Języki Programowania, Laboratorium 12 – zadanie realizowane w parach
Wirtualność i polimorfizm w języku C++

Wirtualność oraz polimorfizm (rozumiany jako możliwość wyboru wersji metody w trakcie działania programu) są jednymi z najważniejszych cech programowania obiektowego w języku C++. Umożliwiają one tworzenie kodu elastycznego, łatwego w rozbudowie oraz niezależnego od konkretnych typów obiektów. Celem dzisiejszego laboratorium jest zapoznanie się z zasadą działania metod wirtualnych, klas abstrakcyjnych oraz polimorfizmu.

1. Klasa bazowa: Bryla

Zaimplementować klasę **Bryla**, będącą klasą bazową dla wszystkich brył geometrycznych obsługiwanych w programie.
Klasa Bryla przechowuje następujące pola:

- float x, y, z – współrzędne położenia bryły w przestrzeni

Pola te są dostępne **wyłącznie dla klas pochodnych**.
W części publicznej klasa zawiera:

- deklarację **wirtualnych** metod: Wypisz() - wypisuje pola na ekranie; oraz Objetosc() - domyślnie metoda powinna zwracać 0.

2. Klasa pochodna: Prostopadloscian

Jako przykład konkretnej bryły zaimplementować klasę **Prostopadloscian**, dziedziczącą po klasie Bryla.
Klasa Prostopadloscian zawiera dodatkowe pola:

- double a, b, c – długości krawędzi prostopadłościanu

Współrzędne x, y, z z klasy bazowej oznaczają położenie jednego z wierzchołków bryły.
Należy zdefiniować:

- **konstruktor domyślny** (wszystkie wartości ustawione na 0),
- **konstruktor inicjalizujący** wszystkie pola klasy.

Implementacje metod:

- Wypisz() powinna wypisać:

Prostopadloscian (x, y, z, a, b, c)
gdzie każda wartość wypisana jest w osobnej linii,

- Objetosc() powinna zwrócić objętość prostopadłościanu.

3. Program testujący działanie klas

W funkcji main:

1. Stworzyć **wskaźnik (wraz z obiektem)** b1 na klasę Bryla (konstruktor domyślny).
   Wywołać Wypisz() oraz wyświetlić Objetosc().
2. Stworzyć obiekt p1 klasy Prostopadloscian (konstruktor domyślny).
   Wywołać Wypisz() oraz wyświetlić Objetosc().
3. Stworzyć **wskaźnik (wraz z obiektem)** p2 na klasę Prostopadloscian
   (x = 1, y = 2, z = 3, a = 4, b = 5, c = 6).
   Wywołać Wypisz() oraz wyświetlić Objetosc().

Następnie zadeklarować wskaźnik:
Bryla* wsk;
i przypisać do niego kolejno obiekty p1 oraz p2, po czym wywołać:
wsk->Wypisz();
Program skompilować i uruchomić.

Dodatkowo należy:

- zadeklarować **wirtualny destruktor** w klasie Bryla

Zapamiętaj: Jeśli klasa deklaruje jedną ze swoich funkcji jako virtual, wówczas jej destruktor deklarujemy także jako virtual!
Skoro w klasie deklarujemy jakąś funkcję wirtualną, to znaczy, że na obiekty klas pochodnych zamierzamy czasem mówić jak na obiekty klasy podstawowej, co przy późniejszym niszczeniu obiektów mogłoby być problemem – nie zwalnialibyśmy pamięci dla niektórych składników klas pochodnych.

Eksperyment

Usunąć słowo kluczowe virtual z metod w klasie Bryla, ponownie skompilować i uruchomić program. Zaobserwować różnice w zachowaniu programu i wyjaśnić je prowadzącemu.

Podsumujmy:

Jeśli kompilator natrafi na **obiekt** danej klasy, np. Bryla, uruchamia dla niego funkcję składową z właściwej mu klasy.
Jeśli kompilator natrafi na **wskaźnik lub referencję** klasy Bryla, ma dwie możliwości wywołania funkcji składowej:
1) Skoro jest to wskaźnik do klasy Bryla, wywołuje metodę składową klasy Bryla. Jest to domyślne zachowanie kompilatora.
2) Kompilator widzi, że jest to wskaźnik do klasy Bryla, ale zamiast na ślepo sięgać do klasy Bryla *używa swojej inteligencji*: nie daje się zwieźć typem i sprawdza, na co faktycznie wskaźnik wskazuje. Wtedy może się zorientować, że wskaźnik tak naprawdę wskazuje na obiekt klasy pochodnej Prostopadloscian, zatem **orientując się według typu obiektu** uruchamia funkcję składową dla prostopadłościanu. Takie zachowanie wymusza słowo kluczowe virtual przed nazwą funkcji składowej.

