

*dr Artur Bartoszewski*  
Katedra Informatyki  
UTH Radom

## Konstruktor kopiujący

## Konstruktor kopiujący

---

**Konstruktor kopiujący** to konstruktor, który może zostać wywołany przez kompilator (niejawnie) jeżeli zachodzi potrzeba stworzenia drugiego egzemplarza obiektu.

Np. podczas przekazywania obiektu do funkcji przez wartość, lub podczas tworzenia nowego obiektu „identycznego” jak już istniejący

Każda klasa ma konstruktor kopiujący – jeżeli programista go nie napisze, to zostanie on automatycznie utworzony przez kompilator.

## Konstruktor kopiujący

---

Konstruktor kopiujący przyjmuje **referencję do swojego typu** i tworzy nowy obiekt.

```
klasa::klasa (klasa &);
```

Konstruktor kopiujący może mieć też inne argumenty, ale muszą one mieć wartości domyślne.

```
klasa::klasa (klasa & obj, int x=0, float pi=3.14, int * p=NULL);
```

## Konstruktor kopiujący

Konstruktor kopiujący otrzymuje w parametrze referencję do obiektu, który należy skopiować (obiektu tej klasy, do której należy).

```
class Klasa
{
public:
    Klasa() {} // konstruktor domyślny
    Klasa(Klasa &obiekt) {} // konstruktor kopiujący
};
```

Taka konstrukcja dopuszcza możliwość ingerencji konstruktora w obiekt który jest kopiowany we wzorzec. Aby temu zapobiec można zadeklarować ten obiekt jako stałą.

```
Klasa(const Klasa &obiekt) {}
```

# Konstruktor kopiujący

```
class A
{
private:
    int x;
public:
    A() {}
    A(int p) : x(p) {}
    A(const A & obj)
    {
        x = obj.x;
    }
};
```

Konstruktor domyślny

Konstruktor z parametrami

Konstruktor kopiujący

## Konstruktor kopiujący

Konstruktor kopiujący możemy wywołać na 3 sposoby:

```
class Klasa
{
public:
    Klasa() {} // konstruktor domyślny
    Klasa(const Klasa &obiekt) {} // konstruktor kopiujący
};
int main()
{
    Klasa wzorzec; // obiekt, który będzie kopiowany
    Klasa kopia01(wzorzec);
    Klasa kopia02 = wzorzec;
    Klasa kopia03 = Klasa(wzorzec);
    return 0;
}
```

Konstruktor kopiujący będzie wywołany także podczas **przesyłania obiektu do funkcji poprzez wartość**.

Nie zostanie wywołany gdy obiekt przesyłamy do funkcji poprzez referencje

# Konstruktor kopiujący

```
class A
{
public:
    int x;
    string podpis;
    A()
    {
        cout<<"konstruktor domyslny"<<endl;
        podpis="obiekt pusty";
    }
    A(int px):x(px)
    {
        cout<<"konstruktor z parametrami"<<endl;
        podpis="obiekt z danymi";
    }
    A(const A & obj)
    {
        x = obj.x;
        cout<<"konstruktor kopiujacy"<<endl;
        podpis="kopia obiektu";
    }
};
```

Sposoby wywołania konstruktora kopiującego prześledzić można na przykładzie klasy posiadającej 3 konstruktory:

- konstruktor domyślny,
- konstruktor z parametrami,
- konstruktor kopiujący

# Konstruktor kopiujący

## Przykład wywołania konstruktora kopiującego

```
int main()
{
    A a1, a2(10);
    a1=A(a2);

    cout<<a1.x<<" " <<a1.podpis;
    return 0;
}
```

konstruktor domyslny  
konstruktor z parametrami  
konstruktor kopiujacy  
10 kopia obiektu

```
int main()
{
    A a2(10);
    A a1(a2);
    cout<<a1.x<<" " <<a1.podpis;
    return 0;
}
```

konstruktor z parametrami  
konstruktor kopiujacy  
10 kopia obiektu




# Konstruktor kopiujący

## Przykład wywołania konstruktor kopiującego c.d.

```
int main()
{
    A a2(10);
    A a1=a2;
    cout<<a1.x<<" "<<a1.podpis;
    return 0;
}
```

