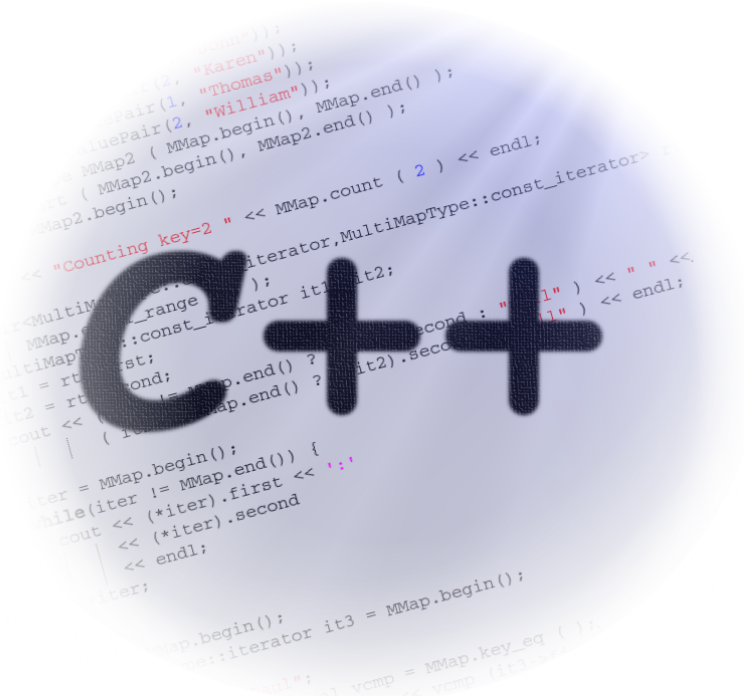
Języki Programowania

Prowadząca:
dr inż. Hanna Zbroszczyk

e-mail: *hanna.zbroszczyk@pw.edu.pl*
tel: +48 22 234 58 51

Konsultacje:
piątek: 14.00 – 15.30

www: http://www.if.pw.edu.pl/~gos/students/jp



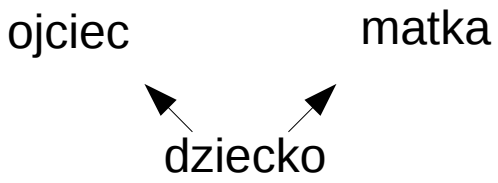
Politechnika Warszawska
Wydział Fizyki
Pok. 117b (wejście przez 115)

DZIEDZICZENIE WIELOKROTNE

Dziedziczenie wielokrotne - I

Dziedzicznie wielokrotne to takie, kiedy klasa wywodzi bezpośrednio z więcej niż jednej klasy.

```
class ojciec {
      // ...
};
class matka {
      // ...
};
class dziecko: public ojciec, public matka {
      // ...
};
```



Dana klasa podstawowa może pojawić się na liście pochodzenia tylko raz.

Definicja klasy podstawowej musi być znana kompilatorowi - nie wystarczy deklaracja zapowiadająca.

Specyfikatory dostępności mówią o tym w jaki sposób klasa pochodna dziedziczy.

Domniemanym specyfikatorem dostępności jest private.

Dziedziczenie wielokrotne - II

W klasie pochodnej można opuścić wywołanie konstruktorów klas podstawowych w sytuacji, kiedy są one domniemane lub kiedy klasa pochodna takiego nie posiada (uruchomi się wtedy konstruktor wygenerowany automatycznie przez kompilator).

Kolejność wywoływania konstruktorów klas podstawowych jest taka, jaka pojawi się na liście inicjalizacyjnej.

```
class czworokat {
        protected:
        double podstawa;
        public:
        czworokat(double aa=0.0): podstawa(aa) {
                cout<<"Konstruktor czworokata"<<endl;
        }
        ~czworokat() { }
};
```

Dziedziczenie wielokrotne - III

```
class prostokat {
        protected:
        double a, b;
        public:
        prostokat(double aa=0.0, double bb=0.0): a(aa), b(bb) {
                cout<<"Konstruktor prostokata"<<endl;
        }
        ~prostokat() {}
};
class kwadrat: public czworokat, public prostokat {
        protected:
        double pole;
        public:
        kwadrat(double pp=0.0, double aa=0.0, double bb=0.0, double po= 0.0):
                czworokat(pp), prostokat(aa, bb) {
                pole = po;
                cout<<"Konstruktor kwadrata"<<endl;
        }
};
```

Ryzyko wieloznaczności

Ryzyko wieloznaczności może pojawić się w dziedziczeniu wielokrotnym w sytuacji, kiedy obie klasy podstawowe mają pole o takiej samej nazwie (dostęp do tego pola nie ma akurat większego znaczenia, dlatego, że nawet składowe private są dziedziczne, tyle, że nie są widoczne) – w tej sytuacji ma znaczenie jednoznaczność, a nie ewentualny dostęp.

Można zawsze posłużyć się specyfikatorem zakresu:

```
klasa1::x
klasa2::x
```

Co jednak w sytuacji, kiedy klasa3 odziedziczy obie składowe, a następnie klasa4 będzie wywodzić się z klasa3?

