Programowanie Obiektowe

Laboratorium 2 – kopiowanie i przenoszenie obiektów w C++

2019-03-06

POLITECHNIKA GDAŃSKA

Spis treści

Przygotowanie do zajęć w domu

Ćwiczenia dotyczą zagadnień związanych z zarządzaniem obiektami w C++. Poruszone zostaną m.in. tematy dotyczące dynamicznej alokacji pamięci w klasach, kopiowania obiektów, przekazywanie parametrów przez wartość i przez referencję, a także przenoszenie obiektów. Mechanizmy te są szczególnie przydatne podczas implementowania i wykorzystywania obiektów kopiowanych i przenoszonych przez wartość.

Przeanalizuj poniższy program.

```
#include <iostream>

class Vektor3d {
        double array[3]; //pole z tablicą liczb
public:
        Vektor3d (double x1,double x2,double x3){ //konstruktor
            array[0] = x1;
            array[1] = x2;
            array[2] = x3;
        }
        const double& operator[](std::size_t i) const
        {
            return array[i];
        }
};

std::ostream& operator<<(std::ostream& ostr, const Vektor3d& v)
{
    ostr << "Vector3d{";
    for(std::size_t i = 0; i < 3; i++)
    {
        if(i > 0)
            ostr << ',';
        ostr << v[i];
    }
    ostr << ')';
    return ostr;
}
int main(){
        Vektor3d v1{1,2,3}; //utworzenie obiektu
        std::cout << sizeof v1 <<std::endl;
        std::cout << "v1: " << v1 << std::endl; // Wypisanie v1
        return 0;
}
```

Zwróć uwagę na rozmiar obiektu v1. Czy wiesz skąd on się bierze? Zwróć uwagę kiedy wywołuje się konstruktor. Możesz przeanalizować przebieg programu za pomocą debugera (lub poprzez dodanie instrukcji w rodzaju cout<<”Komunikat”;)
Dzięki temu że klasa Vektor3d zawiera pole typu tablicowego na obiektach należących do tej klasy wykonywać można różne operacje bez potrzeby tworzenia implementacji. Do tych operacji należą m.in. kopiowanie (tworzenie nowych obiektów przez kopiowanie) oraz przypisywanie jednego obiektu do drugiego obiektu.
Ilustruje to rozbudowana wersja funkcji main() w kolejnym przykładzie.

Przykład 1. Ilustracja kopiowania oraz przypisywania wartości obiektów

```
int main(){
        Vektor3d v1{1,2,3}; //utworzenie obiektu
        std::cout << sizeof v1 << std::endl;
        Vektor3d v2{4,5,6}; //utworzenie obiektu
        Vektor3d v1Kopia1{v1}; //utworzenie obiektu - konstruktor
kopiujący
        Vektor3d v1Kopia2 = v1; //utworzenie obiektu - konstruktor
kopiujący
        std::cout << "v1: " << v1 << std::endl; // Wypisanie v1
        std::cout << "v2: " << v2 << std::endl; // Wypisanie v2
        v1 = v2; //przypisanie wartości
        std::cout << "v1: " << v1 << std::endl; // Wypisanie v1
        std::cout << "v1Kopia1: " << v1Kopia1 << std::endl; // Wypisanie
v1Kopia1
        std::cout << "v1Kopia2: " << v1Kopia2 << std::endl; // Wypisanie
v1Kopia2
        return 0;
}
```

W powyższym przykładzie wykorzystano zarówno kopiowanie obiektu przy tworzeniu (konstruktor kopiujący) jak również przypisywanie wartości obiektu do drugiego.
Możliwości w tym zakresie biorą się z faktu, że język C++ zapewnia domyślne mechanizmy kopiowania pól z jednego obiektu do drugiego podczas operacji przypisania i konstrukcji obiektu na bazie innego obiektu tego samego typu.
Sytuacja komplikuje się gdy kopiowanie wartości poszczególnych pól należących do obiektów nie jest wystarczające. Ilustruje to poniższy przykład.

Przykład 2. Klasa wykorzystując a dynamiczną alokację pamięci

```cpp
#include <initializer_list>
#include <iostream>
class Vector {
    double* array;
    std::size_t size;
public:
    //Parametr typu std::initializer_list pozwoli na uruchomienie
    //konstruktora z dowolną liczbą wystąpień liczb typu double.
    // Będzie to wyglądało jak inicjalizacja tablicy. Przykład:
    //Vector v1({1,2,3,4,5,5.5});
    //lub
    //Vector v1{1,2,3,4,5,5.5};
    //lub
    //Vector v1 = {1,2,3,4,5,5.5};
    //Jest to cecha C++11
    Vector(std::initializer_list<double> initList)
        : size{initList.size()}, array{new double[initList.size()]}
    {
            std::size_t i = 0;
            for(double v : initList)
            {
                array[i] = v;
                i++;
            }
    }
    std::size_t GetSize() const {
        return size;
    }
    const double& operator[](std::size_t i) const {
        return array[i];
    }
    ~Vector()
    {
        if(array != nullptr)
            delete[] array;
    }
};
std::ostream& operator<<(std::ostream& ostr, const Vector& v){
    ostr << "Vector{";
    for(std::size_t i = 0; i < v.GetSize(); i++)
    {
        if(i > 0)
            ostr << ',';
        ostr << v[i];
    }
    ostr << '}';
    return ostr;
}
int main(int argc, char** argv) {
    Vector v1 = {1,2,3,4,5,5.5};
    std::cout << "v1: " << v1 << std::endl;
    return 0;
}
```

