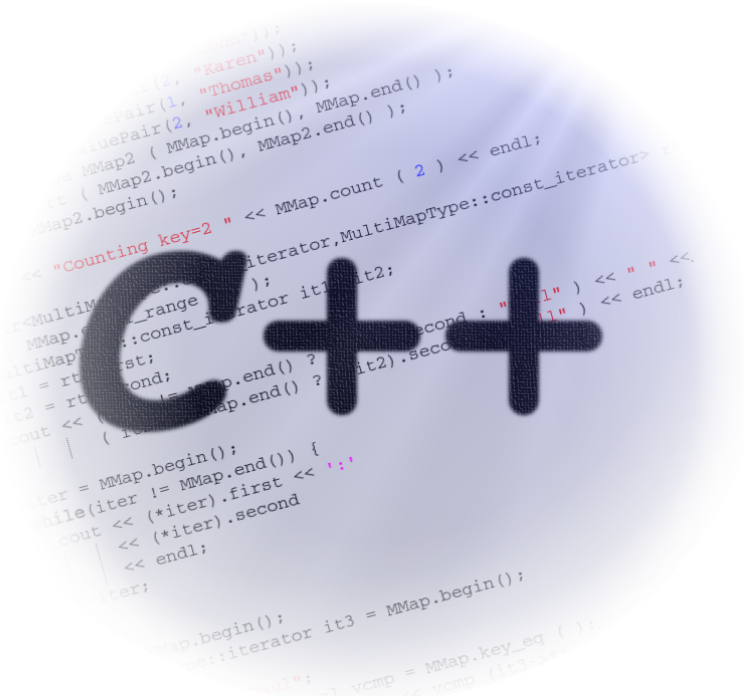
Języki Programowania

Prowadząca:
dr inż. Hanna Zbroszczyk

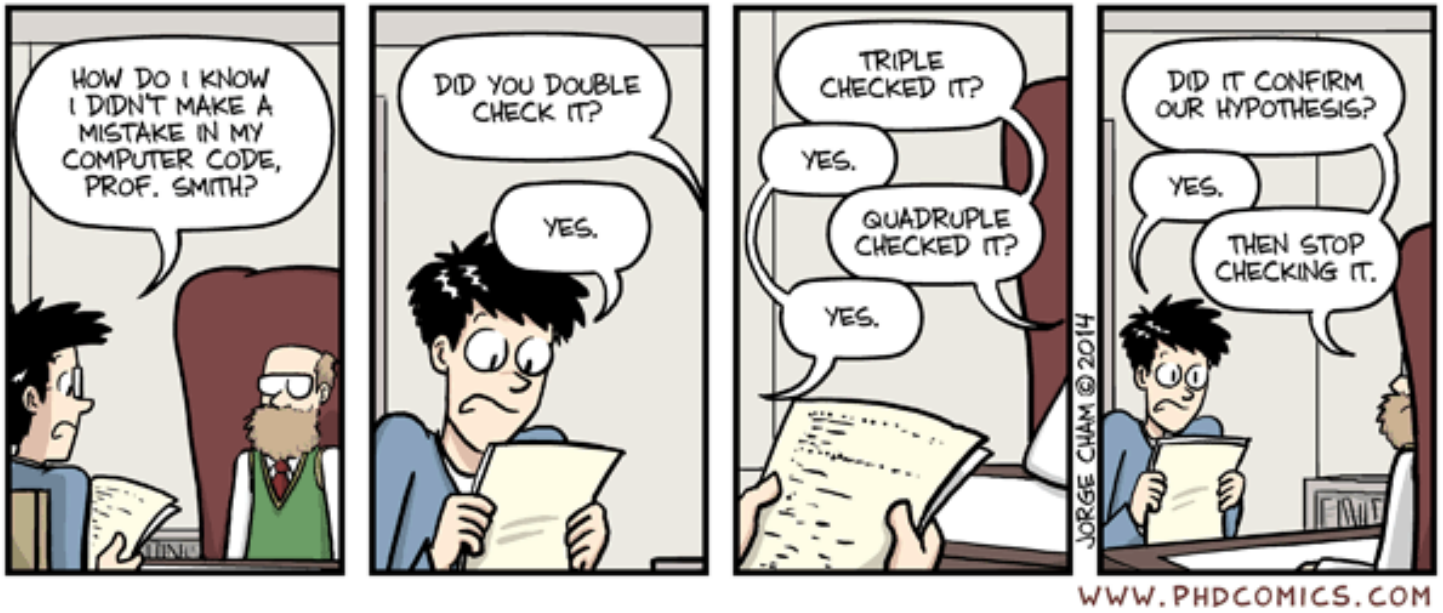
e-mail: *hanna.zbroszczyk@pw.edu.pl*
tel: +48 22 234 58 51

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)



TYP REFERENCYJNY

Typy w języku C++

Podział typów w języku C++ może być dwojaki:

- typy fundamentalne (identyczne, jak w klasycznym C): reprezentują:
  - liczby całkowite (*short int*, *int*, *long int*, *enum*)
  - znaki alfanumeryczne (*char*)
  - liczby zmiennoprzecinkowe (*float*, *double*, *long double*)
- typy pochodne (od typów podstawowych)

  Operatory umożliwiające tworzenie typów pochodnych:
  - [] - tablica obiektów danego typu
  - * - wskaźnik do pokazywania na obiekty danego typu
  - () - funkcja zwracająca wartość danego typu
  - & - referencja (przezwisko, odnośnik) obiektu danego typu

W deklaracjach typów pochodnych może pojawić się typ void (*void *p*, *void funkcja();*)

oraz:

- typy wbudowane
- typy zdefiniowane przez użytkownika

Typy referencyjne - I

Referencja to zmienna odnosząca się (identyfikująca) do innej zmiennej

Wykonanie operacji na referencji skutkuje takim samym wynikiem jak wykonanie jej na zmiennej, do której referencja ta odnosi się.

```
int zmienna = 3;
int &ref= zmienna;
cout<<zmienna<<ref<<endl;          //  3 3
ref--;
cout<<zmienna<<ref<<endl;          //  2 2


const int &ref2 = 12; // możliwe, przydzielony obszar pamięci musi być stałą
```

Użycie w funkcji:

```
void f(int &) {...}
void g(conts int &) {...}
...
g(1); // nie wolno wywołać f(1)
```

W momencie tworzenia referencji kompilator przydziela pewien obszar pamięci, który inicjuje oraz związuje nim referencję

Typy referencyjne - II

Czego nie wolno?

