Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 7

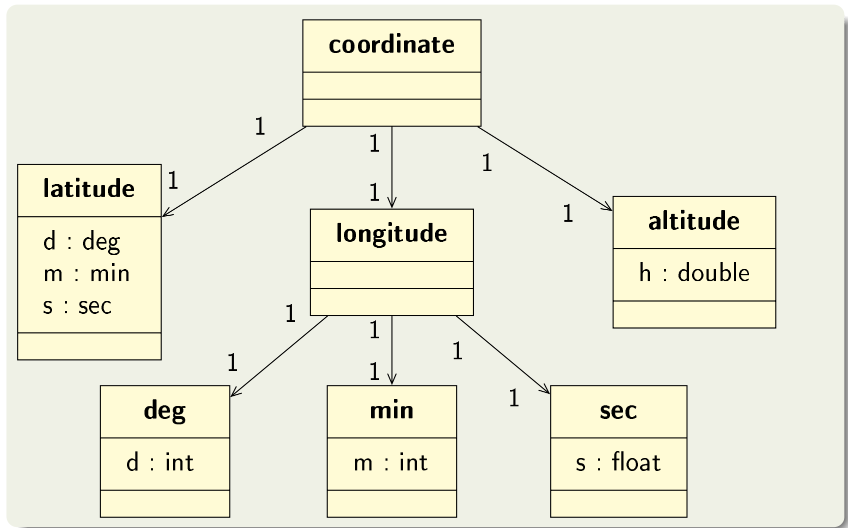
29 listopada 2018

Treść dzisiejszego wykładu

- rodzaje agregacji
- dziedziczenie
- metody i pola w klasach pochodnych
- metody wirtualne
- tworzenie i niszczenie obiektów

agregacja

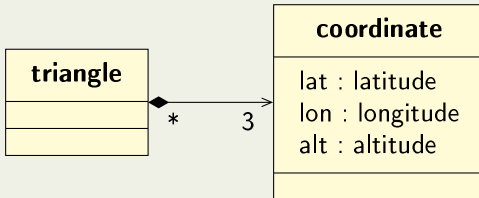
- Nowa klasa może być stworzona z innych klas poprzez przechowywanie obiektów (wskaźników referencji) innych typów.
- Nowe klasy mogą być zdefiniowane z użyciem dowolnej liczby klas już zdefiniowanych.
- Agregacja jest relacją typu **składa się z** lub **zawiera**.
- Szczególnych przypadkiem agregacji jest kompozycja
- Kompozycja jest relacją typu **posiada**


coordinate
latitude
d : deg
m : min
s : sec
longitude
altitude
h : double
deg
d : int
min
m : int
sec
s : float
1
1
1
1
1
1
1
1
1
1
1
1

agregacja

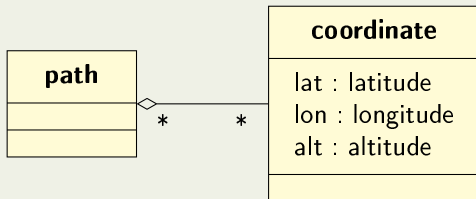
```
struct deg { int d; };

struct min { int m; };

struct sec { float s; };

struct latitude {
    deg d;
    min m;
    sec s;
};

struct longitude {
    deg d;
    min m;
    sec s;
};

struct altitude {
    float h
};

struct coordinate {
    latitude lat;
    longitude lon;
    altitude alt;
};
```

agregacja




```
#include <memory>
struct triangle{
    std::shared_ptr<coordinate> c1;
    std::shared_ptr<coordinate> c2;
    std::shared_ptr<coordinate> c3;
};
```

```
#include <array>
#include <memory>
struct triangle{
    std::array<std::shared_ptr<coordinate>, 3> coordinates;
};
```



