Programowanie obiektowe C++

Programowanie zorientowane obiektowo

Wykład 2

Witold Dyrka
witold.dyrka@pwr.wroc.pl

15/10/2012

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
- *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)

Słowniczek

- **Klasa**
  - reprezentuje pojęcie w programie
  - technicznie: typ danych użytkownika
- **Obiekt**
  - instancja (wystąpienie) klasy
  - technicznie: miejsce w pamięci przechowujące dane określonego typu
- **Zmienna**
  - nazwany obiekt

Klasa jest typem zdefiniowanym przez użytkownika, który określa jak tworzyć i używać obiekty tego typu

klasy

```
class X {           // klasa o nazwie X
public:

    // funkcje
    // typy
    // dane
};
```

Interfejs klasy

```
class X {           // klasa o nazwie X
public:             // publiczne składowe klasy – interfejs użytkownika
                    //     (dostępne dla wszystkich)
    // funkcje
    // typy
    // dane (najczęściej lepiej, żeby były prywatne)
};
```

```
class X {
public:
    int m;                                          // dana składowa
    int mf(int v) { int old = m; m=v; return old; } // funkcja składowa
};
X var;              // zmienna var typu X
var.m = 7;          // dostęp do danej składowej m zmiennej var
int x = var.mf(9);  // wywołanie funkcji składowej mf() dla zmiennej var
```

Interfejs i implementacja klasy

```
class X {          // klasa o nazwie X
public:            // publiczne składowe klasy – interfejs użytkownika
                   //     (dostępne dla wszystkich)
    // funkcje
    // typy
    // dane (najczęściej lepiej, żeby były prywatne)
private:           // prywatne składowe klasy – szczegóły implementacyjne
                   //     (dostępne tylko dla składowych klasy)
    // funkcje
    // typy
    // dane
};
```

Składowe prywatne

- Składowe klasy są domyślnie prywatne, czyli zapis:

```
class X {
        int mf();
        // …
};
```

- oznacza tyle co:

```
class X {
private:
        int mf();
        // …
};
```

- Więc:

```
X x;                    // zmienna x typu X
int y = x.mf();         // błąd kompilacji: mf jest prywatna (niedostępna)
```

Struktury (w C++)

Klasy mogą mieć dwie bliźniacze formy: **class** i **struct**

- Składowe struktury są domyślnie publiczne, czyli zapis:

```
struct X {
    int m;
    // …
};
```

- oznacza tyle co:

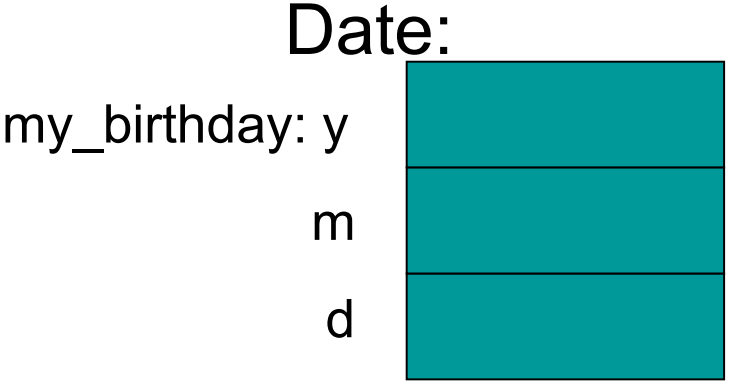
```
class X {
public:
    int m;
    // …
};
```

- Rodzi się pytanie – po co ukrywać składowe?

Po co komu prywatne składowe?

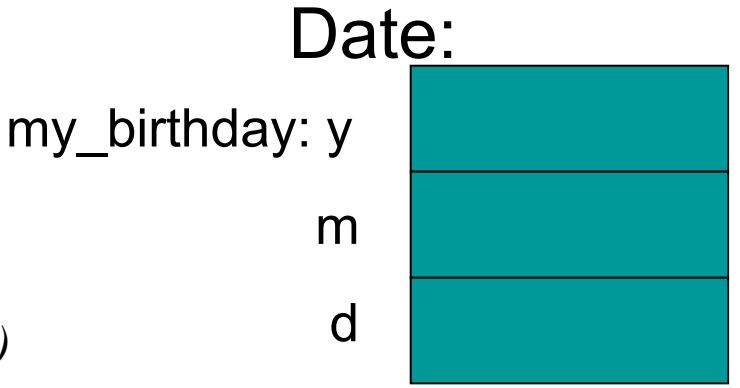
- Aby stworzyć przejrzysty interfejs klasy
  - dane i zagmatwane funkcje można ukryć
- **Aby chronić poprawność obiektu** (kontrolować niezmienniki)
  - tylko ograniczony zbiór funkcji ma dostęp do danych
- Aby ułatwić debugowanie
  - „zawężenie grona podejrzanych”
- Aby umożliwić zmianę reprezentacji (danych składowych)
  - wystarczy zmienić ograniczony zbiór funkcji
  - w przypadku składowej publicznej nie można nigdy wiedzieć, kto ją używa