(1) Można tkwić przy notacji ze specyfikatorem zakresu ::

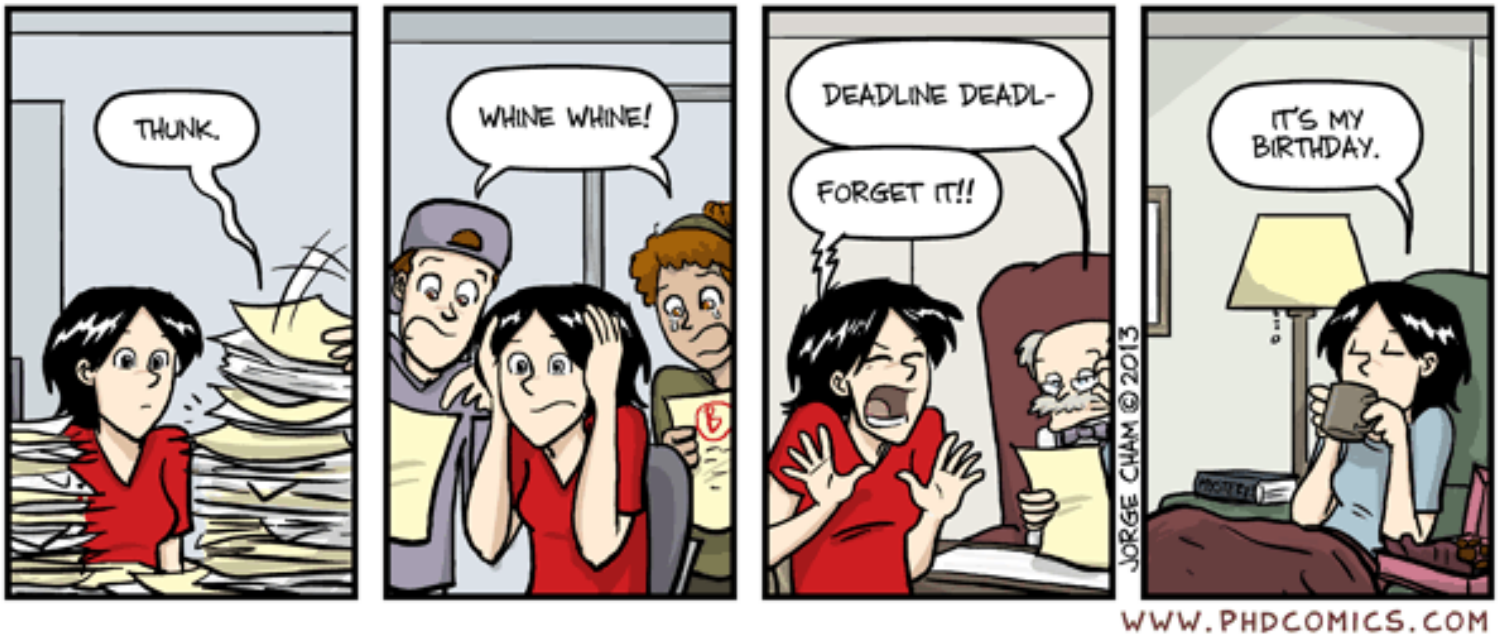
(2) Można w klasa3 przesłonić nazwę tych odziedzicznych pól definiując swoje nowe.

W sytuacji dziedziczenia wielopokoleniowego oraz istnienia pól o tej samej nazwie - problem wieloznaczności nie istnieje, kiedy jedno z pól w danej klasie pochodzi od klasy młodszej w hierarchii – tzn. ma znaczenie bliższe pokrewieństwo.

Dziedziczenie klasy, a zawieranie obiektów składowych

Jeśli mówiąc o obiektach klasy używamy zwrotu: A składa się z B, to mamy zawieranie się obiektów składowych

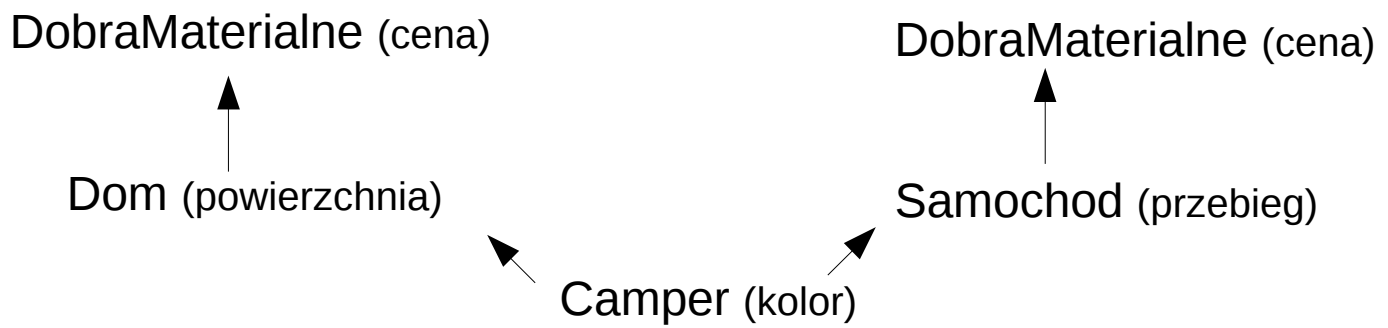
Jeśli mówimy: A jest rodzajem B, to mamy do czynienia z dziedziczeniem klas.



KLASY WIRTUALNE

Wirtualne klasy podstawowe - I

Wirtualne klasy podstawowe są wykorzystywane przy bardzo specyficznym dziedziczeniu klas.