- tworzyć referencji do referencji: int && ref;
- tworzyć wskaźnika do referencji int &* wsk; (UWAGA! referencję do wskaźnika int *&ref; już można);
- tworzyć tablicy referencji: int & tab[10];
- tworzyć “pustej” referencji (referencja musi zostać zainicjowana podczas tworzenia, musi być związana z obszarem pamięci związanym już z innym obiektem)
- zmienić referencji, czyli wiązać jej z innym obiektem

Typy referencyjne - III

inny przykład:

```
#include <iostream>

using namespace std;

int main() {
   int a1, a2;
   a1= 1;
   a2= 3;
   int &ref= a1;

   cout<<a1<<a2<<ref<<endl;      // 1 3 1
   ref++;
   cout<<a1<<a2<<ref<<endl;      // 2 3 2
   ref=a2;
   cout<<a1<<a2<<ref<<endl;      // 3 3 3
   return 1;
}
```

Ponadto: w języku C++, w obrębie bloku deklaracje mogą przeplatać się z instrukcjami, (zmienne mogą być deklarowane "w locie") z tymże deklaracja musi poprzedzać użycie zmiennej.

Zamiana wartości dwóch liczb

```
#include <iostream>

using namespace std;

void zamiana(int x, int y) {
    int tmp= 0;

    tmp =x;
    x= y;
    y= tmp;
}

int main() {
    int a= 1, b=0;

    cout<<"Przed zamiana: a= "<<a<<", b= "<<b<<endl; //1 0
    zamiana(a,b);
    cout<<"Po zamianie: a= "<<a<<", b= "<<b<<endl;      //1 0

    return 1;
}
```

Parametr funkcji jest zmienną lokalną inicjowaną przy pomocy argumentu wywołującego funkcję

Zamiana wartości dwóch liczb przy użyciu wskaźników

```
#include <iostream>

using namespace std;

void zamiana(int *x, int *y) {
    int tmp;

    tmp= *x;
    *x= *y;
    *y= tmp;
}

int main() {
    int a= 1, b=0;

    cout<<"Przed zamiana: a= "<<a<<", b= "<<b<<endl;   //1 0
    zamiana(&a,&b);
    cout<<"Po zamianie: a= "<<a<<", b= "<<b<<endl;      //0 1

    return 1;
}
```

Jeśli parametr funkcji jest wskaźnikiem, to operacje wykonywane na tym parametrze są skojarzone z argumentem wywołania funkcji.

Zamiana wartości dwóch liczb przy użyciu referencji

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

void zamiana(int &x, int &y) {
    int tmp;
    tmp= x;
    x= y;
    y= tmp;
}

int main() {
    int a= 1, b=0;

    cout<<"Przed zamiana: a= "<<a<<", b= "<<b<<endl;      //1 0
    zamiana(a,b);
    cout<<"Po zamianie: a= "<<a<<", b= "<<b<<endl;        //0 1

    return 1;
}
```

Jeśli parametr funkcji jest referencją, to operacje wykonywane na tym parametrze są także skojarzone z argumentem wywołania funkcji.

Dodatkowo wynik zwracany przez funkcję może także być referencją, identyfikuje on wtedy zmienna zwracaną przez funkcję (return).

PRZEŁADOWANIE NAZW FUNKCJI

Przeładowanie nazw funkcji – wprowadzenie - I

W języku C jest niedopuszczalne, ponieważ nazwy funkcji muszą być unikatowe!

W języku C++ jest to możliwe..

Przeładowanie (ang. *overloading*) nazwy funkcji polega na tym, że w danym zakresie ważności jest więcej niż jedna funkcja o takiej samej nazwie. Ta, która z nich zostaje w danym wypadku wywołana zależy od typu, jak liczby jej argumentów.

Inaczej: Przeładowanie (zwane także przeciążeniem) nazwy funkcji polega na nadaniu jej wielu znaczeń, istnieje bowiem kilka funkcji o identycznej nazwie.

To, która jej "wersja" zostanie uruchomiona zależy od kontekstu, w jakim została użyta – - czyli od towarzyszących tej nazwie argumentów wywołania.

Rozważmy następujące funkcje:

```
(1) int dodaj(int, int);
(2) int dodaj(int, int, int);
(3) double dodaj(double, double);
(4) double dodaj(double, double, double);
```

Przeładowanie nazw funkcji – wprowadzenie - II

```
(1) int dodaj(int a, int b) {return a+b;}
(2) int dodaj(int a, int b, int c) {return a+b+c;}
(3) double dodaj(double a, double a){return a+b;}
(4) double dodaj(double a, double b, double c) return a+b+c;}
.....

int x, y, z;
double xx, yy, zz;

z= dodaj(x,y); //OK
z= dodaj(x,x,y); //OK
x= dodaj(y,15,z); //OK

xx= dodaj(yy,zz); //OK
zz= dodaj(xx,yy,yy); //OK
yy= dodaj(1.5, xx); //OK

z= dodaj(xx, y); //ŹLE
yy= dodaj(5, zz, 5); //ŹLE
```

Przeładowanie nazw funkcji - wprowadzenie - III