4. Metoda czysto wirtualna – klasa abstrakcyjna
Zmienić deklarację metody w klasie bazowej na czysto wirtualną:
virtual double Objetosc() = 0;
Poprawić kod tak, aby ponownie się kompilował.
Pytanie:
Dlaczego część wcześniej napisanego kodu nie będzie się teraz kompilować?
Wskazówka: Klasa zawierająca co najmniej jedną metodę czysto wirtualną jest **klasą abstrakcyjną**.

Po co tworzyć klasy abstrakcyjne?
Czasem mamy jakiś obiekt, który łączy cechy kilku innych (jak np. nasza klasa `Bryła`) ale sam nie przedstawia swoją istotną żadnego *konkretnego* obiektu. Mamy kule, prostopadlosciany i inne – jak policzyć pole powierzchni niezidentyfikowanej *bryły*? Klasa abstrakcyjna jest klasą jakby „niedokończoną". Jej dokończenie realizowane jest przez klasy pochodne.

5. Funkcja globalna wykorzystująca polimorfizm

Należy napisać **funkcję globalną**, która będzie operować na **referencji do klasy bazowej Bryla**, a jej zachowanie będzie zależne od rzeczywistego typu obiektu przekazanego do funkcji.

5.1. Zdefiniować funkcję globalną:
void WypiszIObjetosc(Bryla* b);
5.2. Funkcja powinna:
- wywołać metodę Wypisz(),
- wypisać na ekran wartość zwracaną przez Objetosc().

5.3. W programie testującym:
- utworzyć co najmniej dwa obiekty różnych klas pochodnych,
- przekazać je do funkcji WypiszIObjetosc,
- zaobserwować, że wywoływane są odpowiednie wersje metod wirtualnych.

Ten element zadania pokazuje, że **jedna funkcja może poprawnie obsługiwać różne typy obiektów**, o ile operuje na referencji lub wskaźniku do klasy bazowej.

6. Kolekcja brył – tablica wskaźników

Celem poniższego ćwiczenia jest zaprezentowanie **polimorfizmu w połączeniu z dynamiczną alokacją pamięci oraz kolekcją obiektów**.

6.1. W programie głównym należy zadeklarować kontener:
std::vector<Bryla*> bryly;

6.2. Do kontenera dodać dynamicznie utworzone obiekty:
bryly.push_back(new Prostopadloscian());
bryly.push_back(new Prostopadloscian(1,2,3,4,5,6));

6.3. Przejść po wszystkich elementach kontenera i dla każdego:
- wywołać Wypisz(),
- wyświetlić objętość brył.

6.4. Zwolnić pamięć:
```
for (Bryla* b : bryly)
   delete b;
```
Zadanie pokazuje też sens stosowania wirtualnego destruktora.

(zadanie dodatkowe na 3 stronie)

7. Rozszerzenie hierarchii klas – zadanie dodatkowe

Celem zadania jest pokazanie, że **dodanie nowej klasy pochodnej nie wymaga modyfikacji istniejącego kodu korzystającego z polimorfizmu**.

7.1. Zaimplementować klasę Kula, dziedziczącą po klasie Bryla.
Klasa powinna zawierać:

- pole double r – promień kuli,
- konstruktor inicjalizujący wszystkie pola,
- implementacje metod:
  - Wypisz(),
  - Objetosc().

7.3. W programie testującym:

- utworzyć obiekt klasy Kula,
- dodać go do wcześniej utworzonego kontenera vector<Bryla*>,
- wywołać na nim metody Wypisz() i Objetosc() poprzez wskaźnik do klasy Bryla.

7.4. Zaobserwować, że:

- nie było konieczności zmiany kodu pętli obsługującej bryły,
- nowe bryły są automatycznie obsługiwane przez istniejące funkcje.

Zadanie ilustruje jedną z kluczowych zalet programowania obiektowego: **łatwość rozbudowy programu bez ingerencji w już istniejący kod**.