Міністерство освіти і науки України
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Об’єктно-орієнтоване програмування

Методичні вказівки до виконання лабораторних робіт для студентів денної форми навчання за спеціальністю 122 «Комп’ютерні науки», 123 «Комп’ютерні інженерія», 125 «Кібербезпека»

ЗАТВЕРДЖЕНО
на засіданні кафедри кібербезпеки та програмного забезпечення, протокол №10 від 25.03.2025 року.

Кропивницький
2025

УДК 004.4

Об’єктно-орієнтоване програмування: методичні вказівки до виконання лабораторних робіт для студентів за спеціальностями 122 «Комп’ютерні науки», 123 «Комп’ютерні інженерія», 125 «Кібербезпека»/ М-во освіти і науки України, Центральноукраїнський нац. техн. ун-т; [уклад. П.С. Усік, Н.Л. Козірова, Р.О. Ткачук] – Кропивницький: ЦНТУ, 2024. – 114с.

Укладачі: Усік П. С., доктор філософії, старший викладач;
Козірова Н.Л., викладач.
Ткачук Р.О., асистент

Рецензенти: Смірнов О. А., докт. техн. наук, професор;
Коваленко О. В., докт. техн. наук, професор.

ЗМІСТ

ВСТУП

Мета: ознайомлення з ключовими концепціями об'єктно-орієнтованого програмування та навчання навичкам розробки програм за допомогою мови програмування C++. Вивчення структури класів, взаємодії об'єктів, спадкування та поліморфізму. Набуття практичного досвіду вирішення завдань програмування з використанням об'єктно-орієнтованого підходу на базі мови C++.

Завдання:

– Вивчення теоретичних основ об’єктно-орієнтованого програмування.

– Набути практичних навичок роботи по впровадженню парадигми об’єктно-орієнтованого програмування.

У результаті вивчення навчальної дисципліни студент повинен:

– знати новітні технології в галузі інформаційних технологій. Використовувати сучасні методи і мови програмування для розроблення алгоритмічного та програмного забезпечення. Визначати критерії, яким повинен задовольняти проект, щоб його легко було супроводжувати і модифікувати. Створювати системне та прикладне програмне забезпечення комп’ютерних систем та мереж. Використовувати класи-шаблони стандартної бібліотеки мови C ++ (STL) та узагальнені алгоритми при написанні програм.) та узагальнені алгоритми при написанні програм. Застосовувати шаблони (патерни).

– вміти застосовувати знання для ідентифікації, формулювання і розв’язування технічних задач спеціальності, використовуючи методи, що є найбільш придатними для досягнення поставлених цілей.

Структурно логічна схема підготовки бакалавра.

Враховуючи послідовність накопичення знань та інформації, дисципліна вивчається після викладання наступних дисциплін: «Базові методології та технології програмування».

Для опанування матеріалу дисципліни «Об’єктно-орієнтоване програмування» окрім лекційних та лабораторних занять, тобто аудиторного навантаження, значна увага приділяється самостійній роботі.

До основних видів самостійної роботи студента відносимо:

1. Вивчення лекційного матеріалу.

2. Робота з літературними джерелами.

3. Розв’язання практичних задач.

4. Підготовка до модульних, підсумкового контролю, екзамену (денна) та заліку (заочна).

5. Виконання контрольної роботи для заочної форми навчання.

В ході викладання дисципліни викладачем застосовуються види занять, які згідно з програмою навчальної дисципліни передбачають лекційні, та лабораторні заняття, а також виконання самостійної роботи.

Основна мета лекції – дати систематизовані основи знань з навчальної дисципліни, зосередити увагу студентів на найбільш складних та ключових питаннях.

Основна мета лабораторної роботи – закріплення й деталізація знань, а головне – формування навичок і вмінь.

Шкала оцінювання: національна та ЄКТС

Сума балів за всі види навчальної діяльності	Оцінка ЄКТС	Оцінка за національною шкалою
		для екзамену
90-100	A	відмінно
82-89	B	добре
74-81	C	
64-73	D	задовільно
60-63	E	
35-59	FX	незадовільно з можливістю повторного складання
1-34	F	незадовільно з обов’язковим повторним вивченням дисципліни

Вибравши предметну область, над якою ви будете працювати, ви повинні виконати завдання до лабораторних робіт, а також відповісти на питання в кінці кожної лабораторної роботи. Звіт повинен містити хід виконання завдань а також графічні матеріали, що підтверджують виконання цих завдань.

Лабораторна робота №1 Основні поняття ООП. Класи та об’єкти. Функції доступу. Конструктори і деструктори

Мета: ознайомитись з основними поняттями ООП. Вивчити поняття клас, об’єкт, сеттер, геттер та навчитись їх програмно реалізовувати мовою С++.

Теоретична частина

Клас - це конструкція в ООП, яка представляє собою шаблон або опис структури та поведінки об'єктів. Клас є абстракцією, яка групує дані (властивості) та функції (методи), що пов'язані між собою логічно. Він визначає набір атрибутів (члени класу), які представляють стан об'єктів даного класу, а також методи, які визначають операції, які можуть бути виконані над цими об'єктами.

Класи визначаються за допомогою ключового слова class та мають ім'я, за допомогою якого їх можна ідентифікувати. Клас може мати конструктори, які використовуються для ініціалізації об'єктів класу, а також може містити статичні поля та методи, які використовуються для спільного використання даних між усіма об'єктами даного класу.

Об'єкт - це конкретний екземпляр класу. Він створюється на основі класу шляхом процесу, який називається інстанціюванням або створенням об'єкта. Кожен об'єкт має свій власний набір значень властивостей, які відображають його стан, та може виконувати методи, що визначені в класі.

Об'єкти зберігають свій внутрішній стан у вигляді значень властивостей, які можуть бути унікальними для кожного об'єкта. Методи об'єктів представляють собою операції або функції, які можуть бути виконані над цими об'єктами. Об'єкти взаємодіють один з одним шляхом виклику методів та обміну даними.

Основна ідея ООП полягає в організації програми як набору взаємодіючих об'єктів, які співпрацюють для вирішення певних задач. Класи визначають загальну структуру та поведінку, а об'єкти є конкретними представниками цих класів, здатними зберігати стан та виконувати дії, визначені класом.

Наприклад:

```
#include <iostream>
using namespace std;

// Клас "Автомобіль"
class Car {
public:
    string brand;
    string model;
    int year;

    void startEngine() {
        cout << "Двигун автомобіля запущено!" << endl;
    }

    void stopEngine() {
        cout << "Двигун автомобіля зупинено!" << endl;
    }
};

int main() {
    // Створення об'єкта класу "Автомобіль"
    Car myCar;

    // Налаштування властивостей об'єкта
    myCar.brand = "BMW";
    myCar.model = "X5";
    myCar.year = 2022;

    // Виклик методів об'єкта
    cout << "Марка автомобіля: " << myCar.brand << endl;
    cout << "Модель автомобіля: " << myCar.model << endl;
    cout << "Рік випуску автомобіля: " << myCar.year << endl;

    myCar.startEngine(); // Запуск двигуна
    myCar.stopEngine(); // Зупинка двигуна

    return 0;
}
```

У цьому прикладі ми оголосили клас "Car", який представляє автомобіль. Він має властивості brand, model та year, а також методи startEngine() та stopEngine(), які відповідають за запуск та зупинку двигуна.

У функції main() ми створюємо об'єкт myCar класу Car. Потім ми налаштовуємо властивості об'єкта, використовуючи оператор крапки (.), і викликаємо методи об'єкта.

В результаті на консоль будуть виведені властивості автомобіля (марка, модель, рік випуску), а також повідомлення про запуск та зупинку двигуна.

Функції доступу (сеттери та геттери) є методами, використовуваними в об'єктно-орієнтованому програмуванні (ООП) для отримання (читання) та встановлення (запису) значень приватних членів класу. Ці методи надають контроль доступу до даних класу та допомагають забезпечити принцип інкапсуляції, що означає, що дані класу знаходяться внутрішньо і мають обмежений доступ ззовні.

Геттери (getter methods) - це методи, які використовуються для отримання (читання) значень приватних членів класу. Вони зазвичай повертають значення цих членів і мають повернутий тип, що відповідає типу члена класу. Геттери дозволяють зовнішньому коду отримувати доступ до значень приватних даних класу без безпосереднього доступу до них. Вони можуть бути корисними для отримання значень для подальшого використання або виведення на екран.

Наприклад, у класі "Person" може бути приватний член name, і для отримання цього значення може бути створений геттер:

```
class Person {
private:
    string name;

public:
    // Геттер для отримання значення name
    string getName() {
        return name;
    }
};
```

Сеттери (setter methods) - це методи, які використовуються для встановлення (запису) значень приватних членів класу. Вони приймають параметр з новим значенням і встановлюють це значення для відповідного приватного члена класу. Сеттери дозволяють контролювати доступ до приватних даних класу і забезпечувати перевірку або обробку значень перед їх збереженням.

Продовжуючи приклад з класом "Person", може бути створений сеттер для встановлення нового значення для name:

```
class Person {
private:
    string name;
```

```
public:
    // Геттер для отримання значення name
    string getName() {
        return name;
    }

    // Сеттер для встановлення значення name
    void setName(string newName) {
        name = newName;
    }
};
```

За допомогою сеттера зовнішній код може встановити нове значення для name, і внутрішній код класу може здійснити додаткову перевірку або обробку цього значення перед збереженням.

Використання геттерів та сеттерів дозволяє забезпечити кращий контроль над доступом до приватних даних класу, зберігаючи їх недоступними безпосередньо ззовні класу і забезпечуючи правильність операцій з цими даними.

Ключове слово this в об'єктно-орієнтованому програмуванні (ООП) вказує на поточний об'єкт, з яким пов'язаний виклик методу або доступ до членів класу. Воно представляє вказівник на сам об'єкт, на якому викликається метод або з якого доступ до членів класу.

Ключове слово this може використовуватись в контексті класу для посилання на його власні члени (змінні та методи). Це особливо корисно, коли ім'я параметра методу або локальної змінної конфліктує з іменем члена класу. Використання this дозволяє уникнути неоднозначностей і вказати на конкретний член класу.

Наприклад, розглянемо клас "Person" з методом "setName", який встановлює значення приватного члена "name":

```
class Person {
private:
    string name;

public:
    void setName(string name) {
        this->name = name;
    }
};
```

У цьому прикладі параметр методу "setName" має таке ж ім'я, як і приватний член "name". Використання this->name дозволяє чітко вказати, що ми звертаємось саме до члена класу "name", а не до параметра методу.

Крім того, ключове слово this може бути корисним при передачі посилання на поточний об'єкт як аргумент у виклику методу або при створенні ланцюжка методів.

Загалом, this використовується для доступу до членів класу зсередини самого класу і допомагає уникнути конфліктів імен, які можуть виникати між параметрами методів та членами класу.

Конструктор є спеціальним методом в класі, який викликається автоматично при створенні нового об'єкта цього класу. Його основна функція полягає в ініціалізації об'єкта, тобто встановленні його початкового стану.

У мові C++, конструктор має той самий ім'я, що й клас, і не має повертаючого значення (навіть не вказується тип void). Конструктор може мати параметри, але також може бути й без параметрів. Зазвичай використовуються параметри конструктора для передачі значень до членів даних об'єкта.

Основні види конструкторів:

- **Конструктор за замовчуванням (default constructor):** Це конструктор без параметрів, який викликається при створенні об'єкта без передачі яких-небудь значень. Він може ініціалізувати члени даних об'єкта значеннями за замовчуванням.

Наприклад:

```
class MyClass {
public:
    MyClass() {
        // Конструктор за замовчуванням
        // Виконання початкових операцій
    }
};
```

- **Параметризований конструктор (parameterized constructor):** Це конструктор, який приймає параметри і використовує їх для ініціалізації членів даних об'єкта.

Наприклад:

```
class Point {
private:
    int x, y;
public:
    Point(int xCoord, int yCoord) {
        x = xCoord;
        y = yCoord;
    }
};
```

• **Конструктор копіювання (copy constructor):** Це конструктор, який приймає об'єкт того ж класу як параметр і створює новий об'єкт, який є копією цього переданого об'єкта. Конструктор копіювання використовується, коли потрібно зробити копію об'єкта для подальшої роботи з ним, а не просто посилання на нього.

Наприклад:

```
class MyClass {
public:
    MyClass(const MyClass& other) {
        // Конструктор копіювання
        // Виконання операцій копіювання
    }
};
```

Конструктори дозволяють забезпечити коректну ініціалізацію об'єктів і можуть бути потужним інструментом у процесі програмування на C++.

Деструктор є спеціальним методом в класі, який викликається автоматично під час звільнення пам'яті, знищення об'єкта або виходу з блоку, в якому об'єкт був створений. Він використовується для виконання ресурсоозберігаючих операцій або очищення пам'яті, які можуть бути пов'язані з об'єктом.

У мові C++, деструктор має таку ж назву, як і клас, до якого він належить, перед якою ставиться символ тильда (~). Наприклад, якщо у нас є клас під назвою "MyClass", то його деструктор буде мати назву "~MyClass".

Деструктор використовується для звільнення ресурсів, таких як пам'ять, виділену динамічно, або закриття файлів, відкритих об'єктом класу. Це може

бути корисно, наприклад, для видалення об'єктів, які були створені в конструкторі, або для закриття відкритих з'єднань з базами даних.

Деструктор викликається автоматично безпосередньо перед знищенням об'єкта. Він може бути визначений в класі та містити будь-який код, необхідний для очищення ресурсів. Якщо деструктор не визначений в класі, то компілятор створює деструктор за замовчуванням, який не виконує ніяких додаткових дій.

Особливістю деструктора є його автоматичне викликання, що дозволяє забезпечити коректне звільнення ресурсів та уникнути витоку пам'яті або інших проблем, пов'язаних з неочищенням ресурсів, коли об'єкт виходить з області видимості.

Приклад використання:

```
#include <iostream>

class MyClass {
private:
    int data;

public:
    // Конструктор за замовчуванням
    MyClass() {
        data = 0;
         std::cout << "Конструктор за замовчуванням викликаний!"
<< std::endl;
    }

    // Конструктор з параметрами
    MyClass(int value) {
        data = value;
         std::cout << "Конструктор з параметрами викликаний!" <<
std::endl;
    }

    // Копіюючий конструктор
    MyClass(const MyClass& other) {
        data = other.data;
           std::cout << "Копіюючий конструктор викликаний!" <<
std::endl;
    }

    // Деструктор
    ~MyClass() {
        std::cout << "Деструктор викликаний!" << std::endl;
    }

    // Метод для отримання значення даних
    int getData() {
```

```
        return data;
    }
};

int main() {
    // Виклик конструкторів та деструкторів

    // Об'єкт створений за замовчуванням
    MyClass obj1;
    std::cout << "obj1.data: " << obj1.getData() << std::endl;

    // Об'єкт створений з параметрами
    MyClass obj2(10);
    std::cout << "obj2.data: " << obj2.getData() << std::endl;

    // Копіюючий конструктор
    MyClass obj3 = obj2;
    std::cout << "obj3.data: " << obj3.getData() << std::endl;

    return 0;
                                }
```

Завдання

Варіант 1

Завдання 1

Створіть клас Book з приватними полями title, author, year. Реалізуйте сетери та гетери: setTitle(), getTitle(), setAuthor(), getAuthor(), setYear(), getYear(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через сетери та виведіть інформацію про книгу на екран за допомогою гетерів.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 2

Завдання 1

Створіть клас BankAccount з приватними полями balance (double), accountNumber (string), ownerName (string). Реалізуйте сетери та гетери для кожного поля, методи deposit() для поповнення рахунку та withdraw() для зняття коштів, конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів, змініть баланс за допомогою методів deposit() і withdraw(), та виведіть інформацію про рахунок на екран.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 3

Завдання 1

Створіть клас Person з приватними полями name, age, address. Реалізуйте сетери та гетери: setName(), getName(), setAge(), getAge(), setAddress(), getAddress(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через сетери та виведіть інформацію про людину на екран за допомогою гетерів.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 4

Завдання 1

Створіть клас Employee з приватними полями name, id, salary. Реалізуйте сетери та гетери: setName(), getName(), setId(), getId(), setSalary(), getSalary(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через сетери та виведіть інформацію про співробітника на екран за допомогою гетерів.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 5

Завдання 1

Створіть клас Student з приватними полями name, age, major. Реалізуйте сетери та гетери: setName(), getName(), setAge(), getAge(), setMajor(), getMajor(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через сетери та виведіть інформацію про студента на екран за допомогою гетерів.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 6

Завдання 1

Створіть клас Animal з приватними полями name, species, age. Реалізуйте сетери та гетери: setName(), getName(), setSpecies(), getSpecies(), setAge(), getAge(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через сетери та виведіть інформацію про тварину на екран за допомогою гетерів.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 7

Завдання 1

Створіть клас Movie з приватними полями title, director, year, duration. Реалізуйте сетери та гетери: setTitle(), getTitle(), setDirector(), getDirector(), setYear(), getYear(), setDuration(), getDuration(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через сетери та виведіть інформацію про фільм на екран за допомогою гетерів.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 8

Завдання 1

Створіть клас TravelDestination з приватними полями name, country, rating. Реалізуйте сетери та гетери: setName(), getName(), setCountry(), getCountry(), setRating(), getRating(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через сетери, виведіть інформацію про туристичний напрямок за допомогою гетерів, змініть значення рейтингу через метод setRating() і знову виведіть оновлену інформацію на екран.

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх необхідними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 9

Завдання 1

Створіть клас Country з приватними полями name, capital, population. Реалізуйте публічні методи setName(), getName(), setCapital(), getCapital(), setPopulation(), getPopulation(), конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте назву країни за допомогою setName(), виведіть її на екран через getName(), задайте столицю за допомогою setCapital() і виведіть її через getCapital(), задайте населення за допомогою setPopulation() і виведіть його через getPopulation().

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх доцільними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Варіант 10

Завдання 1

Створіть клас Company з приватними полями name, revenue, expenses. Реалізуйте публічні методи setName(), getName(), setRevenue(), getRevenue(), setExpenses(), getExpenses(), метод calculateProfit(), який обчислює прибуток компанії (прибуток = прибуток - витрати), та метод getProfit(), який повертає прибуток. Додайте конструктор за замовчуванням, конструктор з параметрами та деструктор, який виводить повідомлення при видаленні об’єкта.

У функції main() створіть об’єкт класу, задайте значення полів через відповідні методи, виведіть назву, прибуток і витрати на екран за допомогою гетерів, виконайте обчислення прибутку через calculateProfit() і виведіть його через getProfit().

Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

У вашому рішенні можуть бути додаткові методи та поля, якщо ви вважаєте їх доцільними.

Завдання 2

Реалізувати вище наведену задачу за допомогою структурного програмування. У висновку описати різницю цих методів.

Лабораторна робота №2 Типи зв'язків між класами: асоціація, агрегація, композиція, реалізація та залежність

Мета: ознайомитись з основними поняттями асоціація, агрегація, композиція, реалізація та залежність в ООП та навчитись їх програмно реалізовувати мовою С++.

Теоретична частина

Зв'язки між класами є фундаментальним елементом об'єктно-орієнтованого програмування, оскільки вони визначають, як об'єкти взаємодіють і співпрацюють у межах системи. Через ці зв'язки реалізується обмін даними, виклик методів і спільне виконання логіки, що дозволяє створювати складні та масштабовані програми на основі простих взаємодіючих компонентів.

Правильне моделювання зв'язків між класами має критичне значення для архітектури програмного забезпечення, адже від нього залежить зрозумілість, підтримуваність і розширюваність системи. Чітке визначення типів зв'язків допомагає уникнути надмірної залежності між компонентами, зменшує ризики помилок і полегшує подальший розвиток та тестування програми.

