Programowanie obiektowe C++
Programowanie zorientowane obiektowo

Wykład 5

Witold Dyrka
witold.dyrka@pwr.wroc.pl

5/11/2012

Prawa autorskie itp.

Niektóre slajdy do tego wykładu powstało w oparciu o:

- *książkę pt.* Programowanie *i slajdy Bjarne Stroustrupa do kursu* Foundations of Engineering II (C++) *prowadzonego w Texas A&M University* http://www.stroustrup.com/Programming

Dziękuję dr inż. lek. med. Marcinowi Masalskiemu za udostępnienie materiałów do wykładu w roku ak. 2010/11

Program wykładów

1. Wprowadzenie. Obsługa błędów. Klasy (8/10)
2. Abstrakcja i enkapsulacja: klasy i struktury. (15/10)
3. Polimorfizm: przeładowanie funkcji i operatorów. Konwersje (22/10)
4. Zarządzanie pamięcią: konstruktor, destruktor, przypisanie (29/10)
5. **Dziedziczenie. Polimorfizm dynamiczny (5/11)**
6. Programowanie uogólnione: szablony (12/11)
7. Kolokwium zaliczeniowe (19/11)

Materiały

Literatura

- Bjarne Stroustrup. *Programowanie: Teoria i praktyka z wykorzystaniem C*++. Helion (2010)
- Jerzy Grębosz. *Symfonia C*++ *standard.* Edition 2000 (2008)
- Dowolny podręcznik programowania zorientowanego obiektowo w języku C++ w standardzie ISO 98

Środowisko programistyczne

- Microsoft Visual C++ (rekomendowane)
- Dowolne środowisko korzystające z GCC

Plan na dziś

- Dziedziczenie
  - Relacja IS_A (coś jest rodzajem czegoś)
  - Klasa bazowa i klasa pochodna
  - Zasady dziedziczenia
    - modyfikator **protected**
  - Funkcje wirtualne
  - Klasa abstrakcyjna
    - Funkcje czysto wirtualne

Wielomian

- Fragment sensownej implementacji wielomianu – to już umiemy

```
class Wielomian {

    vector<double> wspolczynniki;

    void usunZera();                // usuwanie zerowego wsp. przy najwyzszym stopniu

public:

    Wielomian(): wspolczynniki(vector<double>(1,0)) { };        // wielomian stopnia 0-ego o wsp. 0

    Wielomian(const vector<double> &v) :wspolczynniki(v) {      // konstrukcja na podst. listy wsp.
        if (v.size()==0) throw runtime_error("Zly wektor wspolczynnikow!");
        usunZera();
    };

    double wsp(int ktory) const;                                // zwraca wspolczynnik ktorego stopnia

    int stopien() const { return wspolczynniki.size()-1; };     // zwraca stopien wielomianu

    friend const Wielomian dodaj(const Wielomian& w1, const Wielomian& w2);
    friend const Wielomian odejmij(const Wielomian& w1, const Wielomian& w2);
};
```

Problem 1: Wielomiany szczególne

```
class Wielomian {
        vector<double> wspolczynniki;
        void usunZera();
public:
        Wielomian(): wspolczynniki(vector<double>(1,0)) { };
        Wielomian(const vector<double> &v);
        double wsp(int ktory) const;
        int stopien() const { return wspolczynniki.size()-1; };
        friend const Wielomian dodaj(const Wielomian& w1, const Wielomian& w2);
        friend const Wielomian odejmij(const Wielomian& w1, const Wielomian& w2);
};
```

- Czasami pracujemy z wielomianem szczególnego typu

$$B_0(x) = 1$$
$$B_1(x) = x - 1/2$$
$$B_2(x) = x^2 - x + 1/6$$
$$B_3(x) = x^3 - \frac{3}{2}x^2 + \frac{1}{2}x$$
$$B_4(x) = x^4 - 2x^3 + x^2 - \frac{1}{30}$$
$$B_5(x) = x^5 - \frac{5}{2}x^4 + \frac{5}{3}x^3 - \frac{1}{6}x$$
$$B_6(x) = x^6 - 3x^5 + \frac{5}{2}x^4 - \frac{1}{2}x^2 + \frac{1}{42}.$$

