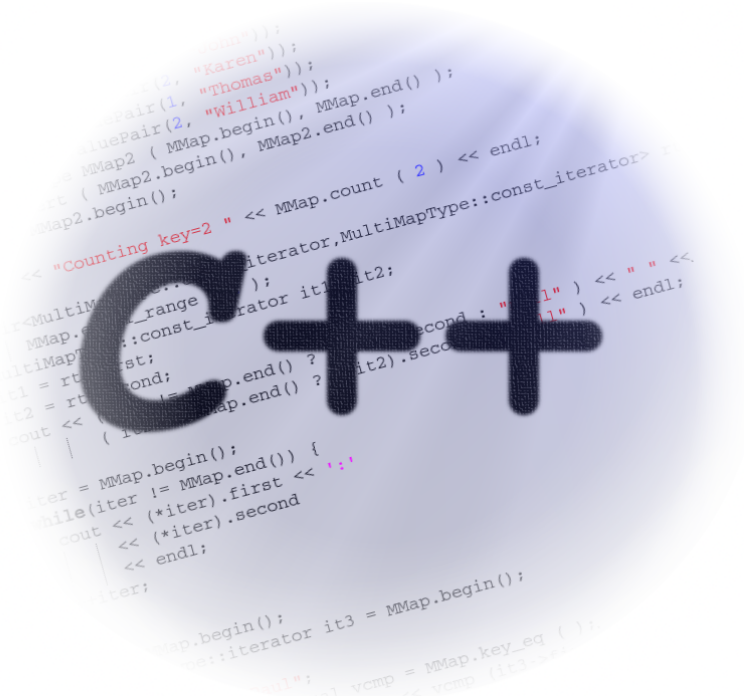
Języki Programowania

Prowadząca:
dr inż. Hanna Zbroszczyk

dr inż. Daniel Kikoła

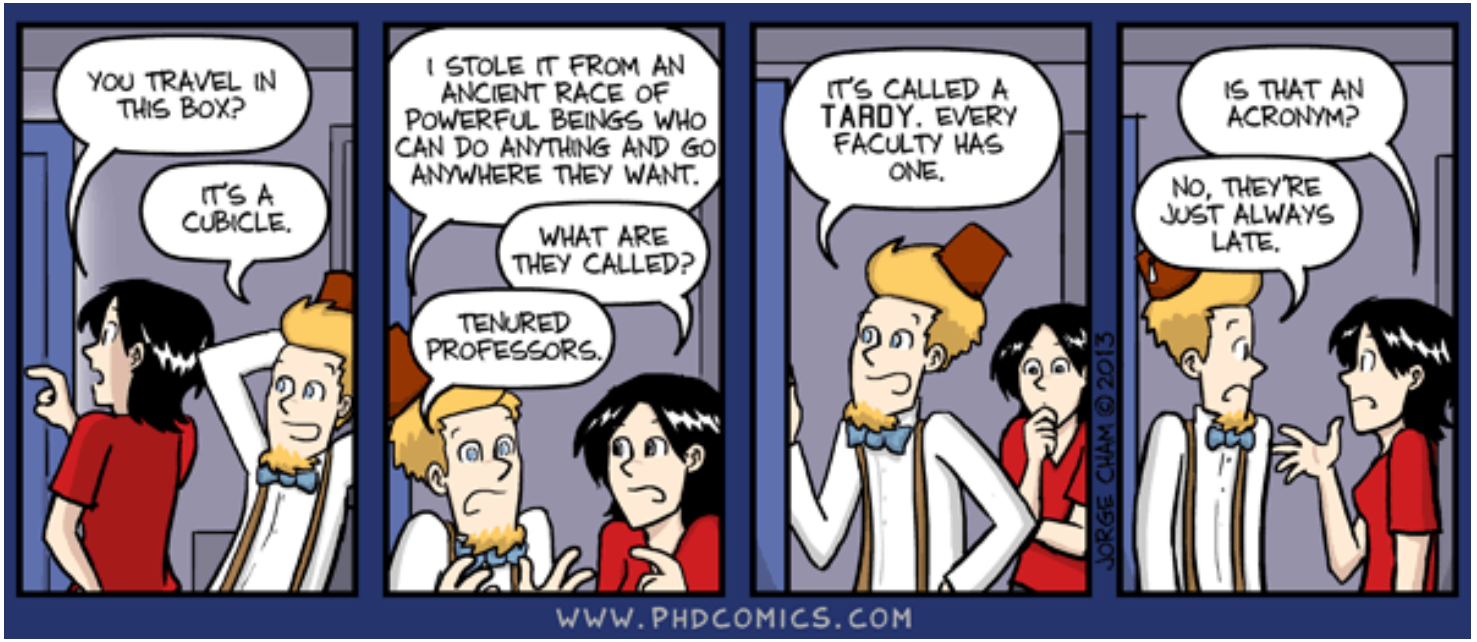
e-mail: *hanna.zbroszczyk@pw.edu.pl*
tel: +48 22 234 58 51

Konsultacje:
Piątek: 14.00 – 15.30
(nie w tym tygodniu)



