# Programowanie obiektowe C++

## Programowanie zorientowane obiektowo

Wykład 3

Witold Dyrka
witold.dyrka@pwr.wroc.pl

22/10/2012

# Prawa autorskie itp.

*Wiele slajdów do tego wykładu powstało w oparciu o:*

- *slajdy Bjarne Stroustrupa*
  *do kursu* Foundations of Engineering II (C++)
  *prowadzonego w Texas A&M University*
  http://www.stroustrup.com/Programming
- C++ Operator Overloading Guidelines, *Donnie Pinkston, Caltech*
  http://www.cs.caltech.edu/courses/cs11/material/cpp/donnie/cpp-ops.html
- *przykładowe programy Jerzego Grębosza do książki*
  Symfonia C++ Standard
  http://www.ifj.edu.pl/~grebosz/symfonia_c++_std_p.html

*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: konstruktory, destruktory, pryzpisanie (29/10)
5. Dziedziczenie. Polimorfizm dynamiczny (5/11)
6. Programowanie uogólnione: szablony (12/11)

# Materiały

## Literatura

- Bjarne Stroustrup. *Programowanie: Teoria i praktyka z wykorzystaniem C++*. Helion (2010)
- Jerzy Grębosz. *Symfonia C++ standard.* Edition 2000 (2008)
- C++ Operator Overloading Guidelines, *Donnie Pinkston, Caltech* http://www.cs.caltech.edu/courses/cs11/material/cpp/donnie/cpp-ops.html
- Dowolny podręcznik programowania zorientowanego obiektowo w języku C++ w standardzie ISO 98

## Środowisko programistyczne

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

# Słowniczek

- **Abstrakcja** – technika tworzenie programu tak, aby można było z niego korzystać bez wnikania w szczegóły implementacji
- **Enkapsulacja** (hermetyzacja) – ukrywanie składowych klasy zawierających szczegóły implementacji
- **Interfejs** – część klasy umożliwiająca korzystanie z obiektu (co można zrobić z obiektem)
- **Implementacja** – część klasy definiująca szczegółowe sposoby przechowywania danych i działania (jak to jest robione)

- **Niezmienniki** – reguły opisujące poprawne stany klasy

# Słowniczek

- **Funkcje składowe** – funkcje zdefiniowane w klasie, mają dostęp do wszystkich składników obiektu
- **Stałe funkcje składowe** – funkcje składowe, które nie mogą modyfikować obiektu
- **Funkcje pomocnicze** – funkcje zdefiniowane poza klasą, które operują na obiektach klasy
- **Funkcje zaprzyjaźnione** – funkcje pomocnicze z prawem dostępu do prywatnych składników klasy
- **Składniki statyczne** – wspólne dla wszystkich obiektów klasy, nie związane z konkretnym obiektem

# Plan na dziś

- Polimorfizm (statyczny)
  - **przeładowanie nazwy funkcji**
  - przeładowanie operatorów
- Konwersje

# Przeładowanie nazwy funkcji

```
void wypisz(int);
void wypisz(char, double, const string);
void wypisz(int, char);
void wypisz(char, int);

int main()  {
        wypisz(12345);
        wypisz(8, 'X');
        wypisz('D', 89.5, " stopni Celsjusza ");
        wypisz('M', 22);
}

void wypisz(int liczba)  {
        cout << "Liczba typu int: " << liczba << endl;
}
void wypisz(char numer_bloku, double wartosc, const string opis )  {
        cout << "Blok " << numer_bloku << ": " << wartosc << opis << endl;
}
void wypisz(int liczba, char znak)  {
        cout << znak << ") " << liczba << endl;
}
void wypisz(char stan, int liczba)  {
        cout << liczba << " razy wystapil stan " << stan << endl;
}
```