- np. wielomian Bernoulliego

źródło: Wikipedia

Named polynomials and polynomial sequences

- Additive polynomials
- Appell sequence
- Askey–Wilson polynomials
- Bell polynomials
- Bernoulli polynomials
- Bessel polynomials
- Binomial type
- Caloric polynomial
- Charlier polynomials
- Chebyshev polynomials
- Ehrhart polynomial
- Exponential polynomials
- Favard's theorem

źródło: Wikipedia

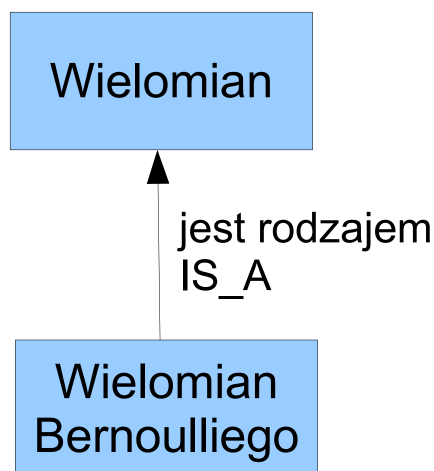
Wielomian Bernoulliego – idea (1)

- Wielomiany Bernoulliego definiujemy wzorem:

  b_k – liczby Bernoulliego

$$B_n(x) = \sum_{k=0}^{n} \binom{n}{n-k} b_k x^{n-k},$$

źródło: Wikipedia



- Wielomian Bernoulliego:
  - dziedziczy wszystkie cechy wielomianu ogólnego, np.
    - tablicę współczynników
    - sposób dodawania, odejmowania itp.
  - charakteryzuje się specjalnym przepisem na jego tworzenie

Wielomian Bernoulliego – idea (2)

```
class Wielomian {
protected:
        vector<double> wspolczynniki;
        // ...
public:
        Wielomian(): wspolczynniki(vector<double>(1,0)) { };
        // …
        friend const Wielomian dodaj(const Wielomian& w1, const Wielomian& w2);
        // ...
};

class WielomianBernoulli
        : public Wielomian
{
public:
        WielomianBernoulli(vector<double> beta);
};


// …
WielomianBernoulli wB(beta);
Wielomian w = dodaj(wB,wB);
wypisz(wB);
wB = dodaj(wB,wB)                    // błąd: suma wielomianów Bernoulliego
                                     // nie jest wielomianem Bernoulliego
```

KLASA BAZOWA

KLASA POCHODNA

is_a

- Wielomian Bernoulliego:
  - dziedziczy wszystkie cechy wielomianu ogólnego, np.
    - tablicę współczynników
    - sposób dodawania
  - charakteryzuje się specjalnym przepisem na jego tworzenie:

$$B_n(x) = \sum_{k=0}^{n} \binom{n}{n-k} b_k x^{n-k},$$

Wielomian Bernoulliego – implementacja

```
const double LICZBA_PI = 3.143159265;
const double LICZBA_E = 2.71828183;

int silnia (int n) { int wynik=1; for (int i=1; i<=wynik; ++i) wynik*=i;return wynik; }        // silnia
int dwumian(int n, int k) {   return silnia(n) / (silnia(k) * silnia(n-k)); }                  // dwumian Newtona

class WielomianBernoulli : public Wielomian {     // Wielomian Bernoulli dziedziczy interfejs po Wielomianie
        void init(const vector<double>& beta)     // inicjuje wielomian Bernoulliego liczbami Bernoulliego
public:
        WielomianBernoulli(const vector<double>& beta) { init(beta); }     // konstruktor – jeśli podamy liczby B.
        WielomianBernoulli(int n);                                        // konstruktor aproksymujący liczby Bernoulliego
};

void WielomianBernoulli::init(const vector<double>& beta)
                                          {       int n = beta.size()-1;
                                                  wspolczynniki.resize(n+1);
                                                  for (int k = 0; k <= n; k++)
                                                          wspolczynniki[n-k] = dwumian(n,n-k)*beta[k];
                                                  usunZera();                                    }
```

