Języki i paradygmaty programowania 2

Laboratorium 4. Przeciążenie operatorów. Funkcje zaprzyjaźnione. Generator liczb losowych. Konstruktor konwertujący. Funkcje konwersji.

1. Przygotuj klasę Vector reprezentującą dwuwymiarowy wektor(x oraz y). Przygotuj konstruktor domyślny, sparametryzowany oraz destruktor. Przygotuj metody umożliwiające wyświetlenie składowych wektora – void displayX() oraz void displayY(), a także metodę void display() wyświetlającą obie składowe. Przygotuj metody zwracające wartości poszczególnych składowych getX() oraz getY(). Przygotuj nagłówek klasy w pliku vector.h a implementację metod w vector.cpp

2. Przygotuj metodę Vector addVector(Vector vect) umożliwiającą dodawanie dwóch wektorów. Ta metoda powinna zwrócić nowy wektor utworzony w tej metodzie, Co może się stać jeżeli nagłówek funkcji będzie wyglądał w sposób następujący Vector& addVector(Vector vect)

3. Funkcję z zadania nr 2 możesz zrealizować poprzez mechanizm przeciążenia operatorów. Jest to kolejny przykład polimorfizmu statycznego. Polega on na tym, że możemy nadać znanym operatorom z języka C++ nowe znaczenie. Dzięki temu możemy przeciążyć operator + i mieć dzięki temu możliwość dodawania wektorów poprzez operator + zamiast funkcji addVector(Vector vect).
   Należy go zaimplementować w sposób następujący:

```
Vector Vector::operator+(const Vector& vect) const
{
        Vector newVector(vect.getX() + this->x, vect.getY() + this->getY());
        return newVector;
}
```

   Wywołanie będzie następujące:

```
Vector vect(1, 2);
Vector vect2(7, 7);

Vector vec3 = vect + vect2;
vec3.display();
```

Czy poprawne będzie:

```
Vector vec3 = vect + vect2 + vect2;
```

Przygotuj analogiczną interpretację dla operatora odejmowania oraz mnożenia(mnożenie przez współczynnik).

Zwróć uwagę przy mnożeniu na:

```
Vector vec4 =  vec3 * 3.0;
```

```
Vector vec4 = 3.0 * vec3;
```

Ograniczenia przeciążenia operatorów:

1) Musi przyjmować przynajmniej jeden operand typu własnego
2) Nie można przeciążyć operatora dla typów podstawowych (nie możemy zmienić sposobu dodawania liczb całkowitych).
3) Nie można zmienić pierwszeństwa operatorów.
4) Nie można tworzyć własnych symboli operatorów.
5) Nie można przeciążyć operatorów: sizeof . .* :: ?: typeid const_cast dynamic_cast reinterpret_cast static_cast

6) Operatory które można przeciążyć tylko w metodzie będącej składową klasy = () [] ->

4. Przyjaciele najważniejsi !

Dostęp do składowych klas w języku C++ możliwy jest generalnie poprzez publiczny interfejs. Jest jednak inna technika umożliwiająca dostęp do składowych klas – zaprzyjaźnienie. Zaprzyjaźnione z klasą mogą być:

a) Funkcje
b) Klasy
c) Metody klas

Zaprzyjaźniając funkcje z klasą dajemy jej pełny dostęp do wszystkich składowych klasy (taki dostęp jak ma metoda w tej klasie).

Spróbujemy ponownie rozwiązać problem mnożenia przez stałą naszego wektora – tym razem z wykorzystaniem funkcji zaprzyjaźnionej.

Dodajemy do nagłówka klasy następującą funkcję: (mimo to nie jest to składowa klasy - nie jest metodą!)

```
friend Vector operator*(double a, Vector vect)
```

Następnie przygotujemy jej implementację:

```
Vector operator*(double a, Vector vect)
{
    Vector newVector(a * vect.getX(), a * vect.getY());
    return newVector;
}
```

Jako, że ta funkcja nie jest metodą klasy nie będziemy używać operatora zasięgu :: Dodatkowo ta funkcja nie ma dostępu do wskaźnika this. Zaprzyjaźnienie może lekko naginać zasadę hermatyzacji. Należy to jednak bardziej rozumieć jako poszerzenie interfejsu oraz oczywiście nie nadużywać tej techniki.

