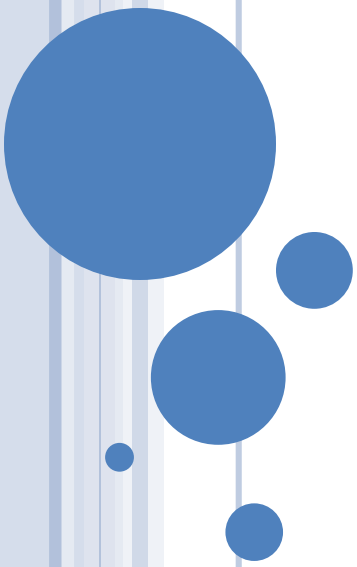

Kurs języka C++ – wykład 2 (5.03.2019)

Klasy i obiekty

Spis treści

Klasy

- Klasa to nowy typ danych (projekt).
- Obiekt to instancja klasy (realizacja projektu).
- Klasa posiada różne pola i metody:
  - wartości pól w obiekcie określają stan obiektu,
  - metody określają funkcjonalność obiektu.
- Klasę definiuje się następująco:

```
class klasa {
    // definicje pól
    // deklaracje metod
};
```

Klasy

- Przykład klasy:

```
class Punkt {
public:
    double x, y;
    Punkt (double a, double b);
    void przesun_x (double dx);
    void przesun_y (double dy);
    double odleglosc (Punkt p);
};
```

Klasy

- Metody w klasie tylko deklarujemy (jak funkcje w plikach nagłówkowych).
- Definicje metod umieszczamy poza klasą (definicje te są kwalifikowane nazwą kasy za pomocą operatora zakresu `::`).
- Przykład definicji metody poza klasą:

```
void Punkt::przesun_x (double dx) {
    x += dx;
}
```

- Zmienne globalne czy funkcje globalne kwalifikujemy operatorem zakresu globalnego:

```
::zmienna;
::f();
```

Konstruktor

- Konstruktor to specjalna metoda uruchamiana tylko podczas inicjalizacji obiektu.
- Konstruktor ma taką samą nazwę jak klasa.
- Konstruktor nie zwraca żadnego wyniku.
- Przykład konstruktora:

```
Punkt::Punkt (double a, double b) {
    x = a, y = b;
}
```

Obiekty

- Można utworzyć obiekt na stosie za pomocą zwykłej deklaracji połączonej z inicjalizacją.
- Przykład obiektu automatycznego:

```
Punkt a = Punkt(4,6);
Punkt b(5,7);
Punkt c {3, 8};
```

- Można też utworzyć obiekt na stercie za pomocą operatora `new`. Pamiętaj o usunięciu go operatorem `delete`, gdy będzie niepotrzebny.
- Przykład obiektu w pamięci wolnej:

```
Punkt *p = new Punkt(-2,-3);
// …
delete p;
```

Składowe w obiekcie

- Do składowych w obiekcie odwołujemy się za pomocą operatora dostępu do składowych (kropka `.` dla obiektów i referencji albo strzałka `->` dla wskaźników).
- Metoda jest wywoływana na rzecz konkretnego jednego obiektu.
- Przykłady odwołania do funkcji składowych w obiekcie:

```
Punkt a(17,23), b(20,19), *p = &a;
double d = a.odleglosc(b);
b.przesun_y(8);
p->przesun_x(6);
```

Hermetyzacja

- **Programowanie obiektowe** to paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących *stan* (czyli dane, nazywane najczęściej *polami*) i *zachowanie* (czyli funkcje składowe, nazywane też *metodami*).
- **Abstarkcja** to I paradygmat programowania obiektowego – każdy obiekt w systemie jest modelem abstrakcyjnego wykonawcy, który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami bez ujawniania, w jaki sposób zaimplementowano jego cechy.
- **Hermetyzacja** (nazywana też **enkapsulacją**) to II paradygmat programowania obiektowego – oznacza zamknięcie w obiekcie danych i funkcji składowych do operowania na tych danych. Hermetyzacja to również ukrywanie implementacji – zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób (tylko własne metody obiektu są uprawnione do zmiany jego stanu). Każdy typ obiektu prezentuje innym obiektom swój **interfejs**, który określa dopuszczalne metody współpracy.