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

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



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);
}
...
```

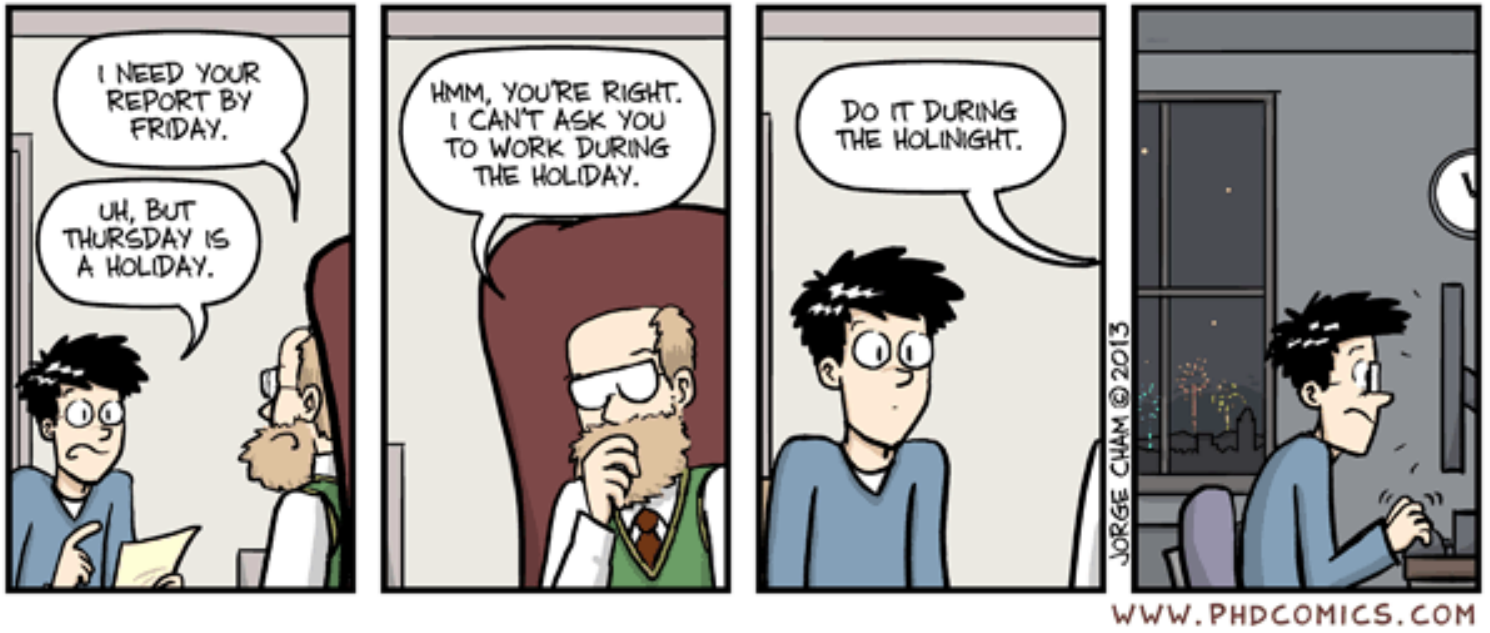
```
int main() {
        char i[100]= "Jan";
        char n[100]= "Kowalski";
        char r[100]= "Nowak";

        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;
}
```



PRZECIĄŻANIE OPERATORÓW

Przeciążanie operatorów – przypomnienie z wykładu 1

Przypomnijmy sobie przykład z wykładu 1 (lekko zmodyfikowany):

```
class cmplx {
        public:
            float rez, imz;
        ...
};


cmplx operator+(cmplx z1, cmplx z2){
        cmplx z3;
        z3.rez = z1.rez+z2.rez;
        z3.imz= z1.imz+z2.imz;
        return z3;
}
...

int main() {
         cmplx zz1, zz2, zz3;
         ...
         zz3 = zz1 + zz2;
         //zz3 = operator+(zz1,zz2); //to samo, co linia wyżej
         ...
}
```

Przeciążanie operatorów - I

Przeciążanie (przeładowanie) operatorów polega na zdefiniowaniu swojej własnej funkcji, która:

- nazywa się operator@ - gdzie @ oznacza symbol operatora, który przeładowujemy (np. +, -, …)
- jako co najmniej jeden z argumentów przyjmuje obiekt danej klasy (musi być obiekt, nie może być składnik do obiektu)

Definicja przeładowanego operatora:

```
typ_zwracany operator@ (argumenty) {
    // cialo funkcji
}
```

Przeładować można bardzo wiele operatorów:

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	<<==	>>==	==	!=
<=	>=	&&	||	++	-	,	->*	->
new	delete	()	[]					

Przeciążanie operatorów - II

Operatory: + -* & mogą być przeładowane w wersji jedno- lub dwuargumentowej

Nie można przeładować operatorów:

. - odniesienie do składnika klasy

.* - wybór składnika wskaźnikiem

:: - określenie zakresu

Przeładowanie operatora polega na nadaniu mu specyficznego znaczenia (zależnego tylko od naszej definicji tego operatora), którego nabiera wtedy, kiedy stoi on obok obiektu dane klasy.

Wyżej wymienione operatory mają już bardzo ważne znaczenia wobec obiektów klasy.

Uwagi do przeciążania operatorów - I

Przeładować można tylko wymienione operatory, nie można definiować własnych

Przeładowując operator nadajemy mu nowe znaczenie, nie ma też ograniczeń co do typu zwracanych wartości (oprócz new oraz delete)

Nie można zmienić priorytetu wykonywanych operatorów: a +b * c → a+(b*c), nie ma znaczenia, czym zajmują się operatory

Nie można zmienić tego, ile argumentów powinien mieć operator (jest albo jedno- albo dwuargumentowy), np. operator / musi być dwuargumentowy:

```
obiekt1 / obiekt2; //OK
/obiektA; //ZLE
obiektB/; ZLE
```

Operator ! Jest zawsze jednoargumentowy:

```
!obiekt ; //OK
obiektA ! ObiektB; //ZLE
```

Uwagi do przeciążania operatorów - II

Nie można zmieniać łączności operatorów, tzn. tego, czy łączy się z argumentem z prawej czy z lewej strony: a=b=c=d → a=(b=(c=d));

Jeśli funkcja operatorowa jest globalna, to przyjmuje tyle argumentów na ilu pracuje operator.

Jeśli operator dzielenia pracuje na dwóch argumentach, to operator/ też ma mieć dwa argumenty: pierwszy argument ma typ taki, jak obiekt stojący po lewej stronie, drugi argument ma typ taki, jak obiekt stojący po prawej stronie:

```
klasaA arg1;
klasaB arg2;
arg1 / arg2;
...
```

operator/(klasaA, klasaB); deklaracja funkcji operator/

Przynajmniej jeden z tych typów (nie ważne który) musi być zdefiniowany przez użytkownika

Argumenty w funkcji operatorowej nie mogą być domniemane.

Operator można przeładować wielokrotnie (przeciążenie nazwy funkcji)

FUNKCJA OPERATOROWA JAKO FUNKCJA SKŁADOWA KLASY

Funkcja operatorowa funkcją składową klasy

Jakie mogą być funkcje operatorowe:

- funkcje składowe klasy (albo!)
- globalne funkcje (zwykłe).

Przykład funkcji globalnej:

```
cmplx operator+(cmplx z1, cmplx z2) {
	cmplx z3;
	z3.re = z1.re + z2.re;
	z3.im = z1.im + z2.im;
	return z3;
}
```

Przykład funkcji składowej:

```
cmplx cmplx::operator+(cmplx z) {
	cmplx z3;
	z3.re = re + z.re;// z3.re = this->re + z.re; z3.re = re + z.re; TO SAMO
	z3.im = im + z.im; //z3.im = this->im + z.im; z3.im = im + z.im; TO SAMO
	return z3;
}
```

Funkcja operatorowa funkcją składową

```
class cmplx {
        public:
            float rez, imz;
            cmplx operator+(cmplx);
        ...
};

cmplx cmplx::operator+(cmplx z){
        cmplx z2;
        z2.rez = rez+z.rez;
        z2.imz= imz+z.imz;
        return z2;
}
...
int main() {
         cmplx zz1, zz2, zz3;
         ...
         zz3 = zz1 + zz2;
         //zz3 = zz1.operator+(zz2); //to samo, co linia wyżej
         ...
}
```

Funkcja operatorowa funkcją składową, a funkcją globalną

Skoro przy przekazywaniu argumentów bierze udział wskaźnik *this, to przeładowany operator nie może być funkcją statyczną.

Jeśli operator jest funkcją składową, to ma zawsze o jeden argument mniej niż miałby jako funkcja globalna. Wynika to z faktu istnienia wskaźnika *this.

Funkcja operatorowa - globalna NIE musi być przyjacielem klasy. Oczywiście może, ale przyjaźń nie jest obowiązkowa.

W praktyce jednak realizowane są najczęściej następujące przypadki:

- funkcja operatorowa jest składową klasy
- funkcja operatorowa jest funkcją globalną i zaprzyjaźnioną z klasą (przyjaźń daje im dostęp do składników prywatnych).

OPERATORY PREDEFINIOWANE, ARGUMENTOWOŚĆ OPERATORÓW

Operatory predefiniowane, argumentowość operatorów

Próba użycia operatora nie zdefiniowanego zakończy się błędem kompilatora (tożsame z wywołaniem niezdefiniowanej funkcji..)

Jest jednak kilka operatorów, które generowane są automatycznie dla każdej klasy (co nie oznacza, że nie można zdefiniować własnych).

= - przypisanie do obiektu danej klasy

& - pobranie adresu obiektu danej klasy

, - identycznie, jak dla typów wbudowanych

new, delete – kreacja i likwidacja obiektów w zapasie pamięci

Jeśli chcemy, aby operatory te wykonywały inną pracę niż ta, jaką wykonują wyżej wymienione operatory predefiniowane – konieczne jest zdefiniowanie własnych!

Operatory mogą być:

- jednoargumentowe (np. !)
- dwuargumentowe (np. +)

Operator wywołania funkcji () jest inny! Może zawierać on (jako jedyny operator) więcej argumentów.

Operatory jednoargumentowe

Operatory takie występują zazwyczaj jako przedrostek (stoją przed obiektem klasy - prefiks).

Przykłady:

```
int a=0;
cmplx z;
-a;
!a;          !z;
++a;
&a;          &z;
~a;          ~z;
```

Mogą też być operatory przyrostkowe (postfiks).

Przykłady:

```
i++;
i--;
```

Operatory jednoargumentowe mogą być zdefiniowane następująco:

(1) nieskładowa (zwykła) funkcja wywołana z 1 argumentem klasy klasaA: operator@(klasaA);

(2) składowa funkcja klasy K wywołana bez argumentów: klasaA::operator@(void);

Operatory dwuargumentowe

Operatory dwuargumentowe moga być zdefiniowane następująco:

(1) nieskładowa (zwykła) funkcja wywołana z 2 argumentami klasy klasaA: operator@(x,y);

(2) składowa funkcja klasy K wywołana bez argumentów: x::operator@(y);

Obie te funckje zostaną wywołane automatycznie, kiedy obok znaczka operatora @ znajdą Się dwa argumenty określnonych przez nas typów: x @ y;

Nie można przeładować tego samego operatora zarówno jako funkcji składowej klasy, jak i funkcji globalnej.

Przemienność w wykonywaniu operacji nie obowiązuje!

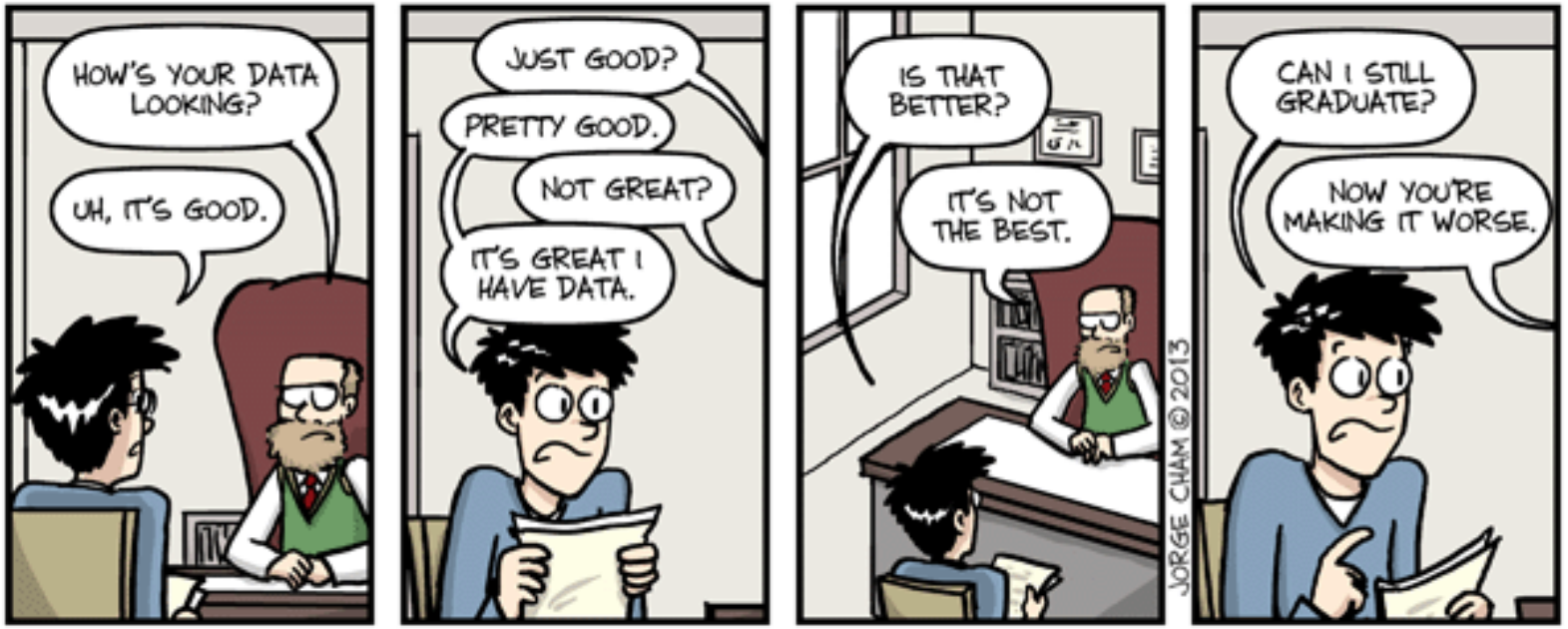
```
cmplx operator*(float ff, cmplx z1) {
        cmplx z2;
        z2.rez = ff * z1.rez;
        z2.imz = ff * z1.imz;
        return z2;
}
cmplx z(1.0, 1.0);
float a=3;
cmplx zz = a * z; //OK
cmplx zz = z * a; /ZLE, bo nie mamy tak przeciążonego operatora
```

Operatory dwuargumentowe

Funkcja operatorowa będąca funkcją składową klasy wymaga, aby obiekt stojący po lewej stronie (znaczka) operatora był obiektem tej klasy.

Operator będący zwykłą funkcją globalną tej klasy tego nie wymaga.

