

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/>

Wykład 4 – przypomnienie

- W C++ mamy dwa rodzaje dziedziczenia publiczne i prywatne
 - Dziedziczenia publicznego używamy jeśli dziedziczymy interfejs i ewentualnie implementację
 - Dziedziczenie prywatnego używamy jeśli chcemy dziedziczyć tylko implementację
- W C++ mamy dwa rodzaje funkcji: zwykłe i wirtualne
 - Funkcji zwykłych używamy jeśli dana metoda nie będzie redefiniowana w podklasach
 - Funkcji wirtualnych używamy jeśli dana metoda jest (może być) redefiniowana w podklasach
 - Jeśli funkcja musi być zdefiniowana w podklasie (dziedziczymy tylko interfejs) to używamy czystych funkcji wirtualnych
- Każda klasa posiada dwa interfejsy: dla innych klas (public) i dla podklas (protected)

Wykład 5 – różności

- Przeładowywanie operatorów
 - Niejawna konwersja typów
 - Funkcje zwracające referencje
 - Stałe – `const`
 - Typy wylicznane – `enum`
 - Operator `new`

Przeładowywanie operatorów

- Przeładowywanie (overloading) operatorów w C++ to kosmetyka, ale bardzo użyteczna.
- Operator @ to po prostu funkcja jedno lub dwu argumentowa o nazwie operator@ np:

```
operator+(int i, int j);  
operator[](int i);
```
- Nie można jednak przeładowywać operatorów dla typów wbudowanych
- Przeładowanie operatorów może polepszyć zrozumienie kodu jeśli definiujemy operatory zgodnie z ich znaczeniem
- Może je jednak bardzo zaciemnić
- W szczególności operatory dla typów definiowanych (klas) są dla kompilatora zwykłymi funkcjami i nie zakłada on żadnych związków między nimi np

```
a+=b i a=a+b
```

```
typedef long long int bigint;

class Wektor {
    ...
}

main()
{
    Wektor w(10), v(10), r(10);
    ...
    dodaj(w, v, r); //bez przeladowanie
    w=dodaj(v, r); //przeladowany =
    w=v+r;         //przeladowany +
}
```

Przeładowywanie typów wbudowanych

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

int operator+(int i,int j) {return i*j;}

main()
{
    cout<<1+2<<endl;
}
```

```
buildin.cpp:4: error: `int operator+(int, int)' must have
an argument of class
    or enumerated type
```

Operatory wewnątrz i na zewnątrz klasy

- Tak jak każda funkcja operator może być zdefiniowany jako funkcja składowa klasy lub poza klasą
- Nektóre operatory muszą być zdefiniowane wewnątrz klasy
(operator= operator[] operator->)
- Wewnątrz klasy operatory jedno argumentowe stają się bez argumentowe a operatory dwuargumentowe stają się jednoargumentowe

```
T operator+(T a,T b) ;
```

```
a+b // operator+(a,b)
```

W tym przypadku na obu argumentach
dokonywana jest niejawna konwersja typów

```
class T {  
public:  
T operator+(T b);  
}
```

```
a+b // a.operator+(b);
```

W tym przypadku niejawna konwersja typów
dokonywana jest tylko na prawym argumencie


```
class Wymierna {  
}
```

```
Wymierna operator+(Wymierna w,Wymierna v);
```

```
main()  
{
```

```
Wymierna w,v,r;
```

```
    w=v+r; //OK
```

```
    w=v+1; //zle!
```

```
    w=1+v; //zle!
```

```
    w=1;   // zle!
```

```
}
```

```
Wymierna operator+(Wymierna w, bigint i);  
Wymierna operator+(bigint i, Wymierna w);
```

