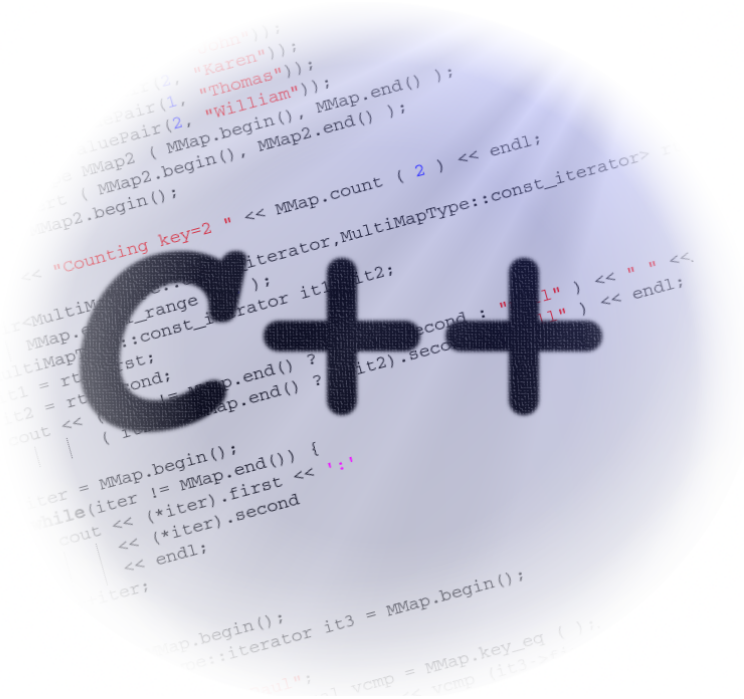
Języki Programowania

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)

.. do kolokwium..

Klasa Pokazowa - I

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

class KlasaPokazowa {
        int tZmA;
        float *tWskB;
        static int tPoleStat;
        int N;
        double *tablica;
        public:
        KlasaPokazowa(int=0, float=0.0, int=10);
        KlasaPokazowa(const KlasaPokazowa&);
        ~KlasaPokazowa();
        KlasaPokazowa& operator=(const KlasaPokazowa&);
        static int MetodaStat();
        void Funkcja();
        friend ostream& operator<<(ostream&, KlasaPokazowa&);
};
ostream& operator<<(ostream&, KlasaPokazowa&);

int KlasaPokazowa::tPoleStat = 10;

KlasaPokazowa::KlasaPokazowa(int a, float b, int n) {
        tZmA =a;
        tWskB =new float;
        *tWskB = b;
        N=n;
        tablica = new double [N];
        for (int i=0; i<N; i++) *(tablica+i) = i;
}
```

Klasa Pokazowa - II

```
KlasaPokazowa::KlasaPokazowa(const KlasaPokazowa &ref) {
        tZmA = ref.tZmA;
        tWskB = new float;
        *tWskB = *ref.tWskB;
        N=ref.N;
        tablica = new double [N];
        for (int i=0; i<N; i++) tablica[i] = ref.tablica[i];

}

KlasaPokazowa::~KlasaPokazowa() {
        delete tWskB;
        delete [] tablica;
}

KlasaPokazowa& KlasaPokazowa::operator=(const KlasaPokazowa &ref) {
        if (&ref == this) return *this;
        delete tWskB;
        delete [] tablica;
        tZmA = ref.tZmA;
        tWskB = new float;
        *tWskB = *ref.tWskB;
        N=ref.N;
        tablica = new double [N];
        for (int i=0; i<N; i++) tablica[i] = ref.tablica[i];
        return *this;
}

int KlasaPokazowa::MetodaStat() {
        return tPoleStat;
}
```

Klasa Pokazowa - III

```
ostream& operator<<(ostream &wyj, KlasaPokazowa &ref ) {
        wyj<<ref.tZmA<<"\t"<<*ref.tWskB<<endl;
        for (int i=0; i<ref.N; i++) wyj<<ref.tablica[i]<<"\t";
        return wyj;
}

int main() {
        int i=5;
        float f=20.0;
        cout<<KlasaPokazowa::MetodaStat()<<endl; //10
        KlasaPokazowa ob(i, f, 7);
        cout<<ob.MetodaStat()<<endl; // 10
        cout<<ob<<endl; // 5 20
                                //0 1 2 3 4 5 6

        KlasaPokazowa ob2, ob3=ob;
        cout<<ob2<<endl; //0 0
                                //0 1 2 3 4 5 6 7 8 9

        cout<<ob3<<endl; // 5 20
                                // 0 1 2 3 4 5 6
        ob3=ob2;
        cout<<ob3<<endl; // 0 0
                                // 0 1 2 3 4 5 6 7 8 9
        return 0;
}
```

ZAGNIEŻDŻONA DEFINICJA KLASY

Zagnieżdżona definicja klasy - wprowadzenie

Jeśli wewnątrz definicji klasy A zamieścimy definicję innej klasy B, to klasa B jest zagnieżdżona w definicji klasy A:

```
class A {
        ...
        class B {
                ...
        };
        ...
};
```

Jest to jedynie deklaracja zagnieżdżona, a nie żaden nowy obiekt!

Definicja klasy wewnętrznej jest lokalna dla klas zewnętrznych, jej zakres ważności jest ograniczony do wnętrza klasy, w której znajduje się.

W bardzo starych wersjach C++ (wcześniejszych niż 2.0), klasa zagnieżdżona miała zakres ważności taki sam, jak klasa zewnętrzna. Zagnieżdżenie było jedynie kwestią notacji. Teraz jako, że definicja klasy wewnętrznej znana jest tylko wewnątrz klasy mamy do czynienia z faktycznym zagnieżdżeniem. Jedynie wewnątrz klasy A można tworzyć obiekty klasy B!

Definicja klasy zagnieżdżonej - I

Gdzie znajdują się definicje składowych funkcji klasy zagnieżdżonej? (zazwyczaj definicje funkcji składowych znajdują się wewnątrz definicji klasy, np. jako funkcje inline albo na zewnątrz definicji klasy).

W przypadku klasy zagnieżdżonej definicje składowych funkcji klasy zagnieżdżonej znajdują się na zewnątrz obu klas. Kompilator I tak rozpozna dla której klasy są przeznaczone.

Nie wolno definiować funkcji przed końcem deklarowanej klasy A:

Przyjęta konwencja w C i C++ mówi:

Funkcje nie mogą być definiowane lokalnie wewnątrz innych funkcji i bloków.

Definicja klasy zagnieżdżonej - II

```
class computer {
        int memory;
        ...
        class monitor {
                ...
                double inches;
                string type;
                double getinches();
                string gettype();
                ...
        };
};
...
double computer::monitor::getinches() { return inches;}
string computer::monitor::gettype() { return type; }
...
```

Klasa o definicji zagnieżdżonej ma zakres ważności ograniczony do klasy, w jakiej się znajduje!

Definicja klasy zagnieżdżonej - III

Klasa wewnętrzna może używać zdefiniowanych w klasie zewnętrznej:

- nazw typów (typedef);
- typów wyliczeniowych (enum)
- publicznych składników statycznych

Do innych składników klasy zewnętrznej w klasie wewnętrznej musimy się odnosić w identyczny sposób, jak z każdej innej obcej klasy:

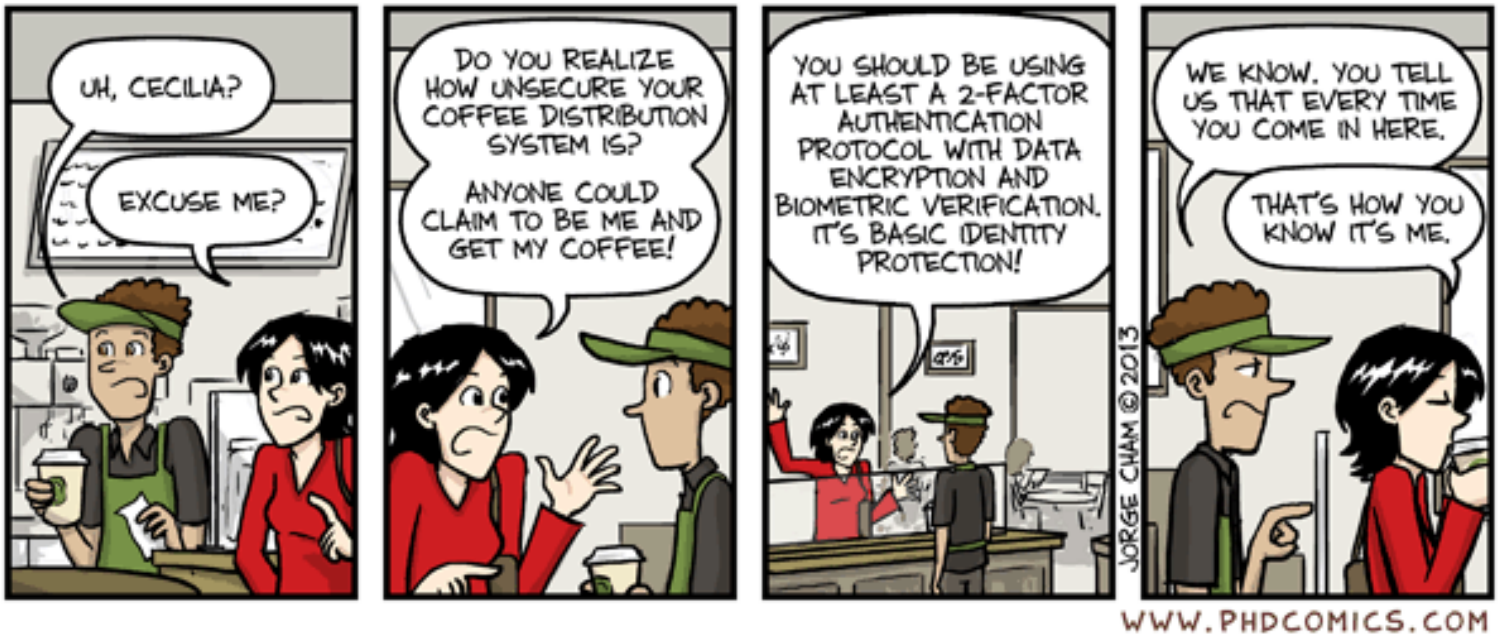
obiekt.skladnik;

wskaznik->skladnik;

referencja.skladnik;

Jeśli klasa wewnętrzna deklaruje przyjaźń z jakąś funkcją zewnętrzna, to nie oznacza, że ta funkcja zewnętrzna przyjaźni się z klasą wewnętrzną.

Korzyści z zagnieżdżania klasy? Jest ona nieznana nigdzie indziej, nie wolno tworzyć jej obiektów poza zakresem danej klasy.



DZIEDZICZENIE

ISTOTA DZIEDZICZENIA

Istota dziedziczenia - I

Dziedziczenie jest techniką programistyczną umożliwiającą tworzenie nowych klas przy wykorzystaniu klasy istniejącej wcześniej.

