Programowanie obiektowe

Laboratorium III

Sprytne wskaźniki, Funkcje wirtualne, zasada Otwarte-Zamknięte

Sprytne wskaźniki

W języku C++ istnieje możliwość dynamicznego tworzenie obiektów za pomocą operatora `new` oraz zwalniania tak przydzielonej pamięci za pomocą operatora `delete`. Wadą stosowania operatora new jest to, że pamięć przydzielona za jego pomocą musi być explicite zwolniona za pomocą operatora delete. Może być to powodem występowania licznych błędów a w szczególności tak zwanego wycieku pamięci, który ma miejsce gdy przydzielona pamięć nie zostaje zwolniona. Z tych powodów we współczesnym języku C++ bezpośrednie użycie operatora `new` jest zdecydowanie nie rekomendowane. Zamiast tego, od standardu C++11 mamy dostępne w bibliotece standardowej dostępne tak zwane sprytne wskaźniki. Sprytny wskaźnik jest obiektem, który przechowuje "surowy" wskaźnik do obiektu. Dzięki temu, że "surowy" wskaźnik jest opakowany w taki obiekt uzyskujemy możliwość automatycznego zwalniania pamięci w momencie gdy obiekt sprytnego wskaźnika przestaje być dostępny w programie. W ten sposób rozwiązujemy problem wycieku pamięci. Na ogół w języku C++ wykorzystujemy sprytny wskaźnik o nazwie *Shared Pointer*, który w jest zdefiniowany w pliku nagłówkowym `memory` jako obiekt o nazwie `shared_ptr`. Do tworzenia obiektów za pomocą wskaźnika `shared_ptr`, na ogół używamy funkcji `make_shared<>()`. Poniżej przykłady tworzenia i użycia sprytnego wskaźnika `shared_ptr`.

```
#include <memory>
shared_ptr<int> p1 = make_shared<int>(); // odpowiada new int;
*p1=78; // dereferencja wskaźnika
shared_ptr<int[]> p2(new int[50]); // odpowiada new int[50]
p2[0]=10; // odwołanie do elementu o indeksie 10 w tablicy
shared_ptr<Object> p3 = make_shared<Object>("Lamp"); // odpowiada new Object("Lamp")
p3->turn_on(); // wywołanie metody turn_on() z obiektu wskazywanego przez p3
Object* p4=p3.get(); // pobranie surowego wskaźnika
shared_ptr<Object> p5=p3; // p4 wskazuje na ten sam obiekt co p3
if (p3==p5) { … } // sprawdzenie czy p3 i p4 wskazują na ten sam obiekt
shared_ptr<Object> tab[3]; // tablica wskaźników do obiektów klasy Object
```

Jeśli chcemy przekazać sprytny wskaźnik do funkcji to najprościej zrobić to przekazując wskaźnik przez wartość, na przykład można zdefiniować funkcję następująco: `void use_shared_ptr_by_value(shared_ptr<int> sp);`.

Polimorfizm

Polimorfizm (wielopostaciowość) jest to cecha programowania obiektowego, umożliwiająca różne zachowanie tych samych metod wirtualnych (funkcji wirtualnych) w czasie wykonywania programu. W języku C++ możemy korzystać z tego mechanizmu za pomocą metod wirtualnych. Dzięki niemu mamy pełną kontrolę nad wykonywanym programem, nie tylko w momencie kompilacji (wiązanie statyczne) ale także podczas działania programu (wiązanie dynamiczne) – niezależnie od różnych wyborów użytkownika. Podręcznik 4.11.

