

Programowanie obiektowe

na przykładzie języka C++

dr hab. Piotr Białas
pok 443, tel 5571
pon. 11⁰⁰-12⁰⁰, śr. 10⁰⁰-11⁰⁰
pbialas@th.if.uj.edu.pl
<http://th.if.uj.edu.pl/~pbialas/OOP/>

Literatura

- Część teoretyczna opiera się głównie na książce Grady Booch, "Object-oriented Analysis and Design" The Benjamin/Cummings Publishing Company, Inc. second ed. 1994
- Osobiście do nauki C++ używałem (używam) książki Bjarne Stroustrup, "C++ Programming language" AT&T third ed. 1997
- Innych podręczników nie znam więc nie mogę polecić ale dobrą opinią cieszą się m.in. Grębosz, Lipman, Eckel

Wykład 1 – przypomnienie

- Programowanie obiektowe to pewna metodologia w której staramy się widzieć świat (i program) jako zbiór oddziaływających ze sobą obiektów.
- Obiekty mają stan, zachowanie i tożsamość i pozwalają tym samym na tworzenie lepszych abstrakcji niż same funkcje (dekompozycja algorytmiczna) czy struktury danych (dekompozycja sterowana danymi).
- Klasa jest “matrycą” do produkcji obiektów. Obiekty są *instancjami* klas.

Wykład 2

- Znajdowanie i projektowanie klas
 - karty CRC
- Klasy w C++
 - składowe funkcje i zmienne
 - Tworzenie
 - Konstruktory
 - Przeciążanie funkcji
 - Operator new
 - Niszczanie
 - Destruktory
 - Operator delete
 - Kopiowanie

Dygresja

- Programowanie obiektowe opiera się w gruncie rzeczy na kilku pojęciach (podobnie jak i inne formy programowania)
- Ale to język decyduje na ile używanie tych pojęć będzie
 - wygodne (szybkość pisania)
 - efektywne (szybkość wykonywania)
- Należy więc rozróżniać te pojęcia (klasa/typ, dziedziczenie, polimorfizm) od szczegółów języka
- Ale to te szczegóły decydują o efektywności programowania i wykonywania kodu
- C++ został tak zaprojektowany aby umożliwić jak największe podobieństwo pomiędzy typami wbudowanymi a typami zdefiniowanymi oraz dużą efektywność (no i zgodność z C :)
W rezultacie jest dość skomplikowany :(

Wykład 2

- Dziedziczenie
 - Dziedziczenie dla interfejsu
 - Klasy abstrakcyjne
 - Funkcje virtualne

Znajdowanie i wymyślanie klas (obiektów)

- Znajdowanie i wymyślanie klas jest podstawą programowania obiektowego
- Klasy znajdujemy na każdym poziomie abstrakcji począwszy od najwyższego
- Na kolejno niższych poziomach szukamy klas potrzebnych do implementacji klas z wyższego poziomu
- Każda klasa ma określony zakres odpowiedzialności
- Aby wypełnić swoje zadania klasa może współpracować z innymi klasami