Klasy

- Klasa to typ zdefiniowany przez programistę.
- Program to zbiór deklaracji i definicji klas.
- Klasa jest modelem (projektem) a obiekt jest instancją klasy (realizacją projektu).
- Klasa posiada różne pola i metody:
  - wartości pól w obiekcie określają stan obiektu.
- Obiekt posiada własne pola a wspólne dla wszystkich obiektów są funkcje składowe (metody):
  - metody pracują na rzecz konkretnego obiektu.

Klasy

- Klasę definiuje się następująco:

```
class klasa {
    // definicje pól
    // deklaracje metod
};
```

- Po zrobieniu definicji można tworzyć obiekty klasy:

```
klasa x, y, z;
```

- Możemy też tworzyć wskaźniki, referencje i tablice obiektów danej klasy:

```
klasa *wsk = &x;
klasa &ref = y;
klasa *&r2w = wsk;
klasa tab[10];
```

Klasy

- Przykład definicji klasy (w pliku nagłówkowym `.hpp`):

```
class punkt {
private:
    double x, y;
public:
    punkt (double a, double b);
    ~punkt ();
    void przesun_x (double dx);
    void przesun_y (double dy);
    double wsp_x ();
    double wsp_y ();
    double odleglosc (punkt &p);
};
```

Obiekty

- Można utworzyć obiekt na stosie za pomocą zwykłej deklaracji połączonej z inicjalizacją.
- Przykład obiektu automatycznego:

```
punkt a = punkt(4,6);
punkt b(5,7);
```

- Można też utworzyć obiekt na stercie za pomocą operatora `new`. Pamiętaj o usunięciu go operatorem `delete`, gdy będzie niepotrzebny.
- Przykład obiektu w pamięci wolnej:

```
punkt *p = new punkt(-2,-3);
// …
delete p;
```

Składowe w klasie

- Wewnątrz klasy można zdefiniować pola składowe (podobnie jak zmienne) oraz zadeklarować funkcje składowe (podobnie jak funkcje globalne).
- Każdy obiekt ma własny zestaw pól składowych. Wartości pól składowych w obiekcie wyznaczają jego stan.
- Funkcje składowe określają funkcjonalność klasy. Za pomocą funkcji składowych można sterować stanem obiektów i ich zachowaniem.

Odwołania do składowych w klasie

- Do składowych w obiekcie odwołujemy się za pomocą operatora dostępu do składowych (kropka . dla obiektów i referencji albo strzałka −> dla wskaźników).
- Metoda jest wywoływana na rzecz konkretnego jednego obiektu.
- Przykłady odwołania do składowych w obiekcie:

```
punkt a(17,23), b(20,19);
punkt *p = &a, &r = b;
double d = a.odleglosc(b);
r.przesun_y(8);
p->przesun_x(6);
```

Pola składowe

- Pola w klasie mogą być danymi typu podstawowego (`bool`, `char`, `int`, `double`, itd), ale mogą też być obiektami innych klas.
- Przyłady:

```
struct lwymierna {
    int licznik, mianownik;
};
struct osoba {
    int rok_ur;
    double waga, wzrost;
    string imie, nazwisko;
};
```

- Budowanie nowej klasy w oparciu o obiekty innych klas nazywa się **kompozycją**.

Funkcje składowe

- Funkcje składowe w klasie tylko deklarujemy (jak funkcje globalne w plikach nagłówkowych).
- Definicje metod umieszczamy poza klasą (definicje te są kwalifikowane nazwą kasy za pomocą operatora zakresu `::`).
- Przykład definicji metod poza klasą (w pliku źródłowym `.cpp`):

```
void punkt::przesun_x (double dx) { x += dx; }
void punkt::przesun_y (double dy) { y += dy; }
double punkt::wsp_x () { return x; }
double punkt::wsp_y () { return y; }
double punkt::odleglosc (Punkt &p) {
  double dx=x-p.x, dy=y-p.y;
  return sqrt(dx*dx+dy*dy);
}
```

- W ciele metody możemy się odnosić do wszystkich składowych w tej samej klasie bez operatora zakresu `::`.

