Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

II. Klasa kontra struktura. Implementacja paradygmatu hermetyzacji w języku C++

1. Klasa – podstawowy typ danych języka obiektowego

1.1. Podstawowym rodzajem danych używanych w językach programowania obiektowego są **obiekty**. Obiekt do pewnego stopnia przypomina **klasyczną strukturę**: podobnie jak ona zawiera pola służące do przechowywania danych. Od klasycznej struktury różni się jednak tym, że oprócz pól zawiera w swym wnętrzu także metody – podprogramy przeznaczone do działań na tych polach. Jeszcze większe podobieństwo zachodzi między obiektem a nowoczesną **strukturą**, w której można umieścić zarówno pola z danymi, jak i metody do działań na tych danych. O ile jednak w strukturze wszystkie pola i metody mają charakter ogólnodostępny, to zakres widoczności pól i metod w obiekcie można zróżnicować. Właściwość języka polegająca na możliwości takiego połączenia danych i kodu programu w jednostkę programową nazywa się **hermetyzacją** (ang. *encapsulation*).

1.2. Do opisu obiektów służą w językach obiektowych **typy klasowe**. Hermetyzacja to metodologia mająca na celu ukrycie w obiekcie klasy danych o charakterze wrażliwym i udostępnianie ich wyłącznie poprzez wybrane metody. Siła wyrazu klasy jako typu danych jest więc duża: klasa to "typ danych z interfejsem", definiowanym za pomocą wybranych funkcji składowych. Naturalną konsekwencją istnienia klas jest możliwość efektywnej modularyzacji programu. Klasę można umieścić w bibliotece, która będzie używana przez różne programy.

1.3. Posługiwanie się typami klasowymi i obiektami jest kwestią **stylu programowania**. Te same działania, które implementuje się przy użyciu obiektów, można zaprogramować także przy użyciu danych globalnych i odpowiednich funkcji. W tym drugim wypadku zarządzanie programem będzie jednak znacznie trudniejsze.

1.4. Typ klasowy można zdefiniować jako typ niezależny lub jako potomka istniejącego typu klasowego. W tym drugim wypadku, pierwotny typ klasowy nazywa się **typem bazowym**, a nowo definiowany typ klasowy – **typem pochodnym** (**potomnym**). Relacja zachodząca pomiędzy klasą bazową a klasą pochodną nazywa się **dziedziczeniem**. Obiekty klasy pochodnej zawierają dane i metody odziedziczone z klasy bazowej oraz dane i metody specyficzne dla klasy pochodnej. Sposób dziedziczenia zadaje się za pomocą praw dostępu definiowanych po stronie klasy bazowej oraz **trybu dziedziczenia** definiowanego po stronie klasy pochodnej.

2. Implementacja klas w języku C++

2.1. W języku C++ typ klasowy definiuje się za pomocą konstrukcji składniowej w postaci:

`class` *id-klasy* {*ciało-klasy*}`;`

w której *ciało-klasy* jest zestawem deklaracji danych, struktur danych oraz funkcji, o określonych (w sposób jawny lub niejawny) prawach dostępu, np.

```
class Punkt
{
 private:
    int x, y;
 protected:
    string odl_szac_pocz;
 public:
    Punkt(int, int);
    ~Punkt();
    void Ustal_odl_szac_pocz(string osp)
      { odl_szac_pocz = osp;}
    void Rysuj(void);
};

//definicja konstruktora (typ I)
Punkt:: Punkt(int wsp_x, int wsp_y)
{x = wsp_x;
 y = wsp_y;}
```

```
//definicja destruktora
Punkt::~ Punkt ()
{}

//definicja metody Rysuj
void Punkt::Rysuj(void)
{
//...
}
```

2.1.1. Prawa dostępu do poszczególnych składowych klasy określa się za pomocą słów kluczowych `public`, `private` i `protected`, dzielących *ciało-klasy* na tzw. sekcje. Widoczność poszczególnych składowych w programie jest następująca:

- `public` – w całym programie,
- `private` – w funkcjach składowych przedmiotowej klasy oraz w funkcjach zaprzyjaźnionych,
- `protected` – w funkcjach składowych przedmiotowej klasy i funkcjach składowych wszystkich klas pochodnych względem tej klasy.