W tym przypadku klasa Vektor zawiera pole będące wskaźnikiem na tablicę liczb typu double. Dodatkowo zawiera pole przechowujące wymiar wektora. Utworzenie i wypełnienie odpowiedniej przestrzeni w pamięci realizowane jest przez konstruktor. Ze względu na dynamiczną alokację pamięci konieczne jest jej zwolnienie w destruktorze.
Używane w przypadku klasy Vektor3d operatory przypisania i konstrukcja obiektu na podstawie kopiowania tym razem powodują kłopoty. W tym celu dodano operator[] pozwalający na modyfikacje zawartości oraz dodano komunikat do destruktora.

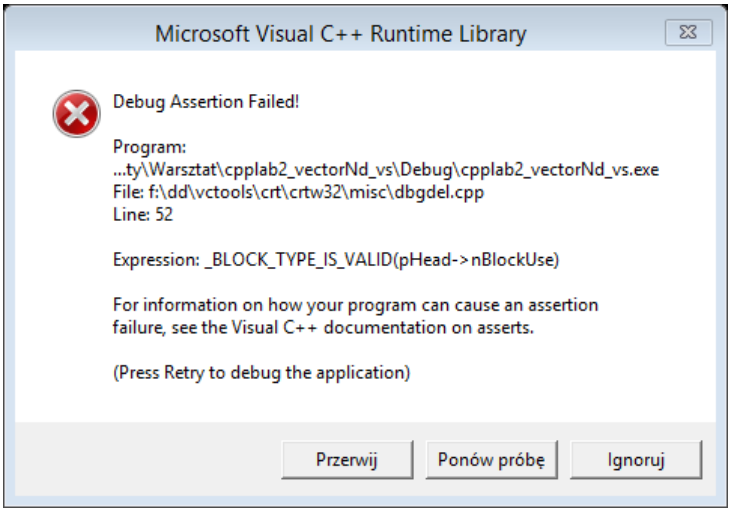
Przykład 3. Ilustracja problemów przy przypisaniu wartości w przypadku pól wskaźnikowych.

```
class Vector {
    /* ..................... */
    double& operator[](std::size_t i)
    {
        return array[i];
    }
    /* ..................... */
    ~Vector()
    {
        if(array != nullptr)
        {
            std::cout << "Releasing memory " << array << std::endl;
            delete[] array;
        }
    }
    /* ..................... */
};
    /* ..................... */

int main(int argc, char** argv) {
    Vector v1 = {1,2,3,4,5,5.5};
    std::cout << "v1: " << v1 << std::endl;
    Vector v2 = {5.5,4,3,2,1};
    std::cout << "v2: " << v2 << std::endl;
    v1 = v2; //Kłopotliwe przypisanie
    std::cout << "v1: " << v1 << std::endl;
    std::cout << "v2: " << v2 << std::endl;
    v1[0] = 100.0; //Ustawienie wartości w obiekcie v1
    std::cout << "v1: " << v1 << std::endl;
    std::cout << "v2: " << v2 << std::endl;
    return 0;
}
```

Po uruchomieniu tak zmodyfikowanego programu można zaobserwować dwa problemy:

1. Zmiana wartości wektora w jednym obiekcie (metodą za pomocą operatora []) powoduje zmianę wartości również w drugim obiekcie.
2. Destruktory obiektów v1 oraz v2, które zostają uruchomione na koniec programu próbują zwolnić tę samą pamięć. Ten problem może objawić się komunikatem w postaci:



Wszystko spowodowane jest instrukcją kłopotliwego przypisania. Podobne objawy pojawiają się w przypadku próby użycia konstrukcji przez kopiowanie.

Sprawdź co dzieje się w przypadku konstrukcji obiektu na podstawie kopiowania.

Reasumując, w języku C++ występuje kłopot w przypadku, gdy w klasie pola są typu wskaźnikowego. Występuje on szczególnie gdy stosuje się operatory przypisania i konstruktory kopiujące. Można sobie z tym poradzić na różne sposoby, m.in.:

1. Nie używać kłopotliwych operatorów i konstruktorów kopiujących.
2. Zapewnić niestandardową implementację tych mechanizmów.

O ile pierwszy sposób jest bardzo drastyczny to warto go zastosować szczególnie gdy programista jest niedoświadczony lub nie wie jak dobrze zaimplementować pożądane mechanizmy.
Jednym z rozwiązań drugiego rodzaju jest samodzielna implementacja konstruktora kopiującego (w celu zapewnienia możliwości tworzenia obiektu przez kopiowanie) jak również samodzielna implementacja operatora przypisania (w celu zapewnienia możliwości przypisywania wartości z jednego obiektu do drugiego) – przeciążenie operatora.

