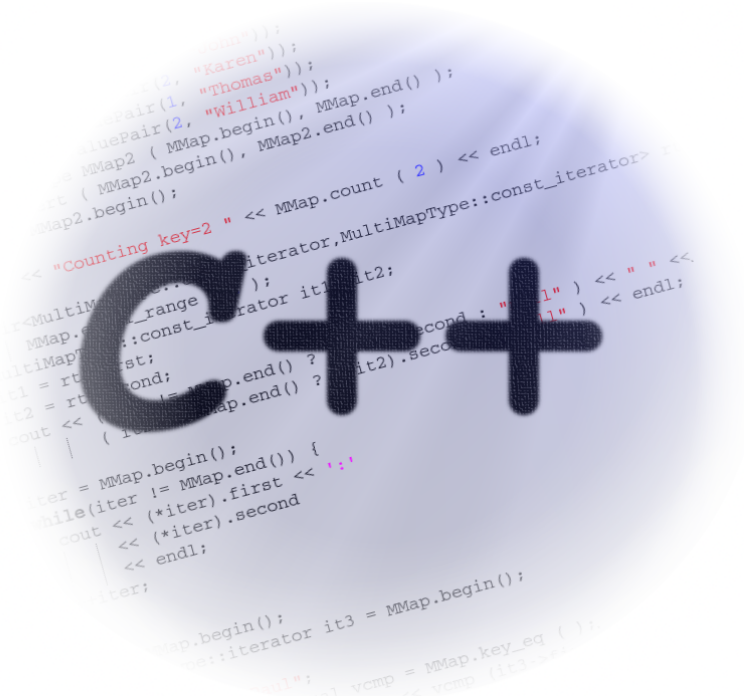
Języki Programowania

Prowadząca:
dr inż. Hanna Zbroszczyk

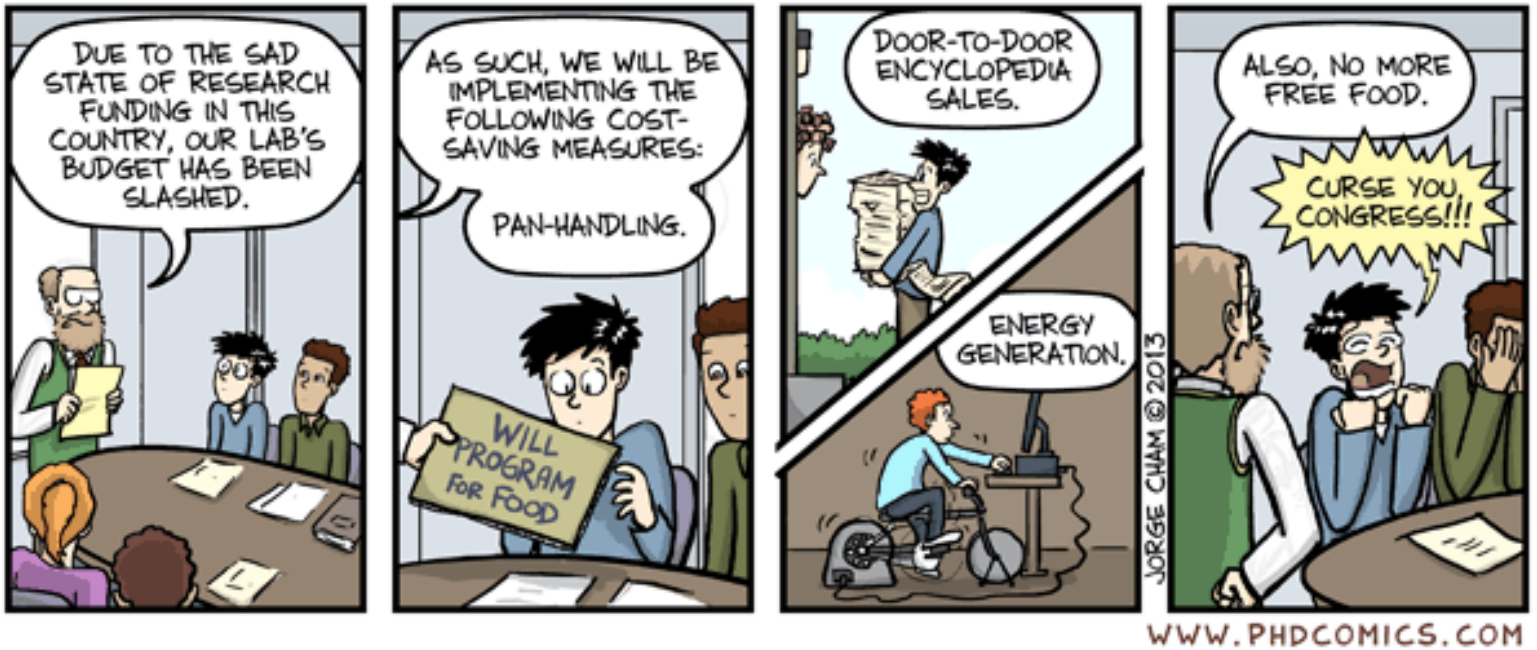
e-mail: *hanna.zbroszczyk@pw.edu.pl*
tel: +48 22 234 58 51

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)


DUE TO THE SAD STATE OF RESEARCH FUNDING IN THIS COUNTRY, OUR LAB'S BUDGET HAS BEEN SLASHED.
AS SUCH, WE WILL BE IMPLEMENTING THE FOLLOWING COST-SAVING MEASURES:
PAN-HANDLING.
WILL PROGRAM FOR FOOD
DOOR-TO-DOOR ENCYCLOPEDIA SALES.
ENERGY GENERATION.
ALSO, NO MORE FREE FOOD.
CURSE YOU, CONGRESS!!!
JORGE CHAM © 2013
WWW.PHDCOMICS.COM

