Wstęp do programowania

Wykład 10. Programowanie obiektowe c.d.

Plan wykładu

1. Tablica dynamiczna
- operatory new, delete,
- wyjątki,
- operator kopiujący =.
2. Dziedziczenie.
3. Polimorfizm.

Tablica dynamiczna

```
class DynamicArray {
  private:
    int arraySize;
    int items;
    double *values;

  public:
    DynamicArray();
    ~DynamicArray();

    void add(double v);
    int size();
    double& operator[](int i);
    DynamicArray& operator=(DynamicArray&);
};
```

Tablica dynamiczna: operatory new i delete

```
#include<string.h>
#include<iostream>

#include "DynamicArray.h"

DynamicArray::DynamicArray() {
  this->arraySize = 1;
  this->items = 0;
  this->values = new double[this->arraySize];
}

DynamicArray::~DynamicArray() {
  delete[] this->values;
}
```

Operatory **new** i **delete** służą do alokacji i zwalniania pamięci. **delete[]** oznacza zwolnienie pamięci zajmowanej przez tablicę.

Tablica dynamiczna

```
void DynamicArray::add(double v) {
  if (this->items == this->arraySize) {
    this->arraySize = (int) (2 * this->arraySize);
    double *newValues = new double[this->arraySize];
    memcpy(newValues, this->values, this->items * sizeof(double));
    delete[] this->values;
    this->values = newValues;
  }
  this->values[(this->items)++] = v;
}

double &DynamicArray::operator[](int i) {
  if (i >= 0 && i < this->items)
    return this->values[i];
  else
    throw std::runtime_error("IndexOutOfBounds");
}
```

throw zwraca wyjątek. Wyjątki są podstawowym mechanizmem informowania o błędach w językach obiektowych

Tablica dynamiczna

```
int DynamicArray::size() {
  return this->items;
}

int main(void) {
  DynamicArray tablica;
  double a;

  do {
    std::cin >> a;
    tablica.add(a);
  } while (a >= 0);

  for (int i = 0; i < tablica.size(); i++)
    std::cout << tablica[i] << std::endl;
}
```

Wyjątki

```
try{
  std::cout << tablica[10] << std::endl;
}catch (std::exception &e){
  std::cout << "wyjatek " << e.what() << std::endl;
}
...
```

W bloku `try{}` umieszcza się kod, który może wygenerować wyjątek / wyjątki. W bloku `catch{}` umieszczamy kod, który wykona się w przypadku gdy pojawi się wyjątek. Jeśli wyjątek nie pojawi się program będzie wykonywał kolejne instrukcje po bloku `catch{}`.

Tablica dynamiczna: operator kopiujący

```
int main(void) {
...
  DynamicArray t1, t2;
  t1.add(6);
  t2 = t1;
  std::cout << t1[0] << ", " << t2[0] << std::endl;

  t2[0] = 3;
  std::cout << t1[0] << ", " << t2[0] << std::endl;
...
}
```

`t2[0]=3` spowoduje także zmianę `t1[0]`, ponieważ oba obiekty posiadają wskaźnik values wskazujący na tą samą tablicę. Wynika to ze sposobu w jaki domyślnie są kopiowane obiekty. Wartości pól (w tym wskaźników) są kopiowane. Aby to zmienić, czyli utworzyć nową tablicę i na nią skierować nowy wskaźnik, należy przeciążyć operator =.

Tablica dynamiczna: operator kopiujący

```
DynamicArray& DynamicArray::operator=(const DynamicArray& t){
  this->arraySize = t.arraySize;
  this->items = t.items;
  delete[] this->values;
  this->values = new double[this->arraySize];
  memcpy(this->values, t.values, this->items * sizeof(double));
  return *this;
}
```

Teraz tablica zostanie poprawnie skopiowana.

Uwaga:

Konstrukcja `DynamicArray t3=t1;` jest poprawna składniowo, ale nie spowoduje wywołania zdefiniowanego operatora =. Wywoła tzw. konstruktor kopiujący:

```
DynamicArray::DynamicArray(const DynamicArray& t);
```

I jeśli go nie będzie, zostanie użyty domyślny konstruktor kopiujący, który nie zadziała poprawnie.

Dziedziczenie

Plik Animal.h

```
#include <string>
#ifndef ANIMAL_H_
#define ANIMAL_H_

class Animal {
    private:
        std::string name;
    public:
        Animal(std::string s);
        void printName();
        void sayHello();
};
#endif // ANIMAL_H
```

Dziedziczenie

Plik Animal.cpp

```
#include <iostream>
#include "Animal.h"

Animal::Animal(std::string s){
    this->name = s;
}

void Animal::printName(){
    std::cout << "Animal::" << this->name << std::endl;
}

void Animal::sayHello(){
    std::cout << "Animal::Hello" << std::endl;
}
```

Dziedziczenie

Plik Dog.h

```
#include <iostream>
#include "Animal.h"

#ifndef DOG_H_
#define DOG_H_

class Dog : public Animal{
    public:
    Dog(std::string s);
    void saySomething();
};

#endif // DOG_H
```

Klasa **Dog** rozszerza klasę (dziedziczy po klasie) **Animal**.