```
cmplx cmplx::operator*(float ff) {
        cmplx zz;
        zz.rez = rez * ff;
        zz.imz = imz * ff;
        return zz;
}
cmplx z(1.0, 1.0);
float a=3;
cmplx zzz = z * a; //OK
cmplx zzz = a * z; //ZLE
```



CZTERY OPERATORY KTÓRĄ MUSZĄ BYĆ NIESTATYCZNYMI FUNKCJAMI SKŁADOWYMI

(1) Operator przypisania =

Dwuargumentowy operator przypisania = służy do przypisanie jednemu obiektowy klasy treści drugiego obiektu tej samej klasy.

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

Jeśli `operator=` nie zostanie zdefiniowany, to zostanie on automatycznie wygenerowany, przepisane zostaną pola “składnik po składniku” (podobnie jak w przypadku automatycznie generowanego konstruktora kopiującego). W rezultacie takiego przypisania będą dwa obiekty bliźniaczej treści.

Kłopoty natomiast mogą pojawić się wtedy, kiedy składnikami klasy są wskaźniki lub jeśli klasa używa operatora new do rezerwacji miejsca w zapasie pamięci.

Operator przypisania składa się z części:

(1) sprawdzenie, czy nie jest kopiowane siebie samego

(2) części “destruktorowej” (np. zwalnianie pamięci)

(3) części “konstruktorowej”, przypominającej konstruktor kopiujący

Operator przypisania = przykład I

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

class czlowiek{
        char *imie;
        char *nazwisko;
        public:
        czlowiek();
        czlowiek(char*, char*);
        czlowiek(const czlowiek &);
        ~czlowiek();
        void przedstaw();
        czlowiek & operator=(const czlowiek &);
};
czlowiek::czlowiek() {
    imie = new char[100];
    nazwisko = new char[100];
}
```

```
czlowiek::czlowiek(char *ii, char *nn) {
        imie = new char[strlen(ii)+1];
        nazwisko = new char[strlen(nn)+1];
        strcpy(imie, ii);
        strcpy(nazwisko, nn);
}

czlowiek::czlowiek(const czlowiek &cz) {
        imie = new char[strlen(cz.imie)+1];
        nazwisko = new char[strlen(cz.nazwisko)+1];
        strcpy(imie, cz.imie);
        strcpy(nazwisko, cz.nazwisko);
        cout<<"Konstruktor kopiujacy.."<<endl;
}
czlowiek::~czlowiek() {
        delete [] imie;
        delete [] nazwisko;
}
```

Operator przypisania = przykład II

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

czlowiek & czlowiek::operator=(const czlowiek & cz) {
        if (&cz == this) return *this;
        delete imie;
        delete nazwisko;
        imie = new char[strlen(cz.imie)+1];
        nazwisko = new char[strlen(cz.nazwisko)+1];
        strcpy(imie, cz.imie);
        strcpy(nazwisko, cz. nazwisko);
        cout<<"Operator przypisania.."<<endl;
        return *this;
}
```

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

        czlowiek c1(i, n);
        czlowiek c2(i, r);
        czlowiek c3 = c1; //tu pracuje konstruktor kopiujący

        cout<<"Pierwsze przedstawienie.."<<endl;
        c1.przedstaw();
        c2.przedstaw();
        c3.przedstaw();

        c3 = c2; //tu pracuje operator przypisania

        cout<<"Drugie przedstawienie"<<endl;
        c1.przedstaw();
        c2.przedstaw();
        c3.przedstaw();
        return 1;
}
```

Kiedy operator= nie jest generowany automatycznie

(1) Jeżeli klasa ma składową const, tzn. że takiego pola nie wolno potem zmieniać (obecność operatora= jest wątpliwa..)

(2) Jeśli klasa ma składową będącą referencją (raz ustawionej referencji nie wolno już zmieniać)

(3) Jeśli dana klasa jest składową innej klasy, w której to operator przypisania = jest prywatny (jeśli operator jest prywatny, tzn. że nie chcemy, aby przypisanie odbywało się spoza klasy)

(4) Jeśli klasa pochodna ma klasę podstawową, w której operator przypisania jest prywatny (analogicznie jak w punkcie 3)

(2) Operator [] - I

Operator [] to operator odwołania się do elementu tablicy, jest dwuargumentowy. musi być przeładowany jako niestatyczna funkcja składowa klasy

```
class A;
A obiekt;
obiekt[argument]
Jest tożsame z: obiekt.operator[argument]
```

Uwaga! Argument wcale nie musi być typu int. W przypadku przeładowania operatora [] argument wcale nie musi oznaczać elementu tablicy, a obiekt wcale nie musi być tablicą. Zazwyczaj jednak operator [] przeładowuje się po to, aby funkcjonował w podobny sposób jak dla typów wbudowanych.

