Laboratorium 3

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.

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 = new Pochodna();

	// typ statyczny: Pochodna
	// typ dynamiczny: Pochodna
	Pochodna *pochodna = new Pochodna();

	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 = new Pochodna();
	Pochodna *pochodna = new Pochodna();

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

	bazowa = new Bazowa();

	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 = new Pochodna();
	Pochodna *pochodna = new Pochodna();

	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()
{
	Pojazd **tablica = new Pojazd*[3];

	tablica[0] = new Samochod();
	tablica[1] = new Rower();
	tablica[2] = new 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...?.

Nie należy przesadzać

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 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.

Zadania

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 `Real` 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 operatora `new` 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`.