W wypadku braku słowa kluczowego określającego prawa dostępu do składowych sekcji przyjmuje się domyślnie, że składowe te mają charakter `private`.

2.1.2. Nadawanie wartości początkowych polom składowym klasy oraz modyfikowanie wartości tych pól powinno się odbywać za pośrednictwem funkcji składowych klasy.

2.1.3. Funkcje składowe klasy można zadawać na dwa sposoby: wewnątrz ciała klasy lub poza ciałem klasy. Ten drugi sposób jest zalecany jednak tylko dla klas zadeklarowanych globalnie (na poziomie pliku). W wypadku jego użycia, konieczna jest zapowiedź definicji (patrz przykład). Dla klas zadeklarowanych lokalnie (w ciele innej klasy) definicje wszystkich składowych klasy należy raczej zadawać wewnątrz ciała klasy (ewentualnie – poza ciałem klasy przy zastosowaniu podwójnego kwalifikatora).

2.1.4. Ze względów semantycznych, funkcje składowe klasy można podzielić na następujące kategorie:
– funkcje do zarządzania (konstruktory, destruktor),
– funkcje dostępu,
– funkcje do przetwarzania,
– funkcje pomocnicze.

2.1.4.1. Przyjmuje się, że wśród funkcji składowych klasy należy zdefiniować przynajmniej trzy **konstruktory** (**bezargumentowy lub z argumentami inicjalizowanymi wartościami domyślnymi**, **kopiujący** i **przenoszący**), jeden **destruktor** oraz dwa **przeciążone operatory przypisania** (kopiujący i przenoszący). Funkcja konstruktora jest aktywowana w sposób automatyczny w procesie tworzenia obiektu danej klasy, natomiast funkcja destruktora – w procesie jego likwidacji. Metody konstruktora i destruktora są funkcjami bez określonego typu. Konstruktor ma nazwę identyczną z nazwą samej klasy, zaś destruktor – nazwę powstałą przez złożenie znaku tyldy z nazwą klasy. W wypadku niezamieszczenia w *ciele-klasy* definicji odpowiedniego konstruktora, destruktora, lub przeciążonego operatora przypisania, kompilator utworzy domyślne definicje tych metod. Konstruktory, destruktor i przeciążone operatory przypisania to funkcje publiczne.

2.1.4.2. Funkcje **dostępu** wraz z funkcjami do **przetwarzania** tworzą interfejs obiektu klasy. Udostępniają obiektom zewnętrznym możliwość pobierania (funkcje dostępu) i modyfikowania (funkcje do przetwarzania) danych i struktur danych o charakterze prywatnym lub chronionym. Funkcje dostępu i funkcje do przetwarzania muszą być funkcjami publicznymi.

2.1.4.3. Funkcje **pomocnicze** to funkcje pełniące rolę usługową względem pozostałych funkcji klasy. Z tego powodu mają najczęściej charakter funkcji prywatnych.

2.2. W pewnych sytuacjach warto posłużyć się deklaracją klasy, w postaci:

`class` *id-klasy* **;**

Deklaracja ta poprzedza (w rozumieniu tekstowym) właściwą definicję klasy i ma charakter jej zapowiedzi (porównaj np. z pascalową konstrukcją `forward`). Przy jej użyciu można do pewnego stopnia rozwiązać problem odwołań dwóch lub więcej różnych typów klasowych do siebie nawzajem.

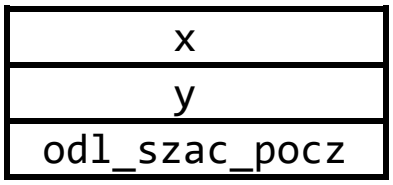
2.3. Wszystkie klasy, zarówno te zdefiniowane globalnie, jak i te zdefiniowane lokalnie mają zakres globalny. Oznacza to, że składowe klas pozostających w relacji zagnieżdżenia nie mają do siebie żadnych specjalnych praw dostępu.

