Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 9

3 grudnia 2020

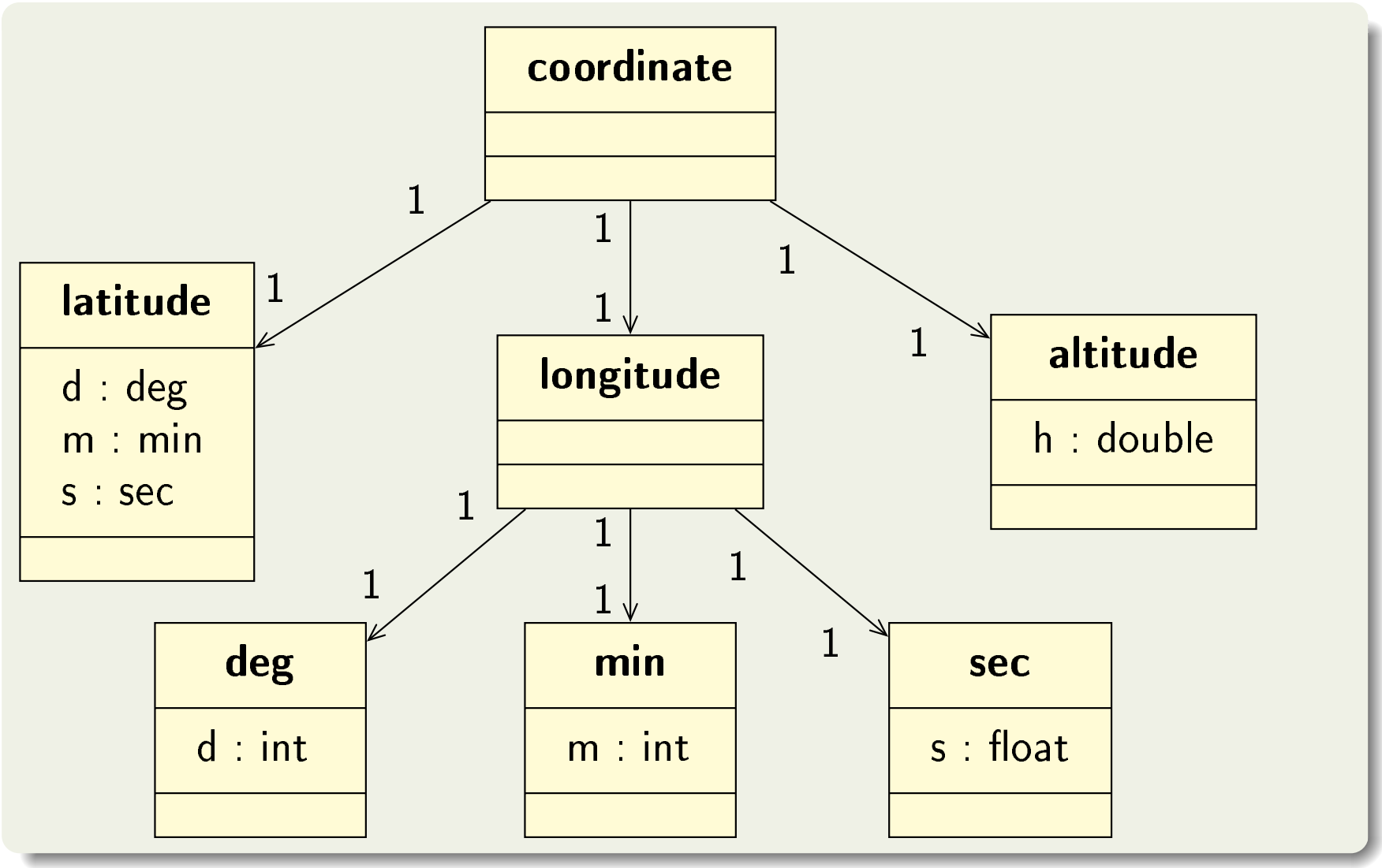
Treść dzisiejszego wykładu

- agregacja i kompozycja - uzupełnienie informacji
- polimorfizm dynamiczny
- dziedziczenie
- metody i pola w klasach pochodnych
- kwalifikatory dostępu
- metody wirtualne
- tworzenie i niszczenie obiektów

