1. Виртуални функции

Дефиниция: Виртуалните функции са член-функции в базов клас, означени с ключовата дума `virtual`. Те позволяват на производните класове да предоставят своя собствена имплементация, като поддържат полиморфно поведение при извикване чрез указател или референция към базовия клас.

Ключови характеристики:

- **Динамично свързване:** Извикването на виртуална функция се разрешава по време на изпълнение, а не по време на компилация.
- **Ключова дума `override`:** Използва се в производния клас, за да се укаже, че функцията замества виртуална функция от базовия клас. Помага за предотвратяване на грешки, като несъответствие в сигнатурата.
- **Ключова дума `final`:** Може да се използва, за да се предотврати по-нататъшно заместване на функцията в производни класове.

Пример:

```
#include <iostream>
#include <vector>

class Animal {
public:
    virtual void speak() const { // Виртуална функция
        std::cout << "Animal: ..." << std::endl;
    }
    virtual void move() const {
        std::cout << "Animal moves..." << std::endl;
    }
    virtual ~Animal() = default; // Виртуален деструктор за правилно
освобождаване
};

class Dog : public Animal {
public:
    void speak() const override { // Замества speak()
        std::cout << "Dog: Woof!" << std::endl;
    }
    // move() се наследява от Animal
};

int main() {
    std::vector<Animal*> zoo = { new Animal(), new Dog() };
    for (const auto* animal : zoo) {
```

```
        animal->speak(); // Полиморфно извикване
        animal->move();
    }
    for (auto* animal : zoo) {
        delete animal; // Правилно освобождаване благодарение на виртуалния
деструктор
    }
    return 0;
}
```

Изход:

```
Animal: ...
Animal moves...
Dog: Woof!
Animal moves...
```

Забележки:

- Без `virtual`, извикването ще бъде статично и ще зависи от типа на указателя, а не от реалния тип на обекта.
- `override` гарантира, че функцията съответства на виртуална функция в базовия клас, като улеснява откриването на грешки по време на компилация.

---

2. Статично и динамично свързване

Статично свързване (Compile-time):

- Извикването на функция се определя по време на компилация.
- Използва се за невиртуални функции, свободни функции и шаблони.
- По-бързо, тъй като няма нужда от индирекция по време на изпълнение.
- Пример: Извикване на невиртуална функция чрез указател към базов клас ще извика функцията от базовия клас, независимо от реалния тип на обекта.

Динамично свързване (Run-time):

- Извикването се разрешава по време на изпълнение чрез виртуални функции.
- Използва се за виртуални функции, което позволява полиморфизъм.
- По-бавно поради индирекцията през виртуалната таблица (`vtable`).

Сравнение:

Тип	Кога се разрешава	Примери
Статично	Compile-time	Невиртуални методи, свободни функции
Динамично	Run-time	Виртуални методи

Пример за разлика:

```
#include <iostream>

class Base {
public:
    void nonVirtual() { std::cout << "Base::nonVirtual" << std::endl; }
    virtual void virtualFunc() { std::cout << "Base::virtualFunc" <<
std::endl; }
};

class Derived : public Base {
public:
    void nonVirtual() { std::cout << "Derived::nonVirtual" << std::endl; }
    void virtualFunc() override { std::cout << "Derived::virtualFunc" <<
std::endl; }
};

int main() {
    Base* ptr = new Derived();
    ptr->nonVirtual();  // Статично: Base::nonVirtual
    ptr->virtualFunc(); // Динамично: Derived::virtualFunc
    delete ptr;
    return 0;
}
```

---

3. Виртуални таблици (vtables)

Как работят:

- При създаването на клас с виртуални функции компилаторът генерира **виртуална таблица (vtable)** за този клас.
- Всеки обект от класа съдържа скрит указател (`vptr`), който сочи към съответната `vtable` .
- `vtable` е масив от указатели към виртуалните функции на класа.
- При извикване на виртуална функция (`obj->virtFunc()`):
  1. Чете се `vptr` от обекта.
  2. Намира се съответният указател към функцията в `vtable` .

3. Извиква се функцията.

Графично представяне:

```
Dog object:
+------------------+
| vptr ──> Dog_vtable |
| data members     |
+------------------+

Dog_vtable:
[0] Dog::speak    (замества Animal::speak)
[1] Animal::move  (наследена)
```

Забележки:

- `vtable` се създава на ниво клас, а не на обект, което пести памет.
- Производните класове имат собствени `vtable`, които съдържат указатели към техните версии на виртуалните функции или наследените функции от базовия клас.
- Разходът за памет и производителност е минимален, но съществува заради индирекцията.

---

4. Абстрактни класове и интерфейси

Абстрактен клас:

- Клас, който съдържа поне една **чисто виртуална функция** (обозначена с `= 0`).
- Не може да се инстанцира директно.
- Служи като основа за производни класове, които трябва да имплементират чисто виртуалните функции.

Интерфейс:

- Специален случай на абстрактен клас, при който всички функции са чисто виртуални и няма данни (или само минимални).
- Използва се за дефиниране на поведение, което производните класове трябва да следват.