Metody wirtualne

Zacznijmy od krótkiego przypomnienia:

1. Podczas dziedziczenia obiekt klasy pochodnej może być wskazywany przez wskaźnik typu klasy bazowej.
2. Typem statycznym obiektu wskazywanego przez wskaźnik jest typ tego wskaźnika. Typem dynamicznym obiektu wskazywanego przez wskaźnik jest typ na jaki dany wskaźnik wskazuje. Dwa powyższe fakty dobrze zobrazuje poniższy przykład:

```
class Bazowa {
public:
	int a;
};

class Pochodna : public Bazowa {
public:
	int b;
};

int main()
{
	// typ statyczny: Bazowa
	// typ dynamiczny: Pochodna
	Bazowa *bazowa = make_shared<Pochodna>().get();

	// typ statyczny: Pochodna
	// typ dynamiczny: Pochodna
	Pochodna *pochodna = make_shared<Pochodna>().get();

	return 0;}
```

Dzięki tym informacjom możemy napisać zwięzłą definicję metody wirtualnej: metoda wirtualna jest to funkcja składowa klasy poprzedzona słowem kluczowym virtual, której sposób wywołania zależy od typu dynamicznego zmiennej, a nie od typu statycznego.

Zilustrujemy to przykładem. Zaczniemy od przypadku, bez użycia metod wirtualnych i polimorfizmu. Aby funkcje zostały przesłonięte muszą mieć taką samą nazwę, argumenty oraz typ zwracany:

```
class Bazowa {
public:
	void fun() { cout << "Bazowa \n"; }
};

class Pochodna : public Bazowa {
public:
	void fun() { cout << "Pochodna \n"; }
};

int main()
{

	Bazowa *bazowa = make_shared<Pochodna>().get();
	Pochodna *pochodna = make_shared<Pochodna>().get();
```

```
	bazowa->fun();  //wyświetli: bazowa
	pochodna->fun();//wyświetli: pochodna

	bazowa = make_shared<Bazowa>().get();

	bazowa->fun();  //wyświetli: bazowa

	return 0;
}
```

Ponieważ w powyższym przypadku nie używamy metod wirtualnych, zatem to, która metoda zostanie wywołana zależy od typu wskaźnika na obiekt. Jest to wspomniane wcześniej wiązanie statyczne. Kompilator już podczas kompilacji programu wie, jakiego typu statycznego są obiekty i jakie metody mają zostać wywołane. Dzięki dodaniu do naszego kodu metod wirtualnych, uruchomimy mechanizm polimorfizmu. Wczesne wiązanie statyczne nie będzie miało w takim przypadku zastosowania, ponieważ to która funkcja zostanie wywołana będzie zależało od późnego wiązania dynamicznego.

```
class Bazowa {
public:
	virtual void fun() { cout << "Bazowa \n"; }
};

class Pochodna : public Bazowa {
public:
	void fun() { cout << "Pochodna \n"; }
};

int main()
{
	Bazowa *bazowa = make_shared<Pochodna>().get();
	Pochodna *pochodna = make_shared<Pochodna>().get();

	bazowa->fun();  //wyświetli: pochodna
	pochodna->fun();//wyświetli: pochodna

	return 0;
}
```

W tym przypadku wywołania metod są zależne od typu dynamicznego. Słowo virtual wystarczy dodać jedynie w klasie bazowej, nie ma konieczności powtarzania go w klasach pochodnych.

Powstaje pytanie do czego potrzebny jest polimorfizm oraz metody wirtualne? Bez używania polimorfizmu, programista musiał już na etapie pisania programu, wiedzieć jak będzie się on zachowywał. To za sprawą wczesnego wiązania, które musi być dostarczone kompilatorowi w momencie kompilacji i linkowania. W przypadku użycia polimorfizmu dostajemy nieograniczone możliwości projektowania aplikacji, gdzie zachowanie programu może się ciągle zmieniać.

