Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania): `std::cout << "Zadanie X"<< std::endl;`,
4. każde zadanie umieść w oddzielnej funkcji (w niej dopiero należy odwoływać się do zaimplementowanych funkcji i klas),
5. zaimplementuj menu wyboru zadania, a następnie wykorzystując pętle **do–while** oraz konstrukcję **switch** wykonaj odpowiedni fragment kodu,
6. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie),
7. w zadaniach polegających na zaprojektowaniu klasy należy utworzyć jej instancję i wykorzystać zaimplementowaną funkcjonalność.

Wprowadzenie

Przeciążanie operatorów

```
class Point {
private:
    double x, y;

public:
    Point() : Point(0.0, 0.0) {}
    Point(double x, double y) : x(x), y(y) {}

    // overloading plus (+) operator
    Point operator+(const Point &other ) {
        return Point(x + other.x , y + other.y );
    }
}

// overloading left shift («) operator
std::ostream& operator << (std::ostream& stream, const Point &p) {
```

```
    stream << p.x << " " <<  p.y;
    return stream;
}
```

Przeciążanie operatorów wykorzystywane jest w celu minimalizacji redundancji kodu oraz zwiększenia jego czytelności. Załóżmy, że program ma na celu dodanie do siebie trzech punktów.

```
    Point p1, p2, p3, wynik;
```

Standardowa implementacja może wyglądać następująco:

```
    wynik = dodaj(p1, dodaj(p2, p3));
```

Z wykorzystaniem przeciążonego operatora dodawania (linia 10) kod można zmodyfikować jak poniżej:

```
    wynik = p1 + p2 + p3;
```

Przeciążanie operatora przekierowania ((linia 16) umożliwia w intuicyjny sposób odwoływanie się na przykład do strumienia standardowego `std::cout`. Jeżeli zadanie polega na wyświetleniu zawartość instancji klasy można wykorzystać operator <<:

```
    Point p1;
    cout << p1 << std::endl;
```

Uwaga! Jeżeli lewym argumentem jest obiekt klasy, której nie możemy modyfikować (np. `cout`) to odpowiedni **dwuargumentowy** należy umieścić **poza** naszą klasą (która jest drugim argumentem operatora). W przypadku, gdy lewym argumentem operatora jest nasza klasa i mamy możliwość jej modyfikacji to wykorzystujemy wersję **jednoargumentową** operatora **wewnątrz** naszej klasy.

Więcej informacji na temat przeciążania operatorów można znaleźć tutaj: `http://en.cppreference.com/w/cpp/language/operators`.

Konstruktory

Domyślnie, każda klasa w C++ posiada następujące elementy (tzn., jeśli nie zostaną one zdefiniowane przez programistę to automatycznie zadeklaruje je kompilator):

- Domyślny konstruktor bezargumentowy (7) – nie robi nic; brak konstruktora domyślnego w przypadku zdefiniowania jakiegokolwiek innego konstruktora.
- Konstruktor kopiujący (8) – kopiuje wszystkie atrybuty.
- Destruktor (10) – nie robi nic.
- Operator przypisania (12) - kopiuje wszystkie atrybuty.

```
class DefaultExample {
private:
    int a;
    char b;

public:
    DefaultExample() {}
    DefaultExample(const DefaultExample& other) :
        a(other.a), b(other.b) {}
    virtual ~DefaultExample() {}

    DefaultExample& operator=(const DefaultExample& other) {
        if (this != &other) {
            a = other.a;
            b = other.b;
        }
        return *this;
    }
}
```

Przykładowo:

```
{
    DefaultExample e1; // wykorzystujemy konstruktor domyślny
    DefaultExample e2 = e1; // wykorzystujemy konstruktor kopiujący
    e1 = e2; // wykorzystujemy operator przypisania
}
```

Zadania

Zadanie 1

Zmodyfikuj podaną na listingu 1 implementację szablonu klasy `Vector` (reprezentującą wielowymiarowy wektor) przeciążając następujące operatory:

- operator indeksowania `[]` będący odpowiednikiem metody `at` (linia 20 oraz 24),
- operator dodawania + będący odpowiednikiem metody `add` (linia 30),
- operator dodawania – (analogicznie do operatora +),
- operator równości == będący odpowiednikiem metody `equals` (linia 39),
- operator nierówności =! (odwrotny w stosunku do operatora ==).
- operator wstawiania << pozwalający na wstawianie danych do strumienia wyjściowego (np. `std::cout`).

Listing 1: Szablon klasy Vector.

```
#pragma once

template <typename T>
class Vector {
private:
    size_t dimensions;
    T* data;

public:
    Vector<T>(size_t dimensions) :
        dimensions(dimensions), data(new T[dimensions]()) {}
    virtual ~Vector<T>() {
        if (data) delete[] data;
    }

    size_t size() const {
        return dimensions;
    }

    T& at(size_t index) {
        return data[index];
    }

    const T& at(size_t index) const {
        return data[index];
    }
};

template<typename T>
Vector<T> add(const Vector<T>& a, const Vector<T>& b) {
    Vector<T> c(a.size());
    for (size_t i = 0; i < a.size(); i++) {
        c[i] = a[i] + b[i];
    }
    return c;
}

template<typename T>
bool equals(const Vector<T>& a, const Vector<T>& b) {
    if (&a == &b) return true;
    if (a.size() != b.size())
        return false;
    for (size_t i = 0; i < a.size(); i++) {
        if (a[i] != b[i]) return false;
    }
    return true;
}
```