```
Wymierna::operator=(bigint i);
```

Te funkcje będą niepotrzebne jeśli powiemy że int jest liczbą wymierną,
czyli zdefiniujemy reguły konwersji z int na Wymierną

Niejawna konwersja typów

- Jeśli obiekt przekazany do funkcji nie jest odpowiedniego typu to kompilator próbuje go przekształcić stosując
 - reguły dla typów wbudowanych (int na double itp)
 - Jednoargumentowe konstruktory
 - Operatory konwersji

Jednoargumentowe konstruktory

- Konstruktor klasy A o jednym argumencie klasy B jest jednocześnie operatorem konwersji z typu B na typ A
- Często jest to porządane
- Czasami jednak nie,
i wtedy możemy tego zabronić deklaracją `explicit`

```
typedef long long int bigint;
class Wymierna {
    bigint _licznik;
    bigint _mianownik;
public:
    Wymierna(bigint l = 0 ,bigint m =1):
        _licznik(l),_mianownik(m) {if(0==m) exit(1);}
}

main()
{
    Wymierna r(23),w(1,2);

    r=1; //OK
        r=w+1;
        r=3+w;
}
```

```
class Wektor {
double _rep;
public:
    Wektor(size_t s) {if(s>0) _rep=new double[s];};
}

main()
{
    Wektor v(20);
    Stos    s(100);
    v=1 // Przypisuje do v wektor o długości 1 !!

    s=27 // to samo!
}
```

Taka “przypadkowa konwersja może być myląca i jest niezgodna z “duchem” abstrakcji: liczba całkowita nie jest stosem ani nie jest wektorem

Explicit

```
class Wektor {  
    double _rep;  
public:  
    explicit Wektor(size_t s) {if(s>0) _re=new double[s];};  
}  
  
main()  
{  
    Wektor v(20);  
  
    v=1 // nie skompiluje się!  
  
}
```

Operator + można zdefiniować wewnątrz klasy:

```
class Wymierna {  
    ....  
public:  
    Wymierna operator+(Wymierna w);  
}
```

Ale wtedy lewy argument MUSI być obiektem klasy Wymierna

```
Wymierna w,v,r;  
int i;  
w=v+r; //OK  
w=v+1; //OK  
w=1+v; //zle -> 1.operator+(v)  
w=i+v; // zle -> i.operator+(v)
```

Operatory “symetryczne” i nie zmieniające obiektu lepiej definiować poza klasą

Operatory konwersji

```
main()  
{  
Wymierna w,v;  
double x;  
    x=w;  
    x=24.9+w;  
}
```

źle! Wymierna nie jest double. Co więcej nie da się zdefiniować odpowiedniego operatora przypisania bo musiałby być zdefiniowany w klasie double

źle! Nie ma operatora + który dodaje double do wymiernej a nawet jakby był to nie ma operatora przypisania

Oba te problemy możemy rozwiązać definiując jak można przekonwertować wymierną na double

Operatory konwersji

```
class Wymierna {  
    ...  
public:  
    ...  
    operator double() {  
        return ((double)_licznik)/(_mianownik);  
    }  
}
```

operator konwersji na typ T ma postać

```
operator T();
```

proszę zwrócić uwagę że nie deklarujemy
typu zwracanego

Operator przypisania

Kompilator automatycznie generuje dla każdego typu operator przypisania, kopiujący obiekt bit po bicie.

```
T &operator=(const T&);
```

W przypadku klasy Wymierna jest to wystarczające ale w przypadku klasy wektor spowoduje że oba wektory będą dzieliły tę samą reprezentację.