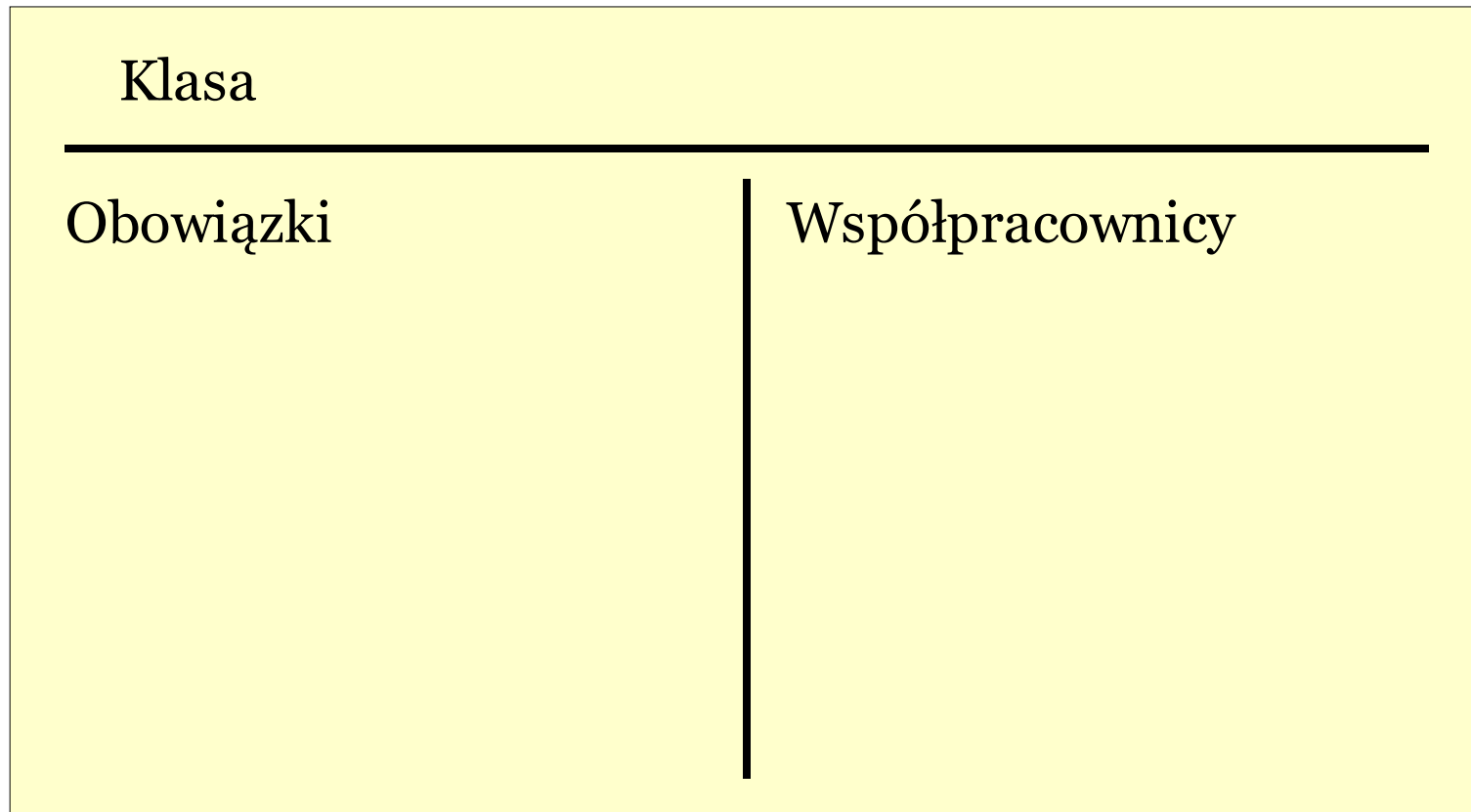
Skąd się biorą klasy ?

- Konkretnie pojęcia i czynności z dziedziny problemu
- Analiza scenariuszy
- Podkreślanie rzeczowników

Projektowanie klasy

- Dla danego pojęcia klasy ustalamy jej zakres obowiązków
- Tworzymy zestaw metod który umożliwia wypełnienie tych obowiązków
- Każdą operację należy rozpatrywać pod względem prostoty. Zbyt skomplikowane operacje należy rozdzielać
- Operacje złożone można przenieść do klas pomocniczych
- Należy przemyśleć operacje tworzenie kopiowania i niszczenia obiektów i ustalić wspólną strategię
- Pomyśleć o zupełności, czyli rozważyć operacje które nie są być może konieczne do wypełnienia obowiązków ale naturalnie wiążą się z daną abstrakcją

Karty CRC - Class, Responsibility, Collaboration



Karty CRC – użycie

- Karty CRC są użytecznym narzędziem wspomagającym analizę obiektową
- Kart CRC używamy podczas “burzy mózgów” do śledzenie scenariuszy
- Po zidentyfikowaniu obiektów przypisujemy im zakres obowiązków i odgrywamy scenariusze

Klasy w C++

```
#ifndef _stosdyn_
#define _stosdyn_

#include<stdlib.h>
using namespace std;

class DynamicznyStosInt {
private:
    int    *_rep;
    size_t _top;

public:
    DynamicznyStosInt(size_t n):_top(0)  {_rep= new int
[n]; };

    int  jestPusty();
    int  pop();
    void push(int);

    ~DynamicznyStosInt() {delete [] _rep;}

};
#endif
```

Definicje i deklaracje

- W C rozróżniamy definicje oraz deklaracje zmiennych i funkcji
- Deklaracja określa jakiego typu jest zmienna lub funkcja (w przypadku funkcji oznacza to podanie sygnatury czyli typów jej argumentów i zwracanej wartości)
- Podczas definicji zmiennym jest przydzielana pamięć a funkcją ich implementacja
- ```
extern double beta;
double x;
void setBeta(double x);
double sqr(double x) {return x;};
```
- Każda definicja powinna się pojawić tylko raz, ale ...