3. Tworzenie, przetwarzanie i likwidacja obiektów klas w programach języka C++

3.1. Obiekty klas tworzy się w języku C++ w sposób statyczny (przechowując je w pamięci statycznej) lub dynamiczny (przechowując je w pamięci dynamicznej). W pierwszym wypadku odbywa się to przy użyciu zwykłej deklaracji, w drugim za pomocą operatora `new` lub funkcji do tworzenia inteligentnych wskaźników (`make_shared`, `make_unique`). Do obiektu utworzonego statycznie można się odwoływać poprzez identyfikator lub referencję. Odwołanie za pomocą wskaźnika jest możliwe po uprzednim jego utworzeniu przy użyciu operatora adresu &. Do obiektu utworzonego dynamicznie można się odwoływać za pomocą wskaźnika, lub wprost – po uprzednim zastosowaniu operatora wyłuskania *, np.

```
//definicja klasy Punkt
Punkt pkt0;
Punkt pkt1(-8,2);
Punkt pkt2 = Punkt(-4,-1);
Punkt *pkt3 = &pkt1;
Punkt &pkt4 = pkt2;
Punkt pkt5 = {2, 0};
Punkt *pkt6 = new Punkt(5,8);
unique_ptr<Punkt> pkt7 = make_unique<Punkt>(-1,3);
Punkt pkt8(*pkt6);
```

3.1.1. Utworzenie obiektu klasy polega w każdym wypadku na przydzieleniu odpowiedniego obszaru pamięci dla reprezentacji tego obiektu oraz wywołaniu odpowiedniej funkcji konstruktora. W obszarze pamięci przechowującym reprezentację obiektu jest miejsce dla każdej składowej danej lub struktury danych. Oto graficzna ilustracja budowy obiektu klasy `Punkt`:



3.1.2. W *ciele-klasy* występuje na ogół wiele konstruktorów. O wyborze odpowiedniego konstruktora decydują: zwykłe **zasady przeciążania identyfikatorów funkcji** – dla konstruktorów klasycznych oraz **charakter wyrażenia obiektowego** będącego argumentem konstruktora (l-wartość lub r-wartość) – dla konstruktorów kopiującego i przenoszącego. Po napotkaniu konstruktora o przeciążonym identyfikatorze, kompilator wybiera tę jego wersję, której sygnatura odpowiada typowi parametrów aktualnych.
W wypadku braku odpowiednich rodzajów konstruktorów, kompilator tworzy brakujące konstruktory domyślne, z zastrzeżeniami:
– jeśli w klasie zdefiniowano przynajmniej jeden konstruktor, to w takiej sytuacji kompilator nie utworzy domyślnego konstruktora bezargumentowego;
– jeśli w klasie zdefiniowano konstruktor kopiujący lub konstruktor przenoszący lub kopiujący operator przypisania lub przenoszący operator przypisania, to kompilator nie utworzy domyślnej wersji żadnej z pozostałych wymienionych metod;

3.1.3. Wywołania konstruktora zdefiniowanego w klasie `Punkt` zachodzą podczas **tworzenia** następujących obiektów tej klasy:

```
//definicja konstruktora (typ I)
Punkt:: Punkt(int wsp_x, int wsp_y)
{x = wsp_x;
 y = wsp_y;}
Punkt pkt0;   // konstruktor domyslny      (1)
Punkt pkt1(-8,2);                         (2)
Punkt pkt2 = Punkt(-4,-1);                (3)
Punkt pkt5 = {2, 0};                      (4)
Punkt *pkt6 = new Punkt(5,8);             (5)
unique_ptr<Punkt> pkt7 =
                make_unique<Punkt>(-1,3); (6)
Punkt pkt8(*pkt6);                        (7)
```

3.1.4. Konstruktor o takiej samej semantyce, jak wprowadzony wyżej konstruktor typu I można też zdefiniować w inny sposób – przy użyciu listy inicjalizacyjnej. Na tej liście umieszcza się identyfikatory tych pól składowych klasy, które mają być zainicjalizowane parametrami konstruktora. Taki konstruktor będzie miał postać:

```
//definicja konstruktora (typu Ia)
Punkt:: Punkt(int wsp_x, int wsp_y):
                        x(wsp_x), y(wsp_y) {}
```

W powyższej sytuacji, tworzenie obiektów (2) - (5) będzie się odbywać przy użyciu tych samych (w sensie składniowym) wywołań konstruktora. Obie metody ustalania wartości początkowych pól obiektu mogą być użyte równolegle w tym samym konstruktorze.
Inicjalizacja pól poprzez ich wiązanie z parametrami metody może być zrealizowana tylko w konstruktorze!