KONWERSJE

RZUTOWANIE WEDŁUG TRADYCYJNYCH METOD

Rzutowanie NIE zalecane

Operatory rzutowania służą do jawnego przekształcania typów, są jednoargumentowe. Ich działanie polega na tym, że biorą wyrażenie lub obiekt jednego typu i zwracają wyrażenie lub obiekt innego typu. “Tradycyjne metody” rzutowania dostępne były już w dawnych wersjach języka C++, ale ich działanie pozwalało rzutować bez ograniczeń, co nie było dobre. W obecnym standardzie C++ są dostępne bardziej subtelne narzędzia do rzutowania, nie mniej używanie tych starych w dalszym ciągu się pojawia..

```
(nazwa_innego_typu)wyrazenie
nazwa_innego_typu(wyrazenie)
```

Przy rzutowaniu należy się liczyć ze stratą informacje (np. rzutowanie int'a na char'a), dlatego kompilator przy próbie przypisania ostrzeże. Przy jawnej konwersji ostrzeżenia nie będzie (kompilator uzna to za działanie celowe).

W praktyce powinno się unikać stosowania tradycyjnego rzutowania, chyba, że mamy świadomość konsekwencji, jakie mogą z takiego rzutowania wyniknąć.

Przykład rzutowania:

```
char c;
cin>>c;
int i = (int)c;
```

Rzutowanie za pomocą nowych operatorów rzutowania - 1

(1) Operator static_cast

static_cast<nazwa_innego_typu> (wyrazenie);

Operator static_cast powinien być używany do sytuacji “możliwych” lub “możliwych w pewnych sytuacjach”.

Stosując ten operator bierzemy na siebie odpowiedzialność za sens danej konwersji.

```
double e = 2.718282;
int ee = e; //kompilator ostrzeze przed utrata informacji
int ee =static_cast<int>(e);  //kompilator uzna to za dzialanie swiadome
```

A w przypadku dziedziczenia:

```
pojazd p;
samochod s;
pojazd *wsk_poj;
samochod *wsk_sam = &s;
wsk_poj = wsk_sam; //OK
wsk_sam = wsk_pojazd; //ZLE
wsk_sam = static_cast<samochod*>(wsk_pojazd);
```


Za pomocą operatora static_cast nie jest możliwe pozbycie się przydomka const lub volatile.

Rzutowanie za pomocą nowych operatorów rzutowania - 2

(2) Operator const_cast

const_cast<nazwa_typu_nie_const> (wyrazenie o typie z const);

Operator const_cast służy do pozbycia się (lub nadania!) przydomka const lub volatile.

```
cost double e = 2.718282;
double ee = e; //kompilator ostrzeze przed utrata informacji
double ee =const_cast<double>(e);  //kompilator uzna to za dzialanie swiadome
```

