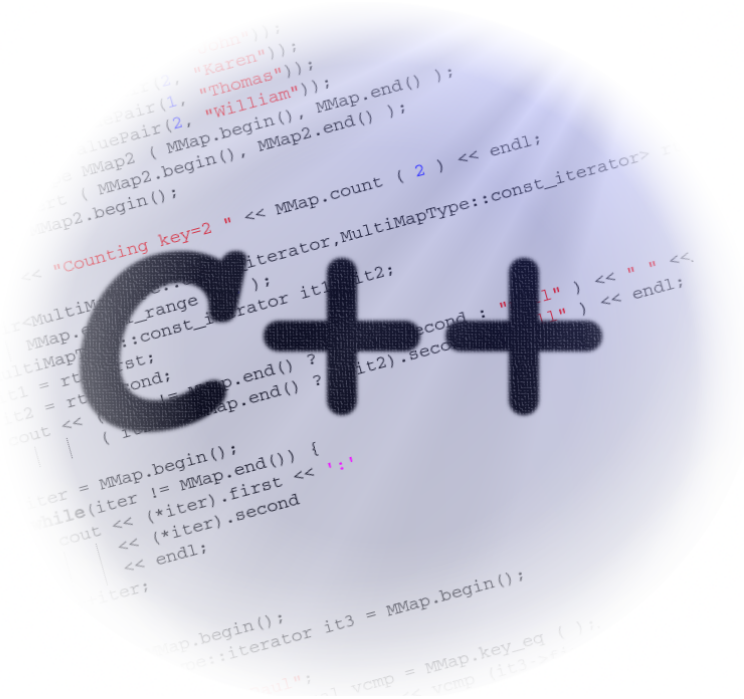
Języki Programowania

Prowadząca:
dr inż. Hanna Zbroszczyk

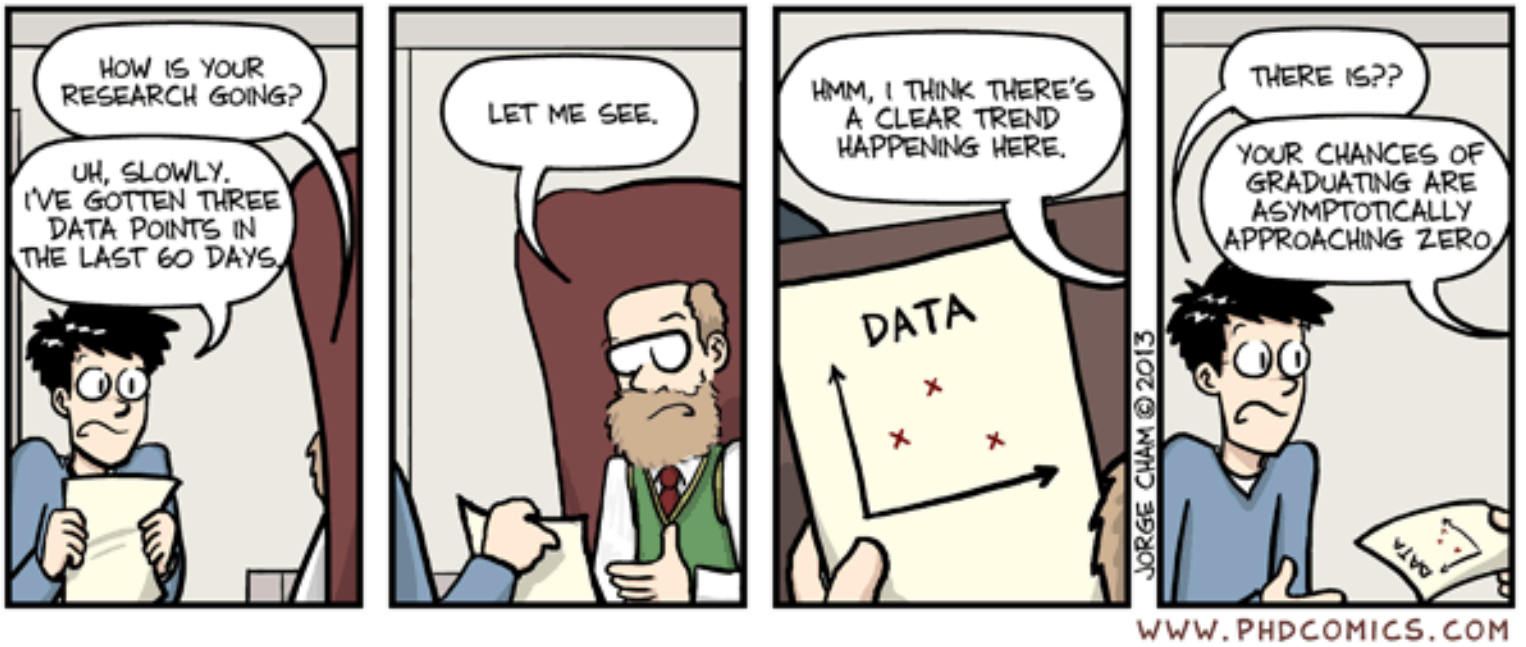
e-mail: *hanna.zbroszczyk@pw.edu.pl*
tel: +48 22 234 58 51

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)



SZABLONY FUNKCJI

Szablony funkcji - 1

Szablony funkcji - schemat funkcji (mechanizm) do wygenerowania zestawu funkcji działających na odmiennych typach argumentów.

```
int isGreater (int a, int b) {
	if (a>b) return a;
	else return b;
}

double isGreater (double a, double b) {
	if (a>b) return a;
	else return b;
}
unsigned int isGreater (unsigned int a, unsigned int b) {
	if (a>b) return a;
	else return b;
}
```

Szablony funkcji - 2

.. możliwe jest stworzenie szablonu funkcji, na podstawie **parametrów** którego wygenerują się odpowiednie wersje **funkcji z** odpowiednimi **argumentami..**

```
template <class T>
T isGreater (T a, T b) {
    if (a>b) return a;
    else return b;
}

int ix = 5, iy = 10;
unsigned int uix = 1, uiy= 2;
double dx = 5.5, dy = 3.14;

cout<<isGreater(ix,iy)<<endl; //10
cout<<isGreater(uix,uiy)<<endl; //2
cout<<isGreater(dx,dy)<<endl; //5.5
cout<<isGreater('A','B')<<endl; //B
cout<<isGreater(8, 10)<<endl; //10
cout<<isGreater(8, 12.5)<<endl; //ZLE
```

Szablony funkcji - 3

- Parametrem szablonu może być jedynie nazwa typu (musi wystąpić jako typ argumentu fukcji zdefiniowanej szablonem)
- Typ zwracanej wartości nie ma znaczenia (z jednego szablonu nie wygenerują się dwie funkcje różniące się jedynie typem zwracanej wartości)
- Nazwa szablonu musi być w zakresie globalnym (na zewnątrz innych funkcji, klas).
- Wywołanie funkcji szablonowej jest identyczne jak wywołanie zwykłej funkcji.

