Języki Programowania

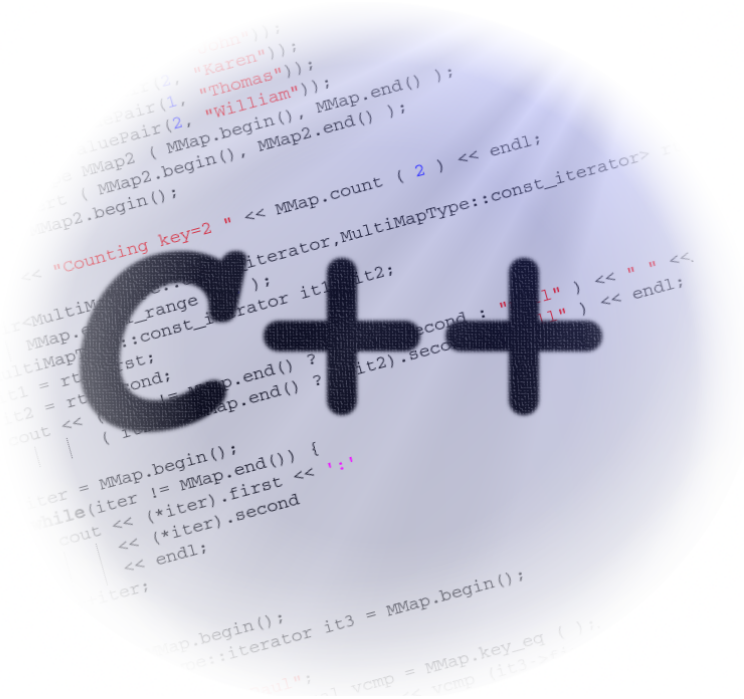
Prowadząca:
dr inż. Hanna Zbroszczyk

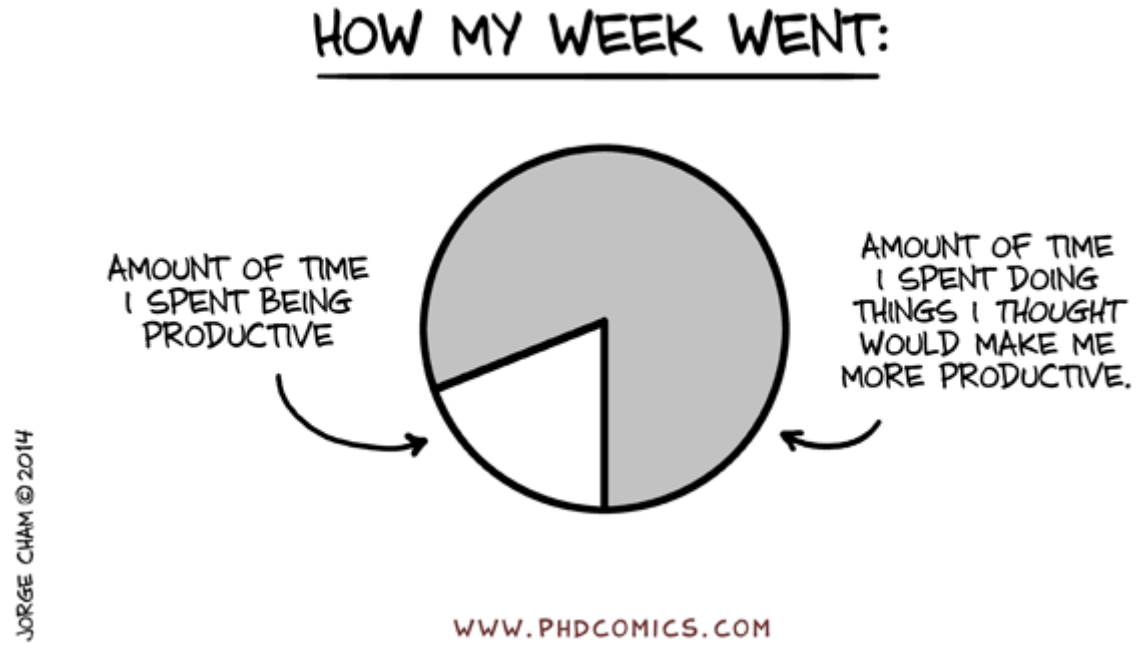
e-mail: *hanna.zbroszczyk@pw.edu.pl*
tel: +48 22 234 58 51

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)





PRZESYŁANIE DO FUNKCJI ARGUMENTÓW BĘDĄCYCH OBIEKTAMI – PRZESYŁANIE PRZEZ WARTOŚĆ

Przesyłanie obiektu przez wartość - I

Przesłanie przez domniemanie: przesłanie przez wartość

(tak samo jak z danymi typów int czy double).

Przesłanie przez wartość znaczy: cały obiekt służy do inicjalizacji swojej kopii wewnątrz funkcji.

Jeśli obiekt jest duży to proces kopiowania może trwać dłużej

(wielokrotne wysyłanie przez wartość może wpłynąć na zwolnienie programu)

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

enum zawod {fizyk= 1, matematyk, informatyk, ekonomista, elektronik, polonista,
        historyk, prawnik, psycholog, lekarz};

class czlowiek {
        zawod kto;
        int wiek;
        public:
        void zapisz(zawod, int);
        void wypisz() { cout<<"Oto "<<kto<<" w wieku "<<wiek<<" lat"<<endl;}
};
```

Przesyłanie obiektu przez wartość - II

```
void czlowiek::zapisz(zawod nazwa, int ile) {
        kto = nazwa;
        wiek = ile;
}