3.1.5. Do rozważanej klasy `Punkt` można wprowadzić drugi (poza domyślnymi) konstruktor, o zapowiedzi:

```
Punkt(int, string);
```

i definicji:

```
//definicja konstruktora (typ II)
Punkt:: Punkt(int wsp = 0, string p = "blisko")
{x = wsp;
 y = wsp;
 odl_szac_pocz = p;}
```

uzyskując nową, następującą postać *ciała-klasy* `Punkt`:

```
class Punkt
{private:
    int x, y;
 protected:
    string odl_szac_pocz;
 public:
  //deklaracja konstruktora typu I
    Punkt(int, int);
  //deklaracja konstruktora typu II
    Punkt (int, string);
    ~ Punkt(){};
    void Ustal_odl_szac_pocz(string osp)
      {odl_szac_pocz = osp;}
    void Rysuj(void);}
```

Wtedy poniższe deklaracje obiektów klasy `Punkt`:

```
Punkt pkt9(2,10);
Punkt pkt10(0, "w miejscu")
Punkt *pkt11 = new Punkt (2);
shared_ptr<Punkt> pkt12 = make_shared<Punkt>(-1,-4);
```

wiążą się z aktywacją konstruktorów, odpowiednio: . . . . .

3.1.6. Szczególną grupę konstruktorów stanowią konstruktory kopiujący i przenoszący, które odpowiednio kopiują lub przenoszą obiekty-parametry pole po polu. W wypadku konstruktora kopiującego, obiekt kopiowany nie ulega zniszczeniu. Parametrem konstruktora jest referencja do obiektu własnej klasy, np.

```
Punkt (const Punkt& p);
```

Konstruktor kopiujący zastosowano przy tworzeniu obiektu `pkt8`:

```
Punkt pkt8(*pkt6);
```

W podobny sposób można tworzyć obiekty przy użyciu kopiującego operatora przypisania:

```
Punkt pkt13 = pkt8;
Punkt pkt14 = *pkt6;
```

W wypadku konstruktora przenoszącego, obiekt kopiowany, reprezentowany przy użyciu r-wartości (wartości tymczasowej), ulega zniszczeniu. Parametrem tego konstruktora jest referencja do r-wartości reprezentującej obiekt własnej klasy, np.

```
Punkt (Punkt&& p);
```

Tworzenie poniższego obiektu `p15` może się odbywać przy zastosowaniu zdefiniowanego konstruktora przenoszącego:

```
Punkt p15(Punkt(1,1) + Punkt(2,2));
```

a modyfikacja obiektu `p15` – przy użyciu przenoszącego operatora przypisania:

```
p15 = Punkt(0,1);
```

3.1.7. Jeśli w ciele klasy nie zadeklarowano żadnego konstruktora, to kompilator utworzy bezargumentowy, „pusty” konstruktor domyślny. W klasie `Punkt`, przyjąłby on następującą postać składniową:

```
Punkt() {}
```

Należy zwrócić uwagę, że taka postać konstruktora wymagałaby wprowadzenia do ciała klasy `Punkt` dodatkowej metody, odpowiedzialnej za inicjalizację składowych prywatnych `x` i `y` (ewentualnie także – `odl_szac_pocz`).
Ponadto, jeśli ciało klasy nie zawiera definicji konstruktora kopiującego lub przenoszącego, kompilator utworzy domyślne wersje tych konstruktorów.

3.1.8. Na kompilatorze można wymusić syntezę pewnych metod specjalnych (konstruktor bezargumentowy, konstruktor kopiujący, konstruktor przenoszący, destruktor, kopiujący operator przypisania, przenoszący operator przypisania). Odbywa się to przy użyciu konstrukcji składniowej `default`, np.:

```
Punkt (const Punkt& p) = default;
~ Punkt() = default;
```

Na kompilatorze można także wymusić zablokowanie dowolnej metody, w tym specjalnej, np.:

```
Punkt (const Punkt& p) = delete;
```

Próba wywołania takiej metody spowoduje błąd kompilacji.