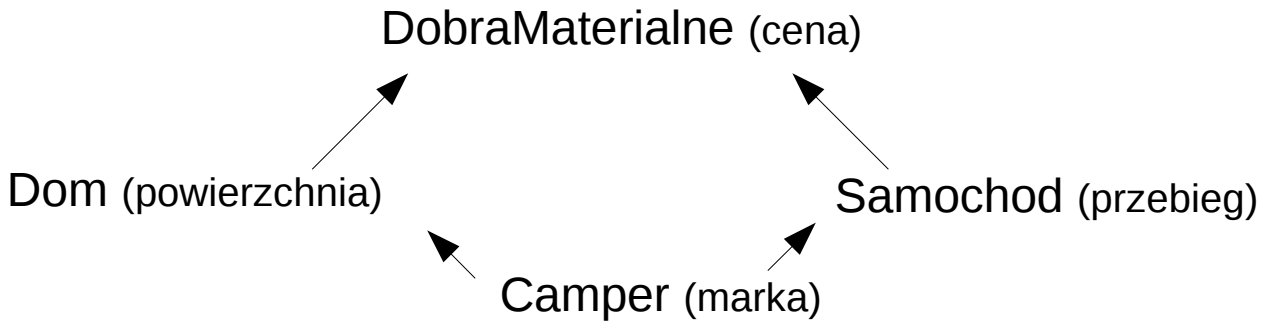


Widać, że obiekt klasy Camper dwukrotnie odziedziczy składowe klasy DobraMaterialne.

Dostęp do składników nie jest jednoznaczny!

Klasę podstawową należy uczynić wirtualną (odziedziczoną inteligentnie).

W hierarchii występuje jeden punkt dzielenia się wspólną informacją.



Wirtualne klasy podstawowe - II

```
class DobraMaterialne {
        protected:
        int cena;
        ...
};
class Dom: public virtual DobraMaterialne {
        protected:
        double powierzchnia;
        ...
};
class Samochod: public virtual DobraMaterialne {
        protected:
        double przebieg;
        ...
};
class Camper: public Dom, public Samochod {
        protected:
        string marka;
        ...
};
```

Wirtualne klasy podstawowe - III

Obiekt klasy Camper zmniejszył się, składniki odziedziczone po klasie DobraMaterialne zostały tylko raz.

Nie ma ryzyka wieloznaczności, mimo dwóch możliwości dojścia do klasy DobraMaterialne (przez Dom oraz Samochod), cel obu dróg jest ten sam: te same komórki w pamięci.

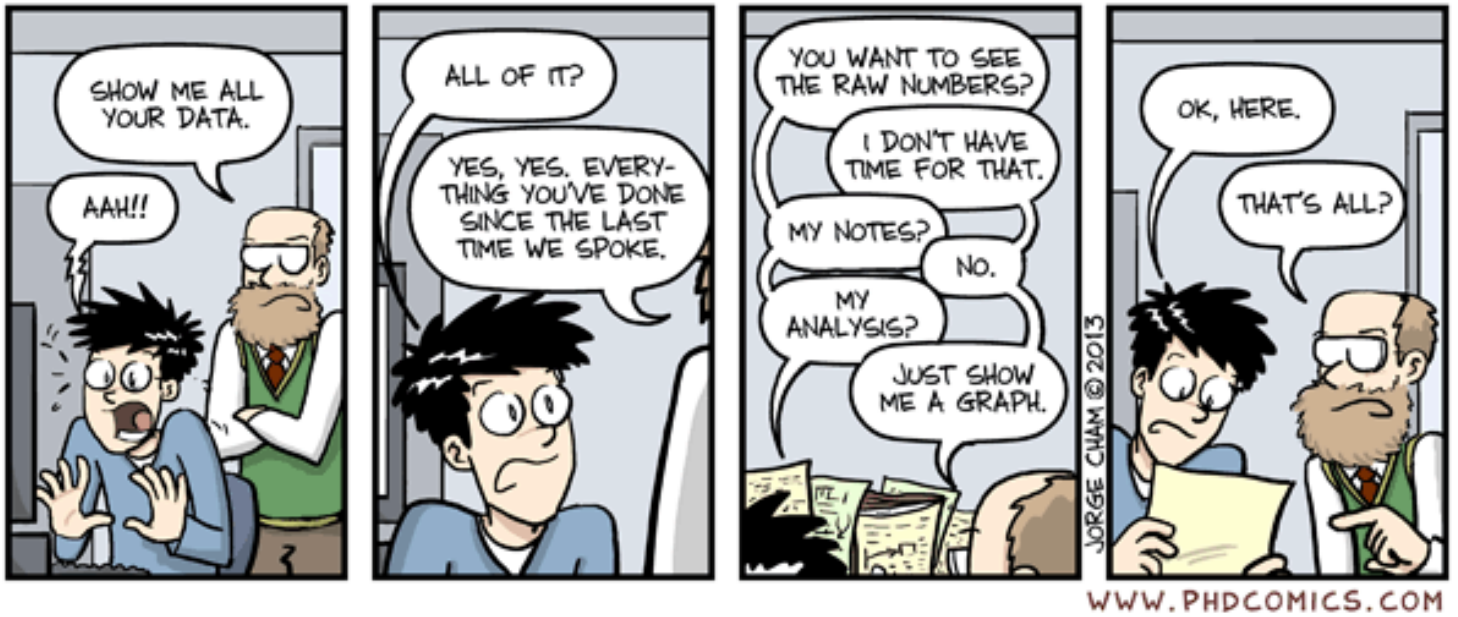
Wirtualny jest dziedziczenie klasy pochodnej, nie klasa sama w sobie!