void przedstaw(czlowiek czl) { //zwykla funkcja
                majaca w argumencie obiekt
        cout<<"Przedstawiamy Panstwu: "<<endl;
        czl.wypisz();
}
void info(){
        cout<<"Zawody: "<<endl;
        cout<<"1 - fizyk"<<endl<<"2 – matematyk"
        <<endl<<"3 – informatyk"<<endl
        <<"4 - ekonomista"<<endl;
        cout<<"5 - elektronik"<<endl<<"6 – polonista"
        <<endl<<"7 – historyk"<<endl<<"8 - prawnik"<<endl;
        cout<<"9 - psycholog"<<endl<<"10 - lekarz"<<endl;
}
```

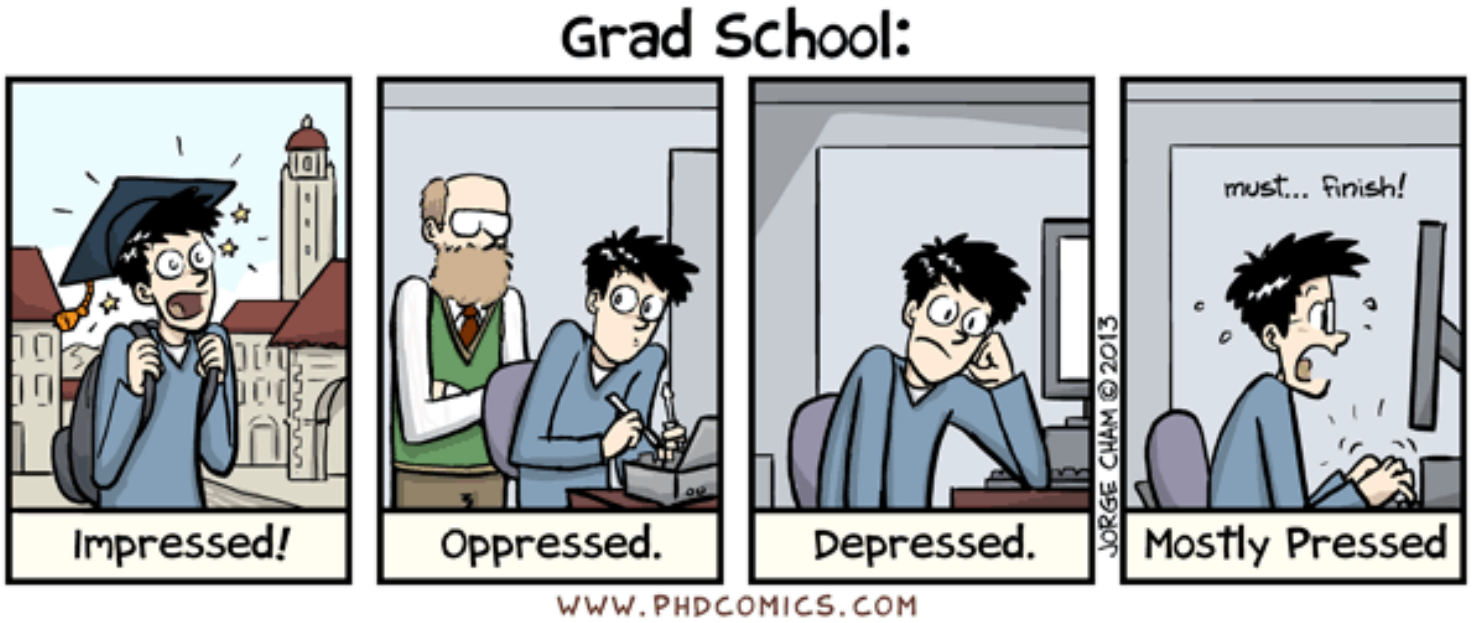
```
int main() {
        czlowiek ff, mm, ll;

        info();

        ff.zapisz(fizyk, 33);
        mm.zapisz(matematyk, 32);
        ll.zapisz(lekarz, 35);

        przedstaw(ff); //przeslanie obiektu przez wartosc
        przedstaw(mm); //przeslanie obiektu przez wartosc
        przedstaw(ll); //przeslanie obiektu przez wartosc
        return 1;
}
```

PRZESYŁANIE DO FUNKCJI ARGUMENTÓW BĘDĄCYCH OBIEKTAMI – PRZESYŁANIE PRZEZ REFERENCJĘ

Przesyłanie obiektu przez referencję

```
void przedstaw(czlowiek &czl) { //przeslanie obiektu przez referencje
        cout<<"Przedstawiamy Panstwu: "<<endl;
        czl.wypisz();
}
...
int main() {
        ...
        przedstaw(ff); //przeslanie obiektu przez referencje
        przedstaw(mm); //przeslanie obiektu przez referencje
        przedstaw(ll); //przeslanie obiektu przez referencje
        return 1;
}
```

Zastosowanie przesłania przez referencję powoduje, że sama funkcja nie pracuje na kopii obiektu, ale na jego oryginale.



KONSTRUKTORY, DESTRUKTORY - WSTĘP

Wprowadzenie do konstruktorów - I

Celem konstruktora NIE jest konstrukcja obiektu!

Celem konstruktora JEST NADANIE WARTOŚCI POCZĄTKOWEJ obiektowi, jaki jest tworzony.

(Do tej pory, w celu przypisania pewnych wartości poszczególnym składowym klasy należało wywołać na rzecz danego obiektu (już stworzonego) odpowiednią funkcję składową)

Czy na prawdę nie jest możliwe zainicjowanie obiektu już w momencie jego deklaracji, tak

Jak można to zrobić z dowolnym typem wbudowanym?

```
int i= 0;
double d=  0.0;
```

Oczywiście, można! Trzeba jednak w tym celu stworzyć konstruktor w klasie.

Wprowadzenie do konstruktorów - II (przykład)

```
class cmplx {
        float rez, imz;
        public:
        cmplx(float a, float b) {rez = a; imz = b;}      //konstruktor
        void print() { cout<<endl<<"Liczba zespolona: "<<rez<<"+i"<<imz<<endl; }
};

int main()
{
        float re, im;

        cout<<endl<<"Podaj czesc rzeczywista liczby zespolonej"<<endl;
        cin>>re;
        cout<<endl<<"Podaj czesc urojona liczby zespolonej"<<endl;
        cin>>im;

        cmplx z1(re, im);
        cout<<endl<<"Oto liczba zespolona: "<<endl;
        z1.print();

    return 0;
}
```

Wprowadzenie do konstruktorów - III

Konstruktor to funkcja składowa klasy, wywoływana zawsze w momencie kreacji obiektu.

Jeśli konstruktor nie zostanie zdefiniowany, to kompilator sam go doda.

Konstruktor jest funkcją, ale nie może zwracać żadnej wartości (nawet void).

Konstruktor nazywa się dokładnie tak samo jak klasa.

Konstruktor może mieć dowolną liczbę argumentów (najczęściej ma tyle, ile składowych klasy chcemy zainicjować)- jego celem jest nadanie składowym obiektu wartości początkowych.

W trakcie definiowania obiektu zostaje mu przydzielone miejsce w pamięci, następnie uruchamiany jest dla niego konstruktor. Konstruktor sam nie przydziela miejsca w pamięci, konstruktor może ją tylko zainicjować.

Konstruktor może tworzyć obiekty typu const oraz volatile, ale sam nie może być funkcją tego typu.

Typy konstruktorów

Konstruktor można przeładować (praktyka bardzo często stosowana)..