- Szablon funkcji może mieć więcej parametrów (jeśli funkcja szablonowa ma więcej niż jeden argument tego samego typu, to w liscie parametrów szablonu pojawia się tylko raz)

```
template <class T1, class T2>
T2 isGreater(T1 a, T2 b) {
      if (a>b) return a;
      else return b;
}
unsigned int uix = 1;
int ix  =5;
cout<<isGreater(uix, ix)<<endl; //5
```

Szablony funkcji - 4

- Za poprawność działania funkcji szablonowej odpowiada użytkownik!
- Fukncja szablonowa nie musi być uniwersalna, lecz dobra.
- Możliwe jest przeładowanie nazwy funkcji szablonowej, ale trzeba uważać, by nie generowały funkcji o takich samych argumentach (o takiej samej ich liczbie, kolejności oraz typie)

```
template <class T>
T isGreater(T a, T b) {
	if (a>b) return a;
	else return b;
}
template <class T1, class T2>
T2 isGreater(T1 a, T2 b) {
	if (a>b) return a;
	else return b;
}
cout<<isGreater(1, 10)<<endl; //i co teraz?
```

Szablony funkcji – przykład

```
template <class T>
void change(T &a, T &b) {
        T tmp;
        tmp = a;
        a = b;
        b = tmp;
}
..
float fx = 5.1, fy= 10.5;
int x = -8, y = 2;
char *sA = new char [100];
char *sB = new char [100];
strcpy(sA, “Anna Kowalska”);
strcpy(sB, “Pawel Nowak”);
change(fx, fy);
change(x, y);
change(sA, sB); //change(char* &a, char* &b)


change(ptrx, ptry);
cout<<”ptrx = ”<<setw(10)<<(*ptrx)<<”ptry =”<<setw(10)<<(*ptry)<<endl;
```

Szablony funkcji, a przydomki

- inline

```
template <class T>
inline T isGreater(T& a, T& b) {
        return (a > b) ? a : b;
}
```

Kompilator widząc że funkcja jest inline w miejsce jej Wywołania wpisuje jej kod w miejsce jej wystąpienia

- static

```
template <class T>
static T isGreater(T& a, T& b) {
        return (a > b) ? a : b;
}
```

W danym bloku programu fukcja / zmienna statyczna posiada dokładnie jedną instancję i istnieje przez cały czas działania programu.

- extern

```
template <class T>
extern T isGreater(T& a, T& b) {
        return (a > b) ? a : b;
}
```

Deklaracja nie jest deklaracją w sensie fizycznym, a jedynie odwołaniem do deklaracji znajdującej się w innej jednostce kompilacji.

Parametr szablonu, a rezultat funkcji

Typ rezultatu funkcji szablonowej nie jest brany pod uwagę w dopasowywaniu wywołań odpowiednich funkcji – parametr szablonu nie może posłużyć jedynie do określeśnia samego rezultatu funkcji.

```
template <class T1, class T2, class T3>
T3 funkcja (T1 a, T2 b); //ZLE

template <class T1, class T2, class T3>
T3 funkcja (T1 a, T2 b, T3); //DOBRZE
```

Obiekty statyczne w szablonie funkcji

Obiekty statyczne to takie, które istnieją przez cały czas działania programu.

```
template <class T>
T isGreater (T& a, T& b) {
      static int counter;
      counter ++;
      return (a > b) ? a : b;
}


change(-7, 20);
change(3.14, 20.5);
unsigned int x = 13, y = 55;
change(x, y);
```

Każda wersja funkcji szablonowej ma swoją zmienną statyczną.

Funkcje specjalizowane

Na podstawie funkcji szablonowej możliwe jest wygenerowanie kompletu funkcji o różnych argumentach. W przypadku, kiedy funkcja wygenerowana z szablonu nie nadaje się do użytku dla danego zestawu argumentów, należy stworzyć funkcję specjalizowaną.

```
template <class T>
T isGreater(T a, T, b) {
      return (a > b) ? a : b;
}
char* isGreater(char* a, char* b) {
      if (strlen(a) > strlen(b)) return a;
      else return b;
}
...
float x = 66.1, y = -8.6;
char name1[50] = {“Anna Kowalska”};
char name2[50] = {“Anna Nowak”};
cout<<isGreater(x, y)<<endl; //66.1
cout<<isGreater(name1, name2)<<endl; //Anna Kowalska
```

Rozpoznawanie typów obiektów

W celu rozpoznania typu obiektu (wbudowanego oraz stworzonego przez nas) należy posłużyć się klasą type_info

#include <typeinfo>

Składowe klasy:

Funkcja składowa: name()

operator==

operator!=

```
int a, b, c;
zesp z1, z2;
double *tmp;
cout<<typeid(a).name()<<endl;
cout<<typeid(z1).name()<<endl;
cout<<typeid(*tmp).name()<<endl;
if (typeid(a)==typeid(b)) cout<<”Te same typy”<<endl;
if (typeid(c)!=typeid(z2)) cout<<”Inne typy”<<endl;
```