agregacja i kompozycja

- 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**

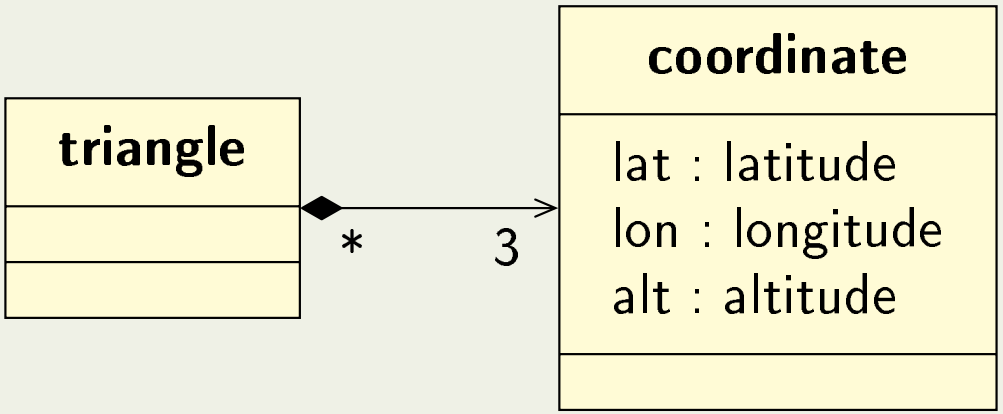
agregacja i kompozycja



agregacja i kompozycja

```
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;
};
```

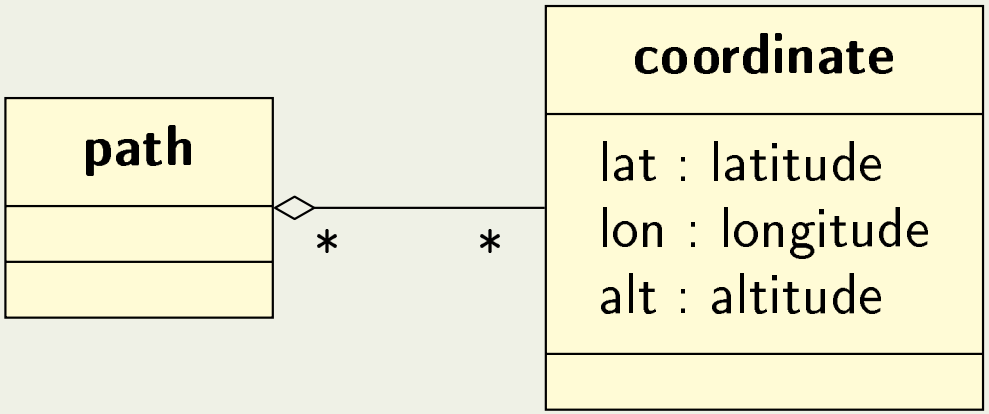
agregacja i kompozycja




```
#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;
};
```

agregacja i kompozycja




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

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

polimorfizm dynamiczny

Polimorfizm jest mechanizmem w programowaniu zorientowanym obiektowo pozwalającym danemu obiektowi na różne zachowanie w zależności od bieżacego wykonania programu.
Mechanizm ten dostępny jest nie tylko w językach ściśle obiektowych, ale języki te silnie go wspierają.
Polimorfizm jest jednym z czterech podstawowych założeń programowania obiektowego.

Filary peogramowania obiektowego:

- **abstrakcja**
- **hermetyzacja**
- ***polimorfizm***
- *dziedziczenie*

polimorfizm dynamiczny

Język C++ wspiera obie formy polimorfizmu:

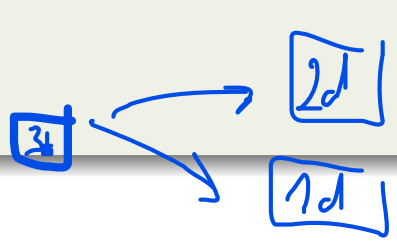
- polimorfizm statyczny
  - przeładowywanie funkcji
  - przeładowywanie metod
  - przeładowywanie operatorów
  - szablony
- polimorfizm dynamiczny
  - metody wirtualne
  - dziedziczenie