Залежність (Dependency) — це тимчасовий зв'язок між класами, коли один клас використовує інший лише під час виконання певної операції або методу, але не зберігає посилання на цей клас як свій член. Це означає, що клас A залежить від класу B, якщо A використовує B у своїй роботі, але не є його власником.

```
class Engine {
public:
    void start() { // Код запуску двигуна
    }
};

class Car {
public:
    void run() {
        Engine engine;  // Залежність від класу Engine
        engine.start(); // Виклик методу класу Engine
```

```
    }
};
```

У цьому прикладі клас Car має залежність від класу Engine, оскільки всередині методу run() він створює об'єкт Engine і виконує його метод start(). Однак Car не зберігає цей об’єкт як член класу, а використовує його лише тимчасово. Це і є прикладом залежності між класами.

Асоціація (Association) — це триваліший зв’язок між двома класами, який означає, що об’єкти одного класу можуть посилатися на об’єкти іншого класу і взаємодіяти з ними. При асоціації клас A і клас B знають один про одного і можуть викликати методи або отримувати доступ до даних. Асоціація може бути однонаправленою або двонаправленою.

```
class Engine {
public:
    void start() {
        // Код запуску двигуна
    }
};

class Car {
private:
    Engine* engine;  // Асоціація — Car має посилання на Engine

public:
    Car(Engine* eng) : engine(eng) {}

    void run() {
        engine->start();  // Виклик методу через посилання
    }
};

int main() {
    Engine myEngine;
      Car myCar(&myEngine);   // Передаємо посилання на об'єкт
Engine в Car
    myCar.run();
}
```

У цьому прикладі клас Car має асоціацію з класом Engine через вказівник engine. Об’єкт Car зберігає посилання на об’єкт Engine і може викликати його методи. На відміну від залежності, тут зв’язок тривалий — Car постійно співпрацює з певним Engine.

Агрегація є одним з важливих концепцій в ООП і відноситься до відношення між класами, де один клас представляє контейнер, що містить

об'єкти іншого класу. У випадку агрегації, об'єкти-члени, які належать до контейнера, можуть існувати самостійно і не залежати від контейнера.

Зазвичай агрегація відбувається за допомогою включення (composition) або посилання (reference). У випадку включення, контейнер містить об'єкти-члени як свої прямі члени (наприклад, об'єкти класів або структур), і вони створюються та знищуються разом з контейнером. У випадку посилання, контейнер містить посилання або вказівники на об'єкти-члени, і вони можуть існувати незалежно від контейнера.

Наприклад, давайте розглянемо клас "Автомобіль" і клас "Двигун". У цьому випадку, клас "Автомобіль" може мати агрегаційне відношення з класом "Двигун". Автомобіль може містити об'єкт "Двигун" як свій прямий член, що означає, що двигун створюється та знищується разом з автомобілем. Або ж клас "Автомобіль" може містити посилання на об'єкт "Двигун", що дозволяє розділяти двигун між декількома автомобілями.

Агрегація дозволяє створювати більш складні структури і групувати пов'язані об'єкти разом для зручної організації та керування. Вона також сприяє розподілу відповідальностей і забезпечує зв'язок між класами.

Наприклад:

```
#include <iostream>

// Клас "Двигун"
class Engine {
public:
    void start() {
        std::cout << "Engine started!" << std::endl;
    }

    void stop() {
        std::cout << "Engine stopped!" << std::endl;
    }
};

// Клас "Автомобіль"
class Car {
private:
      Engine engine;   // Об'єкт класу "Двигун" як член класу
"Автомобіль"

public:
    void startCar() {
        engine.start();
```

```
    }

    void stopCar() {
        engine.stop();
    }
};

int main() {
    Car myCar;
    myCar.startCar();
    myCar.stopCar();

    return 0;
}
```

У цьому прикладі ми маємо клас "Двигун", який має методи start() і stop() для запуску та зупинки двигуна відповідно. Клас "Автомобіль" агрегує об'єкт класу "Двигун" як свій член.

У головній функції ми створюємо об'єкт "Автомобіль" з назвою myCar. Викликаючи метод startCar(), ми викликаємо метод start() об'єкта engine, який запускає двигун. Потім, викликаючи метод stopCar(), ми викликаємо метод stop() об'єкта engine, який зупиняє двигун.

Композиція - це тип відношення між класами, коли один клас (відомий як контейнер або композит) містить об'єкти іншого класу (відомі як компоненти). У випадку композиції, компоненти не можуть існувати без контейнера, тобто вони залежать від його існування і життєвого циклу.

Основна відмінність між агрегацією і композицією полягає в тому, що в композиції компоненти є прямими членами контейнера і створюються та знищуються разом з ним. Це означає, що коли контейнер створюється або знищується, його компоненти також автоматично створюються або знищуються. Крім того, контейнер відповідає за створення та управління життєвим циклом своїх компонентів.

Давайте розглянемо приклад знову. Припустимо, що у нас є клас "Автомобіль" і клас "Двигун". В цьому випадку, композиційне відношення між ними означає, що клас "Автомобіль" містить об'єкт "Двигун" як свій прямий член. Таким чином, коли створюється об'єкт "Автомобіль", він створює також

об'єкт "Двигун". Якщо "Автомобіль" більше не потрібен або видаляється, то разом з ним також буде знищено об'єкт "Двигун".

Композиція дозволяє створювати складніші об'єкти, засновані на ієрархії класів, де один клас повністю залежить від іншого. Вона допомагає керувати взаємозв'язком між компонентами та забезпечує їх єдність в контексті контейнера.

Наприклад:

```
#include <iostream>

// Клас "Серце"
class Heart {
public:
    void beat() {
        std::cout << "Heart is beating!" << std::endl;
    }
};

// Клас "Людина"
class Person {
private:
     Heart heart;  // Об'єкт класу "Серце" як складова частина класу "Людина"

public:
    void doSomething() {
        heart.beat();
    }
};

int main() {
    Person person;
    person.doSomething();

    return 0;
}
```

У цьому прикладі ми маємо клас "Серце", який має метод beat() для симуляції роботи серця. Клас "Людина" композиційно містить об'єкт класу "Серце" як свою складову частину.

У головній функції ми створюємо об'єкт "Людина" з назвою person. Викликаючи метод doSomething(), ми викликаємо метод beat() об'єкта heart, що відображає роботу серця людини.

Розглянемо більш детальний приклад композиції. Для реалізації наступного проєкту нам потрібно буде середовище розробки *Qt Creator.* Ви

можете встановити його, завантаживши з офіційного сайту https://www.qt.io/download-dev. Після завантаження інсталюйте середовище обов’язково включивши наступні компоненти, які важливі для запуску та використання коду прикладу. Необхідні компоненти:

- Qt 6.9 (не потрібно обирати увесь компонент. Достатньо обрати компілятор, який позначено як профіль Desktop. Додатково необхідно встановити додаткові бібліотеки (Additional Libraries), у які входять:
  - Qt 5 Compatibility module
  - Qt Positioning
  - Qt Shader Tools
  - Qt Web Channel
  - Qt Web Sockets
  - Qt WebView
- Extensions (Особливо нас цікавить QtWebEngine, але тільки для версії Qt 6.9.0. Не треба встановлювати абсолютно усі компоненти, адже вони будуть займати дуже багато внутрішньої пам’яті.)
  - Qt 6.9.0

Якщо ви все встановили правильно, то у вас приблизно має так виглядати утиліта Maintenance Tool Qt:

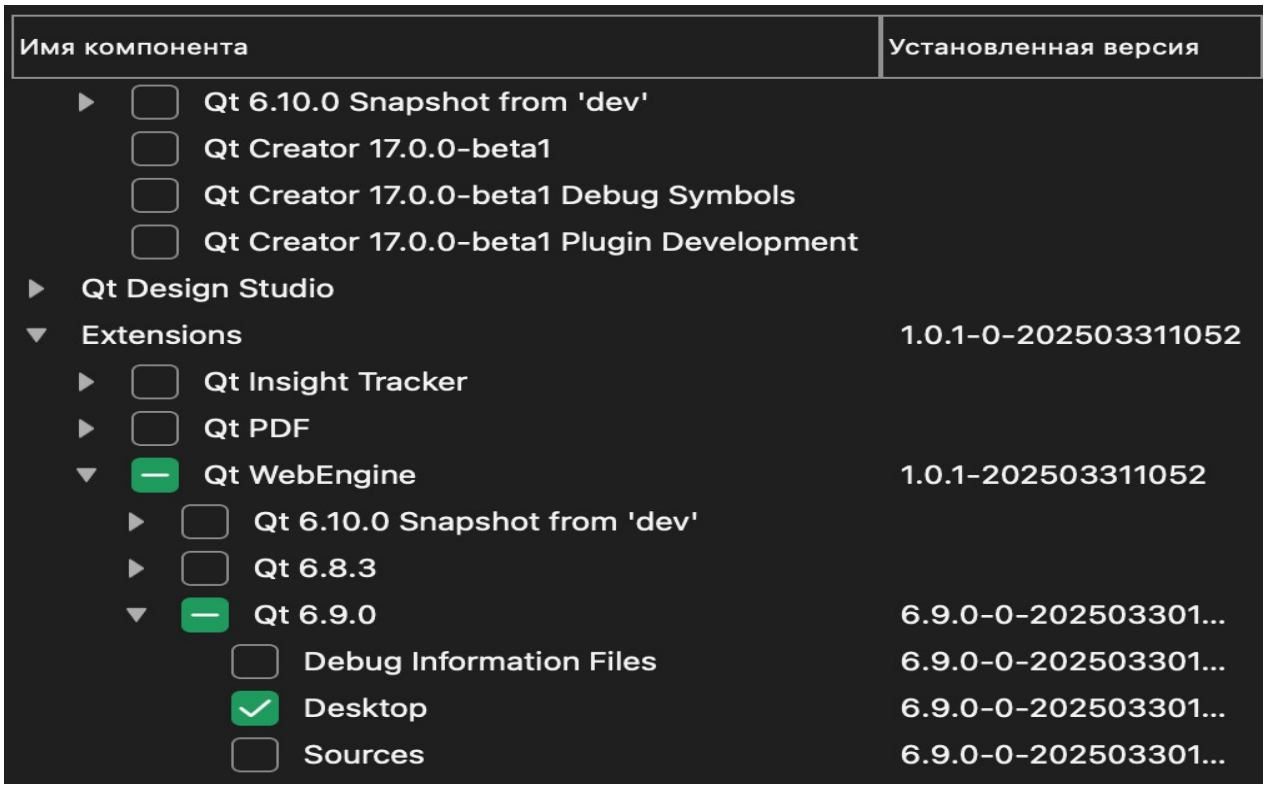

Рис 2.1 Інсталяція основних компонентів QtWebEngine

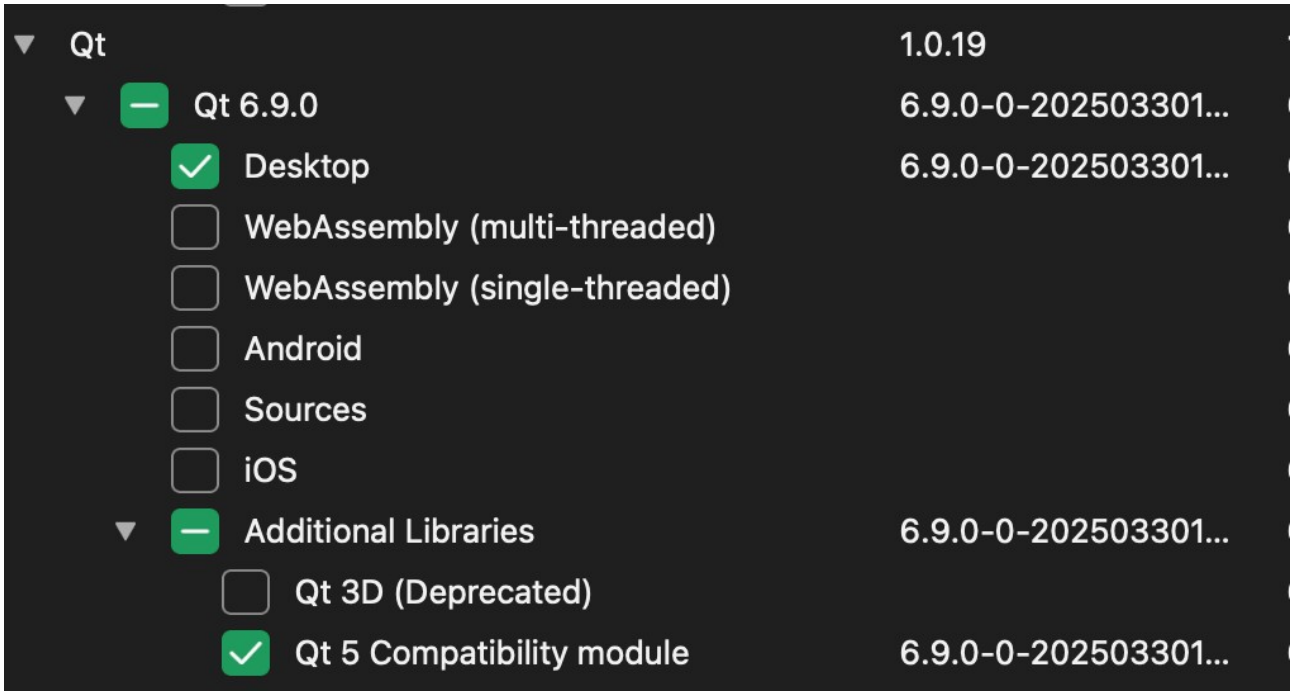

Рис 2.2 Інсталяція основних компонентів Qt 6.9

Після того, як ви встановите собі інтегроване середовище розробки Qt, ви зможете завантажити приклад коду для подальшої роботи з ним. Для завантаження прикладу, вам необхідно перейти за посиланням: https://github.com/rtkachuk/NG_2025_Roman_Tkachuk/tree/master,та завантажити приклад коду:

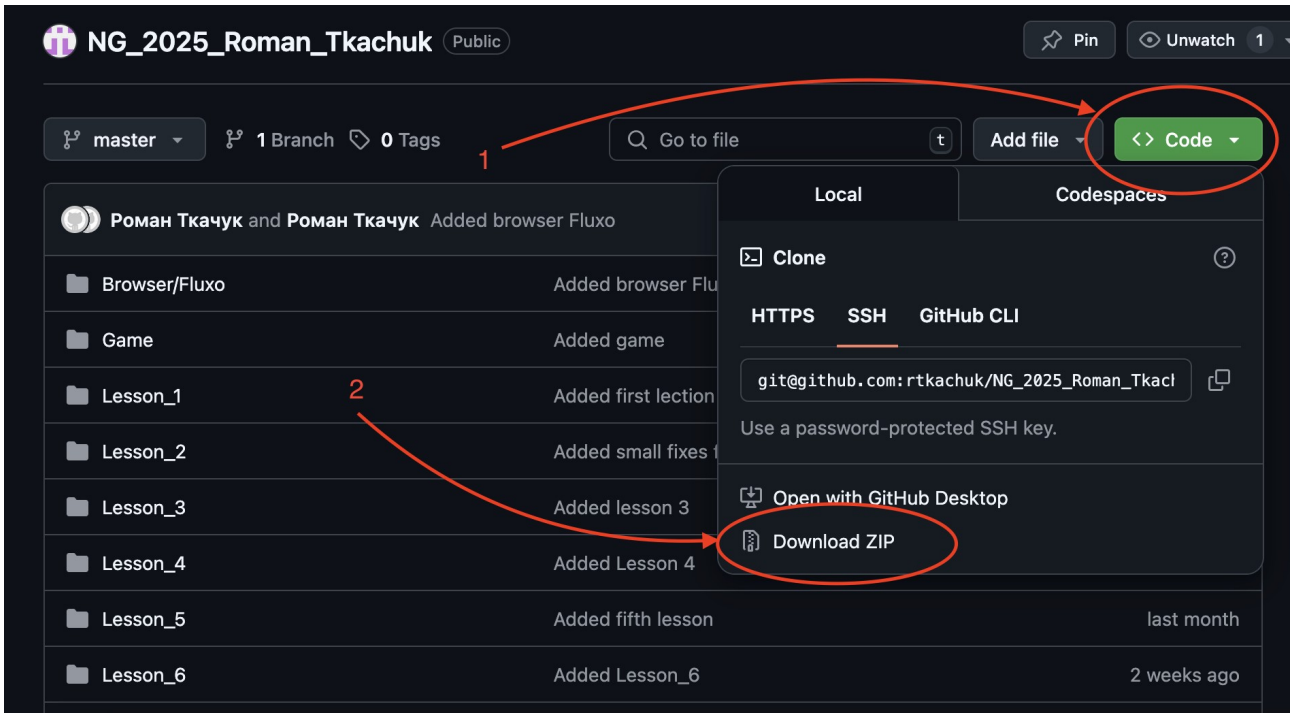

Рис 2.3 Завантаження архіву з прикладом

Після того, як ви завантажите та розпакуєте запропонований файл, ви зможете відкрити програму Qt Creator (яка буде знаходитись на вашій стільниці) та завантажити проект.

Для більш детального розуміння, на Рис 2.3. Перший крок потребує натиснути клавішу «Code» для завантаження основного архіву з програмою на кроці 2. Рис. 2.4. Складається з трьох кроків. 1 крок потребує натиснути клавішу «Open Project» для відкриття віконця «Відкрити файл». Другий крок потребує використати шлях в середині розпакованого архіву (Browser/Fluxo/Fluxo.pro) для відкриття проєкту.

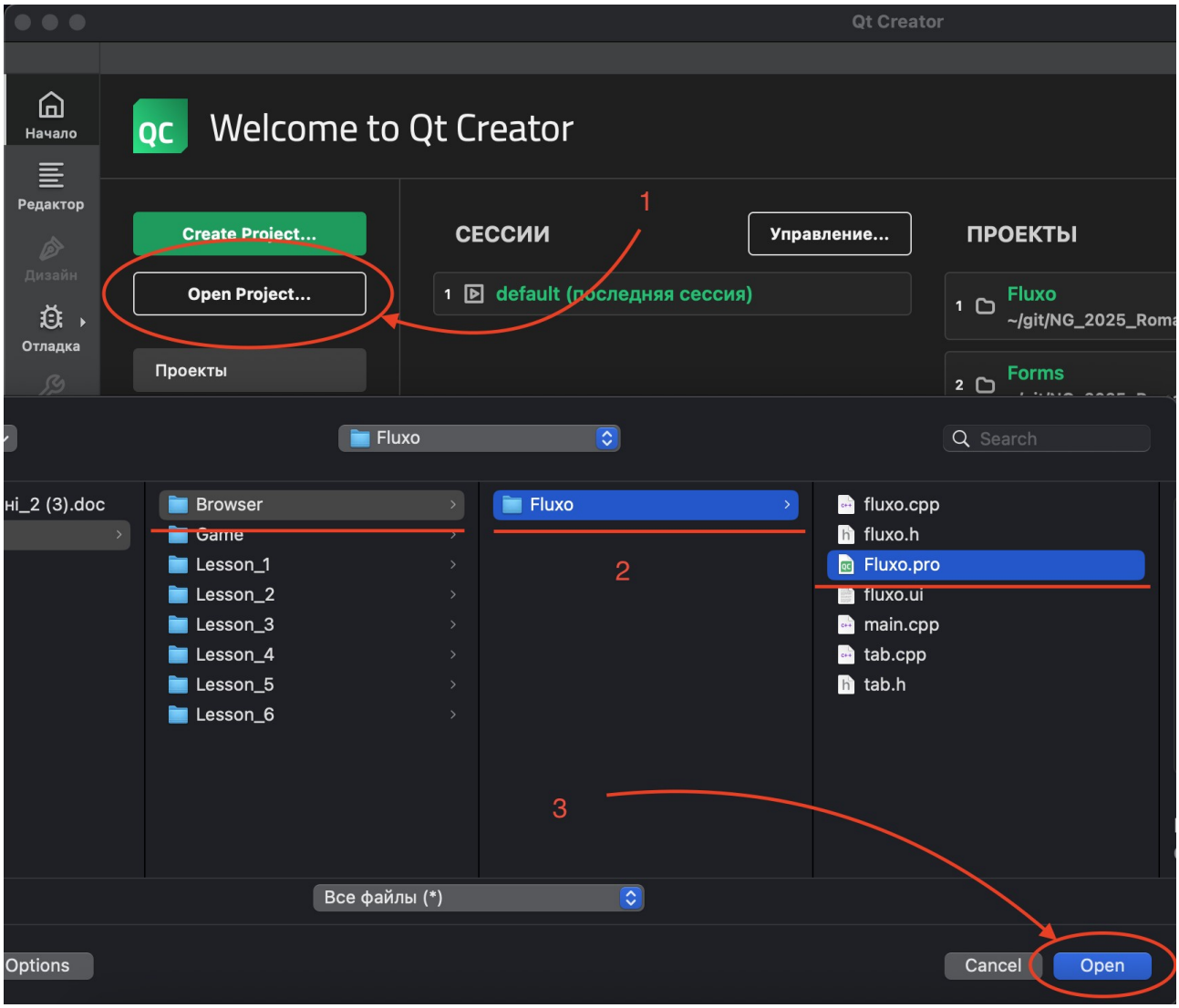

Рис. 2.4

Тепер ви отримали можливість вивчити та відкрити для себе приклад робочого браузеру, який використовує основні компоненти Qt та презентує композицію із двох класів. Перед тим, як почати розбирати логіку роботи програми, необхідно ознайомитись з основним визначенням, яке репрезентує готовий код.

