Тема
Множинне наслідування. Принцип поліморфізму. Віртуальні функції та абстрактні класи.

Мета
Отримати практичні навички використання множинного наслідування, освоїти способи вирішення проблеми неоднозначності при множинному наслідуванні.

Навчитись створювати абстрактні класи, колекції об’єктів базового типу, що включають об’єкти похідних типів. Навчитись використовувати чисті віртуальні функції.

Теоретичні відомості
Множинне наслідування
Множинне наслідування (multiple inheritance) - це можливість використовувати кілька базових класів для класів нащадків. Це означає, що клас може успадковувати властивості і методи з декількох батьківських класів одночасно.

Уявімо ситуацію, що ми створюємо програму, в якій ми працюємо з класами тварин. Один з цих класів Bird (птах), а другий Horse (кінь). Клас Bird містить функцію Fly(), а клас Horse містить функцію Gallop() – біг галопом.

А тепер сталось так, що нам потрібно створити новий міфічний персонаж – крилатого Пегаса (Pegasus), який буде гібридом між птахом та конем. В таких випадках використовують множинне наслідування. Код буде виглядати наступним чином:

```
#include <iostream>
#include <array>
#include <memory>

class Horse
{
public:
    virtual void Gallop() { std::cout << "Gallop!\n"; }
    virtual ~Horse() {}
};

class Bird
{
public:
    virtual void Fly() { std::cout << "Fly!\n"; }
    virtual ~Bird() {}
};

class Pegasus : public Horse, public Bird
{
public:
    void Chirp() { std::cout << "Whinny!\n"; }
};
```

Рис. 1. Множинне наслідування

Використовуючи множинне наслідування ми також вирішуємо проблему попадання Пегаса в два списки – Птахів та Коней:

```
int main()
{
    const int horsesNumber = 5;
    std::array<std::unique_ptr<Horse>, horsesNumber> ranch;
    for (int i = 0; i < horsesNumber; ++i)
```

```
    {
        ranch[i] = std::make_unique<Pegasus>();
    }

    const int birdsNumber = 5;
    std::array<std::unique_ptr<Bird>, birdsNumber> aviary;
    for (int i = 0; i < birdsNumber; ++i)
    {
        aviary[i] = std::make_unique<Pegasus>();
    }

    return 0;
}
```

Рис. 2. Приклад роботи із похідним класом через вказівники на різні базові класи.

Тепер ми зможемо пройтись по списку птахів та в кожного елемента списку викликати метод Fly() а в списку ссавців відповідно метод Gallop(). Тут ми і використовуємо принцип поліморфізму.

Для того щоб викликати функцію з класу Pegasus нам потрібно привести вказівник базового типу до похідного типу. Для цього використовуємо функцію dynamic_cast. Якщо вказівник не вдалось привести до похідного типу функція верне нуль.

```
for (int i = 0; i < birdsNumber; ++i)
{
    Pegasus* peg = dynamic_cast<Pegasus*>(aviary[i].get());
    if (peg)
    {
        peg->Chirp();
    }
}
```

Рис. 3. Використання оператора dynamic_cast для приведення до похідного типу

Проблеми множинного наслідування

Можлива наступна ситуація – клас наслідує батьківський декілька разів. Способи вирішення:

1. Явно звертатись до членів потрібного нам батьківського класу:

```
class Animal
{
private:
    int age{0};

public:
    int GetAge() const { return age; }
};

class Horse : public Animal {};
class Bird : public Animal {};
class Pegasus : public Horse, public Bird {};
```

Рис. 4. Наслідування класу Animal декілька разів

При такому виклику:

```
int main()
{
    Pegasus* peg = new Pegasus;
    peg->GetAge();
}
```

Рис. 5. Неоднозначність при виклику функції GetAge

Ми отримаємо помилку компіляції: "Pegasus::GetAge" is ambiguous.

Щоб це обійти можна явно вказати з якого класу викликаємо функцію:

```
int main()
{
    Pegasus* peg = new Pegasus;
    peg->Horse::GetAge();
}
```

Рис. 6. Розв’язування неоднозначності виклику функції GetAge за допомогою вказування області видимості одного з базових класів

2. Використання віртуального наслідування.