3.1.9. Dowolny konstruktor może być użyty w charakterze delegata – jako element definicji innego konstruktora. Przy zapisie trzeba się posłużyć odpowiednią listą inicjalizacyjną, np.

```
Punkt () : Punkt (0, 0) {}
```

3.1.10. Obiekty klas mogą być zarówno elementami tablic, struktur, czy unii, jak i składowymi innych klas.

3.1.10.1. Tablice obiektów mogą mieć charakter zarówno statyczny, np.:

```
Punkt tab1punkt[30];
array<Punkt, 30> tab2punkt;
```

jak i dynamiczny:

```
Punkt *wsk_tab3punkt = new Punkt[15];
vector<Punkt> tab4punkt(15);
```

W klasie, której obiekty mają być elementami tablicy, musi znajdować się konstruktor bezargumentowy lub konstruktor o argumentach, którym przypisano wartości domyślne.

3.1.10.2. Tworzenie obiektu pewnej klasy, której składowymi są obiekty innej klasy, musi być stowarzyszone z inicjacją tych składowych. Przykładowo, dla klasy `Odcinek`:

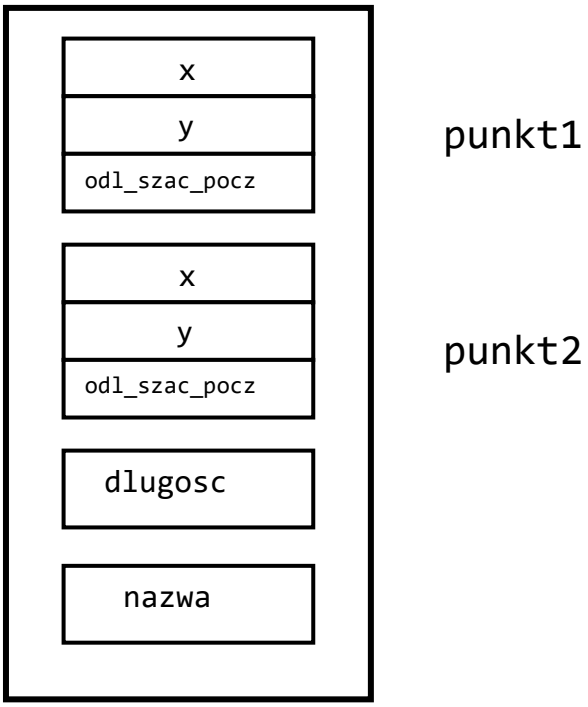
```
class Odcinek
{
 Punkt punkt1, punkt2;
 float dlugosc;
 string nazwa;
  public:
    Odcinek(Punkt, Punkt, string);
    //deklaracje dalszych metod klasy ...
};
```

```
Odcinek:: Odcinek(Punkt p1, Punkt p2, string id):
                          punkt1(p1), punkt2(p2)
{nazwa = id;}
//definicje dalszych metod klasy ...
```

utworzenie jej obiektu `odc1` może przebiegać w następujących krokach:

```
Punkt pkt15(-3,9);
Punkt pkt16(0,-7);
Odcinek odc1(pkt15, pkt16, "odcinekAB");
```

Graficzną ilustrację budowy tego obiektu prezentuje poniższy rysunek:



3.2. Przetwarzanie obiektów klas wymaga dostępu do składowych obiektów. Realizuje się go za pomocą operatorów **.** (obiekty reprezentowane przez identyfikator lub referencję) oraz → (obiekty reprezentowane przez wskaźniki), z zachowaniem znanych praw dostępu.

3.2.1. W klasie `Personalia` zadeklarowano składowe publiczne i chronione:

```
class Personalia
{public:
   string imie, nazwisko;
   char* wydzial = nullptr;
 protected:
   int rok_ur;
   float sr_zarobek;
 public:
   Personalia(string, string, int);
   ~Personalia();
   Personalia& Zmien_person (string, string, int);
   void Ustal_sredni(int*);
  //...
};
```

