Programowanie Obiektowe

Laboratorium 1 - wstęp do programowania obiektowego w języku C++

2019-02-26

Wstęp

W laboratorium zostaną przedstawione podstawowe zagadnienia dotyczące programowania obiektowego na przykładzie aplikacji C++.

Laboratorium prowadzić będziemy analizując kolejne paradygmaty programowania obiektowego i przedstawiając na przykładzie ich implementację w aplikacji. Zadaniem studentów będzie analiza kodu w niniejszej instrukcji oraz stworzenie nowych elementów aplikacji zgodnie ze wskazówkami podanymi w kolejnych zadaniach.

Czym jest programowanie obiektowe?

Klasyczne programowanie obiektowe, abstrahując od konkretnej implementacji językowej, systemowej etc. jest zasadniczo zdefiniowane za pomocą czterech tzw. paradygmatów programowania. Paradygmaty te są nie tyle konkretnymi twierdzeniami czy regułami wg których musi działać każdy program obiektowy, ale są pewnymi założeniami czy też zaleceniami według których powinno się tworzyć aplikacje w środowisku obiektowym. W literaturze można znaleźć cztery takie zagadnienia, a mianowicie:

- ✓ abstrakcja
- ✓ dziedziczenie
- ✓ polimorfizm (wielopostaciowość)
- ✓ hermetyzacja (enkapsulacja)

W niniejszym opisie nie będziemy zagłębiać się w przesłanie teoretyczne każdego z tych pojęć, gdyż informacje te zostały już do tej pory podane na wykładzie i z pewnością każdy student doskonale je rozumie ;>. Skupimy się natomiast na praktycznym znaczeniu tych zagadnień, tworząc przykładową aplikację w C++ składającą się z kilku klas. Zaczniemy od tworzenia klas bazowych (zadanie 1), przechodząc następnie do szczegółowej implementacji w klasach pochodnych (zadanie 2). Wyniki działania programu zostaną przedstawione w zadaniu 3.

Zadanie 1.

Tworzenie aplikacji obiektowej zaczynamy od definicji klasy, która została przedstawiona poniżej. Kod tej klasy będzie znajdował się w dwóch plikach.

FiguraPlaska.h:

```
#include <iostream>


class FiguraPlaska {

protected:

    virtual void Wypisz(std::ostream& out) const = 0;

    friend std::ostream& operator<<(std::ostream& os, const FiguraPlaska&
figura);

public:

    virtual double Pole() = 0;

    virtual double Obwod() = 0;

    virtual ~FiguraPlaska();

};
```

FiguraPlaska.cpp:

```
#include "FiguraPlaska.h"

std::ostream& operator<<(std::ostream& os, const FiguraPlaska& figura) {
    figura.Wypisz(os);
    return os;
}

FiguraPlaska::~FiguraPlaska() {

}
```

Listing zawiera nagłówkową definicję klasy FiguraPlaska zawartą w pliku *.h oraz implementację operatora << i destruktora zawarte w pliku *.cpp. Klasa ta w naszej aplikacji będzie reprezentować abstrakcyjną figurę płaską, która może być prostokątem, kołem, trójkątem lub inną figurą. Przyjrzyjmy się deklaracjom metod i operatorów. Pierwsze dwie deklaracje znajdują się sekcji **protected.** Pierwsza deklaracja wygląda następująco:

```
virtual void Wypisz(std::ostream& out) const = 0;
```

Słowo kluczowe **virtual** oznacza że metoda Wypisz będzie wirtualna. Metody wirtualne mogą być nadpisywane (**override**) w klasach potomnych. Na przykład gdy stworzymy klasę **Prostokat**, która będzie dziedziczyć po klasie **FigruaPlaska** możemy nadpisać metodę **Wypisz,** każąc jej wypisać do strumienia napis „Prostokąt" oraz wymiary opisywanego prostokąta. Cały potencjał metody wirtualnej **Wypisz** ujawnia się jednak, gdy **wskaźnik** na obiekt typu **Prostokat** rzutujemy na wskaźnik na obiekt typu **FiguraPlaska**. Wywołując metodę **Wpisz** na wskaźniku typu **FiguraPlaska***, który pokazuje na obiekt typu **Prostokat**, wywołujemy metodę **Wypisz** właściwą dla kasy **Prostokat**[1]. Gdyby metoda **Wypisz** nie była wirtualna, to uruchomiona zostałaby implementacja metody dla klasy **FigruaPlaska** – zgodnie z typem wskaźnika. Szczególnym rodzajem metod wirtualnych są metody czysto wirtualne.