Якщо віртуально наслідувати клас Animal – компілятор буде знати, що ми наслідуємо лише один клас в двох випадках:

```
#include <iostream>

class Animal
{
private:
    int age{0};

public:
    int GetAge() const { return age; }
};

class Horse : public virtual Animal
{
public:
    virtual void Gallop() { std::cout << "Gallop!\n"; }
};

class Bird : public virtual Animal
{
public:
    virtual void Fly() { std::cout << "Fly!\n"; }
};

class Pegasus : public Horse, public Bird
{
public:
    void Chirp() { std::cout << "Whinny!\n"; }
};
```

Рис. 7. Використання віртуального наслідування

Тому виклик методу таким чином при віртуальному наслідуванні допускається:

```
int main()
{
    Pegasus* peg = new Pegasus;
    peg->GetAge();
}
```

Рис. 8. Виклик функції GetAge тепер однозначний

3. Заміщення функції з базового класу

Заміщення функції вирішує дві проблеми:

- Зникає невизначеність звертання до базових класів;

- Функцію можна замістити таким чином, що в похідному класі при виклику цієї функції викликати функцію з базового класу.

Поліморфізм

Поліморфізм – одна з трьох основних парадигм ООП. Якщо говорити коротко, поліморфізм – це здатність об'єкта використовувати методи похідного класу, про існування якого базовий клас не знає.

Поліморфізм - це властивість об'єктів різних типів відображати специфічну для них поведінку, при цьому використовується загальний інтерфейс, що дозволяє взаємодіяти з цими об'єктами у єдиному контексті. Це означає, що можна працювати із об'єктами різних типів використовуючи базовий тип, що дозволяє підвищувати гнучкість і повторне використання коду.

У контексті C++ поліморфізм досягається за допомогою віртуальних функцій і наслідування, що дозволяє об'єктам похідних класів перевизначати методи базового класу, які є віртуальними. Коли ці методи викликаються на об'єкті базового класу, викликається відповідний метод на об'єкті похідного класу, що дозволяє досягти поліморфізму.

Віртуальні функції повинні мати ключове слово virtual в оголошенні, яке означає, що функція є віртуальною. Клас, що містить віртуальну функцію, називається поліморфним класом. У похідних класах віртуальна функція може бути перевизначена. Починаючи з C++11 перевизначені функції варто позначати ключовим словом override. Це робить код більш зрозумілим та допомагає виявити помилки хибного перевизначення.

Нижче показано приклад використання віртуальних функцій:

```cpp
#include <iostream>

class Shape
{
public:
    virtual void draw() const { std::cout << "Drawing a shape.\n"; }
    virtual ~Shape() = default;
};

class Rectangle : public Shape
{
public:
    void draw() const override { std::cout << "Drawing a rectangle.\n"; }
};

class Circle : public Shape
{
public:
    void draw() const override { std::cout << "Drawing a circle.\n"; }
};

int main()
{
    Shape* shape = new Shape();
    Shape* rectangle = new Rectangle();
    Shape* circle = new Circle();

    shape->draw();
    rectangle->draw();
    circle->draw();

    delete shape;
    delete rectangle;
    delete circle;

    return 0;
}
```

Рис. 9. Приклад використання віртуальних фукнцій

Вивід програми буде наступним:

```
Drawing a shape.
Drawing a rectangle.
Drawing a circle.
```

Рис. 10. Вивід програми із кодом поданим на Рис. 9

Зверніть увагу на використання віртуального деструктора для базового класу. Він є необхідним для забезпечення правильного видалення об'єктів, створених з посиланням на базовий клас, якщо такі об'єкти насправді є екземплярами похідних класів. Якщо деструктор базового класу не є віртуальним, то може виникнути проблема, коли об'єкт похідного класу не буде повністю видалений. Це може призвести до витоку пам'яті та інших непередбачуваних наслідків. Якщо ж деструктор базового класу віртуальний, то при видаленні об'єкта з посиланням на базовий клас буде викликаний деструктор похідного класу, і тим самим забезпечиться повне видалення об'єкта з пам'яті.

Абстрактні типи даних

В ООП часто створюються ієрархії логічно пов’язаних класів. Наприклад уявимо клас Shape, від якого наслідуються класи Rectangle і Circle. Пізніше від класу Rectangle наслідується клас Square, як окремий вид прямокутників. В класі Shape немає змісту визначати функції знаходження площі чи знаходження периметру, оскільки в похідних класах вони будуть перевизначені. Всі методи повинні функціонувати нормально в похідних класах а не в базовому класі Shape, оскільки неможливо створити екземпляр форми як такий. Також програма повинна бути захищена від спроби користувача створити об’єкт цього класу.

C++ підтримує створення абстрактних типів даних з чистими віртуальними функціями. Чисті віртуальні функції це такі, які ініціалізуються нульовим значенням, наприклад

```
virtual void Draw() = 0;
```

Рис. 11. Оголошення чисто віртуальної функції

Клас, який містить хоча б одну чисто віртуальну функцію є абстрактним. Поміщення в клас чистої віртуальної функції означає наступне:

- неможливо створити об’єкт цього класу;
- необхідно замістити чисту віртуальну функцію в похідному класі, або ж він також буде абстрактним.