> Композиція - це коли один об'єкт містить інший і керує його життєвим циклом. Якщо головний об'єкт зникає - зникає й той, що всередині.

Отже уявімо, що ми створюємо браузер. У браузері можуть бути вкладки. Без браузера вкладки не мають сенсу - це частина його інтерфейсу.

В нашому випадку, браузер вміщує в себе два класи. Клас браузера (fluxo.h, fluxo.cpp) та клас вкладки (tab.h, tab.cpp).

Клас Fluxo вміщує в себе функції створення вкладки (createTab) та закриття вкладки (closeTab). Суть *композиції* полягає в тому, що саме клас **Fluxo** виконує роль **композитора**. А саме він *нічого* не знає про функціональність, яка закладена в клас Tab. Єдине, чим керує Fluxo – це кількість робочих вкладок.
У коді це виглядає приблизно так:

```
   #ifndef FLUXO_H
   #define FLUXO_H
   //Підключення основних бібліотек Qt, які використовуються у
класі  Fluxo
   #include <QMainWindow>
   #include <QMenu>
   #include <QAction>

   #include "tab.h" // підключення класу компоненту

   QT_BEGIN_NAMESPACE
   namespace Ui {
   class Fluxo;
   }
   QT_END_NAMESPACE

   class Fluxo : public QMainWindow // клас  Fluxo наслідується
від QMainWindow
   {
       Q_OBJECT

   public:
       Fluxo(QWidget *parent = nullptr);//Конструктор
       ~Fluxo();//деструктор

   private slots:
      void createTab();//функція створення вкладки
      void closeTab(int index);//функція закриття вкладки

   private:
        QMap<QWidget *, Tab*> m_tabs; // створюємо мар за допомогою
класу  Qmap,де QWidget *- це ключ, а Tab* - значення
       Ui::Fluxo *ui; // це форма

         QMenu *m_rootMenu; // меню за допомогою якого ви зможете
додавати вклаки
       QAction *m_addTab; // пункт меню «додати вкладку»
   };
   #endif // FLUXO_H
```

`Fluxo` — це назва нашого базового класу(і браузера). Він наслідується від класу `QMainWindow`, що дає нам можливість працювати з графічними формами. *Графічна форма на основі QMainWindow в Qt — це вікно з повноцінною структурою користувацького інтерфейсу, яке створюється за допомогою конструктора форм (Qt Designer) і класу QMainWindow, та використовується як головне вікно додатку.*

Наш браузер не може містити лише одну вкладку, тому для зберігання декількох ми використовуємо ***Map*** *— це асоціативний контейнер стандартної бібліотеки шаблонів (STL), який зберігає пари ключ–значення.* Більш детально з якими ви познайомитесь в наступних лабораторних роботах.

```
#include "fluxo.h"//підключення h-файлу
#include "ui_fluxo.h" // підключення форми

Fluxo::Fluxo(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::Fluxo)//наш конструктор з параметрами
{
    ui->setupUi(this);

    // Menu
    //
    //наступні два рядки створюють меню для додавання
//нової вкладки
    m_rootMenu = new QMenu("File");
    m_addTab = new QAction("Add tab");

    m_rootMenu->addAction(m_addTab);
    ui->menubar->addMenu(m_rootMenu);

       // Перед запуском браузеру необхідно видалити існуючі
вкладки, аби потім створити «правильну»

     ui->tabWidget->removeTab(0);
     ui->tabWidget->setTabsClosable(true);

  // Тепер потрібно створити першу вкладку, яка створить самий
перший об'єкт екземпляру Tab, та відобразить на екрані усі
необхідні візуальні компоненти
  createTab

// connect - використовується для відслідковування певних
«сигналів» із форми. У Qt більшість об'єктів в коді мають свої
сигнали, які можуть описувати певну поведінку. Завдання функції
connect полягає у прив'язуванні «сигналів» до функцій у коді
```

```
connect(m_addTab,            &QAction::triggered,            this,
&Fluxo::createTab); // прив’язуємо пункт меню  m_addTab до функції
createTab
connect   (ui->tabWidget,   &QTabWidget::tabCloseRequested,   this,
&Fluxo::closeTab); // ініціалізуємо закриття вкладки.
   }

   Fluxo::~Fluxo() //деструктор
   {
       delete ui;
   }
   //функція створення вкладки
   void Fluxo::createTab()
   {
   // Для початку, нам потрібно створити нову вкладку на формі,
яка буде пустою та без будь-яких віджетів.
          int  index  =  ui->tabWidget->addTab(new  QWidget(),  "New
Tab");
   // Далі ми маємо переключитись на цю нову створену вкладку
       ui->tabWidget->setCurrentIndex(index);
   // Наступні три рядки створюють об’єкт класу Tab, який згенерує
усі необхідні візуальні компоненти для вкладки, та за допомогою
функцій setLayout (додати на форму)та getTabView (отримані усі
створені графічні компоненти). Особливу увагу потрібно приділити
використанню Map – саме тут ми пояснюємо браузеру, до якої
візуальної вкладки який об’єкт класу Tab відноситься.
       QWidget *tab = ui->tabWidget->currentWidget();
       m_tabs[tab] = new Tab();
       tab->setLayout(m_tabs[tab]->getTabView());
   }
   //функція закриття вкладки
   void Fluxo::closeTab(int index)
   {
       QWidget *tab = ui->tabWidget->widget(index);
       Tab *bufferTab = m_tabs[tab];
       m_tabs.remove(tab);
       ui->tabWidget->removeTab(index);
       delete bufferTab;

     // Якщо це була остання вкладка – тут же створюємо нову пусту
вкладку.
       if (ui->tabWidget->count() == 0)
           createTab();
   }
```

Кожна окрема вкладка, натомість нічого не знає про клас Fluxo, проте виконує основну роль робочого компонента – надає користувачу змогу продивлятися веб-сторінки та відкривати їх за певною адресою. Для цього клас Tab має декілька функцій:

- updateUrl – під час завантаження веб-сторінки, «браузер» буде переходити за різними посиланнями. Ця функція потрібна для того, щоб візуально оновлювати адресу, яка наразі відображена у вкладці.
- goToWebsite – функція, яка бере адресу, яку ввів користувач, та відкриває її в компоненті WebEngineView, який відповідає за відображення веб-сторінки.
- updateProgressBar – функція, що відповідає за відображення прогресу завантаження веб-сторінки.

```
#ifndef TAB_H
#define TAB_H

// Підключення основних бібліотек Qt, які використовуються у класі Tab
#include <QObject>
#include <QPushButton>
#include <QLineEdit>
#include <QLayout>
#include <QWebEngineView>
#include <QProgressBar>

class Tab : public QObject
{
    Q_OBJECT
public:
    explicit Tab(QObject *parent = nullptr);
    ~Tab();

    QLayout* getTabView() { return tabLayout; }

public slots:
    void updateUrl(QUrl url);
    void goToWebsite();
    void updateProgressBar(int value);

private:
    QPushButton *b_go;
    QLineEdit *le_url;
    QWebEngineView *view;
    QLayout *addressLayout;
    QLayout *tabLayout;
    QProgressBar *progressBar;

signals:
};

#endif // TAB_H
```

і розглянемо реалізацію цього класу `Tab.cpp`

```
#include "tab.h"

Tab::Tab(QObject *parent)
    : QObject{parent}
{
    b_go = new QPushButton();
    b_go->setText("GO");
    le_url = new QLineEdit();

    view = new QWebEngineView();
    progressBar = new QProgressBar();
    progressBar->setVisible(false);

    addressLayout = new QHBoxLayout();
    addressLayout->addWidget(le_url);
    addressLayout->addWidget(b_go);

    tabLayout = new QVBoxLayout();
    tabLayout->addItem(addressLayout);
    tabLayout->addWidget(view);
    tabLayout->addWidget(progressBar);

                  connect(b_go,    &QPushButton::clicked,    this,
&Tab::goToWebsite);
               connect(view,   &QWebEngineView::urlChanged,   this,
&Tab::updateUrl);
             connect(view,   &QWebEngineView::loadProgress,   this,
&Tab::updateProgressBar);
}

Tab::~Tab()
{
                disconnect(b_go,    &QPushButton::clicked,    this,
&Tab::goToWebsite);
            disconnect(view,   &QWebEngineView::urlChanged,   this,
&Tab::updateUrl);
          disconnect(view,   &QWebEngineView::loadProgress,   this,
&Tab::updateProgressBar);

    tabLayout->removeWidget(view);
    tabLayout->removeWidget(progressBar);
    tabLayout->removeItem(addressLayout);

    addressLayout->removeWidget(le_url);
    addressLayout->removeWidget(b_go);

    delete tabLayout;
    delete addressLayout;
    delete progressBar;
    delete view;
    delete le_url;
```

```
        delete b_go;
    }

    void Tab::updateUrl(QUrl url)
    {
        le_url->setText(url.toString());
    }

    void Tab::goToWebsite()
    {
        QString url = le_url->text();

        if (url.indexOf("https://") != 0)
            url = "https://" + url;
           // Саме ця частина коду є найважливішою, адже за
допомогою неї на візуальному компоненті кожної вкладки
завантажується веб-сторінка.
        view->load(url);
    }

    void Tab::updateProgressBar(int value)
    {
        if (value == 100)
            progressBar->setVisible(false);
        else
            progressBar->setVisible(true);

        progressBar->setValue(value);
    }
```

Життєвий цикл

- Об'єкти типу Tab створюються усередині браузера (Fluxo) - наприклад, коли користувач відкриває нову вкладку.
- Коли браузер (Fluxo) закривається або знищується - усі вкладки також зникають. Їх не потрібно зберігати окремо чи вручну видаляти - Qt знищить їх автоматично (якщо parent правильно заданий).

Вкладка (`Tab`) є невід'ємною частиною браузера (`Fluxo`) і не має сенсу існувати самостійно: вона не може функціонувати або відображатися окремо, оскільки створюється, управляється та знищується разом із браузером. Це елемент інтерфейсу, який логічно та технічно залежить від свого "власника" - без браузера вкладка не існує, що чітко вказує на композиційний зв'язок між цими класами.

Крім цього, ви, звичайно, можете створити свій проєкт на базі наданого вам прикладу. Для цього відкрийте **Qt Creator** і дотримуйтесь наступних кроків:

Створення нового проєкту з використанням QMainWindow:

1. У головному меню Qt Creator виберіть:
   File → New Project → Application → Qt Widgets Application.
   Натисніть **"Choose..."**.
2. Введіть ім’я проєкту (наприклад, MyBrowserProject) та оберіть теку для збереження.
3. Оберіть відповідний **Build Kit** (наприклад, Desktop Qt 5/6) і натисніть **Next**.
4. У розділі **Class Information**:
   - **Base class:** оберіть QMainWindow
   - **Class name:** введіть MainWindow
   - Переконайтесь, що встановлені прапорці для створення .h, .cpp і .ui файлів.
5. Натисніть **Finish** — Qt автоматично згенерує скелет проєкту:
   - `mainwindow.h` — оголошення класу
   - `mainwindow.cpp` — реалізація
   - `mainwindow.ui` — графічна форма
   - `main.cpp` — запуск програми
6. Відкрийте файл mainwindow.ui — подвійним кліком у провіднику проєкту — для редагування інтерфейсу у **Qt Designer**:
   - Додайте необхідні віджети (наприклад, кнопку "Назад").
   - За потреби створіть панель інструментів через **"Promote to"** або використайте стандартну QToolBar.
7. Поверніться до mainwindow.cpp і реалізуйте логіку взаємодії

Завдання 1

Ознайомтесь з теоретичним матеріалом щодо створення графічних інтерфейсів у Qt та використання класу `QMainWindow`.

- Скачайте шаблон проєкту браузера Fluxo з GitHub за посиланням, наданим у методичних матеріалах, або скористайтесь прикладом із теоретичної частини.
- Відкрийте проєкт у Qt Creator.
- Проаналізуйте структуру проєкту: головне вікно, реалізація вкладок, використання `QWebEngineView` (або `QWebView`, залежно від версії).
- Додайте в інтерфейс кнопку "Назад" (повернення на попередню сторінку). Для цього:
- Розмістіть кнопку у зручному місці інтерфейсу (наприклад, на панелі інструментів або над вкладками).
- Зв’яжіть кнопку з відповідним слотом, який викликає метод `back()` для поточного віджета веб-перегляду.
- Перевірте, що кнопка коректно працює для активної вкладки: при натисканні сторінка повертається до попередньої.

Завдання 2

Варіант 1

1. Створить систему класів для моделювання інформації про студентів та університет. Основним є клас **"Людина"**, який містить властивості ім’я, вік, а також створить клас "Адреса", який представляє адресу проживання людини (використовуйте композицію). Клас **"Студент"** має бути класом, який має властивості номер студентського квитка та курс. Усі студенти повинні бути пов’язані з певним університетом через клас **"Університет"**, який зберігає список студентів. Зв’язок між університетом і студентами реалізується як **агрегація**. Також до завдання включено **асоціацію**: наприклад, кожен студент може бути пов’язаний з викладачем, але викладач не є частиною студента (і навпаки), тобто це окремі об’єкти, які взаємодіють.

У програмі слід реалізувати методи для додавання і видалення студентів з університету, а також виведення їхньої інформації.

У функції main(), у якій створюються об’єкти класів, встановлюються зв’язки між ними та викликаються відповідні методи.

Варіант 2

Створіть систему класів для моделювання інформації про бібліотеку. Основним є клас **"Бібліотека"**, який містить властивості назва та адреса, а також список книг, що зберігаються в бібліотеці. Зв'язок між бібліотекою та книгами реалізується як **агрегація**, оскільки книги існують незалежно від бібліотеки. Клас **"Бібліотекар"**, який представляє працівника бібліотеки, входить до складу класу "Бібліотека" через **композицію**. Створіть клас **"Книга"**, який містить властивості: назва книги, автор книги та рік видання. Додайте також **асоціацію**: наприклад, кожна книга може бути пов'язана з конкретним читачем (окремий клас "Читач"), який її читає або бере в користування.

Реалізуйте методи для додавання та видалення книг у бібліотеці, а також виведення інформації про книги.

У функції main() створіть об'єкти класів, встановіть між ними зв'язки та продемонструйте взаємодію. За бажанням, додайте додаткові методи та властивості для розширення функціональності.

Варіант 3

Створіть систему класів для моделювання інформації про компанію. Основним є клас **"Компанія"**, який містить властивості назва та адреса, а також список підрозділів, які входять до її складу. Зв'язок між компанією та підрозділами реалізується як **агрегація**, оскільки підрозділи можуть існувати незалежно від компанії. Клас також містить об'єкт **"Головний директор"**, що реалізується через **композицію**, оскільки директор є невід'ємною частиною компанії.

Створіть клас **"Підрозділ"**, який містить назву, кількість працівників і керівника підрозділу. Для моделювання працівників використовуйте клас **"Працівник"**, який включає прізвище, ім'я, посаду та заробітну плату. Додайте також **асоціацію** — наприклад, підрозділ може бути пов'язаний із певним зовнішнім консультантом (окремий клас "Консультант"), який не є частиною підроз-

ділу, але співпрацює з ним. Реалізуйте методи для додавання та видалення підрозділів у компанії, а у функції main() створіть об’єкти всіх відповідних класів, встановіть між ними зв’язки та виведіть інформацію про компанію та її структуру. За бажанням, додайте нові методи та властивості для розширення функціональності.

Варіант 4

Створіть систему класів для моделювання інформації про музичний гурт. Основним є клас **"Музичний гурт"**, який містить властивості назва та стиль музики. Гурт також має список музикантів, які в нього входять — цей зв’язок реалізується як **агрегація**, оскільки музиканти можуть існувати незалежно від гурту. Клас також містить об’єкт **керівника гурту**, що реалізується через **композицію**, оскільки керівник є невід’ємною частиною гурту.

Створіть клас **"Музикант"**, який містить ім’я музиканта та інструмент, на якому він грає. Інструмент представлений окремим класом **"Інструмент"** з властивостями назва інструменту та рік виготовлення. Для реалізації **асоціації** додайте до завдання ще одну сутність — наприклад, кожен музикант може бути пов’язаний із менеджером (окремий клас "Менеджер"), який організовує виступи гурту, але не є його частиною. Реалізуйте методи для додавання та видалення музикантів із гурту, а у функції main() створіть об’єкти класів, встановіть між ними відповідні зв’язки та виведіть інформацію. За бажанням, розширте функціональність класів новими методами чи властивостями.

Варіант 5

Створіть систему класів для моделювання об’єкта **"Телефон"**, який має властивості марка та модель. До складу телефону входять об’єкти класів **"Екран"** та **"Камера"** — ці зв’язки реалізуються як **композиція**, оскільки екран і камера є невід’ємними частинами телефону та не можуть існувати окремо від нього. Клас "Екран" містить інформацію про розмір та тип екрану, а клас "Камера" — про розмір матриці та роздільну здатність.

Крім того, реалізуйте **асоціацію**: наприклад, кожен телефон може бути пов’язаний з постачальником (клас "Постачальник"), який постачає запчастини або займається обслуговуванням, але не є складовою частиною телефону. Створіть методи для встановлення та отримання інформації про телефон, екран та камеру. У функції main() створіть відповідні об’єкти, встановіть між ними зв’язки та виведіть інформацію. За бажанням розширте програму додатковими методами або властивостями.

Варіант 6

Створіть систему класів для моделювання структури **університету**. Основним є клас **"Університет"**, який має властивості назва та адреса, а також містить список факультетів (використовується **агрегація**, оскільки факультети можуть існувати незалежно). Клас **"Факультет"** містить назву та об'єкт класу **"Декан"**, який є невід’ємною частиною факультету (використовується **композиція**), а також список студентів, які навчаються на факультеті (використовується **агрегація**).

Крім цього, реалізуйте **асоціацію**: наприклад, кожен студент може бути пов’язаний із науковим керівником (об’єкт окремого класу), але керівник не є частиною студента — це окремі об'єкти, які співпрацюють. Реалізуйте методи для додавання та видалення студентів із факультету, а також виведення інформації про університет, факультети та студентів. У функції main() створіть відповідні об’єкти, встановіть між ними зв’язки та викличте відповідні методи. За бажанням розширте програму додатковими методами або властивостями.