```
class cmplx {
        float rez, imz;
        public:
        cmplx() {rez = 0.0; imz= 0.0;} //konstruktor domyślny
        cmplx(float a, float b) {rez = a; imz = b;} //konstruktor glowny
        cmplx(const cmplx&); {...} //konstruktor kopiujący
        ...
        void print() { cout<<endl<<"Liczba zespolona: "<<rez<<"+i"<<imz<<endl; }
};
int main()
{
        cmplx z1; //stworzenie obiektu przy pomocy konstruktora domyslnego
        cmplx z2(1,1) //stworzenie obiektu przy pomocy konstruktora glownego
        cmplx z3(z2); //stworzenie obiektu przy pomocy konstruktora kopiujacego
        cout<<endl<<"Oto pierwsza liczba zespolona: "<<endl;
        z1.print();
        cout<<endl<<"Oto druga liczba zespolona: "<<endl;
        z2.print();
        z3.print();
        return 0;
}
```

Wprowadzenie do destruktorów – przykład

Celem destruktora NIE jest usuwanie obiektu!

Destruktor to składowa funkcja klasy (podobnie jak konstruktor), wywoływana automatycznie zawsze wtedy, kiedy obiekt jest usuwany.

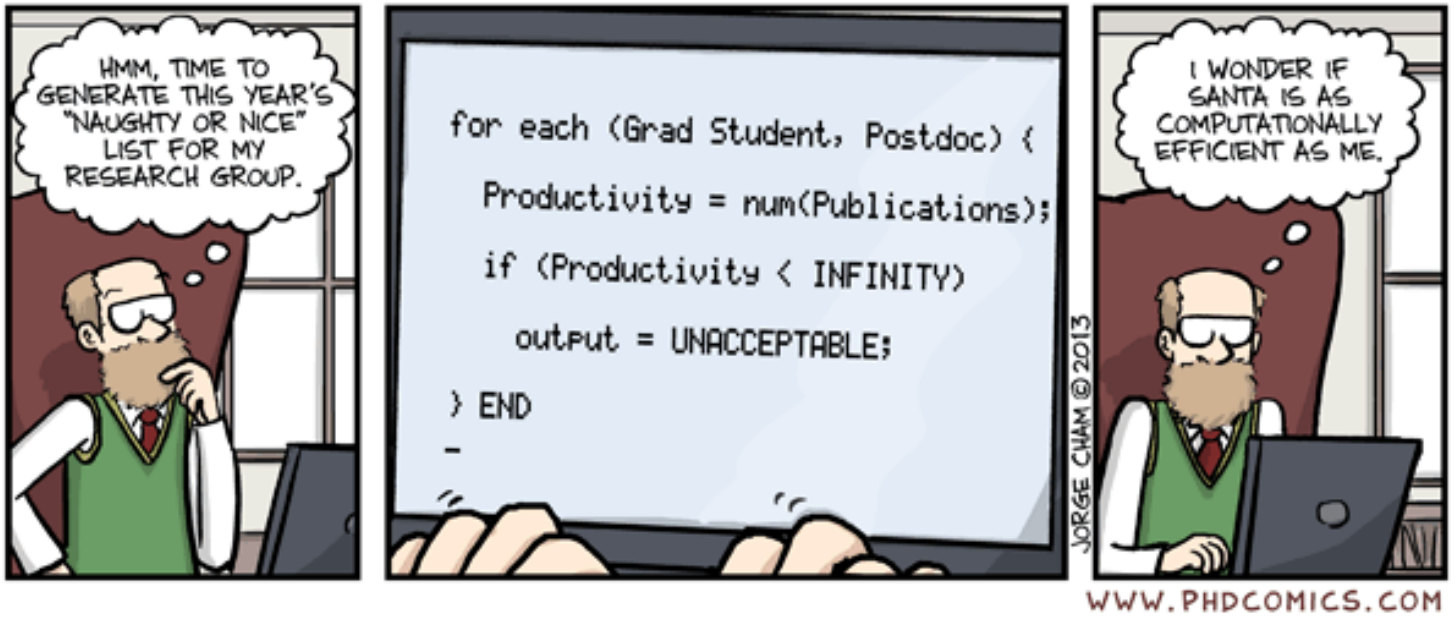
Destruktor nazywa się tak samo jak klasa, poprzedzony jest znakiem ~ (wężyk)

Destruktor nie ma określonego typu, jaki zwraca.

Destruktor nie ma żadnych argumentów wywołania (nie może być przeładowany).

```
#include <iostream>
using namespace std;
class cmplx {
        float rez, imz;
        public:
        cmplx(); //konstruktor domyślny
        cmplx(float, float ); //konstruktor główny
        ~cmplx();  //destruktor
        void print();
};
```

```
cmplx::cmplx() {
        cout<<"Konstruktor domyslny dziala.."<<endl;
        rez = 0.0; imz= 0.0;
}

cmplx::cmplx(float a, float b) {
        cout<<"Konstruktor glowny dziala.."<<endl;
        rez = a; imz = b;
}

cmplx::~cmplx() {
        cout<<"Destruktor dziala.."<<endl;
}

void cmplx::print() {
        cout<<endl<<"Liczba zespolona: "
                <<rez<<"+i"<<imz<<endl;
}
```



POLA I METODY STATYCZNE

Pola statyczne - wprowadznie

Składniki statyczny - pole w klasie, tworzone raz dla wszystkich obiektów w klasie, jednakowe dla wszystkich obiektów danej klasy,
istnieje nawet wtedy, kiedy żaden obiekt klasy nie został jeszcze stworzony.

Każdy obiekt danej klasy ma swój własny zestaw danych, każdy będzie miał zarezerwowane w pamięci odpowiednie miejsce na jego składowe. Przy wielu obiektach danej klasy, ilość miejsca w pamięci potrzebna do przechowania tych składowych zależy od ilości stworzonych obiektów.

Jeśli dana klasa ma składową typu double, przy wielu (np. stu) obiektach tej samej klasy, w pamięci będzie sto odrębnych miejsc przeznaczonych na dany składnik. Jest wiele przypadków kiedy wszystkie obiekty danej klasy powinny mieć tę samą danę (np. cena pralki: wszystkie egzemplarze danego modelu pralki są w tej samej cenie; ilość wszystkich obiektów danej klasy). W tym celu używa się składników statycznych.

Dana deklaracja staje się statyczna, kiedy przed jej deklaracją umieścimy słowo `static`.

Pola statyczne – przykład (plik nagłówkowy)

```
class Point { //plik nagłówkowy
        float x, y;
        public:
        static int counter; //deklaracja pola statycznego
        Point(); //konstruktor domyślny
        Point(float, float ); //konstruktor główny
        ~Point();    //destruktor
        void print();
};
```

Pola statyczne – przykład (plik implementacyjny)

```
int Point::counter =0; //definicja pola statycznego
Point::Point() {
        cout<<"Konstruktor domyślny dziala.."<<endl;
        x = 0.0; y= 0.0;
        counter++;
        cout<<"Wszystkich obiektow klasy Point jest "<<counter<<endl;
}
Point::Point(float a, float b) {
        cout<<"Konstruktor glowny dziala.."<<endl;
        x = a; y = b;
        counter++;
        cout<<"Wszystkich obiektow klasy Point jest "<<counter<<endl;
}
Point::~Point() {
        cout<<"Destruktor dziala.."<<endl;
        counter--;
        cout<<"Wszystkich obiektow klasy Point jest "<<counter<<endl;
}
void Point::print() {
        cout<<"Punkt: ("<<x<<", "<<y<<")"<<endl;
}
```

Pola statyczne – przykład (funkcja główna)