Przykład zastosowania polimorfizmu

Posiadamy klasę bazową `Pojazd` oraz trzy klasy pochodne: `Samochod`, `Rower` i `Rolki`. Wszystkie klasy mają zdefiniowaną metodę `zatrzymaj()` odpowiedzialną za zatrzymanie pojazdu danego typu. Tworzymy tablicę wskaźników na obiekty klasy `Pojazd`. Dzięki użyciu polimorfizmu możemy zatrzymać wszystkie pojazdy w jednej pętli.

```
#include <iostream>
#include <cstdlib>
```

```
using namespace std;

class Pojazd {
public:
      virtual void zatrzymaj() {
        cout << "zatrzymuje pojazd... ale jaki?\n";
      }
};

class Samochod : public Pojazd {
public:
      void zatrzymaj() {
        cout << "zatrzymuje samochod\n";
      }
};

class Rower : public Pojazd {
public:
      void zatrzymaj() {
        cout << "zatrzymuje rower\n";
      }
};

class Rolki : public Pojazd {
public:
      void zatrzymaj() {
        cout << "zatrzymuje rolki\n\n";
      }
};

int main()
{
    shared_ptr<Pojazd> tablica[3];

    tablica[0] = make_shared<Samochod>();
    tablica[1] = make_shared<Rower>();
    tablica[2] = make_shared<Rolki>();

    for (int i = 0; i<3; i++) {
         tablica[i]->zatrzymaj();
    }

      return 0;
}
```

W powyższym przykładzie o wywołaniu odpowiedniej przesłoniętej metody zadecydowało późne wiązanie. Uzyskaliśmy ten efekt dzięki zadeklarowaniu funkcji wirtualnej w klasie bazowej. W ten sposób zadziałał polimorfizm. Gdybyśmy usunęli słowo `virtual`, trzy razy zostałaby wywołana funkcja klasy bazowej – zostałby trzy razy wyświetlony napis: `zatrzymuje pojazd… ale jaki?`.

Nie należy przesadzać ze stosowaniem metod wirtualnych gdyż polimorfizm kosztuje. Gdy używamy polimorfizmu program aż do czasu uruchomienia nie wie jak będzie działał, ponieważ obiekt na jaki wskazuje wskaźnik może często zmienić się, zależnie od działania użytkownika. Obiekt klasy, która posiada metody wirtualne zajmuje więcej miejsca w pamięci komputera niż ten sam obiekt bez metod wirtualnych. Związane jest to ze sposobem wywołania metody wirtualnej w czasie działania programu, który jest bardziej złożony niż w przypadku metod nie wirtualnych. Kolejnym efektem ubocznym jest to, że czas wywołania metody wirtualnej jest dłuższy niż standardowej metody. Z tych powodów należy metody wirtualne stosować jedynie w przypadku gdy jest to konieczne.

Wirtualny destruktor

Z uwagi na konieczność zapewnienia poprawnej kolejności wywołania destruktorów klas potomnych, w języku C++ na ogół każdy destruktor jest wirtualny.

Zasada Otwarte-Zamknięte

Reguła Otwarte-Zamknięte, (ang. *OCP - Open-Closed Principle*) została zdefiniowana w roku 1988 przez Bertranda Meyera. Mówi ona o tym, że element oprogramowania (np. klasa) powinien być otwarty na rozbudowę, ale zamknięty na modyfikację. Dzięki zastosowaniu zasady otwarte-zamknięte rozwijane systemy informatyczne mogą być kompatybilne wstecz. Reguła ta jest szczególnie ważna, jeżeli chcemy tworzyć oprogramowanie, które będzie miało więcej niż jedną wersję. Jeśli podczas tworzenie oprogramowania zastosujemy tą zasadę, to wydając kolejne wersje, nie będziemy psuli poprzednich. Z drugiej strony, jeśli podczas tworzenia kolejnej wersji programu wprowadzimy zmiany, przez które zostanie wygenerowane dużo błędów w modułach zależnych, to znak że prawdopodobnie nie stosujemy się do zasady otwarte-zamknięte.

W przypadku klas, zgodnie z zasadą otwarte-zamknięte, klasa powinna być zamknięta na modyfikacje tego, co już w niej istnieje, ale otwarta na rozszerzenia. Oznacza to, że powinniśmy projektować klasy w taki sposób aby możliwe było dodanie do klasy nowych funkcjonalności bez konieczności zmiany istniejącego już kodu klasy. Zasada otwarte-zamknięta oznacza również to, że, na przykład, jeśli mamy jakąś metodę, która jest używana w innych częściach systemu, to nie powinniśmy jej już modyfikować w kolejnych wersjach programu, bo w ten sposób można łatwo coś zepsuć w innych miejscach systemu. Skutkiem tego może brakować kompatybilności wstecznej z naszymi metodami, które już udostępniamy.