Konstruktor

- Konstruktor to specjalna metoda uruchamiana tylko podczas inicjalizacji obiektu – jego celem jest nadanie początkowego stanu obiektowi.
- Konstruktor ma taką samą nazwę jak klasa.
- Konstruktor nie zwraca żadnego wyniku.
- Przykład konstruktora:

```
punkt::punkt (double a, double b) {
     x = a, y = b;
}
```

Konstruktor domyślny

- Jeśli programista nie zdefiniuje żadnego konstruktora w klasie, wówczas kompilator wygeneruje **konstruktor domyślny** (konstruktor bezargumentowy), który nic nie robi.
- Przykład konstruktora bezargumentowego zdefiniowanego jawnie:

```
punkt::punkt () {
     x = y = 0;
}
```

- Deklaracja obiektu z konstruktorem domyślnym:

```
// punkt p(); - to jest źle!
punkt p = punkt(); // to samo co; Punkt p;
// punkt p; - to jest też dobrze!
```

Konstruktor domyślny

- Jeśli programista zdefiniował jakieś konstruktory w klasie i chciałby mieć **konstruktor domyślny**, to może wymusić na kompilatorze wygenerowanie konstruktora domyślnego za pomocą frazy **`=default`** umieszczonej na końcu deklaracji.
- Przykład konstruktora domyślnego, który zostanie wygenerowany przez kompilator:

```
punkt() = default;
```

Destruktor

- Destruktor to specjalna metoda uruchamiana podczas likwidacji obiektu – jego celem jest posprzątanie po obiekcie (zwolnienie jego zasobów – pamięć na stercie, pliki, itp.).
- Nazwa destruktora to nazwa klasy poprzedzona tyldą.
- Destruktor nie zwraca żadnego wyniku.
- Destruktor nie przyjmuje żadnych argumentów.
- Przykład destruktora:

```
punkt::~punkt () {
     x = y = 0;
}
```

Destruktor

- Destruktor można wywołać jawnie w czasie życia obiektu tak jak zwykłą funkcję składową:

```
punkt p(1,2);
punkt *pp = &p;
//…
p.~punkt();
pp->~punkt();
```

- Destruktor można wywołać w jawny sposób na przykład w przypisaniu kopiującym.
- Destruktora nie powinno się wywoływać w sposób jawny w programie!

Wskaźnik `this`

- Wskaźnik `this` jest ukrytym parametrem każdej instancyjnej funkcji składowej.
- Wskaźnik `this` pokazuje na bieżący obiekt.
- Wskaźnika tego używany tylko w funkcjach składowych.
- Typ wskaźnika `this` jest taki jak klasy, w której jest używany.
- `this` stosujemy najczęściej w przypadku:
  - zasłonięcia nazwy składowej przez nazwę lokalną (na przykład przez nazwę argumentu);
  - jawnego wywołania destruktora (`this->~Klasa();`).

Ukrywanie składowych

- Całą definicję klasy można podzielić na bloki o różnych zakresach widoczności.
- Początek bloku rozpoczyna się od frazy `public:`, `private:` albo `protected:`.
- Składowe publiczne (blok `public:`) są widoczne w klasie i poza klasą.
- Składowe prywatne (blok `private:`) są widoczne tylko w klasie (również w zewnętrznej definicji funkcji składowej danej klasy).
- Składowe chronione (blok `protected:`) są widoczne tylko w klasie i w klasach pochodnych od danej klasy.

Ukrywanie składowych

- Domyślnie wszystkie składowe w klasie są prywatne a w strukturze publiczne.
- Ukrywamy informacje wrażliwe, by ktoś spoza klasy przypadkiem nie zniszczył stanu obiektu.
- Dobrym obyczajem w programowaniu jest ukrywanie pól składowych, do których dostęp jest tylko poprzez specjalne funkcje składowe (zwane metodami dostępowymi albo akcesorami – gettery do czytania i settery do pisania).