$$B_n(x) = \sum_{k=0}^{n} \binom{n}{n-k} b_k x^{n-k},$$

```
WielomianBernoulli::WielomianBernoulli(int n) {          // konstruktor aproksymujący
                vector<double> beta(n+1,0);
                beta[0] = 1.0; beta[1] = -0.5; beta[2] = 1.0/6; beta[3] = 0;       // pierwsze 4 liczby podajemy dokładnie
                for (int k = 2; k <= n/2; k++) beta[2*k] =
                        pow(float(-1),k-1)*4*sqrt(LICZBA_PI*k)*pow(double(k)/(LICZBA_PI*LICZBA_E),2*k);
                init(beta);
}
```

$$B_{2n} \sim (-1)^{n-1}\, 4\sqrt{\pi n} \left(\frac{n}{\pi e}\right)^{2n}$$

źródło: Wolfram MathWorld

Brak dostępu do danych i funkcji prywatnych klasy bazowej

```
class Wielomian {
private:
	vector<double> wspolczynniki;
	void usunZera();
public:
	Wielomian(): wspolczynniki(vector<double>(1,0)) { };
	//...
};

void WielomianBernoulli::init(const vector<double>& beta)  {
	int n = beta.size()-1;
	wspolczynniki.resize(n+1);
	for (int k = 0; k <= n; k++)
		wspolczynniki[n-k] = dwumian(n,n-k)*beta[k];
	usunZera();
}
```

- Klasa pochodna dziedziczy składowe prywatne klasy bazowej,
  - ale nie może się do nich bezpośrednio odwołać!

klasa bazowa Wielomian - zmienne chronione (*protected*)

```
class Wielomian {
protected:
        vector<double> wspolczynniki;
        void usunZera();
public:
        Wielomian(): wspolczynniki(vector<double>(1,0)) { };
        //...
};

void WielomianBernoulli::init(const vector<double>& beta)  {
        int n = beta.size()-1;
        wspolczynniki.resize(n+1);
        for (int k = 0; k <= n; k++)
                wspolczynniki[n-k] = dwumian(n,n-k)*beta[k];
        usunZera();
}
```

- Klasa pochodna dziedziczy składowe prywatne klasy bazowej,
  - ale nie może się do nich bezpośrednio odwołać
- Wprowadzono modyfikator **protected**
  - dostęp do składowych chronionych mają tylko klasy pochodne