```
enum nadwozie  {sedan, hatchback, liftback, kombi, coupe, kabriolet};
enum kolor  {black, red, white, green, blue, orange};
enum model  {polo, golf, passat, sharan, beetle};
class samochod { //klasa podstawowa
        public:
        nadwozie nn;
        kolor kk;
        double cena;
        void info();
        ...
};
class volkswagen : public samochod { //klasa pochodna
        public:
        model marka;
        void info();
        ...
};
```

Istota dziedziczenia - II

W klasie pochodnej możemy:

- zdefiniować nowe dane składowe;
- zdefiniować nowe funkcje składowe
- zdefiniować na nowo składnik, który istnieje już w klasie podstawowej

Do składników odziedziczonych (nie wymienionych ponownie w ciele klasy pochodne) odnosimy się tak, jakby znajdowały się w ciele klasy pochodnej.

Nie jest konieczne "dziedziczenie" wszystkich pól klasy podstawowej:

- można zdefiniować niektóre pola w klasie podstawowej jako private;
- zdefiniować nowe pola o tej samej nazwie w klasie pochodnej (ale wtedy nowe składniki jedynie zasłonią te odziedziczone)

Jeśli z jakiegoś powodu chcemy odnieść się do składnika klasy, który został zasłonięty:

```
samochod s;
s.info() // wywolanie funkcji skladowej z klasy pochodnej
s.samochod::info(); // wywolanie funkcji skladowej z klasy podstawowej
```

Obiekty nie mogą tworzyć obiektów pochodnych.

Dziedziczenie dotyczy jedynie typów danych (czyli klas).

DOSTĘP DO SKŁADNIKÓW

Dostęp do składników – I (jak klasa podstawowa decyduje co i jak jest dziedziczne)

```
enum nadwozie  {sedan, hatchback, liftback, kombi, coupe, kabriolet};
enum kolor  {black, red, white, green, blue, orange};
enum model  {polo, golf, passat, sharan, beetle};
class samochod { //klasa podstawowa
        private:
        int ilosc_drzwi;
        protected:
        nadwozie nn;
        kolor kk;
        double cena;
        public:
        void info();
        ...
};
class volkswagen : public samochod { //klasa pochodna
        protected:
        model marka;
        public:
        void info();
        ...
};
```

Dostęp do składników – II (jak klasa podstawowa decyduje co i jak jest dziedziczne)

Specyfikator dostępności public w klasie podstawowej oznacza, że składowe te są dostępne dla klas pochodnych oraz innych funkcji globalnych (są widoczne wszędzie).

Specyfikator dostępności protected w klasie podstawowej oznacza, że te składniki są zastrzeżone dla klas pochodnych. Spoza klasy podstawowej i pochodnej nie są dostępne (są jak składniki prywatne), natomiast klasy pochodne mają dostęp do pól chronionych.

Specyfikator dostępności private w klasie podstawowej oznacza, że składowe te nie są dostępne ani w klasie pochodnej, ani w żadne innej funkcji zewnętrznej (są dostępne jedynie w klasie podstawowej)

Klasa podstawowa poprzez odpowiednią specyfikację dostępu może decydować, które składniki zamierza udostępniać wszystkim (public), które rezerwuje jedynie na potrzeby klas pochodnych (protected), a którym składnikom zabiera jakąkolwiek możliwość dostępu spoza swojej klasy (private).

Dostęp do składników – III (jak klasa pochodna decyduje co i jak jest dziedziczne)

Klasa pochodna także decyduje jaki chce mieć dostęp do składników jakie dziedziczy!

Jeśli jakieś odziedziczone pole nie jest prywatne (ale chronione lub publiczne), to klasa pochodna decyduje jaki specyfikator dostępu nada tym odziedziczonym składnikom.Wartością domniemaną specyfikatora dostępności jest private