```
class Point { //plik nagłówkowy
        float x, y;
        public:
        static int counter; //deklaracja pola statycznego
        Point(); //konstruktor domyślny
        Point(float, float ); //konstruktor główny
        ~Point();    //destruktor
        void print();
};

int Point::counter = 0; //plik implementacyjny
```

```
int main()
{
        cout<<"Przed stworzeniem pierwszego punktu.."<<endl;
        cout<<"Mamy "<<Point::counter<<" obiektow"<<endl;

        Point p1;
        cout<<"Oto pierwszy punkt: "<<endl;
        p1.print();
        cout<<"Mamy "<<p1.counter<<" punkt/ -ow / -y"<<endl;

        Point p2(1.,1.);
        cout<<"Oto drugi punkt: "<<endl;
        p2.print();
        cout<<"Mamy "<<p2.counter<<" punkt/ -ow / -y"<<endl;

    return 0;
}
```

Pola statyczne – kilka wniosków

Składnik statystyczny (pole, metoda) jest elementem wspólnym (i identycznym!) dla wszystkich obiektów danej klasy.

Deklaracja składnika statycznego w ciele klasy nie jest jego definicją, definicję umieszczamy tam, gdzie definiowalibyśmy zmienną globalną.

Składnik statyczny może być prywatny, jego inicjalizacja jest możliwa nawet wtedy, ale dostęp do niego spoza klasy jest niemożliwy.

Składnik statyczny istnieje nawet, jeśli nie występują żadne obiekty danej klasy.

Sposoby odniesienia do składnika statycznego:

1) za pomocą nazwy klasy oraz operatora zakresu:: `klasa::skladnik_statyczny`

2) jeśli istnieją jakiekolwiek obiekty danej klasy, można posługiwać się operatorem .

```
obiekt.skladnik_statyczny
```

3) jeśli zdefiniowany jest wskaźnik (*wsk) do obiektów klasy, to stosujemy operator ->

```
wsk->skladnik_statyczny
```

Pola i metody statyczne - przykład - I

```
class komputer { //plik naglowkowy
        string procesor;
        int pamiec; // w GB
        int pojemnosc_dysku; //w GB
        bool dvd;
        float cena; //w PLN
        static string system_operacyjny;    //pole statyczne
        public:
        komputer(); //konstruktor domyslny
        komputer(string, int, int, bool, float); //konstruktor glowny
        ~komputer(); //destruktor
        void opis();
        static string jaki_system(); //metoda statyczna
        static int ilosc; //pole statyczne
};
```

Pola i metody statyczne - przykład - II

```
string komputer::system_operacyjny = "Linux";
int komputer::ilosc = 0; //plik implementacyjny

komputer::komputer() {
        procesor = "";
        pamiec = 0;
        pojemnosc_dysku = 0;
        dvd = false;
        cena = 0.0;
        ilosc++;
}
komputer::komputer(string proc, int pam,
        int dysk, bool ddvvdd, float koszt) {
        procesor = proc;
        pamiec = pam;
        pojemnosc_dysku = dysk;
        dvd = ddvvdd;
        cena = koszt;
        ilosc++;
}
```

```
komputer::~komputer() {
        ilosc--;
}

void komputer::opis() {
        cout<<"Oto komputer o parametrach: "<<endl;
        cout<<"Procesor: "<<procesor<<endl;
        cout<<"Ilosc pamieci (GB): "<<pamiec<<endl;
        cout<<"Pojemnosc dysku (GB): "<<
                pojemnosc_dysku<<endl;
        cout<<"Naped dvd: "<<dvd<<endl;
        cout<<"Cena: "<<cena<<" PLN"<<endl;
        cout<<"Obecnie mamy na stanie "<<ilosc<<
                " komputery / -ow"<<endl;
}