Typ zwracany **void** oznacza oczywiście, że metoda nie będzie zwracała żadnego wyniku. Dalej znajduje się nazwa metody. Metoda ma jeden parametr jest typu **std::ostream&** - innymi słowy jest to referencja[2] do obiektu typu **std::ostream** lub typu pochodnego. Następnie za nawiasem zamykającym znajduje się słowo kluczowe **const.** W tym miejscu oznacza ono, iż programista obiecuje, że w metodzie Wypisz nie będzie modyfikował obiektu którego ona dotyczy. Dzięki temu można bez problemu wywołać tę metodę nawet na obiekcie który będzie zadeklarowany jako **const** (czyli stały, nie zmienny).

= 0 oznacza, że metoda ta jest czyta wirtualna. Metoda taka nie posiada implementacji i koniecznym jest aby w którejś klasie potomnej została zaimplementowana przez nadpisanie (**override**). Jeżeli klasa posiada metodę czystą wirtualną, to jest klasą abstrakcyjną – nie może zostać utworzony obiekt tej klasy.

```
friend std::ostream& operator<<(std::ostream& os, const FiguraPlaska& figura);
```

Słowo kluczowe **friend** wskazuje że dalej zadeklarowana albo zdefiniowana funkcja lub zadeklarowana klasa ma dostęp do prywatnych i chronionych składowych tej klasy. Tak zadeklarowana albo zdefiniowana funkcja albo klasa **nie jest** elementem klasy w której deklarowana albo definiowana. Ta deklaracja dotyczy operatora <<, który ma zastosowanie w operacjach na strumieniach (np. std::cout). Według tej konwencji operator taki powinien zwrócić referencję na strumień który jest pierwszym argumentem. Drugim argumentem jest **const FiguraPlaska&** który jest referencją na stały obiekt typu **FiguraPlaska**. Oznacza to właściwie, że operator ten obiecuje, że nie będzie zmieniał obiektu na który wskazuje ta referencja. W związku z tym będzie mógł wywoływać tylko metody oznaczone słowem kluczowym **const**.

W sekcji public zadeklarowane są następujące metody:

```
virtual double Pole() = 0;

virtual double Obwod() = 0;
```

Powyższe dwie pozycje nie powinny sprawić kłopotu z interpretacją.

[1] W sposób analogiczny metody wirtualne działają w przypadku rzutowania na referencję klasy bazowej.

[2] Patrz Dodatek: Sposoby dostępu do zmiennej str. 10

```
virtual ~FiguraPlaska();
```

To jest deklaracja destruktora. Ponieważ **FiguraPlaska** ma być klasą bazową dla innych klas konieczne jest ustawienie destruktora jako wirtualnego. Bez słowa kluczowego **virtual** przy wielostopniowej hierarchii klas występowałyby problemy z uruchomieniem odpowiedniego destruktora, co może prowadzić do wycieków pamięci i innych dziwnych efektów.

UWAGA!!
Oto pytania których możesz się spodziewać od prowadzącego w tej chwili;-)

- ✓ *Zastanów się co to są: typ prosty i złożony, klasa, obiekt i struktura, w C++ - jakie są różnice, podobieństwa i zależności między tymi pojęciami.*
- ✓ *co to jest metoda wirtualna i tzw. metoda czysta wirtualna (ang. pure virtual) i jakiego paradygmatu dotyczy?*
- ✓ *czy klasa FiguraPlaska jest abstrakcyjna i dlaczego?*
- ✓ *co to jest operator zaprzyjaźniony?*

Po stworzeniu klasy bazowej przyszedł czas na stworzenie definicji klasy pochodnej. W pierwszej kolejności zajmiemy się tworzeniem klasy reprezentującej w aplikacji prostokąt.

Przykładowa zawartość pliku nagłówkowego dla klasy nagłówkowy **`Prostokat:`**

```
#include "FiguraPlaska.h"


class Prostokat : public FiguraPlaska {
private:
    double a,b;
protected:
    void Wypisz(std::ostream& out) const override;
public:
    Prostokat(double a, double b);

    double GetA() const;

    void SetA(double a);

    double GetB() const;

    void SetB(double b);

    double Obwod() override;
    double Pole() override;

    ~Prostokat() override;
};
```

oraz zawartość pliku *.cpp:

```
#include "Prostokat.h"
```

```cpp
#include <iostream>

using namespace std;


Prostokat::Prostokat(double a, double b)
: a(a), b(b)
{
    cout << "Konstruktor Prostokąta(" << a <<"," << b << ")" << endl;

}

double Prostokat::GetA() const {
    return a;
}

double Prostokat::GetB() const {
    return b;
}

void Prostokat::SetA(double a) {
    this->a = a;
}

void Prostokat::SetB(double b) {
    this->b = b;
}

double Prostokat::Obwod() {
    //TODO 1
}

double Prostokat::Pole() {
    //TODO 2
}

void Prostokat::Wypisz(std::ostream& out) const {
    //TODO 3
}

Prostokat::~Prostokat() {
    //TODO 4
}
```

W listingu nr 2, widzimy klasę `Prostokat` zdefiniowaną w piku *.cpp.