```
class volkswagen : public samochod { ...}

class volkswagen : protected samochod { ...}

class volkswagen : private samochod { ...}
class volkswagen : samochod { ...} //domyslna wartoscia jest private
```

Dostęp do składników – IV (jak klasa pochodna decyduje co i jak jest dziedziczne)

Specyfikator dostępu public mówi o tym, że pola:

public dziedziczone są jako public

protected dziedziczone są jako protected

private dziedziczone są jako private

Specyfikator dostępu protected mówi o tym, że pola:

public dziedziczone są jako protected

protected dziedziczone są jako protected

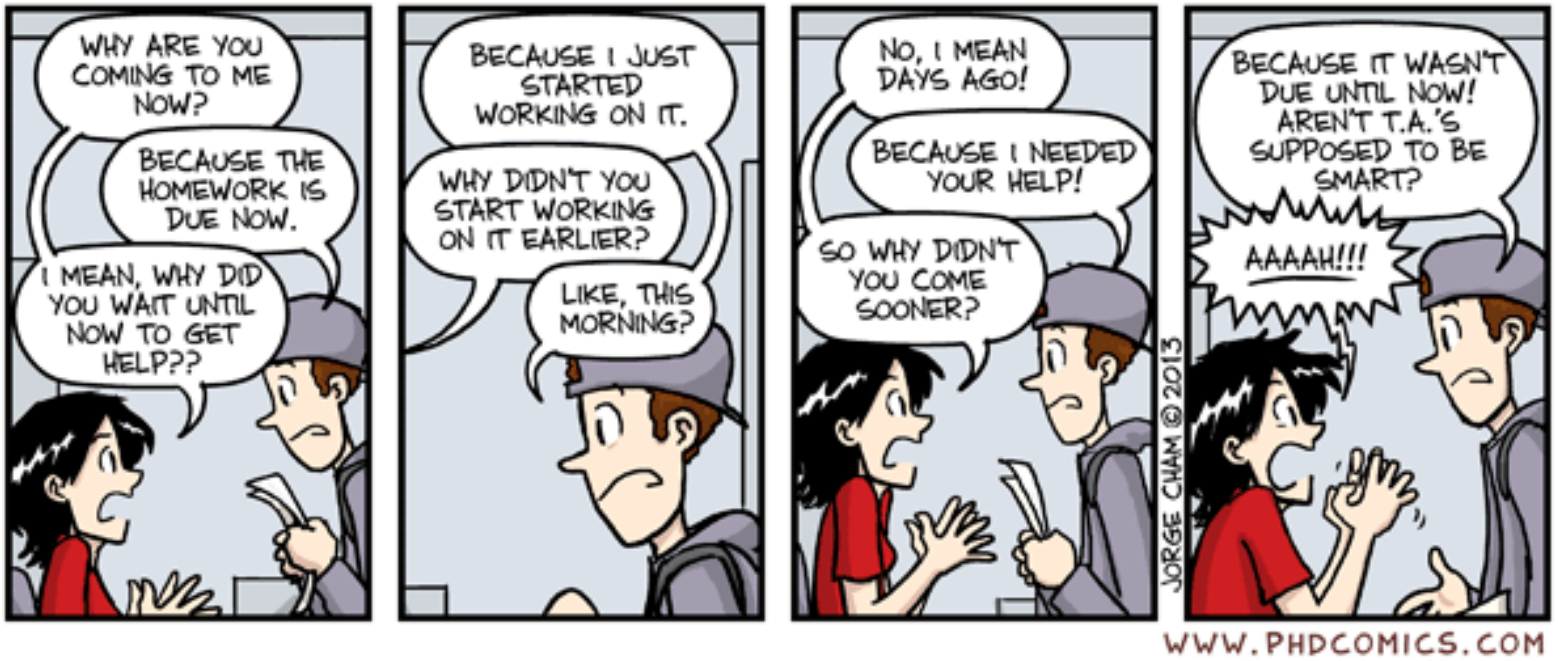
private dziedziczone są jako private

Specyfikator dostępu private mówi o tym, że pola:

public dziedziczone są jako private

protected dziedziczone są jako private

private dziedziczone są jako private



CO NIE JEST DZIEDZICZONE?

Co nie jest dziedziczone?

Mimo specyfikatorów dostępności nie wszystkie składniki klasy są dziedziczone:

(1) konstruktory

(2) destruktor

(3) operator przypisania (operator=)

Ad. 1: Obiekt klasy pochodnej to nie tylko obiekt klasy podstawowej, ale ma zawierać dodatkowe pola. O tych nowych polach z klasy pochodnej konstruktor z klasy podstawowej nie ma prawa nic wiedzieć (gdyby został odziedziczony, to zainicjowane zostałyby pola z klasy podstawowej, a nowe pola z klasy pochodnej już nie). Należy samodzielnie stworzyć konstruktor(y) w klasie pochodnej (co nie oznacza, że w ciele konstruktora klasy pochodnej nie można wywołać konstruktora klasy podstawowej, a następnie zainicjować nowe pola).

Ad. 2: Z tego samego powodu sprawiają, że nie jest dziedziczony operator=. Ten operator ma służyć przypisaniu wszystkich pól w klasie (tych odziedziczonych oraz tych nowych). Jeśli mimo wszystko zapomnimy dopisać operator=, to zostanie on wygenerowany automatycznie przepisując składowe klasy składnik po składniku (nie zawsze polecane, zwłaszcza w sytuacji, kiedy polami w klasie są wskaźniki).

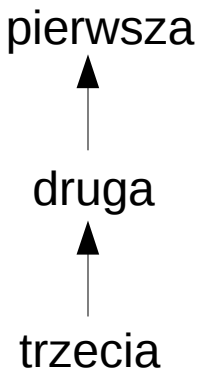
Ad. 3: Działanie destruktora jest ściśle powiązane z działaniem konstruktora. Skoro konstruktor inicjuje pola odziedziczone, jak i te nowe, to destruktor powinien także je uwzględniać w miarę konieczności.

DZIEDZICZENIE KILKUPOKOLENIOWE

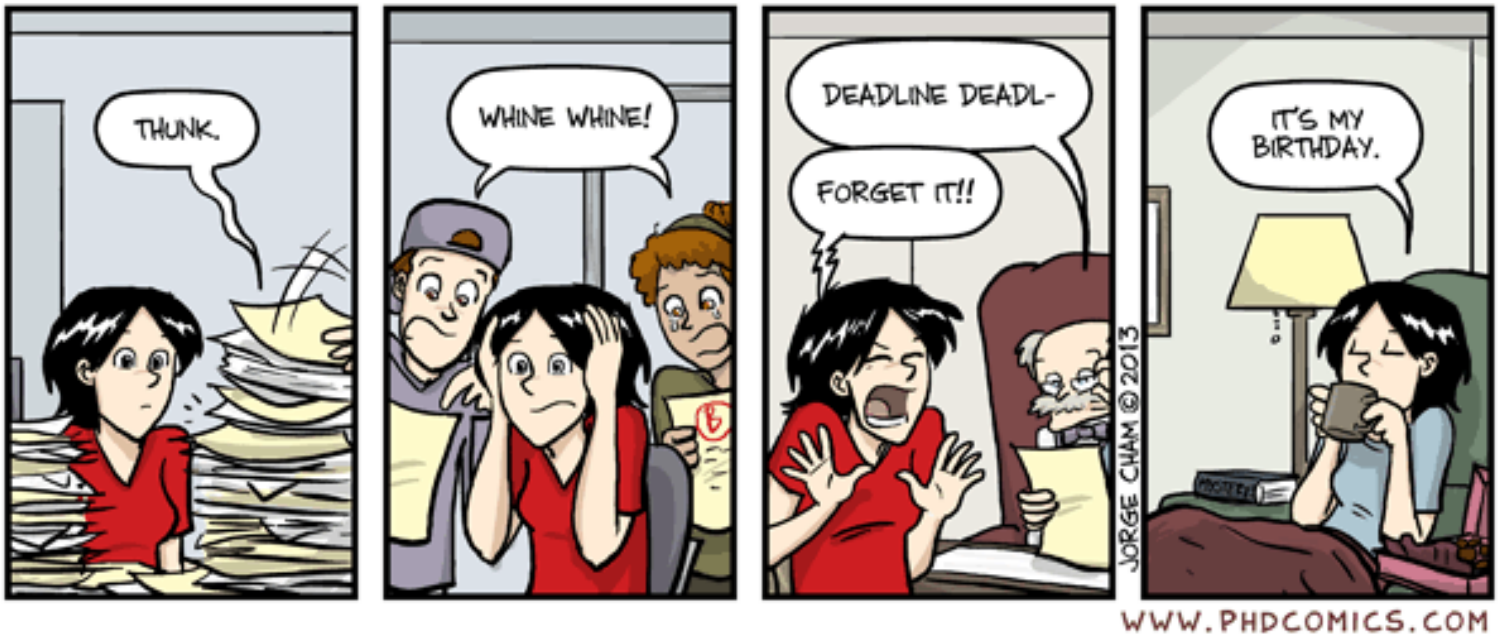
Dziedziczenie kilkupokoleniowe

Dziedziczenie może być kilkupokoleniowe tak, że klasa pochodna jest jednocześnie klasą podstawową dla następnej klasy