Пример:

```cpp
#include <iostream>
#include <string>

class Logger {
```

```
public:
    virtual void log(const std::string& message) = 0; // Чисто виртуална функция
    virtual ~Logger() = default; // Виртуален деструктор за безопасно освобождаване
};

class FileLogger : public Logger {
public:
    void log(const std::string& message) override {
        std::cout << "Logging to file: " << message << std::endl;
    }
};

class ConsoleLogger : public Logger {
public:
    void log(const std::string& message) override {
        std::cout << "Logging to console: " << message << std::endl;
    }
};

int main() {
    Logger* logger = new FileLogger();
    logger->log("Error occurred!");
    delete logger;
    return 0;
}
```

Забележки:

- Виртуалният деструктор е задължителен в абстрактни класове, за да се осигури правилно освобождаване на производни обекти.
- Интерфейсите са често използвани в обектно-ориентирания дизайн за постигане на гъвкавост и модулност.

---

5. Полиморфизъм

Видове полиморфизъм:

5.1. Run-time полиморфизъм (Динамичен)

- Постига се чрез виртуални функции и наследяване.
- Позволява указател или референция към базов клас да извиква методи на производни класове в зависимост от реалния тип на обекта.
- Примери за приложение:

- Плъгини (напр. различни имплементации на един интерфейс).
- Графични елементи (напр. Shape, Button, Window).
- Обработка на събития (event handlers).

Пример:

```
#include <iostream>

class Animal {
public:
    virtual void speak() const = 0;
    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    void speak() const override { std::cout << "Woof!" << std::endl; }
};

class Cat : public Animal {
public:
    void speak() const override { std::cout << "Meow!" << std::endl; }
};

void process(const Animal* animal) {
    animal->speak();
}

int main() {
    process(new Dog()); // Woof!
    process(new Cat()); // Meow!
    return 0;
}
```

5.2. Compile-time полиморфизъм (Статичен)

- Постига се чрез:
  - **Function overloading:** Функции с еднакво име, но различни параметри.
  - **Templates:** Обобщено програмиране, което позволява работа с различни типове.
- Разрешава се по време на компилация, което го прави по-бързо от динамичния полиморфизъм.

Пример:

```cpp
#include <iostream>

template<typename T>
T add(const T& a, const T& b) {
    return a + b;
}

void print(int x) { std::cout << "Int: " << x << std::endl; }
void print(double x) { std::cout << "Double: " << x << std::endl; }

int main() {
    std::cout << add(3, 4) << std::endl;      // int: 7
    std::cout << add(1.2, 3.4) << std::endl;  // double: 4.6
    print(5);                                 // Int: 5
    print(5.5);                               // Double: 5.5
    return 0;
}
```

---

6. Виртуални деструктори

Защо са нужни:

- Когато обект от производен клас се изтрива чрез указател към базов клас, виртуалният деструктор гарантира, че деструкторите на производния и базовия клас ще бъдат извикани в правилния ред.
- Без виртуален деструктор ще се извика само деструкторът на базовия клас, което може да доведе до memory leak или недефинирано поведение.

Правила:

- **Винаги** дефинирайте виртуален деструктор в базови класове, които имат виртуални функции.
- Ако класът е абстрактен, деструкторът обикновено е `= default`, за да се избегне ненужна имплементация.

Пример:

```cpp
#include <iostream>

class Base {
public:
    virtual ~Base() { std::cout << "Base destructor" << std::endl; }
};
```

```cpp
class Derived : public Base {
public:
    ~Derived() override { std::cout << "Derived destructor" << std::endl; }
};

int main() {
    Base* obj = new Derived();
    delete obj; // Извиква Derived::~Derived(), после Base::~Base()
    return 0;
}
```

Изход:

```
Derived destructor
Base destructor
```

Забележка: Ако `~Base()` не е виртуален, ще се извика само `Base::~Base()`, което може да пропусне освобождаването на ресурси в `Derived`.

Задача: Система за обработка на плащания

Разработете система за обработка на плащания, която поддържа различни методи на плащане (кредитни карти, PayPal, биткойн и банкови преводи). Системата трябва да позволява добавяне, премахване, сравнение и групова обработка на плащания, като осигурява валидация на данни, история на транзакциите и защита срещу грешки чрез обработка на изключения.

Изисквания

1. Базов клас `PaymentMethod`

Създайте **абстрактен клас** `PaymentMethod`, който ще служи като основа за всички методи на плащане. Класът трябва да предоставя следните методи:

- **Чисто виртуална функция** `pay(double amount)`:
  - Обработва плащане на посочената сума (`amount`).
  - Връща `bool`, който указва дали плащането е успешно.
- **Чисто виртуална функция** `validate() const`:
  - Проверява валидността на данните, свързани с метода на плащане (напр. формат на имейл или номер на карта).
  - Връща `bool`, указващ дали данните са валидни.
- **Чисто виртуална функция** `getType() const`:

    - Връща `std::string`, представляващ типа на метода на плащане (напр. "CreditCard", "PayPal").
- **Виртуален деструктор**:
    - Гарантира правилно освобождаване на ресурси при изтриване на обекти чрез указател към базовия клас.
- **Оператори за сравнение**:
    - `operator==` за сравняване на методи по техния тип.
    - `operator<` за сортиране на методи по тип (лексикографски).