Przy założeniu poniższych deklaracji obiektów:

```
Personalia dyrektor("Iwan", "Grozny", 1963);
Personalia& naczelny = dyrektor;
Personalia* kadrowa =
         new Personalia ("Jola", "Wazna", 1989);
```

można posłużyć się następującymi odwołaniami do składowych obiektów:

```
//int rocznik=1979;
//string imie = "Jan", nazwisko = "Kowalski";
dyrektor.Zmien_person(imie, nazwisko, rocznik);
//int zarobki[12];
kadrowa→Ustal_sredni(zarobki);
dyrektor.nazwisko
naczelny.imie
kadrowa→nazwisko
kadrowa→sr_zarobek
```

3.2.2. Dostęp do składowych prywatnych klasy mają jedynie składowe tej klasy oraz funkcje z nią zaprzyjaźnione. Każda funkcja zaprzyjaźniona z klasą musi być w niej zadeklarowana ze słowem kluczowym `friend`, np.

```
class Personalia
{public:
   string imie, nazwisko;
   char* wydzial = nullptr;
 protected:
   int rok_ur;
   float sr_zarobek;
 public:
   Personalia(string, string, int);
   ~ Personalia();
   Personalia& Zmien_person
                      (string, string, int);
   void Ustal_sredni(int*);
   friend int Porownaj(Personalia&, float);
   //...
};
```

```
//definicja konstruktora
//definicja destruktora
//definicja metody Ustal_sredni
int Porownaj(Personalia& osoba, float sr_krajowa)
{if (osoba.sr_zarobek > sr_krajowa)
     return 0;
   else
     return 1;}
```

3.2.3. W definicji funkcji składowej klasy można jawnie wskazać obiekt, dla którego ta funkcja jest aktywowana. Do tego celu służy standardowa zmienna wskaźnikowa `this`, np.

```
Personalia& Personalia::Zmien_person
              (string im, string naz, int rocz)
{imie = im;
 nazwisko = naz;
 rok_ur = rocz;
 return *this;}
```

Dzięki zastosowaniu tej zmiennej można, między innymi, skrócić łańcuch wywołań funkcji składowych tego samego obiektu do pojedynczego wyrażenia, np.

```
dyrektor.Zmien_person("Alfons","Mlody",1990).
                Ustal_sredni(nowy_zarobki);
```

gdzie `nowy_zarobki` oznacza 12-elementową tablicę miesięcznych zarobków nowego dyrektora.

Zmienna `this` może być także pomocna przy rozwiązywaniu konfliktów nazw pól i parametrów metod składowych klasy.

3.2.4. Obiekt klasy może być parametrem aktualnym wywołania funkcji. Sygnatura takiej funkcji musi zawierać na odpowiedniej pozycji typ klasowy obiektu. Tak jak w przypadku parametrów pozostałych typów, obiekty klas można przekazywać przez wartość (w tym – typu wskaźnikowego) oraz przez referencję.

3.3. Likwidacja obiektu klasy następuje w momencie zastosowania operatora `delete` do wskaźnika obiektu utworzonego w pamięci dynamicznej za pomocą operatora `new`, lub w momencie osiągnięcia przez sterowanie końca zakresu deklaracji obiektu utworzonego w inny sposób (w pamięci statycznej lub w pamięci dynamicznej przy użyciu inteligentnego wskaźnika). W obu wymienionych sytuacjach następuje automatyczne wywołanie funkcji destruktora klasowego. Ponadto, zwalniana jest pamięć zajęta przez sam obiekt (składowe obiektu).

3.3.1. Ciało klasy musi zawierać deklarację destruktora. W wypadku braku takiej deklaracji, kompilator utworzy destruktor domyślny, o ciele pustym. Takiej praktyki nie zaleca się dla klas zawierających składowe o wartościach tworzonych dynamicznie. Przykładowo, w klasie `Personalia:`

```
class Personalia
{public:
   string imie, nazwisko;
   char* wydzial = nullptr;
 protected:
   int rok_ur;
   float sr_zarobek;
 public:
   Personalia (string, string, int);
   ~ Personalia();
   //... definicje dalszych metod klasy
};
```

ze względu na dynamiczny charakter tablicy znaków wskazywanej przez pole `wydzial`, destruktor powinno się zdefiniować jak następuje:

```
Personalia::~ Personalia()
{delete [] wydzial;}
```

W przeciwnym wypadku, łańcuch znaków przechowujący wydział zatrudnienia pracownika pozostanie w pamięci dynamicznej także po likwidacji dowolnego utworzonego wcześniej obiektu klasy `Personalia`.