```
class pierwsza {
      ...
};

class druga: public pierwsza {
      ...
};

class trzecia: public druga {
      ...
};
```



Dziedziczenie publiczne sprawia, że składniki dostępne w klasie podstawowej są dostępne we wszystkich klasach pochodnych. Zastosowanie dziedziczenia prywatnego sprawia, że kolejne pokolenia do składników tych nie będą miały dostępu.



ZALETY DZIEDZICZENIA

Zalety dziedziczenia - I

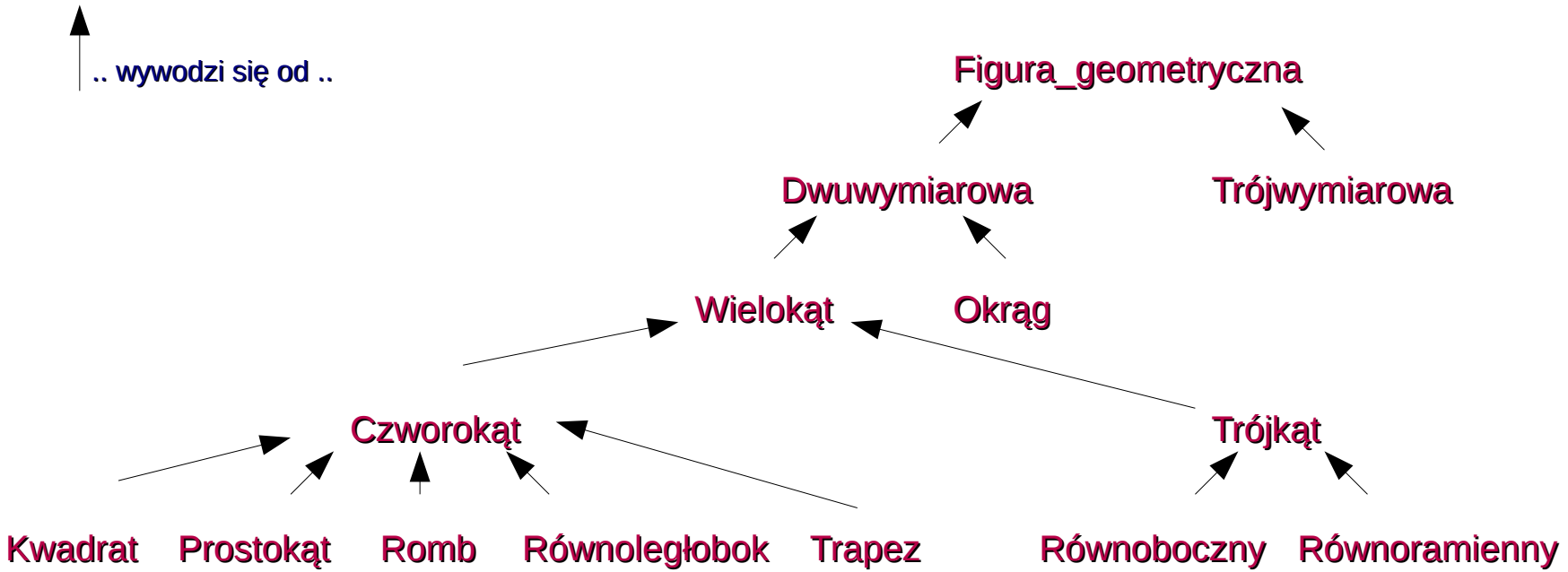
Dziedziczenie jest jedną z największych zalet obiektowo orientowanych języków programowania .

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:



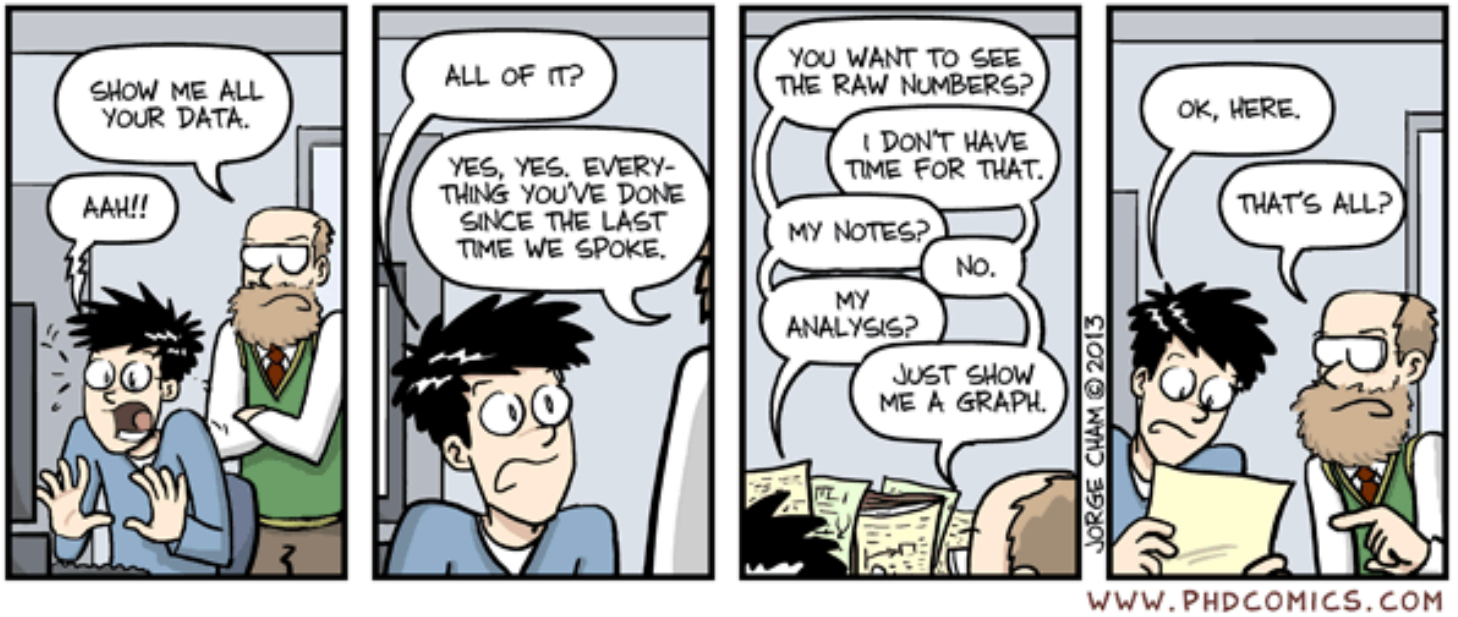
Zalety dziedziczenia - II

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.



KOLEJNOŚĆ WYWOŁANIA KONSTRUKTORÓW

Kolejność wywoływania konstruktorów - I

W obrębie klasy pochodnej “zawiera się” obiekt klasy podstawowej (tak, jak w obiekcie Romb tkwi Czworokąt).

Konstrukcja obiektu klasy pochodnej składa się z:

- konstrukcji obiektu klasy podstawowej,
- dodania pól obecnych tylko w klasie pochodnej

W pierwszej kolejności rusza do pracy konstruktor klasy podstawowej (w dziedziczeniu wielopokoleniowym najpierw wykonują się konstruktory klas najbardziej podstawowych, aż po konstruktory klas najmłodszych). Jeśli dodatkowo w klasie są składniki będące innymi klasami, to konstruktory tych klas ruszą do pracy zaraz po uruchomieniu konstruktorów klas znajdujących się wyżej w hierarchii. Dopiero na samym końcu zacznie wykonywać się własny konstruktor klasy.