Будь-який клас, наслідуваний від абстрактного класу, наслідує від нього чисту віртуальну функцію, яку необхідно замістити щоб отримати можливість створювати об’єкти цього класу. Якщо її не замістити, то похідний клас також буде абстрактним і може бути використаний для наслідування.

Визначення чистих віртуальних функцій

Зазвичай чисті віртуальні функції оголошуються в абстрактному базовому класі і не визначаються. Оскільки неможливо створити об’єкт абстрактного базового класу, як правило, нема необхідності у визначенні чисто віртуальної функції.

Тим не менше, така можливість існує. Визначення чисто віртуальної функції може бути присутнє, а похідний клас може викликати її явно, вказуючи тип базового класу.

При тому базовий клас все одно буде абстрактним, об’єктів такого класу неможливо створити.

Наступний код показує використання чистих віртуальних функцій, що містять реалізацію:

```cpp
#include <iostream>

class Shape
{
public:
    virtual void Draw() = 0;
    virtual ~Shape() = default;
};

void Shape::Draw() { std::cout << "Shape draw\n"; }

class Rectangle : public Shape
{
public:
    virtual void Draw()
    {
        Shape::Draw();
        std::cout << "Rectangle draw\n";
    }
};

int main()
{
    Shape* shape = new Rectangle();
    shape->Draw();
    delete shape;

    return 0;
}
```

Рис. 12. Приклад визначення та виклику чисто віртуальної функції

Вивід програми буде наступний:

```
Shape draw
Rectangle draw
```

Рис. 13. Вивід програми, код якої подано на Рис. 12.

Інтерфейси

У контексті C++, інтерфейс - це абстрактний клас, який містить лише чисто віртуальні функції та деструктор. Інтерфейс визначає набір функцій, які мають бути реалізовані в похідних класах. Класи, які успадковують інтерфейс, повинні реалізувати всі чисто віртуальні функції, або самі бути абстрактними класами.

Інтерфейси ширико використовуються у C++ для встановлення контрактів (залежностей) між класами. Таким чином логіка розділяється на окремі частини, знижуючи складність коду та полегшуючи тестування.

Наприклад, визначимо інтерфейс Shape для представлення геометричних фігур:

```cpp
class Shape
{
public:
    virtual ~Shape() = default;
    virtual double GetArea() const = 0;
    virtual double GetPerimeter() const = 0;
};
```

Рис. 14. Приклад оголошення інтерфейсу

У цьому прикладі Shape є абстрактним класом, оскільки містить чисто віртуальні функції GetArea() та GetPerimeter(). Клас, що успадковується від Shape, повинен реалізувати ці функції.

Наприклад, реалізуємо клас Circle, який успадковується від Shape:

```
static constexpr auto M_PI = 3.14159265358979323846;

class Circle : public Shape
{
public:
    explicit Circle(double r) : radius(r) {}
    double GetArea() const override { return M_PI * radius * radius; }
    double GetPerimeter() const override { return 2 * M_PI * radius; }
private:
    double radius;
};
```

Рис. 15. Клас Circle реалізує інтерфейс Shape

У цьому прикладі Circle реалізує обидві функції GetArea() та GetPerimeter() успадковані від Shape. Це означає, що ми можемо використовувати об'єкти Circle так, як ми використовували б об'єкти Shape. Наприклад:

```
int main()
{
    Shape* s = new Circle(5.0);
    std::cout << "Area: " << s->GetArea() << '\n';
    std::cout << "Perimeter: " << s->GetPerimeter() << '\n';
    delete s;

    return 0;
}
```

Рис. 16. Приклад роботи із похідним класом через інтерфейс

Завдання

1. Внести зміни в ієрархію класів із лабораторної роботи №6, зробивши класи поліморфними.
2. Базовий абстрактний клас має мати віртуальний деструктор та хоча б одне визначення для чисто віртуальної функції.
3. Створити масиви об’єктів базового типу, в них помістити об’єкти похідного типу. Продемонструвати виклик функцій з об’єктів – елементів масиву. Використати оператор dynamic_cast.
4. Використати множинне наслідування (дозволяється додатково створити формальний абстрактний клас), продемонструвати вирішення проблеми з неоднозначністю доступу до членів базових класів за допомогою віртуального наслідування, за допомогою явного звертання до членів класу та за допомогою заміщення функцій в похідному класі.
5. Для роботи із динамічною пам’яттю використовувати std::unique_ptr.
6. Продемонструвати роботу класів таким чином (на вибір):
    a. За допомогою тестів Google Test, які перевірятимуть роботу ієрархії класів.
    b. За допомогою інтерактивної програми із демонстрацією її роботи.
7. Сформувати звіт до лабораторної роботи. Відобразити в ньому діаграму наслідування класів.

