у окремих файлах *.cpp. Таким чином, програмний код для усіх наявних типів об'єктів розподілюється по множині окремих модулів.

8. Ієрархія класів та побудова модулів повинні бути зручними для можливостей додавання нових типів об'єктів без переписування коду вже існуючих модулів.

9. У звіті повинна бути схема успадкування класів – діаграма класів. Побудувати діаграму класів засобами Visual Studio C++.

11. Б**онуси-заохочення**, які можуть суттєво підвищити оцінку лабораторної роботи. Оцінка підвищується за виконання кожного пункту, з наведених нижче:

1). Якщо у вікні таблиці буде передбачено, щоб користувач міг виділити курсором рядок таблиці і відповідний об’єкт буде якось виділятися на зображенні у головному вікні.

2). Якщо у вікні таблиці користувач може виділити курсором рядок таблиці і відповідний об’єкт буде вилучено з масиву об’єктів.

При виконанні бонусів 1 та 2 забороняється робити для цього нові залежності модуля **my_table** від інших .cpp файлів. Тоді як надіслати повідомлення (наприклад, про виділення користувачем якогось рядка таблиці) від вікна таблиці клієнту цього вікна (наприклад, коду головного файлу .cpp)? Підказки можна знайти у матеріалі лекції стосовно технології **Callback,** а також патернів Observer, Listener.

3). Якщо програма не тільки записує у файл опис множини об’єктів, а ще й здатна завантажити такий файл і відобразити відповідні об’єкти у головному вікні та вікні таблиці

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний текст головного файлу .cpp (фрагменти, що ілюструють власний код), та усі вихідні тексти власних модулів
4. Схеми, діаграми згідно завданню. Приклад текстового файлу множини об’єктів.
5. Ілюстрації (скріншоти)
6. Висновки

Контрольні запитання

1. Що таке патерн проєктування?
2. Що таке Singleton?
3. Чим відрізняється класична реалізація Singleton від Singleton Меєрса?
4. Як запрограмувати немодальне діалогове вікно?
5. Як запрограмувати запис у файл об’єктів - геометричних форм?
6. Покажіть у програмі поліморфізм

Лабораторна робота №6. Побудування програмної системи з множини об’єктів, керованих повідомленнями

Мета роботи: отримати вміння та навички використовувати засоби обміну інформацією та запрограмувати взаємодію незалежно працюючих програмних компонентів.

Завдання

1. Створити у середовищі MS Visual Studio C++ проект Win32 з ім’ям **Lab6**.
2. Написати вихідні тексти усіх програм-компонентів згідно варіанту завдання.
3. Скомпілювати вихідні тексти і отримати виконувані файли програм.
4. Перевірити роботу програм. Налагодити взаємодію програм.
5. Проаналізувати та прокоментувати результати та вихідні тексти програм.
6. Оформити звіт.

Теоретичні положення

Загалом, поняття об'єктної орієнтованості можуть бути значно ширшими порівняно з найбільш розповсюдженимим натепер поняттями щодо тлумачення парадигми ООП та популярними засобами створення програм. У цьому сенсі можна згадати ідеї, проголошені Аланом Кеєм ще на початку другого тисячиріччя. Він вважав, що досконалі комп'ютерні системи мають складатися з множини об'єктів подібно до того як живі істоти складаються з клітин. Головне у об'єктів – здатність обмінюватися повідомленнями. Це дозволяє множині об'єктів вибудовуватися у певні структури для вирішення завдань.

До відома: Алан Кей (Alan Curtis Kay) – спочатку інженер фірми Xerox PARC, а потім провідний фахівець фірми Apple. Є фундатором сучасного графічного інтерфейсу користувача та винахідником словосполучення «об'єктно-орієнтований» (*object-oriented*). Розробив мову Smaltalk. Є лауреатом премії Тьюринга 2003р. за роботи над об'єктно-орієнтованим програмуванням а також премії Кіото 2004р.

Одним з тлумачень об’єктно-орієнтованої системи є таке: у системі функціонують декілька об’єктів, які обмінюються повідомленнями. Повідомлення може бути вимогою-запитом зробити щось, повідомленням-відповіддю та повідомленням надсилання даних (у вигляді самих даних, або

у вигляді посилання, де ці дані треба взяти). Хтось організовує повідомлення, або об’єкти налаштовують їх самі для виконання системою певного завдання.