konstruktor z parametrami  
konstruktor kopiujący  
10 kopia obiektu

## Tak nie zadziała!



```
int main()
{
    A a1,a2(10);
    a1=a2;
    cout<<a1.x<<" "<<a1.podpis;
    return 0;
}
```

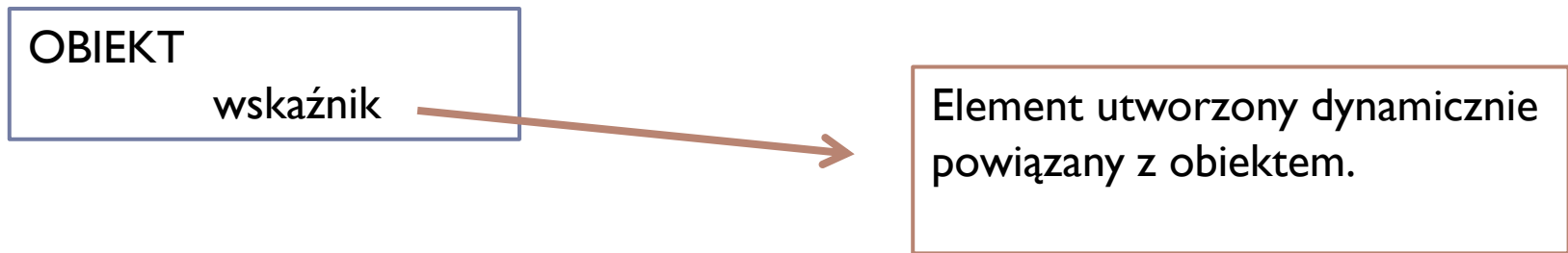
konstruktor domyslny  
konstruktor z parametrami  
10 obiekt z danymi

W ten sposób skopiujemy gotowy obiekt nie wywołując jego konstruktora kopiującego.

## Konstruktor kopiujący

Kiedy i dlaczego definiujemy własny konstruktor kopiujący?

**Wtedy, gdy składnikami obiektu są wskaźniki na obiekty tworzone dynamicznie.**



## Konstruktor kopiujący

Przykład: klasa osoba posiada wskaźnik „imie” do zmiennej, którą tworzy dynamicznie

```
class osoba
{
public:
    string * imie;
    osoba(string kto)
    {
        imie = new string;
        *imie = kto;
    }
    ~osoba() {delete imie;}
    void setImie(string kto) { *imie = kto;}
    string getImie() {return *imie;}
};
```


# Konstruktor kopiujący

---

```
int main()
{
    osoba ktos("Kowalski");
    osoba ktosInny("Nowak");

    cout<<ktos.getimie()<<endl
         <<ktosInny.getimie();

    return 0;
```



Kowalski  
Nowak

Z pozoru wszystko działa poprawnie – dopóki nie spróbujemy tworzyć obiektów poprzez kopiowanie.

## Konstruktor kopiujący

```
int main()
{
    osoba ktos("Kowalski");
    osoba ktosInny = ktos;

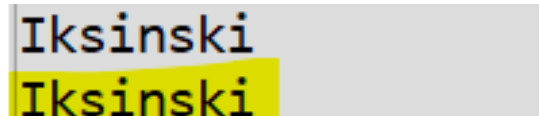
    ktos.setImie("Iksinski");

    cout<<ktos.getimie()<<endl
         <<ktosInny.getimie();

    return 0;
}
```

Obiekt „*ktosInny*” utworzyliśmy kopiując obiekt „*ktos*” i dopiero w następnym kroku zmieniliśmy jego zawartość.