Zasada otwarte-zamknięte jest szczególnie istotna dla programistów, którzy tworzą biblioteki programistyczne, z których korzystają inni programiści. Wyobraźmy sobie sytuację, że stworzyliśmy klasę, na przykład, `Calculator`, która coś oblicza i została ona udostępniona innym programistom w postaci biblioteki łączonej dynamicznie (np. .dll). Programiści Ci zaczęli używać klasy `Calculator` i mają w swoich programach dużą liczbę wywołań tej klasy. W kolejnej wersji do konstruktora klasy `Calculator` dodaliśmy nowy parametr. Wtedy programiści, którzy używają naszej biblioteki, chcąc używać najnowszą wersję, muszę zmienić wszystkie swoje wywołania tej klasy. Na pewno będzie to dla nich bardzo czasochłonna i niezbyt przyjemna praca.

Na pierwszy rzut oka może wydawać się, że jest tu jakaś nieścisłość. W jaki sposób mamy rozwijać i zmieniać zachowanie już istniejących klas bez ich aktualizacji? Jest to możliwe, a odpowiedzią na to pytanie jest zastosowanie abstrakcji. Rozważmy poniższy kod:

```
class AreaCalculator {
public:
	static double CalculateRectanglesAreas(vector<Rectangle>& rectangles) {
	double area = 0;
	for (auto rectangle : rectangles)
		area += rectangle.Height * rectangle.Width;
	return area;
	}

	static double CalculateTrianglesAreas(vector<Triangle>& triangles) {
	double area = 0;
	for (auto triangle : triangles)
		area += (triangle.Base * triangle.Height) / 2;
	return area;
	}
};
```

W powyższej klasie, mamy dwie metody przeznaczone do obliczania sumy pól figur przekazanych w wektorze jako argument funkcji. Oddzielna metoda istnieje dla trójkątów i dla prostokątów. Gdy zechcemy dodać kolejną figurę, będziemy musieli zmodyfikować klasę i dodać kolejną metodę. Przy okazji może okazać się, że złamiemy inną

ważną zasadę inżynierii oprogramowania o nazwie DRY (ang. *Don't Repeat Yourself* – nie powtarzaj się), ponieważ metody te robią w zasadzie to samo, a są oddzielone od siebie tylko ze względu na typ parametru.
Zgodnie z zasadą otwarte-zamknięte, w klasie powinna istnieć tylko jedna metoda do obliczania sumy pól dowolnej figury i w ten sposób być zamknięta na dodanie kolejnej metody po dodaniu nowej figury do systemu. Zamknięcie klasy na dodanie kolejnej metody do obliczania sumy pól nowej figury oznacza, że taka nowa metoda po prostu nie jest konieczna. Możemy to osiągnąć poprzez zastosowanie dodatkowej klasy abstrakcyjnej. Zobaczmy jak można to osiągnąć w naszym przykładzie.

Zacznijmy od utworzenia abstrakcyjnej klasy bazowej o nazwie `Shape`.

```
class Shape {
public:
	virtual double CalculateArea() = 0;
};
```

Następnie utwórzmy klasę `Rectangle` dziedziczącą z klasy `Shape`.

```
class Rectangle : public Shape {
private:
	double Height;
	double Width;
public:
	double getHeight() {
	 return Height;
	}
	void setHeight(double height) {
	 Height = height;
	}
	double getWidth() {
	 return Width;
	}
	void setWidth(double width) {
	 Width = width;
	}
	double CalculateArea() override {
	 return Width * Height;
	}
};
```

Następnie w klasie `AreaCalculator`, modyfikujemy metodę `CalculateArea()` tak aby operowała ona nie na konkretnym typie `Rectangle` ale na abstrakcyjnym typie `Shape`. W ten sposób, metoda `CalculateArea()` obliczy sumę pól dla każdego typu figury.

```
class AreaCalculator {
public:
	double CalculateArea(vector<Shape*> shapes) {
	double area = 0;
	for (auto shape : shapes) {
		area += shape->CalculateArea();
	}
	return area;
	}
};
```