Wywołanie konstruktora klasy podstawowej można umieścić w liście iniclalizacyjnej konstruktora klasy pochodnej. (Lista inicjalizacyjna znajduje się przy definicji konstruktora)

Jeśli klasa podstawowa nie ma zdefiniowanego konstruktora, wtedy oczywistym jest, że go nie wywołujemy.

Jeśli klasa podstawowa ma konstruktor domniemany, to przy nie wywołaniu konstruktora klasy podstawowej w konstruktorze klasy pochodnej zostanie on uruchomiony automatycznie.

Kolejność wywoływania konstruktorów - II

Pominięcie wywołania konstruktora klasy podstawowej przy definiowaniu konstruktora klasy pochodnej może mieć miejsce w dwóch przypadkach:

(1) klasa podstawowa nie ma konstruktora

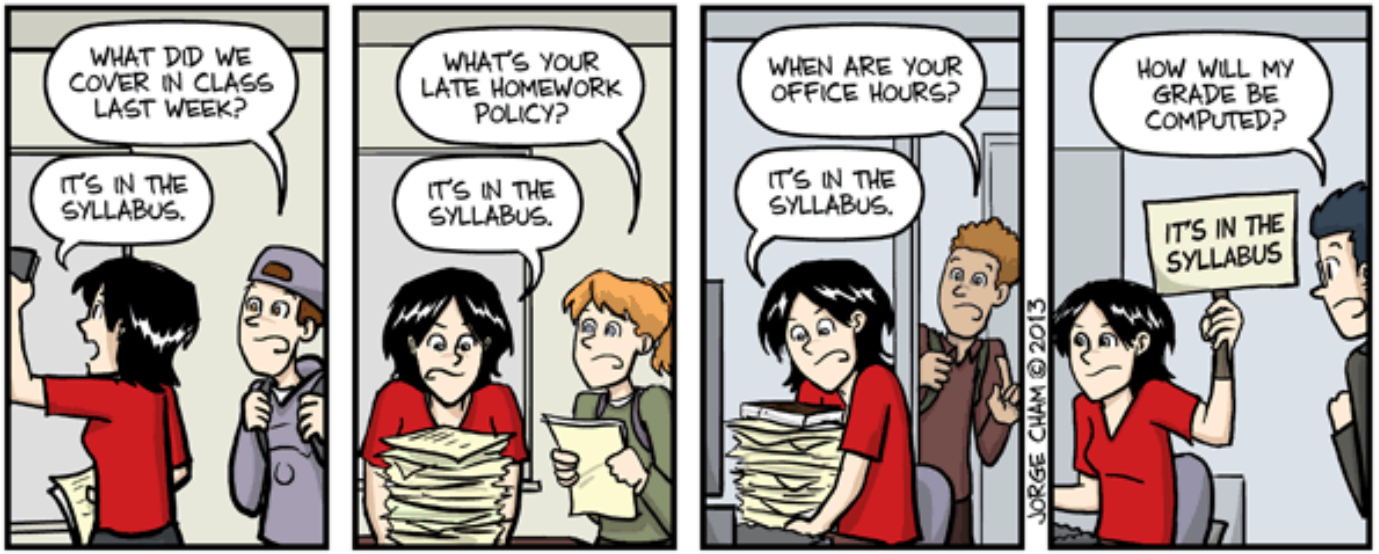
(2) klasa podstawowa ma konstruktor domniemany

Przykład: w jakiej kolejności wykonują się konstruktory..

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

class monitor {
        protected:
        double inches;
        public:
        monitor(double ii): inches(ii) { cout<<"class monitor - constructor running.."<<endl; }
        ~monitor() { cout<<"class monitor - destructor running.."<<endl; }
};
```

