

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 3 – przypomnienie

- Język C++ posiada silny system typów tzn. każda zmienna posiada określony typ i może przechowywać tylko obiekty tego typu (także wskaźniki i referencje)
- Próby użycia niewłaściwych typów są wyłapywane podczas kompilacji
- System typów można "osłabić" za pomocą hierarchii dziedziczenia: jeśli klasa A dziedziczy *publicznie* z klasy B to zmienne typu B są jednocześnie typu A. Ale nie na odwrót!
- Jeśli zmienna może wskazywać na obiekty różnych typów to wywołanie metody poprzez tą zmienną może zależeć od typu obiektu aktualnie wskazywanego przez tą zmienną
- Takie zachowanie nazywamy polimorfizmem i aby je uzyskać w C++ musimy używać wskaźników lub referencji i deklarować zmienne jako virtualne

Wykład 3 -- przypomnienie

- Dziedziczyć możemy interfejs i/lub implementację
- Możemy deklarować czyste funkcje wirtualne bez implementacji. Klasa która deklaruje choć jedną taką funkcję jest klasą abstrakcyjną która nie może posiadać żadnych instancji
- Klasę której wszystkie metody są czystymi funkcjami (i nie ma żadnych atrybutów) nazywamy interfejsem. Mówimy że klasa dziedzicząca implementuje interfejs

Wykład 4

- Przyjaciele
- Dziedziczenie
 - Generalizacja/specjalizacja
 - Dziedziczenie publiczne i prywatne
 - Funkcje wirtualne
 - Rzutowanie w dół i w górę
 - Konstruktory i destruktory w podklasach

Przyjaciele

- Klasa może zezwolić na dostęp do swoich prywatnych składowych wybranym klasom i funkcjom
- Dokonuje się tego poprzez zadeklarowanie tych klas lub funkcji jako przyjaciół klasy
- ```
class ListElement {

public:
 friend class List;
 friend ListIterator::next();
 friend jakasInnaFunkcja();
}
```

Można uważać że funkcje deklarowane jako friend są częścią interfejsu danej klasy

# Generalizacja/specjalizacja

- Mówimy że klasa dziedziczona generalizuje (uogólnia) typ klas dziedziczących np. Osoba to pojęcie bardziej ogólne niż Student czy pracownik, a Student studiów dziennych to pojęcie bardziej szczegółowe niż Student
- Pewne nieporozumienia w wywołuje fakt że klasa dziedzicząca często rozszerza zachowanie (interfejs) klasy dziedziczonej np. przez dodanie nowych funkcji które nie mają sensu w klasie bazowej
- W pewnych sytuacjach podklasa może zawężyć interfejs klasy dziedziczonej. Takiej możliwości raczej w C++ nie ma. Zresztą kłóci się to z filozofią dziedziczenia C++ bo wtedy nie możemy wywołać na obiekcie podklasy każdej metody z klasy bazowej
- Może to być sprzeczne z intuicją

# Zawężanie zachowania



Elipsa

Koło jest elipsą, ale nie każda operacja na elipsie ma sens dla koła np. przeskalowanie