Dla typów wbudowanych operator [] może stać zarówno po lewej, jak i prawej stronie wyrażenia:

```
int tab[100], x;
x= tab[0]; //po prawej stronie
tab[4] = x; //po lewej stronie
tab[1] = tab[2]; //po obu stronach
```

Operator [] - II

```
class tablica {
        int a[100];
        public:
        int operator[](int i) { return a[i]; }
        }
};

tablica tt;
cout<<tt[10]<<"\t"<<tt.operator[](10)<<endl;
int tmp = tt[10]; //OK
tt[10] = tmp; //ZLE!
tt.operator[](10) = tmp; //ZLE
```

Nie jest to bowiem możliwe, aby do wyniku zwracanego przez funkcję wpisać pewną wartość. po lewej stronie równości może stać tylko wyrażenie oznaczające obiekt (także pokazywany przez wskaźnik lub referencję). Tylko wtedy, do takiego wskazania można przypisać wartość liczbową. Problem zostanie rozwiązany, jeśli funkcja operatorowa zwróci referencję do wybranego elementu w tablicy, a nie wartość, jaka jest tam wpisana.

Operator [] - III

```
class tablica {
        int a[100];
        public:
        int& operator[](int i) { return a[i]; }
        }
};

tablica tt;
cout<<tt[10]<<"\t"<<tt.operator[](10)<<endl;
int tmp = tt[10]; //OK
tt[10] = tmp; //OK
tt.operator[](10) = tmp; //OK
```

Chcąc napisać funkcję operatorową [], tak, aby operator [] mógł stać po obu stronach znaku = musimy zadeklarować, że funkcja ta będzie zwracała referencję do tego, czemu mamy przypisywać!

(3) Operator () - I

Operator () jest jedynym, który może mieć więcej niż dwa argumenty (świetnie się sprawdza do przeładowania operatora pracującego na tablicach wielowymiarowych! - operator [] może mieć tylko jeden argument – poza obiektem na rzecz którego jest wywołany). Operatora () można z powodzeniem użyć do skojarzenia z wykonywaniem jakiejś czynności. Nie zakłada się, by operator () stał po lewej stronie znaku = (tym samym zwraca void). (Jeśli mimo wszystko miałby stać po lewej stronie znaku przypisania, to musiałby zwracać referencję).

```
#include <iostream>
#define N 5
using namespace std;

class tab2d {
        int **tt;
        public:
        tab2d();
        ~tab2d();
        int& operator()(int, int);
};
```

Operator () - II

```
tab2d::tab2d() {
        tt= new int *[N];
    for (int i=0; i<N; i++)
                *(tt+i)= new int[N];


    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
                    *((*(tt+i))+j)= 0;
                    //tt[i][j] = 0; //to samo
}

tab2d::~tab2d() {
        for (int i=0; i<N; i++)
                delete [] tt[i];
    delete [] tt;
}
int& tab2d::operator()(int i, int j) {
        return *((*(tt+i))+j); //to samo, co return tt[i][j];
}
```

```
int main() {
        tab2d tab;
        tab.operator()(0,0) = 5;
        tab(1,1) = 10;
        tab(2,2) = tab(0,0);
        cout<<"[0,0] = "<<tab(0,0)<<endl;
        cout<<"[1,1] = "<<tab(1,1)<<endl;
        cout<<"[2,2] = "<<tab(2,2)<<endl;
        return 1;
}
```

(4) Operator ->

Jest to operator odniesienia się do składnika klasy.

Operator -> jest jednoargumentowy.

Operator -> działa na argumencie stojącym z lewej strony

(obiekt -> - argumentem operatora -> jest obiekt).

obiekt -> skladnik jest tożsame z (obiekt.operator->()).skladnik

W powyższym przykładzie argumentem operatora jest obiekt (lub referencja do niego).

Jeśli operator -> ma odnosić się do składnika obiektu, to funkcja operatorowa musi zwrócić wskaźnik do obiektu (ewentualnie referencję).

Zalety przeciążonego operatora ->:

- zwykły operator -> nie może być użyty do obiektu danej klasy (jedynie do wskaźnika)
- może wykonać dodatkową akcję w momencie sięgania do składnika klasy, którą należy określić w funkcji operatorowej.

INNE PRZYKŁADY PRZEŁADOWANYCH OPERATORÓW

Operator new

Operator new służy do rezerwacji obszarów pamięci.

Operator new jest funkcją składową statyczną (jest wywoływany na rzecz klasy, a nie konkretnego obiektu, który jest dopiero tworzony). Funkcja przeładowania operatora new musi zwracać wskaźnik typu void (*void), a jej pierwszy argument musi być typu size_t.

```
void* wektor::operator new(size_t rozmiar) {
        cout<<"Kreacja obiektu.."<<endl;
        return (new char[rozmiar]);
}
```

(słowo static może zostać automatycznie dodane przez kompilator)

```
wektor *wsk;
wsk = new wektor;
```

Klasa musi mieć zdefiniowany konstruktor domniemany.

Użycie wersji globalnej:

```
wektor *wsk2;
wsk2 = ::new wektor;
```

Operator delete

Operator delete służy do zwalniania zarezerwowanych obszarów pamięci. Operator delete jest funkcją składową statyczną (nawet jeśli zapomnimy dodać słowo static, to kompilator doda to za nas)

Funkcja przeładowania operatora new musi mieć jako pierwszy argument wskaźnik typu void (*void), a typ zwracanej wartości to void.

```
void wektor::operator delete(void * wsk) {
        cout<<"Kasowanie obiektu.."<<endl;
        delete wsk;
}
```

Klasa musi mieć zdefiniowany konstruktor domniemany.