W sytuacji, kiedy jedna z klas (np. Dom) dziedziczy wirtualnie składniki klasy DobraMaterialne w sposób prywatny, a druga (np. Samochod) w sposób publiczny, to w efekcie klasa Camper dziedziczy te składniki w sposób publiczny (wystarczy, by choć jedno dziedziczenie wirtualne było publiczne).

Wirtualne klasy podstawowe - IV

Za konstrukcję “wirtualnego” dziedzictwa odpowiada klasa najbardziej pochodna. To klasa najbardziej pochodna ma zadbać o uruchomienia konstruktora klasy podstawowej (wirtualnej). Sprowadzi się to do tego, że każda klasa po kolei będzie wywoływać konstruktor klasy najbardziej podstawowej (można przecież stworzyć obiekt klasy Dom, nie tylko obiekt klasy Camper).
W tym przypadku wszelkie uruchomienia konstruktora klasy najbardziej podstawowej z klas nie najbardziej pochodnych zostaną zignorowane (w przypadku kreacji obiektu Camper wywołanie konstruktora klasy DobraMaterialne w klasie Dom lub Samochod zostaną zignorowane).
Dobrym zwyczajem jest definiowanie zawsze konstruktora domniemanego, który w przypadku pominięcia jakiegoś jego wywołania zostanie uruchomiony automatycznie.

W przypadku dziedziczenie niewirtualnego każda klasa mogła uruchomić jedynie konstruktory swoich bezpośrednich klas podstawowych (rodziców).

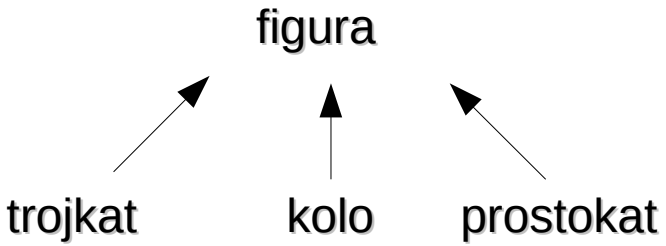


FUNKCJE WIRTUALNE

Funkcje wirtualne - I

Funkcje wirtualne sprawiają, że dziedzicznie nie zachodzi w sposób “bezmyślny”. Dzięki nim program jest orientowany obiektowo (orientuje się według obiektów).

Przykład funkcjonowania funkcji wirtualnych prześledźmy na poniższym przykładzie dziedziczenia:




```
#include <iostream>
#define PI 3.14

using namespace std;

class figura{
        public:
        figura() {}
        virtual ~figura() {}
        double virtual pole() { return -1; }
        //double pole() { return -2; }
};
```

Funkcje wirtualne - II

```
class trojkat: public figura{
        protected:
        double a,h;
        public:
        trojkat(double m=0.0, double n=0.0): a(m), h(n) {} //wywolanie domyslnego
                konstruktora klasy podstawowej mozna w tym przypadku pominac
        //trojkat(double m=0.0, double n=0.0) : figura() { a = m; h = n; }
        //trojkat(double m=0.0, double n=0.0): figura(), a(m), h(n) {} //TO SAMO CO WYZEJ
        ~trojkat(){}
        double pole() { return 0.5*a*h; }
};
class kolo: public figura {
        protected:
        double r;
        public:
        kolo(double a=0.0): r(a) {}
        ~kolo() {}
        double pole() { return PI*r*r; }
};
```

Funkcje wirtualne - III

```
class prostokat: public figura {
        protected:
        double a,b;
        public:
        prostokat(double m=0.0, double n=0.0): a(m), b(n) {}
        ~prostokat() {}
        double pole() { return a*b; }
};
double obliczPole(figura &ref) { return ref.pole(); }

int main() {
        figura f;
        cout<<"Pole figury: "<<f.pole()<<endl; //ujemne
        trojkat t(2.0, 4.0);
        cout<<"Pole trojkata:  "<<t.pole()<<endl; //4
        kolo k(4.0);
        cout<<"Pole kola: "<<k.pole()<<endl; //50.24
        prostokat p(2.0, 3.0);
        cout<<"Pole prostokata: "<<p.pole()<<endl; //6
```

Funkcje wirtualne - IV

```
    cout<<endl<<"Definiujemy wskaznik, ktory bedzie pokazywal na rozne obiekty
          geometryczne.. "<<endl;
    figura *wsk;
    wsk = &f;
    cout<<"Pole figury: "<<wsk->pole()<<endl; //ujemne
    wsk = &t;
    cout<<"Pole trojkata: "<<wsk->pole()<<endl; //4
    wsk = &k;
    cout<<"Pole kola: "<<wsk->pole()<<endl; //50.24
    wsk = &p;
    cout<<"Pole prostokata: "<<wsk->pole()<<endl; //6

    cout<<endl<<"Podobny efekt jest takze w przypadku referencji: "<<endl;
    cout<<"Pole figury: "<<obliczPole(f)<<endl; //ujemne
    cout<<"Pole trojkata: "<<obliczPole(t)<<endl; //4
    cout<<"Pole kola: "<<obliczPole(k)<<endl; //50.24
    cout<<"Pole prostokata: "<<obliczPole(p)<<endl; //6

    return 1;
}
```

Polimorfizm - I

Fragment programu, w którym wywołujemy funkcję wirtualną wykazuje polimorfizm (wielość form).