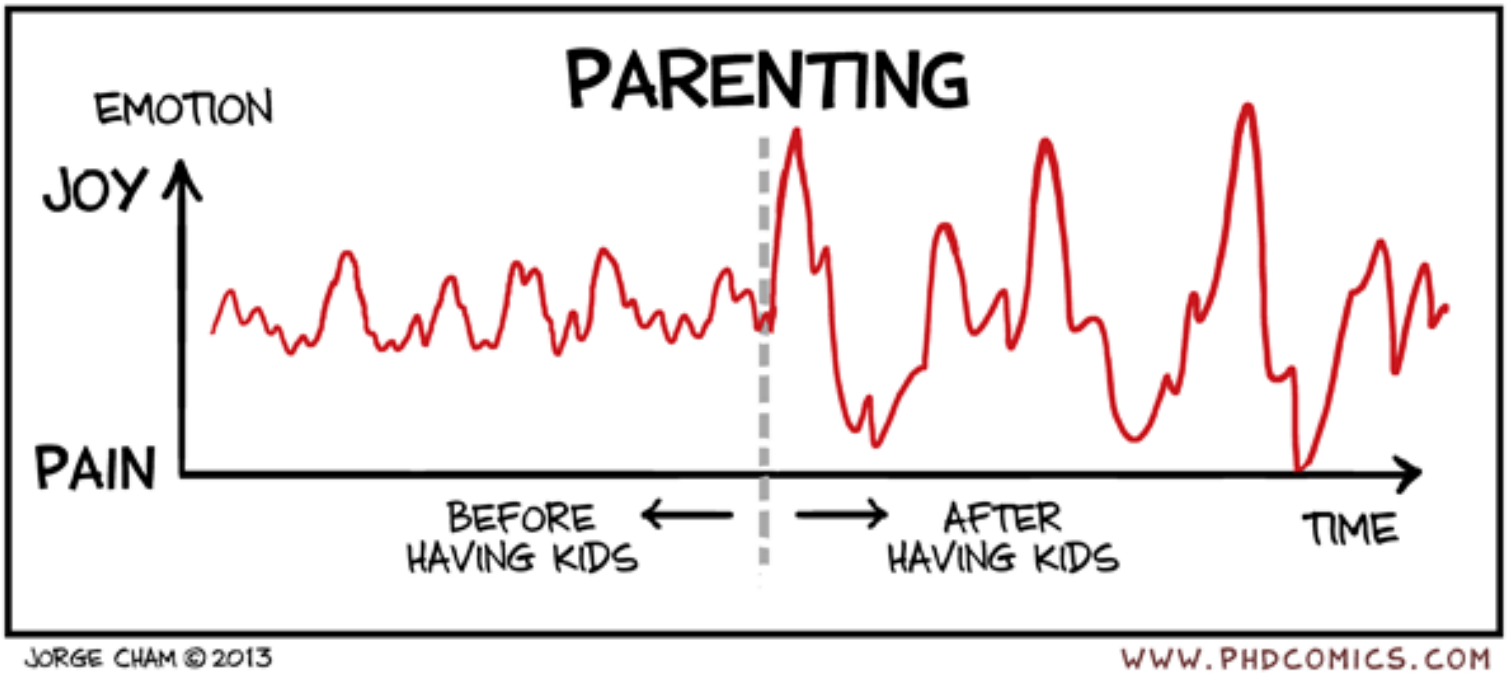
(3) Operator dynamic_cast

Operator ten dokona rzutowania tylko w sytuacji, kiedy ma to sens (nie dzieje się nic tylko na odpowiedzialność użytkownika)

(4) Operator reinterpret_cast

adr_nowy = reinterpret_cast<nazwa_typu_wsk*> (adr_nowy);

Operator ten pozwala rzutować wskaźniki danego typu na wskaźniki drugiego typu.


PARENTING
EMOTION
JOY
PAIN
BEFORE HAVING KIDS
AFTER HAVING KIDS
TIME
JORGE CHAM © 2013
WWW.PHDCOMICS.COM

KONWERSJE

Konwersje – wstęp - 1

Rozważmy klasę liczb zespolonych:

```
class cmplx {
        float rez, imz;
        public:
        cmplx(float a, float b) : rez(a), imz(b) {}
        ...
        friend cmplx dodaj(cmplx, cmplx);
};
cmplx dodaj(cmplx z1, cmplx z2) {
        cmplx z3(0.,0.);
        z3.rez = z1.rez+z2.rez;
        z3.imz = z1.imz+z2.imz;
        return z3;
}
int main()
{
        cmplx z1(0.,0.);
        cmplx z2(1.,1.);
        cmplx z3(0.,0.0);
        z3 = dodaj(z1,z2);
...
}
```

Konwersje – wstęp - 2

...ale wiemy, że liczby rzeczywiste są szczególnym przypadkiem liczby zespolonej..

z3 = dodaj(z1,0.5);

Nie zgadza się typ argumentu, taka operacja nie jest możliwa.. należałoby sformułować dodającą funkcję przeciążoną:

```
cmplx dodaj(cmplx z1, double d) {
        cmplx z3(0.,0.);
        z3.rez = z1.rez+d;
        z3.imz = z1.imz+0.0;
        return z3;
}
cmplx dodaj(double d, cmplx z2) {
        cmplx z3(0.,0.);
        z3.rez = d+z2.rez;
        z3.imz = 0.0+z2.imz;
        return z3;
}
```

Jak w takim wypadku “przekonać” kompilator, że liczba rzeczywista jest szczególnym przypadkiem liczby zespolonej?