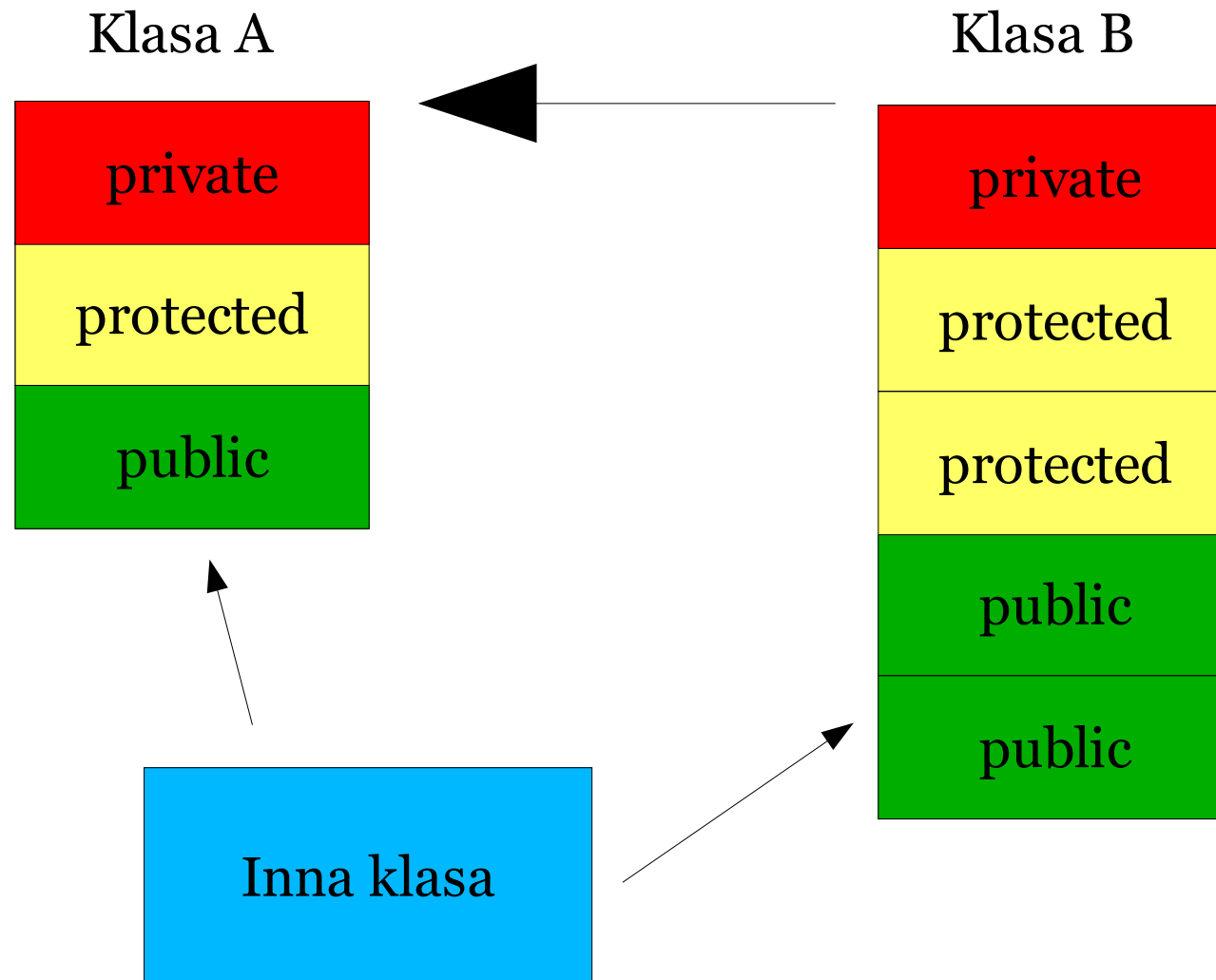
Koło

# Dziedziczenie publiczne

- Dziedziczenie publiczne wyraża relację "jest" pomiędzy klasami
- Oznacza to że:
  - każdy obiekt (wskaźnik do obiektu) klasy dziedziczącej jest obiektem klasy dziedziczonej i może być użyty na jego miejsce
  - Każda metoda klasy dziedziczonej może być wywołana na obiekcie klasy dziedziczącej
    - W C++ oznacza to że składowe publiczne i chronione stają się odpowiednio publicznymi i chronionymi składowymi klasy dziedziczącej
      - Składowe prywatne są w dla klasy dziedziczącej niewidzialne
- Dziedziczenie publiczne oznacza dziedziczenie interfejsu (i być może implementacji) i tak powinniśmy go używać