W przypadku gotowego programu który, należy poszerzyć o nowe klasy (pochodne od klas zdefiniowanych w gotowej części programu nie wymagana jest ingerencja w napisaną część, bowiem to ona zagwarantowała, że w przypadku operowania wskaźnikiem lub referencją do klasy podstawowej odnoszącym się tak naprawdę do obiektu klasy pochodnej została wywołana właściwa funkcja klasy składowej wskaźnika lub referencji odnoszących się do odpowiedniego obiektu).

Język C++ cechuje się rozszerzalnością (ang. e*xtensibility*).

Mechanizm wirtualności jest potrzebny, ale nie ma sensu aplikowanie go we wszystkich przypadkach, a jedynie w tym jego wymagających. Cena, jaką należy zapłacić za korzystanie z funkcji wirtualnych jest taka:

- obiekt klasy z funkcją wirtualną jest większy od obiektu klasy bez takiej funkcji
- podjęcie decyzji, która funkcję należy uruchomić zajmuje dodatkowy czas

Polimorfizm - II

Funkcja wirtualna to specjalny rodzaj funkcji składowej klasy, jej wirtualny charakter ujawnia się jedynie wtedy, gdy jest wywoływana za pomocą referencji lub wskaźnika do obiektu klasy podstawowej. Decyzja, którą funkcję należy wywołać podejmowana jest nie na podstawie typu wskaźnika lub referencji, ale na podstawie tego z obiektem jakiej klasy wskaźnik lub referencja są skojarzone (w funkcjach nie wirtualnych byłoby odwrotnie).

Funkcja składowa jest wirtualna, gdy:

(1) w definicji klasy, przy jej deklaracji stoi słowo virtual (lub)

(2) w jednej z klas podstawowych tej klasy identyczna funkcja składowa ma identyczną sygnaturę (takie same: nazwa, liczba i typ argumentów, typ zwracanego wyniku) oraz jest wirtualna.

Słowo virtual może wystąpić tylko raz w klasie podstawowej I nie musi (choć może) powtarzać się przy analogicznych funkcjach składowych klas pochodnych. Wszystkie funkcje o identycznej sygnaturze w klasach pochodnych będą automatycznie wirtualne.

Funkcją wirtualną (czyli dziedziczoną inteligentnie) może być jedynie składowa funkcja klasy (żadna funkcja globalna).

Słowo virtual pojawia się jedynie przy deklaracji funkcji składowej, przy definicji już nie.

Polimorfizm - III

Klasa pochodna nie musi definiować swojej wersji funkcji wirtualnej, wtedy będzie wywołana wersja z klasy podstawowej.

Jeśli dwie klasy: podstawowa I pochodna mają funkcje wirtualne, ale o różnych zakresach dostępu, to dostęp do tej funkcji w momencie jej wywołania zależy od wskaźnika lub referencji, którym posługujemy się przy wywołaniu (który jest skojarzony z obiektem klasy albo podstawowej albo pochodnej, co definiuje dostęp do funkcji składowej).

Funkcja wirtualna nie może być statyczną (taka wywoływana byłaby na rzecz danej klasy, a nie konkretnego obiektu, a przy funkcjach wirtualnych decyzja, którą funkcję (z której klasy) należy wywołać jest podejmowana na podstawie obiektu, na rzecz którego jest uruchomiona.

Funkcja wirtualna może być zadeklarowana przez inną klasę jako funkcja zaprzyjaźniona, co nie oznacza, że adekwatne funkcje w klasach pochodnych będą także przyjaciółmi klasy, która chce się przyjaźnić z funkcją wirtualną klasy podstawowej.
Przyjaźń nie jest dziedziczna!

Wczesne i późne wiązanie - dygresja

Wczesne wiązanie (ang. *early binding*): następuje wtedy, gdy kompilowany jest program bez funkcji wirtualnych. Na etapie kompilacji odbywa się powiązanie wywołań funkcji z adresami określającymi, gdzie są owe funkcje (nawet jeśli w programie pojawi się blok switch – case, należy zaprogramować podjęcie decyzji.

Późne wiązanie (ang. *late binding*): następuje wtedy, gdy decyzja, którą z funkcji należy wywołać jest podejmowana w trakcie wykonywania programu. W momencie kompilacji nie następuje powiązanie funkcji z adresami określającymi, gdzie te funkcji się znajdują, lecz dopiero w monecie wykonywania programu. W przypadku, kiedy są funkcje wirtualne, decyzja która z funkcji składowych zostanie wywoływana jest podjęta właśnie w momencie wykonania programu.

W przypadku istnienia funkcji wirtualnych także może zajść wczesne wiązanie, gdy:

(1) funkcja wirtualna zostanie wywołana na rzecz obiektu (nie wskaźnika lub referencji)

(2) jawnie zostanie użyty specyfikator zakresu (aby dostać się do funkcji, która została zasłonięta)

```
wsk->klasa::funkcja()
```

(3) zostanie wywołany konstruktor (destruktor) z klasy podstawowej, w której jest wywołana funkcja wirtualna (byłoby irracjonalne, aby konstruktor klasy podstawowej uruchamiał funkcję wirtualną z klasy pochodnej)

Polimorfizm - IV

Funkcja inline, to taka, która w momencie kompilacji nie jest oddzielnie wywoływana, lecz całe jej ciało jest wpisane w program. Takie podejście przeczy idei polimorfizmu.
Jeśli jednak zadeklarujemy funkcję wirtualną jako “w linii”, na etapie późnego wiązania przydomek inline zostanie zignorowany, natomiast na etapie wczesnego wiązania funkcja będzie traktowana jako inline.