Варіанти завдань

Варіант	Завдання
1	Клас CLoan має містити чисту віртуальну функцію (повертає суму виплат у заданому місяці, 1 – січень, 2 – лютий, і тд.): `virtual double GetPaymentAmount(unsigned short usMonthNumber) const = 0;` Класи CEqualPaymentLoan, CPercentPaymentLoan та CConcessionalLoan мають реалізувати цю функцію відповідно до своїх правил обчислення. Написати функцію, яка отримуватиме масив вказівників на базовий клас – CLoan – та, викликаючи GetPaymentAmount() обчислюватиме загальну суму сплати для всіх об'єктів.

2	Клас CBankAccount має містити чисто віртуальну функцію (здійснює транзакцію заданого типу (комунальний платіж, переказ чи зняття готівки) на задану суму. Вертає true, якщо вдалося здійснити, і false, якщо суми на рахунку чи кредитного ліміту не вистачає):
	
	`enum class TransactionType { Utility, Transfer, Withdrawal };`
	
	`virtual bool CommitTransaction(TransactionType eType, double dAmount) = 0;`
	
	Класи CStandardBankAccount, CSocialBankAccount та CPremiumBankAccount мають реалізувати цю функцію відповідно до своїх правил (стан рахунку, комісія, наявність кредитного ліміту, тощо).
	
	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CBankAccount – та, викликаючи CommitTransaction(), здійснюватиме транзакцію. Після здійснення транзакції отримує стан рахунку (суму) та обчислюватиме загальну суму на всіх рахунках.

3	Клас CDeposit має містити чисто віртуальну функцію (повертає суму виплат після заданої кількості місяців): `virtual double GetPayoutAfterNMonths(unsigned int uiMonthCount) const = 0;` Класи CFixedDeposit, CAccumulativeDeposit та CVIPDeposit мають реалізувати цю функцію відповідно до своїх правил. Написати функцію, яка отримуватиме масив вказівників на базовий клас – CDeposit – та, викликаючи GetPayoutAfterNMonths () обчислюватиме загальну суму виплат через задану кількість місяців.
4	Клас CElectricityConsumption має містити чисто віртуальну функцію (отримує на вхід кількість спожитої електроенергії у кВт*год протягом кожної години доби, повертає суму нарахувань у грн): `virtual double GetElectricityBill(double dConsumptionPerHours[24U]) const = 0;` Класи CFixedElectricityConsumption, CSocialElectricityConsumption та CMultiZoneElectricityConsumption мають реалізувати цю функцію відповідно до своїх правил.

	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CElectricityConsumption – та, викликаючи GetElectricityBill(), обчислюватиме мінімальну та максимальну суму до оплати.

5	Клас CShape має містити чисто віртуальні функції:
	
	`virtual double GetArea() const = 0;`
	`virtual double GetPerimeter() const = 0;`
	
	Класи CRectangle, CCircle та CTriangle мають реалізувати ці функції відповідно до своїх правил.
	
	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CShape – та, викликаючи GetArea() та GetPerimeter(), обчислюватиме мінімальну та максимальну площу та мінімальний та максимальний периметр фігур.

6	Клас CSolidFigure має містити чисто віртуальні функції:
	
	`virtual double GetArea() const = 0;`
	`virtual double GetVolume() const = 0;`
	
	Класи CCylinder, CSphere та CCube мають реалізувати ці функції відповідно до своїх правил.
	
	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CSolidFigure – та, викликаючи GetArea() та GetVolume(), обчислюватиме мінімальну та максимальну площу та мінімальний та максимальний об'єм фігур.

7	Клас CAnimal має містити чисто віртуальну функцію (кількість калорій, необхідних для харчування тварини за один день): `virtual double GetFoodCaloriesPerDay() const = 0;` Класи CCat, CDog та CHorse мають реалізувати цю функцію відповідно до своїх правил. Написати функцію, яка отримуватиме масив вказівників на базовий клас – CAnimal – та, викликаючи GetFoodCaloriesPerDay(), обчислюватиме загальну суму калорій необхідних на харчування тварин протягом 30 днів.