# Definicja klasy

- Deklaracja klasy jest po części jej definicją
  - Klasa musi zawierać definicje swoich zmiennych
  - Klasa może zawierać definicje lub deklaracje swoich metod (funkcji)
- Definicja klasy może być powtórzona w każdym osobno kompilowalnym module (pliku) pod warunkiem że identyczna w każdym z nich
- Aby to zapewnić należy definicję klasy umieszczać w osobnych plikach nagłówkowych

# Kontrola dostępu

- Składowe klasy (zmienne i funkcje) mogą mieć różne prawa dostępu:
  - `public`  
Ogólno dostępne
  - `private`  
Dostępne tylko z metod tej samej klasy (lub klas zaprzyjaźnionych)
  - `protected`
- Sprawdzanie praw dostępu następuje na poziomie kompilacji

# Tworzenie obiektu

- Jednym z częstszych błędów jest zapominanie o odpowiedniej inicjalizacji zmiennych lub nieprawidłowa inicjalizacja. Jest to szczególnie istotne w przypadku dynamicznego przydziału pamięci
- Dlatego w każdej klasie może zostać zdefiniowany zestaw funkcji zwanych konstruktorami które są wywoływane podczas tworzenia obiektu. Jeśli nie zdefiniujemy własnego konstruktora to zostanie wygenerowany konstruktor domyślny



# Konstruktory

```
#include<iostream>
using namespace std;

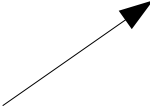
class A {
public:
 A() {cerr<<"wywolano konstruktor "<<this<<endl;}
 ~A() {cerr<<"wywolano destruktor " <<this<<endl;}
};
```

```
main() {
 A a;
 A b;
```

```
 for(int i=0;i<3;i++)
 {
 A c;
 }
```

```
}
```

this jest wskaźnikiem do danej instancji obiektu



# Konstruktory

|          |             |              |
|----------|-------------|--------------|
| wywolano | konstruktor | 0xbffffff710 |
| wywolano | konstruktor | 0xbffffff700 |
| wywolano | konstruktor | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff6e0 |
| wywolano | konstruktor | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff6e0 |
| wywolano | konstruktor | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff700 |
| wywolano | destruktor  | 0xbffffff710 |

# Konstruktory

|          |             |              |
|----------|-------------|--------------|
| wywolano | konstruktor | 0xbffffff710 |
| wywolano | konstruktor | 0xbffffff700 |
| wywolano | konstruktor | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff6e0 |
| wywolano | konstruktor | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff6e0 |
| wywolano | konstruktor | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff6e0 |
| wywolano | destruktor  | 0xbffffff700 |
| wywolano | destruktor  | 0xbffffff710 |

# Inicjalizatory

Pomiędzy listą argumentów a treścią konstruktora można umieścić listę inicjalizatorów

```
class complex {
 double _re;
 double _im;
public:
 complex(): _re(0), _im(0) {};
}
```

# Przeładowanie

- Klasa może mieć więcej niż jeden konstruktor. Różne konstruktory odróżniamy poprzez różne zestawy (typy) argumentów
- Jest to szczególne zastosowanie możliwości przeładowywania funkcji
- W C++ funkcje rozróżniane są nie tylko po nazwie ale i po typach argumentów
- Można również przeładowywać operatory
- Reguły rządzące przeładowywaniem funkcji komplikują się z powodu niejawnej konwersji typów

# Przeładowywanie funkcji

```
#include<iostream>
#include<string>
using namespace std;

void print(int i) {cout<<"int "<<i<<endl;};
void print(double x) {cout<<"double "<<x<<endl;};
void print(string s) {cout<<"string"<<s<<endl;};

main()
{
 print(1);
 print(1.0);
 print(0);
 print(0.0);
 print("Piotrek");
}
```

```
int 1
double 1
int 0
double 0
string Piotrek
```

# Linkowanie kodu z C (lub innych języków)

- Przeładowywanie oznacza w praktyce że kompilator generuje swoje własne nazwy funkcji w oparciu o nazwę funkcji i typy jej argumentów
- W ten sam sposób potraktuje prototypu funkcji w C
- Ale kompilator C nie generuje takich nazw i linker nie będzie mógł ich znaleźć
- W tym celu musimy zaznaczyć kompilatorowi C++ że dana funkcja jest z C

```
extern "C" funkcja() ;
```

```
0000003c T _Z5printf
00000000 T _Z5printfi
0000007e T _Z5printfSs
```