Dopasowanie w przypadku funkcji szablonowych

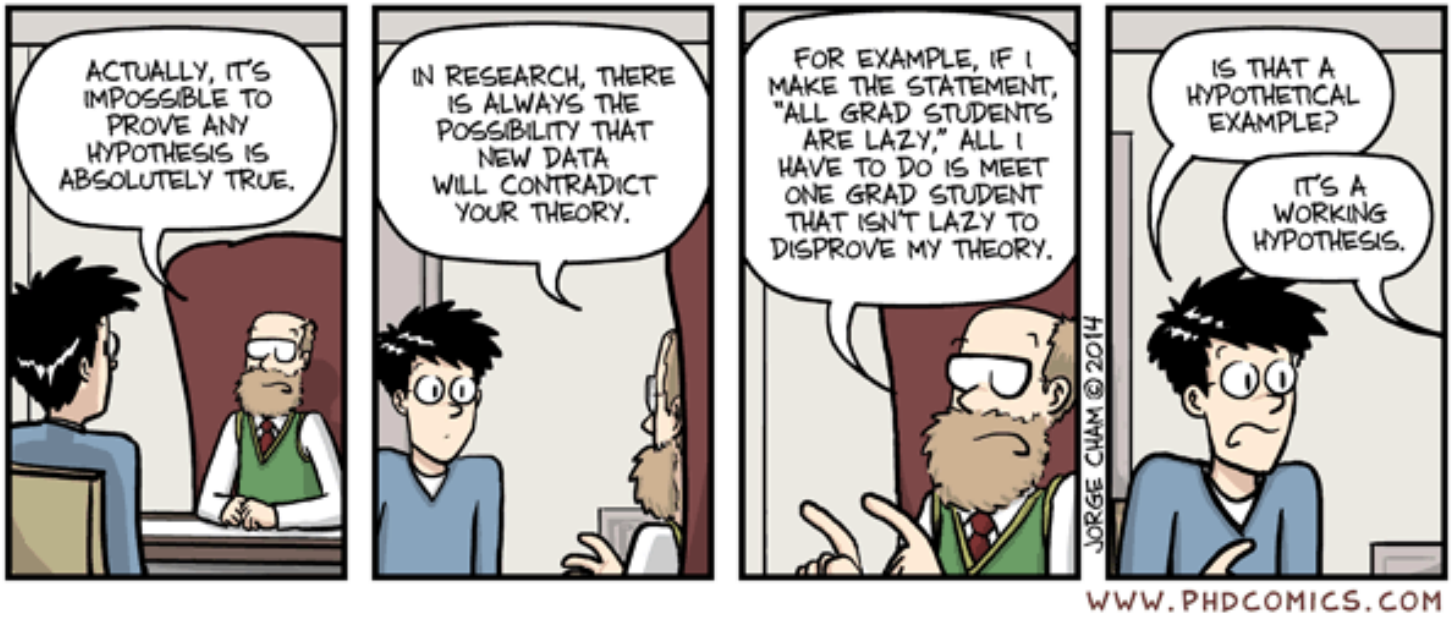
Etapy dopasowania:

(1) Szukanie dopasowania dokładnego (funkcja o tej samej nazwie, typie, ilości i kolejności argumentów)

(2) Szukanie szablonu, na podstawie którego wyprodukowana zostanie odpowiednia funkcja (musi zajść dopasowanie idealne, kompilator nie będzie dokonywał konwersji)

(3) Szukanie dopasowanie dokładnego, ale z:

- trywialną konwersją; float → double, short int → int
- użyciem konwersji zdefiniowanej przez użytkownika
- użyciem funkcji “z wielokropkiem”



SZABLONY KLAS

Szablon klas - przykład 1

Szablon klas – mechaniz do tworzenia na podstawie parametrów formalnych szablonu klas szablonowych.

```
#include <iostream>
#include <iomanip>
using namespace std;
template <class T> // ← T parametr formalny szablonu
class cmplx {
        T re, im;
        public:
        cmplx() {re = 0; im = 0;}
        cmplx(T a, T b) {re = a; im = b;}
        ~cmplx() {}
        cmplx<T> dodaj(cmplx<T> m1) { re += m1.re; im += m1.im; return *this; }
        void get() { cout<<setw(10)<<re<<"+i "<<setw(10)<<im<<endl; }
};
int main() {
        cmplx<int> z1(2,2); z1.get(); // ← int parametr aktualny szablonu
        cmplx<int> z2(3,4); z2.get();
        cmplx<int> z3(0,0); z3 = z1.dodaj(z2); z3.get();
        return 1;
}
```

Szablon klas - przykład 2 - 1

Są dwa sposoby umieszczania definicji funkcji składowych:

(1) wewnątrz definicji (ciała) klasy

(2) na zewnątrz definicji klasy. Funkcja składowa jest definiowana jak szablon funkcji, którego parametry są identyczne jak parametry szablonu klasy.

```
#include <iostream>
#include <iomanip>
using namespace std;
template <class T>
class cmplx {
        T re, im;
        public:
        cmplx();
        cmplx(T, T);
        ~cmplx();
        cmplx<T> dodaj(cmplx<T>);
        void put();
        void get();
};
```

```
template <class T>
cmplx<T>::cmplx(){
        re = 0; im = 0;
}
template <class T>
cmplx<T>::cmplx(T a, T b){
        re = a; im = b;
}
template <class T>
cmplx<T>::~cmplx(){
}
template <class T>
cmplx<T> cmplx<T>::dodaj(cmplx<T> m1) {
        re += m1.re;
        im += m1.im;
        return *this;
}
```

Szablon klas - przykład 2 - 2

```
template <class T>
void cmplx<T>::put() {
        cin>>re>>im;
}
template <class T>
void cmplx<T>::get() {
      cout<<setw(10)<<re<<"+i "<<setw(10)<<im<<endl;
}
int main() {
      cmplx<double> z1; z1.put(); z1.get();
      cmplx<double> z2; z2.put(); z2.get();
      cmplx<double> z3; z3 = z1.dodaj(z2); z3.get();
      cmplx<double> *z4 = new cmplx<double>(z3);
      z4->get();
      delete z4;
      return 1;
}
```

Szablon funkcji korzystający z obiektów klasy szablonowej - 1

```
template <class T>
class cmplx {
        T re, im;
        public:
        ...
        void getCmplx();
};
...
template <class T>
void cmplx<T>::getCmplx() {
        cout<<setw(5)<<re<<"+i"<<setw(5)<<im<<endl;
}
...
Int main() {
        cmplx<double> z1;
        ...
        z1.getCmplx();

}
```

