OOP #3

OBJECT-ORIENTED PROGRAMMING #3

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

AGENDA

1. dziedziczenie
2. wielodziedziczenie
3. funkcje wirtualne
4. funkcje czysto wirtualne
5. klasy abstrakcyjne
6. interfejsy
7. przykładowe rozwiązania zadań z OOP#1
8. polimorfizm
9. pola i metody statyczne

ZADANIA

Repo GH `coders-school/object-oriented-programming`

https://github.com/coders-school/object-oriented-programming/tree/master/module2

POWTÓRKA OBIEKTOWOŚCI

- Co oznacza słowo `virtual`?
- Co oznacza słowo `override`?
- Co to jest dziedziczenie?
- Do czego służą operatory?
- Co to jest konstruktor?
- Czym jest klasa abstrakcyjna?

PROGRAMOWANIE OBIEKTOWE

PRZYKŁADOWE ROZWIĄZANIA

CODERS SCHOOL

DISCLAIMER

W PDFie te rozwiązania mogą być częściowo ucięte. Przejdź do lekcji na platformie lub na GitHubie, aby zobaczyć je w całości.

ZADANIE 1

Cargo.hpp

```
bool operator==(const Cargo& cargo) const;
```

Cargo.cpp

```
bool Cargo::operator==(const Cargo& cargo) const {
    return Cargo.getBasePrice() == base_price_ && Cargo.getName() == name_;
}
```

ZADANIE 2

Cargo.hpp

```
std::string getName() const;
size_t getAmount() const;
size_t getBasePrice() const;
```

Cargo.cpp

```
std::string Cargo::getName() const { return name_; }
size_t Cargo::getAmount() const { return amount_; }
size_t Cargo::getBasePrice() const { return base_price_; }
```

ZADANIE 3 #1

Island.hpp

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

#include "Store.hpp"

class Time;

// Class describes position of island and available store.
class Island {
public:
    class Coordinates {
    public:
        Coordinates() = default;
        Coordinates(size_t pos_x, size_t pos_y)
            : pos_x_(pos_x), pos_y_(pos_y) {}

        bool operator==(const Coordinates& lhs) const {
            return this->pos_x_ == lhs.pos_x_ && this->pos_y_ == lhs.pos_y_;
        }

    private:
```

ZADANIE 3 #2

Island.cpp

```
#include "GTime.hpp"
#include "Island.hpp"

Island::Island(size_t pos_x, size_t pos_y, Time* time)
    : position_(Coordinates(pos_x, pos_y)),
      store_(std::make_unique<Store>(time)) {
}
```

ZADANIE 4/5/6 #1

Map.hpp

```
class Time;

// Class responsible for map generation which will be used to travel.
class Map {
public:
    Map(Time* time);
    void travel(Island* destination);
    Island* getIsland(const Island::Coordinates& coordinate);
    Island* getCurrentPosition() const { return current_position_; }
    friend std::ostream& operator<<(std::ostream& out, const Map& map);

private:
    Island* current_position_;
    std::vector<Island> islands_;
};
```

ZADANIE 4/5/6 #2

Map.cpp

```
constexpr size_t kIslandNum = 10;
constexpr size_t kWidth = 50;
constexpr size_t kHeight = 50;

Map::Map(Time* time) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> widthGen(0, kWidth);
    std::uniform_int_distribution<> heightGen(0, kHeight);
    std::vector<Island> islands(kIslandNum);
    std::vector<std::pair<size_t, size_t>> usedPositions;

    // Generate map
    for (size_t i = 0; i < kIslandNum; ++i) {
        while (true) {
            size_t x = widthGen(gen);
            size_t y = heightGen(gen);
            if (std::find_if(begin(usedPositions),
                             end(usedPositions),
                             [x, y](const auto& pos) {
                                 return pos.first == x && pos.second == y;
```

ZADANIE7/8 #1

Player.hpp

```cpp
#include <memory>

#include "Cargo.hpp"
#include "Ship.hpp"

class Time;