# Przeładowywanie konstruktorów

```
#include<iostream>
using namespace std;

class complex {
 double _re;
 double _im;
public:
 complex():_re(0),_im(0) {};
 complex(double re):_re(re),_im(0){};
 complex(double re,double im):_re(re),_im(im) {};
}

main()
{
 complex c;
 complex z(89);
 complex iz(0,89);
}
```



# Argumenty domyślne

```
#include<iostream>
using namespace std;

class complex {
 double _re;
 double _im;
public:
 complex(double re = 0 ,double im = 0):_re(re),_im(im) {};
}

main()
{
 complex c;
 complex z(89);
 complex iz(0,89);
}
```

# Dynamiczny przydział pamięci – operator new

- Jednym z częstszych zastosowań konstruktorów jest alokacja pamięci
- W tym celu można używać funkcji z rodziny `alloc(...)` z C, ale C++ posiada to tego celu własny operator `new`
- Operator `new` przydziela pamięć i wywołuje odpowiedni konstruktor więc jest bezpieczniejszy
- Operator `new` może przydzielać pamięć dla tablicy, ale w tym przypadku wywoływany jest zawsze bezargumentowy konstruktor (to samo dotyczy zwykłych tablic)

```
complex *c = new complex[100];
complex *I = new complex(0,1.0);
complex ct[1000];
```

Istnieją inne formy operatora `new` które omówię później. Na później zostawiam też kwestię obsługi błędów alokacji pamięci.

# Niszczenie obiektów – destruktory

- Przed zniszczeniem obiektu wywoływany jest destruktory
- W przeciwieństwie do konstruktora destruktory musi być zawsze jeden i jest bezargumentowy
- Najczęstszym zadaniem destruktora jest dealokacja pamięci, lub ogólniej zwolnienie zasobów (posprzątanie)

# Operator delete

- Do zwalniania pamięci służy w C++ operator delete
- Istnieją dwie formy tego operatora jedna do zwalniania pojedynczych obiektów i druga do zwalniania tablic

```
complex *c = new complex[100];
complex *I = new complex(0,1.0);
complex ct[1000];
```

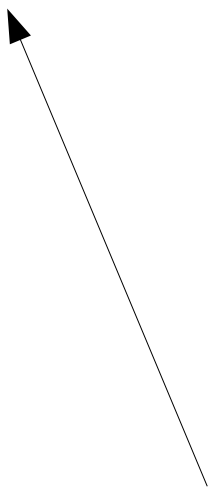
```
delete I;
delete [] c;
```

# Konstruktor kopiujący

- Konstruktor kopiujący służy do inicjalizowania obiektu innym obiektem tej samej klasy

```
complex(const complex &c) {...}
}
```

```
main()
{
 complex c(1,1);
 complex z(c);
 complex r=z;
}
```



Powyższy zapis oznacza że argument `c` jest przekazywany przez referencję a nie przez wartość, tzn do funkcji przekazywany jest w istocie wskaźnik do niego.

Ponieważ funkcja może zmienić wartość argumentu przekazywanego przez referencję, to zabraniamy tego słowem kluczowym `const`.

Taka konstrukcja oznacza że korzystamy z przekazywania przez referencję dla oszczędności. Więcej o referencjach będzie później.

# Konstruktor kopiujący

- Kompilator generuje domyślny konstruktor kopiujący który kopiuje strukturę bit po bicie
- Konstruktor kopiujący jest wywoływany niejawnie podczas przekazywania obiektów do funkcji poprzez wartość

```
#include<iostream>
using namespace std;

class complex {
 double _re;
 double _im;
public:
 complex(double re = 0,double im = 0):_re(re),_im(im) {};
 complex(const complex &c):_re(c._re),_im(c._im) {cerr<<"kopiuje
sie"<<endl;}

 double re() const {return _re;}
 double im() const {return _im;}
};

double im_val(complex c) {return c.im();};
double im_ref(const complex &c) {return c.im();};
```

```
main()
{
 complex c(1,1);
 cout<<"val "<<im_val(c)<<endl;
 cout<<"ref "<<im_ref(c)<<endl;
}
```

```
kopiuje sie
val 1
ref 1
```



# Kopiowanie przez podstawienie

- Każda klasa definiuje standardowy operator przypisania który kopiuje bit po bicie.
- `complex &operator=(const complex &c) ;`

```

#include<iostream>
using namespace std;

class complex {
 double _re;
 double _im;
public:
 complex(double re = 0,double im = 0):_re(re),_im(im) {};
 complex &operator=(const complex &c {
 _im=c._im;_re=c._re;cout<<"przypisanie "<<endl;
 return *this;
 }
};

main()
{
 complex c(1,1);
 complex z;
 z=c;
}

```