Przykład 4. Implementacja konstruktora kopiującego oraz operatora przypisania.

```
class Vector {
    /* ..................... */
    Vector(const Vector& other) //Konstruktor kopiujacy
        : size(other.size), array{new double[other.size]} {
        for(std::size_t i=0; i < size; i++)
            array[i] = other.array[i];
    }

    //Implementacja operatora przypisania z wykorzystaniem copy-swap
    Vector& operator=(const Vector& right) {
        Vector tmp = right; //Tworzenie obiektu tymczasowego
        //Zamiana wskaźników na tablice miejscami
        std::swap(array,tmp.array);
        //Zamiana informacji o wielkości tablicy
        std::swap(size,tmp.size);
        return *this;
    }
    //Po wyjściu z ciała operatora obiekt tmp jest niszczony
    //a wraz z nim stara zawartość obiektu głównego

    /* ..................... */

};
    /* ..................... */
int main(int argc, char** argv) {
    Vector v1 = {1,2,3,4,5,5.5};
    std::cout << "v1: " << v1 << std::endl;
    Vector v2 = {5.5,4,3,2,1};
    std::cout << "v2: " << v2 << std::endl;
    v1 = v2; //To już nie jest kłopotliwe przypisanie
    std::cout << "v1: " << v1 << std::endl;
    std::cout << "v2: " << v2 << std::endl;
    v1[0] = 100.0; //Ustawienie wartości w obiekcie v1
    std::cout << "v1: " << v1 << std::endl;
    std::cout << "v2: " << v2 << std::endl;
    return 0;
}
```

Warto zwrócić uwagę, że zarówno operator przypisania jak również konstruktor kopiujący w swoich deklaracjach przyjmują jako parametr referencję na obiekt macierzystego typu.

```
    Vector(const Vector& other) //Konstruktor kopiujacy

    Vector& operator=(const Vector& right)
```

W przypadku operatora przypisania referencja ta jest odwołaniem do obiektu występującego po prawej stronie operatora przypisania, natomiast w przypadku konstruktora kopiującego jest to odniesienie do obiektu, który jest kopiowany.
Dodatkowym elementem jest zwracanie przez operator przypisania wyrażenia *this. Jest to wymagane ze względu na składnię języka która umożliwia między innymi operacje typu:

```
a = b = c;
```

Istnieje kilka wariantów deklaracji zarówno dla konstruktora kopiującego, jak również dla operatora przypisania[1].
Podsumowując, wyodrębnia się dwie podstawowe kategorie w zakresie kopiowania obiektów:

a. kopiowanie płytkie
b. kopiowanie głębokie

W języku C++ kopiowanie płytkie jest zapewnione przez domyślny konstruktor kopiujący. Chcąc zapewnić kopiowanie głębokie można zaimplementować własny konstruktor kopiujący. Podobna sytuacja dotyczy przypisywania wartości.
Poza sytuacją, gdy wprost wywołujemy konstruktor kopiujący występują jeszcze sytuacje gdy dzieje się to nie wprost. Jest szczególnie ważne, żeby programista zdawał sobie z tego sprawę. Niejawne wywołanie konstruktora kopiującego odbywa się podczas przekazywania parametrów przez wartość do funkcji lub metody. Ilustruje to przykład poniżej.

Przykład 5. Różnica pomiędzy przekazywaniem parametru przez wartość, a przez referencję

```
class Vector {
    /* ..................... */
    Vector(const Vector& other) //Konstruktor kopiujacy
        : size(other.size), array{new double[other.size]} {
        //Informacja diagnostyczna
        std::cout << "Vector(const Vector&)" << std::endl;
        /* ..................... */
    }
    Vector& operator=(const Vector& right) {
        //Informacja diagnostyczna
        std::cout << "operator=(const Vector&)" << std::endl;
        /* ..................... */
    }

    /* ..................... */
    ~Vector()
    {
        //Informacja diagnostyczna
        std::cout << "~Vector()" << std::endl;
        /* ..................... */
    }
    /* ..................... */
};
/* ..................... */
void wypisz1(Vector v)
{
    std::cout << v << std::endl;
}
void wypisz2(Vector& v)
{
    std::cout << v << std::endl;
}
int main() {
```

[1] Szczegóły na ten temat można znaleźć m.in. w:
http://www.cplusplus.com/articles/y8hv0pDG/

```
    Vector v = {1,2,3};
    std::cout << "Test wypisz1" << std::endl;
    wypisz1(v);
    std::cout << "Koniec testu wypisz1" << std::endl;

    std::cout << "Test wypisz2" << std::endl;
    wypisz2(v);
    std::cout << "Koniec testu wypisz2" << std::endl;
    return 0;
}
```