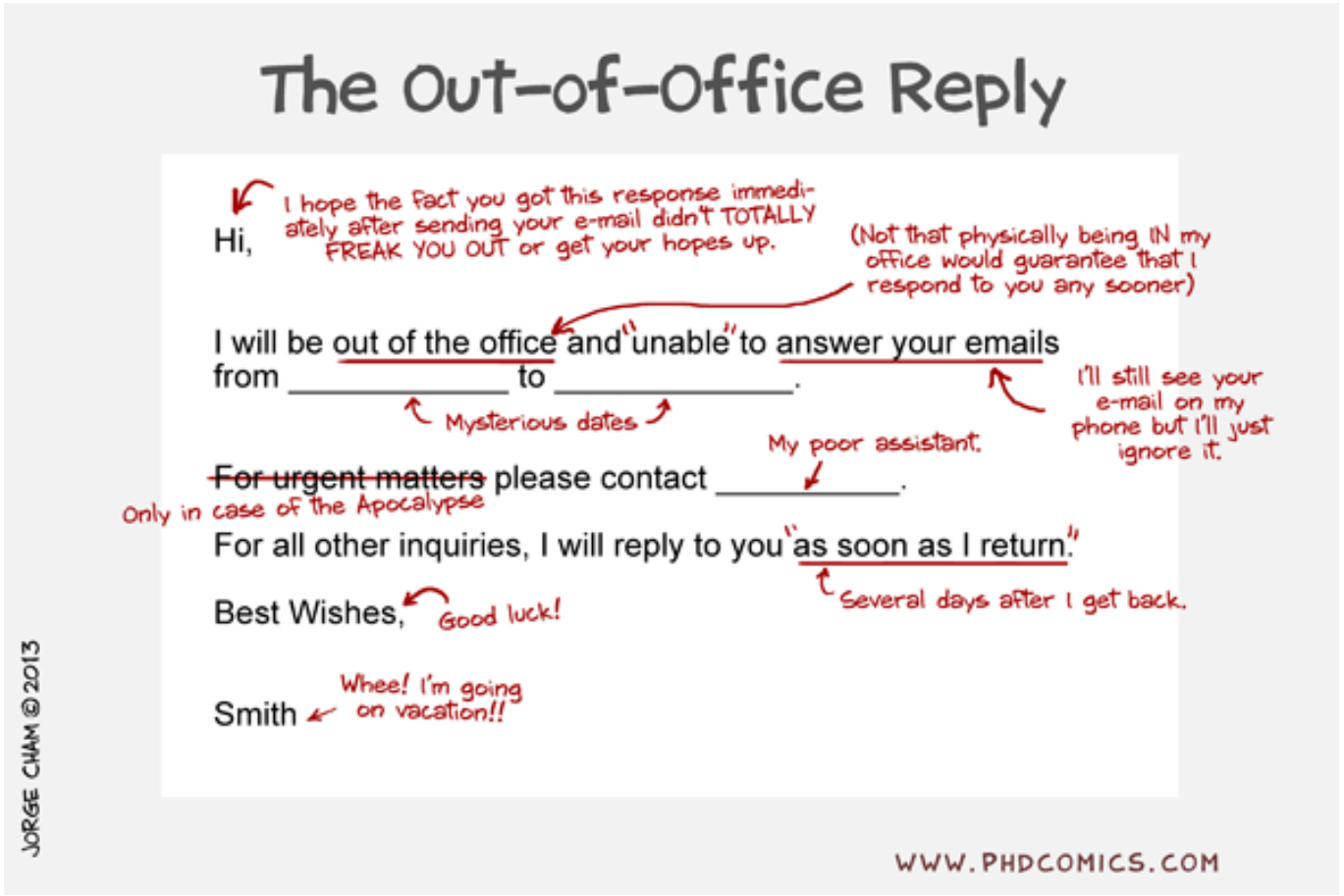
Konwersje – wstęp - 3

Konwersje obiektu A na Z mogą być definiowane przez użytkownika:

(1) Konstruktor klasy Z przyjmujący jako jeden argument obiekt klasy A

(2) Specjalna funkcja składowa klasy A (tzw. funkcja konwertująca – operator konwersji)

W konwersjach definiowanych przez użytkownika chodzi o to, aby mogły zachodzić niejawnie (samoczynnie), zawsze wtedy, gdy zachodzi niedopasowanie typów. Kompilator w takim wypadku sprawdza, czy nie ma jakiejś funkcji konwertującej, by móc zamienić jeden typ w drugi. Konwersje mogą też zachodzić w sposób jawny (to tak, jakby tę funkcję konwertującą wywołali sobie jawnie).



KONSTRUKTOR KONWERTUJĄCY

Konstruktory konwertujące - 1

...to taki konstruktor, który w swojej klasie A przyjmie jeden argument (np. typu Z)

A::A(Z)

Nie posiada przydomka explicit, określa konwersję Z->A

(jeśli konstruktor ma przydomek explicit, to znaczy, że nie wolno go użyć niejawnie – do konwersji)

Taki konstruktor można uzyskać wtedy, kiedy np. drugi argument stanie się domniemany:

cmplx(float a, float b=0.0) : rez(a), imz(b) {}

Dzięki takiemu konstruktorowi mamy możliwość używania jednej funkcji:

```
cmplx dodaj(cmplx, cmplx);
```

zamiast:

```
cmplx dodaj(cmplx, cmplx);
cmplx dodaj(cmplx, double);
cmplx dodaj(double, cmplx);
cmplx dodaj(double, double);
```

Konstruktory konwertujące - 2

Rozpatrzmy przykład:

```
#include <iostream>
using namespace std;
class cmplx {
        float rez, imz;
        public:
        explicit cmplx(float a=0, float b=0.0) : rez(a), imz(b) {}
        friend ostream& operator<<(ostream&, cmplx&);
};
ostream& operator<<(ostream& wyj, cmplx& c) {
        wyj<<"Liczba zespolona: "<<c.rez<<"+i"<<c.imz<<endl;
        return wyj;
}
```

Konstruktory konwertujące - 3

```
int main()
{
        cmplx z1;
        cmplx z2(1.5);
        //cmplx z3= 3.5;
        cmplx z4 = cmplx(2.5);
        cmplx *z5 = new cmplx(4.0);
        cmplx z6 = (cmplx)4.5;
        cmplx z7 = static_cast<cmplx>(5.5);
        cout<<z1;
        cout<<z2;
        //cout<<z3;
        cout<<z4;
        cout<<*z5;
        cout<<z6;
        cout<<z7;
   return 1;
}
```

```
Liczba zespolona: 0+i0
Liczba zespolona: 1.5+i0

Liczba zespolona: 2.5+i0
Liczba zespolona: 4+i0
Liczba zespolona: 4.5+i0
Liczba zespolona: 5.5+i0
```

Konstruktory konwertujące - 4

Konwertować niekoniecznie trzeba z typu wbudowanego, można przekształcać z typu przez nas stworzonego, czyli z innej klasy. Konstruktor musi przyjrzeć się obiektowi tamtej klasy i na jego podstawie stworzyć obiekt swojej klasy – taki konstruktor powinien mieć zatem dostęp do jego pól.

- pola w klasie obiektu, z którego konwertujemy są publiczne (niekoniecznie dobre rozwiązanie)

- klasa, do której należy obiekt konwertowany deklaruje przyjaźń z konstruktorem nowej klasy

- publiczne funkcje składowe klasy, do której należy konwertowany obiekt pozwalające uzyskać stosowne informacje.

FUNKCJA KONWERTUJĄCA

Funkcja konwertująca – operator konwersji - 1

Możemy mieć do czynienia z sytuacja odwrotną, kiedy mamy funkcję:

void funkcja(double x);

A chcemy ją wywołać z argumentem innego typu:

cmplx obiekt;

funkcja(obiekt);

Poprzednie rozwiązanie nie jest możliwe do zrealizowania. Nie możemy przecież dopisać konstruktora w klasie double.