Podstawowe zasady dziedziczenia

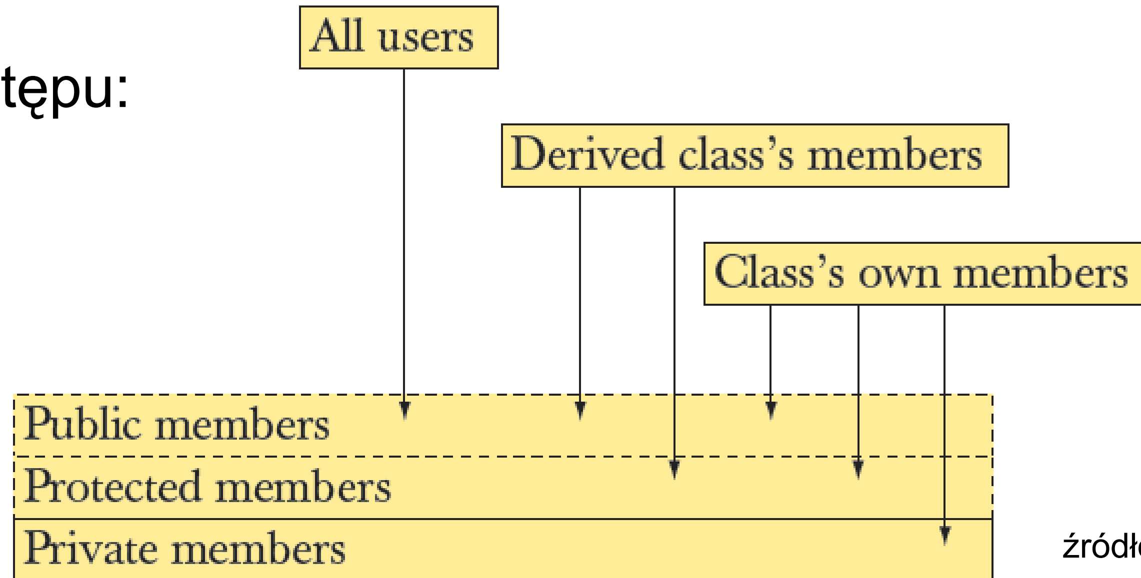
- Dziedziczy się zmienne i funkcje składowe
- Nie dziedziczy się konstruktorów, operatora= i destruktora
  - są generowane automatycznie z wykorzystaniem konstruktorów, operatora= i destruktora klasy bazowej
  - oczywiście mogą być zdefiniowane przez użytkownika
- Nazwy w klasie pochodnej
  - zasłaniają nazwy w klasie bazowej

```
class A {
public:
        void czesc() { cout << "Czesc tu A"; }
};
class B : A {
public:
        void czesc() { cout << "Czesc tu B"; }
};

B b; b.czesc();                // Czesc tu B
```

Podstawowe zasady dziedziczenia (2)

- Zasady dostępu:
  private
  protected
  public



źródło: B.Stroustrup

- Domyślnie **class** dziedziczy składowe chronione i publiczne klasy bazowej jako prywatne:

  class A : B { }; jest równoważne **class A : private B { };**

- Domyślnie **struct** dziedziczy składowe chronione jako chronione, a publiczne jako publiczne:

  struct A : B { }; jest równoważne **struct A : public B { };**

- Można jedynie zawężać dostęp (składowe prywatne nigdy nie będą dziedziczone jako chronione, a chronione jako publiczne)

Problem 2: Lista publikacji

- Każdy naukowiec (i nie tylko) publikuje...
  - artykuły:

W.Dyrka, J.-C.Nebel. A Stochastic Context Free Grammar based Framework for Analysis of Protein Sequences. BMC Bioinformatics 2009, 10:323

  - rozdziały w książkach:

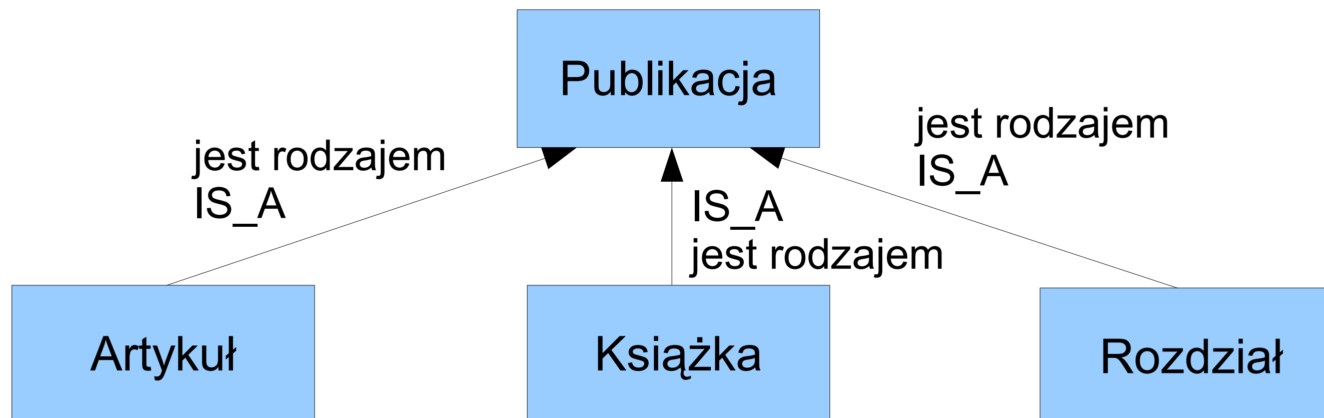
Konopka B , Dyrka W , Nebel J.C, Kotulska M , Accuracy in predicting secondary structure of ionic channels [W:] Challenges in Computational Collective Intelligence, Ed. N.T. Nguyen, R.Katarzyniak, A. Janiak, SCI 244, pp.315-326, Springer, Berlin-Heidelberg, 2009