string komputer::jaki_system() {
        return system_operacyjny;
}
```

Zalety składników statycznych

Komunikacja obiektów między sobą – np. poprzez flagę.

Obiekty mogą wzajemnie zawiadamiać się o danym fakcie

Identyczna cecha wszystkich obiektów danej klasy (mogąca się zmieniać!)

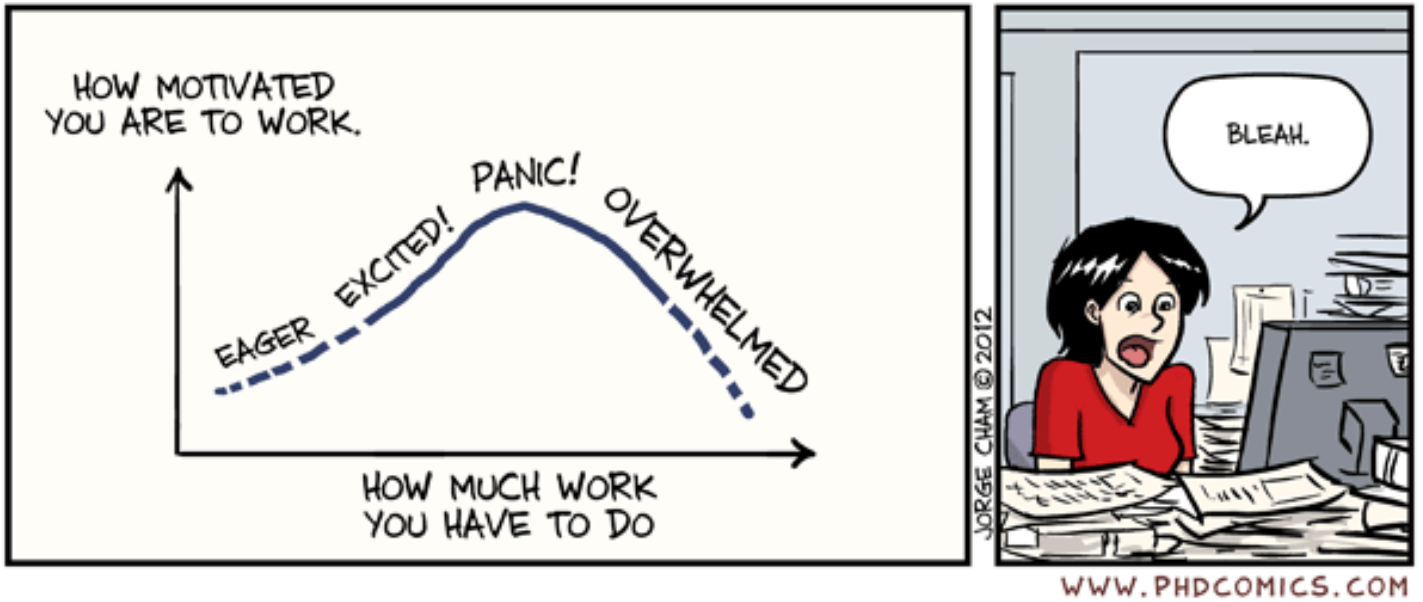
Licznik obiektów danej klasy

Korzystanie z jednego pliku dyskowego – przydzielonego danej klasie obiektów.

Składnikiem statycznym jest strumień (kanał transmisji do pliku)

Składniki statyczne oszczędzają miejsce, zamiast wielu jednakowych pól dla wszystkich obiektów danej klasy tworzone jest tylko jedno

… i wiele innych..



FUNKCJE SKŁADOWE TYPU CONST

Funkcje składowe typu const

Funkcja składowa typu const - funkcja, która wywołana na rzecz danego obiektu nie będzie modyfikować jego danych składowych.

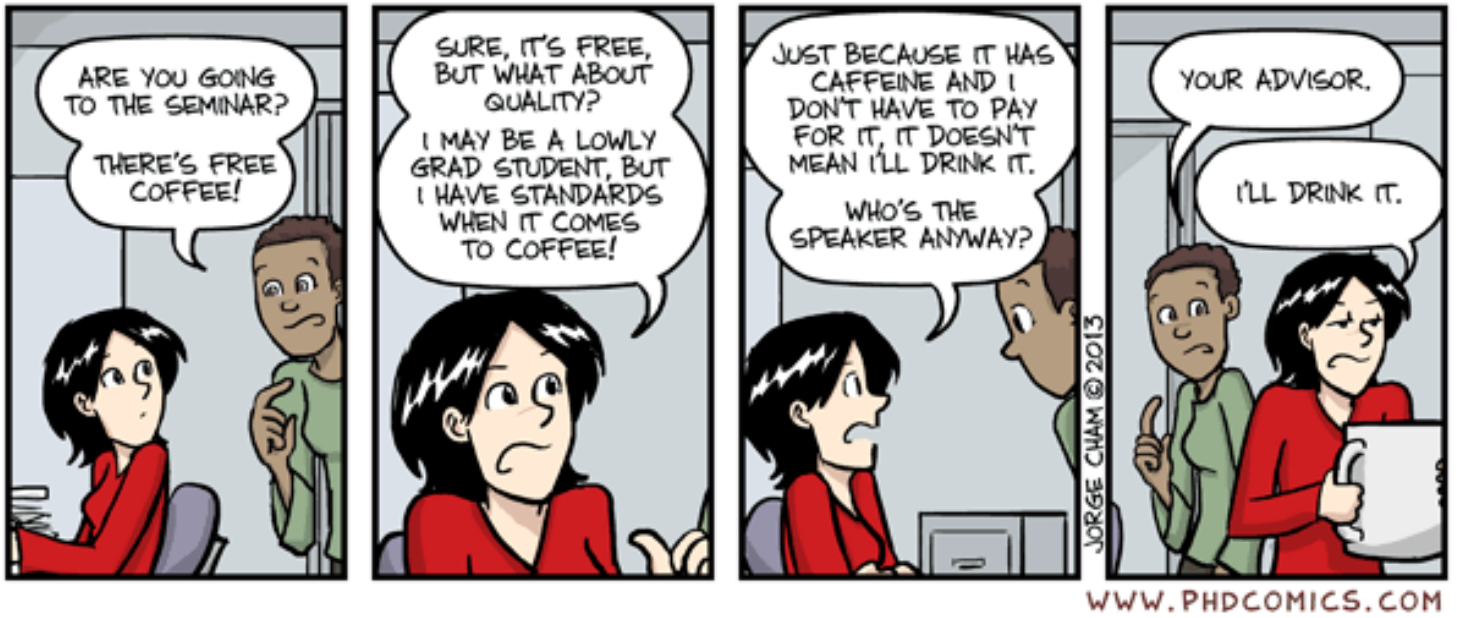
Obiekt typu const dla typu float:

```
const float pi = 3.1416;
```

Możliwe jest też tworzenie obiektów jakiejś klasy, które też będą typu const (ich dane składowe są raz ustalone i nie zmieniane)

Jeśli obiekt jest typu const, to na jego rzecz można wywoływać tylko te funkcje, które także są const, czyli "gwarantują" jego nietykalność.

Dlaczego tak musi być? Dlatego, że na etapie kompilacji potrzebne są jedynie deklaracje funkcji składowych (niekoniecznie definicje), dlatego nie zawsze wiadomo, które funkcje chcą modyfikować zawartość pól składowych, a które nie, temu ma służyć deklaracja const – - deklarująca nietykalność pól składowych.



REZERWACJA OBSZARÓW PAMIĘCI

Rezerwacja obszarów pamięci – operatory new, delete

Operator new zajmuje się kreacją obiektów, operator delete ich unicestwieniem.

(1)

```
int *wsk; //definicja wskaznika *wsk
wsk = new int; //utworzenie nowego obiektu (bez nazwy),
                        //ktorego adres jest przekazany *wsk
...
delete wsk; //likwidacja obiektu
```

(2)

```
char *name; //definicja wskaźnika *name
name = new char[100]; //utworzenie 100-elementowej tablicy znakowej
...
delete [] name; //skasowanie tablicy
```

Operatory new oraz delete kontra malloc oraz free

(1)

Przy tworzeniu obiektów operatorem new (zwłaszcza obiektów klas przez nas zdefiniowanych) automatycznie rusza do pracy konstruktor

Przy kasowaniu obiektów operatorem delete, automatycznie wykonuje się funkcja destruktor.

Przy użyciu malloc() oraz free() tej funkcjonalności nie będzie.

(2)

Wskutek użycia operatora new zostanie zwrócony wskaźnik takiego typu, jaki jest tworzony

Wskutek użycia malloc() zostaje zwrócony wskaźnik typu void *

Cechy obiektów utworzonych operatorem new

Cechy obiektów utworzonych operatorem new:

Czas życia takich obiektów liczony jest od momentu ich kreacji operatorem new, aż do ich skasowania operatorem delete. Użytkownik decyduje o czasie życia obiektu

Obiekt nie posiada nazwy, a operować na nim można jedynie za pomocą wskaźnika

Nie obowiązują zakresy ważności (gdzie takie obiekty są widziane, a gdzie nie), wystarczy, że choćby jeden wskaźnik pokazujący na dany obiekt był dostępny, to już mamy do niego dostęp

Statyczne obiekty inicjalizowane są zerami (lub innymi wartościami, jeśli tak określiliśmy), obiekty tworzone operatorem new już nie! Z racji tego, że są dynamiczne, zaraz po ich utworzeniu istnieją w nich jeszcze śmieci, to użytkownik ma zapisać w nich odpowiednią wartość

WYWOŁANIE KONSTRUKTORA, WYWOŁANIE DESTRUKTORA

Konstrukcja obiektów lokalnych i globalnych

Obiekty lokalne tworzone automatycznie (tworzenie na stosie w trakcie wykonywania programu) - powstają w momencie, gdy program napotka ich definicję, a przestają istnieć, kiedy program wychodzi poza blok, w którym zostały powołane do życia.