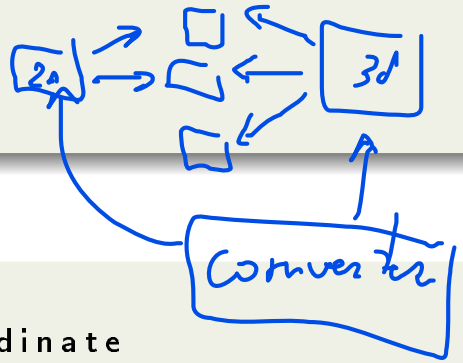
dziedziczenie

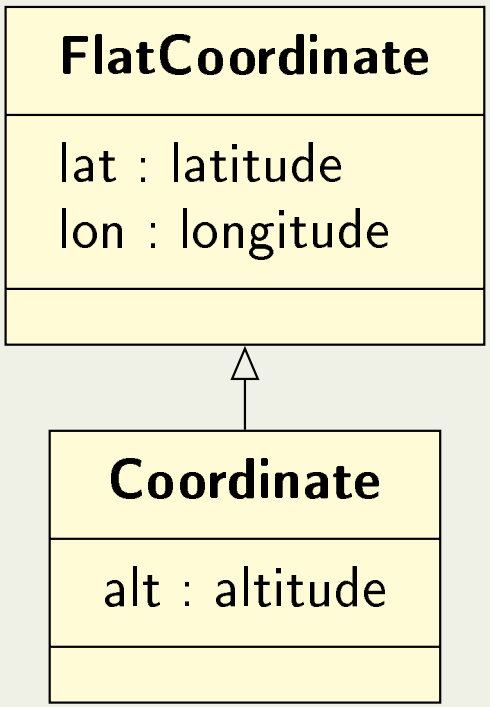
- Dziedziczenie to rodzaj relacji pomiędzy dwoma klasami
- W relacji dziedziczenia wyodrębniamy klasę bazową oraz klasę pochodną
- Klasa pochodna **jest** specjalizacją bardziej ogólnej klasy bazowej
- Obiekty klasy pochodnej **mogą być** traktowane jako obiekty klasy bazowej.
- Dziedziczenie jest relacją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