  - książki:

B.Stroustrup, Programowanie. Teoria i praktyka z wykorzystaniem C++. Helion, Gliwice 2010.

Publikacje

- Publikacja to pojęcie abstrakcyjne
  - konkretna publikacja zawsze jest artykułem, książką, rozdziałem itp.



Publikacje

- Wszystkie publikacje mają pewne elementy opisu wspólne
  - mają autora, tytuł i rok wydania
  - chcemy je wyświetlać
    - w sposób pełny lub skrócony
- Inne cechy opisu zależą od rodzaju publikacji
  - Artykuł: czasopismo, numer, strony
  - Książka: wydawca, miejsce
  - Różnią się formaty wyświetlania

Publikacja
autor tytuł rok
pokaz()

Książka
autor tytuł rok wydawca miejsce
pokaz() pokaz_pelny() pokaz_krotki()

Implementacja klasy **Publikacja**

```
class Publikacja {
public:
        enum Format { krotki, dlugi };          // format wyswietlania

        void pokaz(Format f) const;             // interfejs Publikacji

        string autor() const { return p_autor; }                // funkcje dostępu do zmiennych składowych
        void zmien_autor(const string &a) { p_autor = a; }      // ...
        //...
protected:      // Konstruktor jest chroniony – ustaliliśmy, że nie można tworzyć abstrakcyjnej Publikacji:
        Publikacja(const string &a, const string &t, int r) : p_autor(a), p_tytul(t), p_rok(r);
                // Funkcje wirtualne – ich wersjemogą być ponownie zdefiniowane w klasie pochodnej
        virtual void pokaz_krotki() const { cout << autor() << " (" << rok() << ")" << endl; }
        virtual void pokaz_dlugi() const { cout << autor() << ". " << tytul() << ". " << rok() << endl; };
private:
        string p_autor;         // dane składowe są prywatne – chronimy reprezentacje
        string p_tytul;
        int p_rok;
};

void Publikacja::pokaz(Format f) const {                // funkcja interfejsowa
        if (f==krotki) pokaz_krotki();                  // uruchomienie wirtualnej funkcji pokaz_krotki()
        else if (f==dlugi) pokaz_dlugi();
        else throw runtime_error("Publikacja: nieznany format wyswietlania!");
}
```

Implementacja klasy **Książka**

```
class Ksiazka : public Publikacja {
public:
        Ksiazka(const string& a, const string& t, const string& w, const string& m, int r)
                :Publikacja(a,t,r), k_wydawca(w), k_miejsce(m) {
                        if (w.empty()) throw runtime_error("Ksiazka: nie podano wydawnictwa!");
        };

        // ...
        string wydawca() const { return k_wydawca; }
        void zmien_wydawca(const string &w) { k_wydawca = w; }
        // …

        virtual void pokaz_dlugi() const
{ cout << autor() << ". " << tytul() << ". " << wydawca() << ", " << miejsce() << " " << rok() << endl; }
        virtual void pokaz_krotki() const
{ cout << autor() << ". " << tytul() << ". " << wydawca() << " " << rok() << endl; }

private:
        string k_wydawca, k_miejsce;
};
```