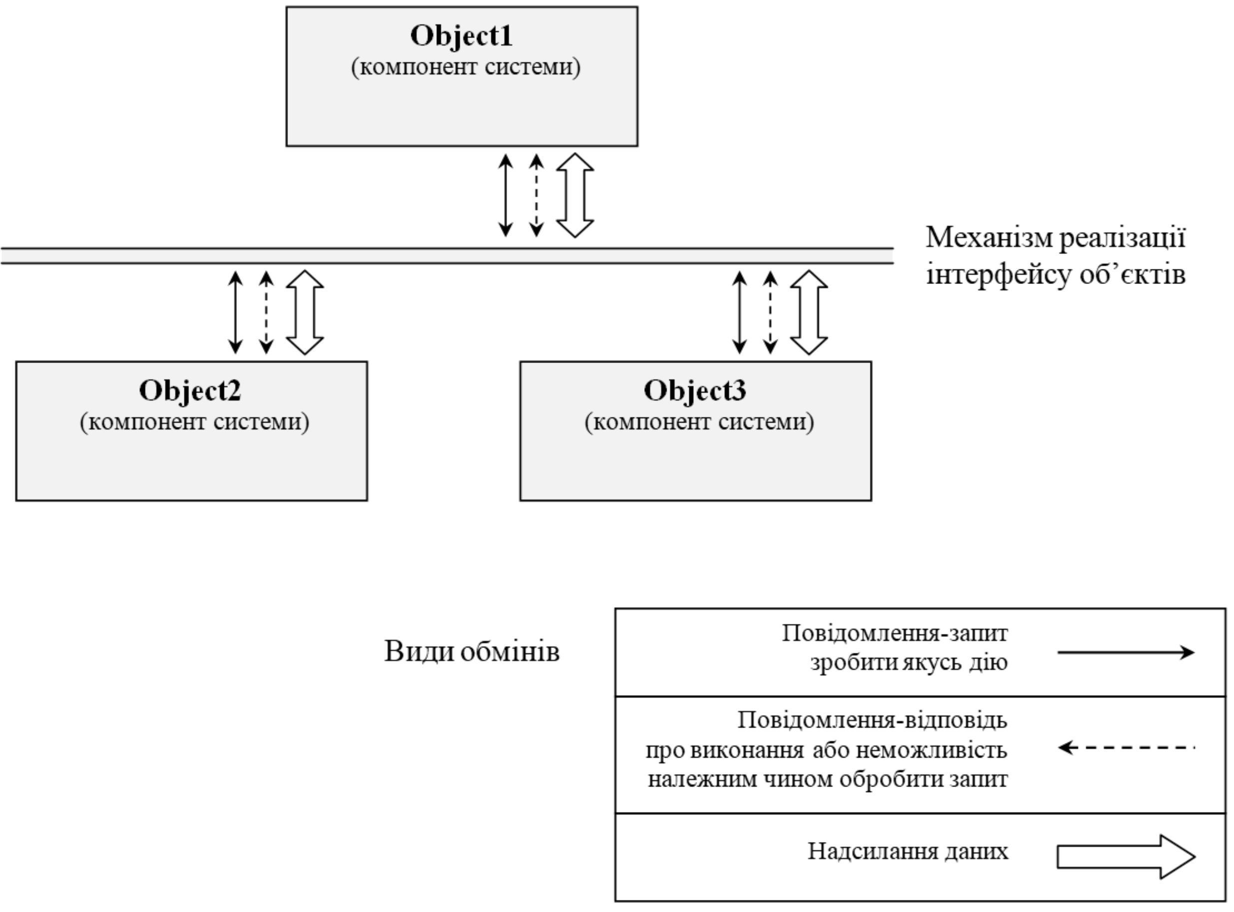

Рис. 6.1. Узагальнена схема взаємодії об’єктів – компонентів системи

Традиційна реалізація об’єктів у мовах програмування на кшалт C++, Java, C# обмін повідомленнями зводить до виклику публічних функцій-методів класів. Це достатньо примітивний спосіб взаємодії об’єктів і він породжує багато проблем реалізації ООП.

Проте, зовсім не обов’язково розуміти повідомлення тільки як виклик метода класу. Для того, щоб об’єкт виконав якісь потрібні дії, загалом, треба якимось чином надіслати йому запит-прохання. Якщо об’єкт може це зробити, то він це робить і надсилає потрібний результат, а якщо ні, наприклад, тому що він зараз зайнятий іншою роботою, або запит містить некоректну вимогу, то об’єкт надсилає відповідне повідомлення-відповідь.

Визначальним при побудові системи об'єктів є середовище та засоби передачі-отримання повідомлень. Повідомлення повинні якось диспетчеруватися, має бути забезпечена організація черг повідомлень тощо.

Методичні рекомендації

Розглянемо можливості організації обміну повідомленнями у середовищі ОС Windows.

У середовищі Windows є багато різноманітних можливостей побудувати інтерфейс повідомленнями між об’єктами-компонентами програмних систем. Одною з таких можливостей є механізм повідомлень Windows (*Windows messages*).