Przeciążanie nazw funkcji

- **Przeciążanie** albo **przeładowanie** nazwy funkcji polega na zdefiniowaniu kilku funkcji o takiej samej nazwie.
- Funkcje przeciążone muszą się różnić listą argumentów – kompilator rozpoznaje po argumentach, o którą wersję danej funkcji chodzi.
- Możemy przeciążać również funkcje składowe i konstruktory w klasie.
- Przykład przeciążenia konstruktora:

```
class punkt {
    double x, y;
public:
    punkt ()
        { x = y = 0; }
    punkt (double x, double y)
        { this->x = x; this->y = y; }
}
```

Stałe

- Modyfikator `const` oznacza stałość (brak zmian) zmiennych albo argumentów funkcji.
- Stałe trzeba zainicjalizować.
- Przykład definicji stałej:

```
const double pi =
  3.14159265358979323864264338327795;
```

- W programie niewolno modyfikować wartości zmiennych ustalonych (poprzez przypisanie nowych wartości).
- Zmienne o ustalonej wartości to przeważnie stałe globalne.
- Pola stałe bardzo często są deklarowane w klasie jako pola publiczne.

Stałe kontra `#define`

- Rozważmy następujące definicje:
  - `#define E 2.718281828459`
  - `const double E = 2.718281828459;`
- W przypadku makrodefinicji nazwa `E` jest kompilatorowi zupełnie nieznana (będzie usunięta w fazie preprocesingu).
- Nazwa `E` w przypadku stałej ma swój zakres ważności.
- Stała `E` to obiekt w pamięci i ma swój adres.
- Stała `E` to obiekt o określonym typie.

Uogólnione wyrażenia stałe

- Stałe wyrażenia to wyrażenia, które zawsze zwracają ten sam wynik i nie wywołują żadnych dodatkowych efektów ubocznych (na przykład 3+5).
- Stałe wyrażenia są dla kompilatora okazją do optymalizacji, ponieważ kompilator często wylicza te wyrażenia w czasie kompilacji i wstawia ich wyniki do programu.
- Zmienne typu `constexpr` (stałowyrażeniowe) są niejawnie przekształcane do typu `const` – mogą one przechować wyniki wyrażeń stałych lub stałowyrażeniowych konstruktorów (czyli zdefiniowanych ze słowem kluczowym `constexpr`).
- Przykład:

```
constexpr double grawitacja = 9.8;
constexpr double grawitKsiezyca = grawitacja / 6;
```

Uogólnione wyrażenia stałe

- Za pomocą słowa kluczowego `constexpr` można zagwarantować, że funkcja lub konstruktor obiektu są stałymi podczas kompilacji.
- Zastosowanie `constexpr` do funkcji narzuca bardzo ścisłe ograniczenia na to, co funkcja może robić:
  - funkcja musi posiadać typ zwracany różny od `void`;
  - cała zawartość funkcji musi być tylko instrukcją `return`;
  - wyrażenie musi być stałym wyrażeniem po zastąpieniu argumentu – to stałe wyrażenie może albo wywołać inne funkcje tylko wtedy, gdy te funkcje też są zadeklarowane ze słowem kluczowym `constexpr` albo używać innych stałych wyrażeń;
  - wszystkie formy rekursji w stałych wyrażeniach są zabronione;
  - funkcja zadeklarowana ze słowem kluczowym `constexpr` nie może być wywoływana, dopóki nie będzie zdefiniowana w swojej jednostce translacyjnej.

Uogólnione wyrażenia stałe

- Stałowyrażeniowy konstruktor służy do konstrukcji wartości stałowyrażeniowych z typów zdefiniowanych przez użytkownika, konstruktory takie muszą być zadeklarowane jako `constexpr`.
- Stałowyrażeniowy konstruktor musi być zdefiniowany przed użyciem w jednostce translacyjnej (podobnie jak metoda stałowyrażeniowa) i musi mieć puste ciało funkcji i musi inicjalizować swoje składowe za pomocą stałych wyrażeń na liście inicjalizacyjnej.
- Destruktory takich typów powinny być trywialne.

Argumenty stałe

- Modyfikator `const` może występować przy argumentach w funkcji.
- Jeśli argument jest stały to argumentu takiego nie wolno w funkcji zmodyfikować.
- Przykład funkcji z argumentami stałymi:

```
int abs (const int a) {
    return a<0 ? -a : a;
}
```

- Często argumentami stałymi są referencje.
- Przykład funkcji z argumentami stałymi:

```
int min (const int &a, const int &b) {
    return a<b ? a : b;
}
```

- Argument stały jest inicjalizowany przy wywołaniu funkcji.