Варіант 8

Створіть систему класів для моделювання інформації про магазин. Основним є клас **"Магазин"**, який містить властивості: назва, адреса, а також список товарів, які є у магазині (використовуйте **агрегацію**, оскільки товари можуть існувати незалежно від магазину). Створіть клас **"Товар"**, який містить такі властивості: назва товару та його ціна. Додайте клас **"Кошик"**, який представляє корзину покупця та містить список товарів (використовуйте **композицію**, оскільки товари в кошику є частиною самого кошика) і загальну вартість

товарів у кошику. Реалізуйте методи для додавання та видалення товарів з кошика, а також для розрахунку загальної вартості.

Продемонструйте **асоціацію**, створивши клас **"Покупець"**, який пов’язаний із кошиком – покупець може взаємодіяти з кошиком, але не володіє ним як частиною себе (і навпаки, кошик не є частиною покупця). У функції main() створіть об’єкти класів **"Магазин"**, **"Товар"**, **"Кошик"** та, за бажанням, **"Покупець"**. Додайте товари до магазину, додайте вибрані товари до кошика, обчисліть загальну вартість товарів та виведіть відповідну інформацію. За потреби розширте функціональність, додаючи додаткові методи та властивості до класів.

Варіант 9

Створіть систему класів для моделювання автомобіля. Основним є клас **"Автомобіль"**, який містить такі властивості: марка, модель, двигун і салон. Клас **"Двигун"** містить тип двигуна (бензиновий, дизельний тощо) та його об’єм. Клас **"Салон"** містить кількість місць і тип салону (тканинний, шкіряний тощо). Зв’язки **композиції** реалізуються між класом "Автомобіль" та об’єктами класів "Двигун" і "Салон", оскільки ці компоненти є невід'ємною частиною автомобіля. Реалізуйте методи встановлення та отримання значень властивостей автомобіля, двигуна й салону.

Продемонструйте **асоціацію**, створивши клас "Автомобіль", який має зв’язки з іншими класами, такими як "Двигун" (двигун), "Салон" (інтер’єр) та "СпортивнийАвтомобіль" (як спеціалізована конфігурація, що може бути пов’язана з об’єктом типу "Автомобіль"). Клас "СпортивнийАвтомобіль" повинен містити додаткові властивості, наприклад, максимальна швидкість або тип приводу, а також методи, пов’язані з динамікою чи керуванням. У функції `main()` створіть об’єкти класів "Автомобіль", "Двигун", "Салон", (за бажанням – також "СпортивнийАвтомобіль"), встановіть відповідні зв’язки між ними, задайте значення їхнім властивостям і виведіть відповідну інформацію. За потреби розширте функціональність, додаючи нові методи, наприклад, порівняння автомобілів за об’ємом двигуна або виведення повної конфігурації авто.

Варіант 10

Створіть систему класів для моделювання комп’ютерної гри. Основним є клас **"Комп'ютерна гра"**, що містить назву, розробника гри та список персонажів — цей список реалізується через **агрегацію**, оскільки персонажі можуть існувати незалежно від гри. Клас **"Персонаж"** містить ім’я, рівень та об’єкт класу **"Характеристики"** (що включає, наприклад, здоров’я, силу, інтелект), реалізований через **композицію**, оскільки характеристики є складовою персонажа. Для додавання й видалення персонажів реалізуйте відповідні методи у класі "Комп'ютерна гра".

Продемонструйте **асоціацію**, створивши клас "Гравець", який має зв’язок із класом "Персонаж". Клас "Гравець" розширюється додатковими властивостями, такими як рівень гравця та кількість очок. Також можна створити додаткові класи, наприклад, "Маг" чи "Воїн", які мають власні поля та методи (наприклад, магічна сила або тип зброї) і також асоціюються з об’єктами типу "Персонаж". У функції `main()` створіть об’єкти класів "Комп’ютерна гра", "Персонаж", "Гравець", а також за бажанням — "Маг" чи "Воїн", задайте їхні властивості, встановіть зв’язки між ними та виведіть відповідну інформацію. За потреби додайте додаткову функціональність, наприклад, метод бою чи підвищення рівня персонажа.

Лабораторна робота №3 Наслідування. Специфікатори доступу

Наслідування (іноді його називають також наслідуванням) в об'єктно-орієнтованому програмуванні є одним із фундаментальних концепцій. Воно дозволяє створювати нові класи на основі вже існуючих (батьківських або базових класів), утворюючи ієрархію класів.

Наслідування дозволяє поширювати властивості і поведінку базового класу на похідні класи. Похідні класи отримують доступ до публічних і захищених членів базового класу (таких як поля, методи, властивості) і можуть додавати свої власні члени або перевизначати члени базового класу.

Один з основних принципів успадкування - це утворення спеціалізованих класів на базі загального базового класу. Наприклад, уявімо ієрархію класів "Транспортний засіб" як базовий клас, а "Автомобіль", "Велосипед" та "Мотоцикл" як його похідні класи. У цьому випадку "Транспортний засіб" міститиме загальну функціональність, а похідні класи розширять його функціональність, додаючи власні особливості.

Однією з головних переваг успадкування є полегшення повторного використання коду. Базовий клас може містити загальні функції і властивості, які можуть бути успадковані і використовані у всіх похідних класах. Це забезпечує ефективніше управління кодом і зменшення дублювання.

Наслідування також сприяє поліморфізму, що дозволяє використовувати об'єкти похідних класів як об'єкти базового класу. Це розширює можливості поліморфного програмування і спрощує роботу зі змінними і колекціями об'єктів.

Наприклад, у мові C++ успадкування виконується за допомогою ключового слова class або struct, за яким слідує ім'я похідного класу і базового класу, відокремлені двокрапкою. Наприклад:

```
class BaseClass {
    // Тіло базового класу
};

class DerivedClass : public BaseClass {
    // Тіло похідного класу
};
```

У цьому прикладі клас DerivedClass успадковує властивості і методи класу BaseClass.

Наслідування є важливим інструментом в об'єктно-орієнтованому програмуванні, оскільки дозволяє створювати ієрархії класів з поліморфними можливостями та забезпечує повторне використання коду. Воно сприяє модульності, розширюваності і зрозумілості програмного коду.

Інкапсуляція та режими доступу

Однією з основних концепцій об'єктно-орієнтованого програмування є **інкапсуляція**. Вона означає об'єднання даних (змінних) і методів (функцій), які з ними працюють, в одну сутність — **клас**, а також **обмеження доступу** до внутрішніх деталей реалізації цього класу. Інкапсуляція забезпечує захист даних від неконтрольованої зміни та сприяє кращій структурованості та модульності коду.

Щоб реалізувати інкапсуляцію, у C++ використовуються **режими доступу (специфікатори доступу)**. Вони визначають, які частини програми можуть отримати доступ до членів класу (змінних і методів).

Існує три основні режими доступу:

- **public** — публічний доступ: члени класу доступні звідусіль (зовнішній код, інші класи).
- **protected** — захищений доступ: члени доступні лише всередині самого класу та його похідних (спадкоємців).
- **private** — приватний доступ: члени доступні тільки всередині класу, де вони оголошені.

Ці режими використовуються як для оголошення окремих членів класу, так і для визначення типу наслідування.

У C++ при успадкуванні класів необхідно вказати режим доступу, який визначає, як нестатичні public і protected члени базового класу будуть доступні у спадкоємному (похідному) класі. Від цього залежить, які елементи базового

класу можна буде використовувати у спадкоємних класах і в зовнішньому коді. Розрізняють три основні режими:

1. Публічне успадкування (**public**)

Відкрите успадкування є одним із найбільш використовуваних типів успадкування. Дуже рідко ви побачите або будете використовувати інші типи, тому основну увагу слід приділити саме розумінню цього типу. На щастя, відкрите успадкування є найпростішим і найлегшим з усіх типів. Коли ви відкрито успадковуєте батьківський клас, то успадковані public-члени залишаються public, успадковані protected-члени залишаються protected, а private-члени залишаються недоступними для дочірнього класу. Тобто — нічого не змінюється.

Специфікатор доступу в батьківському класі	**При відкритому успадкуванні в дочірньому класі**
public	public
private	недоступний
protected	protected

```
class Parent
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Pub: public Parent // відкрите успадкування
{
    // Відкрите успадкування означає, що:
    // - public-члени залишаються public у дочірньому класі;
     // - protected-члени залишаються protected у дочірньому класі;
      // - private-члени залишаються недоступними у дочірньому класі.
public:
    Pub()
    {
         m_public = 1;       // дозволено: доступ до m_public відкритий
        m_private = 2;     // заборонено: доступ до m_private із дочірнього класу закритий
              m_protected = 3;     // дозволено: доступ до m_protected відкритий у дочірньому класі
    }
};
```

```
int main()
{
    Parent parent;
    parent.m_public = 1;      // дозволено: m_public доступний ззовні через батьківський клас
    parent.m_private = 2;     // заборонено: m_private недоступний ззовні через батьківський клас
    parent.m_protected = 3;   // заборонено: m_protected недоступний ззовні через батьківський клас

    Pub pub;
    pub.m_public = 1;         // дозволено: m_public доступний ззовні через дочірній клас
    pub.m_private = 2;        // заборонено: m_private недоступний ззовні через дочірній клас
    pub.m_protected = 3;      // заборонено: m_protected недоступний ззовні через дочірній клас
}
```

2. Захищене успадкування (**protected**)

Цей тип успадкування майже ніколи не використовується, крім особливих випадків. При захищеному наслідуванні `public`- та `protected`-члени стають `protected`, а `private`-члени залишаються недоступними.

Оскільки цей тип успадкування дуже рідко застосовується, ми пропустимо практичний приклад і одразу перейдемо до таблиці:

Специфікатор доступу в батьківському класі	**При захищеному успадкуванні в дочірньому класі**
`public`	`protected`
`private`	недоступний
`protected`	`protected`

Приклад

```
class Parent
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};
```

Клас `Parent` може безперешкодно звертатись до своїх членів. Доступ до `m_public` відкритий для всіх. Дочірні класи можуть звертатись як до `m_public`, так і до `m_protected`.

```
class D2 : private Parent  // приватне успадкування
{
    // Приватне успадкування означає, що:
    // - public-члени стають private у дочірньому класі;
    // - protected-члени стають private у дочірньому класі;
    // - private-члени недоступні для дочірнього класу.
public:
    int m_public2;
private:
    int m_private2;
protected:
    int m_protected2;
};
```

Клас D2 може безперешкодно звертатись до своїх членів. D2 має доступ до членів m_public і m_protected класу Parent, але не до m_private. Оскільки D2 успадковує клас Parent приватно, то m_public і m_protected стають закритими при доступі через D2. Це означає, що інші об’єкти не можуть отримати доступ до цих членів через об’єкт D2, а також будь-які інші класи, які будуть дочірніми класу D2, не матимуть доступу до цих членів:

```
class D3 : public D2
{
    // Відкрите успадкування означає, що:
       //  -  успадковані  public-члени  залишаються  public  у
дочірньому класі;
     // - успадковані protected-члени залишаються protected у
дочірньому класі;
     // - успадковані private-члени залишаються недоступними у
дочірньому класі.
public:
    int m_public3;
private:
    int m_private3;
protected:
    int m_protected3;
};
```

Клас D3 може безперешкодно звертатись до своїх членів. D3 має доступ до членів m_public2 і m_protected2 класу D2, але не до m_private2. Оскільки D3 успадковує D2 відкрито, то m_public2 і m_protected2 зберігають свої специфікатори доступу і залишаються public та protected при доступі через D3. D3 не має доступу до m_private класу Parent. Також він не має доступу до m_protected або m_public класу Parent, які стали закритими, коли D2 їх успадкував.

3. Приватне успадкування (**private**)

При приватному наслідуванні всі члени батьківського класу успадковуються як приватні. Це означає, що `private`-члени залишаються недоступними, а `protected` і `public` члени стають `private` у дочірньому класі.

Зверніть увагу: це не впливає на те, як дочірній клас отримує доступ до членів батьківського класу! Це впливає лише на доступ до цих членів через об’єкти дочірнього класу ззовні:

```
class Parent
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Priv : private Parent // приватне успадкування
{
    // Приватне успадкування означає, що:
     // - public-члени стають private (m_public тепер private) у дочірньому класі;
      // - protected-члени стають private (m_protected тепер private) у дочірньому класі;
       // - private-члени залишаються недоступними (m_private недоступний) у дочірньому класі.
public:
    Priv()
    {
         m_public = 1; // дозволено: m_public тепер private у Priv
         m_private = 2; // заборонено: дочірні класи не мають доступу до private-членів батьківського класу
            m_protected = 3; // дозволено: m_protected тепер private у Priv
    }
};

int main()
{
    Parent parent;
       parent.m_public = 1;// дозволено: m_public доступний ззовні через батьківський клас
     parent.m_private = 2;// заборонено: m_private недоступний ззовні через батьківський клас
         parent.m_protected = 3;// заборонено: m_protected недоступний ззовні через батьківський клас

    Priv priv;
```

```
        priv.m_public = 1;// заборонено: m_public недоступний
ззовні через дочірній клас
        priv.m_private = 2;// заборонено: m_private недоступний
ззовні через дочірній клас
        priv.m_protected = 3;// заборонено: m_protected
недоступний ззовні через дочірній клас
    }
```

Підсумок:

Специфікатор доступу в батьківському класі	**При приватному успадкуванні в дочірньому класі**
`public`	`private`
`private`	недоступний
`protected`	`private`

Приватне наслідування може бути корисним, коли між дочірнім і батьківським класом немає очевидного логічного зв’язку, але дочірній клас використовує функціональність батьківського у своїй реалізації. У такому випадку ми не хочемо, щоб відкритий інтерфейс батьківського класу був доступним через об'єкти дочірнього класу (як це відбувається при відкритому успадкуванні).

На практиці приватне наслідування використовується рідко.

Завдання 1

1. Створіть клас `Base` із трьома полями з різними рівнями доступу:
   - `public` — змінна `publicVar` (тип `int`),
   - `protected` — змінна `protectedVar` (тип `int`),
   - `private` — змінна `privateVar` (тип `int`).
2. Створіть три похідні класи, які успадковують клас `Base` з різними режимами доступу:
   - `PublicDerived` — публічне успадкування,
   - `ProtectedDerived` — захищене успадкування,
   - `PrivateDerived` — приватне успадкування.
3. У кожному похідному класі створіть метод, який спробує змінити значення всіх трьох полів базового класу.

4. У функції `main()` створіть об’єкти всіх трьох похідних класів та спробуйте отримати доступ до полів `publicVar`, `protectedVar` і `privateVar` через ці об’єкти.

5. Проаналізуйте та опишіть, які поля доступні в кожному класі та ззовні, залежно від типу успадкування.

Завдання 2

Використайте свій варіант із лабораторної роботи №2, у якому було реалізовано асоціацію між класами (наприклад, один клас містив поле об'єкта іншого класу), та перепишіть ці класи таким чином, щоб замість асоціації було використано наслідування: створіть базовий клас із загальними властивостями та методами, а також похідні класи, які наслідують базовий і розширюють його додатковими полями та функціональністю. У функції `main()` створіть об’єкти кожного класу, задайте їх властивості, виведіть відповідну інформацію, а також реалізуйте приклади взаємодії між об'єктами, наприклад, оновлення значень або обчислення на основі полів. Програму реалізуйте з використанням роздільної компіляції (.h + .cpp).

Лабораторна робота №4 Поліморфізм. Обробка винятків

Мета: Ознайомитись з поняттям поліморфізму у мові С++ та навчитись використовувати віртуальні функції для досягнення поліморфізму. Також вивчити принципи обробки винятків у мові С++.

Теоретична частина

Поліморфізм - це принцип об'єктно-орієнтованого програмування, який дозволяє об'єктам одного класу використовуватися замість об'єктів інших класів, що мають спільний інтерфейс. Це означає, що об'єкти різних класів можуть бути оброблені і використані за допомогою загального коду без необхідності знати про конкретний клас об'єкту.

Простіше кажучи: *Ми можемо викликати один і той самий метод для різних об’єктів — і кожен об’єкт сам вирішить, як саме виконати цю дію.*

Поліморфізм поділяється на два основні типи:

Тип поліморфізму	Коли визначається	Як реалізується	Приклад
Статичний (компіляторний)	під час компіляції	перевантаження функцій або шаблони	`function overloading`, шаблони
Динамічний (рантайм)	під час виконання	через віртуальні функції	віртуальні методи, `override`

1. Статичний (компіляторний) поліморфізм

Поліморфізм цього типу визначається під час компіляції. Його реалізація не вимагає використання віртуальних функцій.

```
void print(int x) {
    std::cout << "Ціле число: " << x << std::endl;
}

void print(double x) {
    std::cout << "Число з крапкою: " << x << std::endl;
}
```

2. Динамічний (виконуваний) поліморфізм

У C++ поліморфізм можна реалізувати за допомогою віртуальних функцій. **Віртуальна функція** - це функція, оголошена у базовому класі і перевизначена (або заміщена) у похідних класах. Ключове слово "virtual" перед оголошенням функції вказує, що ця функція є віртуальною.

Коли ви викликаєте віртуальну функцію на об'єкті, поведінка функції залежить від типу об'єкта, на якому вона викликається, а не від типу змінної або посилання, яке її містить. Це означає, що коли ви маєте посилання або вказівник на базовий клас, але викликаєте віртуальну функцію, то викликається реалізація цієї функції у відповідному похідному класі.

Оголошення віртуальної функції у базовому класі виглядає так:

```
class Base {
public:
    virtual void virtualFunction() {
        // Реалізація функції у базовому класі
    }
};
```

Основна особливість віртуальних функцій полягає в тому, що при виклику цієї функції на об'єктах через вказівники або посилання базового класу, виконується реалізація цієї функції у відповідному похідному класі, якщо така реалізація існує. Іншими словами, виклик віртуальної функції залежить від фактичного типу об'єкта, на якому вона викликається.

Приклад використання віртуальних функцій у похідних класах:

```
class Derived : public Base {
public:
    void virtualFunction() override {
        // Перевизначена реалізація функції у похідному класі
    }
};
```

У цьому прикладі ми створили похідний клас "Derived", який успадковує від базового класу "Base". У похідному класі ми перевизначили віртуальну функцію "virtualFunction()" зі зміною її реалізації.

Щоб позначити перевизначення функції у похідному класі, використовується ключове слово override. Це покращує зрозуміння коду та дозволяє компілятору перевірити, чи дійсно ми перевизначаємо функцію з базового класу.

У результаті ми можемо працювати з об'єктами похідних класів через вказівники або посилання базового класу і викликати їхні віртуальні функції. При цьому буде виконуватись відповідна реалізація цих функцій у відповідних похідних класах.

Наприклад:

```
Copy code
Base* obj1 = new Derived();
obj1->virtualFunction(); // Виклик функції з похідного класу

Base* obj2 = new Base();
obj2->virtualFunction(); // Виклик функції з базового класу
```

У цьому прикладі ми створюємо об'єкти через вказівники базового класу Base, але викликаємо їхні віртуальні функції. Перший виклик виконує функцію virtualFunction() з похідного класу "Derived", оскільки вказівник obj1 посилається на об'єкт типу "Derived". Другий виклик виконує функцію virtualFunction() з базового класу "Base", оскільки вказівник obj2 посилається на об'єкт типу "Base".

Таким чином, віртуальні функції дозволяють досягати поліморфізму в C++, що дозволяє з легкістю працювати з об'єктами різних класів через їх спільний інтерфейс.

Обробка винятків в мові C++ здійснюється за допомогою механізму винятків (exceptions). Виняток представляє собою спеціальний об'єкт, який викидається під час виникнення помилки або непередбаченої ситуації і може бути перехоплений і оброблений відповідним кодом.

