

Programowanie obiektowe w języku C++

Stanisław Gepner

sgepner@meil.pw.edu.pl

Co już wiemy

- 1 Klasy i struktury
- 2 modyfikatory *public* i *protected*
- 3 Metody - funkcje klasy
- 4 Konstruktory i destruktor klasy

Zmienne statyczne

```
class foobar{
public:
    static int a;
    int b = 5; // -std=c++11
};
int main(){
    foobar f1, f2;
    ...
}
```

- Nowe słówko *static*
- Współdzielone przez wszystkie instancje klasy
- Inicjalizacja poza ciałem bez słówka *static*
- Można użyć bez instancji klasy!
- Można zmienić wartość
- Przykład:

Zmienne statyczne const ...

```
class foobar{  
    public:  
        static const int a;  
        int b = 5; // -std=c++11  
};  
int main(){  
    foobar f1, f2;  
    ...  
}
```

- *static const*
- Współdzielone przez wszystkie instancje klasy
- Inicjalizacja w ciele
- Można użyć bez instancji klasy!
- Nie można zmienić wartości
- Przykład:

Funkcje statyczne

```
#include <iostream>
using namespace std;
class foobar{
public:
    static int a;
    int b;

    static void static_do(int
        new_d)
    {
        cout << "a=" << a << endl;
        a = new_d;
        //b = a; // nie wolno
    }
};
int foobar::a = 9;
int main(){
    foobar f1, f2, f3;
    cout << f1.a << endl;
    foobar::static_do(9);
    cout << f3.a << endl;
}
```

- *static typ nazwa(){}*
- Współdzielone przez wszystkie instancje klasy
- Można użyć bez instancji klasy!
- Może działać tylko na zmiennych, które są *static*
- Przykład:

Operatory

- Operatorem może być symbol, np.: +, -, *, /
- albo *new new[] delete []delete*
- albo też konwersja typów (przemilczeć)
- W C++ użycie operatora jest związane z wywołaniem metody
- C++ pozwala na definiowanie własnych operatorów
- ... ale nie dla typów wbudowanych
- Definicje mogą być w ciele, albo poza
- Operatory mogą być jedno (+a) i dwu argumentowe (a+b)
- W zasadzie metoda może być dowolna (co nie znaczy, że powinna)
- Operator powinien wykonywać przewidywalną operację np:
c=a+b nie powinien zmieniać argumentów a, b (choć właściwie to czemu nie?)
- Jest tego dużo

Operatory

Lista niepełna

+ operator dodawania, może być jedno lub dwuargumentowy
 - operator odejmowania, może być jedno lub dwuargumentowy
 * operator mnożenia, dwuargumentowy
 / operator dzielenia, dwuargumentowy
 % operator dwuargumentowy
 & jedno lub dwu argumentowy
 = operator przypisania, może być domyślny
 ~, !, =, <, > +=, -= *=, /= %=, &= —=, « », »= «=, == !=, <=
 >=, && —, ++ —, —, ->*, ->, [],
 () - wywołania - ile chcemy argumentowy
 poniższe operatory gdy ich nie zdefiniujemy robi to za nas kompilator
 new, new[], delete, delete[]

Operator

Na razie bez sensu

```
//Ogólna składnia
typ operator@ (argumenty)
{
    // operacje
}

//wywołanie
operator@(args);
```

```
#include <iostream>
using namespace std;
class foo{
public:
    int a;
};
void operator+(foo a) //zle
{
    a.a*=-1;
    cout << "Wywołano operator_+_a="
         << a.a << endl;
}
void operator*(foo& a) //tez zle
{
    a.a*=-1;
    cout << "Wywołano operator_*_a="
         << a.a << endl;
}
int main(){
    foo f1;
    f1.a = 9;
    cout << f1.a << endl;
    operator*(f1); //to tez jest
                  //wywołanie
    +f1;
    *f1;
    cout << f1.a << endl;
}
```


Operator dwuargumentowy

Wciąż bez sensu

```
#include <iostream>
using namespace std;
class foo{
public:
    int a;
};
void operator+(foo& a, foo& b) //nie zwraca??
{
    a.a = a.a + b.a;
    cout << "Wywołano operator_+_a=" << a.a << endl;
}

int main(){
    foo f1, f2;
    f1.a = 9;
    f2.a = 3;
    cout << f1.a << endl;
    f1+f2;
    cout << f1.a << endl;
}
```

Na razie trochę bez sensu, więcej przykładów ...