Użycie wersji nie przeładowanej:

```
delete wsk1; //wersja przeladowana
::delete wsk2; //wersja globalna
```

Operator strumieniowy – wyjścia << - I

Klasa biblioteczna ostream (ang. output stream) – strumień wyjściowy pracuje na typach wbudowanych (int, double, float, char, char*, ...).

To dlatego możliwe jest użycie operatora << dla typów wbudowanych:

```
cout<<”Temperatura wynosi ”<<22.5<<”stopni C”<<endl;
```

Nie jest jednak możliwe napisanie:

```
cout<<punkt1<<endl;
```

Kiedy punkt1 jest obiektem klasy punkt (można co najwyżej wypisać odpowiednie pola tej klasy o ile są one publiczne..)

Operator strumieniowy – wyjścia << - II

Można przeciążyć więc operator<< (tylko jako funkcję globalną):

W klasie punkt mamy dwa pola prywatne: składową x oraz y

```
class punkt{
        double x,y;
        public:
        ...
        friend ostream& operator<<(ostream&, punkt&)
};
ostream& operator<<(ostream& wyjscie, punkt& pp) {
        wyjscie<<"punkt: ("<<pp.x<<", "<<pp.y<<")";
        return wyjscie;
}
```

(Jeśli współrzędne punktu są polami prywatnymi w klasie, to funkcję operatorową należy z klasą zaprzyjaźnić). Teraz jest możliwe:

```
punkt punkt1;
..
cout<<punkt1<<endl;
operator<<(cout, punkt1); //TO SAMO
```

Operator strumieniowy – wejścia >> - I

Klasa biblioteczna istream (ang. intput stream) – strumień wejściowy pracuje na typach wbudowanych (int, double, float, char, char*, ...).

To dlatego możliwe jest użycie operatora << dla typów wbudowanych:

```
char a;
cin>>a;
```

Nie jest jednak możliwe napisanie:

cin>>punkt1;

Kiedy punkt1 jest obiektem klasy punkt (można co najwyżej wpisać odpowiednie pola tej klasy o ile są one publiczne..)

Operator strumieniowy – wejścia >> - II

Można przeciążyć więc operator>> (tylko jako funkcję globalną):

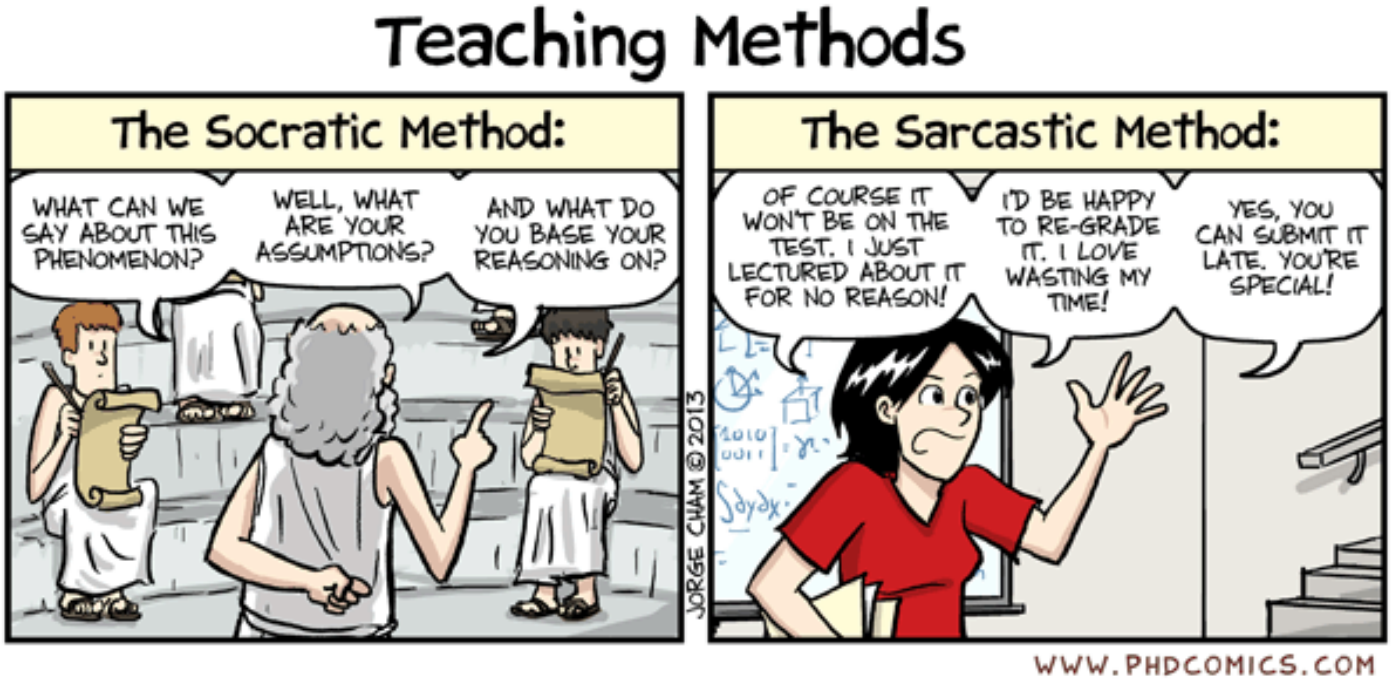
```
class punkt{
        double x,y;
        public:
        ...
        friend istream& operator>>(istream& , punkt&);
};

istream& operator>>(istream& klawiatura, punkt& pp) {
        klawiatura>>pp.x>>pp.y;
        return klawiatura;
}
```