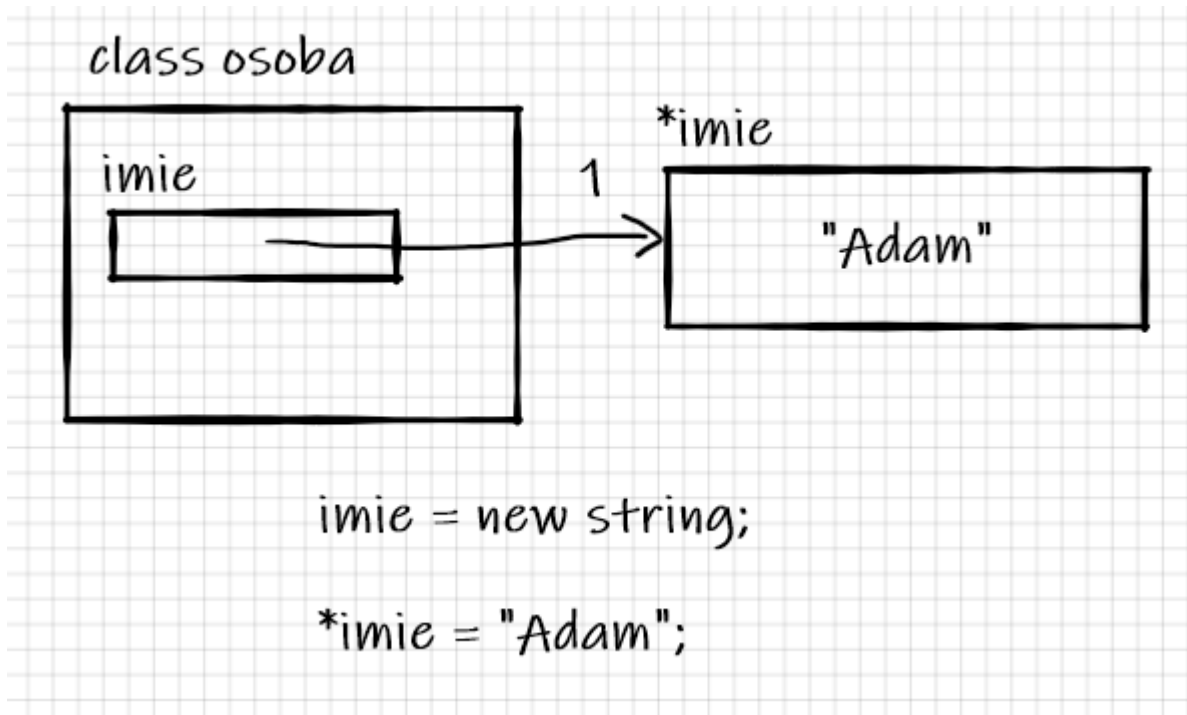
Okazuje się jednak, że zmianie uległa zawartość przechowywana przez oba obiekty.



Iksinski  
Iksinski

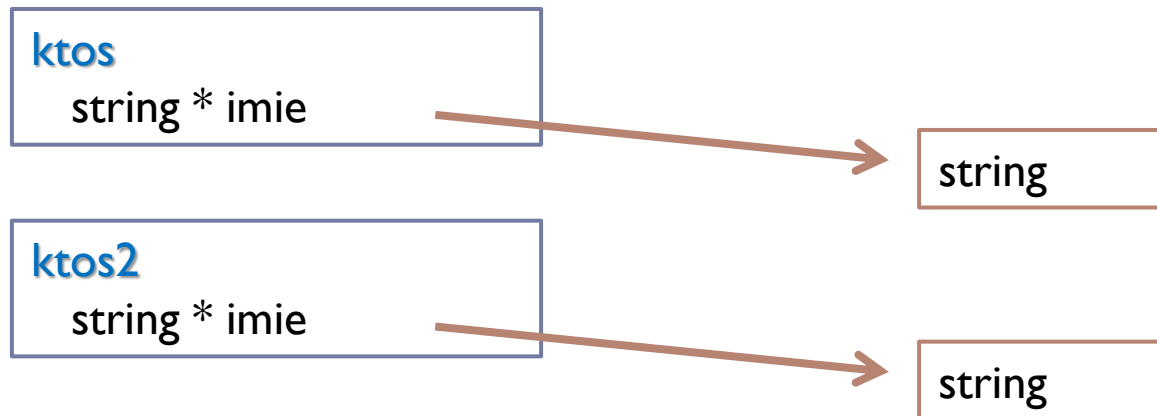
## Konstruktor kopiujący

Obiekt klasy **osoba** zawiera wskaźnik do zmiennej typu *string* tworzonej dynamicznie w konstruktorze obiektu.