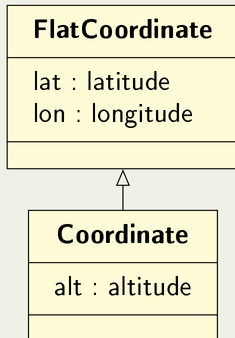

```cpp
#include <vector>
#include <memory>

struct path{
    std::vector<std::shared_ptr<coordinate>> coordinates;
};
```

dziedziczenie

- Dziedziczenie to rodzaj relacji pomiędzy dwoma klasami
- W relacji dziedziczenia wyodrębniamy klasę bazową oraz klasę pochodną
- Klasa pochodna **jest** specjalizacją bardziej ogólnej klasy bazowej
- Obiekty klasy pochodnej **mogą być** traktowane jako obiekty klasy bazowej.
- Dziedziczenie jest relacją typu **jest**
- Może być również wykorzystywane z szablonami (dlaczego?)

Składnia

```
class /*struct*/ KlasaPochodna : /*typ dziedziczenia*/ ListaKlasBazowy
{
    /*definicja klasy*/
};
```

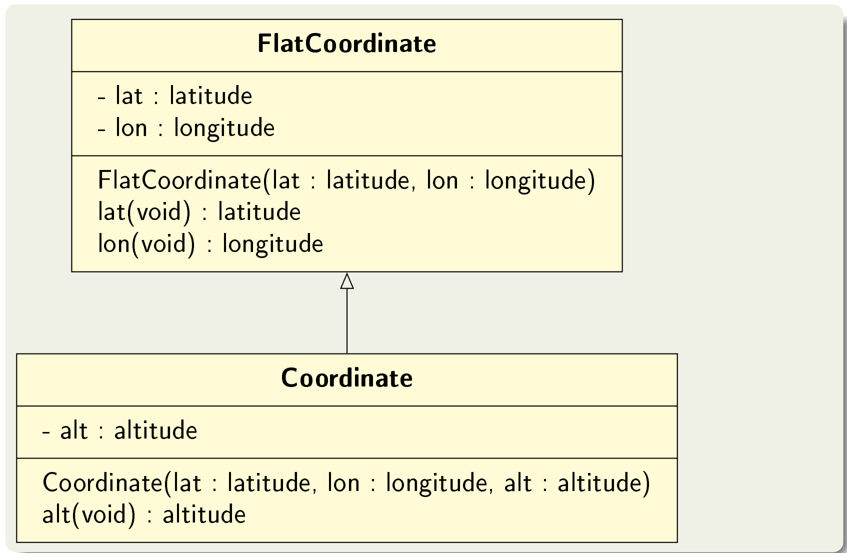



```
struct FlatCoordinate
{
    latitude lat;
    longitude lon;
};

struct Coordinate
    : public FlatCoordinate
{
    altitude alt;
};
```

dziedziczenie

```
struct latitude
{
    double lat;
    latitude(double lat_arg) : lat(lat_arg) {}
};

struct longitude
{
    double lon;
    longitude(double lon_arg) : lon(lon_arg) {}
};

struct altitude
{
    double alt;
    altitude(double alt_arg) : alt(alt_arg) {}
};
```

Co jest niepokojącego w powyższym kodzie?


FlatCoordinate
- lat : latitude
- lon : longitude
FlatCoordinate(lat : latitude, lon : longitude)
lat(void) : latitude
lon(void) : longitude
Coordinate
- alt : altitude
Coordinate(lat : latitude, lon : longitude, alt : altitude)
alt(void) : altitude

```
class FlatCoordinate
{
    latitude  lat_;
    longitude lon_;
public:

    latitude  lat() const { return lat_; }
    longitude lon() const { return lon_; }

    FlatCoordinate( const latitude& lat,
                    const longitude& lon)
        : lat_(lat), lon_(lon) {}
};

class Coordinate : public FlatCoordinate
{
    altitude alt_;
public:
    altitude alt() const { return alt_; }

    Coordinate( const latitude& lat,
                const longitude& lon,
                const altitude& alt)
       : FlatCoordinate(lat,lon), alt_(alt) {}
};
```

```
int main(void)
{
    FlatCoordinate c1({ 52.219164 }, { 21.012487 });
    Coordinate c2({ 52.219164 }, { 21.012487 }, { 100.0 });

    std::cout << c1.lat() << "," << c1.lon() << std::endl;
    std::cout << c2.lat() << "," << c2.lon() << "," << c2.alt() << std::endl;



    getchar();
    return 0;
}
```