```
(1) int dodaj(int a, int b) {return a+b;} //OK
(2) int dodaj(int b, int a) {return b+a;} //ŹLE
(3) double dodaj(int a, int a){return (double)(a+b);} //ŹLE
```

Nie jest możliwe przeładowanie po raz drugi funkcji z taką samą liczbą, jak i typem argumentów.

Przeładowane funkcje różnią się jedynie liczbą i typem wywoływanych argumentów, a NIE typem wartości zwracanej. Z tego powodu funkcja (3) jest identyczna jak (1) oraz (2).

Problem jest wtedy, gdy kompilator nie wie, którą z przeładowanych funkcji ma uruchomić.

Przeładowanie nazw funkcji – przypadki szczególne - I

A) int, unsigned int.

```
(1) int dodaj(int a, int b) {return a+b;} //OK
(2) int dodaj(unsigned int a, unsigned int b) {return a+b;} //OK
```

Oba przeładowania są poprawne, bowiem int oraz unsigned int to różne typy!

B) Kolejność argumentów.

```
(1) void funkcja(int a, double b) {...} //OK
(2) void funkcja(double b, int a) {...} // OK
```

Oba przeładowania są poprawne, bowiem mimo, że obie funkcje mają taką samą liczbę argumentów Tych samych typów, to ich kolejność jest inna (kompilator nie będzie miał problemu z dopasowaniem właściwej funkcji).

Jeśli z jakiegoś powodu zamiany kolejności argumentów nie jest dobre, należy rozważyć dodanie nowego argumentu, tak aby ich lista stała się unikatowa.

Przeładowanie nazw funkcji – przypadki szczególne - II

C) Tablica, a wskaźnik.

```
(1) void funkcja(int tab[]) {...} //OK
(2) void funkcja(int *tab) {...} //ŹLE
.............
int ttt[10];
funkcja(ttt); // I CO TERAZ?
```

Zarówno int tab[], jak int *tab mogą mieć te same inicjalizatory, czyli te typy argumentów są uznane Przy przeładowaniu za identyczne.

D) Referencja.

```
(1) void funkcja(int a) {...} //OK
(2) void funkcja(int &a) {...} //ŹLE
.............
int tmp;
funkcja(tmp); // I CO TERAZ?
```

Referencja do danego typu, jak i ten sam typ są przez kompilator nierozróżnialne, dlatego takie przeciążenie jest błędne.

Przeładowanie nazw funkcji – przypadki szczególne - III

E) Przeładowanie, a typedef.

```
typedef int calkowite;
(1) void funkcja(int a, int b) {...} //OK
(2) void funkcja(calkowite a, calkowite b) {...} //ŹLE
```

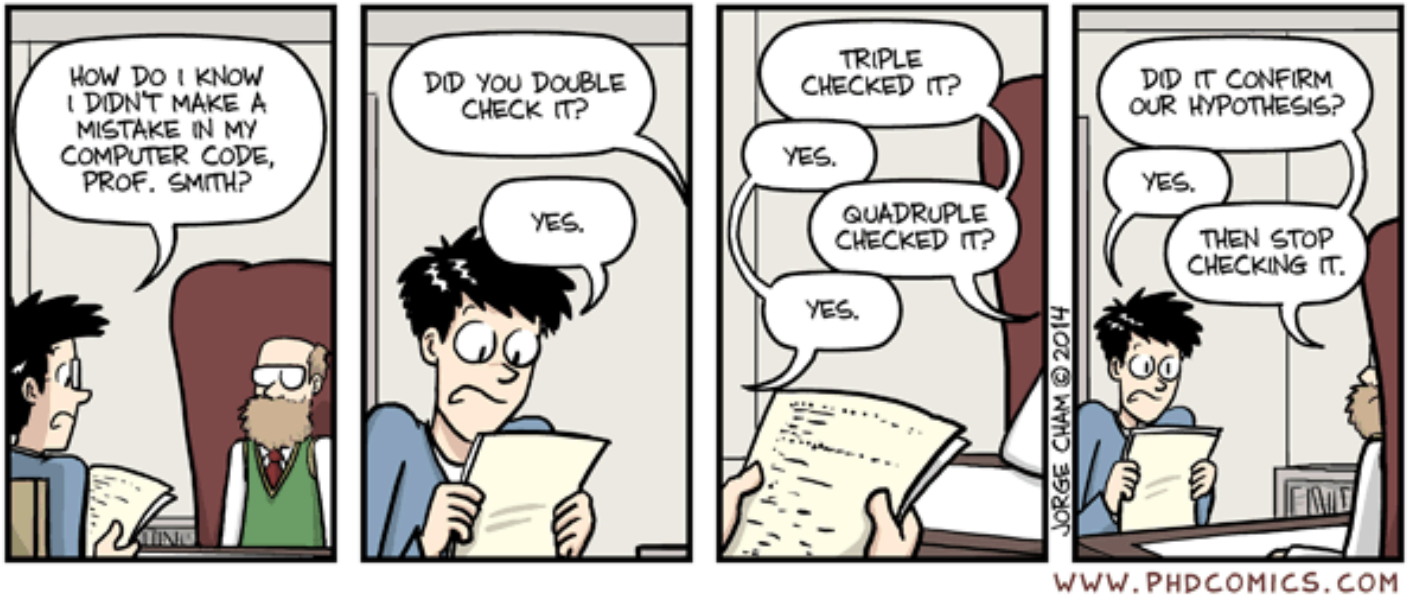
typedef to synonim dla istniejącego typu, NIE tworzy nowego. Powyższa próba przeładowania zostanie uznana za błąd.

F) Przeładowanie, a typ wyliczeniowy enum.