W sekcji **private** widać:

```cpp
double a,b;
```

Jest to deklaracja pól klasy a i b o typie **double.**

W sekcji **protected** znajduje się jedna deklaracja:

```
void Wypisz(std::ostream& out) const override;
```

Jest ona odpowiednikiem deklaracji metody czystej wirtualnej **Wypisz** z klasy **FiguraPlaska**. Deklaracja ta pojawiła się w tym miejscu, gdyż chcieliśmy nadpisać (w tym przypadku zaimplementować) metodę **Wypisz** klasie **Prostakat**. Nie ma słowa kluczowego virtual, ale za to pojawiło się słowo kluczowe **override**. Słowo to informuje kompilator, że chcemy nadpisać metodę z klasy bazowej. Gdyby takiej metody w klasie bazowej by nie było, to kompilator zgłosiłby błąd. Słowo kluczowe **override** jest nieobowiązkowe. Po jego usunięciu kod by się skompilował i uruchomił poprawnie. Jednak gdyby w klasie bazowej zabrakło metody Wypisz, a słowo **override** byłoby obecne w klasie pochodnej to kompilator zgłosiłby błąd. Słowo kluczowe **override**, jest nowym elementem wprowadzonym w **C++11**.

Dalej w sekcji **public** znajdują się następujące pozycje:

```
Prostokat(double a, double b);
```

Jest to deklaracja konstruktora który przyjmuje dwa parametry typu double – odpowiadają one długością boków.

```
double GetA() const;

void SetA(double a);

double GetB() const;

void SetB(double b);
```

To są deklaracje tzw. geterów i seterów, czyli zazwyczaj prostych metod które odpowiadają za odczytywanie i zapisywanie właściwości obiektu. Getery i setery mają za zadanie ukryć implementacje tych właściwości, dlatego właściwości te niekoniecznie muszą być one zaimplementowane w postaci pól składowych klasy. Dobrze jest aby przy metodach Get* znajdowało się słowo kluczowe **const** umożliwiające wywołanie tej metody nawet na obiekcie stałym.

```
double Obwod() override;
double Pole() override;

~Prostokat() override;
```

Powyżej widać deklaracje nadpisania metod Obwod, Pole oraz destruktora.

W pliku *.cpp wspomniano znajdują definicje poszczególnych metod:

```
Prostokat::Prostokat(double a, double b) : a(a), b(b) {
    cout << "Konstruktor Prostokąta(" << a <<"," << b << ")" << endl;

}
```

Powyżej znajduje się definicja konstruktora. Składa się ona z prefiksu będącego nazwą klasy, dwóch dwukropków i nazwy konstruktora która jest też nazwą klasy. Powtórzone są definicje dwóch parametrów typu double. Elementem charakterystycznym konstruktora jest możliwość użycia tzw, listy inicjalizacyjnej, która pozwala na inicjalizacje pól składowych klasy oraz wywołanie konstruktora klasy bazowej. W przypadku niestałych pól klasy, które są danymi typów prostych, lista inicjalizacyjna nie ma szczególnego znaczenia. Można z niej skorzystać lub nie. Natomiast w przypadku pola będącego obiektem jakiejś klasy pozwala na uruchomienie innego konstruktora niż konstruktora domyślnego dla tego pola. Lista inicjalizacyjna składa się z dwukropka oraz listy pól oraz typów bazowych będącymi wywołaniami konstruktorów.

```
double Prostokat::GetA() const {
    return a;
}

double Prostokat::GetB() const {
    return b;
}

void Prostokat::SetA(double a) {
    this->a = a;
}

void Prostokat::SetB(double b) {
    this->b = b;
}
```

Powyżej znajdują się definicje metod. Definicja zaczyna się od typu zwracanego nazwy klasy dwóch dwukropków oraz nazwy metody. Ponieważ słowo kluczowe **const** jest ważnym elementem sygnatury metody, konieczne jest powtórzenie go przy definicji.