Dlaczego argumenty wywołania konstruktorów podane zostały w nawiasach klamrowych?

dziedziczenie

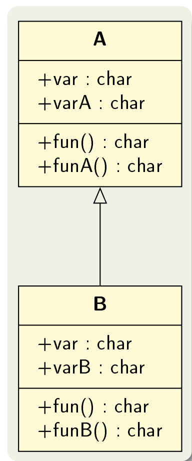
```
std::ostream& operator<<(std::ostream& o, const FlatCoordinate& c)
{
    o << c.lat() << "," << c.lon();
    return o;
}
std::ostream& operator<<(std::ostream& o, const Coordinate& c)
{
    o << static_cast<const FlatCoordinate&> (c) << "," << c.alt();
    return o;
}

int main(void)
{
    FlatCoordinate c1({ 52.219164 }, { 21.012487 });
    Coordinate c2({ 52.219164 }, { 21.012487 }, { 100.0 });



    std::cout << c1 << std::endl;
    std::cout << c2 << std::endl;

    getchar();
    return 0;
}
```

Dlaczego argumenty wywołania konstruktorów podane zostały w nawiasach klamrowych?

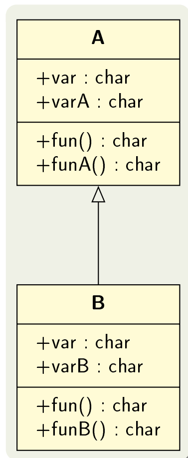
metody i pola w klasach pochodnych

- klasa pochodna ma dostęp do wszystkich publicznych metod i pól klasy bazowej.
- o ile klasa nie dziedziczy w sposób prywatny to wszystkie metody i pola klasy bazowej są dostępne dla użytkowników klasy pochodnej
- klasa pochodna może redefiniować (przesłaniać) metody i pola klasy bazowej
- istnieje możliwość bezpośredniego odwołania się do metody klasy bazowej podając nazwę tej klasy przed nazwą metody

metody i pola w klasach pochodnych

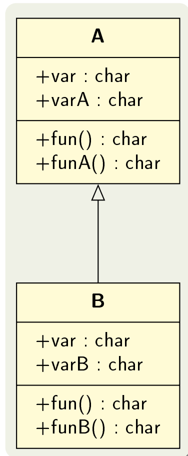


metody i pola w klasach pochodnych




```
struct A
{
    char var = 'a';
    char varA = 'a';

    char fun(void) const
    { return 'a'; }

    char funA(void) const
    { return 'a'; }
};

struct B : public A
{
    char var = 'b';
    char varB = 'b';

    char fun(void) const
    { return 'b'; }

    char funB(void) const
    { return 'b'; }
};
```

metody i pola w klasach pochodnych




```cpp
struct A
{
    char var = 'a';
    char varA = 'a';

    char fun(void) const
    { return 'a'; }

    char funA(void) const
    { return 'a'; }
};

struct B : public A
{
    char var = 'b';
    char varB = 'b';

    char fun(void) const
    { return 'b'; }

    char funB(void) const
    { return 'b'; }
};
```

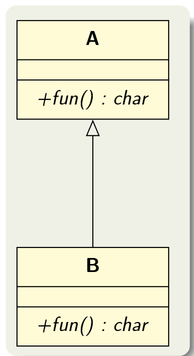
```cpp
int main(void){
    A a;
    std::cout << a.varA;

    B b;
    std::cout << b.varB;
    std::cout << b.varA;
    std::cout << b.var;
    std::cout << b.A::var;
    std::cout << b.B::var;

    std::cout << a.fun();
    std::cout << b.fun();
    std::cout << b.funA();
    std::cout << b.funB();
    std::cout << b.A::fun();
    std::cout << b.B::fun();

    A& ra = b;
    std::cout << ra.fun();

    return 0;
}
```

metody i pola w klasach pochodnych

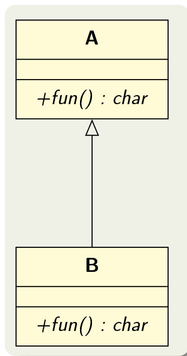
Niestatyczne metody składowe klasy poprzedzone słowem kluczowym `virtual` są tzw. metodami wirtualnymi (polimorficznymi).