Operator =

Teraz lepiej

```
class Foo {
public:
    ...
    Foo & operator=(const Foo &
        prawa);
    ...
}

Foo& Foo::operator=(const Foo &p)
{
    if (this != &p) // Samo kopia?
    {
        ... // Wykonaj
    }
    return *this; //zwroc siebie
}

Foo a, b;
...
b = a;    // Jak b.operator=(a);
```

- Musi być w ciele klasy (?)
- Argument (prawa strona) jest *const*
- Operator zwraca by umożliwić $a=b=c$
- Samo przypisanie? $a=a$ if ($this \neq \&rhs$)
- przykład ...

Operator $+=$, $-=$, $*=$

Teraz lepiej

```
class Foo {
public:
    ...
    Foo & operator+=(const Foo &
        prawa);
    ...
}

Foo& Foo::operator+=(const Foo &p
    )
{
    if (this != &p) // a+=a?
    {
        ... // Wykonaj
    }
    return *this; //zwroc siebie
}

Foo a, b;
...
b += a;    // Jak b.operator+=(a);
```

- W ciele lub poza klasy
- Argument (prawa strona) jest *const*
- Operator zwraca by umożliwić $a+=b+=c$?
- przykład ...

Operator dwuargumentowy

W ciele - niebezpiecznie

```
#include <iostream>
using namespace std;
class Foo{
public:
    Foo operator+(const Foo &prawy) const {
        Foo res = *this; // Kopia res(*this);
        res += prawy;     // Uzyj +=
        return res;       // !
    }
};

int main(){
    Foo a, b, c;

    c = a+b;
```

Operator dwuargumentowy

Poza ciałem - *friend*

```
#include <iostream>
using namespace std;
class foo{
public:
    foo(int a) : a(a) {}
    void print() { cout << a << endl; }
private:
    int a;
    friend foo operator+(const foo& a, const foo& b);
};
foo operator+(foo& a, foo& b)
{
    foo tmp = a;
    tmp.a += b.a;
    return tmp;
}

int main(){
    foo f1(3), f2(2);
    f1.print();
    f2.print();
    foo f3 = f1+f2;
    f3.print();
}
```

friend

- *friend*
- Ma dostęp do atrybutów *private* i *protected*
- Pozwala na pełniejszą hermetyzację
- To co musiało by być *public* może zostać udostępnione tylko pewnym klasom
- Zwiększona zależność pomiędzy klasami
- Przyjaźń jest jednostronna, deklarowana przez klasę dopuszczającą
- Nieistotne gdzie w ciele
- Można do wielu
- Klasy jak i funkcje
- Nie jest przechodnia



```
#include <iostream>
using namespace std;
class foo{
public:
    foo(int a, int b) : a(a), b(b) {}
private:
    int a;
    int b;
    friend std::ostream& operator<<(std::ostream& os, const foo& f);
};

std::ostream& operator<<(std::ostream& os, const foo& f)
{
    os << "[" << f.a << ", " << f.b << "]";
    return os;
}

int main()
{
    foo f1(3,4), f2(5,6);
    std::cout << f1 << std::endl;
    std::cout << f2 << std::endl;
}
```

Obiekt funkcyjny ()

funktor

- Obiekt, który można wywołać jak funkcję
- Konieczne zdefiniowanie operatora ()
- Zaletą nad funkcją jest to, że funktor jest pełnoprawnym obiektem
- ... może więc mieć określony stan
- ... konstruktor, destruktor, itd.

```
#include <iostream>

class foo {
public:
    foo (int x) : _x( x ) {}
    int operator() (int y) { return _x + y; }
    int operator() (int a, int b, int c) {return _x + a + b - c;}
private:
    int _x;
};

int main(){
    foo f1(5);
    foo f2(4);
    std::cout << f1( 6 ) << endl;
    std::cout << f2( 6, 7, 2 ) << endl;
    return 0;
}
```


Dziedziczenie

- Gdy kolejna klasa rozszerza możliwości już istniejącej
- Pozwala tworzyć nowe w oparciu o stare
- Współdzielenie kodu
- Interpretowanie obiektu tak jakby był obiektem z którego dziedziczy
- Hierarchia klas

```
class nazwa :[operator_widoczności] nazwa_klasy_bazowej //na razie
{
    //definicja_klasy
};
```