```
...
        { //otwarcie lokalnego bloku
                klasa obiekt;
        } //zamknięcie lokalnego bloku; obiekt 'obiekt' jest likwidowany
... tu nie ma już obiektu 'obiekt'
```

Konstruktor rusza do pracy w momencie kreacji obiektu.

Obiekty lokalne statyczne

Jeśli przy powyższej deklaracji / definicji stałoby słowo `static`, to mielibyśmy obiekt lokalny statyczny, tzn. istniałby od początku, aż do samego końca wykonywania się programu, ale byłby dostępny jedynie w zakresie ważności powyższego bloku. Konstruktor rusza do pracy przed rozpoczęciem wykonywania funkcji głównej `main`.

Obiekty tworzone globalnie, to takie, które tworzone są poza wszystkimi funkcjami. Zakresem Ważności takiego obiektu jest cały plik, a czas życia przez cały czas wykonywania programu. Konstruktor rusza do pracy jeszcze przed rozpoczęciem wykonywania funkcji `main`.

Konstrukcja obiektów globalnych - cont.

Obiekty tworzone operatorem new.

Powołanie obiektów do życia przy pomocy operatora new jest bardzo pożyteczne – umożliwia Tworzenie w trakcie wykonywania programu bardzo wielu obiektów.

```
int main() {
        cmplx *z1;
        z1= new cmplx(3.0,3.0);
        ...
}
```

Najpierw deklarujemy wskaźnik *z1 mogący pokazywać na obiekty klasy cmplx.

Za pomocą operatora new kreujemy obiekt klasy cmplx – dzieje się to w zapasie dostępnej pamięci, gdzie rezerwuje się pamięć, po czym rusza do pracy konstruktor inicjując pola klasy cmplx. Kiedy obiekt jest gotowy, wskaźnikowi *z1 przekazywany jest jego adres, od teraz możliwe jest posługiwanie się obiektem, mimo, że nie ma swojej nazwy, ale ma wskaźnik, który na niego pokazuje. Czas życia takiego obiektu jest liczony od jego kreacji operatorem new, aż po jego skasowanie operatorem delete.

UWAGA! Jeśli przez nieuwagę stracimy wskaźnik pokazujący na ten obiekt – tracimy możliwość zasięgu tego obiektu, mimo, że on nadal istnieje.

Jawne wywołanie konstruktora - II

```
class cmplx {
        float rez, imz;
        public:
        cmplx(float = 0.0, float = 0.0);
        ~cmplx() {};
        friend void print(cmplx &);
        ....
};
...
void print(cmplx &z) {
        cout<<z.rez<<"+ i"<<z.imz<<endl;
}
...
int main() {
                cout<<print(cmplx(2.0, 3.0));
        ...
}
```

Argumentem wywołania funkcji print jest wyrażenie będące jawnym wywołaniem konstruktora. Na użytek tego wyrażenia zostaje chwilowo powołany do życia nienazwany obiekt klasy cmplx, a następnie wysłany do funkcji print jako argument. Obiekt ten istnieje tylko w tej linii.

Jawne wywołanie konstruktora - III

```
(1) cmplx z1(2.0, 3.0);
(2) cmplx z1 = cmplx(2.0, 3.0);
```

W przypadku (1) tworzony jest obiekt klasy cmplx o nazwie z1.

W przypadku (2) wywołany jest konstruktor w sposób jawny, tworzy się nienazwany obiekt klasy cmplx. Po lewej stronie znaku = tworzony jest drugi obiekt klasy cmplx, o nazwie z1. Chwilowo istnieją 2 obiekty. Znak przypisania oznacza, że obiekt z1 ma dokładnie taką samą treść składników klasy, co obiekt bez nazwy.

Z funkcjonalnego punku widzenia obie kreacje obiektu z1 są identyczne.

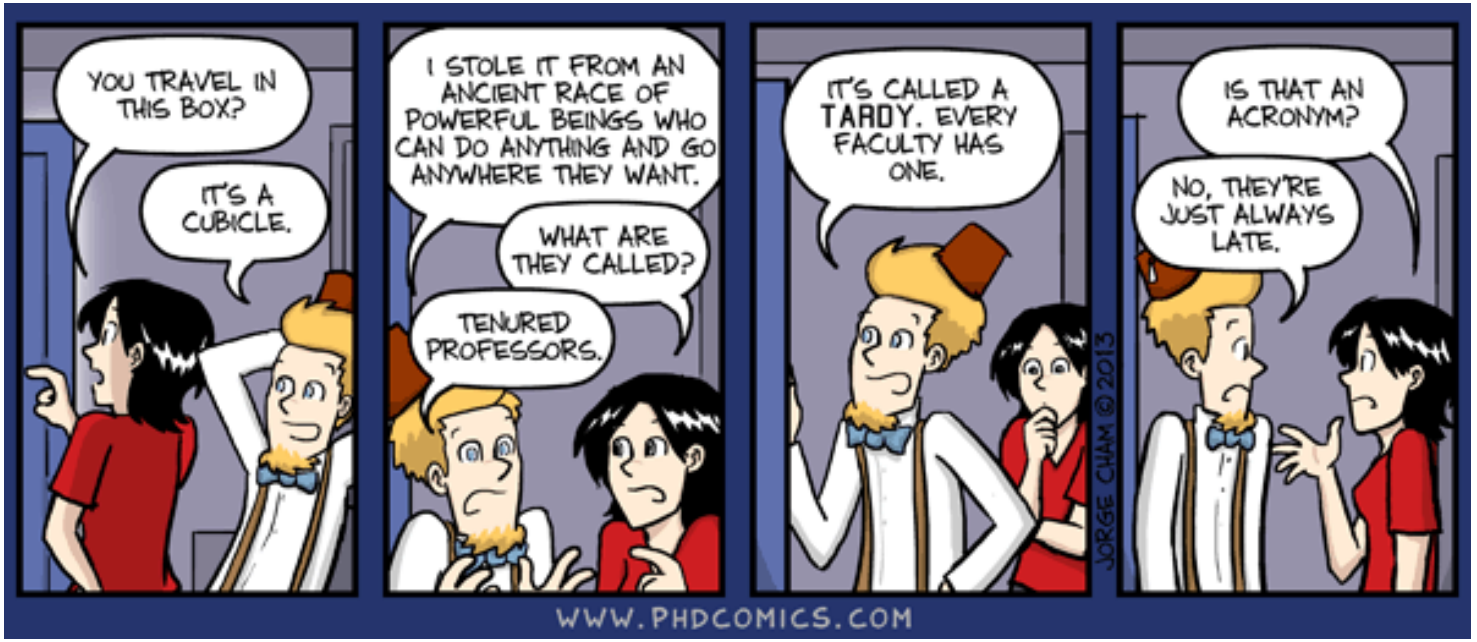
Jawne wywołanie destruktora

Destruktor może być wywołany jawnie poprzez podanie całej jego nazwy, razem z wężykiem ~
Jawne wywołanie destruktora może odbyć się jedynie na rzecz danego obiektu, wskaźnika lub referencji.

Nawet, kiedy wywołujemy jawnie destruktor w funkcji składowej klasy, jego nazwę wraz z wężykiem należy poprzedzić wskaźnikiem *this. this->~cmplx();

Jawne wywołanie obiektu nie likwiduje go! Obiekt istnieje dalej, została jedynie wywołana funkcja, jaka jest wywołana w momencie likwidacji obiektu (wskaźnika, referencji).

W przypadku, kiedy klasa nie ma destruktora (obecność destruktora w klasie nie jest obowiązkowa!) jego jawne wywołanie zostanie zignorowane.



KONSTRUKTOR DOMNIEMANY I KONSTRUKTOR GŁÓWNY

Konstruktor domniemany, konstruktor główny - I

Konstruktor domniemany (domyślny) to taki konstruktor, który można wywołać bez żadnego argumentu.

Konstruktor domyślny to taki, który nie ma argumentów lub ma je wszystkie domniemane.

Klasa może mieć tylko jeden taki konstruktor.

Jeśli klasa nie ma konstruktora, to kompilator sam wygeneruje sobie konstruktor domniemany, publiczny.