Szablon funkcji korzystający z obiektów klasy szablonowej - 2a

```
#include <iostream>
#include <iomanip>
using namespace std;
template <class T>
class cmplx {
        T re, im;
        public:
        cmplx(T=0, T=0);
        ~cmplx() {}
        cmplx<T> operator+(cmplx<T> zz) {
                re+= zz.re; im+=zz.im; return *this;
        }
        friend ostream& operator<<(ostream &out, cmplx<T> &tmp) {
                out<<setw(10)<<tmp.re<<"+i"<<setw(10)<<tmp.im<<endl; return out;
        }
        friend istream& operator>>(istream &in, cmplx<T> &tmp) {
                in>>tmp.re>>tmp.im; return in;
        }
};
```

Szablon funkcji korzystający z obiektów klasy szablonowej - 2b

```
template <class T>
cmplx<T>::cmplx(T a, T b){
        re = a; im = b;
}
int main() {
        cmplx<double> z1;
        cin>>z1;
        cout<<z1;
        cmplx<double> z2;
        cin>>z2;
        cout<<z2;
        cmplx<double> z3;
        z3 = z1+z2;
        cout<<z3;
        return 1;
}
```

Obiekt klasy szablonowej składnikiem innego szablonu klasy

```
#include <iostream>
using namespace std;
template <class T>
class point {
        T x, y;
        public:
        point(T = 0, T = 0);
        ~point();
        ...
};
template <class T>
point<T>::point(T a, T b): x(a), y(b){
}
template <class T>
point<T>::~point(){ }
...
```

```
template <class A>
class line {
        point<A> a, b;
        public:
        line(point<A>, point<A>);
        ~line() {}
};
template <class A>
line<A>::line(point<A> aa, point<A> bb){
        a = aa; b = bb;
}
...
int main() {
        point<float> p1(1.0, 2.5);
        point<float> p2(3.7, -5.4);
        line<float> l(p1, p2);
        ...
        return 1;
}
```

Składniki statyczne w szablonie klas - 1

Każda klasa szablonowa (generowana z szablonu) ma swój własne zestaw składników statycznych.

```
#include <iostream>
#include <iomanip>
#define N 3
using namespace std;
template <class T>
class wektor {
        T mm[N];
        static int licznik;
        public:
        wektor();
        wektor(T[]);
        wektor(const wektor&);
        ~wektor();
        wektor& operator=(const wektor&);
        void wypiszWektor();
        static int wypiszLicznik();
};
```

```
template <class T>
wektor<T>::wektor(){
        for (int i=0; i<N; i++) mm[i] = 0;
        licznik++;
}
template <class T>
wektor<T>::wektor(T t[N]){
        for (int i=0; i<N; i++) mm[i] = t[i];
        licznik++;
}
template <class T>
wektor<T>::wektor(const wektor &tmp){
        for (int i=0; i<N; i++) mm[i] = tmp.mm[i];
        licznik++;
}
template <class T>
wektor<T>::~wektor() {
        licznik--;
}
```

Składniki statyczne w szablonie klas - 2

```
template <class T>
wektor<T>& wektor<T>::operator=(const wektor& tmp) {
        if (&tmp==this) return *this;
        for (int i=0; i<N; i++) mm[i] = tmp.mm[i];
        return *this;
}

template <class T>
void wektor<T>::wypiszWektor() {
        for (int i=0; i<N; i++) cout<<setw(10)<<mm[i];
        cout<<endl;
}

template <class T>
int wektor<T>::licznik =0;

template <class T>
int wektor<T>::wypiszLicznik() {
        return licznik;
}
```

Składniki statyczne w szablonie klas - 3

```
int main() {
	cout<<wektor<unsigned int>::wypiszLicznik()<<endl;
	//cout<<wektor<unsigned int>::licznik<<endl; //jesli licznik bylby publiczny
	unsigned int tab1[] = {1, 2, 5};
	wektor<unsigned int> m1(tab1);
	m1.wypiszWektor();

	cout<<wektor<float>::wypiszLicznik()<<endl;
	float tab2[] = {3.14, 2.71, 4.45};
	wektor<float> m2(tab2);
	m2.wypiszWektor();

	cout<<wektor<short>::wypiszLicznik()<<endl;
	wektor<short> m3;
	m3.wypiszWektor();

	cout<<wektor<unsigned int>::wypiszLicznik()<<endl;
	cout<<wektor<float>::wypiszLicznik()<<endl;
	cout<<wektor<short>::wypiszLicznik()<<endl;
	return 1;
}
```

Parametry szablonu klas - 1

Parametry szablonu klas są umieszczone w <..., ..., ...>

- parametrem szablonu funkcji może być nazwa typu T
- parametrem szablonu klas może być:
    - nazwa typu (1)
    - stałe wyrażenie będące:
        - wartością całkowitą (2)
        - adresem komórki w pamieci
        - adresem obiektu globalnego (znanym w danym miejscu lub dostępnemu dzięki deklaracji extern)
        - adresem funkcji globalnej
        - adresem składnika statycznego klasy

(1) Parametr będący nazwą typu

template <class Type>

Nie ma potrzeby definiować:

template <class Type, class *Type, class Type[]>

Parametry szablonu klas - 2

(2) Parametr będący wartością całkowitą

```
template <class Type, int N>
class wektor {
      Type mm[N];
      public:
      ...
      wektor(Type[]);
      ...
};
double t[] = {-5.5, 4.3, 6.4, 9.2, -8.8}
wektor<double, 5> w(t); //mozliwe tez uzycie innych konstruktorow
```

Tworząc klasę z szablonu należy pamiętać, aby parametry wartości całkowitej z szablonu klasy, jak i z jej odpowiedniej realizacji były zgodne!

sizeof, operacje arytmetyczne to też stała:

```
wektor<double, 2+sizeof(char)> w(t);
```

Parametry szablonu klas - 3

Nie może być parametrem szablonu klas:

- stała dosłowna string
- adres elementu tablicy
- adres niestatycznego (zwykłego) składnika klasy
- typ (klasa) zdefiniowany lokalnie (np. def. klasy zagnieżdżona w funkcji)
- stała dosłowna (kiedy szablon oczekuje parametru będącego obiektem – wymagałoby to stworzenia obiektu chwilowego dla stałej)