Основні елементи обробки винятків в C++:

1. Викидання винятка: Для викидання винятка використовується оператор throw. Ви можете викинути будь-який об'єкт, який може бути скопійований або кинутий за допомогою оператора throw. Наприклад:

```
throw SomeException(); // Викидання об'єкта винятка
```

2. Перехоплення винятка: Для перехоплення винятка використовується блок try-catch. Ви можете вказати блок коду, в якому виняток може бути перехоплений, і надати відповідний обробник для обробки винятка. Наприклад:

```
try {
    // Блок коду, в якому може виникнути виняток
} catch (const SomeException& e) {
    // Обробник для винятка типу SomeException
} catch (const AnotherException& e) {
    // Обробник для винятка типу AnotherException
} catch (...) {
    // Обробник для всіх інших винятків
}
```

У блоку try вказується код, в якому виняток може виникнути. У блоках catch вказуються обробники для різних типів винятків. Обробники співставляються з викинутими винятками на основі їхніх типів.

3. Розгортання стеку: Якщо виняток викидається в блоку try, система виконання розгортає стек (stack unwinding), що означає виклик деструкторів для об'єктів, які були створені в блоку try, перед тим, як виняток буде перехоплено.

4. Об'єкти винятків: Ви можете визначати власні типи винятків у вигляді класів. Ці класи можуть мати власні поля і методи, які дозволяють додатково ідентифікувати та обробляти винятки.

```
class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Це мій виняток!";
    }
};
```

У цьому прикладі MyException є власним класом винятка, який успадковує від std::exception. Він перевизначає метод what(), який повертає повідомлення про виняток.

5. Обробка винятків за межами функцій: Винятки можуть бути перехоплені не тільки у тому місці, де вони викидаються, але й у вищих рівнях виклику функцій. Якщо виняток не оброблений у поточній функції, він буде переданий наступному обробнику вищого рівня, і так далі, до тих пір, поки виняток не буде перехоплено або програма не завершиться.

Це основні принципи обробки винятків в мові C++. Використання винятків дозволяє контролювати та обробляти помилки у програмі, дозволяючи зручно реагувати на непередбачені ситуації і забезпечувати безпеку виконання коду.

Примітка

Для виконання цього завдання вам потрібно середовище розробки Qt Creator, якщо Ви раніше не працювали з ним то встановіть, та ознайомтесь. Наступним кроком створіть новий проект Qt: Використовуйте Qt Creator для створення нового проекту типу "Qt Widgets Application". При створені проекту ви отримаєте всі файли, які потрібні і навіть більше, отже вітаю, з цим «більше» вам і прийдеться працювати, а якщо точніше то до цього завдання вам треба зробити інтерфейс. Тему для завдання можна обрати на вибір

1. Напишіть гру «Піймай муху», на формі знаходиться муха, яка має тікати від курсора, також на формі має знаходитися пастка, коли муха попадеться в пастку гра закінчується. В реалізації програми має бути похідний клас, що унаслідується від базового класу «QMainWindow», в похідному класі, перевизначить функцію «event»

2. Напишіть гру «арештуй таракана» на формі хаотично рухається таракан, гравцю потрібно курсором миші обвести таракана, таким чином заперши його в намальованій фігурі. В реалізації програми має бути похідний клас, що унаслідується від базового класу «QMainWindow», в похідному класі, перевизначить функцію «event».

3. Напишіть інтерфейс до лабораторної роботи №1. Реалізуйте можливість введення інформації про об’єкт, а також виведення інформації різного типу (наприклад тип1: кількість об’єктів, тип2: властивості об’єкту) в одному label по запиту з клавіатури. В реалізації програми має бути похідний клас, що унаслідується від базового класу «QMainWindow», в похідному класі, перевизначить функцію «event».

Варіант 1

1. Розробіть систему керування транспортними засобами, яка включатиме різні типи транспортних засобів, такі як автомобілі, мотоцикли і вантажівки. Кожен з цих типів має власні характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен транспортний засіб повинен мати наступні характеристики:

- Запас палива (у літрах)
- Швидкість (у км/год)
- Вартість (у доларах)

Кожен тип транспортного засобу має власні додаткові характеристики:

Автомобіль:

- Кількість дверей
- Максимальна швидкість (у км/год)

Мотоцикл:

- Тип двигуна (бензиновий, електричний і т.д.)
- Об'єм двигуна (у куб. см)

Вантажівка:

- Максимальне навантаження (у кг)
- Кількість вісей

2. Створіть базовий абстрактний клас Vehicle з віртуальними функціями та використати поліморфізм для реалізації методу обчислення вартості палива на відстань. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи Car, Motorcycle і Truck, які успадковуються від класу Vehicle. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо введений негативний запас палива або швидкість, або якщо відстань для обчислення вартості палива менше або дорівнює нулю.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 2

1. Розробіть систему керування ЖД вокзалом, яка включатиме обробку різних типів поїздів, таких як пасажирські поїзди та вантажні поїзди. Кожен тип поїзда має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен поїзд має наступні характеристики:

- Номер поїзда
- Пункт відправлення
- Пункт прибуття
- Час відправлення
- Час прибуття

Кожен тип поїзда має власні додаткові характеристики:

Пасажирський поїзд:

- Кількість місць (сидячих та ліжачих)

Вантажний поїзд:

- Максимальна вага вантажу (у тоннах)
- Кількість вагонів

2. Створіть базовий абстрактний клас Train з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи PassengerTrain та FreightTrain, які успадковуються від класу Train. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо введений час прибуття раніше або дорівнює часу відправлення.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 3

1. Розробіть систему керування продуктовим магазином, яка включатиме обробку різних типів продуктів, таких як фрукти, овочі та молочні продукти. Кожен тип продукту має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен продукт має наступні характеристики:

- Назва продукту

- Ціна (за одиницю товару)
- Кількість на складі

Кожен тип продукту має власні додаткові характеристики:

Фрукти:

- Тип фрукту (яблуко, банан, апельсин і т.д.)
- Вага (у кілограмах)

Овочі:

- Тип овоча (морква, капуста, буряк і т.д.)
- Країна походження

Молочні продукти:

- Тип продукту (молоко, йогурт, сир і т.д.)
- Виробник

2. Створіть базовий абстрактний клас Product з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи Fruit, Vegetable та DairyProduct, які успадковуються від класу Product. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо введена некоректна кількість на складі або негативна ціна.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 4

1. Розробіть систему керування банком, яка включатиме обробку різних типів банківських рахунків, таких як звичайний рахунок та рахунок з відсотковою ставкою. Кожен тип рахунку має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен банківський рахунок має наступні характеристики:

- Номер рахунку

- Власник рахунку
- Баланс

Кожен тип рахунку має власні додаткові характеристики:

Звичайний рахунок:

- Мінімальний дозволений баланс

Рахунок з відсотковою ставкою:

- Відсоткова ставка

2. Створіть базовий абстрактний клас BankAccount з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи RegularAccount та InterestAccount, які успадковуються від класу BankAccount. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо некоректний номер рахунку або негативна відсоткова ставка.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 5

1. Розробіть систему керування університетом, яка включатиме обробку різних типів студентів та викладачів. Кожен тип особи має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожна особа (студент або викладач) має наступні характеристики:

- Ім'я
- Вік
- Стать

Кожен тип особи має власні додаткові характеристики:

Студент:

- Курс
- Спеціальність

Викладач:

- Посада
- Предмет, який він викладає

2. Створіть базовий абстрактний клас Person з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи Student та Professor, які успадковуються від класу Person. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо некоректний вік або порожнє ім'я.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 6

1. Розробіть систему керування магазином, яка включатиме обробку різних типів товарів. Кожен тип товару має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен товар має наступні характеристики:

- Назва
- Ціна
- Кількість на складі

Кожен тип товару має власні додаткові характеристики:

Меблі:

- Матеріал
- Колір

Електроніка:

- Бренд
- Гарантійний термін

2. Створіть базовий абстрактний клас Product з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи Furniture та Electronics, які успадковуються від класу Product. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо некоректна ціна товару або від'ємна кількість на складі.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 7

1. Розробіть систему керування човнами, яка включатиме обробку різних типів човнів. Кожен тип човна має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен човен має наступні характеристики:

- Назва
- Максимальна швидкість
- Вантажопідйомність

Кожен тип човна має власні додаткові характеристики:

Весловий човен:

- Кількість весел
- Тип матеріалу

Моторний човен:

- Потужність двигуна
- Тип палива

2. Створіть базовий абстрактний клас Boat з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи Rowboat та Motorboat, які успадковуються від класу Boat. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо некоректна швидкість човна або від'ємна вантажопідйомність.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 8

1. Розробіть систему керування одягом, яка включатиме різні типи одягу. Кожен тип одягу має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен одяг має наступні характеристики:

- Назва
- Розмір
- Колір

Кожен тип одягу має власні додаткові характеристики:

Футболка:

- Матеріал
- Тип коміра

Штани:

- Матеріал
- Тип застібки

2. Створіть базовий абстрактний клас Clothing з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи TShirt та Pants, які успадковуються від класу Clothing. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо некоректний розмір одягу або порожнє ім'я.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 9

1. Розробіть систему управління аеропортом, яка включатиме різні типи повітряних суден. Кожен тип повітряного судна має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожне повітряне судно має наступні характеристики:

- Номер рейсу
- Максимальна вага
- Максимальна кількість пасажирів

Кожен тип повітряного судна має власні додаткові характеристики:

Літак:

- Модель
- Кількість двигунів

Гвинтокрил:

- Кількість гвинтів
- Радіус дії

2. Створіть базовий абстрактний клас Aircraft з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи Airplane та Helicopter, які успадковуються від класу Aircraft. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо некоректний номер рейсу або від'ємна вага повітряного судна.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Варіант 10

1. Розробіть систему управління лікарнею, яка включатиме різні типи медичних спеціалістів та пацієнтів. Кожен тип спеціаліста та пацієнта має свої характеристики, які потрібно реалізувати за допомогою наслідування, сетерів та гетерів.

Кожен медичний спеціаліст має наступні характеристики:

- Ім'я
- Спеціалізація
- Вік

Кожен пацієнт має наступні характеристики:

- Ім'я
- Вік
- Діагноз

Кожен тип медичного спеціаліста має власні додаткові характеристики:

Лікар:

- Стаж роботи
- Число успішно проведених операцій

Медсестра:

- Рівень кваліфікації
- Число виконаних процедур

2. Створіть базовий абстрактний клас MedicalPersonnel (Медичний персонал) з віртуальними функціями та використати поліморфізм для реалізації додаткових методів та функцій. Також, додайте виняткові ситуації для обробки некоректних даних.

3. Створіть похідні класи Doctor та Nurse, які успадковуються від класу MedicalPersonnel. Реалізуйте в них відповідні віртуальні функції та додайте додаткові характеристики, які були зазначені вище.

4. У вашій програмі мають бути використані виняткові ситуації для обробки некоректних даних, наприклад, якщо некоректне ім'я пацієнта або вік, який перевищує припустимий діапазон.

Ви можете розширити його, додати додаткові методи та функціональні можливості, які вам здаються відповідними.

Лабораторна робота №5 Перевантаження операторів

Мета: ознайомитись з поняттям перевантаження операторів та навчитись їх програмно реалізовувати мовою С++.

Теоретична частина

Перевантаження операторів в С++ дозволяє нам змінювати стандартну поведінку операторів для об'єктів класу. Вихідні типи операторів, такі як +, -, *, /, =, <, == та багато інших, можна змінити таким чином, щоб вони працювали з об'єктами класу, враховуючи наші власні правила.

Оператори можна перевантажити як функції-члени класу або як зовнішні функції. Коли оператор перевантажується як функція-член класу, вона має наступний формат:

```
return_type operatorоператор(parameters)
{
    // Тіло функції
}
```

Тут operatorоператор є ключовим словом operator з послідовністю операторів, які ми хочемо перевантажити. Наприклад, operator+ перевантажує оператор додавання.

Оператори також можуть бути перевантажені як зовнішні функції. У цьому випадку, формат функції виглядає наступним чином:

```
return_type operator оператор(parameters)
{
    // Тіло функції
}
```

Зазвичай зовнішні функції-оператори викликаються з двома аргументами, але можуть мати інший кількість параметрів, в залежності від типу оператора, який ми перевантажуємо.

Важливо зазначити, що не всі оператори можна перевантажити. Наприклад, оператори . (крапка) і .* (стрілка на вказівник на член) не можуть бути перевантажені. Крім того, оператори, які змінюють пріоритети виразів, такі як () (дужки) і [] (квадратні дужки), також не можуть бути перевантажені.

При перевантаженні операторів важливо враховувати загальні правила їх використання і зберігати логічну інтерпретацію відповідних операцій. Перевантажені оператори повинні залишатися зрозумілими і прийнятними з точки зору інших розробників, які можуть використовувати ваш код.

Наприклад:

```
class Complex {
private:
    double real;
    double imaginary;

public:
        Complex(double real, double imaginary) : real(real), 
imaginary(imaginary) {}

    // Перевантаження оператора +
    Complex operator+(const Complex& other) const {
            return Complex(real + other.real, imaginary + 
other.imaginary);
    }
};
```

У цьому прикладі ми перевантажили оператор + для класу Complex. У функції-члені operator+ ми додаємо відповідні реальні та уявні частини об'єктів Complex і повертаємо новий об'єкт Complex, який є сумою двох комплексних чисел.

Тепер ми можемо використовувати перевантажений оператор + для додавання комплексних чисел:

```
Complex a(2.0, 3.5);
Complex b(1.5, 4.2);

Complex c = a + b;  // Виклик перевантаженого оператора +

// Виведення результату
std::cout << "Результат: " << c.getReal() << " + " << 
c.getImaginary() << "i" << std::endl;
```

У цьому прикладі ми створюємо два об'єкти Complex a і b. Потім ми використовуємо перевантажений оператор + для додавання цих об'єктів і отримання результату в об'єкті c. Нарешті, ми виводимо результат додавання.

Одним із найпоширеніших прикладів є **перевантаження оператора виводу <<**, що дає змогу виводити об'єкти класу у зручному форматі через std::cout, як

звичайні змінні. Розглянемо приклад, у якому перевантажено оператор виводу для класу Matrix, що представляє просту двовимірну матрицю:

```
#include <iostream>

class Matrix {
private:
    int data[2][3]; // Матриця 2x3

public:
    // Конструктор, що задає значення елементів
    Matrix() {
        data[0][0] = 1; data[0][1] = 2; data[0][2] = 3;
        data[1][0] = 4; data[1][1] = 5; data[1][2] = 6;
    }

    // Перевантаження оператора <<
      friend std::ostream& operator<<(std::ostream& out, const
Matrix& m) {
        for (int i = 0; i < 2; ++i) {
            for (int j = 0; j < 3; ++j) {
                out << m.data[i][j] << " ";
            }
            out << std::endl;
        }
        return out;
    }
};

int main() {
    Matrix m;
    std::cout << "Matrix:\n" << m;
    return 0;
}
```

friend std::ostream& operator<< — ми оголошуємо функцію друком другом класу, щоб вона мала доступ до приватних полів.

Функція виводить усі елементи матриці у вигляді рядків і стовпців.

Завдання 1:

Перевантажте оператори для завдання № 2 з лабораторної роботи №2.

Завдання 2:

Продовжить розробку гри «Спіймай муху», тепер ваша муха має стати класом, реалізуйте декілька рівнів гри, на кожному наступному рівні, має з’являтися нова муха, яка буде класом наслідником. Також, з другого рівня на

вашій формі має з’явитися вікно, муха має намагатися вилетіти в це вікно, задача гравця відігнати муху від вікна і загнати її в пастку.

Завдання 3:

Варіант 1

Створіть клас Vector, який представляє тривимірний вектор у просторі. В цьому класі перевантажте наступні оператори:

1. Оператор + для додавання двох векторів.
2. Оператор - для віднімання двох векторів.
3. Оператор * для множення вектора на скаляр.
4. Оператор == для порівняння двох векторів на рівність.
5. Оператор != для порівняння двох векторів на нерівність.
6. Оператор << для виводу вектора у форматі (x, y, z).

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з векторами.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Vector і виконайте з ними операції додавання, віднімання, множення на скаляр, порівняння на рівність та виведення на екран.

Варіант 2

Створіть клас Complex, який представляє комплексне число у вигляді a + bi, де a та b - це дійсні числа, а i - уявна одиниця. В класі Complex перевантажте наступні оператори:

1. Оператор + для додавання двох комплексних чисел.
2. Оператор - для віднімання двох комплексних чисел.
3. Оператор * для множення двох комплексних чисел.
4. Оператор / для ділення одного комплексного числа на інше.
5. Оператор == для порівняння двох комплексних чисел на рівність.
6. Оператор != для порівняння двох комплексних чисел на нерівність.
7. Оператор << для виводу комплексного числа у форматі "a + bi".

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з комплексними числами.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Complex і виконайте з ними операції додавання, віднімання, множення, ділення, порівняння на рівність та виведення на екран.

Варіант 3

Створіть клас Matrix, який представляє квадратну матрицю розміром NxN. В класі Matrix перевантажте наступні оператори:

1. Оператор + для додавання двох матриць.
2. Оператор - для віднімання двох матриць.
3. Оператор * для множення двох матриць.
4. Оператор * для множення матриці на скаляр.
5. Оператор == для порівняння двох матриць на рівність.
6. Оператор != для порівняння двох матриць на нерівність.
7. Оператор << для виводу матриці на екран.

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з матрицями.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Matrix і виконайте з ними операції додавання, віднімання, множення, порівняння на рівність та виведення на екран.

Варіант 4

Створіть клас String, який представляє рядок символів. В класі String перевантажте наступні оператори:

1. Оператор + для конкатенації двох рядків.
2. Оператор == для порівняння двох рядків на рівність.
3. Оператор != для порівняння двох рядків на нерівність.
4. Оператор [] для доступу до символу за індексом.

5. Оператор << для виводу рядка на екран.

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з рядками.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу String і виконайте з ними операції конкатенації, порівняння на рівність, доступу до символу за індексом та виведення на екран.

Варіант 5

Створіть клас Fraction, який представляє дріб у вигляді чисельника та знаменника. В класі Fraction перевантажте наступні оператори:

1. Оператор + для додавання двох дробів.
2. Оператор - для віднімання двох дробів.
3. Оператор * для множення двох дробів.
4. Оператор / для ділення одного дробу на інший.
5. Оператор == для порівняння двох дробів на рівність.
6. Оператор != для порівняння двох дробів на нерівність.
7. Оператор << для виводу дробу у форматі "чисельник/знаменник".

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з дробами.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Fraction і виконайте з ними операції додавання, віднімання, множення, ділення, порівняння на рівність та виведення на екран.

Варіант 6

Створіть клас Time, який представляє час у годинах, хвилинах та секундах. В класі Time перевантажте наступні оператори:

1. Оператор + для додавання двох часів.
2. Оператор - для віднімання двох часів.
3. Оператор ++ (постфіксний) для інкременту часу на одну секунду.

4. Оператор -- (постфіксний) для декременту часу на одну секунду.
5. Оператор == для порівняння двох часів на рівність.
6. Оператор != для порівняння двох часів на нерівність.
7. Оператор << для виводу часу у форматі "гг:хх:сс".

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з часом.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Time і виконайте з ними операції додавання, віднімання, інкременту, декременту, порівняння на рівність та виведення на екран.

Варіант 7

Створіть клас Date, який представляє дату у форматі день, місяць, рік. В класі Date перевантажте наступні оператори:

1. Оператор ++ (префіксний) для інкременту дати на 1 день.
2. Оператор -- (префіксний) для декременту дати на 1 день.
3. Оператор + для додавання до дати заданої кількості днів.
4. Оператор - для віднімання від дати заданої кількості днів.
5. Оператор == для порівняння двох дат на рівність.
6. Оператор != для порівняння двох дат на нерівність.
7. Оператор << для виводу дати у форматі "дд/мм/рррр".