# Przeładowanie nazwy funkcji

```
void wypisz(int);
void wypisz(char, double, const string);
void wypisz(int, char);
void wypisz(char, int);

int main()  {
        wypisz(12345);
        wypisz(8, 'X');
        wypisz('D', 89.5, " stopni Celsjusza ");
        wypisz('M', 22);
}

void wypisz(int liczba)  {
        cout << "Liczba typu int: " << liczba << endl;
}
void wypisz(char numer_bloku, double wartosc, const string opis )  {
        cout << "Blok " << numer_bloku << ": " << wartosc << opis << endl;
}
void wypisz(int liczba, char znak)  {
        cout << znak << ") " << liczba << endl;
}
void wypisz(char stan, int liczba)  {
        cout << liczba << " razy wystapil stan " << stan << endl;
}
```

```
Liczba typu int: 12345
X) 8
Blok D: 89.5 stopni Celsiusza
22 razy wystapil stan M
Aby kontynuować, naciśnij dowolny klawisz . . .
```

# Polimorfizm statyczny

- Dla kompilatora i linkera nazwa funkcji składa się z nazwy funkcji i typu argumentów, np. w MS VC++ 2005:
  - **void wypisz(int) → (?wypisz@@YAXH@Z)**
  - **void wypisz(char, double, const string) → (?wypisz@@YAXDNV?**
    **$basic_string@DU?$char_traits@D@std@@V**
    **$allocator@D@2@@std@@@Z)**
  - **void wypisz(int, char) → (?wypisz@@YAXHD@Z)**
  - **void wypisz(char, int) → (?wypisz@@YAXDH@Z)**
  - **int wypisz(double) → (?wypisz@@YAHN@Z)**
- Decyzja, którą wersję funkcji wybrać jest podejmowana na etapie kompilacji
  - przeładowujemy nazwę funkcji – nie samą funkcję

*Dociekliwi: zajrzyjcie do pliku .OBJ*

# Reguły

- Różne wersje funkcji muszą różnić się typami argumentów

```
int funkcja1(int argument1);
int funkcja1(int parametr1);

int funkcja2(int);
double funkcja2(int);

int funkcja3(int argument1);
int funkcja3(double argument2);

int funkcja4(int, char);
int funkcja4(char, int);

int funkcja5(double);
int funkcja5(const double);

int funkcja6(double &);
int funkcja6(const double&);
```

- Kompilator musi móc rozróżnić o którą wersję chodzi:

```
void funkcja7(int parametr1);
void funkcja7(int &parametr1);

void funkcja8(int parametr1, int p2 = 0);
void funkcja8(int parametr1);
```

# Reguły

- Różne wersje funkcji muszą różnić się typami argumentów

```
int funkcja1(int argument1);
int funkcja1(int parametr1);
```

```
int funkcja2(int);
double funkcja2(int);
```

```
int funkcja3(int argument1);
int funkcja3(double argument2);
```

```
int funkcja4(int, char);
int funkcja4(char, int);
```

**Poprawny kod, ale łatwo o pomyłkę NIE POLECAM**

```
int funkcja5(double);
int funkcja5(const double);
```

```
int funkcja6(double &);
int funkcja6(const double&);
```

- Kompilator musi móc rozróżnić o którą wersję chodzi:

```
void funkcja7(int parametr1);
void funkcja7(int &parametr1);

int a,&b=a;
funkcja7(a);      // Błąd!
funkcja7(b);      // Błąd!
```

```
void funkcja8(int parametr1, int p2 = 0);
void funkcja8(int parametr1);

funkcja8(10);     // Błąd!
funkcja8(10,1); // ale to ok
```

*Dokładne zasady dopasowywania? Poszukaj sam(a)!*

# Plan na dziś

- Polimorfizm (statyczny)
  - przeładowanie nazwy funkcji
  - **przeładowanie operatorów**
- Konwersje

# Przeładowanie operatorów

- Można zdefiniować (czyli przeładować) prawie wszystkie operatory dla operandów typu użytkownika (klasa, wyliczenie), np.

```
enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

Month operator++(Month& m) {     // operator inkrementacji prefiksowej
        m = (m==dec) ? jan : Month(m+1);     // "zawijanie"
        return m;
}


Month m = nov;
++m;        // m zmienia się na dec
++m;        // m zmienia się na  jan
```