Пошук об’єкта-компонента для співпраці

Уявимо собі, що програма хоче надіслати повідомлення вікну іншої програми. А для цього треба знати хендл її вікна. У складі API Windows є функція **FindWindow**, яка дозволяє отримати hWnd будь-якої програми, якщо та зараз працює. Рекомендується шукати іншу програму по імені класу її вікна (який потрібно заздалегідь знати). Наприклад, якщо клас головного вікна програми зветься "DATACREATOR" то

```
hWndDataCreator = FindWindow("DATACREATOR", NULL);
```

Де знайти назву класу головного вікна програми? У головному cpp-файлі (якщо він спочатку був автоматично згенерований при створенні нового проекту Win32) є перемінна szWindowClass. Значення цієї перемінної – рядок тексту записується у файлі ресурсів у розділі STRINGTABLE

```
STRINGTABLE
BEGIN
   IDC_DATACREATOR    "DATACREATOR"
   IDS_APP_TITLE      "DataCreator"

END
```

Таким чином, на початку обміну повідомленнями з іншою програмою треба виконати наступне

```
hWndOther = FindWindow(classNameOther, NULL);
if (hWndOther)
   {      //потрібна програма вже функціонує
   . . . //надсилаємо повідомлення на адресу вікна hWndOther
   }
else      //потрібної програми немає
   {
   . . . //виклик на виконання потрібної програми
   }
```

Запуск програми на виконання іншою програмою

Уявимо собі, що зараз працює програма, яка хоче звернутися (наприклад, надіслати повідомлення) до іншої програми – а тої просто немає, вона ще не завантажена – її потрібно спочатку викликати. Для виклику будь-якої програми можна скористатися функцією API Windows **WinExec**

```
WinExec(LPCSTR lpCmdLine, UINT uCmdShow);
```

Рядок тексту **`lpCmdLine`** є командним рядком виклику програми. Цей рядок повинен містити ім’я виконуваного файлу потрібної програми. Наприклад:

```
WinExec("otherprog.exe", SW_SHOW);
```

Можна вказувати як повне ім’я виконуваного файлу, так і скорочене – буде шукатися вказана програма, у першу чергу, у поточній директорії. Якщо ім’я програми містить проміжки, то це ім’я треба записувати у лапки, наприклад:

```
WinExec("\"c:\\Program files\\otherprog.exe\"",
SW_SHOW);
```

Окрім імені програми, яка викликається, у командному рядку можуть записуватися параметри – слова, розділені проміжками. Часто у якості параметра вказується ім’я файлу, який ця програма повинна обробити одразу після виклику, наприклад

```
WinExec("otherprog.exe myfile.txt", SW_SHOW);
```

`otherprog.exe` – ім’я виконуваного файлу програми;

`myfile.txt` – файл, з яким ця програма повинна щось зробити, наприклад, завантажити дані з цього файлу, або, навпаки, записати дані у вказаний файл.

Крім того, у командному рядку можна можна вказувати й інші параметри. Скажімо, для того, щоб інша програма знала, яка програма її викликала – щоб потім надсилати їй повідомлення, ніхто не заважає у командному рядку записати також ідентифікатор вікна програми-викликача. Наприклад, якщо hWnd викликача дорівнює 3471, то

```
WinExec("otherprog.exe myfile.txt 3471", SW_SHOW);
```

Коли програма, яка буде викликана, розшифрує значення hWnd = 3471, то тепер вона, у свою чергу, може повідомити програмі, яка її викликала, значення hWnd свого вікна.

Як програма може узнати командний рядок, з яким її викликали? Це дуже просто – така можливість є стандартною для мов програмування C/C++. Командний рядок є параметром-аргументом функції **`main(arg. . .)`**. У середовищі Visual Studio проект програми у стилі Win API містить головний файл .cpp, у якому така функція є, проте, вона зветься трохи інакше (це ще залежіть від версії Visual Studio), наприклад

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,   //це командний
                                              //рядок
                       int       nCmdShow)
{
. . .
}
```

Третій параметр цієї функції – **`lpCmdLine`**, є вказівником на командний рядок виклику. Потрібно запрограмувати розбір (парсінг) вмісту цього рядка.

Як програмі найпростіше передати іншій програмі одне числове значення? Щоб надіслати іншій програмі 32-бітове значення **`value`**, можна порекомендувати надіслати повідомлення, наприклад, WM_COMMAND, у якому в якості параметра **`lParam`** буде записане потрібне значення:

```
PostMessage(hWndOther,WM_COMMAND,(WPARAM)wParam, (LPARAM)value);
```

Потрібно враховувати, що тип LPARAM для Win32 означає тип **long**.

Для третього параметра – wParam потрібно вказати значення, наприклад, 10000, яке не співпаде з жодним ідентифікатором пункта меню цієї програми.

Чому **PostMessage**, а не **SendMessage?** Функція PostMessage записує повідомлення у чергу повідомлень, а SendMessage надсилає позачергово. Порада: завжди користуйтесь PostMessage – тоді менша вірогідність конфіліктів у системі при обробці довготривалих процедур обробки

повідомлень. Використовуйте SendMessage тільки для окремих випадків, як вимагає документація по API Windows, наприклад, для WM_COPYDATA.

Коли надсилати іншій програмі (вікну hWndOther) значення hWnd свого вікна? Звичайно, можна будь-коли, але якщо потрібно надіслати це одразу після розшифрування командного рядка у функції WinMain, то можна порекомендувати зробити це впродовж обробки повідомлення WM_CREATE

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
   {
   . . .
   case WM_CREATE:
     if (hWndOther) PostMessage(hWndOther, WM_COMMAND, 10000,
                                (long)hWnd);
     break;
   . . .
```

Повідомлення WM_COPYDATA

Щоб надіслати масив даних іншій програмі, програма може зробити це за допомогою Windows-повідомлення WM_COPYDATA. Таке повідомлення надсилається функцією API Windows **SendMessage**

```
SendMessage(hWndDest,   WM_COPYDATA,   (WPARAM)hWndSrc,
(LPARAM)&cds);
```

де:

`hWndDest` – HWND вікна призначення (програми-отримувача)

`hWndSrc` – HWND вікна, яке надсилає (програми-передавача)

`cds` – структура типу COPYDATASTRUCT, яка містить дані, які надсилаються

Структура COPYDATASTRUCT має такі поля:

```
COPYDATASTRUCT
{
ULONG_PTR dwData;    //будь-яке ціле (32-бітове) число
DWORD cbData;        //кількість байтів даних, на які
                     //вказує член lpData
LPVOID lpData;       //адреса даних.
};
```

Таким чином можна надіслати одне числове значення (член **dwData**) плюс масив даних (його адреса – член **lpData**). Дані, які надсилаються не можуть містити вказівники або посилання на об'єкти, які є недоступними для іншої програми. Вказівник **lpData** можна використовувати тільки упродовж обробки повідомлення WM_COPYDATA. Дані тільки для читання. Якщо програмі-приймачу потрібно потім обробляти прийняті дані, то треба скопіювати їх у якійсь буфер.

Розглянемо як надіслати іншій програмі п'ять числових значень, наприклад, значень перемінних **nPoints,xMin,xMax,yMin,yMax**

```
long nPoints,xMin,xMax,yMin,yMax;
. . .

//--це наша власна функція--
int SendCopyData(HWND hWndDest, HWND hWndSrc, void *lp, long cb)
{
COPYDATASTRUCT cds;

cds.dwData = 1;     //а можна і будь-яке інше значення
cds.cbData = cb;
cds.lpData = lp;
return SendMessage(hWndDest, WM_COPYDATA, (WPARAM)hWndSrc,
(LPARAM)&cds);
}
. . .

//сформуємо дані як суцільний масив, наприклад, так
long params[5] = {nPoints, xMin, xMax, yMin, yMax};

//а тепер відправимо масив params вікну hWndOther іншої програми
SendCopyData(hWndOther, hWnd, params, sizeof(params));
```

Як приймати дані? Потрібно у функції вікна програми-приймача зробити обробник повідомлення WM_COPYDATA

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam)
{
switch (message)
   {
   . . .
   case WM_COPYDATA:
      OnCopyData(hWnd,wParam,lParam);     //це наша власна
                                          //функція-обробник
      break;
   . . .
```

Якщо OnCopyData для прийому тільки п'яти параметрів, то можна запрограмувати так:

```
void OnCopyData(HWND hWnd,WPARAM wParam,LPARAM lParam)
{
COPYDATASTRUCT *cds;
cds = (COPYDATASTRUCT *)lParam;
long *p = (long *)cds->lpData;
nPoints = p[0];
xMin = p[1];
xMax = p[2];
yMin = p[3];
yMax = p[4];
}
```

А якщо програма через повідомлення WM_COPYDATA повинна приймати різноманітні порції даних, то можна використати поле **cds->dwData** для ідентифікації кількості та типу даних, що надсилаються-отримуються.

Буфер обміну – Clipboard Windows

Усі користувачі Windows знайомі з цим буфером – майже будь-яка програма копіює у буфер якісь дані після натискування клавіш Ctrl-C. Потім дані з буфера можна вставити у потрібне місце (у тому числі у середовищі іншої програмі) натискуванням клавіш Ctrl-V. Такі натискування клавіш не є обов'язковими атрибутами Windows – програмісти повинні спеціально запрограмувати такі можливості для своїх програм (зауваження: це не зовсім так – для дочірніх вікон вводу стрічок, редагування тексту та у деяких інших випадках копіювання та вставка вже реалізовані на рівні операційної системи).

Буфер – Clipboard Windows призначений для обміну даними між програмами. У цей буфер можна записувати тексти, зображення, та, взагалі, будь-що.

Рис. 6.2. Буфер – Clipboard Windows

Для реалізації Clipboard Windows використовується глобальна динамічна віртуальна пам'ять. Рекомендується використовувати Clipboard для обміну даними обсягом не більше декількох десятків мегабайтів.

Для того, щоб програма записала дані у Clipboard, потрібно викликати функцію API Windows **SetClipboardData**. Ця функція має у якості параметрів ідентифікатор формату та *handle* блоку пам'яті, відкритого за допомогою функцій **GlobalAlloc**. Нижче наведений код функції, яку можна рекомендувати для запису ANSI-текстів у Clipboard

```
int PutTextToClipboard(HWND hWnd, char *src)
{
HGLOBAL hglbCopy;
BYTE *pTmp;
long len;

if (src == NULL) return 0;
if (src[0] == 0) return 0;
len = strlen(src);
hglbCopy = GlobalAlloc(GHND, len+1);
if (hglbCopy == NULL) return 0;
pTmp = (BYTE *)GlobalLock(hglbCopy);
memcpy(pTmp, src, len+1);
GlobalUnlock(hglbCopy);
if (!OpenClipboard(hWnd))
   {
   GlobalFree(hglbCopy);
   return 0;
   }
EmptyClipboard();
SetClipboardData(CF_TEXT, hglbCopy);
CloseClipboard();
return 1;
}
```

Примітка: цей приклад для проектів multibyte. Для Unicode – трохи інакше.

Приклад запису в Clipboard декількох рядків тексту – пар числових значень, розділених табуляцією

```
char text[] =
"x    y\r\n"
"1.0  3.12\r\n"
"2.0  5.3\r\n"
"3.0  6.11\r\n"
"4.0  7.1\r\n"
"5.0  8.05\r\n";

PutTextToClipboard(hWnd, text);
```

Якщо програма це зробить, то можна відкрити Блокнот Windows або інший текстовий редактор і вставити вміст Clipboard – натиснути Ctrl-V. У результаті повинно бути:

```
x   y
1.0 3.12
2.0 5.3
3.0 6.11
4.0 7.1
5.0 8.05
```

Для читання вмісту Clipboard у форматі ANSI-тексту можна скористатися функцією

```
long GetTextFromClipboard(HWND hWnd, char *dest, long maxsize)
{
HGLOBAL hglb;
LPTSTR lptstr;
long size, res;