Struktura – tylko dane (jak w języku C)




```
// najprostsza wersja Date (tylko dane)
struct Date {
    int y,m,d;      // rok, miesiąc, dzień
};

Date my_birthday;       // zmienna typu Date (obiekt)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950;   // Ups! (nie ma dnia 1950 w miesiącu 30)
                        // Błąd nie został zgłoszony, ale dalej w programie
                        // będziemy mieli problemy
```

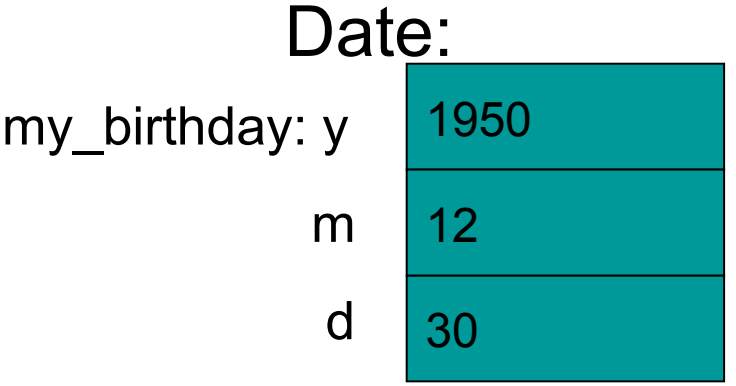
Struktura – tylko dane (jak w języku C) podejście 2




```
// prosta wersja Date (dodaliśmy kilka funkcji pomocnicznych)
struct Date {
    int y,m,d;        // rok, miesiąc, dzień
};

Date my_birthday;          // zmienna typu Date (obiekt)

// funkcje pomocnicze:

void init_day(Date& dd, int y, int m, int d);    // sprawdza poprawność daty i inicjalizuje

void add_day(Date&, int n);                      // zwiększa obiekt Date o n dni
// …

init_day(my_birthday, 12, 30, 1950);       // błąd czasu wykonania:
                                           // nie ma dnia 1950 w miesiącu 30
```

Świetnie, ale nie ma gwarancji, że użytkownik struktury użyje funkcji **init_day()** po utworzeniu zmiennej typu **Date**.

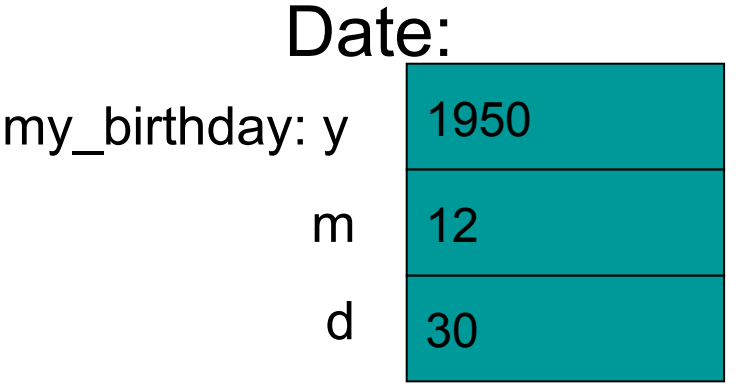
Struktura w stylu C++




```
// prosta Date
//          gwarantuje inicjalizację przy użyciu konstruktora
//          zapewnia pewną wygodę notacyjną
struct Date {
    int y,m,d;                  // rok, miesiąc, dzień
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);        // zwiększa datę o n dni
};

// …
Date my_birthday;                   // błąd: my_birthday nie zainicjowane (I dobrze!)
Date my_birthday(12, 30, 1950);     // ups! Błąd czasu wykonania
Date my_day(1950, 12, 30);          // ok
my_day.add_day(2);                  // 1 stycznia 1951
my_day.m = 14;                      // grrrhh! (teraz my_day jest niepoprawną datą)
```