```
enum operacja {dodawanie=1, odejmowanie, mnozenie, dzielenie};
(1) void dzialanie(int);
(2) void działanie(operacja);
```

Typ wyliczeniowy enum - to odrębny typ, mimo, że bazuje na liczbach całkowitych (int).

Powyższa próba przeładowania jest poprawna.



DOMYŚLNE WARTOŚCI ARGUMENTÓW FUNKCJI

Domyślne wartości argumentów - I

W języku C++ możliwe jest nadanie wartości domyślnych argumentów funkcji:

```
int dodaj(int a, int b, int c=0) {return a+b+c;}
....
(1) int wynik = dodaj(10, 20, 0);
(2) int wynik = dodaj(10,20); // TO JEST TO SAMO, CO W LINIJCE WYŻEJ

Inny przykład:
int dodaj(int a, int b=0, int c=0) {return a+b+c;}
....
(1) int wynik = dodaj(10, 0, 0);
(2) int wynik = dodaj(10,0); // TO JEST TO SAMO, CO W LINIJCE WYŻEJ
(3) int wynik = dodaj(10); // TO JEST TO SAMO, CO W LINIJCE WYŻEJ
```

Domyślne wartości argumentów - II

I jeszcze inny przykład:

```
int dodaj(int a=0, int b=0, int c=0) {return a+b+c;}
....
(1) int wynik = dodaj(0, 0, 0);
(2) int wynik = dodaj(0,0); // TO JEST TO SAMO, CO W LINIJCE WYŻEJ
(3) int wynik = dodaj(0); // TO JEST TO SAMO, CO W LINIJCE WYŻEJ
(3) int wynik = dodaj(); // TO JEST TO SAMO, CO W LINIJCE WYŻEJ
```

Domyślne wartości argumentów (nie tylko 0) można nadawać licząc od OSTATNIEGO argumentu w funkcji.

```
int dodaj(int a=0, int b, int c) {return a+b+c;} //ŹLE
int dodaj(int a=0, int b=0, int c) {return a+b+c;} //ŹLE
int dodaj(int a=0, int b, int c=0) {return a+b+c;} //ŹLE
```

Dygresja do przeładowania funkcji:

```
int dodaj(int a, int b, int c=0) {return a+b+c;} //OK
int dodaj(int a, int b) {return a+b;} //ŹLE
...
int wynik = dodaj(2,5); // I CO TERAZ?
```

WPROWADZENIE DO KLAS

Struktura, a klasa - I

W języku C nie ma pojęcia klasy, ale mieliśmy struktury:

```
struct Prostokat {
        double a, b; //PUBLICZNE składowe struktury
        ......
};
```

W języku C++ możemy dalej posługiwać się strukturą, lub.. użyć klasy :

```
class Prostokat {
        double a, b; //PRYWATNE składowe klasy
        ......
};
```

To samo, co wyżej:

```
class Prostokat {
        private:
        double a, b; //PRYWATNE składowe klasy
        ......
};
```

Klasa to TYP obiektu, a NIE sam jej obiekt! (definicja klasy nie definiuje żadnych jej obiektów)

Struktura, a klasa - II

W celu zapewnienia jednakowego dostępu do elementów klasy:

```
struct Prostokat {
        double a, b; //PUBLICZNE składowe struktury
        ......
};
```

W języku C++ możemy dalej posługiwać się strukturą, lub.. użyć klasy:

```
class Prostokat {
        public:
        double a, b; //PUBLICZNE składowe klasy
        ......
};
```

Jeśli pewne składowe struktury nie mają swojego specyfikatora dostępu, to domyślnie są prywatnymi składnikami klasy.

Specyfikatory dostępności - I

Określenie dostępności, zwane specyfikatorami dostępu (public - publiczne, private – prywatne) mówią nam o tym, jaki mamy dostęp do tych składowych klasy.

Domyślnie, dla składowych struktury **struct**, w celu zapewnienia kompatybilności z językiem C, składowe struktury są publiczne.

Domyślnie, składowe klasy **class** są prywatne.

Do jawnego określenia dostępności składowych służą słowa kluczowe:
- **public** (publiczne)
- **private** (prywatne)
- **protected** (chronione).

Są to specyfikatory dostępu.

Specyfikatory dostępności - II

Specyfikatory dostępności składowych klasy:

- **public** – dostęp do tych składowych mają **wszystkie funkcje** mające dostęp do tej klasy (dostęp jest dla składowych tej samej klasy, klas pochodnych, innych klas, funkcji, funkcji zaprzyjaźnionych)

- **protected** - dostęp do tych składowych mają jedynie składowe tej samej klasy, składowe klas pochodnych oraz funkcje zaprzyjaźnione z daną klasą

- **private** – dostęp do tych składowych mają jedynie składowe tej samej klasy oraz funkcje zaprzyjaźnione z daną klasą

Klasa, a jej obiekt (wskaźnik oraz referencja do obiektu)

Do składników klasy możemy odnieść się na następujące sposoby:

```
class Prostokat {
        public:
        double a, b;
        ......
};
```

....

```
Prostokat z1; //deklaracja egzemplarza obiektu (obiektu)
Prostokat *z2; //deklaracja wskaźnika
Prostokat &z3= z1; //deklaracja i definicja referencji
z2 = &z1;
```

```
z1.a= 1;  //sposób odniesienia się do składnika obiektu
z2->a = 1; //sposób odniesienia się do składnika “pokazywanego” przez wskaźnik
z3.a = 1;  //sposób odniesienia się do składnika “pokazywanego” przez referencję
```

Definiowanie klas ciąg dalszy..

Definicja klasy nie może mieć zainicjalizowanych składników (przecież to tylko typ, a nie egzemplarz danej klasy, czyli obiekt..)