res = 0;
if (!IsClipboardFormatAvailable(CF_TEXT)) return 0;
if (!OpenClipboard(hWnd)) return 0;
hglb = GetClipboardData(CF_TEXT);
if (hglb != NULL)
   {
   lptstr = (char *)GlobalLock(hglb);
   if (lptstr != NULL)
      {
      size = strlen(lptstr);
      if (size > maxsize)
         {
         lptstr[maxsize] = 0;
         size = strlen(lptstr);
         }
      strcpy(dest, lptstr);
      res = size;
      GlobalUnlock(hglb);
      }
   }
CloseClipboard();
return res;
}
```

Ця функція копіює вміст Clipboard у буфер, вказаний параметром dest. Примітка: цей приклад для проектів multibyte. Якщо проект орієнтований на Unicode – трохи інакше.

Алгоритм обміну повідомленнями об'єктів-компонентів системи

Програма Object1 спочатку просить щось зробити програму Object2. А потім у роботі приймає участь Object3.

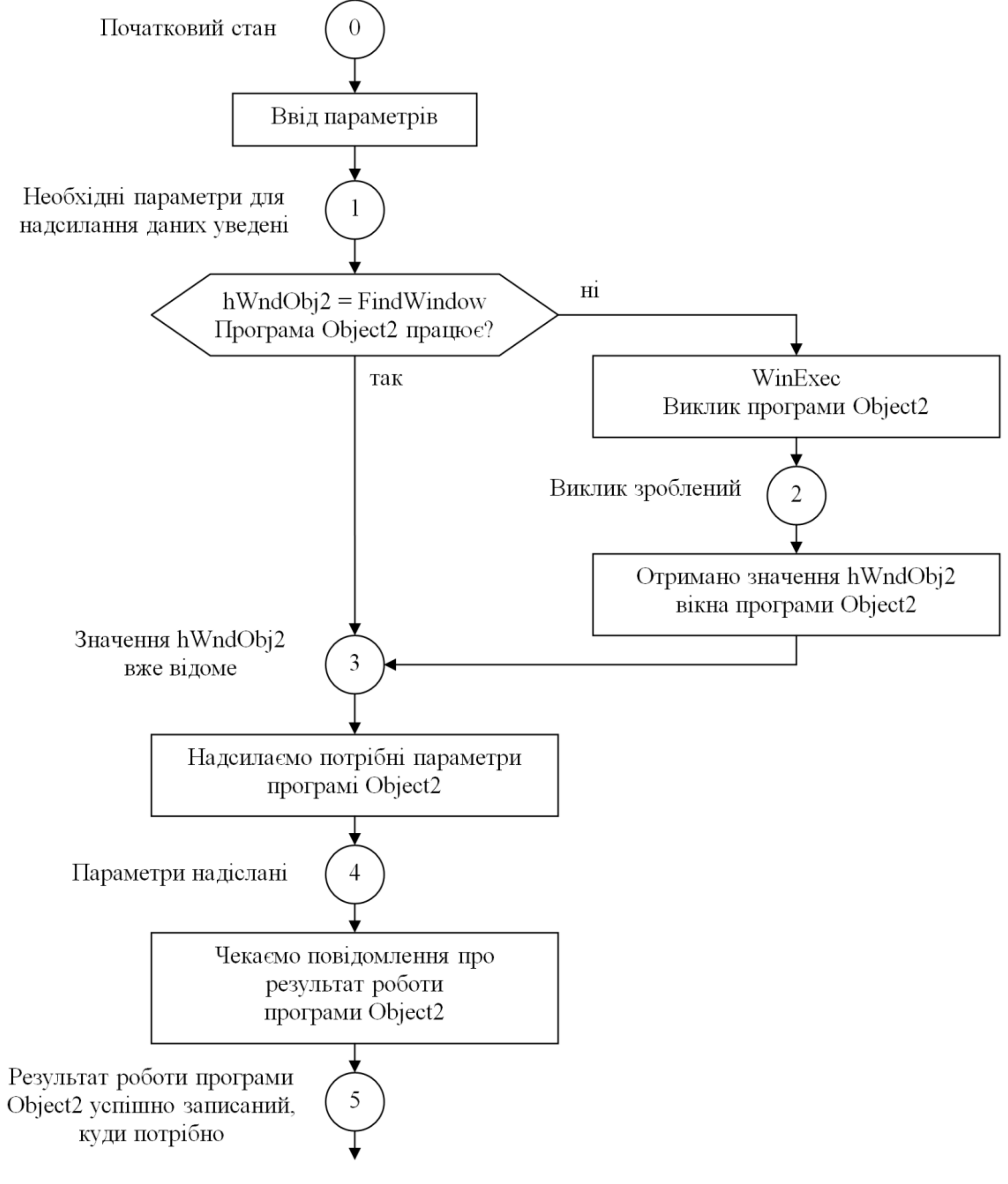

Рис. 6.3. Алгоритм взаємодії об'єктів – компонентів системи

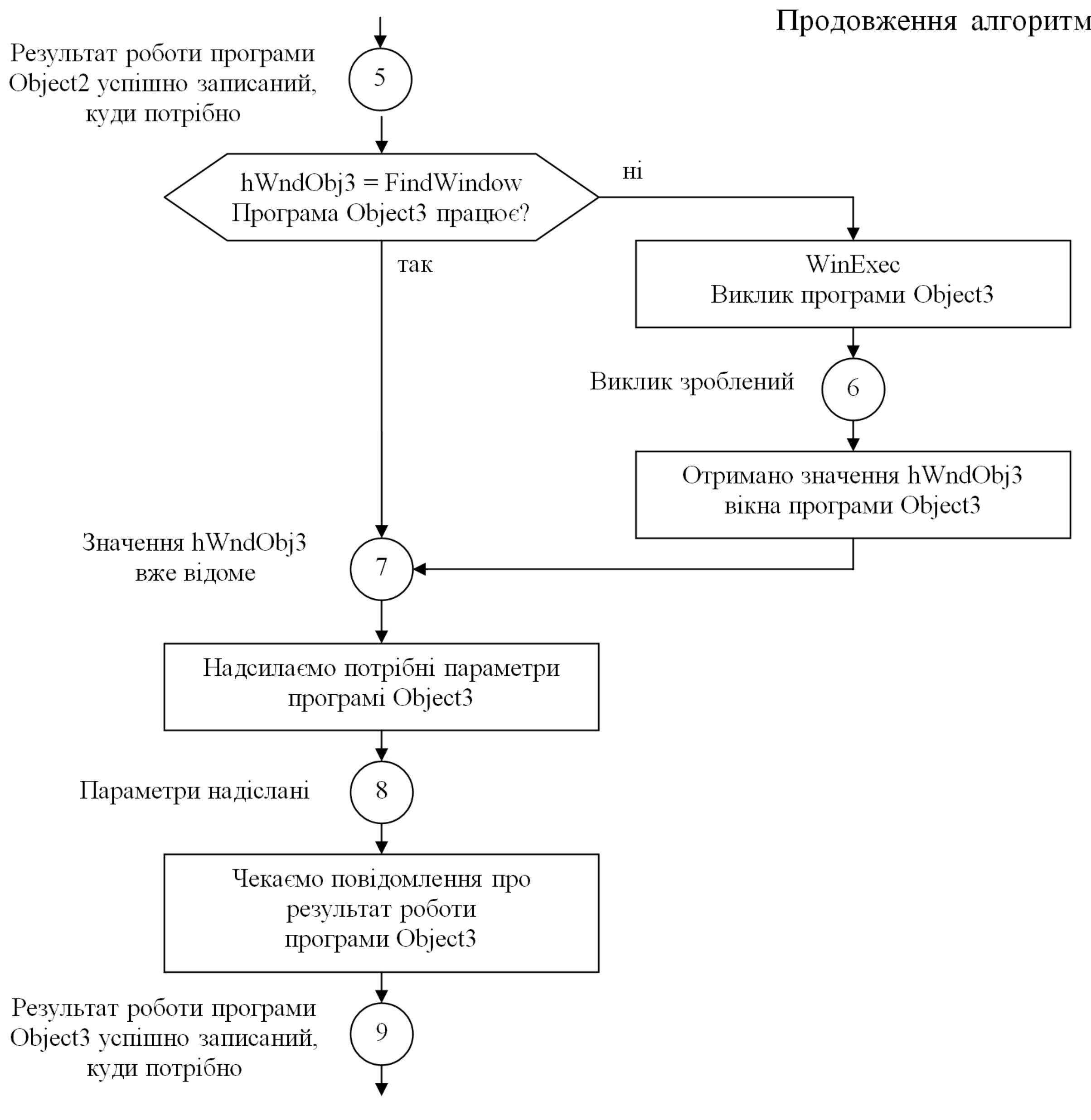

Рис. 6.4. Продовження алгоритму

Вимоги щодо організації системи

Для виконання лабораторної роботи потрібно створити три незалежні програми, для чого можна створити три проекта у одному рішенні (Solution) Microsoft Visual Studio C++. У цьому випадку усі виконувані файли будуть знаходитися у спільній папці \Debug або \Release.

Головний проект – програма **Lab6** має бути менеджером, який керує двома іншими програмами – **Object2** та **Object3**. Програма **Lab6** повинна автоматично, без участі користувача налагоджувати співпрацю та виконувати обмін повідомленнями з програмами **Object2** та **Object3** для виконання потрібного завдання згідно наведеному вище алгоритму.

У кожній програмі передбачити автоматичну обробку потрібних повідомлень і переходів з одного стану в інший.

Далі вимоги щодо функціонування системи об'єктів.

1. Для початку роботи користувач програми вибирає потрібний пункт меню програми **Lab6**. Далі з'являється вікно діалогу, у якому потрібно ввести параметри згідно варіанту завдання. У вікні діалогу користувач натискує кнопку "Так" (або "Виконати") і на цьому місія користувача закінчується – далі він тільки спостерігає, як програма сама автоматично виконає усе, що потрібно для отримання результату. Виклик інших програм – **Object2** та **Object3** головна програма **Lab6** повинна робити без участі користувача.

2. Обмін повідомленнями та масивами даних між програмами **Lab6, Object2** та **Object3** повинен відбуватися автоматично, без участі користувача. Програма **Lab6** повинна автоматично у певній послідовності знаходити та викликати програми **Object2** та **Object3.**