Metoda zdeklarowana w klasie bazowej jako wirtualna pozostaje wirtualną w klasie pochodnej gdy ich prototypy są tożsame (ta sama nazwa, ten sam typ wartości zwracanej, ta sama lista parametrów). Jeżeli w klasie bazowej (lub pośredniej) metoda wirtualna jest zdefiniowana klasa pochodna nie musi jej definiować - jeżeli nie ma takiej potrzeby.

Gdy wywołanie metody wirtualnej dokonuje się poprzez użycie wskazania na obiekt lub referencji to wywoływana jest wersja zdefiniowana w klasie bazowej albo w klasie pochodnej zgodnie z typem pełnego obiektu wskazywanego. Wybór metody następuje w czasie wykonywania programu.

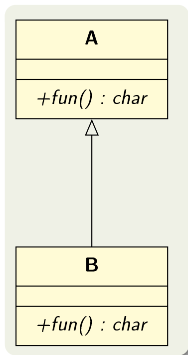

A
+fun() : char
B
+fun() : char

metody i pola w klasach pochodnych



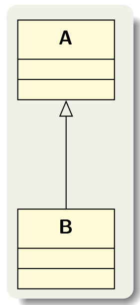

```
struct A
{
    virtual char fun(void) const { return 'a'; }
};

struct B : public A
{
    /*virtual*/char fun(void) const { return 'b'; }
};
```

metody i pola w klasach pochodnych

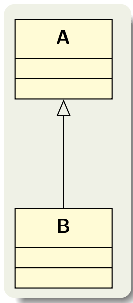



```
struct A
{
    virtual char fun(void) const { return 'a'; }
};

struct B : public A
{
    /*virtual*/char fun(void) const { return 'b'; }
};
```

```
int main(void){
    A a;
    B b;

    std::cout << a.fun() << std::endl;
    std::cout << b.fun() << std::endl;

    A& ra = b;
    std::cout << ra.fun() << std::endl;

    return 0;
}
```

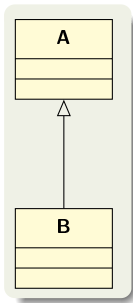

A
B

tworzenie i niszczenie obiektów




```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

struct A
{
    A(void) {
        print_info("A␣c-tor");
    }
    ~A(void) {
        print_info("A␣d-tor");
    }
};

struct B : public A
{
    B(void) {
        print_info("B␣c-tor");
    }
    ~B(void) {
        print_info("B␣d-tor");
    }
};
```

tworzenie i niszczenie obiektów

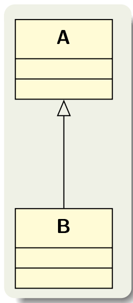



```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

struct A
{
    A(void) {
        print_info("A c-tor");
    }
    ~A(void) {
        print_info("A d-tor");
    }
};

struct B : public A
{
    B(void) {
        print_info("B c-tor");
    }
    ~B(void) {
        print_info("B d-tor");
    }
};
```

```
int main(void){
    A a;
    B b;
    A& ra = b;

    return 0;
}
```

```
int main(void){
    A* pa = new A;
    delete pa;

    B* pb = new B;
    delete pb;

    pa = new B;
    delete pa;

    return 0;
}
```

tworzenie i niszczenie obiektów



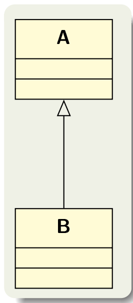

```cpp
void print_info(const char* info) {
    std::cout << info << std::endl;
}

struct A
{
    A(void) {
        print_info("A c-tor");
    }
    ~A(void) {
        print_info("A d-tor");
    }
};

struct B : public A
{
    B(void) {
        print_info("B c-tor");
    }
    ~B(void) {
        print_info("B d-tor");
    }
};
```

```cpp
int main(void){
    A a;
    B b;
    A& ra = b;

    return 0;
}
```

```cpp
int main(void){
    A* pa = new A;
    delete pa;

    B* pb = new B;
    delete pb;

    pa = new B;
    delete pa;

    return 0;
}
```