Specjalizowana wersja klasy szablonowej – przykład 1 - 1

Jeśli klasa wygenerowana z szablonu nie spełnia oczekiwań można stworzyć jej specjalizowaną wersję.

```
#include <iostream>
#include <string>
#include <iomanip>
#define N 3
using namespace std;

template <class T>
class Tablica {
	T mm[N];
	public:
	Tablica();
	Tablica(T[]);
	~Tablica() {}
	void wypiszTablica();
	void sortujTablica();
};
```

```
template <class T>
Tablica<T>::Tablica(){
	for (int i=0; i<N; i++) mm[i] = 0;
}

template <class T>
Tablica<T>::Tablica(T t[N]){
	for (int i=0; i<N; i++) mm[i] = t[i];
}

template <class T>
void Tablica<T>::wypiszTablica() {
	for (int i=0; i<N; i++) cout<<setw(10)<<mm[i];
	cout<<endl;
}
```

Specjalizowana wersja klasy szablonowej – przykład 1 - 2

```
template <class T>
void Tablica<T>::sortujTablica() {
        T tmp;
        for (int i=0; i<N; i++) {
                for (int j=0; j<N-1; j++) {
                        if (mm[j+1]<mm[j]) {
                                tmp = mm[j+1];
                                mm[j+1] = mm[j];
                                mm[j] = tmp;
                        }
                }
        }
}
```

```
template <> //kompilator gcc 4.4.3 domaga sie tej skladni
class Tablica<string> {
        string mm[N];
        public:
        Tablica();
        Tablica(string[]);
        ~Tablica() {}
        void wypiszTablica();
        void sortujTablica();
};

Tablica<string>::Tablica(){
        for (int i=0; i<N; i++) mm[i] = "";
}

Tablica<string>::Tablica(string t[N]){
        for (int i=0; i<N; i++) mm[i] = t[i];
}
```

Specjalizowana wersja klasy szablonowej – przykład 1 - 5

```
void Tablica<string>::wypiszTablica() {
	for (int i=0; i<N; i++) cout<<setw(10)<<mm[i];
	cout<<endl;
}

void Tablica<string>::sortujTablica() {
	string tmp;
	for (int i=0; i<N; i++) {
		for (int j=0; j<N-1; j++) {
			if (mm[j+1].size() < mm[j].size()) {
				tmp = mm[j+1];
				mm[j+1] = mm[j];
				mm[j] = tmp;
			}
		}
	}
}
```

Specjalizowana wersja klasy szablonowej – przykład 1 - 6

```
int main() {
	int tab1[] = {5, 1, 2};
	Tablica<int> m1(tab1);
	m1.wypiszTablica(); //5 1 2
	m1.sortujTablica();
	m1.wypiszTablica(); //1 2 5
	char tab2[] = {'E', 'w', 'a',};
	Tablica<char> m2(tab2);
	m2.wypiszTablica(); // E w a
	m2.sortujTablica();
	m2.wypiszTablica(); //E a w

	string tab3[] = {"Ala", "ma", "kota"};
	Tablica<string> m3(tab3);
	m3.wypiszTablica(); //Ala ma kota
	m3.sortujTablica();
	m3.wypiszTablica(); //ma Ala kota
	//m3.wypiszTablica(); //Ala kota ma – bez specjalizacji klasy szablonowej
	return 1;
}
```

Specjalizowana wersja klasy szablonowej – przykład 2 - 1

```
template <class T, int N>
class Tablica {
        T mm[N];
        public:
        Tablica();
        Tablica(T[]);
        ~Tablica() {}
        void wypiszTablica();
        void sortujTablica();
};
...
template <class T, int N>
Tablica<T,N>::Tablica(T t[N]){
        for (int i=0; i<N; i++) mm[i] = t[i];
}
...
```

```
template <int N>
class Tablica<string, N> {
        string mm[N];
        public:
        Tablica();
        Tablica(string[]);
        ~Tablica() {}
        void wypiszTablica();
        void sortujTablica();
};
...
template <int N>
Tablica<string,N>::Tablica(string t[N]){
        for (int i=0; i<N; i++) mm[i] = t[i];
}
```

Specjalizowana wersja klasy szablonowej – przykład 2 - 2

```
int main() {
	int tab1[] = {5, 1, 2};
	Tablica<int, 3> m1(tab1);
	m1.wypiszTablica(); //5 1 2
	m1.sortujTablica();
	m1.wypiszTablica(); //1 2 5
	char tab2[] = {'P', 'o', 'r', 't', 'o', 's'};
	Tablica<char, 6> m2(tab2);
	m2.wypiszTablica(); //P o r t o s
	m2.sortujTablica();
	m2.wypiszTablica(); //P o o r s t

	string tab3[] = {"Ala", "ma", "kota", "Mruczka"};
	Tablica<string, 4> m3(tab3);
	m3.wypiszTablica(); // Ala ma kota Mruczka
	m3.sortujTablica();
	m3.wypiszTablica(); //ma Ala kota Mruczka
	return 1;
}
```

Specjalizowana funkcja składowa

Nie w każdym przypadku konieczne jest tworzenie specjalizowanej wersji klasy, czasem wystarczy napisać specjalizowaną funkcję składową.

```
#define N 3
template <class T>
class Tablica {...} //bez zmian


template <> //kompilator gcc 4.4.3 domaga sie tej skladni
void Tablica<string>::sortujTablica() {
        string tmp;
        for (int i=0; i<N; i++) {
                for (int j=0; j<N-1; j++) {
                        if (mm[j+1].size() < mm[j].size()) {
                                tmp = mm[j+1];
                                mm[j+1] = mm[j];
                                mm[j] = tmp;
                        }
                }
        }
}
```

Przyjaźń, a szablony klas - 1

W wypadku szablonów klas są warianty przyjaźni (z funkcją lub z inną klasą):

(1) jeden wspólny przyjaciel dla każdej klasy szablonowej (funkcja, klasa)

(2) każda klasa szablonowa ma swojego przyjaciela

```
#include <iostream>
#include <iomanip>
using namespace std;
class wektory;
template <int T>
class wektor {
        double w[T];
        public:
        wektor();
        wektor(double[T]);
        ~wektor() {}
        friend void wypisz(wektor<3>);
        friend class wektory;
};
```

```
template <int T>
wektor<T>::wektor() {
        for (int i=0; i<T; i++) w[i] = 0.0;
}
template <int T>
wektor<T>::wektor(double tab[T]) {
        for (int i=0; i<T; i++) w[i] = tab[i];
}
void wypisz(wektor<3> w1) {
        for (int i =0; i<3; i++) cout<<setw(5)<<w1.w[i];
        cout<<endl;
}
class wektory {
        wektor<3> w1, w2;
        public:
        wektory(wektor<3>, wektor<3>);
        ~wektory() {}
};
```