```
class Prostokat {
        public:
        double a; //OK
        double b= 0.0; //źLE
};
        ......
```

Definiowanie prostych klas z funkcjami składowymi - I

```
#include<iostream>

using namespace std;

class cmplx {
        float rez, imz; //składniki prywatne
        public:
                void read() {
                cout<<endl<<"Podaj czesc rzeczywista
                        liczby zespolonej"<<endl;
                cin>>rez;
                cout<<endl<<"Podaj czesc urojona
                        liczby zespolonej"<<endl;
                cin>>imz;

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

```
cmplx sum(cmplx z1, cmplx z2){ //źLE!
        cmplx z3;
        z3.rez = z1.rez+z2.rez;
        z3.imz= z1.imz+z2.imz;
        return z3;
}

int main()
        {
         cmplx z1, z2, z3;
         z1.read(); //OK
         z1.print(); //OK
         //podobnie wczytujemy z2

         cout<<"Oto suma liczb zespolonych: "<<endl;
         z3=sum(z1,z2); //ŹLE
```

Program działałby poprawnie, gdyby składowe rez, imz były także publiczne..

Definiowanie prostych klas z funkcjami składowymi -II

… ale problem można rozwiązać definiując publiczne metody zwracające wartości składowych prywatnych

UWAGA! Poprzez nadanie pewnym składowym klasy specyfikatora dostępu prywatnego gwarantujemy, że do składowych tych nie będzie dostępu dla żadnych składowych z innych klas oraz funkcji. Aby mieć możliwość "obejrzeć" zawartości tych składowych (bez możliwości ich modyfikacji) można zdefiniować właśnie publiczne metody (funkcje w danej klasie), które zwracają wartości tych składowych oraz metody pozwalające inicjować zmienne prywatne:

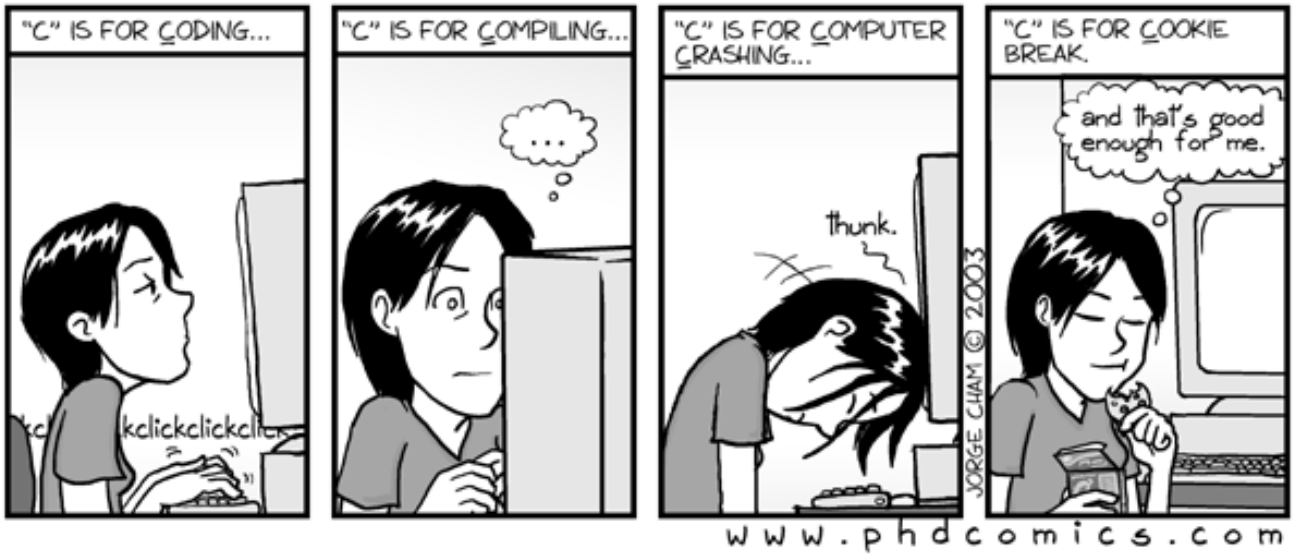
```
class cmplx {
        float rez, imz;
        public:
        void read() { cin>>rez>>imz; }
        void print() { cout<<endl<<"Liczba zespolona: "
                <<rez<<"+i"<<imz<<endl; }
        float getRez() {return rez;}
        float getImz() {return imz;}
        void setRe(float aa) {rez= aa;}
        void setImz(float bb) {imz= bb;}
};

...
```

```
cmplx sum(cmplx z1, cmplx z2){
        cmplx z3;
        z3.setRez(z1.getRez()+z2.getRez());
        z3.setImz(z1.getImz()+z2.getImz());
        return z3;
}
...
int main() {
        cmplx z1, z2, z3;
        z1.read(); //OK
        z1.print(); //OK
        //podobnie wczytujemy z2
        z3=sum(z1,z2); //OK
}
```

Definiowanie prostych klas z funkcjami składowymi -III

… możliwe jest też zaprzyjaźnienie funkcji dodającej z klasą:

```
class cmplx {
        float rez, imz;
        public:
        void read() { cin>>rez>>imz; }
        void print() { cout<<endl<<"Liczba zespolona: "<<rez<<"+i"<<imz<<endl; }
        friend cmplx sum(cmplx &, cmplx &);
};
cmplx sum(cmplx z1, cmplx z2){ //OK
        cmplx z3;
        z3.rez = z1.rez+z2.rez;
        z3.imz= z1.imz+z2.imz;
        return z3;
}
int main(){
          cmplx z1, z2, z3;
          z1.read(); //OK
          z1.print(); //OK
          //podobnie wczytujemy z2
          z3=sum(z1,z2); //OK
          return 1;
}
```



FUNKCJE ZAPRZYJAŹNIONE

Funkcje zaprzyjaźnione - wprowadzenie

Funkcja zaprzyjaźniona z klasą- funkcja, która (mimo, że nie jest składnikiem klasy) ma dostęp do wszystkich (nawet tych prywatnych – private oraz chronionych - protected) składników klasy.