```
class computer {
        int memory;
        int hard_drive;
        string processor;
        bool dvd;
        public:
        computer(); //konstruktor domniemany bez argumentow
        computer(int, int, string, bool); //konstruktor główny
        ...
};
...
int main () {
        computer cc; //uzycie konstruktora domyslnego
        computer cc2(8, 500, “Intel”, 1); //uzycie konstruktora z argumentami (glownego)
}
```

Konstruktor domniemany, konstruktor główny - II

Konstruktor domyślny to też taki, który ma domyślne argumenty. Warto tworzyć jeden konstruktor: domyślny oraz główny jako jeden:

```
class computer {
        int memory;
        int hard_drive;
        string processor;
        bool dvd;
        public:
        computer(int = 0, int = 0, string ="" , bool= 0); //konstruktor domniemany oraz glowny
        ...
};
...
int main () {
        computer cc; //uzycie konstruktora domyslnego
        computer cc2(8, 500, "Intel", 1); //uzycie konstruktora z argumentami
}
```

Lista inicjalizacyjna konstruktora - I

Lista inicjalizacyjna konstruktora pozwala na nadanie wartości początkowych składnikom klasy

```
class prosta {
        double a, b;
        public:
        prosta(double aa, double bb) : a(aa), b(bb) {} //lista inicjalizacyjna konstruktora po :
};
```

Powyższy zapis jest tożsamy z:

```
class prosta {
        double a, b;
        public:
        prosta(double aa, double bb) {
                a= aa;  b= bb;
        }
};
```

lub rozdzielając na dwa pliki:

pr.h

```
class prosta {
        double a, b;
        public:
        prosta(double, double);
};
```

pr.cpp

```
prosta::prosta(double aa, double bb) {
                a=aa;
                b=bb;
}
```

Lista inicjalizacyjna konstruktora - II

UWAGA! Lista inicjalizacyjna umożliwia nadanie wartości początkowej także składnikowi typu const.

(1)

```
class kolo {
        double r;
        const double pi;
        public:
        kolo(double rr, double ppii) : r(rr), pi(ppii) {} //lista inicjalizacyjna konstruktora
};
```

też poprawnie:

(2)

```
class kolo {
        double r;
        const double pi;
        public:
        kolo(double rr, double ppii) : pi(ppii) {
                r ==rr;
        }
};
```

Konstruktor z listą inicjalizacyjną MOŻE zawierać swoje ciało (pomiędzy nawiasami { })

Lista inicjalizacyjna konstruktora - II

Powyższy zapis (1) NIE jest tożsamy z:

```
class kolo {
        double r;
        const double pi = 3.14; // ZLE! Deklaracji/definicji klasy NIE wolno inicjować składowych
        public:
        kolo(double rr) {
                r= rr;
        }
};
```

Składnikowi nie-const można w konstruktorze nadać wartość na dwa sposoby:

- przez listę inicjalizacyjną konstruktora
- przez podstawienie w ciele konstruktora

Składnikowi const można nadać wartość początkową jedynie za pomocą listy inicjalizacyjnej

Lista inicjalizacyjna NIE może zawierać składnika static

(to pole przecież istnieje od początku wykonywania się programu, nawet wtedy, gdy nie ma jeszcze żadnego obiektu danej klasy)

KONSTRUKTOR KOPIUJĄCY

Konstruktor kopiujący - I

Konstruktor kopiujący (inaczej: inicjalizator kopiujący) to taki konstruktor, którego można wywołać tylko z jednym argumentem: referencją innego obiektu tej samej klasy.

Konstruktor kopiujący służy do tworzenia obiektu danej klasy będącego kopią innego obiektu tej samej klasy.

```
class parabola {
        double a, b, c;
        public:
        parabola(double aa=0.0, double bb=0.0, double cc=0.0): a(aa), b(bb), c(cc) {};
        parabola(parabola &);     //konstruktor kopiujacy
        ~parabola() {};
};
parabola::parabola(parabola &p) {
        a = p.a;
        b = p.b;
        c = p.c;
}
...
int main() {
        parabola p1; //kreacja obiektu z uzyciem konstruktora domyslnego
        parabola p2(1.0,2.0,1.0); //kreacja obiektu z uzyciem konstruktora glownego
        parabola p3(p2); //kreacja obiektu z uzyciem konstruktora kopiujacego
        parabola p4 = p1; //kreacja obiektu z uzyciem konstruktora kopiujacego
        ...
}
```

Konstruktor kopiujący - II

Jeśli konstruktor kopiujący nie zostanie zdefiniowany, to kompilator sam wygeneruje sobie jego treść kopiując odpowiednie pola jednego obiektu do drugiego (składnik po składniku).

Konstruktor kopiujący może też zostać uruchomiony niejawnie! Dzieje się tak gdy:

(1) Przesyłamy obiekty do funkcji przez wartość (argumentem funkcji jest obiekt, który przesyłany jest przez wartość). W obrębie funkcji argument jest kopiowany, funkcja pracuje na kopii. W celu skopiowania obiektu do pracy rusza bez naszej wiedzy konstruktor kopiujący.

(2) Funkcja jako rezultat swojej pracy zwraca (przez return) obiekt przez wartość. Wtedy także rusza do pracy bez naszej wiedzy konstruktor kopiujący.

Konstruktor kopiujący pracuje na argumencie przesyłanym przez referencję, czyli teoretycznie oprócz stworzenia kopii obiektu mógłby go sobie zmodyfikować!
A co w przypadku, kiedy chcielibyśmy użyć konstruktora kopiującego na rzecz obiektu stałego (const)? Kompilator zaprotestuje, bowiem z definicji konstruktor kopiujący może, choć nie musi modyfikować przesłanego mu obiektu. W takim wypadku należy przesłać do konstruktora kopiującego obiekt stały!

Konstruktor kopiujący - III