Należy więc klasę cmplx wyposażyć w funkcję konwertującą (operator konwersji), ogólnie: A::operator B(); u nas:

cmplx::operator double();

Musimy być przede wszystkim świadomi skutków takiej konwersji. Tak długo, póki tak jest (chcemy np. wyłuskać część rzeczywistą liczby zespolonej) używanie operatora konwersji ma sens.

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

Funkcja konwertująca – operator konwersji - 2

Od teraz możliwe jest następujące użycie:

```
void funkcja(int);
cmplx z1(1.0, 2.0);
funkcja(z1);

A także:
int I = (int)z1;
int I = int(z1);
```

Cechy funkcji konwertujących:

(1) musi być składową klasy (jest w niej wskaźnik *this do obiektu poddanego do konwersji)

(2) nie ma określonego typu zwracanego (bo zwraca taki typ, jaki sama zawiera..)

(3) ma pustą listę argumentów (nie można jej przeładować)

(4) jest dziedziczona

(5) może być funkcją wirtualną

Funkcja konwertująca – operator konwersji – program - 1

Konwertować można także na typy zdefiniowane przez użytkownika.

```
#include <iostream>
using namespace std;
class numer;
class cmplx {
        float rez, imz;
        public:
        cmplx(float a=0.0, float b=0.0) : rez(a),
                imz(b) {}
        cmplx(numer);
        operator float();
        operator numer();
        friend ostream& operator<<(ostream&,
                cmplx&);
        friend cmplx dodaj(cmplx, cmplx);
};
```

```
class numer {
        float n;
        string opis;
        public:
        numer(float k, string ss="") : n(k), opis(ss) {}
        operator float() {return (float) n;}
        friend cmplx::cmplx(numer);
};

cmplx::cmplx(numer ob): rez(ob.n), imz(0.0) {}

cmplx::operator float() {
        return (float) rez; //return rez;
}
cmplx::operator numer() {
        numer nn(rez, "powstal z zespolonej");
        return nn;
}
```

Funkcja konwertująca – operator konwersji – program - 2

```
ostream& operator<<(ostream& wyj,
        cmplx& c) {
        wyj<<"Liczba zespolona: "<<c.rez
                <<"+i"<<c.imz<<endl;
        return wyj;
}

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

float pole_kwadratu(float bok) {
        return bok*bok;
}
```

```
int main() {
        float x = 5.5;
        numer nn(13, "jego numer..");
        cmplx z1(2.5, 1.5);
        cout<< pole_kwadratu(x)<<endl; //niepotrzeba konwersja
        cout<< pole_kwadratu(nn)<<endl; //konwerja numer -> float
        cout<< pole_kwadratu(z1)<<endl; //konwersja cmplx -> float
        cmplx z2(1.5, 2.5);
        cmplx z3;
        z3 = dodaj(z1,z2); //niepotrzebna konwersja
        cout<<z3<<endl;
        z3 = dodaj(z1,x); //konwersja float -> cmplx
        cout<<z3<<endl;
        z3 = dodaj(z1, nn); //konwersja numer ->cmplx
        cout<<z3<<endl;
    return 1;
}
```

Funkcja konwertująca – operator konwersji – ciąg dalszy..

Na co nie konwertujemy:

Jeśli w klasie A zdefiniujemy funkcję konwertującą na typ Z, to ten typ Z nie powinien być:

- tym samym co typ A
- typem &A
- typem void
- typem jakiejś funkcji
- typem reprezentującym tablicę
- typem klasy podstawowej klasy A

Konstruktor jednoargumentowy kontra operator konwersji - 1

W przypadku przekształcenia jednego typu w drugi mamy do wyboru narzędzia:

- konstruktor jednoargumentowy
- operator konwersji (funkcję konwertującą)

Konstruktor zapewnia konwersję obcego typu na typ swojej klasy.

Funkcja konwertująca przekształca od typu swojej klasy na typ inny.

Są to konwersje definiowane przez użytkownika.

Jeśli mamy dokonać konwersji z typu A do Z to:

- albo piszemy konstruktor w klasie Z
- albo piszemy funkcję konwertująca w klasie A

(nie można zastosować obu jednocześnie – kompilator przy przekształceniu niejawnym nie będzie wiedział którego sposobu użyć, jedynie użytkownik stosując konwersję jawną)

Konstruktor jednoargumentowy kontra operator konwersji - 2

Wady konstruktora jednoargumentowego:

- nie można zdefiniować konstruktora dla typu wbudowanego
- nie można napisać konstruktora klasy, do której nie mamy dostępu, np. do klasy bibliotecznej
- w przypadku, kiedy taki konstruktor można stworzyć, ta druga klasa musi zapewnić dostęp do swoich składników
- argument w konstruktorze musi pasować dokładnie do typu argumentu deklarowanego
- konstruktor służący do konwersji nie jest dziedziczony (jak I żaden konstruktor).

  Funkcja konwertująca jest funkcją składową, może być wirtualna (dziedziczona “inteligentnie”)

<u>Wnioski:</u>

- w przypadku konwersji klasy A na typ wbudowany należy posłużyć się operatorem konwersji
- w przypadku konwersji obiektu klasy A na obiekt klasy Z:
  - lepiej napisać funkcję konwertującą w klasie A – nie wymaga to ingerencji klasę Z
  - jeśli klasa A jest niedostępna, to w klasie Z należy napisać konstruktor konwertujący obiekt klasy A na Z : Z::Z(A)

Kiedy zachodzi niejawna konwersja?