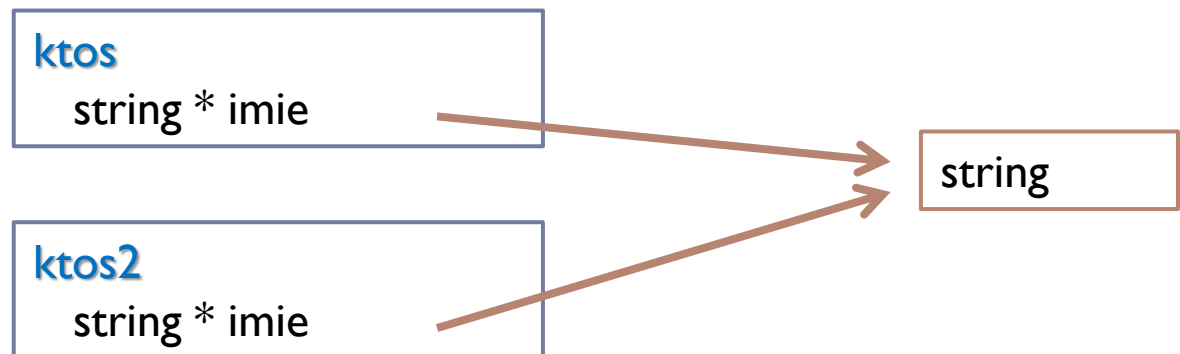
## Konstruktor kopiujący

Kopiując obiekt ( `ktos2 = ktos` ) spodziewamy się takiej sytuacji:



Jednak otrzymamy taką:

Wskaźniki `imie` we wszystkich kopiach obiektu zostały dosłownie skopiowane – czyli wskazują na ten sam element. Zmiana w jednym obiekcie pociąga za sobą zmianę we wszystkich



# Konstruktor kopiujący

Konstruktor kopiujący, który nie skopiuje pola *imie* jeden do jednego, a utworzy nowy łańcuch dynamiczny, przekopiuje do niego dane z wzorca i wstawi jego adres do pola *imie*

```
class osoba
{
public:
    string * imie;
    osoba(string kto)
    {
        imie = new string;
        *imie = kto;
    }

    osoba (osoba &);
    ~osoba() {delete imie;}
    void setImie(string kto){ *imie = kto;}
    string getImie(){return *imie;}
};

osoba::osoba(osoba & wzorzec)
{
    imie = new string;
    *imie = *wzorzec.imie;
    /*imie = wzorzec.getImie(); //- inna metoda
}
```