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з датами.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть об'єкт класу Date і виконайте з ним операції інкременту, декременту, додавання, віднімання, порівняння на рівність та виведення на екран.

Варіант 8

Створіть клас Rectangle, який представляє прямокутник з заданими шириною та висотою. Ширина та висота повинні бути додатними цілими числами. В класі Rectangle перевантажте наступні оператори:

1. Оператор + для об'єднання двох прямокутників. Результатом має бути новий прямокутник, який охоплює обидва початкових прямокутники.
2. Оператор - для віднімання одного прямокутника від іншого. Результатом має бути новий прямокутник, який представляє різницю між початковими прямокутниками.
3. Оператор * для масштабування прямокутника. Передавайте ціле число як аргумент, і результатом має бути новий прямокутник, який має ширину та висоту, помножені на це число.
4. Оператор / для зменшення прямокутника. Передавайте ціле число як аргумент, і результатом має бути новий прямокутник, який має ширину та висоту, поділені на це число.
5. Оператор == для порівняння двох прямокутників на рівність.
6. Оператор != для порівняння двох прямокутників на нерівність.
7. Оператор << для виводу прямокутника у зрозумілому форматі.

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з прямокутниками.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Rectangle і виконайте з ними операції об'єднання, віднімання, масштабування, зменшення, порівняння на рівність та виведення на екран.

Варіант 9

Створіть клас Point, який представляє точку в тривимірному просторі з координатами (x, y, z). Координати точки повинні бути дійсними числами. В класі Point перевантажте наступні оператори:

1. Оператор + для додавання двох точок. Результатом має бути нова точка, координати якої обчислюються як сума координат відповідних точок.

2. Оператор - для віднімання однієї точки від іншої. Результатом має бути нова точка, координати якої обчислюються як різниця координат відповідних точок.
3. Оператор * для множення точки на дійсне число. Передайте дійсне число як аргумент, і результатом має бути нова точка, координати якої обчислюються як добуток кожної координати точки на це число.
4. Оператор == для порівняння двох точок на рівність.
5. Оператор != для порівняння двох точок на нерівність.
6. Оператор << для виводу точки у зрозумілому форматі.

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з точками.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Point і виконайте з ними операції додавання, віднімання, множення на дійсне число, порівняння на рівність, а також виведіть їх на екран.

Варіант 10

Створіть клас Employee, який представляє працівника в офісі. Клас Employee повинен мати наступні властивості:

- Ім'я працівника (name) - рядок.
- Вік працівника (age) - ціле число.
- Заробітна плата працівника (salary) - дійсне число.

В класі Employee перевантажте наступні оператори:

1. Оператор + для об'єднання двох працівників. Результатом має бути новий об'єкт Employee, в якому значення властивостей встановлені згідно з наступними правилами:
   - Ім'я нового працівника має бути поєднанням імен обох початкових працівників.
   - Вік нового працівника має бути середнім значенням віку двох початкових працівників (округлено до найближчого цілого).

- Заробітна плата нового працівника має бути сумою заробітних плат двох початкових працівників.

2. Оператор == для порівняння двох працівників на рівність. Працівники вважаються рівними, якщо всі їхні властивості (ім'я, вік, заробітна плата) мають однакові значення.
3. Оператор != для порівняння двох працівників на нерівність. Працівники вважаються нерівними, якщо хоча б одна з їхніх властивостей відрізняється.

Додайте в клас Employee також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з працівниками.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Employee і виконайте з ними операції об'єднання, порівняння на рівність. Виведіть результати на екран.

Варіант 11

Створіть клас Book, який представляє книгу. Клас Book повинен мати наступні властивості:

- Назва книги (title) - рядок.
- Автор книги (author) - рядок.
- Рік видання книги (year) - ціле число.

В класі Book перевантажте наступні оператори:

1. Оператор == для порівняння двох книг на рівність. Книги вважаються рівними, якщо їхні назви, автори та роки видання мають однакові значення.
2. Оператор != для порівняння двох книг на нерівність. Книги вважаються нерівними, якщо хоча б одна з їхніх властивостей відрізняється.
3. Оператор < для порівняння двох книг за алфавітним порядком назви. Книга book1 вважається меншою за книгу book2, якщо назва book1 передує алфавітно назві book2.

4. Оператор > для порівняння двох книг за алфавітним порядком назви. Книга book1 вважається більшою за книгу book2, якщо назва book1 слідує алфавітно за назвою book2.
5. Оператор << для виводу книги на екран у зрозумілому форматі. Виведіть на екран назву, автора та рік видання книги.

Додайте в клас Book також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з книгами.

Напишіть програму, де ви використовуєте цей клас та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу Book і виконайте з ними порівняння, виведення на екран та інші операції.

Варіант 12

Створіть клас Car, який представляє автомобіль. Клас Car повинен мати наступні властивості:

- Марка автомобіля (brand) - рядок.
- Рік випуску автомобіля (year) - ціле число.
- Швидкість автомобіля (speed) - десяткове число.

В класі Car перевантажте наступні оператори:

1. Оператор + для об'єднання двох автомобілів. Результатом має бути новий автомобіль, який має марку та рік випуску першого автомобіля, а швидкість - суму швидкостей обох автомобілів.
2. Оператор - для віднімання одного автомобіля від іншого. Результатом має бути новий автомобіль, який має марку та рік випуску першого автомобіля, а швидкість - різницю швидкостей обох автомобілів.
3. Оператор * для масштабування швидкості автомобіля. Передайте ціле число як аргумент, і результатом має бути новий автомобіль, який має марку та рік випуску початкового автомобіля, а швидкість - швидкість початкового автомобіля, помножену на це число.
4. Оператор / для зменшення швидкості автомобіля. Передайте ціле число як аргумент, і результатом має бути новий автомобіль, який має марку та

рік випуску початкового автомобіля, а швидкість - швидкість початкового автомобіля, поділену на це число.

5. Оператор == для порівняння двох автомобілів на рівність. Автомобілі вважаються рівними, якщо марка, рік випуску та швидкість мають однакові значення.
6. Оператор != для порівняння двох автомобілів на нерівність. Автомобілі вважаються нерівними, якщо марка, рік випуску або швидкість мають різні значення.

Додайте в клас Car також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи.

Напишіть програму, де ви використовуєте клас Car та перевірте роботу всіх перевантажених операторів. Створіть кілька об'єктів класу Car і виконайте з ними операції об'єднання, віднімання, масштабування, зменшення, порівняння на рівність та нерівність.

Варіант 13

Створіть клас BankAccount, який представляє банківський рахунок. Клас BankAccount повинен мати наступні властивості:

- Номер рахунку (accountNumber) - рядок.
- Баланс рахунку (balance) - дійсне число.

В класі BankAccount перевантажте наступні оператори:

1. Оператор + для додавання двох рахунків. Результатом має бути новий об'єкт BankAccount, у якого номер рахунку буде зберігати поєднання номерів рахунків, а баланс - суму балансів початкових рахунків.
2. Оператор - для віднімання одного рахунку від іншого. Результатом має бути новий об'єкт BankAccount, у якого номер рахунку буде зберігати початковий номер першого рахунку, а баланс - різницю балансів початкових рахунків.
3. Оператор * для множення балансу рахунку на певний множник. Передавайте дійсне число як аргумент, і результатом має бути новий об'єкт

BankAccount, у якого номер рахунку та баланс будуть зберігати початкові значення, помножені на заданий множник.

4. Оператор / для ділення балансу рахунку на певний дільник. Передавайте дійсне число як аргумент, і результатом має бути новий об'єкт BankAccount, у якого номер рахунку та баланс будуть зберігати початкові значення, поділені на заданий дільник.
5. Оператор += для додавання до поточного балансу рахунку певної суми. Передавайте дійсне число як аргумент, і змінюйте поточний об'єкт BankAccount, збільшуючи його баланс на задану суму.
6. Оператор -= для віднімання від поточного балансу рахунку певної суми. Передавайте дійсне число як аргумент, і змінюйте поточний об'єкт BankAccount, зменшуючи його баланс на задану суму.
7. Оператор == для порівняння двох рахунків на рівність за номером рахунку та балансом.
8. Оператор != для порівняння двох рахунків на нерівність за номером рахунку та балансом.
9. Оператор << для виводу рахунку у зрозумілому форматі. Виведіть номер рахунку та баланс у форматі "Account: {номер_рахунку}, Balance: {баланс}".

Додайте в клас також необхідні конструктори, деструктор та інші методи, які можуть знадобитись для роботи з рахунком.

Напишіть програму, де ви використовуєте цей клас BankAccount та перевірте роботу всіх перевантажених операторів. Створіть декілька об'єктів класу BankAccount і виконайте з ними операції додавання, віднімання, множення, ділення, порівняння на рівність, зміни балансу та виведення на екран.

Лабораторна робота №6 Шаблони в С++

Мета: ознайомитись з основними поняттями шаблони та навчитись їх програмно реалізовувати мовою С++.

Теоретична частина

Шаблони є потужним інструментом у мові програмування С++. Вони дозволяють програмістам створювати загальні, параметризовані класи та функції, які можуть працювати з різними типами даних. Використання шаблонів дозволяє писати більш загальні й універсальні програми, що збільшує перевикористання коду, покращує його зрозумілість та підтримуваність.

Шаблони класів дозволяють визначити загальну структуру класу, незалежну від конкретного типу даних. Вони визначаються за допомогою ключового слова "template" і параметрів шаблону, які можуть бути типами даних, константами або іншими шаблонами. Наприклад, ось простий шаблон класу "Stack", який реалізує структуру стеку:

```
template <typename T>
class Stack {
private:
    std::vector<T> elements; // Зберігає елементи стеку

public:
    void push(const T& element) {
        elements.push_back(element);
    }

    void pop() {
        elements.pop_back();
    }

    T& top() {
        return elements.back();
    }

    bool empty() const {
        return elements.empty();
    }
};
```

У цьому прикладі тип даних "T" є параметром шаблону. Клас "Stack" може бути використаний для створення стеку будь-якого типу даних. Наприклад:

```
Stack<int> intStack; // Стек цілих чисел
Stack<std::string> stringStack; // Стек рядків
```

Крім шаблонів класів, в C++ також є шаблони функцій. Вони дозволяють визначати функції, які працюють з різними типами даних. Шаблони функцій використовують таку саму синтаксичну конструкцію з ключовим словом "template" і параметрами шаблону. Наприклад, ось проста шаблонна функція "max", яка повертає більший із двох переданих значень:

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Ця функція може бути використана для порівняння значень будь-якого типу даних, для якого визначені оператори порівняння.

```
int result1 = max(10, 20); // Поверне 20
double result2 = max(3.14, 2.71); // Поверне 3.14
```

Шаблони в C++ також підтримують спеціалізацію, що дозволяє визначати варіанти шаблону для конкретних типів даних або спеціальних випадків. Зазвичай спеціалізацію використовують для оптимізації або визначення специфічної поведінки для певного типу даних.

Шаблони в C++ - це потужний інструмент, який дозволяє створювати загальні та універсальні програми. Вони забезпечують перевикористання коду, покращують зрозумілість і підтримуваність програм, а також дають можливість програмістам працювати з різними типами даних без необхідності дублювання коду.

Завдання

Варіант 1

Створи програму для керування бібліотекою книжок. Кожна книжка має заголовок, автора та рік видання. Необхідно створити клас Book, що містить ці дані.

Крім того, створи шаблонний клас Library, який представляє колекцію книжок. Цей клас має методи для додавання книжки, видалення книжки за заголовком, виведення списку всіх книжок та пошуку книжок за автором.

Забезпеч універсальність класу Library, використовуючи шаблонний параметр для типу книжки. Це дозволить тобі працювати з різними типами книжок.

Для демонстрації роботи програми:

1. Створи об'єкт Library для зберігання книжок.

2. Додай до бібліотеки декілька книжок різних типів, наприклад, художніх, наукових, документальних тощо.

3. Виведи на екран список всіх книжок в бібліотеці.

4. Знайди та виведи на екран усі книжки певного автора.

5. Видали з бібліотеки книжку за заголовком.

6. Знову виведи на екран оновлений список книжок в бібліотеці.

Не забудь використовувати шаблонний клас Library для зберігання книжок будь-якого типу. Також, забезпеч, щоб клас Book мав методи доступу до своїх полів (заголовок, автор, рік видання).

Варіант 2

Створи програму для керування студентським списком. Кожен студент має ім'я, прізвище та середній бал. Необхідно створити клас Student, який містить ці дані.

Крім того, створи шаблонний клас StudentList, який представляє колекцію студентів. Цей клас має методи для додавання студента, видалення студента за прізвищем, виведення списку всіх студентів та пошуку студентів за середнім балом.

Забезпеч універсальність класу StudentList, використовуючи шаблонний параметр для типу студента. Це дозволить тобі працювати з різними типами студентів (наприклад, інженерний факультет, медичний факультет тощо).

Для демонстрації роботи програми:

1. Створи об'єкт StudentList для зберігання студентів.

2. Додай до списку декілька студентів різних факультетів.

3. Виведи на екран список всіх студентів у списку.

4. Знайди та виведи на екран усіх студентів з певним середнім балом.

5. Видали зі списку студента за прізвищем.

6. Знову виведи на екран оновлений список студентів у списку.

Не забудь використовувати шаблонний клас StudentList для зберігання студентів будь-якого типу. Також, забезпеч, щоб клас Student мав методи доступу до своїх полів (ім'я, прізвище, середній бал).

Варіант 3

Створи програму для керування банківським рахунком клієнтів. Кожен клієнт має ім'я, прізвище та баланс на рахунку. Необхідно створити клас Customer, який містить ці дані.

Крім того, створи шаблонний клас BankAccount, який представляє банківський рахунок. Цей клас має методи для додавання коштів на рахунок, зняття коштів з рахунку, переказу коштів між рахунками та виведення балансу рахунку.

Забезпеч універсальність класу BankAccount, використовуючи шаблонний параметр для типу клієнта. Це дозволить тобі працювати з різними типами клієнтів (наприклад, фізичні особи, юридичні особи тощо).

Для демонстрації роботи програми:

1. Створи об'єкт BankAccount для керування рахунком клієнта.

2. Додай кошти на рахунок за допомогою методу deposit().

3. Зніми кошти з рахунку за допомогою методу withdraw().

4. Перекажи кошти з одного рахунку на інший за допомогою методу transfer().

5. Виведи на екран баланс рахунку за допомогою методу getBalance().

Не забудь використовувати шаблонний клас BankAccount для керування рахунками різних типів клієнтів. Також, забезпеч, щоб клас Customer мав методи доступу до своїх полів (ім'я, прізвище, баланс).

Варіант 4

Створи програму для керування складом товарів у магазині. Кожен товар має назву, ціну та кількість на складі. Необхідно створити клас Product, який містить ці дані.

Крім того, створи шаблонний клас Inventory, який представляє інвентар магазину. Цей клас має методи для додавання товару на склад, видалення товару зі складу за назвою, виведення списку всіх товарів та пошуку товару за ціною.

Забезпеч універсальність класу Inventory, використовуючи шаблонний параметр для типу товару. Це дозволить тобі працювати з різними типами товарів (наприклад, продукти харчування, електроніка, одяг тощо).

Для демонстрації роботи програми:

1. Створи об'єкт Inventory для зберігання товарів на складі.
2. Додай до інвентаря декілька товарів різних типів.
3. Виведи на екран список всіх товарів на складі.
4. Знайди та виведи на екран всі товари за певною ціною.
5. Видали з інвентаря товар за назвою.
6. Знову виведи на екран оновлений список товарів на складі.

Не забудь використовувати шаблонний клас Inventory для зберігання товарів будь-якого типу. Також, забезпеч, щоб клас Product мав методи доступу до своїх полів (назва, ціна, кількість).

Варіант 5

Створи програму для керування рестораном. У ресторані є різні страви, які клієнти можуть замовити. Кожна страва має назву, опис та ціну. Необхідно створити клас Dish, який містить ці дані.

Крім того, створи шаблонний клас Menu, який представляє меню ресторану. Цей клас має методи для додавання страви до меню, видалення страви за назвою, виведення списку всіх страв та пошуку страви за ціною.

Забезпеч універсальність класу Menu, використовуючи шаблонний параметр для типу страви. Це дозволить тобі працювати з різними типами страв (наприклад, страви із м'ясом, страви для вегетаріанців тощо).

Для демонстрації роботи програми:

1. Створи об'єкт Menu для зберігання страв.

2. Додай до меню декілька різних страв.

3. Виведи на екран список всіх страв у меню.

4. Знайди та виведи на екран всі страви за певною ціною.

5. Видали з меню страву за назвою.

6. Знову виведи на екран оновлений список страв у меню.

Не забудь використовувати шаблонний клас Menu для зберігання страв будь-якого типу. Також, забезпеч, щоб клас Dish мав методи доступу до своїх полів (назва, опис, ціна).

Варіант 6

Створи програму для керування туристичним агентством. Туристичне агентство пропонує різні тури та послуги для подорожей. Кожен тур має назву, опис, ціну та тривалість. Необхідно створити клас Tour, який містить ці дані.

Крім того, створи шаблонний клас TravelAgency, який представляє туристичне агентство. Цей клас має методи для додавання туру до агентства, видалення туру за назвою, виведення списку всіх доступних турів та пошуку туру за ціною.

Забезпеч універсальність класу TravelAgency, використовуючи шаблонний параметр для типу туру. Це дозволить тобі працювати з різними типами турів (наприклад, пляжні тури, екскурсійні тури, гірськолижні тури тощо).

Для демонстрації роботи програми:

1. Створи об'єкт TravelAgency для зберігання турів.

2. Додай до агентства декілька різних турів.

3. Виведи на екран список всіх доступних турів.

4. Знайди та виведи на екран всі тури за певною ціною.

5. Видали з агентства тур за назвою.

6. Знову виведи на екран оновлений список доступних турів.

Не забудь використовувати шаблонний клас TravelAgency для зберігання турів будь-якого типу. Також, забезпеч, щоб клас Tour мав методи доступу до своїх полів (назва, опис, ціна, тривалість).

Варіант 7

Створи програму для керування автомобільним сервісним центром. Сервісний центр займається обслуговуванням автомобілів та наданням різних послуг. Кожен автомобіль має марку, модель та рік випуску. Необхідно створити клас Car, який містить ці дані.

Крім того, створи шаблонний клас ServiceCenter, який представляє сервісний центр. Цей клас має методи для додавання автомобіля до списку обслуговування, видалення автомобіля за маркою, виведення списку всіх автомобілів у сервісному центрі та пошуку автомобіля за роком випуску.

Забезпеч універсальність класу ServiceCenter, використовуючи шаблонний параметр для типу автомобіля. Це дозволить тобі працювати з різними типами автомобілів (наприклад, легкові, вантажні, спортивні тощо).

Для демонстрації роботи програми:

1. Створи об'єкт ServiceCenter для зберігання автомобілів.
2. Додай до сервісного центру декілька різних автомобілів.
3. Виведи на екран список всіх автомобілів у сервісному центрі.
4. Знайди та виведи на екран всі автомобілі за певним роком випуску.
5. Видали з сервісного центру автомобіль за маркою.
6. Знову виведи на екран оновлений список автомобілів у сервісному центрі.

Не забудь використовувати шаблонний клас ServiceCenter для зберігання автомобілів будь-якого типу. Також, забезпеч, щоб клас Car мав методи доступу до своїх полів (марка, модель, рік випуску).

Варіант 8

Створи програму для керування музичним магазином. Музичний магазин продає різні музичні інструменти та аксесуари. Кожен інструмент має назву, тип та ціну. Необхідно створити клас Instrument, який містить ці дані.