```

Wektor &Wymierna::operator=(const Wektor&v)
{
    if(this != &v)// sprawdzamy czy nie jest to a=a
    {
        if(_size != v._size)
        {
            delete [] _rep;
            _size=v._size;
            _rep=new double[_size]
        };
        for(int i=0;i<_size;i++)
        {
            _rep[i]=v.rep[i];
        }
    }
    return *this; //umożliwiamy (a=b)=c lub (a=b)== c
}

```

Operator przypisania

Operator

```
T &operatorT=(const T&)
```

jest szczególny tylko dlatego że jest generowany automatycznie, ale można oczywiście definiować inne

```
Wektor &Wektor::operator=(double x)
{
    for(int i=0;i<_size;i++)
    {
        _rep[i]=x;
    }
    return (*this)
}
main()
{
    Wektor w(20);
    w=1.0;
}
```

Tego nie da się załatwić konwersją typów. Nie można stworzyć wektora z dubla, bo nie wiadomo jak długi miałby być.

Funkcje zwracające reference

- Argumenty można przekazywać do funkcji poprzez wartość lub poprzez referencje
- Podobnie wyniki można zwracać poprzez wartość lub przez referencje
- Jeśli funkcja zwraca wynik poprzez referencje to w istocie zwraca wskaźnik do **istniejącego** obiektu i ten obiekt może być modyfikowany poprzez ten wskaźnik.
- Operatory przypisania z poprzednich przykładów zwracały referencje do samych siebie
- Powodem zwracania poprzez referencje jest zazwyczaj możliwość postawienia takiej funkcji po lewej stronie operatora przypisania (l-value)

Indeksowanie

```
class Wektor {  
    ...  
public:  
    ...  
  
    double    element(int i) {return _rep[i];}  
    void ustawElement(int i, double x) {_rep[i]=x;}  
}  
  
const int n=10;  
main()  
{  
    Wektor w(n),v(n);  
  
    for(int i=0;i<n;i++)  
    {  
        w.ustawElement(i,v.element(i)*2);  
    }  
}
```

Indeksowanie (lepiej)

```
class Wektor {  
...  
public:  
...  
  
double & element(int i) {return _rep[i];}  
}  
  
const int n=10;  
main()  
{  
    Wektor w(n),v(n);  
  
    for(int i=0;i<n;i++)  
    {  
        w.element(i)=v.element(i)*2;  
    }  
}
```


Indeksowanie (najlepiej)

```
class Wektor {  
...  
public:  
...  
  
double &operator[](int i) {return _rep[i];}  
double &operator()(int i) {return _rep[i];}  
}  
  
const int n=10;  
main()  
{  
    Wektor w(n),v(n);  
  
    for(int i=0;i<n;i++)  
    {  
        w[i]=v[i]*2;  
        w(i)=w(i)*v(i); //Styl fortranowski  
    }  
}
```

Operator ()

Operator () jako jedyny może przyjmować dowolną ilość parametrów. Często jest alternatywą dla operatora [] ponieważ łatwiej za jego pomocą implementować indeksy wielowymiarowe

```
class Macierz {  
  
public:  
  
double &operator()(int i, int j) {return _rep[i][j];}  
}  
  
main()  
{  
    Macierz m(10,20);  
  
    m(1,7)=0.75;  
}
```

Zwracanie referencji

Przekazywanie parametrów przez referencję jest często stosowane z powodów wydajnościowych aby uniknąć kopiowania dużych obiektów. Jest kuszące aby w ten sam sposób usprawnić zwracanie wyników

```
Wymierna &operator+(const Wymierna &w,const Wymierna &v)
{
    bigint licznik=w.licznik()*v.mianownik()+
                w.mianownik()*v.licznik();
    bigint mianownik=w.mianownik()*v.mianownik();
    Wymierna r(licznik,mianownik);
    return r;
}
```

źle! zwracana jest referencja do obiektu r który jest obiektem tymczasowym i zostanie zniszczony po wyjściu z funkcji

Zwracanie referencji

Nie wiele pomoże alokowanie pamięci dla zwracanego obiektu

```
Wymierna &operator+(const Wymierna &w,const Wymierna &v)
{
    bigint licznik=w.licznik()*v.mianownik()+
                w.mianownik()*v.licznik();
    bigint mianownik=w.mianownik()*v.mianownik();
    Wymierna *pr=new Wymierna(licznik,mianownik);
    return *pr;
}
```

Teraz każde dodawanie liczb wymiernych alokuje pamięć która musi być jawnie zwalniana jest to niewygodne a czasami wręcz niemożliwe