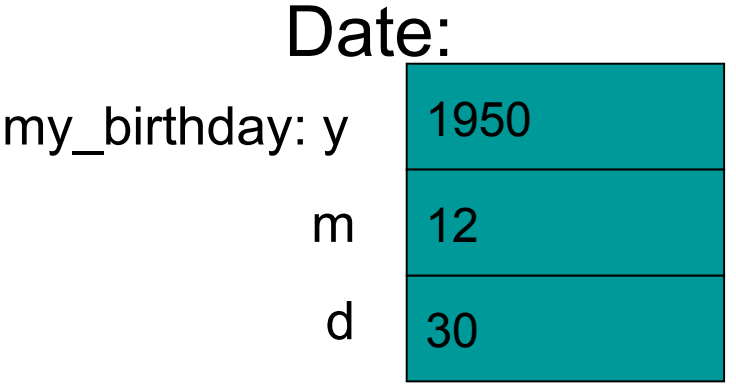
Świetnie, poprawna inicjacja jest zagwarantowana. Niestety, publiczny dostęp do danych składowych **Date** grozi “zepsuciem” daty

Struktura w stylu C++
(z kontrolą dostępu)



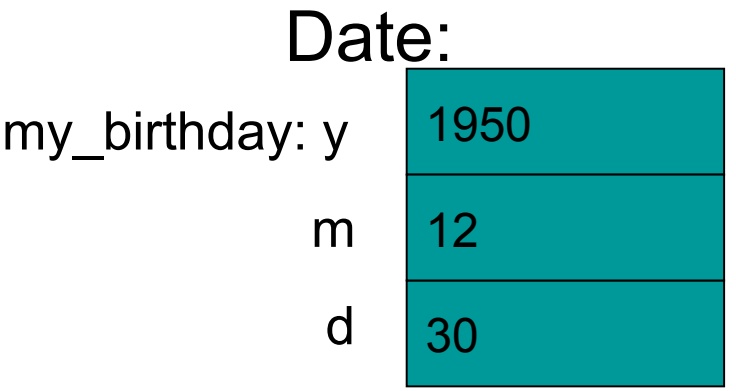

```
// prosta Date (z kontrolą dostępu)
struct Date {
private:
    int y,m,d;       // rok, miesiąc, dzień
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);        // zwiększa datę o n dni
};
```

Klasa
(z kontrolą dostępu)




```
// prosta Date (z kontrolą dostępu)
class Date {
    int y,m,d;       // rok, miesiąc, dzień
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);       // zwiększa datę o n dni
};


// …
Date my_birthday(1950, 12, 30);          // ok
my_birthday.m = 14;                      // błąd: składowa Date::m jest prywatna
cout << my_birthday.m   ;                // błąd: składowa Date::m jest prywatna
```

Funkcje dostępowe

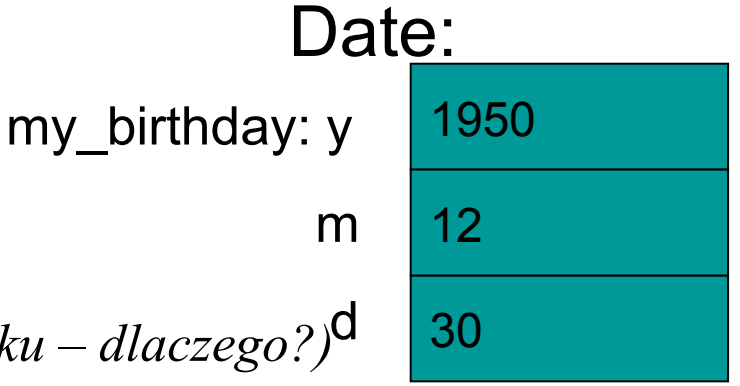



```
// prosta Date (z kontrolą dostępu)
class Date {
    int y,m,d;      // rok, miesiąc, dzień
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje

    // funkcje dostępowe:
    void add_day(int n);      // zwiększa datę o n dni
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};

// …
Date my_birthday(1950, 12, 30);          // ok
my_birthday.m = 14;                      // błąd: składowa Date::m jest prywatna
cout << my_birthday.m                    // błąd: składowa Date::m jest prywatna

cout << my_birthday.month() << endl;     // możemy odczytać miesiąc
```