Końcem zadania 1. jest skompilowanie stworzonej aplikacji. Zwróć uwagę że w niektórych miejscach powyższej definicji zostawiono zadania dla Ciebie:

- ✓ TODO 1 – metoda ma zwracać obwód figury,
- ✓ TODO 2 – metoda ma zwracać pole figury,
- ✓ TODO 3 – metoda ma wypisywać nazwę i parametry figury
- ✓ TODO 4 – destruktor ma drukować na konsoli zawartość obiektu poddawanego destrukcji (W celach diagnostycznych).

Zadanie 2.

W zadaniu należy stworzyć definicję klasy **`Trojkat`**, która będzie reprezentowała figurę trójkąt. Definicja nagłówkowa klasy przedstawia się następująco:

```
#include "FiguraPlaska.h"


class Trojkat : public FiguraPlaska {
    double a,b,c;
protected:
    void Wypisz(std::ostream& out) const override;
public:
    Trojkat(double a, double b, double c);

    double GetA() const;

    void SetA(double a);

    double GetB() const;

    void SetB(double b);

    double GetC() const;

    void SetC(double c);

    double Obwod() override;
    double Pole() override;

    ~Trojkat() override;
private:

};
```

Zauważ, że w przypadku trójkątów inaczej należy zdefiniować takie elementy klasy jak konstruktor oraz metody obliczające pole i obwód a także metodę wypisującą obiekt na wyjście.

Pytania:

- ✓ *jaki dostęp zdefiniowano dla pól i metod klasy Trojkat?*
- ✓ *jakie konstruktory zdefiniowano dla klasy Trojkat w powyższym przykładzie?*

W następnej części zadania należy samemu zdefiniować pozostałą część klasy w pliku *.cpp.

Pamiętaj że

- ✓ *klasy powinno się definiować w osobnych plikach.*
- ✓ *nazwy klas powinny zaczynać się dużych liter, nazwy obiektów z małych liter.*

Następnie wzorując się na poprzednich przykładach stwórz od postaw klasę `Kolo`, reprezentującą koło.

Zadanie 3.

- ✓ Stwórz funkcję *main* i zaprezentuj w niej działanie stworzonych wcześniej poszczególnych elementów klas: metod, konstruktorów, destruktorów, zdefiniowanych metod oraz operatora wypisania. Zdefiniuj obiekty zarówno poprzez wskaźnik jak i zmienną lokalną.
- ✓ Stwórz fragment kodu, który będzie pokazywał że metody **Obwod**() oraz **Pole**() w zdefiniowanych klasach są wirtualne. W trakcie zajęć prowadzący może Cię poprosić o wytłumaczenie na tym przykładzie na czym polega polimorfizm. W tym celu możesz utworzyć tablicę obiektów typu bazowego. Wypełnij ją, a następnie iterując po niej wywołuj kolejno zdefiniowane postacie metody wirtualnej.
- ✓ Stwórz przykładowy fragment kodu pokazujący różnicę pomiędzy przeciążaniem (*ang. overloading*) a nadpisywaniem (*ang. overriding*) funkcji/metod.

Dodatek: Sposoby dostępu do zmiennej

W języku C++ istnieje kilka sposobów na dostęp do zmiennej. Poniżej zostaną omówione najbardziej podstawowe.

Załóżmy następującą definicję struktury:

```
typedef struct A
{
    int i;
}A;
```

Gdy utworzymy zmienną typu A możemy się do niej odwołać w następujący sposób:

- Przez wartość:

```
A a;
a.i = 5;
```

- Przez wskaźnik:

```
A a;
A* pa = &a;
pa->i = 5;
```

- Przez referencję:

```
A a;
A& refA = a;
refA.i = 5;
```

W podanych wyżej przykładach następuje zmiana wartości pola `i` w zmiennej `a`. Nowością w języku C++ (względem języka C) jest istnienie referencji. Referencja jest nazywana niekiedy aliasem na zmienną. W większości przypadków można o niej myśleć jak o stałym wskaźniku. Dlatego przy tworzeniu trzeba ją natychmiast zainicjować. Jak widać zapis dostępu do wartości zmiennej ukrytej za

referencją jest taki sam jak zapis operacji na zmiennej dostępnej bezpośrednio przez wartość (nazwę). Wynika to z tego że referencja zawsze na coś wskazuje. Toteż nie jest konieczne sprawdzanie czy jest ona poprawna, tak jak jest to w przypadku wskaźnika. Wskaźniki oraz referencje mają zastosowanie podczas wykorzystywania cechy hierarchii klas jakim jest polimorfizm. Wskaźniki oraz referencje można rzutować na wskaźniki/referencje klas bazowych.