```
class parabola {
        double a, b, c;
        public:
        parabola(double aa=0.0, double bb=0.0, double cc=0.0): a(aa), b(bb), c(cc) {};
        parabola(const parabola &);    //konstruktor kopiujacy
        ~parabola() {};
};
parabola::parabola(const parabola &p) {
        a = p.a;
        b = p.b;
        c = p.c;
}
...
int main() {
        parabola p1; //kreacja obiektu z uzyciem konstruktora domyslnego
        parabola p2(1.0,2.0,1.0); //kreacja obiektu z uzyciem konstruktora glownego
        parabola p3(p2); //kreacja obiektu z uzyciem konstruktora kopiujacego
        parabola p4 = p1; //kreacja obiektu z uzyciem konstruktora kopiujacego
        ...
}
```

Konstruktor kopiujący - IV

Niestety powyższe rozwiązanie nie zawsze jest możliwe do zastosowania, zwłaszcza wtedy kiedy klasa ma składniki, które są innymi klasami. Powyższy efekt jest możliwy do osiągnięcia, kiedy oba konstruktory kopiujące “zagwarantują nietykalność” (oba będą const). Niestety, zwykle bywa tak, że dopisanie kilku const'ów spowoduje reakcję lawinową i dojdziemy prędzej czy później do sytuacji, że nie będzie możliwa implementacja pewnej składowej jako const.

Do identycznych kopii wystarczy konstruktor kopiujący generowany automatycznie.

Są jednak sytuacje, kiedy dosłowna kopia obiektu nie jest pożądana..
Wtedy należy samemu zdefiniować konstruktor kopiujący.

Konstruktor kopiujący – kiedy jest niezbędny? - I

Wtedy, gdy tworzenie wiernej kopii, element po elemencie nie jest tym, co zamierzamy osiągnąć..

```
#include <iostream>
#include <cstring>

using namespace std;

class czlowiek{
        char *imie; //skladniki klasy to wskazniki, NIE tablice
        char *nazwisko; //kopiujac skladnik po skladniku robimy wierne kopie wskaznikow
        public:             //pokazujacych na to samo miejsce w pamieci
        czlowiek();
        czlowiek(char*, char*);
        ~czlowiek();
        void przedstaw();
        void zmiana(char*);
};

czlowiek::czlowiek() {
   imie = new char[100];
   nazwisko = new char[100];
}
```

Konstruktor kopiujący – kiedy jest niezbędny? - II

```
        imie = new char[100];
        nazwisko = new char[100];
        strcpy(imie, ii);
        strcpy(nazwisko, nn);
}

czlowiek::~czlowiek() {
        delete [] imie;
        delete [] nazwisko;
}

void czlowiek::przedstaw() {
        cout<<"Oto: "<<imie<<"\t"<<nazwisko<<endl;
}

void czlowiek::zmiana(char *nowe) {
        strcpy(nazwisko, nowe);
}
```

Konstruktor kopiujący – kiedy jest niezbędny? - III

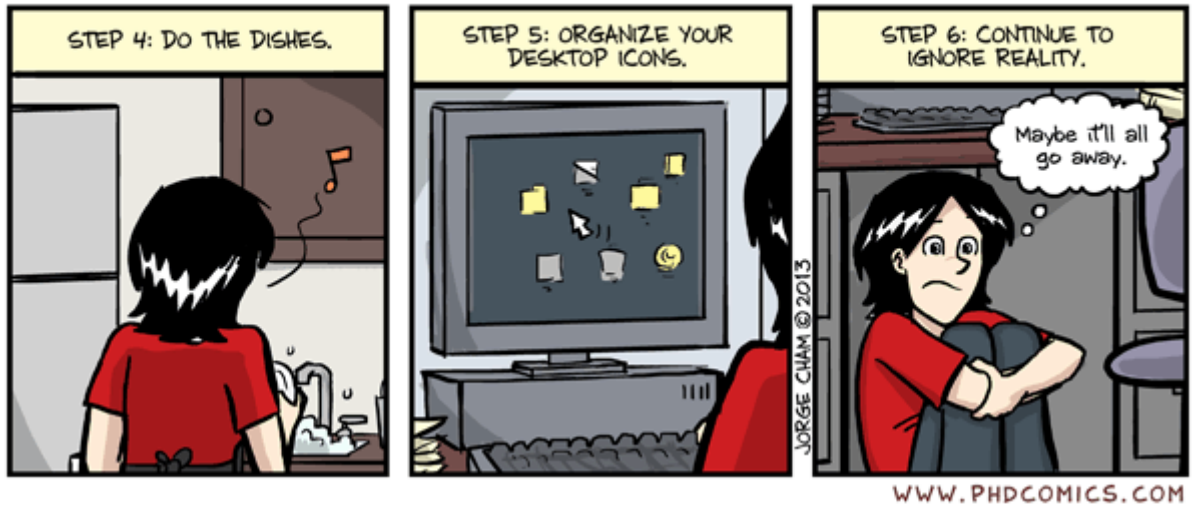
```
int main() {
        char i[100]= "Jan";
        char n[100]= "Kowalski";
        char r[100]= "Nowak";

        czlowiek c1(i, n);
        czlowiek c2(c1);

        c1.przedstaw(); // Jan Kowalski
        c2.przedstaw(); // Jan Kowalski

        c1.zmiana(r); // Kowalski → Nowak

        c1.przedstaw(); // Jan Nowak
        c2.przedstaw(); // Jan Nowak
        return 1;
}
```

Konstruktor kopiujący – kiedy jest niezbędny? - IV

A jeśli samodzielnie stworzymy konstruktor kopiujący?

```
class czlowiek{
        char *imie;
        char *nazwisko;
        public:
        czlowiek();
        czlowiek(char*, char*);
        czlowiek(const czlowiek &);
        ~czlowiek();
        void przedstaw();
        void zmiana(char*);
};...

czlowiek::czlowiek(const czlowiek &cz) {
        imie = new char[100];
        nazwisko = new char[100];
        strcpy(imie, cz.imie);
        strcpy(nazwisko, cz.nazwisko);
}
...
```

```
int main() {
        char i[100]= "Jan";
        char n[100]= "Kowalski";
        char r[100]= "Nowak";

        czlowiek c1(i, n);
        czlowiek c2(c1);

        c1.przedstaw(); // Jan Kowalski
        c2.przedstaw(); // Jan Kowalski

        c1.zmiana(r); // Kowalski → Nowak

        c1.przedstaw(); // Jan Nowak
        c2.przedstaw(); // Jan Kowalski
        return 1;
}
```



KONIEC WYKŁADU 3/4