Dziedziczenie

Plik Dog.cpp

```
#include <iostream>
#include "Dog.h"

Dog::Dog(std::string s) : Animal(s){}

void Dog::saySomething(){
    std::cout << "Dog::Hau, hau" << std::endl;
}
```

Klasa **`Dog`** automatycznie przejmuje (dziedziczy) implementację wszystkiego, co znajduje się w klasie **`Animal`**. Konstruktor **`Dog`** odwołuje się do konstruktora **`Animal`**. Dodatkowo klasa **`Dog`** implementuje własną metodę **`saySomething()`**, której nie ma w klasie **`Animal`**.

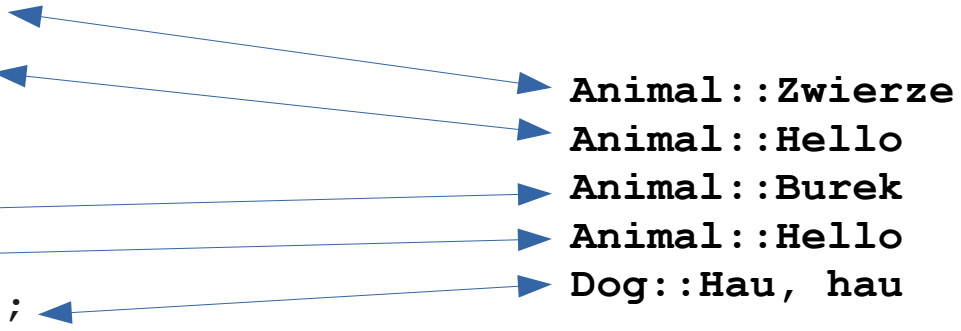
Dziedziczenie

Plik Dog.cpp

```
int main(){
    Animal zwierze("Zwierze");
    zwierze.printName();
    zwierze.sayHello();

    Dog burek("Burek");
    burek.printName();
    burek.sayHello();
    burek.saySomething();

    return 1;
}
```



Obiekt **burek** korzysta „za darmo” z metod w klasie **Animal**.

Przeciążanie metod w podklasie

Klasa potomna (rozszerzająca) może „podmienić” metodę w klasie rodzica (rozszerzanej)? Np.

Plik Dog.h w sekcji public:

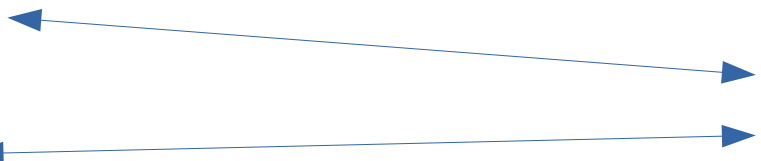
```
void sayHello();
```

Plik Dog.cpp:

```
void Dog::sayHello(){
    std::cout << "Dog::Hello" << std::endl;
}
```

```
...
Animal zwierze("Zwierze");
zwierze.sayHello();

Dog burek("Burek");
burek.sayHello();
...
```



```
Animal::Hello
Dog::Hello
```

Przeciążanie metod w podklasie

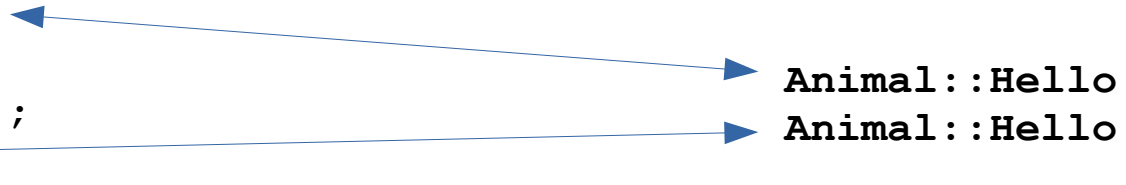
Przypuśćmy, że mamy funkcję:

```
void sayHello(Animal &a){
    a.sayHello();
}
```

Wtedy

```
...
Animal zwierze("Zwierze");
sayHello(zwierze);

Dog burek("Burek");
sayHello(burek);
...
```



Obiekt **burek** (instancja klasy **Dog**, poprzez mechanizm dziedziczenia, jest instancją klasy **Animal**. Dlatego może być używany wszędzie tam, gdzie instancja klasy **Animal** (ma wszystkie cechy: metody, atrybuty klasy **Animal**). Instancje **Animal** nie mogę zastępować instancji **Dog**, bo **Dog** może mieć cechy, których nie ma **Animal**. Co zrobić, aby funckja **sayHello()** użyła kodu klasy **Dog**?

Polimorfizm

Modyfikujemy deklaracje w plikach nagłówkowych. Animal.h:

```
vvirtual void sayHello();
```

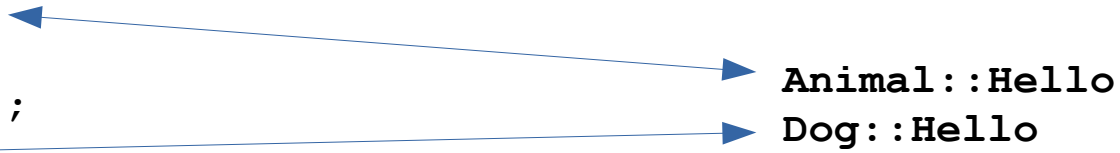
Dog.h

```
void sayHello() override;
```

Wtedy

```
...
Animal zwierze("Zwierze");
sayHello(zwierze);

Dog burek("Burek");
sayHello(burek);
...
```



Polimorfizm pozwala tworzyć ogólne algorytmy, których działanie może zależeć od charakteru obiektów, na których te algorytmy działają.

Grafy

Dziękuję za uwagę