Kolejność wywoływania konstruktorów - III

```
class computer {
        protected:
        string processor;
        int memory, harddrive;
        public:
        computer(string ss="", int mm = 0, int dd= 0): processor(ss), memory(mm), harddrive(dd)
                {       cout<<"class computer - constructor running.. "<<endl; }
        ~computer() { cout<<"class computer  - destructor running.. "<<endl; }
};
class notebook : public computer {
        protected:
        double price;
        monitor screen;
        public:
        notebook(string ss="", int mm = 0, int dd= 0, double pp = 0.0, double ii=0.0):
                computer(ss,mm,dd), price(pp), screen(ii)
                { cout <<"class notebook - constructor running.."<<endl; }
        ~notebook() { cout <<"class notebook - destructor running.."<<endl; }
};
int main() {
        notebook c1("Intel", 8, 512);
        return 1;
}
```



PRZYPISANIE I INICJALIZACJA OBIEKTÓW W WARUNKACH DZIEDZICZENIA

Przypisanie i inicjalizacja przy dziedziczeniu - I

Konstruktor kopiujący (jak i żaden konstruktor!) oraz operator przypisania nie są dziedziczne. Obiekt klasy pochodnej to obiekt klasy podstawowej powiększony o nowe pola składowe (z klasy pochodnej).

Stworzenie konstruktora (kopiującego) oraz operatora przypisania to:

(1) przypisanie (lub inicjalizacja) części odziedziczonej (warto skorzystać z konstruktora i / lub operatora przypisania z klasy podstawowej – o ile są one tylko dostępne)

(2) przypisanie (lub inicjalizacja) części nowej

(I) Klasa pochodna nie definiuje swojego operatora przypisania (konstruktora kopiującego)

Zostanie wygenerowany automatycznie operator przypisania (konstruktor kopiujący) przypisujący "składnik po składniku":

Operator przypisania:

```
klasa& klasa::operator=(klasa &);
```

Konstruktor kopiujący:

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

Przypisanie i inicjalizacja przy dziedziczeniu - II

a) Jeśli klasa podstawowa ma operator przypisania (konstruktor kopiujący), to zostanie on użyty w celu stworzenia operatora przypisania (konstruktora kopiującego) klasy pochodnej dla odziedziczonych składników.

b) Jeśli klasa podstawowa ma prywatny operator przypisania (konstruktor kopiujący), to kompilator respektując prawo klasy podstawowej do prywatności nie wygeneruje dla klasy pochodnej operatora przypisania (konstruktora kopiującego).

c) Jeśli klasa będzie miała jakiś składnik stały (typu const) lub referencję, to operator przypisania (konstruktor kopiujący) także nie zostanie wygenerowany.

d) Inicjalizacja I przypisanie według stałego obiektu wzorcowego może odbyć się jedynie, kiedy w argumentach operatora przypisania (konstruktora kopiującego) pojawi się deklaracja“nietykalności” (const). Automatycznie wygenerowany operator przypisania (konstruktor kopiujący) będzie taki jedynie w sytuacji, kiedy wszystkie klasy podstawowe oraz składniki klasy mają takie operatory przypisania (konstruktory kopiujące).