(1) Niezgodność argumentów wywołanych z argumentami formalnymi,

(2) Niezgodność przy zwracanym wyniku funkcji (return) oraz zadeklarowanego typu dla tej funkcji

(3) Przy operatorach (np. dodawania)

(4) W sytuacji:

```
A obj;
if (obj)..
switch(obj)..
while(obj)..
for(..; obj; ...)
```

Kompilator będzie chciał wykonać operacje: (int)obj, należy stworzyć konstruktor:

```
A::operator int ();
```

(5) W przypadku niejawnej konwersji typu, np:

```
cmplx z1;
float re = z1;
```

Konwersja jawna

```
cmplx z1(1,1);
inny n;
n = (inny)z1; //”forma rzutowania”
n = inny (z1); //”forma funkcji”
```

W wielu sytuacjach obie formy są jednoznaczne, ale nie zawsze..

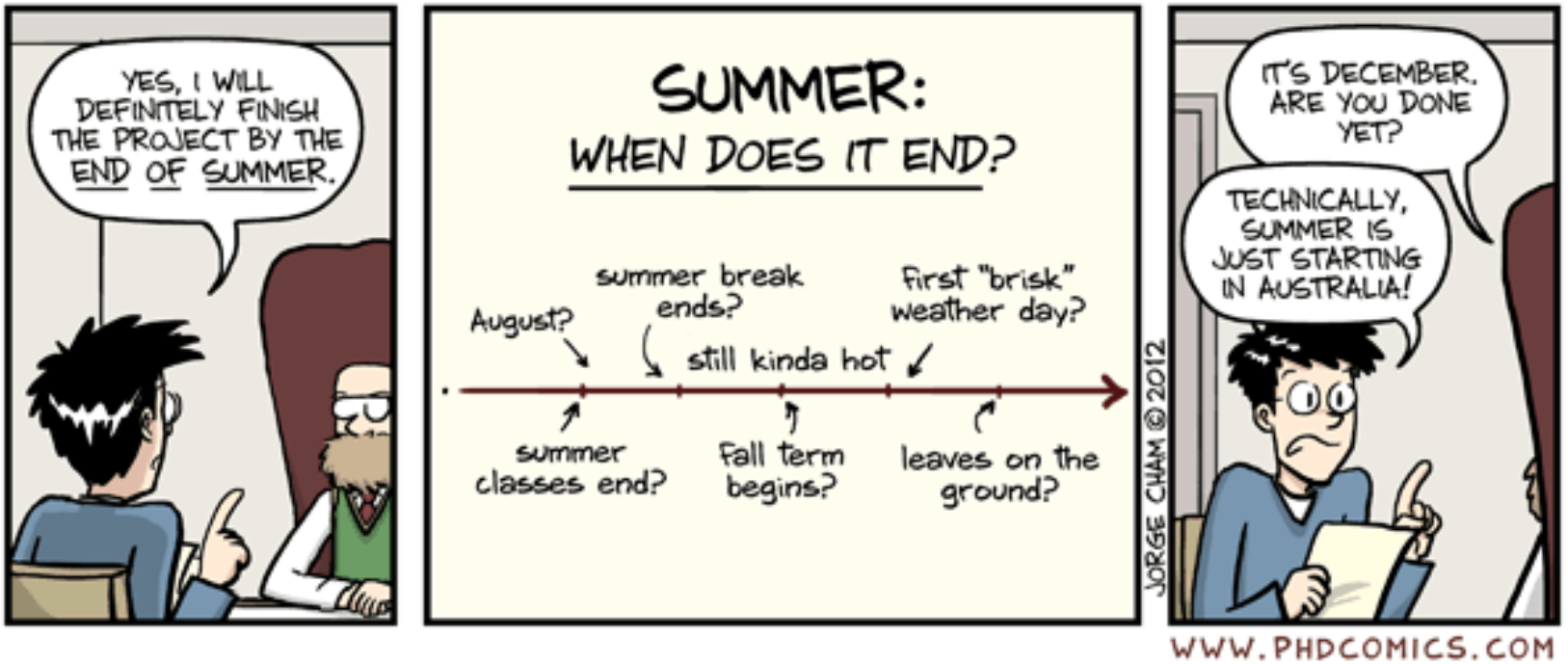
(1) “forma funkcji”: inny::inny(cmplx z,int i = 5) to konstruktor konwertujący

Ale przy próbie: n = inny(z1, 15) – możliwa jest tylko “forma funkcji”

(2) “forma rzutowania””

```
A::operator char*();
A obj;
char *napis;
napis = (char*) obj; //OK
napis = char * (obj); //ZLE
```

Jest jeszcze rzutowanie static_cast..



BIBLIOTECZNA KLASA *STD::STRING* DO OPERACJI Z TEKSTAMI

Biblioteczna klasa *std::string* - 1

Dla typów wbudowanych, np. int bardzo łatwo jest wykonywać operacje matematyczne, inicjalizować je lub przypisywać im nowe wartości. Niestety nie jest to takie proste dla ciągów znakowych (C-string'ów), ponieważ mimo, że one istnieją, to nie są typem wbudowanym.

W C++ nie ma żadnych operacji na C-string'ach.

C-string to string (ciąg znakowy) w rozumieniu języka C. Jest to stała tekstowa – napis – ciąg znaków ujętych w cudzysłów. C-string'i są przechowywane w pamięci jaki ciąg znaków, na którego to końcu dodawany jest znak 0 w kodzie – NULL; w ten sposób kompilator oznacza sobie koniec ciągu znakowego. To dlatego długość każdego C-string'u jest powiększona o 1 znak. W dawniejszych implementacjach języka C++ C-string był typu char[n+1], w nowszych implementacjach jest typu: const char[n+1].