Funkcje wirtualne cechują się takimi samymi nazwami I takimi samymi argumentami (ale mają różne zakresy ważności).
Funkcje przeładowane mają taką samą nazwę, taki sam zakres ważności (ale różne listy argumentów)
Mając w jednej klasie dwie przeładowane funkcje, mamy wirtualną tylko taką, której lista argumentów zgadza się z listą argumentów funkcji wirtualnej z klasy podstawowej. Przeładowanie funkcji odbywa się na etapie wczesnego wiązania.

Konstruktor z założenie nie może być wirtualny, jeśli natomiast bardzo zależy nam, aby funkcjonalność wirtualnego konstruktora była zachowana (nie wiemy jakie klasy pochodnej będzie dany obiekt), to należy umiejętnie skonstruować inne funkcje składowe klasy, aby wykonywały zamierzoną czynność.


I just spent a week at home with my son. This is what I learned:
WISDOM FROM MY 3 YEAR OLD
Where you ARE is ALWAYS better than where you're SUPPOSED to be.
bleah.
If you don't like something, SPIT IT OUT.
SING, even if you don't know all the words.
why
why
why
because!
ALWAYS ASK WHY.
JORGE CHAM © 2013
WWW.PHDCOMICS.COM

ATD

Abstrakcyjne typy danych (ATD) - I

ATD (inaczej klasy abstrakcyjne), to takie klasy, które nie reprezentują żadnego konkretnego obiektu.

Klasa abstrakcyjna istnieje tylko po to, aby ją dziedziczyć. Jak zatem uczynić klasę abstrakcyjną?

```
class samochod{
        protected:
        string kolor;
        double cena;
        public:
        samochod(string str="", double c = 0.0) : kolor(str), cena(c) {}
        virtual ~samochod() {}
        void virtual opis() = 0;
};
class osobowy: public samochod {
        protected:
        int ilosc_osob;
        public:
        osobowy(string str="", double c=0.0, int o = 0): samochod(str, c),
                ilosc_osob(o) {}
        ~osobowy() {}
        void opis();
};
```

Abstrakcyjne typy danych (ATD) - II

```
void ciezarowy::opis() { ...}
void osobowy::opis() { ...}
int main() {
        //samochod s("czerwony", 30000); //ZLE!
        osobowy o("zielony", 40000, 5);
        samochod *wsk = &o;
        wsk->opis();
        ...
}
```

Jeśli klasa jest abstrakcyjna, tzn, że nie będzie wykonywana jej funkcja składowa.

```
void virtual opis() = 0;
```

jest funkcją czysto wirtualną (ang. pure virtual). Możliwe jest wykonanie funkcji opis() z klasy pochodnej, ale nie podstawowej. Jeśli nie zostanie dodane = 0, tzn, że funkcją wirtualna nie jest czysto wirtualną i możliwe jest (nawet jeśli nie zamierzamy) tworzenie obiektów z takiej klasy. Uczynienie funkcji wirtualnej czysto-wirtualną gwarantuje, że żaden obiekt takiej klasy nie będzie mógł powstać.

Abstrakcyjne typy danych (ATD) - III

Jeśli w klasie abstrakcyjnej jest składowa funkcja wirtualna (ale nie czysto-wirtualna)

```
void virtual funkcja();
```

to w sytuacji, kiedy w klasie pochodnej nie zostanie zdefiniowana funkcja() zostanie odziedziczona jej wirtualna wersja z klasy podstawowej.

Jeśli natomiast w klasie abstrakcyjnej jest czysto-wirtualna funkcja składowa

```
void virtual funkcja() = 0;
```

To w sytuacji, kiedy w klasie pochodnej nie zostanie zdefiniowana funkcja() zostanie odziedziczona jej czysto-wirtualna wersja z klasy podstawowej, co będzie skutkowało tym, że klasa pochodna stanie się także klasą abstrakcyjną.

WIRTUALNY DESTRUKTOR

Wirtualny destruktor - I

Jeśli w klasie jest choćby jedna funkcja wirtualna (niekoniecznie czysto-wirtualna), to destruktor w klasie także powinien być wirtualny.

Wynika to z faktu, że kiedy wskaźnikiem klasy podstawowej pokazujemy na obiekt klasy pochodnej, po czym chcemy dany obiekt (poprzez wskaźnik na niego pokazujący) unicestwić, to należy użyć destruktora z klasy, z jakiej obiektem pracowaliśmy.

Nie w każdej klasie ma sens tworzenie wirtualnego konstruktora z powodu oszczędności miejsca w pamięci (obiekt klasy z funkcją wirtualną jest większy) oraz czasu wykonywania programu (powiązanie danego obiektu z odpowiednim destruktorem na etapie wykonywania programu jest dłuższe niż jego powiązanie na etapie wiązania wczesnego)

Wirtualny destruktor w klasie podstawowej gwarantuje, że we wszystkich klasach pochodnych destruktor będzie także wirtualny.