3.3.2. Ze względu na specyfikę (nieokreślony typ funkcji, pusta lista parametrów), funkcji destruktora nie da się przeciążyć. W *ciele-klasy* można zatem zdefiniować co najwyżej jeden destruktor.

Przykład:

```
#include "stdafx.h"
#include <iostream>
#include<vector>
using namespace std;
class A {
public:
	class B {
	public:
		B() {
			cout << "Jestem zwyklym konstruktorem
				bez argumentow klasy B" << endl;
		};
		~B() {
			 cout << "Juz mnie nie ma - B" << endl;
		};
		void opis() {
			cout << "Jestem obiektem klasy wewn. B" << endl;
		}
	};
private:
	int x;
public:
	A() {
		cout << "Jestem zwyklym konstruktorem
			bez argumentow klasy A" << endl;
	}
	A(int a) {
		cout << "Jestem zwyklym konstruktorem
			z argumentami klasy A" << endl;
		x = a;
	}
	~A() {
		 cout << "Juz mnie nie ma - A" << endl;
	}
	void opis() {
		cout << "Jestem obiektem klasy zewn. A" << endl;
	}
	A(const A& a)
	{
		cout << "Jestem konstruktorem kopiujacym klasy A" << endl;

	}
	A(A&& a) {
		cout << "Jestem konstruktorem przenoszacym
			klasy A" << endl;
	}
```

```
	A& operator = (A&& a) {
		cout << "Jestem przenoszacym operatorem przypisania
			 klasy A" << endl;
		return a;
	}
	A operator = (const A& a) {
		cout << "Jestem kopiujacym operatorem
			 przypisania klasy A" << endl;
		return a;
	}
};

int main()
{
	A a1(1);
	a1.opis();
	A a2 = A();
	a2.opis();
	A::B b1 = A::B();
	b1.opis();
	cout << endl;
	A a3 = a1;
	a3.opis();
	cout << endl;
	A tabA[3];
	cout << endl;
	tabA[0] = A(3);
	cout << endl;
	vector<A>(2);
	cout << endl;
	return 0;
}
```

```
#include "stdafx.h"
#include <string>
#include <iostream>
#include <memory>
using namespace std;
class Punkt
{
private:
	int x, y;
protected:
	string odl_szac_pocz;
public:
	Punkt(int, int);
	Punkt(const Punkt &p) {
		cout << "kopiujacy " << endl;
	}
	Punkt(Punkt&& p) {
		cout << "przenoszacy " << endl;
		return;
	}
	~Punkt();
	void Ustal_odl_szac_pocz(string osp) {
		odl_szac_pocz = osp;
	}
	void Rysuj(void);
	Punkt operator +(const Punkt& p) {
		Punkt temp = Punkt(x + p.x, y + p.y);
		return temp;
	}
	Punkt& operator = (const Punkt& p) {
		cout << "operator kopiowania" << endl;
		return *this;
	}
	Punkt& operator = (Punkt&& p) {
		cout << "operator przenoszenia" << endl;
		return *this;
	}
};

Punkt::Punkt(int wsp_x, int wsp_y) {
	cout << "klasyczny" << endl;
	x = wsp_x;
	y = wsp_y;
}
//definicja metody destruktora
Punkt::~Punkt() {
}
```

```
//definicja metody RYSUJ
void Punkt::Rysuj(void) {
	//...
}

int main() {
	Punkt pkt1(-8, 2);
	Punkt pkt2 = Punkt(-4, -1);
	Punkt* pkt3 = &pkt1;
	Punkt& pkt4 = pkt2;
	Punkt pkt5 = { 2, 0 };
	Punkt* pkt6 = new Punkt(5, 8);
	unique_ptr<Punkt> pkt7 = make_unique<Punkt>(-1, 3);
	Punkt pkt8(*pkt6);
	Punkt pkt9 = pkt8;
	Punkt pkt10 = *pkt6;
	Punkt p11(Punkt(1, 1) + Punkt(2, 2));
	pkt1 = Punkt(1, 1);
	pkt1 = pkt10;
	return 0;
}
```