Niezmienniki

- Pojęcie **poprawnej daty** jest szczególnym przypadkiem pojęcia **poprawnej wartości**
- Zasada: zagwarantować poprawne wartości dla projektowanego typu (**poprawny stan obiektu**)
  - inaczej musielibyśmy cały czas sprawdzać poprawność danych lub liczyć, że zrobi to użytkownik klasy
- Regułę opisującą poprawną wartość nazywa się **niezmiennikiem**
  - w przypadku daty jest to wyjątkowo trudne (28 luty, rok przestępny itd.)
- Jeśli nie można znaleźć dobrego niezmiennika – pracujesz na samych „czystych” danych
  - użyj struktury
  - pierwszą opcją jest jednak zawsze szukanie niezmienników!

Prosty konstruktor klasy




```
// prosta Date (niektórzy wolą interfejs klasy na początku – dlaczego?)
class Date {
public:
    Date(int y, int m, int d);  // konstruktor: sprawdza poprawność daty i inicjuje
    void add_day(int n);        // zwiększa datę o n dni
    int month();
    // ...
private:
    int y,m,d;      // rok, miesiąc, dzień
};

Date::Date(int yy, int mm, int dd)          // definicja konstruktora klasy
    :y(yy), m(mm), d(dd) { /* ... */ };     // inicjacja składowych

void Date::add_day(int n) { /* ... */ };    // definicja funkcji
```

Konstruktor z kontrolą poprawności

```
// prosta Date (co zrobimy z nieprawidłową datą)
class Date {
public:
    class Invalid { };                  // użyta jako rakieta sygnalizacyjna
                                        // przy obsłudze wyjątków
    Date(int y, int m, int d);          // sprawdza poprawność daty i inicjuje
    // …
private:
    int y,m,d;                          // rok, miesiąc, dzień
    bool check(int y, int m, int d);    // czy (y,m,d) jest poprawną datą?
};

Date:: Date(int yy, int mm, int dd)
    : y(yy), m(mm), d(dd)               // inicjuje dane składowe
{
    if (!check(y,m,d)) throw Invalid();     // sprawdza poprawność daty
}
```

Konstruktor z kontrolą poprawności

```
// prosta Date (co zrobimy z nieprawidłową datą)
class Date {
public:

    Date(int y, int m, int d);          // sprawdza poprawność daty i inicjuje
    // …
private:
    int y,m,d;                          // rok, miesiąc, dzień
    bool check(int y, int m, int d);    // czy (y,m,d) jest poprawną datą?
};

Date:: Date(int yy, int mm, int dd)
    : y(yy), m(mm), d(dd)                   // inicjuje dane składowe
{
    if (!check(y,m,d)) throw 1;             // sprawdza poprawność daty
}
```

Wyliczenia

- Wyliczenie **enum** *(ang. enumaration)* jest typem użytkownika określającym zestaw wartości (elementy wyliczenia)

- Na przykład:

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

Month m = feb;
m = 7;                  // błąd: nie można przypisać int do Month
int n = m;              // ok: możemy dostać wartość liczbową Month
Month mm = Month(7);    // konwersja typu int na Month (nie sprawdzana)
```

Rodzaje wyliczeń

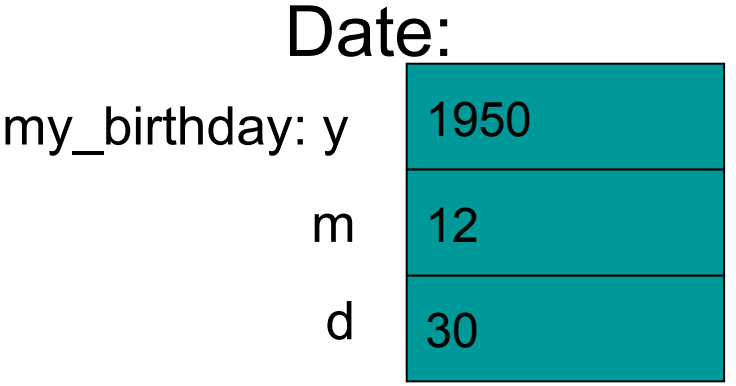
- Typ z listą stałych

```
enum Day { sun, mon, tue, /* ... */ };   // domyślnie: sun==0, mon==1, tue==2
enum Color { red=100, green=1, blue=10, /* ... */  };