// Class is responsible for every action made by player
class Player {
public:
    Player(size_t money, Time* time);
    virtual ~Player() = default;

    virtual size_t getAvailableSpace() const;
    virtual size_t getMoney() const;
    virtual size_t getSpeed() const;
    virtual Cargo* getCargo(size_t index) const;

private:
    std::unique_ptr<Ship> ship_;
    size_t money_;
```

ZADANIE 7/8 #2

Player.cpp

```
constexpr size_t kCapacity = 100;
constexpr size_t kCrew = 50;
constexpr size_t kSpeed = 10;
constexpr char kName[] = "BLACK WIDOW";
constexpr size_t kId = 10;

Player::Player(size_t money, Time* time)
    : ship_(std::make_unique<Ship>(kCapacity, kCrew, kSpeed, kName, kId, time,
      money_(money),
      available_space_(kCapacity) {
}

size_t Player::getMoney() const {
    return money_;
}

size_t Player::getSpeed() const {
    return ship_->getSpeed();
}

Cargo* Player::getCargo(size_t index) const {
```

Q&A

PROGRAMOWANIE OBIEKTOWE

POLIMORFIZM

CODERS SCHOOL

SŁOWO KLUCZOWE `virtual`

Jeżeli chcemy, aby przy używaniu wskaźników lub referencji na klasę bazową, jakaś metoda zachowywała się inaczej w zależności od prawdziwego typu obiektu, to należy ją oznaczyć słowem kluczowym `virtual`. Jest to tzw. funkcja wirtualna.

FUNKCJA NIE-WIRTUALNA

```cpp
#include <iostream>

struct Bird {
    void sing() { std::cout << "tweet, tweet\n"; }
};

struct Sparrow : Bird {
    void sing() { std::cout << "chirp, chirp\n"; }
};

int main() {
    Sparrow sparrow;
    Bird& bird = sparrow;
    bird.sing();
    return 0;
}
```

Co pojawi się na ekranie?

`tweet, tweet`

FUNKCJA WIRTUALNA

```
#include <iostream>

struct Bird {
    virtual void sing() { std::cout << "tweet, tweet\n"; }
};

struct Sparrow : Bird {
    void sing() { std::cout << "chirp, chirp\n"; }
};

int main() {
    Sparrow sparrow;
    Bird& bird = sparrow;
    bird.sing();
    return 0;
}
```

Co pojawi się na ekranie?

`chirp, chirp`

Sprawdź na ideone.com

SŁOWO KLUCZOWE `override`

Jeżeli w klasie pochodnej **nadpisujemy** metodę wirtualną, czyli zmieniamy jej zachowanie, to należy dodać słowo `override`.

```
class Interface {
public:
    virtual void doSth() = 0;
};

class SomeClass : public Interface {
public:
    doSth() override;   // there should be an implementation in cpp file
};

int main() {
    Interface interface;    // Compilation error, Interface is pure virtual
    SomeClass someClass;    // OK
    Interface* interface = &someClass;  // OK, we hold a pointer
}
```

MAŁA UWAGA

`override` jest opcjonalne. Jeśli go nie podamy za sygnaturą funkcji klasy pochodnej to metoda z klasy bazowej i tak zostanie nadpisana.

Jego użycie jest jednak dobrą praktyką, bo dzięki niemu kompilator sprawdzi czy faktycznie przeciążamy metodą z klasy bazowej i jeśli nie, to program się nie skompiluje.

Bez `override` mogłaby zostać utworzona nowa metoda w klasie pochodnej, która nie nadpisuje niczego z klasy bazowej.

Metody wirtualne **nadpisujemy**, nie przeciążamy.

NADPISYWANIE METOD - `override`

Wracając do przykładu o ptakach, klasy `Penguin`, `Hummingbird` oraz `Goose` to klasy pochodne, które dziedziczą po pewnych klasach bazowych jak `Bird`, `Flyable`, `Soundable`, `Swimmable` oraz nadpisują kilka ich metod jak:

- `void eat() override`
- `void sleep() override`
- `void makeSound() override`
- `void fly() override`
- `void swim() override`

Nadpisanie takich metod powoduje, że możemy zmienić ich implementacje.

override