Definiowanie obiektów klasy *std::string*

Jak w przypadku każdej klasy posługujemny się konstruktorami tej klasy:

```
void wyswietl(string str1, string str2) { cout<<str1<<"\t"<<str2<<endl; }
int main() {
        string ss1;
        string ss2("String drugi");
        char tab[15] = {"String trzeci"};
        string ss3(tab);
        string ss4(&tab[7]);
        string ss5("String piaty", 4);
        string ss6(10, '*');
        string ss7 = "String siodmy";
        string ss8(ss7, 4, 6);
        wyswietl("ss1", ss1);
        wyswietl("ss2", ss2);
        wyswietl("ss3", ss3);
        wyswietl("ss4", ss4);
        wyswietl("ss5", ss5);
        wyswietl("ss6", ss6);
        wyswietl("ss7", ss7);
        wyswietl("ss8", ss8);
        return 1;
}
```

```
ss1
ss2 String drugi
ss3 String trzeci
ss4 trzeci
ss5 Stri
ss6 **********
ss7 String siodmy
ss8 ng sio
```

Użycie operatorów =, +, += w pracy ze stringami

Operatory =, +, += w pracy ze stringami niezwykle uproszczają posługiwanie się typami znakowymi

Operator =

```
string stary = “przyklad uzycia operatora =”
string nowy = stary;
cout<<nowy<<endl; //przyklad uzycia operatora =
```

Operator +

```
string cz_1 = “przyklad uzycia”;
string cz_2 = “operatora +”;
String nazwa = cz_1 + cz_2;
cout<<nazwa<<endl; //przyklad uzycia operatora +
```

Operator +=

```
string nazwa(“przyklad uzycia”);
string cz_2(“operatora +=”);
nazwa+=cz_2;
cout<<nazwa<<endl; //przyklad uzycia operatora +=
```

Przegląd funkcji składowych klasy *std::string* - 1

(1) Sprawdzenie długości stringu:

```
size_type string::size();
size_type string::length();
```

(2) Sprawdzenie, czy string jest pusty:

```
bool string::empty();
```

(3) Sprawdzenie jak duża jest rezerwacja miejsca na przechowanie tekstu:

```
size_type string::capacity();
```

(4) Dodatkowa rezerwacja miejsca na przyszły tekst:

```
void string::reserve(size_type);
```

(5) Zmiana długości stringu do określonej na nowo:

```
void string::resize(size_type, char);
```

Przy wydłużeniu stringu zostają dopisane znaki podane w drugim argumencie

Przy skróceniu stringu tekst zostanie ucięty

Przegląd funkcji składowych klasy *std::string* - 2

(6) Kasowanie dotychczasowej zawartości stringu:

```
void string::clear();
```

(7) Odniesienie się do konkretnego elementu stringu (w razie błędu wyrzucany jest wyjątek):

```
char& string::at(size_type);
```

(8) Odniesienie się do konkretnego elementu stringu (w razie błędu nie ma jego sygnalizacji)

```
char& string::operator[](size_type);
```

(9) Przeszukanie konkretnego znaku (lub stringu) w danym stringu .

```
size_type string::find(const string &, size_type=0);
size_type string::find(char, size_type=0);
size_type string::rfind(const string &, size_type=npos);
size_type string::rfind(char, size_type=npos);
```

W przypadku sukcesu zwracany jest numer pozycji, gdzie znajduje się znak lub zaczyna string, w przypadku porażki zwracany jest string::npos. Funkcje szukają od początku string'u do końca (find) lub od końca do początku (rfind)

Przegląd funkcji składowych klasy *std::string* - 3

(10) Poszukanie w danym stringu dowolnego znaku z (..of..) lub spoza (..not_of..) załączonego zestawu. Przeszukiwanie rozpocznie się od pozycji wskazanej w drugim argumencie. Przeszukiwanie ..first.. Rozpocznie się od początku pliku, a przeszukiwanie ..last.. od końca pliku. Rezultatem jest wynik pozycji Pierwszego znalezienia lub string::npos w przypadku nie znalezienia.

```
size_type string::find_first_of(const string&, size_type=0);
size_type string::find_first_not_of(const string&, size_type=0);
size_type string::find_last_of(const string&, size_type=npos);
size_type string::find_last_not_of(const string&, size_type=npos);
```

(11) Kasowanie znaków w stringu począwszy od zadanej pozycji (pierwszy argument) przez ilość znaków określonych argumentem drugim

```
string& string::erase(size_type=0, size_type=npos);
```

(12) Wstawienie do środka stringu, począwszy od pozycji wyrażonej przez argument pierwszy, takiego stringu, który jest wpisany w argument drugi, zaczynając od pozycji określonej w argumencie trzecim i ma on długość równą ilości znaków wyrażoną czwartym argumentem

```
string& string::insert(size_type, const string &, size_type, size_type);
```

Przegląd funkcji składowych klasy *std::string* - 4

(13) Zastąpienie fragmentu stringu przez inny tekst. Pierwszy argument mówi odkąd zastępować, drugi argument ile zastąpić, trzeci argument co wstawić, czwarty argument mówi od którego znaku ze stringu wstawianego należy wziąć, a piąty argument o tym, ile znaków z nowego stringu wziąć.

```
string& string::replace(size_type, size_type, const string&, size_type, size_type);
```

(14) Udostępnienie treści stringu w postaci C-string'u:

```
const char* string::c_str();
```

(15) Udostępnienie adresu tablicy, w której obiekt string przechowuje swoje dane:

```
const char* string::data();
```