# Dziedziczenie publiczne

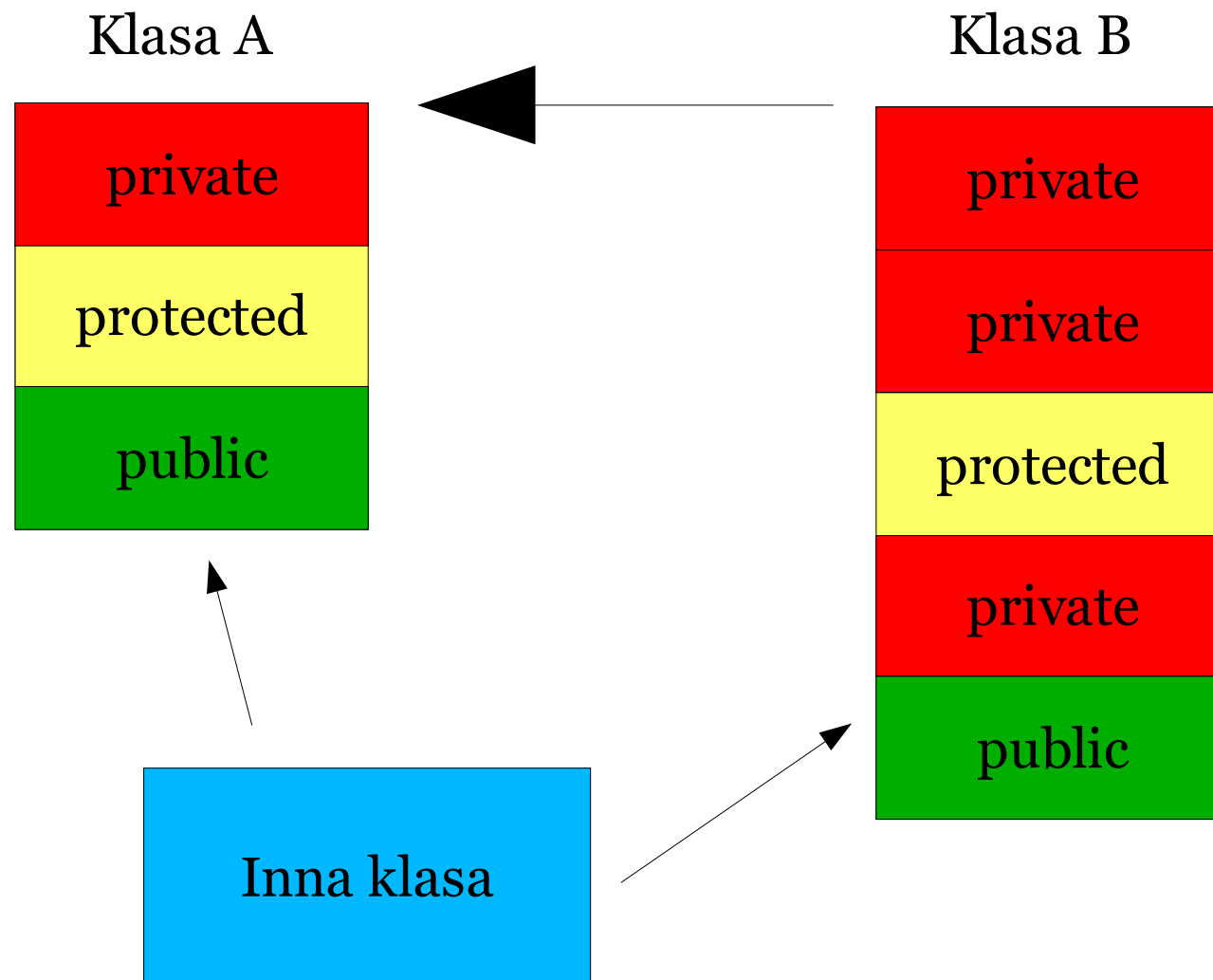


W dziedziczeniu publicznym interfejs klasy dziedziczonej staje się częścią interfejsu klasy dziedziczącej

# Dziedziczenie prywatne

- Dziedziczenie prywatne wyraża relację “jest implementowana za pomocą”
- Oznacza to że obiekt klasy dziedziczącej nie jest obiektem klasy dziedziczonej. Nie ma pomiędzy nimi żadnej relacji
  - W C++ oznacza to że składowe publiczne i chronione klasy dziedziczonej stają się składowymi prywatnymi klasy dziedziczącej
- Dziedziczenie prywatne oznacza dziedziczenie implementacji (tylko) a nie interfejsu

# Dziedziczenie prywatne



W dziedziczeniu prywatnym interfejs klasy dziedziczonej nie jest częścią interfejsu klasy dziedziczącej

# Dziedziczenie implementacji

```
class IntList {
public:
 void add(int);
 int current();
 void remove();
 void next();
 void reset();
 int length();
};

class IntStos : private IntList {
public:
 bool isEmpty() {return (length() > 0);}
 void push(int i) {add(i);};
 int pop() {reset(); int c = current(); remove(); return c;}
};
```