W powyższym przykładzie porównano dwie funkcje wypisz1 oraz wypisz2. Pierwsza z nich jako parametr przyjmuje wartość wektora natomiast druga referencję. Różnice w działaniu można zaobserwować m.in. w innym adresie pod jakim znajduje się wektor (adres jest raportowany przez destruktor), ale przede wszystkim po informacji w konsoli o wywołaniu konstruktora kopiującego.
Jedyna różnica pomiędzy funkcjami występuje w deklaracji typu parametru. Jednak efektywność drugiej funkcji jest znacznie wyższa, gdyż nie następuje niejawne kopiowanie i utworzenie de facto dodatkowego obiektu (obiektu tymczasowego).
Sprawdź, że w przypadku funkcji wypisz1 rzeczywiście następuje wywołanie konstruktora kopiującego. Sprawdź że w przypadki funkcji wypisz1 nie występuje wywołanie konstruktora kopiującego.
Problem z efektywnością może pojawić się również podczas przekazywania obiektów jako wartości zwracanej z funkcji. Przykładem może być tu następujący kod:

Przykład 6. Przykład zagadnienia zwracania obiektu przez wartość

```
#include <cmath>
/* ..................... */
Vector generuj(double poczatek, double koniec, double krok)
{
    std::size_t liczba = floor((koniec - poczatek)/krok);
    Vector result(liczba);

    for(std::size_t i = 0; i < liczba; i++)
    {
        result[i] = poczatek + i*krok;
    }
    return result;
}
int main() {
    std::cout << "Test generuj 1" << std::endl;
    Vector v = generuj(1,4,0.5);
    std::cout << "Koniec testu generuj 1" << std::endl;
    std::cout << "v: " << v << std::endl << std::endl;

    std::cout << "Test generuj 2" << std::endl;
    v = generuj(5,10,0.5);
    std::cout << "Koniec testu generuj 2" << std::endl;
    std::cout << "v: " << v << std::endl << std::endl;
    return 0;
}
```

Po uruchomieniu tego kodu w Visual Studio 2013[2] otrzymamy następujące wyjście:

```
Test generuj 1
Vector(6)
Vector(const Vector&)
~Vector()
Releasing memory 00B89FD0
Koniec testu generuj 1
v: Vector{1,1.5,2,2.5,3,3.5}

Test generuj 2
Vector(10)
Vector(const Vector&)
~Vector()
Releasing memory 00B8A200
operator=(const Vector&)
Vector(const Vector&)
~Vector()
Releasing memory 00B8A190
~Vector()
Releasing memory 00B8A290
Koniec testu generuj 2
v: Vector{5,5.5,6,6.5,7,7.5,8,8.5,9,9.5}

~Vector()
Releasing memory 00B8A200
```

Widać że w pierwszym teście został użyty raz konstruktor kopiujący, który skopiował zawartość wygenerowanego obiektu do v. W drugim teście konstruktor kopiujący został uruchomiony aż dwa razy. Raz aby prawdopodobnie stworzyć obiekt tymczasowy na podstawie obiektu result, oraz drugi raz w czasie wykonywania operatora przypisania. Każda taka operacja kopiowania jest potencjalnie kosztowna. Z tego powodu w **C++11** wprowadzono operacje przenoszenia zawartości. Operacje te są możliwe dzięki dodaniu tzw. r-referencji, czyli referencji do wartości prawostronnej[3]. Zapisuje się ją w następujący sposób:

```
Vector&& other
```

Zapis ten informuje nas, że obiekt wskazywany przez r-referencje other po obecnym działaniu nie będzie już do niczego wykorzystywany i najprawdopodobniej zostanie usunięty. Dlatego możemy swobodnie wykorzystać jego zawartość. Należy jednak pamiętać, że obiekt ten należy pozostawić w stanie poprawnie obsługiwanym przez jego destruktor. Stan nie musi być taki sam jak na początku, ale jednak musi być poprawny.
Dzięki istnieniu r-referencji stało się możliwe stworzenie konstruktora przenoszącego oraz operatora przypisania przenoszącego. W przypadku klasy Vector wygląda to następująco:

[2] Ze względu na różnice między kompilatorami – w tym wypadku strategię optymalizacji, wynik może być inny dla innych kompilatorów, a nawet dla innych ustawień tych samych kompilatorów

[3] R-wartość, wartość prawostronna – jest to wartość która może pojawiać się tylko z prawej strony operatora przypisania. Przykładem mogą tu być stałe (5, 4.3 , ‘c’) lub wyniki operacji arytmetycznych (a+5*b). W języku C++ jako r-wartości są traktowane także wszelkie obiekty tymczasowe.