```
class Soundable {
public:
    virtual void makeSound() = 0;
};
```

```
class Goose : public Soundable {
public:
    void makeSound() override { std::cout << "Honk! Honk!"; }
};
```

```
class Hen : public Soundable {
public:
    void makeSound() override { std::cout << "Cluck! Cluck!"; }
};
```

```
class Duck : public Soundable {
public:
    void makeSound() override { std::cout << "Quack! Quack!"; }
};
```

WSPÓLNA KLASA BAZOWA

Ponieważ wspólnym rodzicem wszystkich klas jest klasa `Soundable` możemy przechowywać w kontenerze wskaźniki typu `Soundable`.

```
std::vector<std::shared_ptr<Soundable>> birds_;
```

JAKIE DANE OTRZYMAMY NA WYJŚCIU?

```
std::vector<std::shared_ptr<Soundable>> birds_;
birds_.push_back(std::make_shared<Goose>());
birds_.push_back(std::make_shared<Hen>());
birds_.push_back(std::make_shared<Duck>());

std::cout << birds_[0]->makeSound() << '\n';
std::cout << birds_[1]->makeSound() << '\n';
std::cout << birds_[2]->makeSound() << '\n';
```

POLIMORFIZM

Zjawisko, które właśnie zaobserwowaliśmy, nazywa się polimorfizmem.

Polimorfizm pozwala funkcji przybrać różne formy (implementacje), tak jak na przykładzie.

Dlatego, jeżeli utworzymy kolejno obiekty `Goose`, `Hen` i `Duck` w zależności od obiektu zostanie wywołana jego wersja metody `makeSound`.

Polimorfizm włącza się, gdy mamy funkcje wirtualne i używamy wskaźników lub referencji na typ bazowy.

KTO GRAŁ LUB CZYTAŁ WIEDŹMINA?

DOPPLER :)

W uniwersum wykreowanym przez naszego rodzimego pisarza Andrzeja Sapkowskiego, występuje pewna intrygująca i ciekawa rasa zwana Dopplerami.

Rasa ta potrafi przyjąć, postać różnych form życia, może stać się człowiekiem, elfem, krasnoludem. Zmienia w ten sposób swoje cechy jak głos, kolor włosów, a nawet ubranie!

Pomimo że rasa ta jest typu Doppler, potrafi w różnych okolicznościach podszywać się pod inne rasy jak elf, krasnolud czy człowiek.

Z punktu widzenia C++ nasz Doppler podlega zjawisku polimorfizmu.

```cpp
class Doppler {
public:
    virtual sayHello() { std::cout << "I'm Doppler!"; }
};

class Dwarf : public Doppler {
public:
    virtual sayHello() { std::cout << "I'm Dwarf!"; }
};

class Elf : public Doppler {
public:
    virtual sayHello() { std::cout << "I'm Elf!"; }
};

class Human : public Doppler {
public:
    virtual sayHello() { std::cout << "I'm Human!"; }
};

int main() {
    std::shared_ptr<Doppler> doppler1 = std::make_shared<Dwarf>();
    std::shared_ptr<Doppler> doppler2 = std::make_shared<Elf>();
    std::shared_ptr<Doppler> doppler3 = std::make_shared<Human>();
```

Jak widzimy, nasz Doppler może przyjąć różne formy i zachowywać się tak jak one. Wskaźnik jest typu Doppler, ale program dobrze wie, kiedy Doppler podszywa się pod człowieka, kiedy pod krasnoluda, a kiedy pod elfa.

NIE-WIRTUALNE DESTRUKTORY

Bardzo ważne w przypadku tworzenia metod wirtualnych i dziedziczenia jest tworzenie wirtualnych destruktorów. Jeżeli korzystamy z dobroci polimorfizmu i nie oznaczymy destruktor jako `virtual` to destruktor ten nie zostanie wywołany.

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

class Parent {
public:
    Parent() { std::cout << "PARENT C'tor called\n"; }
    ~Parent() { std::cout << "PARENT D'tor caller\n"; }
};

class Child : public Parent {
public:
    Child() { std::cout << "CHILD C'tor called\n"; }
    ~Child() { std::cout << "CHILD D'tor caller\n"; }
};

int main() {
    Child child;    // ok, object on stack, not a pointer
}
```