Referencja do stałej jako argument w funkcji

- Referencja do stałej może się odnosić do obiektu zewnętrznego (może być zadeklarowany jako stały) ale również do obiektu tymczasowego.
- Przykład referencji do stałej:

```
const int &rc = (2*3-5)/7+11;
```

- Przykład argumentu funkcji, który jest referencją do stałej:

```
int fun (const int &r);
// wywołanie może mieć postać
// fun(13+17);
// gdzie argumentem może być wyrażenie
```

Argumenty tymczasowe

- Obiekty tymczasowe (określane jako r-wartości), mogą być przekazywane do funkcji, ale tylko jako referencje do stałej.
- Z punktu widzenia funkcji, której argumentem jest referencja do stałej, nie jest możliwe rozróżnienie pomiędzy aktualną r-wartością a zwykłym obiektem przekazanym referencję.
- Referencję do r-wartości definiujemy:
  `TYP &&zm`
- Referencja do r-wartości może być akceptowana jako niestała wartość, co pozwala obiektom na ich modyfikację.
- Taka zmiana umożliwia obiektom na stworzenie semantyki przenoszenia.

Argumenty tymczasowe

- Obiekty tymczasowe (określane jako r-wartości), to wartości stojące po prawej stronie operatora przypisania (analogicznie zwykła referencja do zmiennej stojącej po lewej stronie przypisania nazywa się l-wartością).
- Argument w funkcji będący referencją do r-wartości definiujemy jako `TYP &&arg`.
- Argument będący r-referencją może być akceptowany jako niestała wartość, co pozwala funkcjom na ich modyfikację.
- Z punktu widzenia funkcji, której argumentem jest referencja do stałej, nie jest możliwe rozróżnienie pomiędzy aktualną r-wartością a zwykłym obiektem przekazanym referencję.
- Argumenty r-referencyjne umożliwiają pewnym obiektom na stworzenie semantyki przenoszenia za pomocą konstruktorów przenoszących oraz przypisań przenoszących.

Argumenty tymczasowe

- Przykład (1):

```
class Simple {
  void *Memory; // The resource
public:
  Simple() {
    Memory = nullptr; }
  // the MOVE-CONSTRUCTOR
  Simple(Simple&& sObj) {
    // Take ownership
    Memory = sObj.Memory;
    // Detach ownership
    sObj.Memory = nullptr; }
  Simple(int nBytes) {
    Memory = new char[nBytes]; }
  ~Simple() {
    if(Memory != nullptr) delete[] Memory; }
};
```

Argumenty tymczasowe

- Przykład (2):

```
Simple GetSimple() {
  Simple sObj(10);
  return sObj; }

// R-Value NON-CONST reference
void SetSimple(Simple&& rSimple) {
  // performing memory assignment here
  Simple object;
  object.Memory = rSimple.Memory;
  rSimple.Memory = nullptr;
  // Use object...
  delete[] object.Memory; }
```

Konstruktor kopiujący

- **Konstruktor kopiujący** służy do utworzenia obiektu, który będzie kopią innego już istniejącego obiektu.
- Konstruktorem kopiującym jest konstruktor w klasie `Klasa`, który można wywołać z jednym argumentem typu:
  `Klasa::Klasa (const Klasa&);`
- `Klasa::Klasa (Klasa&);`
  Wywołanie konstruktora może nastąpić:
  - w sposób jawny:
    ```
    Klasa wzor;
    //…
    Klasa nowy = wzor;
    ```
  - niejawnie, gdy wywołujemy funkcję z argumentem danej klasy przekazywanym przez wartość;
  - niejawnie, gdy wywołana funkcja zwraca wartość w postaci obiektu danej klasy.

Konstruktor kopiujący

- Jeśli programista nie zdefiniuje konstruktora kopiującego to może wygenerować go kompilator (wtedy wszystkie pola są kopiowane za pomocą operatora przypisania).
- Kompilator nie wygeneruje konstruktora kopiującego, jeśli dla pewnego pola w klasie nie będzie można zastosować operatora przypisania = (na przykład do pola stałego).