8	Клас CPayment має містити чисто віртуальні функції:
	
	`virtual double GetFee() const = 0;`
	`virtual double GetAmount() const = 0;`
	`virtual double GetMaxAmount() const = 0;`
	
	Класи CCreditCardPayment, CPayPalPayment та CCashPayment мають реалізувати ці функції відповідно до своїх правил.
	
	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CPayment – та, викликаючи GetFee() та GetAmount(), обчислюватиме загальну суму та загальну суму комісії серед переданих оплат.

9	Клас CDoctor має містити чисто віртуальну функцію:
	
	`virtual void ExaminePatient(const CPatientInfo & cPatient,`
	`CProtocol & protocol, CDiagnosis & diagnosis) const = 0;`

	Де CPatientInfo -це клас, що містить інформацію про стан здоров'я пацієнта, CProtocol – клас, що містить протокол проведених досліджень (назва дослідження, результат), CDiagnosis – клас, що містить діагноз пацієнта. Класи CPediatrician, CTherapist та CDentist мають реалізувати цю функцію відповідно до своїх правил. Написати функцію, яка отримуватиме масив вказівників на базовий клас – CDoctor – та, викликаючи ExaminePatient(), отримуватиме інформацію про результати досліджень та діагнози та агрегуватиме їх. Результатом роботи функції має бути загальний діагноз.
10	Клас CParcel має містити чисто віртуальну функцію (отримання вартості відправлення при заданій відстані): `virtual double GetTransferValue(double dDistance) const = 0;` Класи CCommonParcel, CDeclaredValueParcel та CdigitalTransfer мають реалізувати цю функцію відповідно до своїх правил. Написати функцію, яка отримуватиме масив вказівників на базовий клас – CParcel – та, викликаючи GetTransferValue(), обчислюватиме загальну вартість для всіх відправлень.
11	Клас CEmployee має містити чисто віртуальну функцію (обчислення заробітної плати за певну кількість робочих годин): `virtual double CalculateSalaryForHours(double dWorkedHours) const = 0;` Класи CManagerEmployee, CSalesmanEmployee та CEngineerEmployee мають реалізувати цю функцію відповідно до своїх правил. Написати функцію, яка отримуватиме масив вказівників на базовий клас – CEmployee – та, викликаючи CalculateSalaryForHours(), обчислюватиме загальну суму виплат для всіх працівників.

12	Клас CFood має містити чисто віртуальні функції:
	
	`virtual double GetCaloriesForOneKg() const = 0;`
	`virtual double GetAmountForOnePersonPerDay(unsigned short usAge) const = 0;`
	
	Класи CChickenMeat, CMilk та CChickenEgg мають реалізувати ці функції відповідно до своїх правил.
	
	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CFood – та, викликаючи GetAmountForOnePersonPerDay(), обчислюватиме рекомендовану масу продукту в день для особи певного віку.

13	Клас CVehicle має містити чисто віртуальні функції:
	
	`virtual double GetDistancePerOneLiter(double dUsefulMassInKg) const = 0;`
	`virtual double GetServicePricePerThousandKm() const = 0;`
	
	Класи CCar, CTruck та CMotorcycle мають реалізувати ці функції відповідно до своїх правил.
	
	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CVehicle – та, викликаючи GetDistancePerOneLiter() та GetServicePricePerThousandKm(),

	обчислюватиме затрати на перевезення заданої маси на задану відстань. Функція має враховувати максимальну корисну масу, яку може перевозити транспортний засіб, і відповідно брати необхідну кількість траспортних засобів певного типу.
14	Клас CChessman має містити чисто віртуальну функцію (вертає true, якщо фігура може бити задану позицію, задану індексом від 0 до 63): `virtual bool CanCapturePosition(unsigned char ucPosition) const = 0;` Класи CKnight, CQueen та CRook мають реалізувати цю функцію відповідно до свого алгоритму. Написати функцію, яка отримуватиме індекс позиції та масив вказівників на базовий клас – CChessman – та, викликаючи CanCapturePosition(), перевірятиме, чи задана позиція може бути бита будь-якою фігурою із переданого масиву.

15	Клас CSolidFigure має містити чисто віртуальні функції:
	
	`virtual double GetArea() const = 0;`
	`virtual double GetVolume() const = 0;`
	
	Класи CRightPrism, CParallelepiped та CRegularTetrahedron мають реалізувати ці функції відповідно до своїх правил.
	
	Написати функцію, яка отримуватиме масив вказівників на базовий клас – CSolidFigure – та, викликаючи GetArea() та GetVolume(), обчислюватиме середню площу та середній об’єм фігур.