NIE-WIRTUALNE DESTRUKTORY - PROBLEM

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

class Parent {
public:
    Parent() { std::cout << "PARENT C'tor called\n"; }
    ~Parent() { std::cout << "PARENT D'tor caller\n"; }
};

class Child : public Parent {
public:
    Child() { std::cout << "CHILD C'tor called\n"; }
    ~Child() { std::cout << "CHILD D'tor caller\n"; }
};

int main() {
    // Problem
    std::unique_ptr<Parent> child = std::make_unique<Child>();

    // But shared_ptr will cleanup properly
```

WIRTUALNY DESTRUKTOR

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

class Parent {
public:
    Parent() { std::cout << "PARENT C'tor called\n"; }
    virtual ~Parent() { std::cout << "PARENT D'tor caller\n"; }
};

class Child : public Parent {
public:
    Child() { std::cout << "CHILD C'tor called\n"; }
    ~Child() override { std::cout << "CHILD D'tor caller\n"; }
};

int main() {
    std::unique_ptr<Parent> child2 = std::make_unique<Child>();
}
```

Q&A

ZADANIE 3

Napisz klasę `DryFruit`, która dziedziczyć będzie po klasie `Fruit`.

Klasa ta powinna przeciążać metody `getPrice()`, `getName()` oraz `operator--`.

`operator--` powinien odejmować zużycie raz na 10 wywołań.

Metoda `getPrice()` powinna zwracać trzykrotnie większą wartość w porównaniu do ceny bazowej.

Przetestuj wywołania polimorficzne oraz podziel się wnioskami.

PROGRAMOWANIE OBIEKTOWE

ZMIENNE I FUNKCJE STATYCZNE

CODERS SCHOOL

"ZMIENNA LUB STAŁA KLASY"

Czasami chcielibyśmy przypisać jakąś stałą cechę do klasy. Nie konkretnych obiektów, a klasy samej w sobie. Np. każdy obiekt klasy ma nazwę "Object".

```
class Object {
public:
    std::string GetName() const { return name_; }

private:
    const std::string name_ = "Object";
};
```

W celu otrzymania nazwy tego obiektu, musimy najpierw utworzyć obiekt, a następnie zawołać `getName()`.

```
int main() {
    Object obj;
    std::cout << obj.getName() << '\n';
}
```

Nawet jeżeli obiekt zajmowałby dużo miejsca w pamięci a my chcielibyśmy tylko jego nazwę i tak musimy go utworzyć, ponieważ tylko na nim możemy zawołać metodę `getName()`.

static

Rozwiązaniem tej uciążliwości jest `static`. Co więcej, problem ten możemy rozwiązać na 2 sposoby. Nie musimy w ten sposób tworzyć specjalnie obiektu, aby dostać się do cechy klasy, jaką jest jej nazwa.

```
class ObjectA {
public:
    static std::string getName() { return "ObjectA"; }
};

class ObjectB {
public:
    static std::string name_;
};

std::string ObjectB::name_{"ObjectB"};

int main() {
    std::cout << ObjectA::getName() << '\n';
    std::cout << ObjectB::name_ << '\n';

    return 0;
}
```

Q&A

ZADANIE 4

Przekształć klasę bazową `Coordinates`, tak aby miała funkcję statyczną

```
static size_t distance(const Coordinates& lhs, const Coordinates& rhs)
```

Funkcja ta powinna zwracać dystans pomiędzy dwoma pozycjami.

PROGRAMOWANIE OBIEKTOWE

PODSUMOWANIE

CODERS SCHOOL

CO PAMIĘTASZ Z DZISIAJ?

NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

1. dziedziczenie
2. wielodziedziczenie
3. funkcje wirtualne
4. funkcje czysto wirtualne
5. klasy abstrakcyjne
6. interfejsy
7. polimorfizm
8. pola i metody statyczne

PRE-WORK

- Dowiedzcie się czym jest problem diamentowy
- Poczytajcie o zasadach SOLID, dotyczących pisania dobrego kodu obiektowego
- Lektura o wzorcach projektowych z przykładami w C++ - refactoring.guru
- Spróbujcie w grupie metodą Copy & Paste dorzucić system budowania cmake do projektu. W tym celu popatrzcie na dotychczasowe zadania domowe i plik CMakeLists.txt.

PROJEKT GRUPOWY

Wykorzystajcie kod napisany podczas zajęć. Możecie też skorzystać z kodu w katalogu solutions

Projekt grupowy - kontynuacja. Możecie zmienić grupę jeśli chcecie ;)

ORGANIZACJA PRAC

- Jak wyglądało wasze daily?
- Czy Code Review nie jest zaniedbane?
- Czy współpraca idzie gładko?
- Zróbcie sobie retrospektywę :)

PUNKTACJA

- 3 pierwsze zadania - 5 punktów
- zadania 4, 5, 6 - 8 punktów
- 20 punktów za dostarczenie wszystkich 6 zadań przed 05.07.2020 (niedziela) do 23:59
- brak punktów bonusowych za dostarczenie tylko części zadań przed 05.07
- 6 punktów za pracę w grupie dla każdej osoby z grupy.

ZADANIE 1

Napisz klasę `Store`, która będzie umożliwiała dokonywanie zakupów. Wykorzystaj poniższy enum i funkcje.

```
enum class Response {done, lack_of_money, lack_of_cargo, lack_of_space};

Response buy(Cargo* cargo, size_t amount, Player* player);
Response sell(Cargo* cargo, size_t amount, Player* player);
```

ZADANIE 2

W klasach `Alcohol`, `Fruit`, `Item` dopisz brakujące metody oraz ich implementacje.

```
// override from Cargo
size_t getPrice() const override;
std::string getName() const override { return name_; }
size_t getAmount() const override { return amount_; }
size_t getBasePrice() const override { return base_price_; }
Cargo& operator+=(size_t amount) override;
Cargo& operator-=(size_t amount) override;
bool operator==(Cargo& cargo) const override;
```

ZADANIE 3

Dopisz do klasy `Ship`, `Cargo` oraz `Stock` metodę `nextDay()`

- Klasa `Ship`: Metoda powinna odejmować po 1 sztuce monety za każdego członka załogi.
- Klasa `Cargo`: Metoda powinna powodować psucie się towarów.
- Klasa `Stock`: Metoda powinna zmieniać ilość towaru w sklepach.

ZADANIE 4 (DLA AMBITNYCH)

Spróbuj napisać klasę `Time`, która będzie odpowiadać za zarządzanie czasem w grze.

Klasa ta powinna informować inne klasy, takie jak `Cargo`, `Ship`, `Stock` o upłynięciu każdego dnia.

Poczytaj czym jest wzorzec projektowy `Observer`.

ZADANIE 5 (DLA AMBITNYCH)

Napisz zaprzyjaźniony operator wypisywania do strumienia

```
friend std::ostream& operator<<(std::ostream& out, const Store& store);
```

Ma on w przystępny sposób wypisywać towar, jaki znajduje się w danym sklepie.

ZADANIE 6 (DLA AMBITNYCH)

Napisz klasę `Game`, która zarządzać będzie całą rozgrywką.

Dodaj jej jedną metodę publiczną `startGame()`.

Finalnie plik main powinien wyglądać tak:

```cpp
#include "Game.hpp"

constexpr size_t start_money = 1'000;
constexpr size_t game_days = 100;
constexpr size_t final_goal = 2'000;

int main() {
    Game game(start_money, game_days, final_goal);
    game.startGame();

    return 0;
}
```

CODERS SCHOOL