Składnia

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





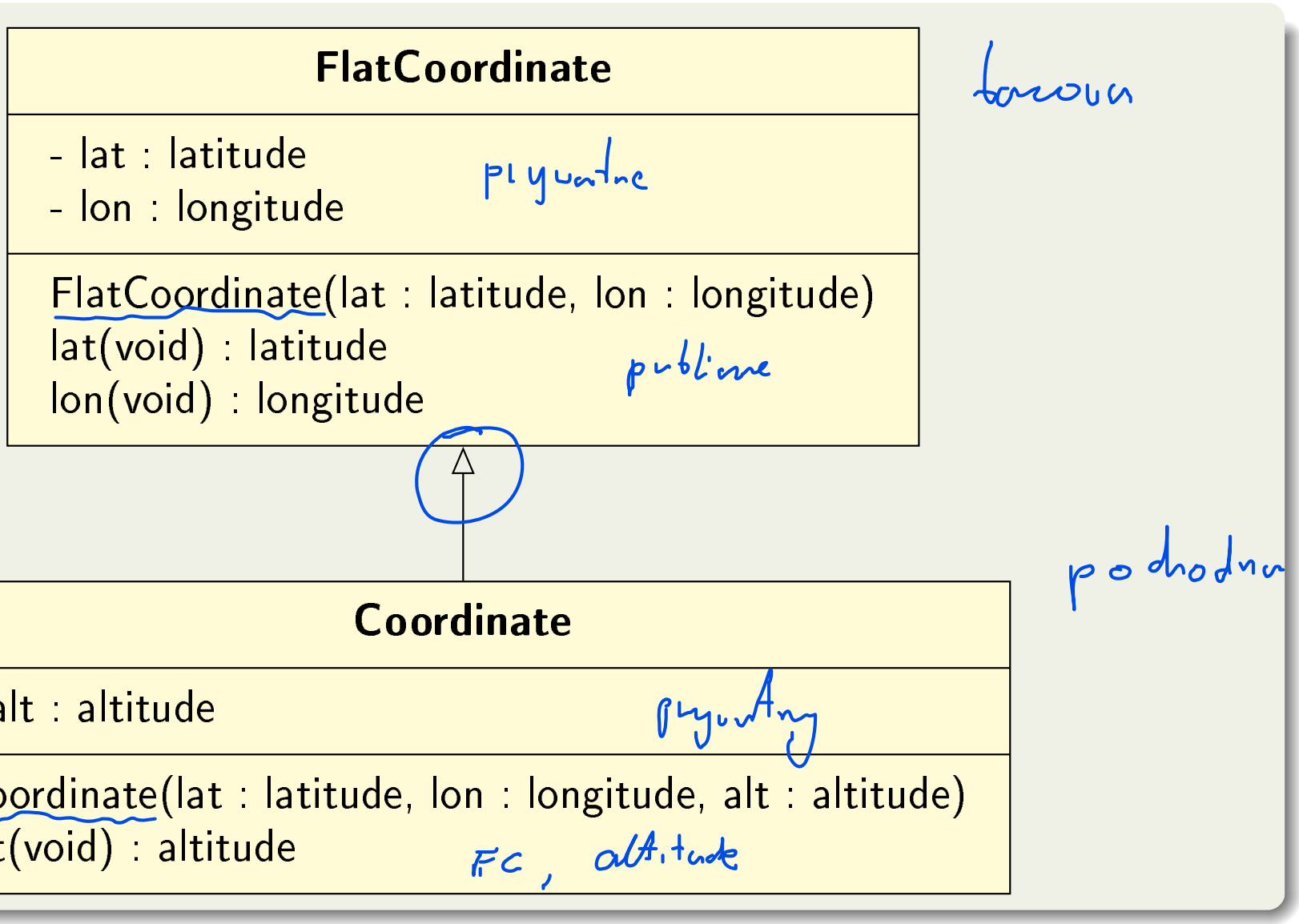



```
struct FlatCoordinate
{
    latitude lat;
    longitude lon;
};

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

dziedziczenie

```
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) {}
};
```

bicie konstruktora domyślnego

1. 2.

```
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;
}
```

automatyczna konwersja

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?

kwalifikatory dostępu

```
class A
{
public:
    void fun() {}
protected:
    void pfun() {}
private:
    void _fun() {}
    int var_;
};
```

A
-var_ : int
+fun() : void #pfun() : void -_fun() : void

- w zakresie `public` umieszcza się metody i pola klasy dostępne dla wszystkich użytkowników; na diagramie UML oznaczane przez +
- w zakresie `protected` umieszcza się metody i pola klasy, które mają być widoczne jedynie w klasach pochodnych; na diagramie UML oznaczane przez #
- składowe w zakresie chronionym w klasie bazowej dostępne są dla pozostałych składowych tej klasy, funkcji i klas zaprzyjaźnionych z klasą bazową, składowych klas dziedziczących po klasie bazowej oraz funkcji i klas zaprzyjaźnionych z klasami dziedziczącymi po klasie bazowej.
- w zakresie `private` umieszcza się metody i pola klasy dostępne jedynie dla metod tej klasy; na diagramie UML oznaczane przez -

typy dziedziczenia

- istnieją 3 typy dziedziczenia : `public`, `protected`, `private`
- w dziedziczeniu publicznym kwalifikatory dostępu w klasie pochodnej jest taki sam jak w klasie bazowej dla wszystkich jej składowych
- w dziedziczeniu chronionym kwalifikatory dostępu metod i pół publicznych zmienia się na chroniony (`protected`); pozostałe bez zmian
- w dziedziczeniu prywatnym wszystkie metody i pola klasy bazowej stają się prywatne w klasie pochodnej

Przy użyciu relacji dziedziczeni nie można osłabić ochrony dostępu do składowych klasy bazowej a jedynie można ją wzmocnić.

typy dziedziczenia

Gdy tryb dziedziczenia po klasie bazowej wzmacnia ochronę danej składowej, to można zmienić zakres ochrony umieszczając deklarację `using` nazwa_klasy_bazowej::nazwa_składowej we właściwej sekcji klasy pochodnej.