Dołącz do własnego projektu załączony plik *Vector.hpp* lub skorzystaj z kodu na listingu 1. Następnie przetestuj swoją implementację z wykorzystaniem poniższego kodu:

```
#include <iostream>
#include "Vector.hpp"
using namespace std;

int main() {
    typedef int T;
    const size_t d = 10;

    Vector<T> v1(d);
    for (size_t i = 0; i < d; i++) {
        v1[i] = (T) i;
    }
    cout << "Wektor v1: " << v1 << endl;

    Vector<T> v2 = v1 + v1;
    cout << "Wektor v2: " << v2 << endl;

    cout << "v1 == v1: " << (v1 == v1 ? "TAK" : "NIE") << endl;
    cout << "v1 != v1: " << (v1 != v1 ? "TAK" : "NIE") << endl;
    cout << "v1 == v2: " << (v1 == v2 ? "TAK" : "NIE") << endl;
    cout << "v1 != v2: " << (v1 != v2 ? "TAK" : "NIE") << endl;

    Vector<T> v3(5);
    cout << "Wektor v3: " << v3 << endl;
    cout << "v1 == v3: " << (v1 == v3 ? "TAK" : "NIE") << endl;

    v3 = v1;
    cout << "Wektor v3: " << v3 << endl;
    cout << "v1 == v3: " << (v1 == v3 ? "TAK" : "NIE") << endl;

    getchar();
    return 0;
}
```

Zadanie 2

Po uruchomieniu przykładu z poprzedniego zadania przygotowany program najprawdopodobniej nie zakończył się prawidłowo. Wynika to z zastosowania domyślnego konstruktora kopiującego oraz operatora przypisania. Popraw definicję klasy `Vector` dodając własną implementację funkcji domyślnych. W tym celu zaimplementuj:

- konstruktor kopiujący (`Vector<T>(const Vector<T>&);`), oraz
- operator przypisania = (`Vector<T>& operator=(const Vector<T>&);`).

Konstruktor kopiujący powinien kopiować wszystkie atrybuty referencyjnej instancji (wskaźnik na tablicę oraz jej rozmiar). Zauważ, że dla rozmiaru tablicy wystarczy jedynie ustawić zmienną `dimensions` odpowiednią wartością. W przypadku samej tablicy przechowującej dane (`data`) konieczne jest natomiast utworzenie nowej tablicy oraz skopiowanie do niej wszystkich wartości.

Operator przypisania oprócz kopiowania wszystkich atrybutów powinien również zwalniać pamięć po poprzedniej tablicy (w przeciwnym wypadku mogłoby dojść do wycieku pamięci). Na koniec, funkcja powinna zwracać wskaźnik do obiektu (`*this`).

Po przygotowaniu odpowiednich definicji przetestuj ponownie swoją implementację z wykorzystaniem kodu z poprzedniego zadania. Korzystając z debuggera lub wypisując odpowiednie komentarze w konsoli sprawdź, w których momentach wywoływane są:

- główny konstruktor (`Vector<T>(size_t)`),
- konstruktor kopiujący, oraz
- operator przypisania.

Napisz w komentarzach swoje spostrzeżenia.

Dodatkowe informacje:

- Konstruktory przenoszące i przenoszące operatory przypisania - `https://msdn.microsoft.com/library/dd293665.aspx`

Zadanie 3

Przygotuj implementację klasy `ShapeContainer` z poprzednich laboratoriów. Zaimplementuj przeciążenie operatora `operator >>`, które będzie pełniło funkcję dodawania nowych figur do bazy. Deklaracja tej metody może mieć np. następującą postać:

```
ShapeContainer& operator >>(Shape* shape);
```

Wskazówka metoda operatora `>>` powinna zwracać obiekt na którym jest ona wywoływana. W przeciwnym razie nie będzie można łączyć łańcuchowo tego operatora.

Zadanie 4*

Wykorzystując przeciążenie operatora `()` utwórz obiekty typu *callable* (zachowuje się jak funkcja, ale posiada własny stan):

1. klasa `Add`
2. klasa `Mul`
3. klasa `Biggest`.

Przykład wykorzystania:

```
Add add;
Mul mul;
Biggest biggest;
for (auto x : vec) {
    add(x);
    cout << "Aktualny iloczyn " << mul(x);
    biggest(x);
}

cout << add.value << mul.value << biggest.value << endl;
```

Zadanie 5*

Wykorzystując przeciążenie operatora indeksowania `[]` rozszerz funkcjonalność klasy do obsługi bazy danych osobowych studentów (z laboratorium nr 2). Operator indeksowania powinien umożliwiać wyszukiwanie studentów o zadanym numerze indeksu. Np.:

```
BazaStudentow baza;

// uzupełnij bazę przykładowymi danymi
wypelnijBaze(baza);

// pobierz oraz wyświetl dane studenta o numerze indeksu 123456
Student s = baza[123456];
cout << s << endl;
```

Przetestuj działanie operatora na przykładowych danych.