Konstruując daną klasę ustalamy, że pewne jej składniki będą prywatne, tzn, że mogą na nich pracować jedynie funkcje składowe tej klasy – inne już nie! Kiedy jednak chcemy, aby funkcja zewnętrzna miała dostęp do składników prywatnych – należy ją zaprzyjaźnić z klasą.
W tym celu wewnątrz klasy należy umieścić deklarację przyjaźni klasy z tą funkcją poprzez zamieszczenie jej deklaracji poprzedzonej słowem friend. Dzięki temu funkcja ta ma dostęp do wszystkich prywatnych (oraz chronionych) składników klasy.

Uwaga! To nie funkcja ma twierdzić, że przyjaźni się z klasą, ale sama klasa daje jej dostęp do swoich prywatnych i chronionych składników. Słowo friend pojawia się jedynie wewnątrz definicji klasy.

Dobrym rozwiązaniem jest zamiast zaprzyjaźniania funkcji z klasą uczynić funkcję funkcją składową samej klasy, ale..
… funkcje zaprzyjaźnione mają też swoje zalety..

Zalety funkcji zaprzyjaźnionych

1) Funkcja zaprzyjaźniona może być przyjacielem więcej niż jednej klasy (może mieć dostęp do składników prywatnych i chronionych więcej niż jednej klasy).

2) Funkcja zaprzyjaźniona może na argumentach jej wywołania dokonywać konwersji zdefiniowanych przez użytkownika.

… do tego zagadnienia jeszcze wrócimy...

Na razie: funkcję składową można wywołać na rzecz danego obiektu , ale nie np. liczby 5.

Funkcja zaprzyjaźniona może za swój argument przyjąć nawet i liczbę 5, pod warunkiem, że określone jest w jaki sposób liczba 5 jest zamieniona na obiekt danej klasy.

3) Funkcja zaprzyjaźniona może mieć dostęp do składników prywatnych i chronionych takich klas, których funkcją składową nie mogłaby być z powodów zasadniczych,

tzn. na przykład dlatego, że zostały one napisane w innym języku programowania

Jak się przyjaźnić z więcej niż jedną klasą?

```
class klasa2; //deklaracja zapowiadająca
class klasa1 {
        //składniki prywatne..
        public:
        //składniki publiczne..
        friend void funkcja_zaprzyjazniona(klasa1 &ob1, klasa2 &ob2);
        //przed pierwszym uzyciem klasy – musi byc ona zadeklarowana
};
class klasa2 {
        //składniki prywatne..
        public:
        //składniki publiczne..
        friend void funkcja_zaprzyjazniona(klasa1 &ob1, klasa2 &ob2);
};

void funkcja_zaprzyjazniona(klasa1 &ob1, klasa2 &ob2) { ... }
```

Funkcji zaprzyjaźnionych używamy jak “normalnych” funkcji (nie jak składnika klasy).

To jest zwykła funkcja, nie będąca związana z klasą, a jedynie mająca dostęp do jej prywatnych i chronionych składników.

Funkcje zaprzyjaźnione ciąg dalszy..

I) Deklaracja przyjaźni funkcji z klasą może znaleźć się w dowolnym miejscu klasy,
nie ma znaczenia specyfikator dostępu (nie obowiązują jej słowa private i protected).

II) Zazwyczaj wewnątrz klasy znajduje się jedynie deklaracja przyjaźni klasą z daną funkcją

III) .. ale czasem wewnątrz klasy umieszczana jest nie tylko deklaracja funkcji zaprzyjaźnionej, ale wręcz jej definicja. Konsekwencje takiego stanu są następujące:

- funkcja jest typu inline;
- funkcja jest nadal tylko przyjacielem, a NIE składnikiem klasy;
- funkcja leży w zakresie leksykalnym deklaracji tej klasy, tzn. że w definicji funkcji zaprzyjaźnionej można korzystać z właśnie obowiązujących instrukcji typedef (wewnątrz deklaracji tej klasy) oraz korzystać ze zdefiniowanych w tej klasie typów wyliczeniowych enum.

IV) W przypadku funkcji przeładowanych przyjacielem danej klasy jest ta wersja funkcji, która odpowiada liście argumentów widocznej w deklaracji przyjaźni w definicji danej klasy.

V) Funkcja zaprzyjaźniona może być składową innej klasy.

VI) Funkcja jest zaprzyjaźniona z klasą, a nie tylko z obiektem danej klasy. Funkcja zaprzyjaźniona ma prawa dostępu do wszystkich obiektów dane klasy.