5. Przeciążenie operatora << jako standardowy przykład przyjaźni.

Chcielibyśmy wreszcie wyświetlić nasz obiekt vect poprzez cout<<vect;

Gdybyśmy chcieli przeciążyć ten operator w klasie przeciążenie wyglądało by:

vect<<cout; //nieintuicyjne – lewy operand to musi być wywoływany obiekt

Dlatego musimy przygotować to przeciążenie w formie zaprzyjaźnienia.

Deklaracja takiej funkcji będzie wyglądać następująco:

```
friend void operator<<(ostream &os, Vector vect);
```

A implementacja:

```
void operator<<(ostream &os, Vector vect)
{
    cout << "x: " << vect.x << " " << "y: " << vect.y << endl;
}
```

Zróbmy teraz:

cout<<vect;

A spróbujmy:

cout << vec4 <<endl;

cout<<vect<<”tekst”<<endl;

Dlaczego tak: cout << vec4 <<endl; nie można ?

(cout << vec4) => void

(void<<endl) => nie da się tak

Poprawny nagłówek oraz implementacja przedstawia się więc w sposób następujący:

```
ostream & operator<<(ostream &os, Vector& vect)
{
    os << "x: " << vect.x << " " << "y: " << vect.y << endl;
    return os;
}
```

6. W sposób analogiczny spróbuj przeciążyć operator >>.
7. Dokonaj przeciążenia operatora preinkrementacji oraz postinkrementacji. Zwróć uwagę jak powinien wyglądać nagłówek funkcji dla tych operatorów:

```
Vector & operator++();
Vector operator++(int);
```

Niech implementacja tych operatorów wygląda tak, że zwiększają one wartość pierwszej składowej wektora o 1.

8. Zmodyfikuj domyślny konstruktor tak by generował wartości składowych wektora z przedziału [0,10).

Do tego celu wykorzystaj:

```
#include <ctime>
```

Następnie w funkcji main zainicjalizuj generator liczb losowych:

```
srand(time(0));
```

By wylosować jakąś liczbę użyj konstrukcji:

```
rand() % 10
```

9. Konwersje.

Załóżmy, że przygotowaliśmy konstruktor jednoparametrowy:

```
Vector::Vector(double value)
{
        x = value;
        y = value;
}
```

Dzięki niemu dopuszczalny staje się zapis:

```
Vector vec6 = 19.4;
```

Umożliwia to konwersje typu double na klasę. Jest to konwersja niejawna (implicit). Tylko konstruktory jednoparametrowe umożliwiają taką konwersję.

Możemy ten konstruktor zmodyfikować dopisując przed nim słówko explicit w nagłówku klasy.

Które z wywołań będzie poprawne:

```
Vector vec6 = 19.4;
```

Vector vec7 = (Vector)19.4;

Słówko explicit uniemożliwia niejawną konwersję – dzięki temu unikniemy sporo błędów i każda konwersja będzie świadoma.

10. Funkcje konwersji.

Spróbujmy wykonać odwrotną konwersję - > przejść z typu Vector na typ double. Załóżmy, że sposób konwersji jest taki, że skonwertowana wartość to suma składowych wektora.

Do tego celu możemy przygotować funkcję konwersji. Musi być ona składową klasy, nie definiuje typu zwracanego i nie przyjmuje argumentów.

Przykład funkcji konwersji:

```
Vector::operator double() const
{
    return x + y;
}
```

Oraz użycie:

```
Vector vec7 = (Vector)19.4;

double y = vec7;
cout << y << endl;
```

Tego laboratorium nie wysyłamy na platformę Delta ☺