W przypadku, kiedy destruktor w klasie podstawowej uczynimy czysto-wirtualnym, a w klasie pochodnej nie zdefiniujemy destruktora, to klasa pochodna stanie się klasą abstrakcyjną oraz niemożliwym będzie kreacja jakiegokolwiek jej obiektu.

Wirtualny destruktor – program - I

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

class figura{
        public:
        figura() { cout<<"Konstruktor figury.."<<endl; }
        virtual ~figura() { cout<<"Destruktor figury.."
                <<endl; }
        double virtual pole() { return -1; }
};
class kolo: public figura {
        protected:
        double *r;
        public:
        kolo(double=0.0);
        kolo(const kolo &);
        kolo& operator=(const kolo &);
        ~kolo();
        double pole() { return (PI * *r * *r); }
};
```

```
class prostokat: public figura {
        protected:
        double *a;
        double *b;
        public:
        prostokat(double=0.0, double=0.0);
        prostokat(const prostokat &);
        prostokat& operator=(const prostokat &);
        ~prostokat();
        double pole() { return (*a * *b); }
};

kolo::kolo(double a) {
        r = new double;
        *r = a;
        cout<<"Konstruktor kola.."<<endl;
}
```

Wirtualny destruktor – program - II

```
kolo::kolo(const kolo &tmp) {
        r = new double;
        *r = *tmp.r;
        cout<<"Konstruktor kola.."<<endl;
}
kolo& kolo::operator=(const kolo &tmp){
        if (&tmp==this) return *this;
        delete r;
        r = new double;
        *r = *tmp.r;
        return *this;
}
kolo::~kolo() {
        delete r; cout<<"Destruktor kola.."<<endl;
}
prostokat::prostokat(double m, double n) {
        a = new double;
        b = new double;
        *a = m;
        *b = n;
        cout<<"Konstruktor prostokata.."<<endl;
}
```

```
prostokat::prostokat(const prostokat &tmp) {
        a = new double;
        b = new double;
        *a = *tmp.a;
        *b = *tmp.b;
        cout<<"Konstruktor prostokata.."<<endl;
}
prostokat& prostokat::operator=(const prostokat &tmp){
        if (&tmp==this) return *this;
        delete b;
        delete a;
        a = new double;
        b = new double;
        *a = *tmp.a;
        *b = *tmp.b;
        return *this;
}
prostokat::~prostokat() {
        delete b;
        delete a; cout<<"Destruktor prostokata.."<<endl;
}
```

Wirtualny destruktor – program - III

```
int main() {
        cout<<endl;
        figura *f = new figura;
        cout<<"Pole figury: "<<f->pole()<<endl<<endl; //ujemne
        figura *k = new kolo(4.0);
        cout<<"Pole kola: "<<k->pole()<<endl<<endl; //50.24
        figura *p = new prostokat(2.0, 3.0);
        cout<<"Pole prostokata: "<<p->pole()<<endl; //6

        cout<<endl; delete p;
        cout<<endl; delete k;
        cout<<endl; delete f;