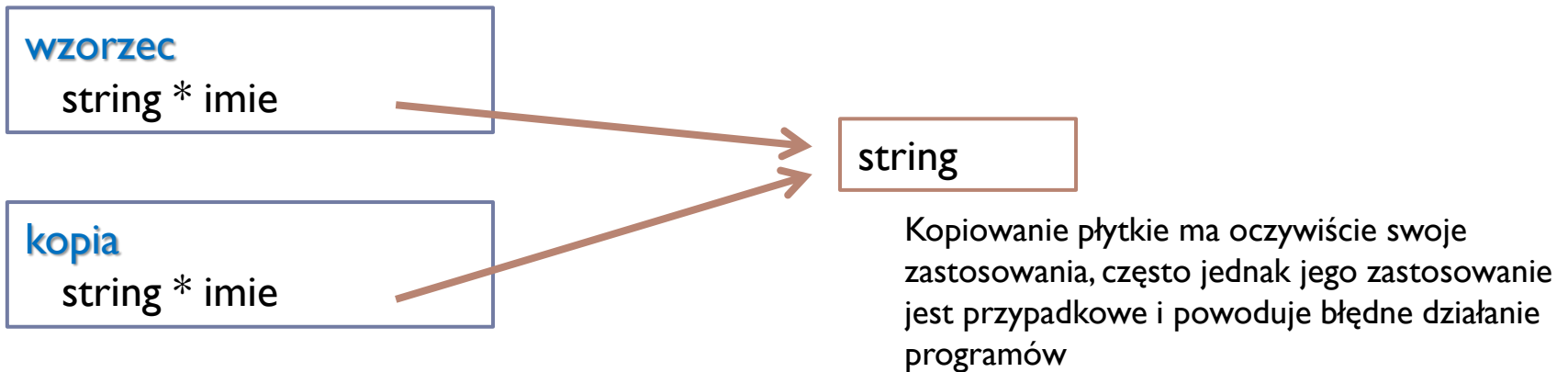
Tym razem zadział prawidłowo

Kowalski  
Iksinski

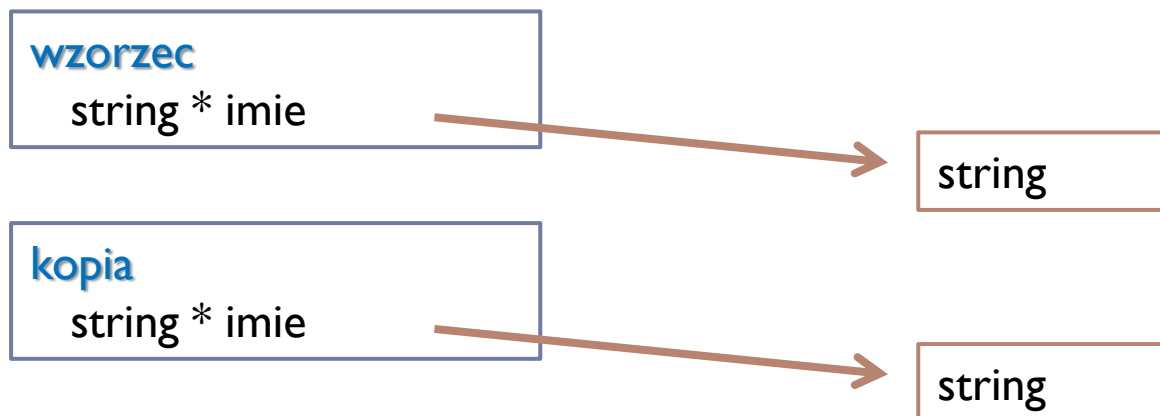


# Kopiowanie płytkie i głębokie

## Kopiowanie płytkie



## Kopiowanie głębokie



# Konstruktor kopiujący

---

## Wywołanie konstruktora kopiującego dla obiektów tworzonych dynamicznie

```
int main()
{
    osoba * wzor = new osoba("Kowalski");
    osoba * ktos = new osoba(*wzor);

    ktos->setImie("Iksinski");

    cout<<wzor->getimie()<<endl
         <<ktos->getimie();

    return 0;
}
```

# Konstruktor kopiujący

## Przykład: klasa student

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

class student
{
protected:
    int Id;
    string imie;
    string nazwisko;
    string kierunek;
    string wydzial;
    string *historia_studiow;
public:
    static int ile;
    student(string im = "", string nazw = "", string k = "", string w = "") : imie(im), nazwisko(nazw), kierunek(
k), wydzial(w)
    {
        Id = ++ile;
        historia_studiow = new string;
        *historia_studiow = "";
    }
    student(const student &wzorzec, string im = "", string nazw = "")
    {
        Id = ++ile;
        imie = im;
        nazwisko = nazw;
        kierunek = wzorzec.kierunek;
        wydzial = wzorzec.wydzial;
        // historia_studiow = wzorzec.historia_studiow; // - kopiowanie płytkie
        historia_studiow = new string;
        * historia_studiow = "";
    }
}
```

# Konstruktor kopiujący

---

```
void addWpis(string tekst)
{
    string temp = *historia_studiow;
    temp += "\n";
    temp += tekst;
    *historia_studiow = temp;
}

string toString()
{
    stringstream temp;
    temp << "(" << Id << ")" << nazwisko << " " << imie << " Wydział: " << wydzial << " Kierunek: " <<
kierunek
    << endl
    << "Historia studiow: " << *historia_studiow;
    return temp.str();
}

~student()
{
    delete historia_studiow;
}

};

int student::ile = 0;
```

# Konstruktor kopiujący

---

```
int student::ile = 0;

student kierunki[] = {
    student("", "", "Informatyka techniczna", "WTEiI"),
    student("", "", "Informatyka ogolnoakademicka", "WTEiI"),
    student("", "", "Pedagogika", "WFP"),
    student("", "", "Transport", "WTEiI")};

int main()
{
    student::ile = 0;

    student s1(kierunki[0], "Jan", "Kowalski");
    s1.addWpis("Przyjety na studia.");
    s1.addWpis("Zdal na 2 semestr");

    student s2(kierunki[0], "Andrzej", "Nowak");
    s2.addWpis("Przyjety na studia.");
    s2.addWpis("Skreslony z listy studentow");

    cout << s1.toString()<<endl;
    cout << s2.toString();

    return 0;
}
```

## Literatura:

---

W prezentacji wykorzystano przykłady i fragmenty:

- Grębosz J.: **Symfonia C++**, Programowanie w języku C++ orientowane obiektowo, Wydawnictwo Edition 2000.
- Jakubczyk K.: *Turbo Pascal i Borland C++ Przykłady*, Helion.

Warto zajrzeć także do:

- Sokół R.: **Microsoft Visual Studio 2012 Programowanie w Ci C++**, Helion.
- Kernighan B.W., Ritchie D. M.: **język ANSI C**, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

- Grębosz J.: **Pasja C++**, Wydawnictwo Edition 2000.
- Meyers S.: **język C++ bardziej efektywnie**, Wydawnictwo Naukowo Techniczne