Testujemy … funkcje wirtualne

```
void Publikacja::pokaz(Format f) const { // wywołuje pokaz_krotki() lub pokaz_dlugi() ... }
virtual void Publikacja::pokaz_krotki() const { cout << autor() << " (" << rok() << ")" << endl; }

void Ksiazka::pokaz_krotki() const {
        cout<<autor()<<". "<<tytul()<<". "<<wydawca()<<" "<<rok()<<endl;
}

Ksiazka k("B.Stroustrup","Programowanie...","Helion", "Gliwice", 2010);
k.pokaz(Publikacja::krotki);          // dostajemy: B.Stroustrup. Programowanie.... Helion 2010
k.pokaz(Publikacja::dlugi);           // dostajemy: B.Stroustrup. Programowanie.... Helion, Gliwice 2010

Publikacja *p = &k;                   // Uwaga! Wskaźnik na obiekt klasy Publikacja
p->pokaz(Publikacja::krotki);         // dostajemy: B.Stroustrup. Programowanie.... Helion 2010
```

- Działa! Pomimo że funkcja **pokaz()** jest zdefiniowana tylko w klasie **Publikacja**
  - funkcja **pokaz()** wywołuje funkcję **pokaz_krotki()** dla klasy **Ksiazka**, a nie klasy **Publikacja**
- Gdyby funkcja **pokaz_krotki()** w klasie **Publikacja** nie była wirtualna
  - funkcja **pokaz()** wywołałaby funkcję **pokaz_krotki()** dla **Publikacja**

Implementacja klasy **Artykuł**

- Dodawanie kolejnych rodzajów publikacji wygląda podobnie

```
class Artykul : public Publikacja {
public:
	Artykul(const string& a, const string& t, const string& cz, int r, int nr, int sp, int sk=0):
		Publikacja(a,t,r), a_czasopismo(cz), a_numer(nr), a_strona_poczatek(sp), a_strona_koniec(sk)
	{	/*...*/	};

	// funkcje dostępu do zmiennych składowych dodanych w klasie pochodnej

	void pokaz_dlugi() const ;	// formatowanie wyświetlania
	void pokaz_krotki() const;
private:
	string a_czasopismo;
	int a_numer, a_strona_poczatek, a_strona_koniec;
};
```

- Interfejs korzystania z klas pochodnych, określony w klasie bazowej – nie zmienia się:

```
Artykul a("W.Dyrka, J.-C.Nebel","A stochastic context free grammar...",
	"BMC Bioinformatics",2009,10,323);
a.pokaz(Publikacja::krotki);
```

  - Realizujemy ideał rozszerzalności – dodajemy nowe klasy bez konieczności zmieniania istniejącego kodu

Jeszcze jedno

```
Publikacja *pk = new Ksiazka k("B.Stroustrup","Programowanie...","Helion", "Gliwice", 2010);

delete pk; // który destruktor zostanie wywołany?
           // - destruktor klasy Publikacja, który nic nie wie o wydawnictwie i miejscu
           // Wniosek: potrzebny jest destruktor wirtualny
```

- Ogólna zasada
    - klasa bazowa potrzebuje wirtualnego destruktora

```
class Publikacja {
//...
public:
        virtual ~Publikacja();
//...
};
```

Potencjalny problem

```
Ksiazka k("B.Stroustrup","Programowanie...","Helion", "Gliwice", 2010);
Artykul a("W.Dyrka, J.-C.Nebel","A stochastic context free grammar...","BMC
        Bioinformatics",2009,10,323);

// k = a;                                   // Tego nam zrobić nie wolno – i dobrze!

Publikacja *pk = &k;                        // Uwaga! Wskaźnik na obiekt klasy Publikacja
Publikacja *pa = &a;                        // Uwaga! Wskaźnik na obiekt klasy Publikacja

*pk = *pa;
pk->pokaz(Publikacja::krotki);// dostajemy niezłą bzdurę
                              // W.Dyrka, J.-C.Nebel. A stochastic context free grammar.... Helion 2009
```