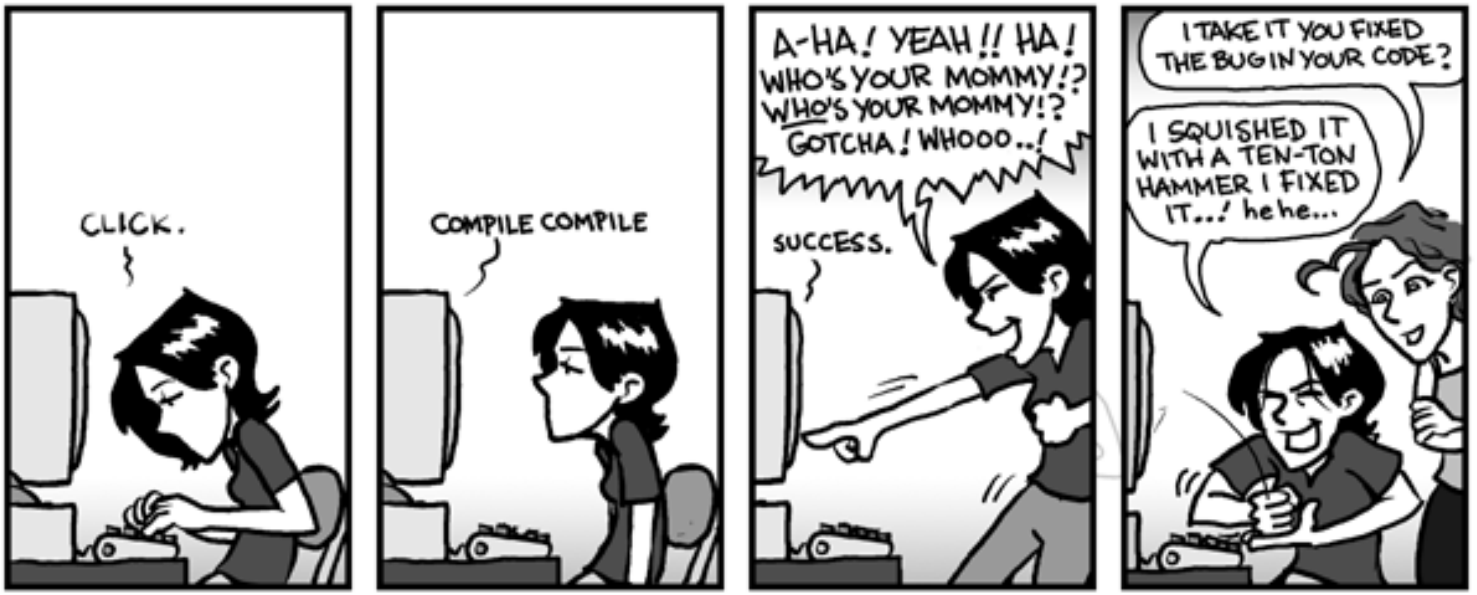
Klasy zaprzyjaźnione

Dana klasa może deklarować przyjaźń z więcej niż jedną, a nawet wszystkimi funkcjami składowymi innej klasy. W takim wypadku warto zadeklarować przyjaźń z inną klasą, zamiast kilku (i więcej!) deklaracji przyjaźni z funkcjami. Od tej pory wszystkie funkcje innej klasy mają dostęp do składników prywatnych I chronionych pierwszej klasy.

Przyjaźń nie jest obustronna! Jeśli druga klasa także chce przyjaźnić się z klasą pierwszą, to musi przyjaźń zadeklarować.

```
class druga; // deklaracja zapowiadajaca
class pierwsza {
        friend class druga;
        //reszta ciala klasy pierwszej
};
class druga {
        friend class pierwsza;
        //reszta ciala klasy drugiej
};
```

Przyjaźń nie jest przechodnia (przyjaciel mojego przyjaciela nie jest moim przyjacielem).
Przyjaźń nie jest dziedziczna.



WSKAŹNIK *this

Wskaźnik (*this)

Kiedy wywoływana jest na danym obiekcie funkcja składowa jego klasy, do funkcji przesyłany jest wskaźnik do danego obiektu (*this).

```
class Punkt {
	float x, y;
	public:
		void read() {
		cout<<endl<<"Podaj wspolrzedna x punktu"<<endl;
		cin>>(this->x); //tozsame z cin>>x;
		cout<<endl<<"Podaj wspolrzedna x punktu"<<endl;
		cin>>(this->y); //tozsame z cin>>y;

	}
	void print() {
		cout<<endl<<"Oto punkt: (" <<(this->x)<<", "<<(this->y)<<")" <<endl;
	}
};
```

- Możliwe jest używanie tego wskaźnika, ale nie jest to obligatoryjne.;
- Nie wolno "poruszać" wskaźnikiem this. (jest typu X const *);
- Wskaźnik this stojący przed składnikami klasy sprawia, że operacje są przeprowadzane na składnikach tego (this) konkretnego egzemplarza obiektu, dla którego dana funkcja została wywołana. Nie ma żadnej wieloznaczności.

ZASŁANIANIE NAZW

Zasłanianie nazw – najlepiej pokazać na programie.. - I

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

using namespace std;

float pole =100.0; //zmienna globalna

class kolo {
        public:
        float r; //promien
        float obwod() {return (2*r*PI);}
        float pole() {return (PI*r*r);} //funkcja skladowa klasy kolo
        void info();  //funkcja skladowa klasy kolo
};

class prostokat {
        public:
        float a, b; //boki
        float obwod() {return (2*a + 2*b);}
        float pole() {return (a*b);} //funkcja skladowa klasy prostokat
        void info(); //funkcja skladowa klasy prostokat
};
```

Zasłanianie nazw – najlepiej pokazać na programie.. - II

```
void kolo::info() {
        cout<<endl<<"Kolo o promieniu "<<r<<", o obwodzie "<<obwod()<<" oraz polu "<<
                pole()<<endl;

        //cout<<"Pierwsza proba uzycia zmiennej globalnej pole "<<pole<<endl; //ZLE
        cout<<"Pierwsza proba uzycia zmiennej globalnej pole "<<::pole<<endl; //OK

        int pole = 0; //zmienna lokalna 'pole'
        cout<<"Zmienna lokalna funkcji skladowej klasy kolo, o nazwie pole, takiej samej,
                jak nazwa innej funkcji skladowej klasy kolo "<<pole<<endl;