```
v=q+r;
delete v //OK ale łatwo o tym zapomnieć
```

```
v=q+r+z //pojawiają się obiekty pośrednie do których nie ma
        //dostępu
```

Czasami konieczne jest zwracanie obiektu i nie należy próbować być zbyt sprytnym :)

Zresztą często kompilator jest sprytniejszy od nas

```
Wymierna operator+(const Wymierna &w,const Wymierna &v)
{
    bigint licznik=w.licznik()*v.mianownik()+
                w.mianownik()*v.licznik();
    bigint mianownik=w.mianownik()*v.mianownik();
    return Wymierna(licznik,mianownik);
}
```

Kompilator może wyoptymalizować to wywołanie

operatory += itp.

Często łatwiej jest definiować operatory arytmetyczne, definiując najpierw operatory typu +=

```
Wymierna &Wymierna::operator+=(const Wymierna &r)
{
    bigint licznik    = licznik()*r.mianownik()+
                        r.licznik()*mianownik();
    bigint mianownik = mianownik()*r.mianownik();

    _licznik=licznik;
    _mianownik=mianownik;
    return (*this);
}
Wymierna Wymierna operator+(const Wymierna &w,
                             const Wymierna &v)
{
    return Wymierna(w)+=v;
}
```

```
Wektor &Wektor::operator*=(double x)
{
    for(int i=0;i<size();i++)
    {
        _rep[i]*=x;
    }
    return (*this);
}
```

```
Wektor operator*(const Wektor &v,double x)
{
    return Wektor(w)*=x;
}
```

```
Wektor operator*(double x,const Wektor &v)
{
    return v*x;
}
```

Operatory jednoargumentowe

```
Wymierna Wymierna::operator-()  
{  
    return Wymierna(-licznik(),mianownik());  
}
```

```
Wymierna operator-(const Wymierna &w)  
{  
    return Wymierna(-w.licznik(),-w.mianownik());  
}
```


Operatory wejścia – wyjścia

```
ostream &operator<<(ostream &out, const Wymierna &w)
{
    out<<w.licznik()<<"/"<<w.mianownik();
    return out;
}
```

Zwracanie referencji do strumienia umożliwia łączenie:
`cout<<w<<z<<endl;`

```
istream &operator>>(istream &in, Wymierna &w)
{
    bigint l,m;
    in>>l>>m;
    w=Wymierna(l,m);
    return in;
}
```

Operator ++ i --

```
#ifndef _licznik_
#define _licznik_

#include<iostream>
using namespace std;

class Licznik {
    size_t _l;
public:
    Licznik():_l(0){};

    size_t wartosc() const {return _l;};

    Licznik &operator++()    {_l++;return *this;}; //prefix
    Licznik operator++(int) {Licznik l(*this);_l++;return l;};
    //postfix
};

ostream &operator<<(ostream &out,const Licznik &);

#endif
```

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

```
#include"licznik.h"
```

```
main()
{
```

```
    Licznik l;
```

```
    cout<<l++<<endl;
    cout<<++l<<endl;
```

```
}
```

```
0
```

```
2
```

Stałość

```
void f(const Rational &r)
{
    r+=1; //Zabronione!

    r.licznik()
}
```

Kompilator nie wie czy dana funkcja zmienia wartość obiektu czy nie. Zakłada że zmienia o ile nie zadeklarowano jej jako stałej

```
bigint licznik() const {...};
```

```

#include<iostream>
using namespace std;

class A {
public:
    A(int i = 0) {};
    void f() const {cerr<<"ja jestem stala"<<endl;};
    void f() {cerr<<"a ja nie stala"<<endl;};
};
void c(const A &a) {a.f();}
main()
{
    const A a;
    A b;                ja jestem stala
                        a ja nie stala
    a.f();              ja jestem stala
    b.f();
    c(b);
}

```