Przyjaźń, a szablony klas - 2

```
wektory::wektory(wektor<3> ww1, wektor<3> ww2) {
        w1 = ww1; w2 = ww2;
}
int main() {
        double tab1[]={1.5, 5.3, -7.4};
        double tab2[]={-8.8, 4.7, 2.0};
        wektory ww(wektor<3>(tab1),wektor<3>(tab2));
        wypisz(wektor<3>(tab1));
        wypisz(wektor<3>(tab2));
        return 1;
}
```

Jest jeszcze **dziedziczenie szablonów,**

Informacje w literaturze (np. Jerzy Grębosz, Pasja C++, tom I, str. 218)

OPERACJE WEJŚCIA / WYJŚCIA

Operacje wejścia / wyjścia

…jak program porozumiewa się z użytkownikiem, pamięcią zewnętrzną itd..

Operacje wejścia / wyjścia nie są częścią języka C++ (są dostarczane przez producenta kompilatora)

- Biblioteka stdio (standard input / output), istnieje w ANSI C.

  W C++ istnieje także zapewniając m.in. kompatybilność języka C oraz C++

- Biblioteka iostream (input / output stream), zalecana w C++.

  Jej założenia weszły w skład obecnego standardu ISO języka C++.

Zastosowanie:

(1) wprowadzanie i wyprowadzanie informacji z urządzeń standardowych (klawiatura, ekran)

(2) operacje na plikach znajdujących się na zewnętrznych nośnikach

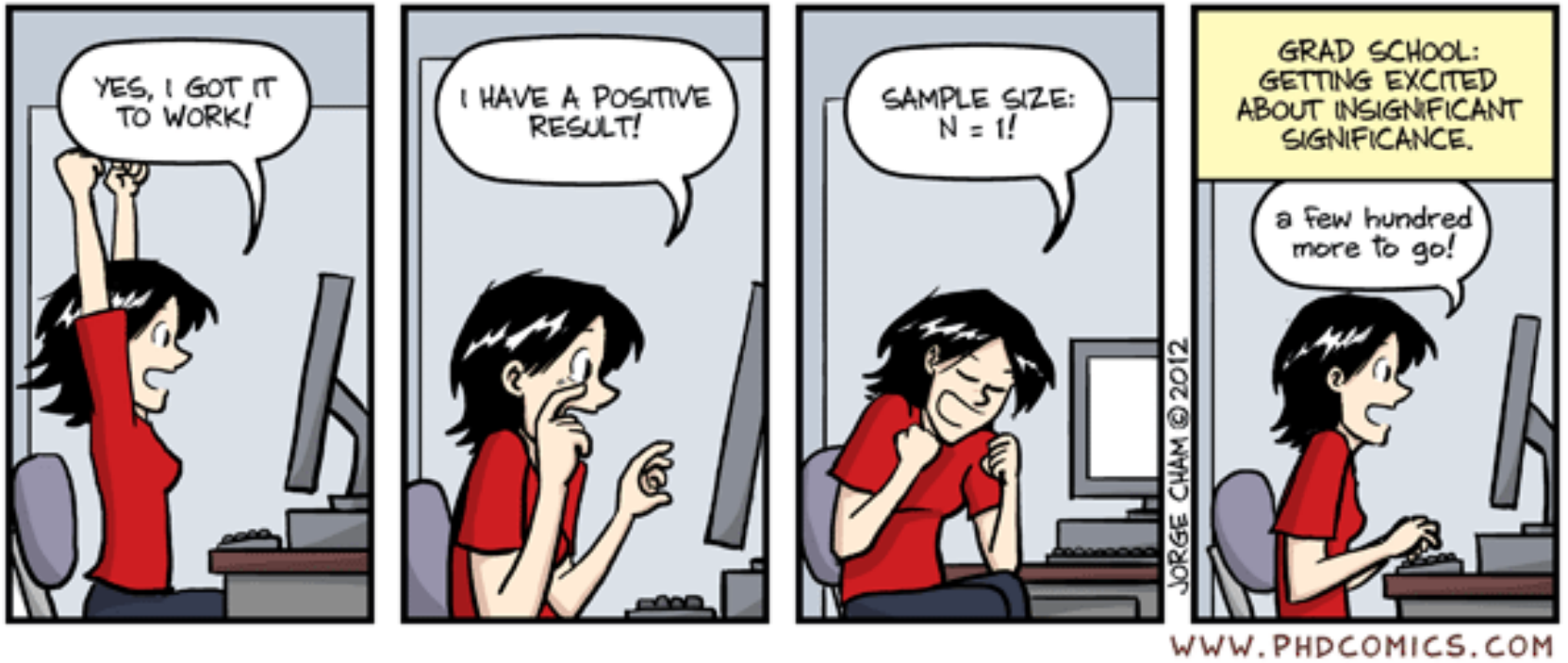
(3) wprowadzanie / wyprowadzanie informacji z obszarów pamięci (string).

W celu korzystania z biblioteki iostream, należy włączyć plik nagłówkowy:

```
#include <iostream>
```

Jeśli dodatkowo mają być przeprowadzane operacje na plikach I / lub obiektach klasy string:

```
#include <fstream>
#include <sstream>
```