Day m1 = sun;
Day m2 = red;    // błąd: red nie jest wartością Day
Day m3 = 7;      // błąd: 7 nie jest wartością Day
int i = m1;      // ok: element wyliczenia jest konwertowany do
                 // swojej wartości, i==0
```

- Prosta lista stałych

```
enum { red, green };          // enum { } nie definiuje zakresu

int a = red;                  // red jest dostępne tutaj

enum { red, blue, purple };   // błąd: red jest już zdefiniowany
```

Klasa z wyliczeniem




```
// prosta Date (używa typu Month)
class Date {
public:
    enum Month {
        jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
    };
    Date(int y, Month m, int d);  // sprawdza poprawność daty i inicjuje
    // …
private:
    int y;          // rok
    Month m;
    int d;          // dzień
};

Date my_birthday(1950, 30, Date::dec);   // błąd: 2. argument nie jest miesiącem
Date my_birthday(1950, Date::dec, 30);   // ok – enum pomaga kontrolować poprawność
                                         // zwróć też uwagę na zakres: Date::dec
```

Obiekty stałe

```
class Date {
public:
    // ...
    int day() { return d; }
    void add_day(int n);
    // ...
};

Date d(2000, Date::jan, 20);          // zmienna d typu Date
const Date cd(2001, Date::feb, 21);   // stała cd typu Date

cout << d.day();          // ok
cout << cd.day();         // błąd: cd jest const – ale przecież day()
                          // nie modyfikuje obiektu!
d.add_day(1);    // ok
cd.add_day(1);   // błąd: cd jest const – tutaj błąd „ma sens”
```

Składniki stałe klasy (**const**)

```
class Date {
public:
    // ...
    int day() const { return d; }     // stały składnik: nie może modyfikować
    void add_day(int n);              // nie-stały składnik: może modyfikować
    // ...
};

Date d(2000, Date::jan, 20);          // zmienna d typu Date
const Date cd(2001, Date::feb, 21);   // stała cd typu Date

cout << d.day();              // ok
cout << cd.day();             // ok

d.add_day(1);    // ok
cd.add_day(1);   // błąd: cd jest const – dobrze! tylko stałe funkcje składowe są
                                  dopuszczane do działania na stałych obiektach klasy
```

Składniki stałe klasy (2)

```
//
Date d(2004, Date::jan, 7);                 // zmienna
const Date d2(2004, Date::feb, 28);         // stała
d2 = d;             // błąd: d2 jest const
d2.add(1);          // błąd: d2 jest const
d = d2;             // w porządku
d.add(1);           // w porządku

d2.f();   // Taka operacja powinna zostać wykonana wtedy i tylko wtedy, gdy
          // funkcja składowa f() nie modyfikuje danych składowych obiektu d2
          // Jak to zapewnić? (zakładając, że tego właśnie chcemy)
```

Składniki stałe klasy (3)

```
// Rozróżnij pomiędzy funkcjami, które mogą modyfikować obiekty
// I tymi, które nie mogą – zwanymi stałymi funkcjami składowymi
class Date {
public:
    // …
    int day() const;          // pobierz dzień (w zasadzie jego kopię)
    // …
    void add_day(int n);  // przesuń datę o n do przodu
    // …
};

const Date dx(2008, Month::nov, 4);
int d = dx.day();             // w porządku
dx.add_day(4);                // błąd: nie można modyfikować stałych danych
```

Dobry interfejs klasy

- Minimalny
  - Tak mały jak to możliwe
- Kompletny
  - I nie mniejszy
- Bezpieczny dla typów (ang. *type-safe*)
  - Uwaga na kolejność argumentów
- Poprawnie określać stałe składniki klasy (ang. *const-correct*)

Minimalny zestaw składowych

- Najbardziej podstawowe operacje
  - Konstruktor domyślny (**domyślnie:** nie robi nic lub brak domyślnego konstruktora jeśli zadeklarowano inny konstruktor
  - Konstruktor kopiujący (**domyślnie:** kopiuje składowe)
  - Operator przypisania (**domyślnie:** kopiuje składowe)
  - Destruktor (**domyślnie:** nie robi nic)
- Na przykład