Przykład 7. Konstruktor przenoszący oraz przenoszący operator przypisania

```
/* ..................... */
class Vector {
    /* ..................... */
    Vector(Vector&& other){
        std::cout << "Vector(Vector&&)" << std::endl;
        array = other.array; //Pobranie wskaźnika na dane
        size = other.size; //Kopiowanie rozmiaru wektora
        other.array = nullptr; //Usuwanie odniesienia do danych obiekcie
other
        other.size = 0; //Ustawianie rozmiaru wektora na 0
    }
    //Po wyjściu z ciała konstruktora obiekt na który wskazuje
    //referencja other został pozostawiony w poprawnym stanie.
    Vector& operator=(Vector&& right) {
        std::cout << "operator=(Vector&&)" << std::endl;
        std::swap(array,right.array); //Zamiana wskaźników na tablice
miejscami
        std::swap(size,right.size); //Zamiana informacji o wielkości
tablicy
        return *this;
    }
    //Po wyjściu z ciała operatora obiekt na który wskazuje
    //referencja right został pozostawiony w poprawnym stanie.

    /* ..................... */
};
/* ..................... */
Vector generuj(double poczatek, double koniec, double krok)
{
/* ..................... */
}
int main() {
    std::cout << "Test generuj 1" << std::endl;
    Vector v = generuj(1,4,0.5);
    std::cout << "Koniec testu generuj 1" << std::endl;
    std::cout << "v: " << v << std::endl << std::endl;

    std::cout << "Test generuj 2" << std::endl;
    v = generuj(5,10,0.5);
    std::cout << "Koniec testu generuj 2" << std::endl;
    std::cout << "v: " << v << std::endl << std::endl;
    return 0;
}
```

Po dodaniu wspominanych operacji wyjście programu ulegnie zmianie:

```
Test generuj 1
Vector(6)
Vector(Vector&&)
~Vector()
Koniec testu generuj 1
v: Vector{1,1.5,2,2.5,3,3.5}

Test generuj 2
Vector(10)
Vector(Vector&&)
~Vector()
operator=(Vector&&)
~Vector()
Releasing memory 00FF9FD0
Koniec testu generuj 2
v: Vector{5,5.5,6,6.5,7,7.5,8,8.5,9,9.5}

~Vector()
Releasing memory 00FFA190
```

Jak widać w przykładzie pierwszym zamiast konstruktora kopiującego został uruchomiony konstruktor przenoszący. W drugim przykładzie został wywołany raz konstruktor przenoszący i raz przenoszący operator przypisania. Operacje te są szybsze – nie wykonują zbędnych kopiowań.
Nie zawsze jednak kompilator domyśli się który obiekt ma charakter tymczasowy i można go przekazać jego r-referencje do jakieś metody. Czasami trzeba mu to zasugerować. Do tego służy funkcja std::move, która rzutuje daną nazwę lub l-referencję (zwykłą referencję) na r-referencje. Zilustruje do dobrze następny przykład:
Przykład 8. Użycie funkcji std::move

```
/* ..................... */
int main() {
    Vector v1 = {1,2,3,4};
    Vector v2 = std::move(v1);

     //Co wypiszą to dwie linie?
     std::cout << "v1: " << v1 << std::endl;
     std::cout << "v2: " << v2 << std::endl;


     Vector v3 = {3,4,5};
     v3 = std::move(v2);

     //A co wypiszą te dwie linie?
     std::cout << "v2: " << v2 << std::endl;
     std::cout << "v3: " << v3 << std::endl;
}
```

Rzutowanie to powoduje, że jest uruchamiana zawsze ta wersja funkcji, metody, operatora lub konstruktora, która jako parametr przyjmuje r-referencję. Oczywiście jeśli taka nie

istnieje, to zostanie zastosowana funkcja, metoda, operator lub konstruktor z l-referencją jako parametrem.
R-referencję jako parametr można stosować jako parametr w dowolnych funkcjach i metodach, a nie tylko w specjalnych konstruktorach lub operatorach. Zilustruje to poniższy przykład:

Przykład 9. Użycie funkcji z l-referencją i r-referencją

```
/* ..................... */
Vector podzielPrzez2(const Vector& arg)
{
    std::cout << "podzielPrzez2(const Vector&)" << std::endl;
    Vector result = arg;

    for(std::size_t i = 0; i < result.GetSize(); i++)
    {
        result[i] = result[i]/2;
    }
    return result;
}

Vector podzielPrzez2(Vector&& arg)
{
    std::cout << "podzielPrzez2(Vector&&)" << std::endl;
    /*
     * Mimo iż arg jest R-referencją, należy zastosować std::move,
     * aby wykorzystać jej właściwości. Bez tego zostanie wywołany
     * konstruktor kopiujący. Jest to cecha języka wymuszająca na
     * programiście jawne określenie miejsca w którym zostają użyte
     * specjalne właściwości R-referencji.
     */
    Vector result = std::move(arg);

    for(std::size_t i = 0; i < result.GetSize(); i++)
    {
        result[i] = result[i]/2;
    }
    return result;
}
int main() {
    Vector v1 = {1,2,3};
    Vector v2 = podzielPrzez2(v1);

    Vector v3 = podzielPrzez2(Vector{6,5,4});

    Vector v4 = {10,9,8};
    Vector v5 = podzielPrzez2(std::move(v4));

    std::cout << "v1: " << v1 << std::endl;
    std::cout << "v2: " << v2 << std::endl;
    std::cout << "v3: " << v3 << std::endl;
    std::cout << "v4: " << v4 << std::endl;
    std::cout << "v5: " << v5 << std::endl;
}
```

R-referencji oraz funkcji std::move można używać także w bardziej złożonych klasach, których pola są obiektami. Oto przykład:

Przykład 10. Złożona klasa

```
/* ..................... */
class Envelope
{
    Vector vector;
    friend std::ostream& operator<<(std::ostream&, const Envelope&);
public:
    Envelope(const Vector& vector) :
    vector{vector} {
        std::cout << "Envelope(const Vector&)" << std::endl;
    }
    Envelope(Vector&& vector) :
    vector{std::move(vector)} {
        std::cout << "Envelope(Vector&&)" << std::endl;
    }
    Envelope(const Envelope& other) :
    vector{other.vector} {
        std::cout << "Envelope(const Envelope&)" << std::endl;
    }
    Envelope(Envelope&& other) :
    vector{std::move(other.vector)} {
        std::cout << "Envelope(Envelope&&)" << std::endl;
    }
    Envelope& operator=(const Envelope& right) {
        std::cout << "operator=(const Envelope&)" << std::endl;
        Envelope tmp(right);
        std::cout << "operator=(const Envelope&) swap początek" <<
std::endl;
        std::swap(vector, tmp.vector);
        std::cout << "operator=(const Envelope&) swap koniec" <<
std::endl;
        return *this;
    }
    Envelope& operator=(Envelope&& right) {
        std::cout << "operator=(Envelope&&)" << std::endl;
        vector = std::move(right.vector);
        return *this;
    }
    ~Envelope() {
        std::cout << "~Envelope()" << std::endl;
    }
};
std::ostream& operator<<(std::ostream& ostr, const Envelope& e)
{
    ostr << "Envelope{vector: " << e.vector << '}';
    return ostr;
}
int main() {
    std::cout << "Konstruktor z r-referencją" << std::endl;
    Envelope e1{Vector{1,2,3}};
    std::cout << "e1: " << e1 << std::endl;
```

```
    std::cout << "Konstrukor przenoszący" << std::endl;
    Envelope e2 = std::move(e1);
    std::cout << "e1: " << e1 << std::endl;
    std::cout << "e2: " << e2 << std::endl;

    std::cout << "Przenoszący operator przypisania" << std::endl;
    e1 = std::move(e2);
    std::cout << "e1: " << e1 << std::endl;
    std::cout << "e2: " << e2 << std::endl;

    std::cout << "operator przypisania copy-swap początek" << std::endl;
    e2 = e1;
    std::cout << "operator przypisania copy-swap koniec" << std::endl;
    std::cout << "e1: " << e1 << std::endl;
    std::cout << "e2: " << e2 << std::endl;
}
```

Podczas analizowania wyjścia tego programu proszę zwrócić szczególną uwagę na działanie operacji std::swap w operatorze przypisania – powinno się uwidocznić wykorzystywanie przez nią operacji przenoszących zamiast kopiujących.
Z pewnością wygląda to dosyć skomplikowanie. Jednak jak się okazuje, gdy składnikami klasy są tylko pola będące obiektami podanymi przez wartość nie ma konieczności implementacji tylu konstruktorów, metod i operatorów. Otóż ten sam kod z funkcji main zadziałałby, gdyby klasę Envelope zaimplementować w ten sposób:

Przykład 11. Złożona klasa, wersja alternatywna

```
class Envelope
{
public:
    Vector vector;
};

std::ostream& operator<<(std::ostream& ostr, const Envelope& e)
{
    ostr << "Envelope{vector: " << e.vector << '}';
    return ostr;
}
```

Można to zauważyć analizując wyjście programu – kod klasy Vector będzie nadal informował o tym które elementy tej klasy zostały zastosowane. Wynika to z tego, iż kompilator domyślnie implementuje wszystkie wymagane konstruktory oraz operatory[4] implementacja ta sprawdza się w przypadku pól które są obiektami i typami prostymi. Natomiast przestaje być

[4] Szczegóły tego mechanizmu zostały opisane w książce *Bjarne Stroustrup, „Język C++. Kompendium wiedzy. Wydanie IV", Wydawnictwo Helion, 2014* w rozdziale 17, a w szczególności 17.6.
Informacje te dostępne są również w Internecie:

- http://en.cppreference.com/w/cpp/language/default_constructor,
- http://en.cppreference.com/w/cpp/language/copy_constructor,,
- http://en.cppreference.com/w/cpp/language/move_constructor
- http://en.cppreference.com/w/cpp/language/copy_assignment
- http://en.cppreference.com/w/cpp/language/move_assignment
- http://en.cppreference.com/w/cpp/language/destructor

wystarczająca w przypadku pól będących wskaźnikami lub innymi zasobami, którymi zarządza się w nietrywialny sposób.

Zadania na zajęcia

Zadanie 1.

Zaimplementować klasę Book która będzie miała dwa pola prywatne typu std::string. Jedno pole powinno zawierać autora, zaś drugie tytuł książki. Należy zaimplementować:

- konstruktor bezparametrowy,
- konstruktor, który przyjmuje autora i tytuł jako stałe l-referencje,
- konstruktor, który przyjmuje autora i tytuł jako r-referencje,
- getery,
- setery z stałymi l-referencjami,
- setery z r-referencjami,
- operator << wypisania na strumień std::ostream,

oraz także dla praktyki:

- konstruktor kopiujący,
- konstruktor przenoszący,
- kopiujący operator przypisania,
- przenoszący operator przypisania.