# Dziedziczenie prywatne

```
#include "prstos.h"
```

```
main()
```

```
{
```

```
 IntStos *s = new IntStos;
```

```
 IntList *l = new IntList;
```

```
 l=s;
```

```
 s->push(2);
```

```
 s->add(1);
```

```
}
```

```
private.cpp: In function `int main()':
```

```
private.cpp:8: error: `IntList' is an inaccessible base of
`IntStos'
```

```
prstos.h:3: error: `void IntList::add(int)' is inaccessible
```

```
private.cpp:11: error: within this context
```

```
private.cpp:11: error: `IntList' is not an accessible base of
```

# Dziedziczenie prywatne

- Można dodać części interfejsu klasy dziedziczonej prywatnie do klasy dziedziczącej redefiniując je w klasie dziedziczonej
-

```

class IntList {
 void prywatna();
public:
 void add(int);
 int current();
 void remove();
 void next();
 void reset();
 int length();
};

class IntStos : private IntList {
public:
 bool isEmpty() {return (length() > 0);}
 void push(int i) {add(i);};
 int pop() {reset(); int c = current(); remove();
return c;}
 int IntList::length();
 void IntList::prywatna(); //niedozwolone
};

```

# Dziedziczenie dla implementacji

- W przypadku dziedziczenia publicznego powyższy program kompiluje się
- To znaczy że na obiektach klasy StosInt można wywołać metody klasy ListInt, jest to użycie abstrakcji niezgodnie z jej przeznaczeniem
- W szczególności można by wywołać metodę `remove()` lub inne zmieniające stan obiektu
- Jeżeli więc korzystamy z dziedziczenia dla implementacji to powinniśmy używać dziedziczenia prywatnego



# Korzystanie ze funkcji z nadklas

- W podklasach możemy korzystać z funkcji zdefiniowanych w nadklasie odwołując się do nich poprzez w pełni kwalifikowane nazwy

- ```
class Student : public Osoba {  
public: print();  
}
```

```
Student::print() {  
Osoba::print();  
.....  
}
```

Funkcje wirtualne

- Funkcje deklarowane jako wirtualne są “późno łączone” tzn o tym która implementacja zostanie wywołana decyduje typ obiektu na którym została wywołana
- Jeśli funkcja nie jest zadeklarowana jako wirtualna to o ty jaka implementacja zostanie użyta zależy od typu wskaźnika do obiektu
- Dlatego nie należy redefiniować niewirtualnych funkcji w klasach dziedziczących

```
class Base {
public:
    virtual void wirtualna() {
        cout<<" wirtualna w klasie Base"<<endl;}
    void zwyczajna()
        {
            cout<<" zwyczajna  w klasie Base"<<endl;}
};

class Sub :public Base {
public:
    virtual void wirtualna() {
        cout<<" wirtualna w klasie Sub"<<endl;}
    void zwyczajna()
        {
            cout<<" zwyczajna  w klasie Sub"<<endl;}
};
```

```
#include<iostream>
using namespace std;
#include"wirtual.h"

main()
{
    Base *b,*pb = new Base;
    Sub *s,*ps = new Sub;

    b=ps;
    s=ps;
    b->wirtualna();
    b->zwyczajna();
    s->wirtualna();
    s->zwyczajna();

}
```

wirtualna w klasie Sub
zwyczajna w klasie Base
wirtualna w klasie Sub
zwyczajna w klasie Sub

Dziedziczenie

- Jeśli chcemy dziedziczyć interfejs używamy dziedziczenia publicznego
- Jeśli chcemy dziedziczyć tylko implementację to używamy dziedziczenia prywatnego
- Jeśli chcemy dziedziczyć tylko interfejs danej metody używamy czystych funkcji wirtualnych
- Jeśli chcemy dziedziczyć interfejs i defaultową implementację metody to używamy funkcji wirtualnych
- Jeśli chcemy dziedziczyć interfejs i obowiązkową implementację metody to używamy zwykłych funkcji. Nie redefiniujemy w klasach podrzędnych zwykłych (nie wirtualnych) funkcji w klasach nadrzędnych