        //cout<<"Jeszcze raz wypiszemy pole kola"<<pole()<<endl; //ZLE
        cout<<"Jeszcze raz wypiszmy pole kola "<<kolo::pole()<<endl; //OK
}

void prostokat::info() {
        cout<<endl<<"Prostokat o bokach "<<a<<", "<<b<<", o obwodzie "<<obwod()<<
                " oraz polu "<<pole()<<endl;
}

void info() { cout<<endl<<"To jest program do liczenia obwodu i pola kola oraz
        prostokata "<<endl;} //funkcja globalna
```

Zasłanianie nazw – najlepiej pokazać na programie.. - III

```
int main() {
        info();
        cout<<"Zmienna globalna pole (nie wiadomo czego) "<<pole<<endl;

        char info[100]= "Zaprezentujmy dzialanie programu na kole";

        //info(); //nieudana proba wywolania funkcji globalnej - ZLE
    ::info(); //udana proba uzycia globalnej funkcji - OK
        cout<<endl<<info<<endl;

        kolo k1;
        k1.r = 3;
        k1.info();
        cout<<"... i jeszcze raz wypiszmy pole kola "<<k1.pole()<<endl;

        prostokat p1;
        p1.a = 1;
        p1.b = 2;
        p1.info();

        return 1;
}
```

Zasłanianie nazw – kilka wniosków

Zasłanianie nazw jest dozwolone, ale utrudnia zrozumienie funkcjonowania programu.

Odnoszenie się do zasłoniętych nazw odbywa się dzięki pomocy operatora zakresu ::

Nazwa zasłania inną nazwę, niezależnie od tego, czy jest to zmienna, czy funkcja.

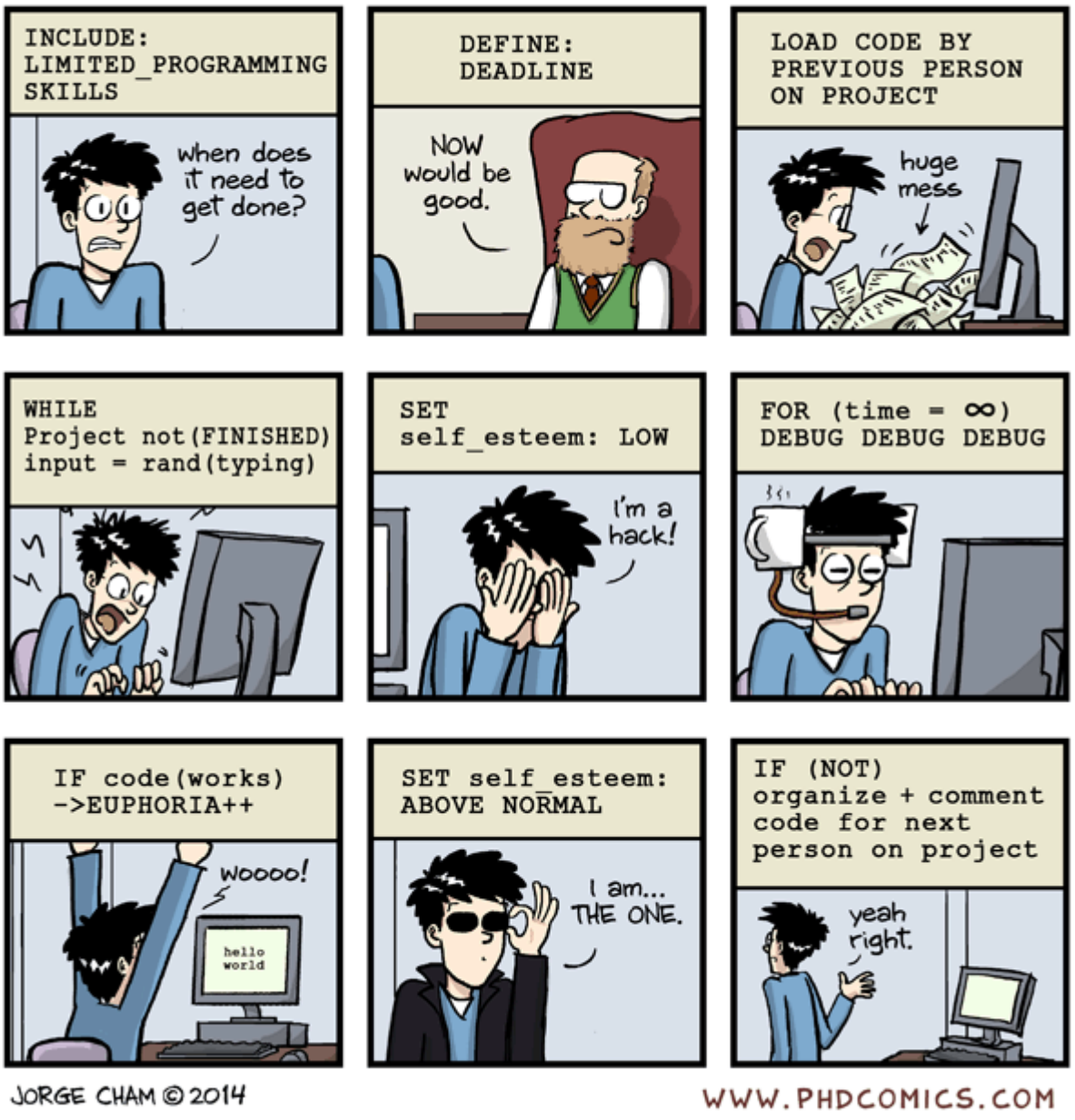
… aby zasłanianie nazw było jeszcze 'ciekawsze' –

- przeładowana nazwa funkcji może być też przeciążona!

Wystarczy w klasie kolo dodać funkcje o przeładowanej nazwie info(...), np:

```
void info(char*);
int info(char*, int);
```

.. I jeszcze kilka innych :-)



KONIEC WYKŁADU 2