```
class A
{
public:
    void fun1() {}
    void fun2() {}
    int var;
};
```

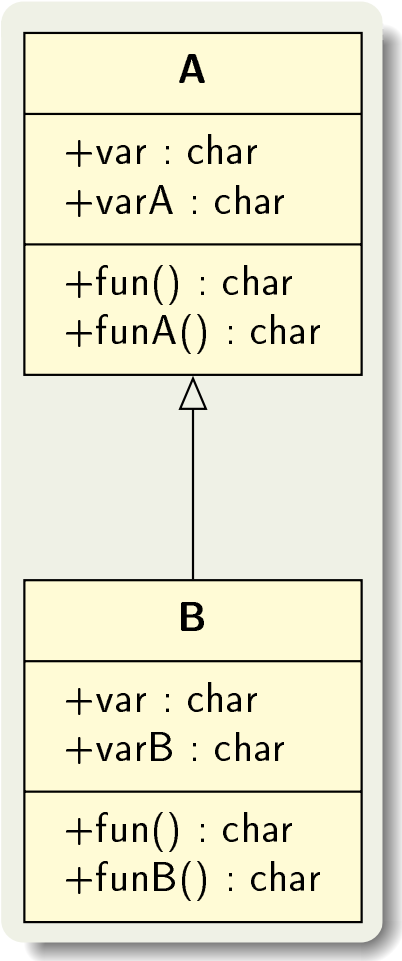
```
class B
    : private A
{
public:
    using A::fun1;
    using A::var;
};
```

```
int main(void)
{
    B().fun1();
    //B().fun2();
    B().var;

    return 0;
}
```

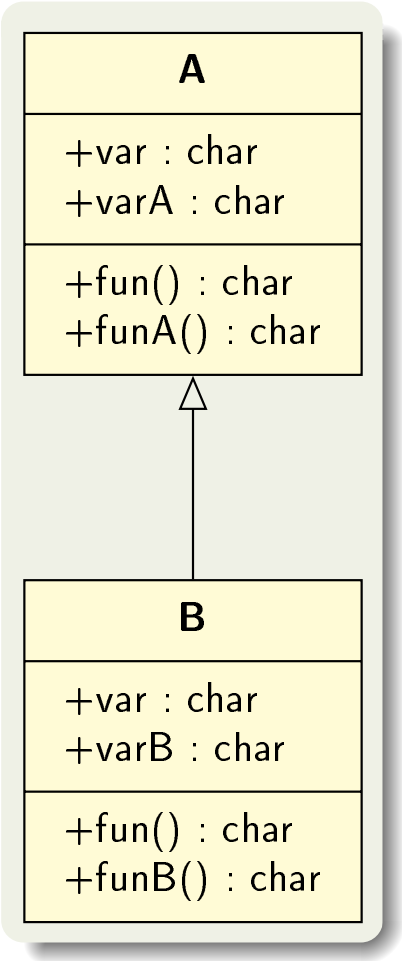
metody i pola w klasach pochodnych

- 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

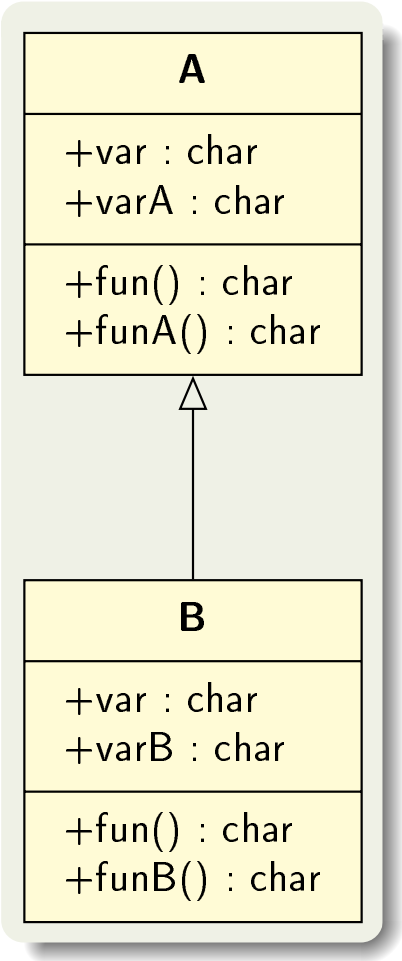


metody i pola w klasach pochodnych