# Zasady przeładowania operatorów

- Można definiować działania tylko istniejących operatorów
  - np. + - * / % [] () ^ ! & < <= > >=
- Trzeba zachować konwencjonalną liczbę operandów
  - np., nie ma unarnego (jednoargumentowego) <= (mniejsze-równe) albo binarnego (dwuargumentowego) ! (negacja)
- Każdy przeładowany operator musi mieć przynajmniej jeden operand typu użytkownika
  - **int operator+(int,int);** *// błąd: nie można przeładować* ***wbudowanego*****+**
  - **Wielomian operator+(const Wielomian&, const Wielomian &);** *// ok*
  - **Wielomian operator+(const Wielomian&, int);** *// ok*
- Dobre rady:
  - nadawaj operatorom tylko konwencjonalne znaczenie

    + powinnien być dodawaniem, * - mnożeniem, [] - dostępem, a () wywołaniem
  - przeładowuj tylko jeśli to naprawdę potrzebne

# Złożone operatory przypisania +=, -=, *=, ...

- Załóżmy że potrzebujesz operatorów + i -
  - Naturalne jest oczekiwanie, że udostępnisz też += i -=

- Zacznij definiowanie operatorów właśnie od nich

- Modyfikują obiekt na którym działają
  - Zdefiniuj je jako nie-statyczne składowe klasy

```
Punkt & Punkt::operator+=(const Punkt &prawa_strona) {
    this->x += prawa_strona.x;    // this jest wskaźnikiem na obiekt
            y += prawa_strona.y;  // na rzecz którego wywoływana
    this->z += prawa_strona.z;    // jest funkcja lub operator
    return *this;     // zamiana obiektu *this na referencję następuje automatycznie
}
```

# Złożone operatory przypisania +=, -=, *=, ...

```
Punkt & Punkt::operator+=(const Punkt &prawa_strona) {
    this->x += prawa_strona.x;
    this->y += prawa_strona.y;
    this->z += prawa_strona.z;
    return *this;
}
```

- Przekazujemy zmienną **prawa_strona**
  - przez referencję – ze względu na efektywność
  - stała referencja – obiecujemy, że nie zmieniamy prawej strony przypisania
- Zwracamy referencję do lewej strony przypisania – umożliwia sekwencję przypisań, jak dla typów wbudowanych, np.

```
int a, b, c;
(a += b) += c;

Punkt A(0,1,2), B(3,4,5), C(6,7,8);
(A += B) += C;
```

# Dwuargumentowe operatory arytmetyczne +, -, *, ...

- Nie modyfikują obiektów, które są ich operandami
  - Zdefiniuj je jako funkcje pomocnicze poza klasą

```
const Punkt operator+(const Punkt &lewa_strona, const Punkt &prawa_strona) {
    Punkt wynik = lewa_strona;      // tworzy zmienną wynik i inicjuje ją
                                    // zmienną lewa_strona
                                    // (używa konstruktora kopiującego)
    wynik += prawa_strona;          // wykorzystuje operator złożonego przypisania
    return wynik;
}
```

- To samo, tylko zwieźle:

```
const Punkt operator+(const Punkt &lewa_strona, const Punkt &prawa_strona) {
    return Punkt(lewa_strona) += prawa_strona;
}
```

- Zwracamy stałą wartość by udaremnić nonsensowny kod typu:
  `(a + b) = c;` // *bez sensu*

# Pre- i postinkrementacja ++C i C++

```
class Proba {
        int numer_kolejny;
        vector<bool> wynik;
        …
 };
Proba pr(0);
```

- **Preinkrementacja:** `cout << ++pr;` // *równoważne* ***pr.operator++();***

```
Proba &Proba::operator++() {                // operator inkrementacji prefiksowej
        this->numer_kolejny++;              // modyfikuje obiekt, więc w klasie
        this->wynik.push_back(false);
        // ...
        return *this;
}
```

- **Postinkrementacja:** `cout << pr++;` // *równoważne* ***pr.operator++(0);***

```
Proba Proba::operator++(int pomijany) {     // operator inkrementacji postfiksowej
        Proba kopia = *this;    // wykonaj kopię bieżącego stanu obiektu this
        ++(*this);              // wykorzystaj operator preinkrementacji
        return kopia;
}
```

# Operatory porównania ==, !=

```
bool operator==(const Punkt &lewa_strona, const Punkt &prawa_strona) {
        return lewa_strona.wezX() == prawa_strona.wezX()
         && lewa_strona.wezY() == prawa_strona.wezY
         && lewa_strona.wezZ() == prawa_strona.wezZ;
}
```