        return 1;
}
```

Zalety dziedziczenia – podsumowanie 1

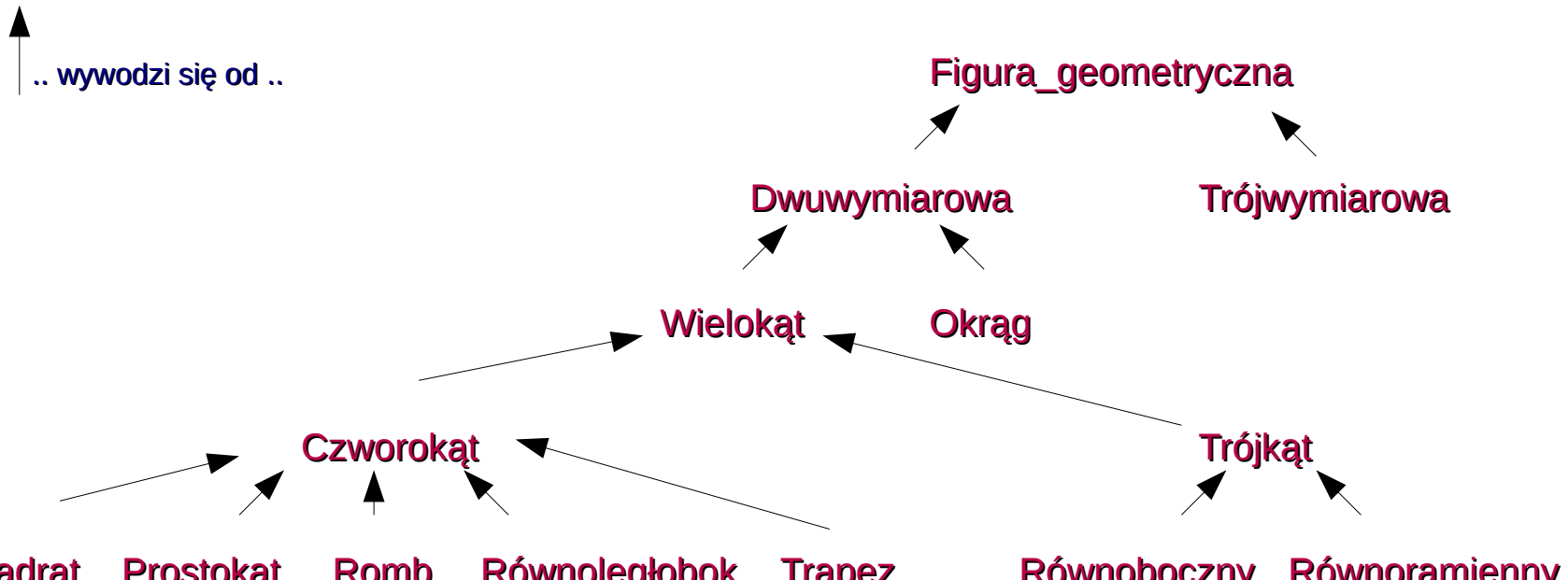
Dziedziczenie jest jedną z największych zalet obiektowo orientowanych języków programowania .

1) Oszczędność pracy

Kiedy należy zdefiniować nową klasę, podobną do tej, jaką już mamy zdefiniowaną, wystarczy zdefiniować jedynie różnice, nie trzeba definiować klasy od nowa. Nie jest konieczna znajomość kodu źródłowego klasy podstawowej, przy dziedziczeniu akceptujemy to, co zawiera klasa podstawowa lub zasłaniamy odziedziczone składniki zdefiniowanymi na nowo w klasie pochodnej. W klasie pochodnej definiujemy także nowe pola, które nie istniały w klasie podstawowej.

2) Hierarchia

Proces dziedziczenia umożliwia wprowadzenie relacji pomiędzy poszczególnymi klasami. Zamiast mnożenia nowych klas bazujemy na zdefiniowanych wcześniej:

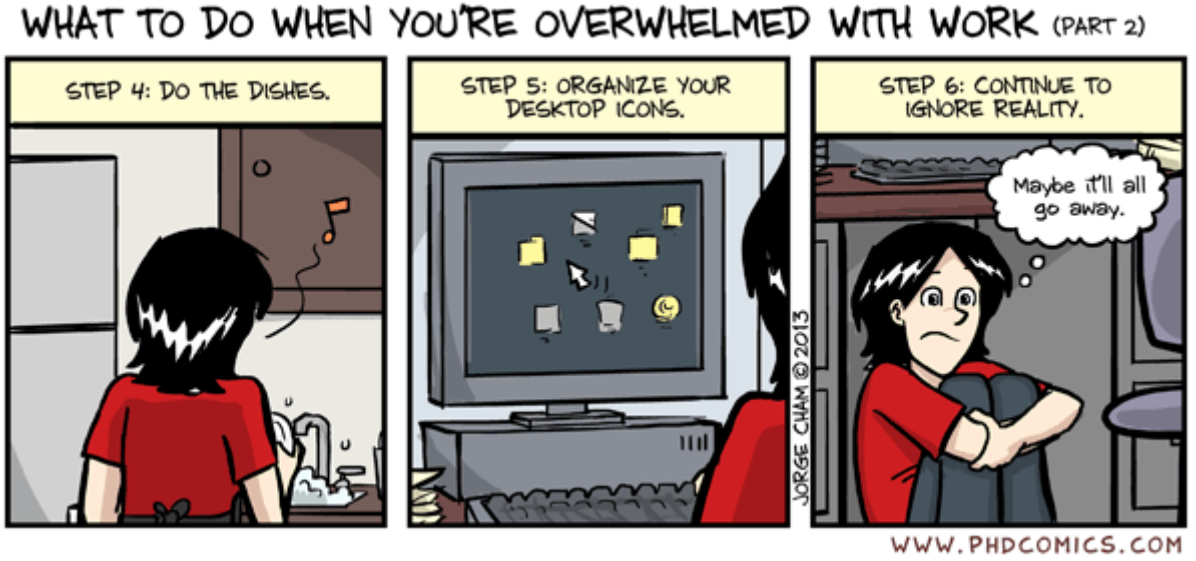


Zalety dziedziczenia – podsumowanie 2

3) Klasy ogólne

Możliwe jest definiowanie takich klas, których obiektów nie będziemy tworzyć, natomiast klasy te przeznaczone są wyłącznie do dziedziczenia. Przykładem takiej klasy może być kolejka. Nie ma obiektu klasy kolejka, ale dopiero ludzie mogą w niej stać.

Dziedziczenie to sposób definiowania nowych klas, który jednocześnie wprowadza pomiędzy klasami relacje. Dziedziczenie nie daje obiektom klas pochodnych żadnych dodatkowych praw wobec obiektów klasy podstawowej. Jeśli taki efekt ma być osiągnięty – należy zadeklarować przyjaźń klasy podstawowej z klasą pochodną.



KONIEC WYKŁADU 9

Nieobowiązkowe zadania do wykonania

1. Zaprojektować i stworzyć “Hurtownię ubrań”. Podzielić produkty na kategorie: “sportowe”, “codzienne”, “wieczorowe”,...

2. Zaprojektować aptekę. Stworzyć klasę abstrakcyjną “Lek” oraz klasy pochodne: “Przeziębienie”, “Witaminy”, “Dzieci”.

Nieobowiązkowe zadania do wykonania

1. Zaprojektować i stworzyć “Hurtownię ubrań”. Podzielić produkty na kategorie: “sportowe”, “codzienne”, “wieczorowe”,...

2. Zaprojektować aptekę. Stworzyć klasę abstrakcyjną “Lek” oraz klasy pochodne: “Przeziębienie”, “Witaminy”, “Dzieci”.