Крім того, створи шаблонний клас MusicStore, який представляє музичний магазин. Цей клас має методи для додавання інструменту до асортименту магазину, видалення інструменту за назвою, виведення списку всіх доступних інструментів та пошуку інструменту за типом.

Забезпеч універсальність класу MusicStore, використовуючи шаблонний параметр для типу інструменту. Це дозволить тобі працювати з різними типами інструментів (наприклад, гітари, барабани, клавішні тощо).

Для демонстрації роботи програми:

1. Створи об'єкт MusicStore для зберігання інструментів.
2. Додай до магазину декілька різних інструментів.
3. Виведи на екран список всіх доступних інструментів у магазині.
4. Знайди та виведи на екран всі інструменти певного типу.
5. Видали з магазину інструмент за назвою.
6. Знову виведи на екран оновлений список доступних інструментів у магазині.

Не забудь використовувати шаблонний клас MusicStore для зберігання інструментів будь-якого типу. Також, забезпеч, щоб клас Instrument мав методи доступу до своїх полів (назва, тип, ціна).

Варіант 9

Створи програму для керування спортивним клубом. Спортивний клуб має різні секції, кожна з яких спеціалізується на певному виді спорту. Кожна секція має назву, тренера та кількість учасників. Необхідно створити клас Section, який містить ці дані.

Крім того, створи шаблонний клас SportsClub, який представляє спортивний клуб. Цей клас має методи для додавання секції до клубу, видалення секції за назвою, виведення списку всіх доступних секцій та пошуку секції за кількістю учасників.

Забезпеч універсальність класу SportsClub, використовуючи шаблонний параметр для типу секції. Це дозволить тобі працювати з різними видами спорту (наприклад, футбол, баскетбол, плавання тощо).

Для демонстрації роботи програми:

1. Створи об'єкт SportsClub для зберігання секцій.

2. Додай до клубу декілька різних секцій.

3. Виведи на екран список всіх доступних секцій у клубі.

4. Знайди та виведи на екран всі секції з певною кількістю учасників.

5. Видали з клубу секцію за назвою.

6. Знову виведи на екран оновлений список доступних секцій у клубі.

Не забудь використовувати шаблонний клас SportsClub для зберігання секцій будь-якого типу. Також, забезпеч, щоб клас Section мав методи доступу до своїх полів (назва, тренер, кількість учасників).

Варіант 10

Створи програму для керування онлайн-курсами. Кожен курс має назву, викладача та кількість студентів. Необхідно створити клас Course, який містить ці дані.

Крім того, створи шаблонний клас OnlineLearningPlatform, який представляє платформу для онлайн-навчання. Цей клас має методи для додавання курсу до платформи, видалення курсу за назвою, виведення списку всіх доступних курсів та пошуку курсу за кількістю студентів.

Забезпеч універсальність класу OnlineLearningPlatform, використовуючи шаблонний параметр для типу курсу. Це дозволить тобі працювати з різними типами курсів (наприклад, програмування, мови, маркетинг тощо).

Для демонстрації роботи програми:

1. Створи об'єкт OnlineLearningPlatform для зберігання курсів.

2. Додай до платформи декілька різних курсів.

3. Виведи на екран список всіх доступних курсів на платформі.

4. Знайди та виведи на екран всі курси з певною кількістю студентів.

5. Видали з платформи курс за назвою.

6. Знову виведи на екран оновлений список доступних курсів на платформі.

Не забудь використовувати шаблонний клас OnlineLearningPlatform для зберігання курсів будь-якого типу. Також, забезпеч, щоб клас Course мав методи доступу до своїх полів (назва, викладач, кількість студентів).

Лабораторна робота №7 Контейнерні класи. Стандартна бібліотека шаблонів (STL) в C++.

Мета: ознайомитись Контейнерні класи та навчитись їх програмно реалізовувати мовою C++.

Теоретична частина

Контейнерний клас (або “клас-контейнер”) в мові C++ — це клас, призначений для зберігання і організації декількох об’єктів певного типу даних (користувацьких чи фундаментальних).

В мові програмування C++, контейнерні класи є класами, які призначені для зберігання та управління колекціями об'єктів. Вони надають зручний інтерфейс для додавання, видалення і доступу до елементів у колекції.

Контейнерні класи в C++ доступні завдяки бібліотеці стандартного шаблону (STL - Standard Template Library), яка включає в себе різні типи контейнерів, такі як вектори, списки, множини, асоціативні масиви тощо.

Основні типи контейнерів, які можна знайти в STL, включають наступні:

Вектор (vector) — це динамічний масив, реалізований як шаблонний контейнер у Стандартній бібліотеці шаблонів (STL) мови C++. Він дозволяє зберігати елементи одного типу, автоматично змінюючи розмір за потреби. Це один з найпопулярніших контейнерів у C++ завдяки своїй гнучкості, зручності використання та ефективності.

Контейнер vector працює подібно до масиву, але з розширеною функціональністю:

- Він зберігає елементи у послідовному порядку в пам’яті.
- Дозволяє швидкий доступ до елементів за індексом.
- Автоматично змінює розмір при додаванні нових елементів.
- Підтримує вставку, видалення, сортування, ітерацію тощо.
- Має тісну інтеграцію з ітераторами та алгоритмами STL.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric> // Для accumulate
```

```
using namespace std;

// Функція для додавання елемента до вектора
void addElement(vector<int>& vec, int value) {
    vec.push_back(value);
}

// Функція для видалення всіх входжень елемента з вектора
void removeElement(vector<int>& vec, int value) {
            vec.erase(remove(vec.begin(),   vec.end(),   value),
vec.end());
}

// Функція для пошуку елемента у векторі
bool findElement(const vector<int>& vec, int value) {
    return find(vec.begin(), vec.end(), value) != vec.end();
}

// Функція для виведення всіх елементів вектора
void printVector(const vector<int>& vec) {
    for (int num : vec) {
        cout << num << " ";
    }
    cout << endl;
}
// Функція для обчислення суми всіх елементів
int sumVector(const vector<int>& vec) {
    return accumulate(vec.begin(), vec.end(), 0);
}
int main() {
    vector<int> numbers;

    // Додаємо елементи від 0 до 9 по двічі
    for (int i = 0; i < 10; ++i) {
        addElement(numbers, i);
        addElement(numbers, i);
    }

    cout << "Початковий вектор:\n";
    printVector(numbers);

    // Сортування вектора
    sort(numbers.begin(), numbers.end());
    cout << "Вектор після сортування:\n";
    printVector(numbers);

    // Видалення числа 5 з вектора
```

```
        removeElement(numbers, 5);
        cout << "Після видалення 5:\n";
        printVector(numbers);
        // Пошук елемента 7
        int searchValue = 7;
        cout << "Число " << searchValue
              << (findElement(numbers, searchValue) ? " знайдено" :
" не знайдено")
             << " у векторі.\n";

        // Обчислення суми елементів
         cout << "Сума всіх елементів: " << sumVector(numbers) <<
endl;

        return 0;
    }
```

Дек (deque (double-ended queue — черга з двома кінцями)) — це контейнер з динамічним розміром, що дозволяє ефективно додавати та видаляти елементи як з початку, так і з кінця. Він реалізований у STL як шаблонний клас. Це потужний інструмент для ситуацій, коли потрібно швидко працювати з обома кінцями послідовності.

- На відміну від vector, який оптимізований для операцій з кінця, deque забезпечує:
- Швидкий доступ до елементів за індексом (як і vector);
- Швидке додавання/видалення як на початку, так і в кінці;
- Можливість використання STL-алгоритмів і ітераторів.
- Внутрішньо deque зберігає дані не в одному безперервному масиві, а в наборах блоків, що забезпечує ефективність для операцій з початку.

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <numeric> // для accumulate

using namespace std;

// Додати елемент в кінець
void addToEnd(deque<int>& dq, int value) {
    dq.push_back(value);
}
```

```
// Додати елемент на початок
void addToFront(deque<int>& dq, int value) {
    dq.push_front(value);
}

// Видалити елемент з кінця
void removeFromEnd(deque<int>& dq) {
    if (!dq.empty()) dq.pop_back();
}

// Видалити елемент з початку
void removeFromFront(deque<int>& dq) {
    if (!dq.empty()) dq.pop_front();
}

// Знайти елемент у deque
bool findElement(const deque<int>& dq, int value) {
    return find(dq.begin(), dq.end(), value) != dq.end();
}

// Вивести вміст deque
void printDeque(const deque<int>& dq) {
    for (int val : dq) {
        cout << val << " ";
    }
    cout << endl;
}

// Обчислити суму елементів
int sumDeque(const deque<int>& dq) {
    return accumulate(dq.begin(), dq.end(), 0);
}

int main() {
    deque<int> numbers;

    // Додавання елементів у кінець і початок
    for (int i = 1; i <= 5; ++i) {
        addToEnd(numbers, i);        // 1 2 3 4 5
        addToFront(numbers, i * 10); // 50 40 30 20 10 ...
    }

    cout << "Початковий deque:\n";
    printDeque(numbers);

    // Видалення одного елемента з початку та з кінця
    removeFromFront(numbers);
    removeFromEnd(numbers);
```

```
        cout << "Після видалення з початку і кінця:\n";
        printDeque(numbers);

        // Пошук елемента
        int searchValue = 3;
        cout << "Число " << searchValue
              << (findElement(numbers, searchValue) ? " знайдено" :
" не знайдено")
             << " у deque.\n";

        // Сума елементів
          cout << "Сума всіх елементів: " << sumDeque(numbers) <<
endl;

        return 0;
    }
```

Список(list) — це двозв’язний список, реалізований у вигляді шаблонного контейнера в STL. Кожен елемент списку зберігає: своє значення, вказівник на попередній елемент, вказівник на наступний елемент. Це дозволяє ефективно вставляти й видаляти елементи у будь-якому місці списку.

На відміну від vector чи deque, list не гарантує послідовне розміщення елементів у пам’яті. Замість цього: доступ до елементів не є миттєвим за індексом (list не підтримує [] або at()), але вставка та видалення елементів в будь-якому місці (включаючи середину) відбувається дуже швидко (O(1)), підтримуються двонапрямлені ітератори.

Цей контейнер добре підходить для ситуацій, де потрібно часто додавати або видаляти елементи в середині послідовності.

Приклад з використанням списків:

```
int main() {
    // Оголошення двозв'язного списку цілих чисел
    list<int> numbers;

    // Заповнення списку числами від 0 до 9:
     // Кожне число додається і на початок, і в кінець
списку
```

```
        for (int i = 0; i < 10; i++) {
              numbers.push_back(i);   // Додаємо елемент в
кінець списку
              numbers.push_front(i); // Додаємо той самий
елемент на початок
        }

        // Виведення елементів списку до сортування
        cout << "Елементи списку до сортування:\n";
          for (list<int>::iterator it = numbers.begin();
it != numbers.end(); ++it) {
              cout << *it << " "; // Виводимо кожен елемент
з пробілом
        }
        cout << endl;

        // Сортування елементів списку за зростанням
        numbers.sort();

        // Виведення елементів після сортування
        cout << "Елементи списку після сортування:\n";
          for (list<int>::iterator it = numbers.begin();
it != numbers.end(); ++it) {
              cout << *it << " "; // Виводимо кожен елемент
з пробілом
        }
        cout << endl;
        return 0;
    }
```

Завдання

Варіант 1

Розробіть систему керування магазином електроніки з використанням контейнерів vector, deque або list для зберігання інформації про продукти. Кожен продукт повинен мати назву, ціну та кількість на складі. Реалізуйте наступне:

- Додайте декілька продуктів до інвентаря.
- Виведіть список продуктів.
- Оновіть кількість одного або декількох продуктів.
- Виведіть оновлений список продуктів.
- Обчисліть загальну вартість усього інвентаря.

Додатково: реалізуйте окремі методи для додавання, редагування та виведення інформації про продукти. Виберіть один або кілька відповідних контейнерів (vector, deque, list) і поясніть свій вибір.

Варіант 2

Розробіть систему управління бібліотекою, використовуючи послідовні контейнерні класи (`vector`, `deque`, або `list`). У системі необхідно зберігати інформацію про книги (назва, автор, рік видання, ISBN).

Реалізуйте:

- Додавання декількох книг до бібліотеки.
- Виведення списку всіх книг.
- Видалення книги за ISBN.
- Пошук книг за автором або роком видання.
- Виведення результатів пошуку.

Додатково: реалізуйте метод для підрахунку загальної кількості книг у бібліотеці. Поясніть, чому вибраний контейнер найкраще підходить для реалізації задачі.

Варіант 3

Розробіть **систему управління завданнями**, яка використовує послідовні контейнерні класи (`vector`, `deque`, або `list`) для зберігання завдань. Кожне завдання має містити:

- назву,
- опис,
- дедлайн (у вигляді рядка або дати).

Реалізуйте:

- Додавання кількох завдань до списку.
- Виведення всіх завдань.
- Видалення одного або кількох завдань за назвою.
- Виведення оновленого списку завдань.
- Виведення завдань з певним дедлайном.

Додатково:

Реалізуйте метод сортування завдань за дедлайном або алфавітом. Поясніть, чому обраний контейнер є доцільним для задачі.

Варіант 4

Розробіть систему управління складом магазину автозапчастин, використовуючи послідовні контейнери STL (`vector`, `deque`, або `list`) для зберігання інформації про запчастини.

Кожна запчастина повинна містити:

- Назву,
- Виробника,
- Ціну,
- Кількість на складі.

Реалізуйте:

- Додавання кількох запчастин до списку.
- Виведення списку запчастин.
- Видалення однієї або кількох запчастин за назвою.
- Оновлення ціни та кількості для певної запчастини.
- Виведення оновленого списку запчастин.

Додатково:

Реалізуйте метод сортування запчастин за назвою або ціною. Поясніть, чому обрали саме цей тип контейнера.

Варіант 5

Реалізуйте систему, в якій дзвінки надходять і зберігаються за допомогою STL-контейнерів vector, deque або list.

Функціонал:

- Створіть структуру Call з полями: ім’я клієнта, тема звернення.
- Додайте кілька дзвінків до контейнера (vector, list, або deque).
- Виведіть список усіх дзвінків у порядку надходження.
- Видаліть перші N дзвінків (як оброблені).
- Виведіть оновлений список дзвінків.

Додатково (на вибір): пошук дзвінків за ключовим словом у темі або сортування за іменем.

Варіант 6

Реалізуйте систему керування запасами продуктів у супермаркеті (послідовні контейнери)

Функціонал:

- Створіть структуру Product з полями: назва, категорія, ціна, кількість.
- Використовуйте один із контейнерів: vector, deque або list для зберігання продуктів.
- Додайте кілька продуктів до контейнера.
- Виведіть повний список продуктів.
- Видаліть продукт(и) зі списку за назвою.
- Оновіть ціну та кількість для одного або кількох продуктів.
- Виведіть оновлений список продуктів.

Додатково:

- Сортування списку за назвою або ціною.

- Фільтрація продуктів за категорією.

Варіант 7

Реалізуйте систему керування готелями — основна інформація та обробка (лінійні контейнери)

Функціонал:

- Створіть структури Room (номер, категорія, ціна, доступність) та Reservation (номер кімнати, дати початку/кінця, кількість гостей).
- Зберігайте доступні кімнати у vector, deque або list.
- Додайте декілька кімнат до списку.
- Виведіть список доступних кімнат.
- Створіть список Reservation, додайте кілька записів про бронювання.
- Виведіть всі резервації.
- Скасуйте резервацію (видалення зі списку Reservation), оновіть статус кімнати (доступність).
- Виведіть оновлений список доступних кімнат.

Розширення:

- Пошук кімнат за категорією або ціною.
- Підрахунок кількості вільних кімнат.

Варіант 8

Реалізуйте систему керування поштовими відправленнями — основна робота з колекціями

Функціонал:

- Створіть структуру Parcel, яка містить:
  - Номер відправлення
  - Дані відправника
  - Дані одержувача
  - Вагу
  - Тип (лист, посилка, експрес тощо)

- Використайте vector, deque або list для збереження списку відправлень.
- Додайте декілька відправлень до списку.
- Виведіть список усіх відправлень.
- Видаліть певне відправлення за номером.
- Оновіть вагу та тип для конкретного відправлення.
- Виведіть оновлений список.

Розширення:

- Фільтрація за типом або мінімальною/максимальною вагою.
- Сортування списку за вагою або за типом.

Варіант 9

Реалізуйте систему керування студентськими оцінками — базові операції з колекціями

Функціонал:

- Створіть структуру Student, яка містить:
  - Ім’я
  - Прізвище
  - Список оцінок (vector<int> або list<int>)
- Використайте vector, deque або list для збереження списку студентів.
- Додайте кількох студентів із початковими оцінками.
- Виведіть список оцінок усіх студентів.
- Видаліть певного студента за ім’ям і прізвищем.
- Додайте оцінку певному студенту.
- Видаліть оцінку у певного студента.
- Виведіть оновлений список.

Розширення:

- Обчислення середнього балу кожного студента.
- Пошук студентів з середнім балом вище заданого порогу.

Варіант 10

Реалізуйте систему керування замовленням — базова реалізація

Функціонал:

- Створіть структуру Dish, яка містить:
    - Назву страви
    - Кількість
    - Ціну за одиницю
- Використайте vector, deque або list для збереження поточного замовлення.
- Додайте кілька страв до замовлення.
- Обчисліть і виведіть загальну суму замовлення.
- Видаліть певну страву за назвою.
- Додайте інші страви.
- Знову виведіть оновлену загальну суму.

Розширення:

- Вивід меню (список доступних страв).
- Вивід лише дорогих або дешевих страв у замовленні.

Лабораторна робота №8 Стандартна бібліотека шаблонів (STL) в C++: Контейнери-адаптери, ітератори, стек, черга, черга з пріоритетом.

Мета: Ознайомитись з контейнерами-адаптерами, ітераторами, стеком, чергою та чергою з пріоритетом зі стандартної бібліотеки шаблонів (STL) в C++. Навчитись використовувати їх на практиці.

Теоретична частина

Контейнери-адаптери в STL — це спеціальні шаблонні класи, які забезпечують специфічний інтерфейс доступу до елементів. Вони побудовані на основі інших контейнерів (наприклад, deque, vector) і надають обмежений доступ до даних.

Основні контейнери-адаптери:

- stack — реалізує структуру даних "стек" (LIFO).
- queue — реалізує структуру "черга" (FIFO).
- priority_queue — черга з пріоритетом, де першим витягується елемент з найвищим пріоритетом.

Ітератори — це об'єкти, які дозволяють послідовно проходити елементи контейнерів STL. Вони подібні до покажчиків та підтримують операції інкременту, розіменування тощо.

Приклад використання:

```
#include <iostream>
#include <stack>
#include <queue>
#include <vector>
#include <string>

int main() {
    std::stack<std::string> bookStack;
    bookStack.push("Book A");
    bookStack.push("Book B");
    bookStack.push("Book C");

    std::cout << "Stack top: " << bookStack.top() << std::endl;
    bookStack.pop();
     std::cout << "After pop, new top: " << bookStack.top() << 
std::endl;

    std::queue<int> taskQueue;
```

```
        taskQueue.push(10);
        taskQueue.push(20);
        taskQueue.push(30);

           std::cout << "Queue front: " << taskQueue.front() <<
std::endl;
        taskQueue.pop();
         std::cout << "After pop, new front: " << taskQueue.front()
<< std::endl;

        std::priority_queue<int> priority;
        priority.push(5);
        priority.push(15);
        priority.push(1);

        std::cout << "Top of priority queue: " << priority.top() <<
std::endl;

        return 0;
    }
```

У цьому прикладі спочатку створюється об'єкт стеку bookStack, який зберігатиме рядки (назви книг). За допомогою функції push() до стеку додаються значення "Book A", "Book B" і "Book C". Оскільки стек працює за принципом LIFO (останній увійшов — перший вийшов), на вершині стеку буде "Book C", що і демонструється при виведенні за допомогою функції top(). Потім елемент "Book C" видаляється за допомогою функції pop(), і новим верхнім елементом стає "Book B".