Oto wzorzec:

```
#include <string>
class Book {
    std::string author, title;
public:
/* ..................... */
};
```

Implementacje klasy Book proszę przestawić w następujący sposób:

```
string a="<?>", t="<?>";
Book e;
cout << "e: "<< e << endl;
Book b1 = {a, t};
cout << "b1: "<< b1 << endl;
Book b2 = {"<?>","<?>"};
cout << "b2: "<< b2 << endl;
Book b3 = b1;
cout << "b3: "<< b3 << " b1: " << b1 <<  endl;

e = std::move(b2);
cout << "e: "<< e << " b2:" << b2 <<  endl;
e.SetAuthor("<?>");
cout << "e: "<< e << endl;
e.SetTitle("<?>");
cout << "e: "<< e << endl;
```

W miejsca oznaczone przez <?> proszę wpisać dowolnie wybrany przez siebie tekst. Prowadzący może zmienić sposób prezentacji klasy Book.

Zadanie 2.

Wymagane zadanie 1!

Zaimplementować klasę Library, która będzie przechowywała obiekty klasy Book. Do wyboru jest implementacja **jako tablica**[5] lub **jako lista**. Należy zaimplementować:

- konstruktor bezparametrowy,
- konstruktor z listą inicjalizacyjną,
- konstruktor kopiujący,
- konstruktor przenoszący,
- kopiujący operator przypisania,
- przenoszący operator przypisania,
- operator[],
- wersja const operatora [],
- operator << wypisania na strumień std::ostream.

Używanie kontenerów stl jest niedozwolone!

Przy implementowaniu listy proszę stworzyć oddzielną strukturę lub klasę która będzie węzłem listy i będzie zawierać pole typu Book.

Oto wzorzec:

```
#include "Book.h"
#include <initializer_list>
class Library {
/* ..................... */
public:
    Library();
    Library(std::initializer_list<Book> list);
    Library(const Library& orig);
    Library(Library&& orig);
    Library& operator=(const Library& right);
    Library& operator=(Library&& right);
    Book& operator[](std::size_t index);
    const Book& operator[](std::size_t index) const;
    std::size_t GetSize() const;
    ~Library();

    //Tylko dla implementacji jako lista
    void push_back(const Book&);
    void push_back(Book&&);
    Book pop_back();
};
```

Implementacje klasy Library jako **tablicy** proszę przestawić w następujący sposób:

```
Library e;
cout << "e: "<< e << endl;
//3-5 książek
Library l1 = {{"<?>", "<?>"},
{"<?>", "<?>"},
{"<?>", "<?>"}};
```

[5] Analogicznie do klasy Vector

```
cout << "l1: "<< l1 << endl;
Library l2(2);
cout << "l2: "<< l2 << endl;
l2[0] = {"<?>", "<?>"};
l2[1] = {"<?>", "<?>"};
cout << "l2: "<< l2 << endl;
e = std::move(l2);
cout << "e: " << e << " l2: "<< l2 << endl;
l1[0] = std::move(e[1]);
cout << "l1: " << l1 << " e: "<< e << endl;
```

Implementacje klasy Library jako **listy** proszę przestawić w następujący sposób:

```
Library e;
cout << "e: "<< e << endl;
//3-5 książek
Library l1 = {{"<?>", "<?>"},
{"<?>", "<?>"},
{"<?>", "<?>"}};
cout << "l1: "<< l1 << endl;
LibraryAsList l2;
cout << "l2: "<< l2 << endl;
l2.push_back({"<?>", "<?>"});
l2.push_back({"<?>", "<?>"});
cout << "l2: "<< l2 << endl;
e = std::move(l2);
cout << "e: " << e << " l2: "<< l2 << endl;
l1[0] = std::move(e[1]);
cout << "l1: " << l1 << " e: "<< e << endl;
```

W miejsca oznaczone przez <?> proszę wpisać dowolnie wybrany przez siebie tekst.

Prowadzący może zmienić sposób prezentacji klasy Library.

Dodatek: O wywołaniu konstruktorów

Załóżmy istnienie klasy Vector wraz z implementacją[6]. Posiada ona następujące konstruktory:

```
Vector(); //Konstruktor bezparametrowy
Vector(std::size_t size); //Konstruktor z parametrem
Vector(std::size_t size, double init); //Konstruktor z parametrami

//Z lista inicjalizacyjną, tylko C++11
Vector(std::initializer_list<double> initList);

Vector(const Vector& other); //Konstruktor kopiujący
Vector(Vector&& other); //Konstruktor przenoszący, tylko C++11
```

Przed wprowadzeniem standardu C++11 konstruktory można było wywoływać w następujące sposoby:

```
Vector v1; //Konstruktor bezparametrowy
Vector v2();//Konstruktor bezparametrowy

Vector v3(5); //Konstruktor z parametrem
Vector v4 = 5; //Konstruktor z parametrem

Vector v5(5,7.0); //Konstruktor z parametrami

Vector v6(v5); //Konstruktor kopiujący
Vector v7 = v5; //Konstruktor kopiujący
```