W jaki sposób prawidłowo usuwać obiekty klas pochodnych?

tworzenie i niszczenie obiektów

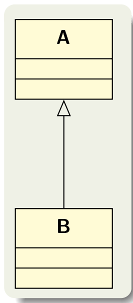



```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

struct A
{
    A(void) {
        print_info("A c-tor");
    }
    virtual ~A(void) {
        print_info("A d-tor");
    }
};

struct B : public A
{
    B(void) {
        print_info("B c-tor");
    }
    virtual ~B(void) {
        print_info("B d-tor");
    }
};
```

```
int main(void){
    A a;
    B b;
    A& ra = b;

    return 0;
}
```

Należy zdefiniować destruktor jako metodę wirtualną.

tworzenie i niszczenie obiektów

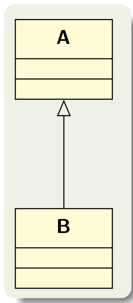



```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

struct A
{
    A(void) {
        print_info("A c-tor");
    }
    virtual ~A(void) {
        print_info("A d-tor");
    }
};

struct B : public A
{
    B(void) {
        print_info("B c-tor");
    }
    virtual ~B(void) {
        print_info("B d-tor");
    }
};
```

```
int main(void){
    A a;
    B b;
    A& ra = b;

    return 0;
}
```

Należy zdefiniować destruktor jako metodę wirtualną.

tworzenie i niszczenie obiektów



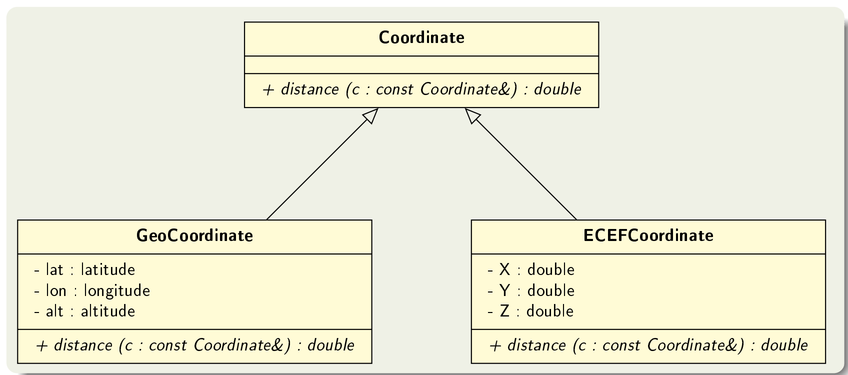

```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

struct A
{
    A(void) {
        print_info("A␣c−tor");
    }
    virtual ~A(void) {
        print_info("A␣d−tor");
    }
};

struct B : public A
{
    B(void) {
        print_info("B␣c−tor");
    }
    virtual ~B(void) {
        print_info("B␣d−tor");
    }
};
```

```
int main(void){
    A a;
    B b;
    A& ra = b;

    return 0;
}
```

```
int main(void){
    A* pa = new A;
    delete pa;

    B* pb = new B;
    delete pb;

    pa = new B;
    delete pa;

    return 0;
}
```

Należy zdefiniować destruktor jako metodę wirtualną.

dziedziczenie - przykład



dziedziczenie - przykład

```
struct Coordinate
{
public:
    virtual double distance(const Coordinate& c) const;
};

struct GeoCoordinate : public Coordinate
{
    latitude  lat;
    longitude lon;
    altitude  alt;

    /*virtual*/ double distance(const Coordinate& c) const override;
};

struct ECEFCoordinate : public Coordinate
{
    double x;
    double y;
    double z;
    /*virtual*/ double distance(const Coordinate& c) const override;
};
```