Operator przypisania:

```
klasa& klasa::operator=(const klasa &);
```

Konstruktor kopiujący:

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

Przypisanie i inicjalizacja przy dziedziczeniu – program - I

(II) Klasa pochodna definiuje operator przypisania (konstruktor kopiujący):

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

class computer {
        protected:
        string processor;
        int memory, harddrive;
        public:
        computer(string ss="", int mm = 0, int dd= 0):
                processor(ss), memory(mm),
                harddrive(dd) {}
        computer(const computer& );
        computer& operator=(const computer& );
        ~computer() {}
};
```

```
class notebook : public computer {
        protected:
        double price;
        double inches;
        public:
        notebook(string ss="", int mm = 0, int dd= 0,
                double pp = 0.0, double ii=0.0):
                computer(ss,mm,dd), price(pp), inches(ii) {}
        ~notebook() { }
        notebook(const notebook& );
        notebook& operator=(const notebook& );
        friend ostream& operator<<(ostream& screen,
                const notebook& obj);
};

computer::computer(const computer& com) {
        processor = mikroprocesor;
        memory = com.memory;
        harddrive = com.harddrive;
}
```

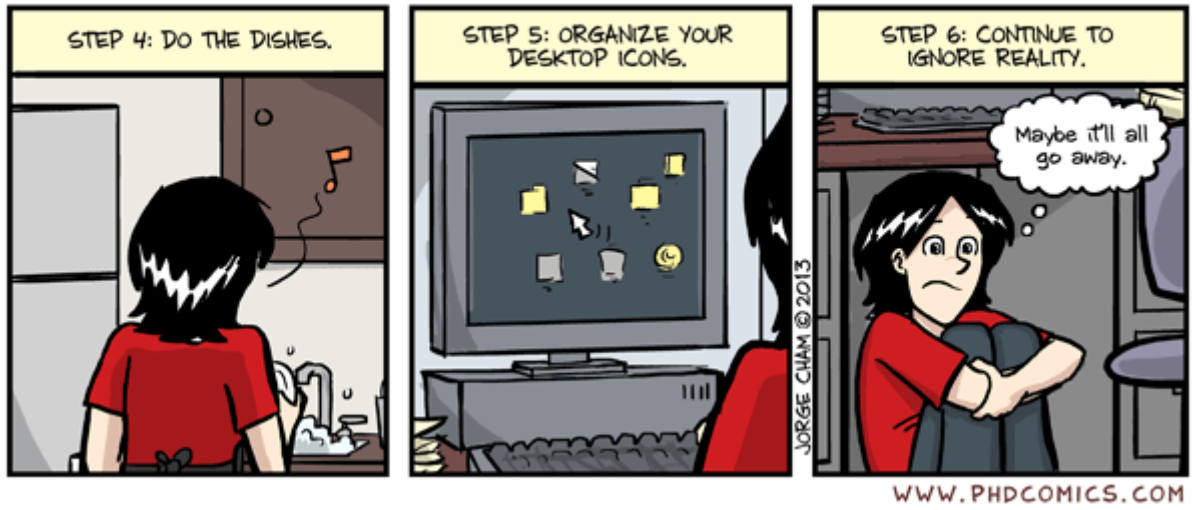
Przypisanie i inicjalizacja przy dziedziczeniu – program - II

```
computer& computer::operator=(const computer& com) {
        processor = com.processor;
        memory = com.memory;
        harddrive = com.harddrive;
        return *this;
}

notebook::notebook(const notebook& note): computer(note) {
        price = note.price;
        inches = note.inches;
}
```

```
notebook& notebook::operator=(const notebook& note) {
        (*this).computer::operator=(note);

        //computer *com = this; //TO SAMO
        //(*com) = note;

        //computer &ref = *this; //TO SAMO
        //ref = note;

        price = note.price;
        inches = note.inches;
        return *this;
}
ostream& operator<<(ostream& screen,
        const notebook& obj) {
        screen<<"This is computer:"<<endl<<"Processor: "
                <<obj.processor<<"\t        Memory: "<<
                obj.memory<<"\t Harddrive: "<<obj.harddrive
                <<"\t Screen: "<<obj.inches<<"\t Price: "<<
                obj.price<<endl;
        return screen;
}
```

Przypisanie i inicjalizacja przy dziedziczeniu – program - III

```
int main() {
        notebook c1("Intel", 8, 512, 4200, 17);
        cout<<"We have first computer: "<<endl;
        cout<<c1<<endl;
        notebook c2 = c1;
        cout<<"We have second computer being a copy of the first one: "<<endl;
        cout<<c2<<endl;
        notebook c3("Athlon", 4, 250, 2500, 15);
        cout<<"We have third computer: "<<endl;
        cout<<c3<<endl;
        c3 = c1;
        cout<<"Now third computer is the same as the first one: "<<endl;
        cout<<c3<<endl;
        cout<<"We have fourth computer: (const)"<<endl;
        const notebook c4("Intel", 4, 256, 2000, 14);
        cout<<c4<<endl;
        c3 = c4;
        cout<<"Third computer is now the same as the fourth one: "<<endl;
        cout<<c3<<endl;
        return 1;
}
```



KONIEC WYKŁADU 7/8

Nieobowiązkowe zadania do wykonania

1. Obliczenie długości wektora o współrzędnych wprowadzonych przez użytkownika

   Należy przeciążyć operatory: strumieniowe (wejścia i wyjścia) punktów lub wektora

2. Napisanie programu liczącego sumę, różnicę, iloczyn wektorów (przeciążyć operatory: +, - *)

3. Powyższe zadania można dodatkowo skomplikować, jeśli polami w klasach będą wskaźniki.

   Należy wtedy zadbać o odpowiednie zarządzenie pamięcią. Konieczne będzie przeciążenie Operatora przypisania (=), jak również napisanie konstruktora kopiującego.

---

4. Napisać program „sklep komputerowy”, gdzie klasą podstawową będzie „komputer”, klasą pochodną „laptop”. Należy przeciążyć w obu klasach operatory strumieniowe. Napisać program także w wersji zawierającej wskaźniki jako pola w klasie (podstawowe), co wymusi napisanie konstruktorów kopiujących oraz operatora przypisania.