(16) Porównanie części stringu (określony swym początkiem przez pierwszy argument oraz długością określoną przez drugi argument) z innym stringiem przesłanym w trzecim argumencie

```
int string::compare(size_type, size_type, const string&);
```

W celu porównania całych stringów wygodniej jest użyć operatorów: ==, !=, <, >

Przegląd funkcji składowych klasy *std::string* - 5

(17) Kopiowanie fragmentu obiektu klasy string do zwykłej tablicy znaków (pierwszy argument). Drugi argument określa ile znaków ma być skopiowanych, trzeci argument mówi o tym ile znaków ma być skopiowanych.

```
size_type string::copy(char*, size_type, size_type=0);
```

(18) Dwa obiekty klasy string zamieniają się zawartościami

```
void string::swap(string&);
```

(19) Przypisanie nowej treści do stringu (działanie podobne jak operatora przypisania=)

```
string string::assign();
```

(20) Dopisanie nowej zawartości do istniejącego stringu (działanie jak z operatorem+=)

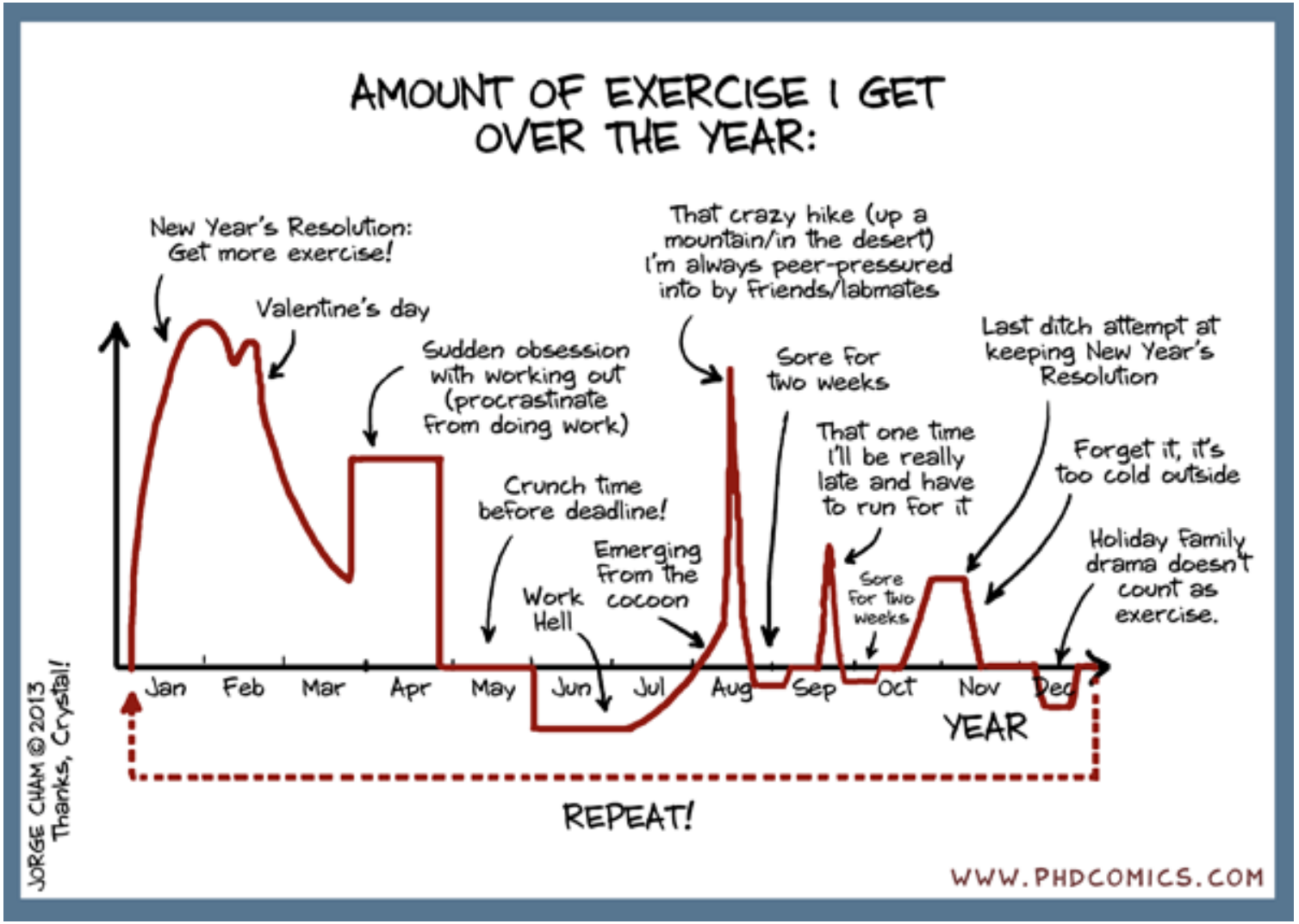
```
string string::append();
```

(21) Wczytanie z klawiatury całej linii (nie tylko jednego wyrazu), aż do napotkania znaku ograniczającego

```
istream& getline(istream, string, char);
```

Przykład użycia (to nie jest funkcja składowa!!):

```
string miasto;
getline(cin, miasto, 'v');
```



KONIEC WYKŁADU 11

Nieobowiązkowe zadania do wykonania

1. Zaprojektować i napisać klasy (w tym klasa ATD) do oddziaływań (grawitacyjne, elektromagnetyczne, silne, ..). Program powinien mieć funkcjonalność wyliczenia siły oddziaływania oraz potencjału.