- Ogólna zasada
  - nie mieszaj hierarchii klas oraz przekazywania przez referencję i domyślnego kopiowania
    - wyłącz konstruktor kopiujący i operator przypisania w klasie bazowej:

```
class Publikacja {
//...
private:
        Publikacja(const Publikacja &);
        Publikacja& operator=(const Publikacja &);
//...
};
```

Dopasowanie funkcji do danych

- Koncepcja:
  - podajemy zbiory punktów dziedziny i wartości obserwowanych
  - podajemy funkcję
  - obliczamy odległość średnio-kwadratową funkcji od wartości obserwowanych
- Prototyp

```
typedef double Funk(double)        // definiujemy typ danych
                                   // funkcja jednoargumentowa
                                   // zwraca i przyjmuje liczby rzeczywiste


double dopasowanie(const vector<double>& dziedzina,
                   const vector<double>& dane,
                   Funk f);

double d = dopasowanie(dziedzina, dane, sqrt)
```

Implementacja f-cji **dopasowanie()**

```
double dopasowanie(const vector<double>& dziedzina, const vector<double>& dane, Funk f)
 {
	if (dziedzina.size()>dane.size()) throw runtime_error("Funkcja nieokreslona dla dziedziny!");

	double odlegloscKwadratowa = 0;
	for (int i=0;i<dziedzina.size();i++)
		odlegloscKwadratowa += pow(f(dziedzina[i])-dane[i],2);

	return odlegloscKwadratowa;
}
```

- Świetnie, ale co jeśli chcemy policzyć dopasowanie do f-cji liniowej?
  - Problem! Funkcja liniowa przyjmuje 3 parametry, np.

    ```
    double funkcja_liniowa(double a, double b, double x) { return a*x+b; }
    ```

  - Możemy stworzyć nowy typ:

    ```
    typedef double Funk2(double,double,double)
    ```

    i przeciążyć f-cję **dopasowanie()**
  - Ok, ale co jeśli weźmiemy f-cję kwadratową?

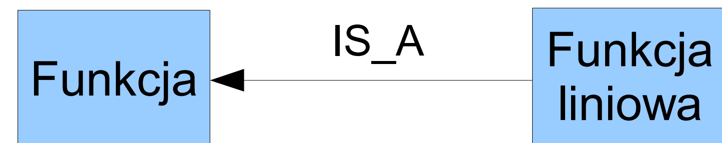
Pomysł – stwórzmy klasę funkcyjną

```
double dopasowanie(const vector<double>& dziedzina, const vector<double>& dane, const Funkcja& f)
{
        if (dziedzina.size()>dane.size()) throw runtime_error("Funkcja nieokreslona dla dziedziny!");

        double odlegloscKwadratowa = 0;
        for (int i=0;i<dziedzina.size();i++)
                odlegloscKwadratowa += pow(f(dziedzina[i])-dane[i],2);
        return odlegloscKwadratowa;
}

class Funkcja { // reprezentuje pojęcie funkcji, jest interfejsem z którego może korzystać dopasowanie()
public:
        virtual double operator()(double x) const = 0;    // funkcja czysto wirtualna, tutaj tylko prototyp,
                                                           // musi być zdefiniowana w klasie pochodnej
        virtual ~Funkcja() { };      // wirtualny destruktor
};

class FunkcjaLiniowa : public Funkcja {
        double a, b;
public:
        FunkcjaLiniowa(double aa, double bb): a(aa), b(bb) { };   // tworzy obiekt reprezentujący funkcję
                                                                  // liniową a*x + b
        double operator()(double x) const { return a*x+b; }       // wykorzystujemy operator()
                                                                  // do zwrócenia wartości (udajemy funkcję)
};
```