Po tych modyfikacjach, gdy przekażemy do metody `CalculateArea()` jako argument wektor obiektów typu `Rectangle` lub `Triangle`, to metoda ta będzie poprawnie obliczać sumę pól w zależności od rodzaju przekazanej figury. Teraz gdy przyjdzie nam dodać nową figurę, nie będziemy musieli dodawać nowej metody do klasy `AreaCalculator`, która będzie przyjmować wektor obiektów nowego kształtu. Wystarczy, że przy tworzeniu nowej figury stworzymy nową klasę dziedziczącą po klasie `Shape` i w podklasie zaimplementujemy odpowiednią dla nowego kształtu metodę `CalculateArea()`. W ten sposób zamknęliśmy klasę `AreaCalculator` na modyfikacje, a dodatkowo dodaliśmy możliwość obliczania pól dla nowych figur bez edycji już istniejącego kodu.

Zadania

Zadanie 1

1. Stwórz klasę o nazwie `Number` reprezentującą dowolną liczbę. Klasa powinna posiadać jedynie publiczną wirtualną metodę virtual `void print(ostream& out) const;`, która wypisuje na strumień wyjściowy out wartość liczby. Następnie stwórz operator wyjścia dla tej klasy jako standardową funkcję poza klasą. Operator wyjścia ma wykorzystywać wirtualną metodę `print()` w celu wyświetlenia wartości liczby na strumieniu wyjściowym.

2. Stwórz klasę `Complex` reprezentującą liczbę zespoloną i która dziedziczy z klasy `Number`. Umieść w klasie `Complex` odpowiednie pola, konstruktor, gettery i settery. Nadpisz metodę wirtualną `print()` odziedziczoną z klasy `Number` i która poprawnie wypisze liczbę zespolona na strumieniu wyjsciowym `out`.

3. Stwórz klasę `Real` dziedziczącą z klasy `Complex` i reprezentującą liczbę rzeczywistą. Stwórz odpowiedni konstruktor dla klasy `Real`. Nadpisz również metodę print odziedziczoną z klasy `Complex` i zaimplementuj w niej poprawne wypisanie liczby rzeczywistej na strumieniu wyjściowym.

4. Stwórz klasę `Rational` dziedziczącą z klasy `Number` i reprezentującą liczbę wymierną. Umieść w klasie odpowiednie pola reprezentujące licznik oraz mianownik liczby. Dodaj również odpowiedni konstruktor. Podobnie jak w poprzednich przypadkach nadpisz metodę print i zaimplementuj w niej wyświetlanie liczby wymiernej w postaci licznik/mianownik (np. 2/5).

5. W funkcji `main()` utwórz za pomocą wskaźnika `shared_ptr` przynajmniej po jednym obiekcie każdej z klas `Number`, `Complex`, `Real` oraz `Rational`. Przypisz utworzone obiekty do wskaźników klasy `Number`. Wypisz na ekranie wartość każdej liczby za pomocą operatora wyjścia. Zaobserwuj jak zmieni się wynik działania programu w przypadku gdy zostanie usunięte słowo kluczowe virtual przed metodą print w klasie `Number`.

Zadanie 2

1. Załóżmy, że tworzysz system zarządzania treścią (ang. *Content Management System*). System będzie zarządzał następującymi typami plików:
    a. PDF (twórca, opis, rozmiar pliku),
    b. dokumenty Word (twórca, opis, rozmiar pliku),
    c. obrazy(twórca, opis, wymiary obrazu, rozmiar pliku),
    d. pliki wideo(twórca, opis, wymiary obrazu, czas trwania, rozmiar pliku)

    Stwórz odpowiednią hierarchię klas. Dla każdego typu pliku powinna istnieć funkcja sprawdzająca czy rozmiar pliku nie przekracza przyjętego progu. Dodatkowo, w przypadku obrazów oraz plików wideo powinna istnieć możliwość sprawdzenia czy wymiary obrazu nie przekraczają zadanych wartości. W funkcji `main()` napisz program demonstrujący przykładowe użycie powyższych klas.