*// Zauważ:* ***operator==*** *wcale nie musi być zaprzyjaźniony z klasą* ***Punkt***

```
bool operator!=(const Punkt &lewa_strona, const Punkt &prawa_strona) {
        return !(lewa_strona==prawa_strona);      // wykorzystaj operator równości
}
```

# Operatory wczytania >> i wypisania <<

**MojaKlasa mk;**
**cin >> mk;**
**cout << mk;**

- Modyfikują strumienie wejścia (**cin**) / wyjścia (**cout**)
  - **operator**>> modyfikuje obiekt **mk** typu **MojaKlasa**
- Wymagana składnia:

  **ostream& operator<<(ostream &, const MojaKlasa &);** // *wypisywanie*
  **istream& operator>>(istream &, MojaKlasa &);** // *wczytywanie*

  - Operator NIE może być funkcją składową **MojaKlasa::operator>>(...)** (pierwszy argument NIE jest typu **MojaKlasa)**

# Przykład: Operatory wczytania >> i wypisania << dla klasy **Date**

```
ostream& operator<<(ostream os&, const Date &d) {   // wypisywanie daty
      return os << '(' << d.year()                  // obiekt os klasy ostream może być
            << '/' << d.month()                     // ekranem, plikiem lub łańcuchem zn.
            << '/' << d.day() << ')';
}


istream& operator>>(istream &is, Date &d) { // wczytywanie daty
                                            // w formacie (rok/miesiać/dzień)
      int y,m,d;                    // rok, miesiąc, dzień
      char ch1,ch2,ch3,ch4;         // znaki formatujące
      is >> ch1 >> y >> ch2 >> m3 >> ch3 >> d >> ch4;
      if (!is) return is;           // jeśli wystąpił błąd wejścia (bool operator!() const)
      if (ch!='(' || ch2!='/' || ch3!='/' || ch4!=')')    // błąd formatu daty
            is.clear(ios_base::failbit) ;         // ustawia bit błędu oznaczający zły format
            // Dla dociekliwych: możnaby is skonfigurować tak aby wyrzucała tutaj wyjątek
      return is;
}
```

# Operator dostępu []

- Musi być składową funkcji
  - pobiera jeden parametr – indeks
  - powinien być przeładowany dwukrotnie
    - w wersji zmieniającej i niezmieniającej obiektu

```
double &Wielomian::operator[](int który)  {
      return wektor_wspolczynnikow.at(ktory);        // at() - funkcja z kontrolą zakresu
}


class Agent { vector<string> nazwisko, kryptonim; … };

string &Agent::operator[](string szukany_kryptonim) {              // można indeksować
      for (int i=0; i<kryptonim.size(); i++)                       // innymi typami niż int
            if (kryptonim[i]==szukany_kryptonim) return(nazwisko[i]);
      throw runtime_error("Brak danych");
}
Agent listaAgentow;
cout << listaAgentow["007"];
```

# Operator wywołania ()

- Kiedy klasa ma jedną główną funkcję, np.

```
void Klakson::operator()(int ile_razy) {  for (int i=0;i<ile_razy;i++) cout << "Ti-tit!\t"; }
```

- lub gdy tworzymy obiekt funkcyjny, np.

```
struct Punkt {
        double x;
        double y;
        double z;
        // ...
};

std::vector<Punkt> punkty;

//Tworzymy obiekty funkcyjne
struct PorownajPunktyX {
        bool  operator()(const Punkt &a, const Punkt& b) const {  return a.x < b.x; }
};

struct PorownajPunktyY {
        bool  operator()(const Punkt &a, const Punkt& b) const {  return a.y < b.y; }
};

std::sort(punkty.begin(), punkty.end(), PorownajPunktyX());          // sortuj wg współrzędnej X
std::sort(punkty.begin(), punkty.end(), PorownajPunktyY());          // sortuj wg współrzędnej Y
```

# Plan na dziś

- Polimorfizm (statyczny)
- **Konwersje**

# Konwersje

```
struct zespol {
    double rzeczyw;
    double urojon;

    zespol(double r, double i):rzeczyw(r),urojon(i) { }          // konstruktor
};

int main() {
    double    x = 3.21;
    zespol    z(6, -2);
}
```