```
Date d;          // błąd: brak domyślnego konstruktora (zadeklarowaliśmy inny!)
Date d2 = d;     // ok: inicjacja kopią (kopiuje elementy – domyślne)
d = d2;          // ok: przypisanie kopii (kopiuje elementy – domyślne)
```

Interfejs klasy i funkcje pomocnicze

- Chcemy minimalny interfejs klasy (zestaw funkcji publicznych), bo to:
  - ułatwia zrozumienie klasy
  - upraszcza debugowanie
  - upraszcza pielęgnację

- Potrzebujemy funkcji pomocniczych, operujących na obiektach klasy, ale zdefiniowanych poza klasą, np.
  - == (równość) , **!**= (nierówność)
  - **next_weekday()**, **next_Sunday()**

Funkcje pomocnicze

```
Date next_Sunday(const Date& d)
{
    // ma dostęp do obiektu d poprzez d.day(), d.month(), oraz d.year()
    // tworzy nowy obiekt typu Date który zwraca
}

Date next_weekday(const Date& d) { /* ... */ }

bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b) { return !(a==b); }
```

Funkcje zaprzyjaźnione

- Czasem warto dać funkcji pomocniczej dostęp do prywatnych składowych klasy
  - np. zamiast tworzenia funkcji **day(), month(), year().**

```
class Date {

public:
    // ..
    friend void print(Date &d) const;      // funkcja zaprzyjaźniona wyświetlająca datę

private:
    int y,m,d;

};


void print(Date &d) const {
    cout << "Today is " << d.d << "/" << d.m << "/" << d.y << endl;

}

Date dx;

print(&dx);
```

Składniki statyczne

...czyli składniki wspólne dla wszystkich obiektów danej klasy („zmienne globalna dla klasy”)

- Zastosowania:
  - Do porozumiewania się pomiędzy obiektami klasy (jako flaga)
  - Jako istniejących licznik obiektów danej klasy
  - Gdy pewna cecha będzie się zmieniać jednocześnie dla wszystkich obiektów danej klasy
  - Dostęp do wejścia/wyjścia (np. pliku)
    - wspólnego dla wszystkim obiektów klasy

Składniki statyczne – przykład

```
class piorko {

        enum tak_czy_nie { nie, tak } ;

        int poz_x, poz_y ;
        static int zezwolenie ;
        string kolor ;

public :
        // funkcje ------------------------------------------------------------
        void jazda(int x , int y);
        static void czy_mozna(tak_czy_nie odp)
        {
                zezwolenie = odp ;                  // składnik statyczny zezwolenie
                // poz_x = 5 ;                      // błąd ! bo składnik zwykły poz_x
                                                    // Funkcja statyczna może odwoływać się
                                                    // tylko do składników statycznych klasy

        }
        piorko(const string kol) : kolor(kol), poz_x(0), poz_y(0) { };
};
```

dana statyczna
tu: prywatna

funkcja statyczna

Składniki statyczne – przykład (kont.)

```
void piorko::jazda(int x , int y) {
        cout << "Tu "<< kolor << " piorko : " ;
        if(zezwolenie) {
                poz_x = x ;  poz_y = y ;
                cout << "Jade do punktu ("<< poz_x  << ", " << poz_y << ")\n" ;
        } else {
                cout << "     Nie wolno mi jechac !!!!!!!!!!!! \n" ;
        }
}

int piorko::zezwolenie = piorko::nie;        // Daną statyczną definujemy i inicjujemy poza ciałem klasy
                                             // Wyjątek: stała statyczna

int main() {
        piorko::czy_mozna(piorko::tak) ;     // Funkcję statyczną możemy wywołać mimo, że
                                             // nie został jeszcze utworzony obiekt klasy piorko
        piorko czarne("SMOLISTE"), zielone("ZIELONIUTKIE") ;
        czarne.jazda(0, 0) ;
        zielone.jazda(1100, 1100) ;
        piorko::czy_mozna(piorko::nie) ;     // zabraniamy ruchu piórkom
        czarne.jazda(10, 10) ;
        zielone.jazda(1020, 1020) ;
        zielone.czy_mozna(piorko::tak);      // zezwalamy w taki sposób
        czarne.jazda(50, 50) ;
        zielone.jazda(1060, 1060) ;
}
```

Dziś najważniejsze było to, że...

- Oddzielenie interfejsu od implementacji
  - dane składowe przeważnie powinny być prywatne
- Zasady tworzenia dobrego interfejsu:
  - minimalny, ale kompletny
  - bezpieczny dla typów
  - określa składowe stałe

A za tydzień...

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