Użycie

- Łatwa rozszerzalnośc:

```
class FunkcjaKwadratowa : public Funkcja {
        double a, b, c;
public:
        FunkcjaKwadratowa(double aa, double bb, double cc): a(aa), b(bb), c(cc) { };
        double operator()(double x) const { return a*x*x+b*x+c; }
};
```

- Przykład:

```
vector<double> dziedzina;
dziedzina.push_back(0); dziedzina.push_back(1); dziedzina.push_back(2);

vector<double> dane;
dane.push_back(3); dane.push_back(4); dane.push_back(5);

FunkcjaLiniowa fl(1,2);                         // 1*x + 2
cout << dopasowanie(dziedzina, dane, fl) << endl;

FunkcjaKwadratowa fk(1,2,3);                    // 1*x^2 + 2*x + 3
cout << dopasowanie(dziedzina, dane, fk) << endl;
```

Funkcje czysto wirtualne

- Często funkcja stanowiąca część interfejsu nie może zostać zaimplementowana na poziomie klasy bazowej
  - np. potrzebne dane są „schowane” w klasie pochodnej
  - musimy mieć pewność, że klasa pochodna zdefiniuje taką funkcję
  - w tym celu tworzymy w klasie bazowej funkcję czysto wirtualną
- W rezultacie klasa bazowa staje się abstrakcyjna
  - nie można utworzyć obiektu takiej klasy

```
class Funkcja {         // reprezentuje pojęcie funkcji, jest interfejsem: nie zawiera żadnych danych
public:
    virtual double operator()(double x) const = 0;   // funkcja czysto wirtualna, tutaj tylko prototyp,
                                                     // musi być zdefiniowana w klasie pochodnej
    virtual ~Funkcja() { };     // wirtualny destruktor
};

Funkcja f;              // Błąd! Klasa Funkcja jest abstrakcyjna
```

Doszliśmy do celu: programowanie zorientowane obiektowo

Definicja programowania zorientowanego obiektowo (*Object-Oriented Programming*)

OOP == dziedziczenie + polimorfizm + enkapsulacja

- Dziedziczenie - klasy bazowe i pochodne
  - **class Artykul : Publikacja { … };**
- Polimorfizm czasu wykonania – funkcje wirtualne
  - **virtual void pokaz_krotki() const;**
- Enkapsulacja – składowe prywatne i publiczne
  - **protected: Publikacja(const string&, const string&, int);**
  - **private: string p_autor**

Korzyści z dziedziczenia

- Dziedziczenie interfejsu
  - Funkcja oczekująca obiektu danej klasy może przyjąć dowolny obiekt klasy pochodzącej od tej klasy
    - taki model jest często zdecydowanie łatwiejszy w użyciu
  - Dodanie nowej klasy pochodnej nie wymaga modyfikacji kodu użytkownika
    - który kontaktuje się tylko z klasą bazową
    - dodawanie bez modyfikacji starego kodu jest jednym z ideałów programistycznych
- Dziedziczenie implementacji
  - Ułatwia implementację klas pochodnych
    - wspólna funkcjonalność znajduje się w jednym miejscu
    - zmiany wystarczy wykonać w jednym miejscu – kolejny ideał

Dziś najważniejsze było...

- Programowanie obiektowe to:
  - Dziedziczenie
  - Polimorfizm czasu wykonania
  - Enkapsulacja
- Dziedziczenie interfejsu
  - a dziedziczenie implementacji
- Funkcje wirtualne,
  - czysto wirtualne
    - i klasy abstrakcyjne

Absolutnie nie wyczerpaliśmy tematu dziedziczenia

A za tydzień...

- Programowanie uogólnione: szablony
- za 2 tygodnie:
  - kolokwium zaliczeniowe!