(Jeśli współrzędne punktu są polami prywatnymi w klasie, to funkcję operatorową należy z klasą zaprzyjaźnić). Teraz jest możliwe:

punkt punkt1;

```
..
cin>>punkt1;
operator>>(cin, punkt1); //TO SAMO
```



UWAGI ODNOŚNIE PRZEŁADOWANIA OPERATORÓW

Kilka słów komentarza.. - I

Mechanizm przeładowania operatorów to dobrodziejstwo języka C++, z którego warto korzystać.

Nie ma sensu przeładowywać wszystkich operatorów na rzecz danej klasy, wybór przeładowanego operatora powinien kojarzyć się z czynnością przez niego wykonywaną (możliwe, choć nie rozsądne jest przeciążanie operatora* na rzecz odejmowania i odwrotnie..).

Nie ma potrzeby przeładowywać operatorów na siłę, a jedynie po to, aby uprościć sobie pisanie programu (jeśli np. nazwa funkcji lepiej oddaje jej czynność, to nie ma sensu używać operatora).

Przeładowanie operatora + oraz = nie pociąga za sobą jednoczesnego przeładowania ++ lub += (to są oddzielne operatory)

Jeśli operator dla typów wbudowanych nie modyfikuje obiektu, to w dobrym tonie jest zachowanie tej konwencji dla typu zdefiniowanego przez nas (chyba, że działanie modyfikacji jest całkowicie świadome)

Większość operatorów można przeładować na dwa sposoby:

- jako funkcję globalną (często zaprzyjaźnioną z klasą, choć przyjaźń jest obowiązkowa)
- jako funkcję składową klasy

Kilka słów komentarza.. - II

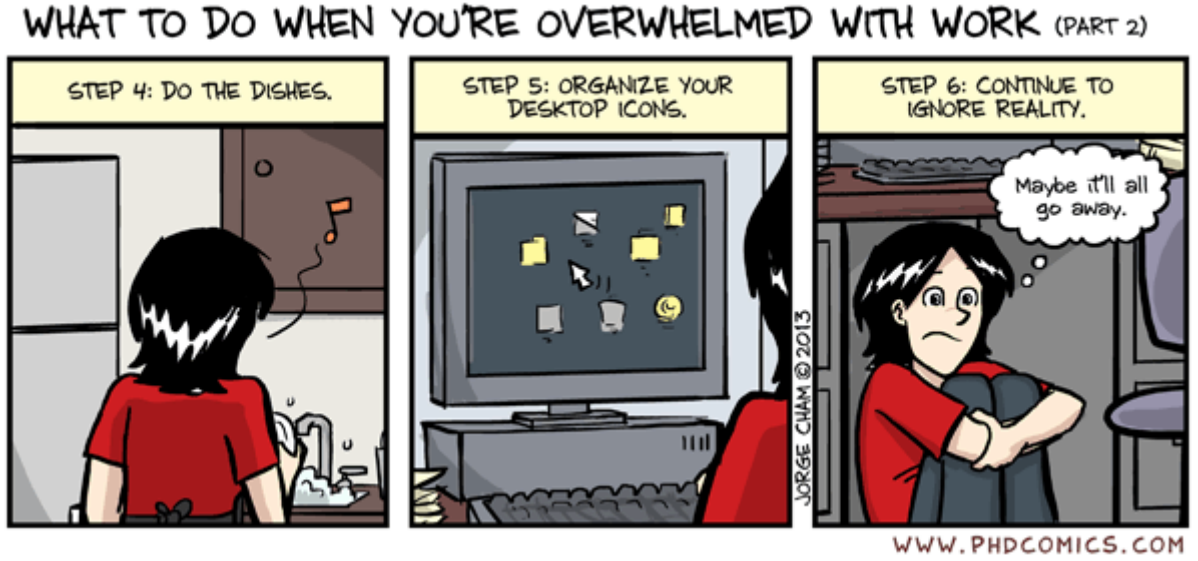
Jeśli operator modyfikuje obiekt, na którym pracuje, to powinien być zdefiniowany jako funkcja składowa klasy. Operatory te zwracają l-wartość.

Jeśli operator sięga po obiekt, aby jedynie na nim pracować bez jego modyfikacji, to lepiej jest zdefiniować go jako globalną funkcję. Jeśli ma pracować na polach prywatnych danej klasy, to należy przeciążany operator zaprzyjaźnić z klasą.

Użycie operatora dodawania:

```
cmplx a, b, c;
...
c= a.operator+(b); //operator jako składowa klasy
c= operator+(a, b); //operator jako funkcja globalna
c= a+b; //przeciazenie operatora
```

Przypomnienie: nie wolno jest jednocześnie przeładować danego operatora zarówno jako funkcji składowej klasy oraz funkcji globalnej (co w przypadku ostatniej próby wywołania?)



KONIEC WYKŁADU 5