Rzutowanie w górę

- Używając wskaźników i zmiennych wirtualnych uzyskujemy zachowanie polimorficzne
- ale przypisując wskaźnik podklasy do wskaźnika klasy dziedziczonej dokonujemy "rzutowania w górę" i tracimy dostęp do tej części interfejsu podklasy która nie jest odziedziczona z klasy wyższej
- Aby odzyskać ten interfejs musimy rzutować wskaźnik z powrotem czyli dokonać rzutowania w dół.

```
class Base {
public:
    virtual void wirtualna()
        {cout<<" wirtualna w klasie Base"<<endl;}
};

class Sub :public Base {
public:
    virtual void wirtualna()
        {cout<<" wirtualna w klasie Sub"<<endl;}
    virtual void sub()
        {cout<<" sub w klasie Sub"<<endl;}
};
```

Rzutowanie w górę

```
#include<iostream>
using namespace std;
#include"upcast.h"

main()
{
    Base *b = new Sub;

    b->wirtualna();
    b->sub();
}
```

```
upcast.cpp: In function `int main()':
upcast.cpp:10: error: `sub' undeclared (first use this
function)
upcast.cpp:10: error: (Each undeclared identifier is
reported only once for
each function it appears in.)
```


Rzutowanie w dół

```
#include<iostream>
using namespace std;
#include"upcast.h"

main()
{
    Base *b = new Sub;
    Sub *s;

    b->wirtualna();
    s=static_cast<Sub *>(b);           wirtualna w klasie Sub
    s->sub();                          sub w klasie Sub
}
```

Rzutowanie w dół nie jest bezpieczne bo nie wiemy na co b wskazuje, zwłaszcza jeśli dostaniemy go z “zewnątrz”

Konstruktory w klasach dziedziczących

- Podczas tworzenia podklas wywoływane są po kolei automatycznie konstruktory wszystkich nadklas. Po kolei tzn od najwyższej do najniższej
- ale tylko jeśli każda klasa posiada konstruktory standardowe (tzn bez argumentów)
- jeśli nie to musimy je wywołać jawnie

Konstrukcja podklas

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

class Osoba {
    string _imie;
    string _nazwisko;
public:
    Osoba() {cout<<"Osoba " <<endl;};
};

class Student : public Osoba {
public:
    Student() {cout<<"Student " <<endl;}
};
```

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

#include"konstrukcja.h"
```

```
main()
{
    Student s;
}
```

Osoba
Student

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

class Osoba {
    string _imie;
    string _nazwisko;
public:
    Osoba(const char *imie,const char *nazwisko):
        _imie(imie),_nazwisko(nazwisko) {};
};

class Student : public Osoba {
public:

};
```

```
In file included from konstrukcja2.cpp:4:
konstrukcja2.h:12: error: base `Osoba' with only non-default
constructor in
    class without a constructor
konstrukcja2.cpp: In function `int main()':
konstrukcja2.cpp:8: error: no matching function for call to
`Student::Student()
    ,
konstrukcja2.h:12: error: candidates are: Student::Student
(const Student&)
```

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

class Osoba {
    string _imie;
    string _nazwisko;
public:
    Osoba(const char *imie,const char *nazwisko):
        _imie(imie),_nazwisko(nazwisko) {};
};

class Student : public Osoba {
public:
    Student(const char *imie = "Jan" ,
        const char *nazwisko = "Nowak" ):
        Osoba(imie,nazwisko) {};
};
```

Niszczenie podklas

- Podczas niszczenia podklas wywoływane są destruktory nadklas w kolejności od klasy najniższej do najwyższej
- ale jeśli destruktor nie jest wirtualny a niszczony jest obiekt klasy podrzędnej wskazywany poprzez wskaźnik do klasy nadrzędnej to wywołany zostanie destruktor klasy nadrzędnej


```
class Base {
public:
    virtual void set(int i,double)=0;
    ~Base() {};
};

class Sub : public Base{
    double *_rep;
public:
    Sub(int i) {_rep= new double[i];};
    void set(int i,double x) {_rep[i]=x;}
    ~Sub() {delete [] _rep;}
};
```

```
#include<iostream>
using namespace std;
#include"destrukcja.h"

main()
{
    for(int i=0;i<1000000;i++)
    {
        cerr<<i<<endl;
        Base *s = new Sub(10000);
        delete s;
    }
}
```