2. Производни класове за методи на плащане

Имплементирайте следните четири класа, които наследяват `PaymentMethod`:

2.1. `CreditCardPayment`

- **Полета**:
    - `cardNumber`: `std::string` (номер на картата).
    - `cvv`: `std::string` (CVV код).
    - `expiryDate`: `std::string` (срок на валидност във формат "MM/YY").
- **Валидация (в `validate()`)**:
    - `cardNumber` трябва да съдържа точно 16 цифри.
    - `cvv` трябва да съдържа точно 3 цифри.
    - `expiryDate` трябва да е във формат "MM/YY" и да представлява дата след текущата.
- **Плащане (в `pay()`)**:
    - Ако данните са невалидни или сумата е неположителна, връща `false` и извежда съобщение за грешка.
    - При успех връща `true` и извежда съобщение за успешно плащане.

2.2. `PayPalPayment`

- **Полета**:
    - `email`: `std::string` (имейл адрес).
    - `accountBalance`: `double` (наличен баланс в акаунта).
- **Валидация (в `validate()`)**:
    - `email` трябва да е валиден (да съдържа символите `@` и `.`).
    - `accountBalance` трябва да е достатъчен за покриване на плащането.
- **Плащане (в `pay()`)**:
    - Ако данните са невалидни, балансът е недостатъчен или сумата е неположителна, връща `false` и извежда съобщение за грешка.
    - При успех намалява `accountBalance` със сумата, връща `true` и извежда съобщение за успешно плащане.

2.3. `BitcoinPayment`

- **Полета**:
  - `walletAddress`: `std::string` (адрес на биткойн портфейл).
  - `transactionFee`: `double` (фиксирана такса за транзакция).
- **Валидация (в `validate()`)**:
  - `walletAddress` трябва да е низ с дължина поне 26 символа.
- **Плащане (в `pay()`)**:
  - Ако данните са невалидни или сумата е неположителна, връща `false` и извежда съобщение за грешка.
  - При успех добавя `transactionFee` към сумата, връща `true` и извежда съобщение за успешно плащане.

2.4. `BankTransferPayment`

- **Полета**:
  - `iban`: `std::string` (IBAN на банковата сметка).
  - `bankName`: `std::string` (име на банката).
- **Валидация (в `validate()`)**:
  - `iban` трябва да започва с "BG" и да е с дължина точно 22 символа.
- **Плащане (в `pay()`)**:
  - Ако данните са невалидни или сумата е неположителна, връща `false` и извежда съобщение за грешка.
  - При успех връща `true` и извежда съобщение за успешно плащане.

3. Клас за управление на плащания `PaymentProcessor`

Създайте клас `PaymentProcessor`, който управлява списък с методи на плащане и история на транзакциите. Класът трябва да:

- **Съхранява методи**:
  - Използвайте `std::vector<PaymentMethod*>` за съхранение на указатели към обекти от тип `PaymentMethod`.
- **Поддържа история на транзакциите**:
  - Използвайте `std::vector<std::pair<PaymentMethod*, double>>` за запис на успешните плащания (метод и сума).
- **Предоставя следните методи**:
  - `addMethod(PaymentMethod* method)`:
    - Добавя метод към списъка, ако неговата валидация (`validate()`) е успешна.
    - Ако валидацията не успее, хвърля `std::invalid_argument` с подходящо съобщение.

- `removeMethod(const PaymentMethod& method)`:
  - Премахва първия метод, който съвпада с подадения по `operator==`.
- `processAll(double amount)`:
  - Извиква `pay(amount)` за всеки метод в списъка.
  - Записва успешните плащания в историята на транзакциите.
  - Връща брой на успешните плащания (цяло число).
  - Ако сумата е неположителна, хвърля `std::invalid_argument`.
- `getTotalProcessed() const`:
  - Връща общата сума на всички успешни транзакции в историята (`double`).
- `sortMethods()`:
  - Сортира списъка с методи лексикографски по тип, използвайки `operator<`.

4. Обработка на изключения

- В `addMethod()`: Хвърля `std::invalid_argument`, ако валидацията на метода не успее, с описателно съобщение за причината.
- В `pay()`: Не хвърля изключения; вместо това връща `false` при неуспех (напр. недостатъчен баланс или невалидни данни).
- В `processAll()`: Хвърля `std::invalid_argument`, ако сумата е неположителна, с подходящо съобщение.

5. Тестове

- Напишете модулни тестове, използвайки библиотеката **Catch2**, за да проверите:
  - Валидацията на всеки метод на плащане.
  - Коректността на обработката на плащания (успешни и неуспешни случаи).
  - Управлението на методи и транзакции в `PaymentProcessor` (добавяне, премахване, сортиране, история).
  - Обработката на изключения при невалидни данни или суми.