Лабораторна робота №4

Вдосконалення структури коду графічного редактора об’єктів на C++

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.

Порядок виконання роботи та 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 лаб. 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 у клієнтській частині вікна
    ED.OnLBdown(hWnd);
    break;
  case WM_LBUTTONUP:        //відпущено ліву кнопку миші у клієнтській частині вікна
    ED.OnLBup(hWnd);
    break;
  case WM_MOUSEMOVE:        //пересунуто мишу у клієнтській частині вікна
    ED.OnMouseMove(hWnd);
    break;
  case WM_PAINT:            //потрібно оновлення зображення клієнтської частини 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 PointShape);
     break;
  case IDM_LINE:
     ED.Start(new LineShape);
     break;
  case IDM_RECT:
     ED.Start(new RectShape);
     break;
  case IDM_ELLIPSE:
     ED.Start(new EllipseShape);
     break;

  case IDM_SOMETHING:
     ED.Start(new Something);    //ще якійсь тип фігур з ієрархії класів Shape
     break;

  . . .                //щось ще
```

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;

  . . .                //щось ще

ped = new MyEditor;    //динамічне створення об'єкта-редактора

  . . .                //щось ще

  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 Windows, надійніше так:

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

Гумовий слід

При вводі об'єкта його "гумовий" слід повинен вимальовуватися поліморфним викликом відповідної функції-члена 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**:

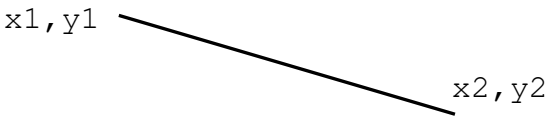
```
//---оголошення базового класу---
class Shape
{
protected:
   long xs1,ys1,xs2,ys2;
public:
   void Set(long x1,long y1,long x2,long y2);
   virtual void Show(HDC) = 0;    //визначається у похідних класах
};

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

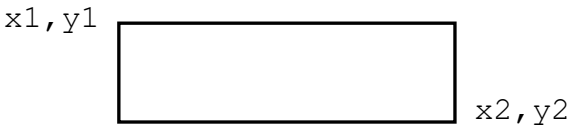
Від класу **Shape** утворюються похідні класи: **PointShape**, **LineShape**, **RectShape**, **EllipseShape.** У кожному похідному класі визначається метод показу **Show** відповідно до геометрії об'єкта.

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

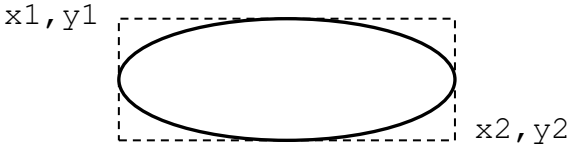
2. У класі **LineShape** лінія (відрізок прямої) малюється викликом функцій
`MoveToEx(hdc,x1,y1,NULL);`
`LineTo(hdc,x2,y2);`



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



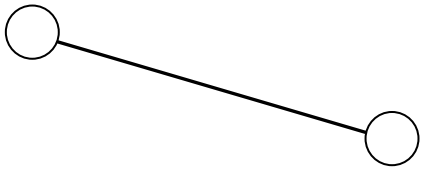
4. У класі **EllipseShape** еліпс малюється викликом функції
`Ellipse(hdc,x1,y1,x2,y2);` (еліпс із заповненням)
або `Arc(hdc, x1,y1, x2,y2, 0,0,0,0);` (якщо потрібен тільки контур).



Нові геометричні форми та множинне успадкування

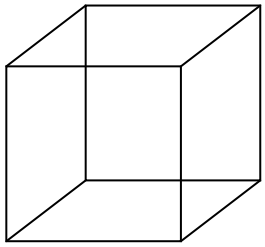
5. Лінія з кружочками на кінцях

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



6. Каркас кубу

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



Примітка. Необхідно **забезпечити коректну підтримку поліморфізму** для усіх типів об'єктів – як старих та нових.

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 і файлів.

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

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 лаб. роботі №3. "Гумовий" слід усіх фігур малювати пунктирною лінією викликом відповідної функції-члена класів Shape. Можливо (але не обов’язково) для малювання гумового сліду-контуру внести додаткову функцію у класи Shape.

4. Окрім чотирьох типів форм, які були у попередніх лаб. №2 та 3, запрограмувати ще введення та відображення двох нових фігур – лінія з кружочками та каркас куба.
Кольори ліній та заповнення цих нових фігур студент визначає на свій смак.
Відповідні класи цих фігур запрограмувати саме **множинним успадкуванням**. У цій лабі не можна замінювати множинне спадкування, наприклад, композицією.

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

6. Не створювати ієрархію класів-редакторів для вводу об'єктів різних форм (як було у попередніх лаб. №2, 3.

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

8. Документи звіту – тексти, діаграми, схеми тощо оформлювати у електронному форматі так, щоб їх легко було сприймати у надрукованому звіті. Забороняється текст або графіка "світле на світлому фоні" або "темне на темному фоні". Тільки чорний текст та чорні лінії на білому фоні. Оформлення звіту впливатиме на оцінку.

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?

У ході захисту-прийняття роботи викладач може також запитувати інше, що стосується виконання роботи.

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