Wraz z standardem C++11 pojawiły się dodatkowe możliwości

```
Vector v8(std::move(v6)); //Konstruktor przenoszący
Vector v9 = std::move(v7); //Konstruktor przenoszący

Vector v10({2,3,4,5,5.5}); //Konstruktor z listą inicjalizacyjną
Vector v11 = {2,3,4,5,5.5}; //Konstruktor z listą inicjalizacyjną
```

Postanowiono dodać możliwość zastąpienia nawiasów zwykłych (), klamrowymi {}. I tak powstały kolejne możliwości takie jak:

```
Vector v12{};//Konstruktor bezparametrowy

Vector v13{v5}; //Konstruktor kopiujący

Vector v14{std::move(v6)}; //Konstruktor przenoszący

Vector v15{2,3,4,5,5.5}; //Konstruktor z listą inicjalizacyjną
```

Gdyby nie było konstruktora z listą inicjalizacyjną dodatkowo istniałaby możliwość wywoływania konstruktorów z parametrami:

```
Vector v16{5}; //Konstruktor z parametrem
Vector v17{5,7.0}; //Konstruktor z parametrami
```

Ze względu na definicje konstruktora z listą inicjalizacyjną powstaje niejednoznaczność, która jest rozstrzygana na korzyść konstruktora z lista inicjalizacyjną[7].

---

[6] Patrz Dodatek: Kod klasy Vector str.21

[7] Więcej szczegółów znaleźć można w książce *Bjarne Stroustrup, „Język C++. Kompendium wiedzy. Wydanie IV", Wydawnictwo Helion, 2014* w rozdziałach 6.3.5 oraz 17.3

Dodatek: Kod klasy Vector

```
#include <initializer_list>
#include <iostream>
#include <algorithm>

class Vector {
    double* array;
    std::size_t size;
public:
    //Parametr typu std::initializer_list pozwoli na uruchomienie
konstruktora
    //z dowolną liczbą wystąpień liczb typu double. Będzie to wyglądało
jak
    //inicjalizacjia tablicy. Przykład:
    //Vector v1({1,2,3,4,5,5.5});
    //lub
    //Vector v1{1,2,3,4,5,5.5};
    //lub
    //Vector v1 = {1,2,3,4,5,5.5};
    //Jest to cecha C++11
    Vector(std::initializer_list<double> initList)
        : size{initList.size()}, array{new double[initList.size()]}
    {
        std::cout << "Vector(std::initializer_list)" << std::endl;
        std::size_t i = 0;
        for(double v : initList)
        {
            array[i] = v;
            i++;
        }
    }

    Vector(const Vector& other) //Konstruktor kopiujacy
        : size{other.size}, array{new double[other.size]} {
        std::cout << "Vector(const Vector&)" << std::endl;
        for(std::size_t i=0; i < size; i++)
            array[i] = other.array[i];
    }

    Vector(Vector&& other)
    {
        std::cout << "Vector(Vector&&)" << std::endl;
        array = other.array;
        size = other.size;

        other.array = nullptr;
        other.size = 0;
    }


    Vector(std::size_t size)
        : size{size}, array{new double[size]}
    {
        std::cout << "Vector("<< size << ")" << std::endl;
    }
```

```
    Vector()
        : Vector(0)
    {
        std::cout << "Vector()" << std::endl;
    }

    Vector(std::size_t size, double init)
        : Vector(size)
    {
        for(std::size_t i = 0; i < size; i++)
            array[i] = init;
    }

    //Implementacja operatora przypisania z wykorzystanie copy-swap
    Vector& operator=(const Vector& right) {
        std::cout << "operator=(const Vector&)" << std::endl;
        Vector tmp = right; //Tworzenie obiektu tymczasowego
        std::swap(array,tmp.array); //Zamiana wskaźników na tablice
miejscami
        std::swap(size,tmp.size); //Zamiana informacji o wielkości
tablicy
        return *this;
    }
    //Po wyjściu z ciała operatora obiekt tmp jest niszczony
    //a wraz z nim stara zawartość obiektu głównego

    Vector& operator=(Vector&& right) {
        std::cout << "operator=(Vector&&)" << std::endl;
        std::swap(array,right.array); //Zamiana wskaźników na tablice
miejscami
        std::swap(size,right.size); //Zamiana informacji o wielkości
tablicy
        return *this;
    }
    //Po wyjściu z ciała operatora obiekt, na który wskazuje referencja
right
    //został pozostawiony w poprawnym stanie.

    std::size_t GetSize() const {
        return size;
    }
    double& operator[](std::size_t i)
    {
        return array[i];
    }
    const double& operator[](std::size_t i) const
    {
        return array[i];
    }
    ~Vector()
    {
        std::cout << "~Vector()" << std::endl;
        if(array != nullptr)
        {
```

```
            std::cout << "Releasing memory " << array << std::endl;
            delete[] array;
        }
    }
};
```