Przypisanie kopiujące

- **Przypisanie kopiujące** służy do skopiowania do obiektu danych z innego obiektu.
- Przypisanie kopiujące w klasie `Klasa` może być zdefiniowane z jednym (prawym względem `=`) argumentem (drugim argumentem, lewym względem `=`, jest bieżący obiekt):
```
Klasa & Klasa::operator= (Klasa &);
Klasa & Klasa::operator= (const Klasa &);
```
- Przypisanie kopiujące powinno zwrócić referencję do bieżącego obiektu (aby umożliwić kaskadowe wykorzystanie operatora `=`).
- Jeśli programista nie zdefiniuje przypisania kopiującego to może wygenerować go kompilator (wtedy wszystkie pola są kopiowane za pomocą operatora przypisania).

Konstruktor kopiujący i przypisanie kopiujące

- Programista może zablokować automatyczne wygenerowanie konstruktora kopiującego czy przypisania kopiującego w klasie za pomocą frazy **`=delete`** umieszczonej na końcu deklaracji.
- Przykład zablokowania konstruktora kopiującego i przypisania kopiującego, który zostałby wygenerowany przez kompilator:

```
punkt(const punkt&) = delete;
```

Stały wskaźnik i wskaźnik do stałej

- Wskaźnik do stałej pokazuje na obiekt, którego nie można modyfikować. Przykład:

```
int a=7, b=5;
const int *p = &a;
// *p = 12; to jest błąd
p = &b; // ok
```

- Stały wskaźnik zawsze pokazuje na ten sam obiekt. Przykład:

```
int a=13, b=11;
int *const p = &a;
*p = 12; // ok
// p = &b; to jest błąd
```

- Można również zdefiniować stały wskaźnik do stałej. Przykład:

```
int c=23;
const int *const p = &c;
```

Pola stałe w klasie

- W klasie można zdefiniować pola stałe z deklaratorem `const`. Przykład:

```
class zakres {
     const int MIN, MAX;
public:
     zakres(int mi, int ma);
     // …
};
```

- Inicjalizacji pola stałego (i nie tylko stałego) można dokonać tylko poprzez **listę inicjalizacyjną** w konstruktorze (po dwukropku za nagłówkiem). Przykład:

```
zakres::zakres(int mi, int ma) : MIN(mi), MAX(ma) {
     if (MIN<0||MIN>=MAX)
          throw string("złe zakresy");
}
```

Inicjalizacja pól na liście ma postać konstruktorową.

Stałe funkcje składowe

- W klasie można zadeklarować stałe funkcje składowe z deklaratorem `const`. Przykład:

```
class zakres {
    const int MIN, MAX;
public:
    int min () const;
    int max () const;
    // …
};
```

- Stała funkcja składowa gwarantuje nam, że nie będzie modyfikować żadnych pól w obiekcie (nie zmieni stanu obiektu). Przykład:

```
int zakres::min () const { return MIN; }
int zakres::max () const { return MAX; }
```

- Na obiektach stałych możemy działać tylko stałymi funkcjami składowymi.

Pola zawsze modyfikowalne

- Jeśli obiekt zostanie zadeklarowany jako stały, to można na nim wywoływać tylko stałe funkcje składowe, które nie zmieniają stanu obiektu.
- W klasie można jednak zdefiniować zawsze modyfikowalne pola składowe za pomocą deklaratora `mutable`. Przykład:

```
class zakres
{
    mutable int wsp;
public:
    void nowyWsp (int w) const;
    // …
};
```

- Pole zawsze modyfikowalne może być zmieniane w stałym obiekcie przez stałą funkcję składową. Przykład:

```
void zakres::nowyWsp (int w) const
{
    if (w<0||w<wsp/2||w>wsp*2)
        throw string("zły współczynnik");
    wsp = w;
}
```

Ulotne funkcje składowe

- W klasie można również zadeklarować ulotne funkcje składowe z deklaratorem `volatile`. Przykład:

```
class licznik {
    volatile int ile;
public:
    int ilosc() volatile;
    // …
};
```

- Ulotna funkcja składowa gwarantuje nam, że nie będzie optymalizować kodu przy korzystaniu z pól w obiekcie (nie przechowywać stanu obiektu w podręcznej pamięci). Przykład:

```
int licznik::ilosc() volatile {
    return ile;
}
```

- Na obiektach ulotnych możemy działać tylko ulotnymi funkcjami składowymi.