KURS JĘZYKA C++

2. KLASY I OBIEKTY

SPIS TREŚCI

PROGRAMOWANIE OBIEKTOWE

- **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*).
- Programowanie obiektowe opiera się na czterech paradygmatach:
  - abstrakcja,
  - hermetyzacja,
  - dziedziczenie,
  - polimorfizm.

ABSTRAKCJA

- **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

- **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 obiektowy to zbiór deklaracji i definicji klas.
- Klasa jest modelem (projektem) a obiekt jest instancją klasy (realizacją projektu).
- Klasa posiada zestaw różnych pól i metod:
  - wartości pól w obiekcie określają stan obiektu.
- Obiekt posiada własne pola ale wspólne dla wszystkich obiektów są funkcje składowe (metody):
  - metody pracują na rzecz konkretnego obiektu za pomocą niejawnie przekazanego wskaźnika do obiektu źródłowego.

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`) z wykorzystaniem hermetyzacji:

```
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);
};
```

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();
```

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.
- Konstruktor można przeciążyć.
- 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;
```

KONSTRUKTORY DELEGATOWE

- **Konstruktor delegatowy** wywołuje inny konstruktor do zainicjalizowania obiektu.
- Wywołanie konstruktora właściwego w konstruktorze delegatowym następuje na liście inicjalizacyjnej (jest to jedyne wywołanie na liście inicjalizacyjnej):
  `K::K(...) : K(...) { ... }`
- Treść konstruktora delegatowego pracuje na zainicjalizowanym już obiekcie.

KONSTRUKTORY DELEGATOWE

- Wywołanie innych równorzędnych konstruktorów, zwanych delegacjami, umożliwia wykorzystanie cech innego konstruktora za pomocą niewielkiego dodatku kodu.
- Przykład:

```
class SomeType {
   int number;
public:
   SomeType (int num) : number(num) {}
   SomeType () : SomeType(45) {}
// …
};
```

KONSTRUKTORY DELEGATOWE

- W C++ obiekt jest skonstruowany, jeśli dowolny konstruktor zakończy swoje działanie.
- Jeśli wielokrotne wykonywanie konstruktorów jest dozwolone, to znaczy, że każdy konstruktor delegatowy będzie wykonywany na już skonstruowanym obiekcie.
- Konstruktory klas pochodnych będą wywołane wtedy, gdy wszystkie konstruktory delegatowe ich klas bazowych będą zakończone.

DESTRUKTOR

- Destruktor to specjalna metoda uruchamiana tuż przed likwidacją 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();
```

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

WSKAŹNIK `THIS`

- Wskaźnik `this` jest ukrytym parametrem każdej instancyjnej funkcji składowej.
- Wskaźnik `this` pokazuje na bieżący obiekt.
- Wskaźnika tego używany tylko w instancyjnych funkcjach składowych i w konstruktorach.
- 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.

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`;
  - zaleca się aby cała zawartość funkcji składała się tylko z instrukcji `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
```

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.

- Przypisanie kopiujące zostanie wygenerowane automatycznie tylko wtedy, gdy w klasie nie ma pól stałych.

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;
}
```

KOPIOWANIE OBIEKTÓW

- Kompilator automatycznie wygeneruje przypisanie kopiujące i konstruktor kopiujący dla klasy.
- Przykład:

```
class K { … };
…
K x;
K y(x); // konstruktor kopiujący
K z = x; // też konstruktor kopiujący
x = y; // przypisanie kopiujące
```

KOPIOWANIE OBIEKTÓW

- W definicji klasy można zablokować wygenerowanie przypisania kopiującego i konstruktora kopiującego dla klasy przez użycie frazy `delete`.
- Przykład:

```
class K {
//…
public:
    K(const K &k) = delete;
    K& operator = (const K &k) = delete;
//…
};
```