3. У результаті одного сеансу роботи користувач повинен бачити головні вікна програм **Object2** та **Object3**, у яких відображатимуться потрібні результати відповідно варіанту завдань. Для цього вікна програм повинні автоматично розташуватися так, щоб усі результати було видно. Програма **Lab6** повинна залишатися у активному стані, щоб користувач мав можливість повторно виконати роботу.

4. Передбачити варіанти успішної роботи у випадках, коли програми **Object2** та **Object3** (одна або обидві) до того вже були викликані.

5. По завершенні роботи програми **Lab6** повинні автоматично завершуватися і програми **Object2** та **Object3.**

Вибір варіанту завдання

Номер варіанту завдання згідно формулі

Номер варіанту = Ж mod 4,
де: Ж – номер в списку студентів в журналі. Зміст завдань у таблиці 6.1

Таблиця 6.1

Номер варіанту	Програма Lab6	Програма Object2	Програма Object3
0	1. Користувач вводить значення параметрів **nPoint, xMin, xMax, yMin, yMax** у вікні діалогуі. 2. Програма викликає програми Object2, 3 і забезпечує обмін повідомленнями для передавання та отримання інформації.	1. Створює **nPoint** пар цілих (**int**) значень (x,y) в діапазонах **xMin** – **xMax**, **yMin** – **yMax**. 2. Показує числові значення у власному головному вікні 3. Записує дані в Clipboard Windows у текстовому форматі	1. Зчитує дані з Clipboard Windows 2. Відображає графік **y=f(x)** у власному головному вікні. Графік, як в математиці – лінія, що проходить через точки (x,y) в порядку зростання **x**; осі координат з підписами числових значень x, y.
1	1. Користувач вводить значення **n, Min, Max** у діалоговому вікні. 2. Програма викликає програми Object2, 3 і забезпечує обмін повідомленнями для передавання та отримання інформації.	1. Створює матрицю **n×n** цілих (**int**) чисел у діапазоні **Min – Max** 2. Показує числові значення у власному головному вікні 3. Записує дані в Clipboard Windows у текстовому форматі	1. Зчитує дані з Clipboard Windows 2. Відображає значення детермінанту матриці у власному головному вікні
2	1. Користувач вводить значення **n, Min, Max** у діалоговому вікні. 2. Програма викликає програми Object2, 3 і	1. Створює вектор **n** дробових (**double**) чисел у діапазоні **Min – Max** 2. Показує числові значення у декількох	1. Зчитує дані з Clipboard Windows 2. Виконує сортування масиву чисел і відображає його у декількох стовпчиках та рядках у власному

	забезпечує обмін повідомленнями для передавання та отримання інформації.	стовпчиках та рядках у власному головному вікні 3. Записує дані в Clipboard Windows у текстовому форматі	головному вікні