OPERACJE NA PLIKACH

Operacje na plikach - 1

Klasy umożliwiające pracę z plikami:

ofstream – (ang. output file stream) - zapis

ifstream – (ang. input file stream) – czytanie

fstream – (ang. file stream) - oba

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fileIn;
ofstream fileOut;
int var;
fileIn.open("data.dat");
fileOut.open("data2.dat");
fileIn>>var;
fileOut<<var;
fileOut.close();
fileIn.close();
```

Pliki tekstowe to strumienie,

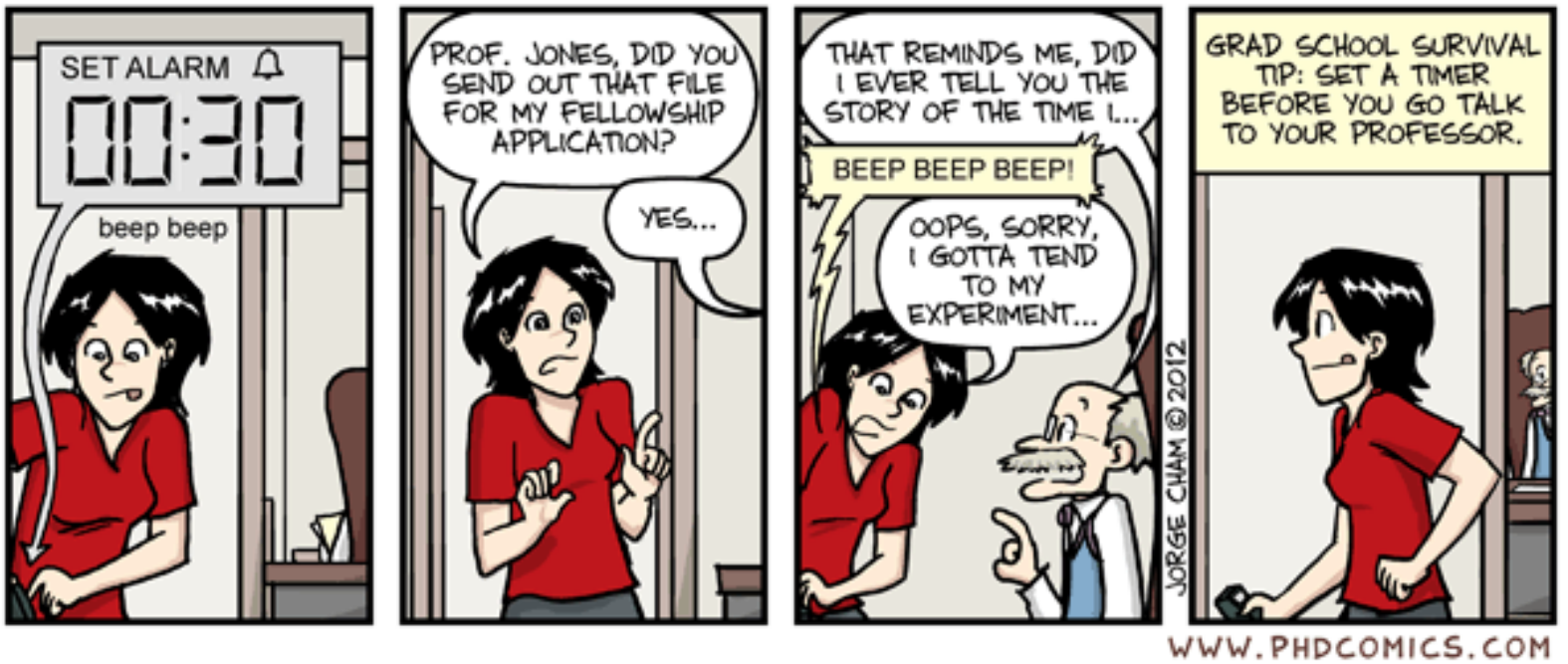
czytanie z nich oraz

zapis do nich jest

taki sam jak w przypadku

Standardowych strumieni

wejscia (*cin*) i wyjścia (*cout*)



FORMATOWANIE

Manipulatory bezargumentowe - 1

Manipulatory – specjalne wartości, zdefiniowane w standardowej przestrzeni nazw, wstawiane lub wyjmowane ze strumienia w celu zmiany sposobu formatowania.

Manipulatory bezargumentowe - **domniemane**

(1) std::boolaplha, **std::noboolalpha** – sterują wczytywaniem / wypisywaniem wyrażen bool

```
bool wyr = true;
cout<<”wyrazenie: ”<<boolalpha<<endl;
cout<<”wyrazenie: “<<noboolalpha<<endl;
```

(2) std::hex, **std::dec**, **std::oct** – sterują konwersją liczb

```
int i = 25;
int j;
cout<<oct<<i<<endl;
cin>>dec>>j;
```

Manipulatory bezargumentowe - 2

(3) **std::flush** – wypisanie zawartości bufora

```
cout<<wyr<<flush;
```

(4) **std::endl** – wstawienie nowej linii '\n' oraz wywołanie funkcji flush

```
cout<<j<<endl;
```

(5) **std::ends** – wstawienie do strumienia znaku null

std::skipws, std::noskipws – ignorowanie białych znaków

std::noskipws – wczytywanie białych znaków

```
char wyraz[100];
cin>>noskipws>>wyraz;
```

(6) **std::ws** – usuwanie z bufora białych znaków

Manipulatory bezargumentowe - 3

(7) std::showpoint, **std::noshowpoint** – wyświetlenie / nie wyświetlenie kropki dziesiętnej

```
cout<<showpoint<<15.0<<endl; //15.0
cout<<noshowpoint<<15.0<<endl; //15
```

(8) std::showpos, **std::noshowpos** – dodanie / nie dodanie znaku (+) przy wypisaniu liczby dodatniej

```
cout<<showpos<<15.0<<endl; //+15
cout<<noshowpos<<15.0<<endl; //15
```

(9) std::unitbuf, **std::nounitbuf** – buforowanie / nie buforowanie strumienia

– nie buforowanie strumienia

(10) **std::showbase**, std::noshowbase– ustawienie / nie ustawienie przedrostka 0x przy wypisie liczb szesnastkowych oraz 0 w zapisie liczb ósemkowych

Manipulatory bezargumentowe - 4

(11) std::uppercase, **std::nouppercase** – przy wypisie liczb (w odpowiedniej notacji) są duże / małe litery

```
int war = 5342;
cout<<hex<<war<<endl; //0x14de
cout<<showbase<<war<<endl; //14de
cout<<uppercase<<war<<endl; //0X14DE
```

(12) std::fixed, std::scientific – wypisanie liczby w notacji dziesiętnej / naukowej

– wypisanie liczby w notacji naukowej (wykładniczej)

```
double pi = 3.14
cout<<scientific<<pi<<endl; // 3.14e+000
cout<<fixed<<pi<<endl; //3.14
```

(13) std::left, **std::right, std::internal** – ustawienie pola justowania “lewego” / “prawego”, “wewnętrznego”

Manipulatory parametryzowane - 1

Aby używać standardowych manipulatorów: #include <iomanip>

(1) std::setw(int) – ustawienie szerokości w ilości znaków wypisywanego wyrażenia