# Konstruktor konwertujący

```
struct zespol {
    double rzeczyw;
    double urojon;

    zespol(double r, double i):rzeczyw(r),urojon(i) { }          // konstruktor
    zespol(double r):rzeczyw(r),urojon(0) { }     // konstruktor konwerujący double -> zespol
};

int main() {
    double    x = 3.21;
    zespol    z(6, -2);
    zespol    zz(x);
}
```

# Konwersja

```
struct zespol {
    double rzeczyw;
    double urojon;

    zespol(double r, double i):rzeczyw(r),urojon(i) { }          // konstruktor
    zespol(double r):rzeczyw(r),urojon(0) { }     // konstruktor konwerujący double -> zespol
};

const zespol dodaj(const zespol& a, const zespol& b);

int main() {
    double    x = 3.21;
    zespol    z(6, -2);
    zespol    zz(x);

    zespol wynik = dodaj(z, zz);    // niepotrzebna żadna konwersja

   // poniższe wywołanie nie jest dopasowane, ale mimo to możliwe, bo
   // kompilator samoczynnie zastosuje naszą konwersję
    wynik = dodaj(z, 4);       // konstr. konwertujący double --> zespol      dodaj(z,zespol(4))
}
```

# Zwięzły konstruktor

```
struct zespol {
    double rzeczyw;
    double urojon;

    zespol(double r, double i = 0):rzeczyw(r),urojon(i) { }        // konstruktor (konwertujący)
};

const zespol dodaj(const zespol& a, const zespol& b);

int main() {
    double    x = 3.21;
    zespol    z(6, -2);
    zespol    zz(x);

    zespol wynik = dodaj(z, zz);    // niepotrzebna żadna konwersja

   // poniższe wywołanie nie jest dopasowane, ale mimo to możliwe, bo
   // kompilator samoczynnie zastosuje naszą konwersję
    wynik = dodaj(z, 4);        // konstr. konwertujący double --> zespol      dodaj(z,zespol(4))
}
```

# Funkcja konwertująca

```
struct zespol {
    double rzeczyw;
    double urojon;

    zespol(double r, double i = 0):rzeczyw(r),urojon(i) { }      // konstruktor (konwertujący)
    operator double() { return rzeczyw; }                        // operator konwersji
};

double pole_kwadratu(double bok) { return bok*bok; }

int main() {
    double    x = 3.21;
    zespol    z(6, -2);

    double pole = pole_kwadratu(x);          // niepotrzebna żadna konwersja

    // poniższe wywołanie nie jest dopasowane, ale mimo to możliwe, bo
    // kompilator samoczynnie zastosuje naszą konwersję
    pole = pole_kwadratu(z);       // operator konwersji zespol -> double
                                   //     pole_kwadratu(double(z))
}
```

# Uwaga!

```
struct zespol {
    double rzeczyw;
    double urojon;

    zespol(double r, double i = 0):rzeczyw(r),urojon(i) { }      // konstruktor (konwertujacy)
    operator double() { return rzeczyw; }                        // operator konwersji
};


const zespol dodaj(const zespol& a, const zespol& b);
const zespol operator+(const zespol& a, const zespol& b);

double pole_kwadratu(double bok) { return bok*bok; }

int main() {
    double    x = 3.21;
    zespol    z(6, -2);


    wynik = dodaj(z,x);        // ok, dodaj(z,zespol(x))

    wynik = z+x;               // Błąd! z+zespol(x) czy double(z)+x

}
```

- Rzeczywiście, biblioteczny typ **complex** nie ma operatora konwersji
  - zamiast tego są funkcje **real** i **imag**

# Dziś najważniejsze było to, że...

- Przeładowanie funkcji
  – różne zachowania dla różnych typów argumentów
- Przeładowanie operatorów
  - Można definiować działania tylko istniejących operatorów
  - Trzeba zachować konwencjonalną liczbę operandów
  - Każdy przeładowany operator musi mieć przynajmniej jeden operand typu użytkownika
  - Nadawaj operatorom tylko konwencjonalne znaczenie
  - Przeładowuj tylko jeśli to naprawdę potrzebne

# A za tydzień...

Zarządzanie pamięcią:

- konstruktor,
- destruktor,
- przypisanie
- ...