3	1. Користувач вводить значення **n, Min, Max** у діалоговому вікні. 2. Програма викликає програми Object2, 3 і виконує обмін повідомленнями з ними для передавання, отримання інформації.	1. Створює вектор **n** дробових (**double**) чисел у діапазоні **Min – Max** 2. Показує числові значення у декількох стовпчиках та рядках у власному головному вікні 3. Записує дані в Clipboard Windows у текстовому форматі	1. Зчитує дані з Clipboard Windows 2. Відображає графік **y=f(x)** у власному головному вікні Значення **y** – це значення вектора, **x** – індекси елементів. Графік, як в математиці – лінія, що проходить через точки (x,y) в порядку зростання **x**; осі координат з підписами числових значень x, y.

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний текст файлів .cpp трьох програм
4. Схема послідовності надсилання-обробки повідомлень
5. Ілюстрації (скріншоти)
6. Висновки

Контрольні запитання

1. Як можна тлумачити поняття об'єктної орієнтованості?
2. Як об'єкт може надіслати повідомлення іншому об'єкту?
3. Як запрограмувати виклик у програмі іншої програми?
4. Як надіслати іншій програмі повідомлення?
5. Як надіслати іншій програмі одне число?
6. Як надіслати іншій програмі масив числових значень?
7. Як запрограмувати обмін інформацією з Clipboard Windows?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

Базова

1. В.М. Порєв. Об'єктно-орієнтоване програмування. Конспект лекцій: Навч. посібник. [Електронний ресурс] / В.М. Порєв. – Електронні текстові дані (1 файл: 4,8 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 271 с. Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол №1 від 02.09.2022р.) за поданням Вченої ради факультету інформатики та обчислювальної техніки (протокол №11 від 11.07.2022 р.)

2. Microsoft. Creating and Managing Visual C++ Projects [Online] Available from: https://msdn.microsoft.com/en-us/library/4457htyc.aspx

3. Microsoft. Overview of Windows Programming in C++ [Online] Available from: https://docs.microsoft.com/en-us/cpp/windows/

Додаткова

4. Charles Petzold. Programming Windows [Online] Available from: https://docs.google.com/file/d/0B73JwvIHVHaiSFdpekJCOUdoeE0/view?resourcekey=0-JFez95uS9jETJQdFGKY_1w

5. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley ISBN-13: 978-0201633610

6. Gerbert Schildt. C++: A Beginner's Guide ISBN-13: 978-0072232158, 2012, 542 p.

7. Samuel Oloruntoba. SOLID: The First 5 Principles of Object Oriented Design [Online] Available from: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

8. The Unified Modeling Language [Online] Available from: https://www.uml-diagrams.org/