Далі створюється об'єкт черги taskQueue, який зберігає цілі числа. За допомогою функції push() до черги додаються значення 10, 20 і 30. Оскільки черга працює за принципом FIFO (перший увійшов — перший вийшов), першим елементом буде 10, що виводиться на екран за допомогою функції front(). Потім цей елемент видаляється за допомогою функції pop(), і новим першим елементом стає 20. Після цього створюється об'єкт пріоритетної черги priority, яка також зберігає цілі числа. До неї додаються значення 5, 15 і 1 за допомогою функції push(). Пріоритетна черга автоматично впорядковує елементи так, щоб найбільший завжди був першим. Тому при виклику top() буде виведене значення 15, як найбільший елемент серед усіх у черзі.

Вивід програми буде наступним:

```
Stack top: Book C
After pop, new top: Book B
```

```
Queue front: 10
After pop, new front: 20
Top of priority queue: 15
```

Set — це контейнер стандартної бібліотеки шаблонів (STL), який зберігає унікальні елементи у відсортованому порядку. Кожен елемент у set з'являється лише один раз, і всі елементи автоматично впорядковуються згідно з оператором <.

Контейнер реалізовано на основі самобалансного бінарного дерева пошуку (зазвичай — червоно-чорного дерева), що забезпечує ефективність основних операцій.

Принцип роботи:

- При додаванні елементів set автоматично перевіряє їх унікальність: якщо елемент уже є — він не додається.
- Елементи автоматично сортуються у зростаючому порядку (можна задати власний порядок через компаратор).
- Швидкий пошук, вставка і видалення — кожна з цих операцій виконується за O(log n) часу.
- Доступ до елементів відбувається через ітератори — індексація типу set[i] недоступна.
- Оскільки значення елементів впливають на їх розташування, змінювати значення вже вставленого елемента заборонено (це може порушити структуру дерева).

Multiset — це контейнер стандартної бібліотеки шаблонів (STL), який зберігає елементи у відсортованому порядку, але, на відміну від set, дозволяє дублікати. Тобто в multiset один і той самий елемент може з'являтися кілька разів.

Контейнер реалізовано на основі самобалансного бінарного дерева пошуку, аналогічно до set.

Принцип роботи:

- Елементи автоматично впорядковуються за зростанням (можна використовувати власний компаратор).

- Дублікати дозволені: при додаванні елементів не виконується перевірка на унікальність.
- Підтримуються операції вставки, видалення та пошуку з логарифмічною складністю — O(log n).
- Доступ до елементів здійснюється через ітератори, індексація ([]) недоступна.
- Пошук конкретного значення (find), підрахунок кількості входжень (count), або отримання діапазону однакових значень (equal_range) — часті типові операції.

Map — це асоціативний контейнер стандартної бібліотеки шаблонів (STL), який зберігає пари ключ–значення. Усі ключі є унікальними та відсортованими відповідно до оператора < (або іншого компаратора, заданого користувачем). Кожен ключ пов’язаний лише з одним значенням.

Принцип роботи

- Реалізований на основі самобалансного бінарного дерева пошуку.
- Під час вставки перевіряється, чи вже існує ключ.
- Елементи автоматично сортуються за ключем.
- Швидкі операції пошуку, вставки та видалення — з логарифмічною складністю O(log n).
- Доступ до значення здійснюється через map[key] або ітератори.
- map[key] автоматично створює пару, якщо ключа ще не існує.

Multimap — це асоціативний контейнер STL, аналогічний до map, але з однією важливою відмінністю: дозволяє зберігати кілька елементів з однаковим ключем.

Принцип роботи:

- Також реалізований як самобалансне бінарне дерево пошуку.
- Кожен ключ може мати декілька значень — дублікати ключів допускаються.
- Елементи автоматично впорядковуються за ключем.
- Оскільки ключі не є унікальними, доступ до елементів здійснюється лише через ітератори — оператор [] не підтримується.

- Для роботи з кількома значеннями за ключем використовують методи:
- equal_range(key) — повертає діапазон елементів з однаковим ключем.
- count(key) — кількість входжень ключа.
- find(key) — повертає перший елемент із заданим ключем.

Стек(stack) — це контейнер-адаптер у STL, який реалізує структуру стек — "останній увійшов — перший вийшов" (LIFO, Last-In, First-Out). Доступ можливий лише до верхнього елемента. Контейнер stack базується на іншому контейнері (за замовчуванням — deque, але також можна використовувати vector або list).

Принцип роботи:

- Елемент додається тільки в верхню частину стека (через push).
- Вилучення можливе лише з верхньої частини (через pop).
- Перегляд верхнього елемента здійснюється через top() — без його видалення.
- Немає прямого доступу до інших елементів: не підтримує ітерацію, індексацію чи пошук.

Черга(queue) — це контейнер-адаптер стандартної бібліотеки шаблонів (STL), який реалізує структуру черга — "перший увійшов — перший вийшов" (FIFO, First-In, First-Out). Доступ можливий лише до першого (фронтального) і останнього (тильного) елементів. Як і stack, контейнер queue працює на базі іншого контейнера — за замовчуванням це deque.

Принцип роботи:

- Елементи додаються в кінець черги (push() або emplace()).
- Вилучення елементів здійснюється лише з початку черги (pop()).
- Огляд першого та останнього елементів можливий через front() і back().
- Немає підтримки індексації чи довільного доступу до елементів.

Черга з пріорітетом(priority_queue) — це контейнер-адаптер стандартної бібліотеки шаблонів (STL), який реалізує чергу з пріоритетами. На відміну від звичайної queue, елементи в priority_queue виходять не у порядку додавання, а відповідно до свого пріоритету: першим виходить елемент з найвищим

пріоритетом (тобто найбільший за значенням за замовчуванням). За замовчуванням реалізована як макс-купа на базі контейнера vector з використанням функцій make_heap, push_heap, pop_heap.

Принцип роботи:

- Елементи додаються за допомогою методів push() або emplace().
- Доступ до елемента з найвищим пріоритетом здійснюється через top().
- Видалення елемента з найвищим пріоритетом — через pop().
- Прямого доступу до інших елементів, як і у queue, немає.
- За замовчуванням пріоритет вищий у більшого значення (тобто 5 має більший пріоритет, ніж 2).

Приклад використання:

```
#include <iostream>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <string>

int main() {
      // SET - зберігає унікальні елементи у відсортованому порядку
      std::set<int> mySet = {3, 1, 4, 1, 5, 9};
      std::cout << "Set (унікальні, відсортовані): ";
      for (int num : mySet)
          std::cout << num << " ";
      std::cout << "\n";

      // MULTISET - дозволяє дублікати, також відсортований
      std::multiset<int> myMultiSet = {3, 1, 4, 1, 5, 9};
      std::cout << "Multiset (можуть бути дублікати): ";
      for (int num : myMultiSet)
          std::cout << num << " ";
      std::cout << "\n";

      // MAP - пари ключ-значення, унікальні ключі
      std::map<std::string, int> myMap = {
          {"apple", 2},
          {"banana", 4},
          {"cherry", 3}
      };
```

```
        std::cout << "Map (ключ-значення):\n";
        for (const auto& pair : myMap)
            std::cout << pair.first << ": " << pair.second << "\n";

        // MULTIMAP - дозволяє кілька пар з однаковими ключами
        std::multimap<std::string, int> myMultiMap;
        myMultiMap.insert({"apple", 2});
        myMultiMap.insert({"apple", 5});
        std::cout << "Multimap (повторювані ключі):\n";
        for (const auto& pair : myMultiMap)
            std::cout << pair.first << ": " << pair.second << "\n";

          // STACK - структура LIFO (останній прийшов – перший
вийшов)
        std::stack<std::string> myStack;
        myStack.push("first");
        myStack.push("second");
        myStack.push("third");
        std::cout << "Stack (LIFO): ";
        while (!myStack.empty()) {
            std::cout << myStack.top() << " ";
            myStack.pop();
        }
        std::cout << "\n";

        // QUEUE - структура FIFO (перший прийшов – перший вийшов)
        std::queue<std::string> myQueue;
        myQueue.push("first");
        myQueue.push("second");
        myQueue.push("third");
        std::cout << "Queue (FIFO): ";
        while (!myQueue.empty()) {
            std::cout << myQueue.front() << " ";
            myQueue.pop();
        }
        std::cout << "\n";

        // PRIORITY_QUEUE - максимальний елемент завжди на початку
        std::priority_queue<int> myPQ;
        myPQ.push(4);
        myPQ.push(1);
        myPQ.push(7);
        myPQ.push(3);
        std::cout << "Priority Queue (max-heap): ";
        while (!myPQ.empty()) {
            std::cout << myPQ.top() << " ";
            myPQ.pop();
```

```
    }
    std::cout << "\n";

    return 0;
}
```

Цей приклад демонструє використання основних контейнерів STL у C++, зокрема `set`, `multiset`, `map`, `multimap`, `stack`, `queue` та `priority_queue`. Вони слугують для зберігання й ефективного управління даними: множини для зберігання унікальних або повторюваних значень, асоціативні масиви для пар ключ-значення, а також структури даних типу стек (LIFO), черга (FIFO) та черга з пріоритетом (max-heap). Кожен контейнер має специфічну поведінку й оптимізований для певних операцій, що робить їх зручними для широкого кола задач у програмуванні.

Використання контейнерних класів дозволяє забезпечити ефективне управління даними та спростити роботу з колекціями в мові програмування C++.

Завдання

Варіант 1

Розширте попередню систему(ЛР7) керування магазином електроніки, використовуючи асоціативні (set, map, multiset, multimap) та адаптивні (stack, queue, priority_queue) контейнери STL. Реалізуйте такі функції:

- Зберігайте унікальні назви продуктів у set або map.
- Дозвольте зберігати товари з однаковою назвою, але різними параметрами у multiset або multimap.
- Створіть чергу нових поставок з queue або priority_queue, де терміновим замовленням надається вищий пріоритет.
- Імітуйте додавання та обробку постачань (через stack, queue, priority_queue).
- Виведіть інформацію про всі товари в системі.

Додатково: поясніть призначення кожного використаного контейнера та чому він підходить для даної задачі.

Варіант 2

Розширте попередню систему(ЛР7) управління бібліотекою, використовуючи асоціативні та адаптивні контейнери STL.

Реалізуйте:

- Збереження унікальних ISBN книг у `set` або `map` для швидкого доступу.
- Збереження книг з однаковими авторами або роком видання у `multimap`.
- Використання `stack` або `queue` для моделювання черги користувачів, що хочуть взяти книгу.
- Використання `priority_queue` для організації пріоритетного списку замовлень (наприклад, вчителям чи дослідникам — вищий пріоритет).
- Пошук і вивід книг за автором або роком видання з використанням асоціативних контейнерів.

Додатково: поясніть роль кожного контейнера у вашій реалізації та обґрунтуйте вибір.

Варіант 3

Розширте попередню систему(ЛР7) управління завданнями, використовуючи асоціативні та адаптивні контейнери STL.

Реалізуйте:

- Збереження завдань у `map`, де ключ — це дедлайн, а значення — завдання.
- Збереження завдань з однаковими дедлайнами в `multimap`.
- Збереження унікальних назв завдань у `set` або `multiset`.
- Моделювання черги термінових завдань за допомогою `queue` або `priority_queue` (де пріоритет визначається наближенням дедлайну).
- Створення історії виконаних завдань за допомогою `stack` (останнє виконане — перше в списку).

Додатково:

Реалізуйте метод пошуку найближчого дедлайну або пріоритетного

завдання. Поясніть вибір кожного контейнера та переваги його використання для конкретних операцій.

Варіант 4

Розширте попередню систему(ЛР7) управління складом магазину автозапчастин, використовуючи асоціативні та адаптивні контейнери STL.
Реалізуйте:

- Збереження запчастин у `map`, де ключ — це назва запчастини, а значення — структура з виробником, ціною та кількістю.
- Збереження дубльованих запчастин у `multimap`, якщо в магазині можуть бути однакові запчастини від різних виробників.
- Збереження унікальних назв запчастин у `set` або `multiset`.
- Моделювання черги постачання запчастин за допомогою `queue`.
- Використання `priority_queue` для черги запчастин з найменшою кількістю на складі (пріоритетне поповнення).
- Використання `stack` для історії змін даних про запчастини.

Додатково:

Реалізуйте пошук запчастин за виробником або фільтрацію за ціною. Поясніть, як контейнери полегшують організацію даних і які з них доцільні для конкретних задач.

Варіант 5

Реалізуйте попередню систему(ЛР7) черги обслуговування клієнтів з використанням контейнерів queue, priority_queue, map, set.
Функціонал:

- Використовуйте queue<Call> для зберігання дзвінків у порядку надходження.
- Додайте декілька дзвінків.
- Виведіть усю чергу дзвінків.
- Обробіть кілька дзвінків (видалення з черги).

- Виведіть оновлену чергу.

Розширення:

- Використовуйте priority_queue для пріоритетних дзвінків (наприклад, VIP-клієнтів).
- Застосуйте map<string, vector<Call>> для групування дзвінків за темами.
- Застосуйте set<string> для зберігання унікальних імен клієнтів.

Варіант 6

Реалізуйте попередню систему(ЛР7) керування запасами продуктів у супермаркеті (асоціативні контейнери та адаптери)

Функціонал:

- Використовуйте map<string, Product> або unordered_map для зберігання продуктів за назвою.
- Додайте продукти до map, де ключ — назва, значення — структура Product.
- Виведіть список продуктів.
- Видаліть продукт(и) за назвою (через map::erase).
- Оновіть інформацію про продукт за ключем (назвою).

Розширення:

- Використайте multimap<string, Product> для групування продуктів за категорією.
- Створіть set<string> для зберігання унікальних категорій.
- За допомогою priority_queue реалізуйте чергу на поповнення складу (товари з найменшою кількістю мають більший пріоритет).

Варіант 7

Реалізуйте попередню систему(ЛР7) керування готелями — асоціативне зберігання та пріоритети

Функціонал:

- Використовуйте map<int, Room> — ключ: номер кімнати, значення: структура Room.

- Додайте кімнати до map.
- Виведіть список усіх кімнат, фільтруючи доступні.
- Створіть multimap<int, Reservation> для збереження декількох резервацій на одну кімнату.
- Додайте кілька резервацій до multimap.
- Видаліть резервацію за номером кімнати.
- Розширення:
    - Використайте set<int> для відстеження зайнятих номерів.
    - Реалізуйте чергу (priority_queue) для обробки бронювань — кімнати з найнижчою ціною мають найвищий пріоритет.
    - Застосуйте stack для історії скасованих резервацій або дій клієнта.
    - Створіть чергу (queue) очікування, якщо всі кімнати зайняті.

Варіант 8

Реалізуйте попередню систему(ЛР7) керування поштовими відправленнями - асоціативне зберігання та черговість

Функціонал:

- Використовуйте map<int, Parcel> для зберігання відправлень з ключем — номер відправлення.
- Додайте кілька відправлень до map.
- Виведіть весь список, перебравши map.
- Видаліть відправлення за номером.
- Оновіть дані відправлення.
- Застосуйте multimap<string, Parcel> для групування відправлень за типом.
- Застосуйте priority_queue для обробки термінових відправлень (пріоритет за вагою або типом).
- Застосуйте queue — загальна черга обробки відправлень.
- Використовуйте stack — історія змін або видалених відправлень.
- Розширення:
    - Виведення найважчого або найтерміновішого відправлення.

- Пошук усіх відправлень певного одержувача або відправника (multimap або set).

Варіант 9

Реалізуйте попередню систему(ЛР7) керування студентськими оцінками — асоціативні структури та додаткові можливості

Функціонал:

- Використовуйте map<pair<string, string>, vector<int>> — ключ: (ім’я, прізвище), значення: список оцінок.
- Додайте кількох студентів із початковими оцінками до map.
- Виведіть повний список студентів та їх оцінок.
- Видаліть студента за ім’ям і прізвищем.
- Додайте або видаліть оцінку студенту.
- Додаткові можливості з іншими контейнерами:
    - Застосуйте multimap<string, int> для групування оцінок за предметами.
    - set<int> — унікальні оцінки студента.
    - priority_queue<pair<float, string>> — черга студентів за рейтингом (середнім балом).
    - stack — історія змін оцінок (undo).
    - queue — черга на перегляд оцінок.

Варіант 10

Реалізуйте попередню систему(ЛР7) замовленням — асоціативна логіка і обробка пріоритетів

Функціонал:

- Використайте map<string, pair<int, double>> — ключ: назва страви, значення: (кількість, ціна за одиницю).

- Додайте кілька страв до map, обчисліть загальну суму.
- Видаліть певну страву.
- Додайте нові страви.
- Виведіть оновлену суму.
- Додаткові можливості:
    - multimap<string, double> — зберігання замовлень з повтореннями страв.
    - priority_queue<pair<double, string>> — вивід найдорожчих страв у замовленні.
    - set<string> — збереження унікальних назв страв.
    - stack — історія змін замовлення (наприклад, скасування останніх дій).
    - queue — черга замовлень, що очікують приготування.

Список використаної літератури

1. B. Stroustrup: The C++ Programming Language (Fourth Edition). May 2013. Addison Wesley. Reading Mass. USA. May 2013. ISBN 0-321-56384-0. 1360 pages. Softcover, hardcover, and electronic versions.
2. Ira Pohl: Object-oriented programming using C++. 1997. Addison-Wesley. ISBN 978-0201895506. 543 pages.
3. B. Stroustrup: Programming -- Principles and Practice Using C++. December 2008. Addison-Wesley. ISBN 978-0321543721. 1264 pages. Softcover.
4. B. Stroustrup: Programming -- Principles and Practice Using C++ (Second Edition). May 2014. Addison-Wesley. ISBN 978-0321992789. 1312 pages. Softcover and electronic versions.
5. B. Stroustrup: A Tour of C++ (Second Edition). July 2018. Addison-Wesley. ISBN 978-0-13-499783-4. 240 pages. Softcover and electronic versions.
6. Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, Kelli A. Houston / Object-Oriented Analysis and Design with Applications (3rd Edition) - 2007/ 720 p. ISBN 978-5-8459-1401-9, 0-201-89551-X
7. Grady Booch James Rumbaugh Ivar Jacobson Б90 Язык UML. Руководство пользователя. 3-е изд.: Пер. с англ. Мухин Н. – М.: ДМК Пресс, 2016. – 496 с.: ил.
8. Rainer Grimm. C++ Core Guidelines. Addison-Welsey Professional. 2022. 403 c.
9. Bill Weinman. C++20 STL Cookbook. Packt Publishing. 2022. 450 c.
10. Ткачук В. М. Програмування на С++ : Лабораторний практикум / В. М. Ткачук. – Івано-Франківськ : Видавництво Прикарпатського національного університету імені Василя Стефаника, 2011. – 160 с.
11. Жуковський С.С., Вакалюк Т.А. Програмування мовою С++. Структурне програмування (лабораторний практикум). Навчальний посібник для студентів фізико-математичного факультету. – Житомир: Вид-во ЖДУ, 2011. – 92 с. (видання друге, перероблене та доповнене).
12. Грицюк Ю.І., Рак Т.Є. Г 85 Об'єктно-орієнтоване програмування мовою С++ : навчальний посібник. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 404 с. –

Статистика: іл. 18, табл. 12, бібліогр. 34. ISBN 978-966-3466-86-3BN 978-966-3466-86-3

13. Adrian Ostrowski, Piotr Gaczkowski. Software Architecture with C++. Packt Publishing. 2021. 522 с.

Інформаційні ресурси

14. Он-лайн ресурс LEARN C++. – URL: https://www.learncpp.com/

15. Он-лайн ресурс aCode .– URL: https://acode.com.ua/