```
int war = 5342;
cout<<"*"<<setw(10)<<war<<"*"<<endl; //*      5342*
```

(2) std::setfill(Ch) – dopełnienie znaków (char lub wchar_t) z wyrażenia zabierającego pewną liczbę znaków

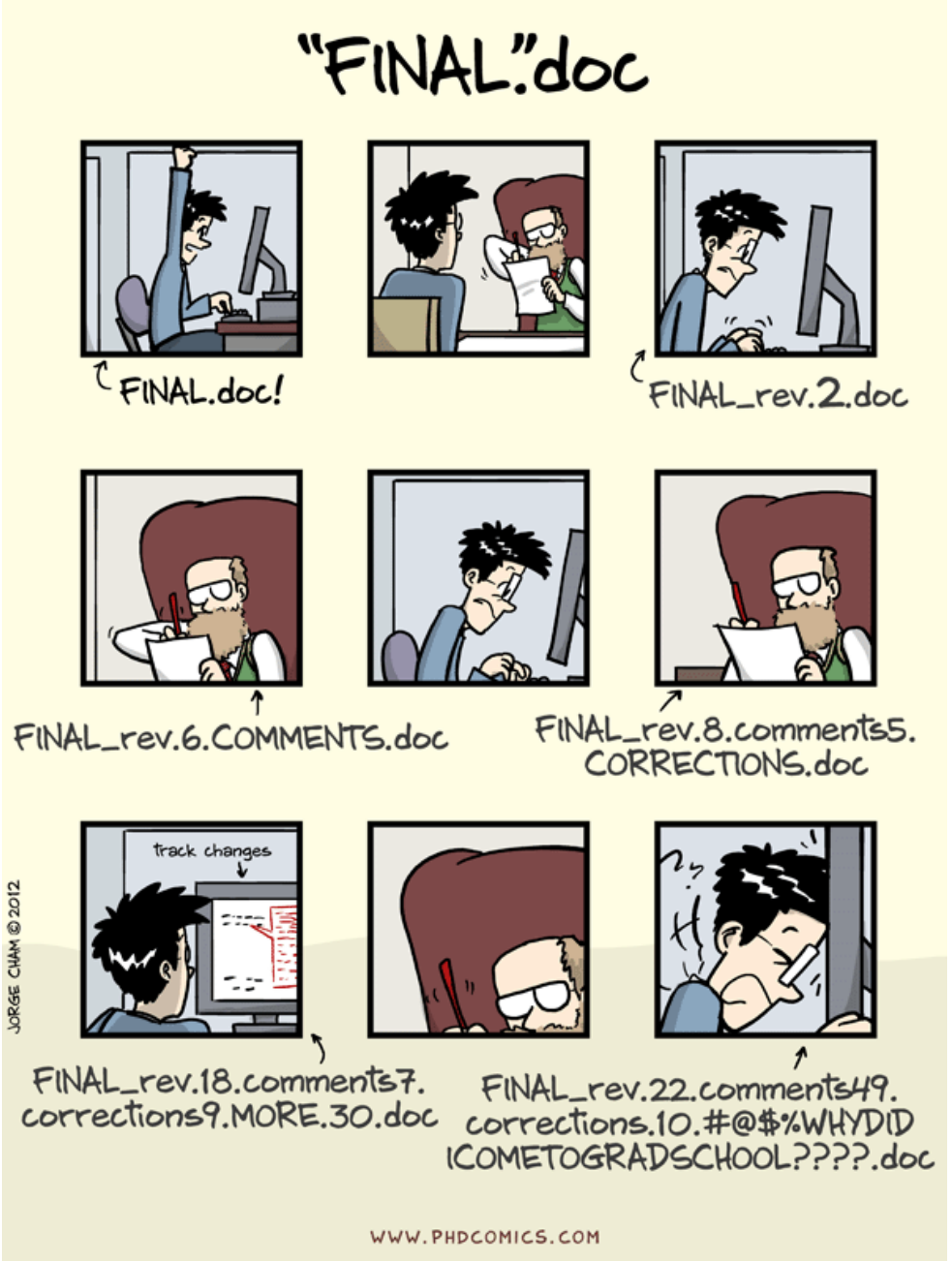
```
int war = 5342;
cout<<setfill('*')<<setw(10)<<war<<endl; //******5342
```

(3) std::setprecision(int) – ustawienie dokładności wypisywanej liczby zmiennoprzecinkowej

```
double pi = 3.14
cout<<setprecision(8)<<pi<<endl; // 3.14000000
```

(4) std::setbase – ustawienie podstawy konwersji liczb

```
cout<<dec<<endl;
cout<<setprecision(10)<<endl;
```



KONIEC WYKŁADU 9

Nieobowiązkowe zadania do wykonania

1. Napisać / zaprojektować /napisać szablon klasy do operacji na tablicach dwuwymiarowych, gdzie Szablonowe będę: typy wartości przechowywanych w tablicy oraz wymiary: N x M. Dodać Funkcjonalność prostych operacji macierzach: dodawanie, odejmowanie, mnożenie, mnożenie przez Wartość.

Szablony funkcji – przykład 1

```
template <class T>
void change(T &a, T &b) {
      T tmp = a;
      a = b;
      b = tmp;
}

int x = 5, y= 10;
cout<<”x = ”<<setw(10)<<x<<”y =”<<setw(10)<<y<<endl; // 5          10
change(x, y);
cout<<”x = ”<<setw(10)<<x<<”y =”<<setw(10)<<y<<endl;// 10         5
```

Szablony funkcji – przykład 2

```
void change(int* &a, int* &b) {
        int* tmp;
        tmp = a;
        a = b;
        b = tmp;
}
..
int x = 5, y= 10;
int *ptrx, *ptry;
ptrx = &x;
ptry = &b;
cout<<”ptrx = ”<<setw(10)<<(*ptrx)<<”ptry =”<<setw(10)<<(*ptry)<<endl;
change(ptrx, ptry);
cout<<”ptrx = ”<<setw(10)<<(*ptrx)<<”ptry =”<<setw(10)<<(*ptry)<<endl;
```