```
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




```
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'; }
};
```

```
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;
}
```

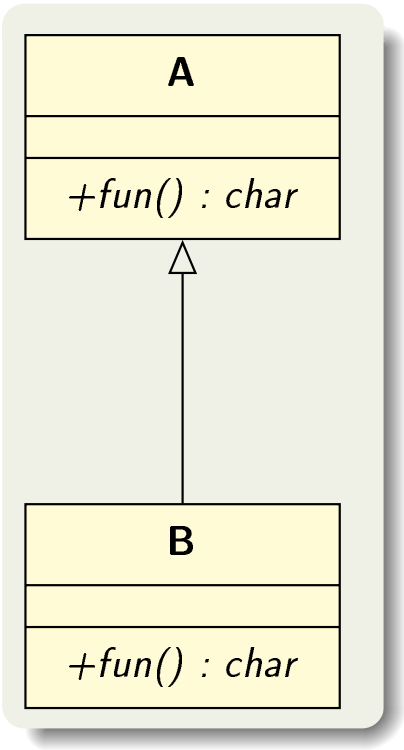
metody i pola w klasach pochodnych

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

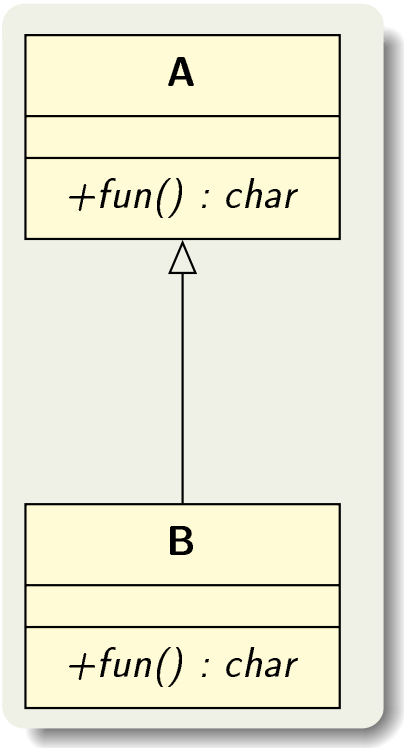
Metoda zdeklarowana w klasie bazowej jako wirtualna pozostaje wirtualną w klasie pochodnej gdy ich prototypy są tożsame (ta sama nazwa, ten sam typ wartości zwracanej, ta sama lista parametrów). Jeżeli w klasie bazowej (lub pośredniej) metoda wirtualna jest zdefiniowana klasa pochodna nie musi jej definiować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

metody i pola w klasach pochodnych



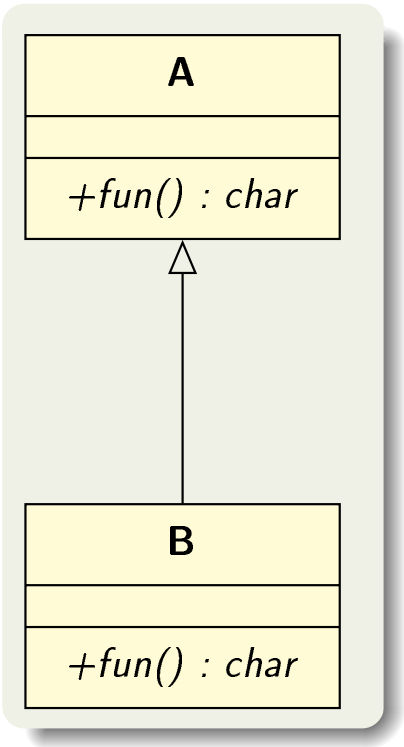
metody i pola w klasach pochodnych




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

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

nie jest konieczne

metody i pola w klasach pochodnych

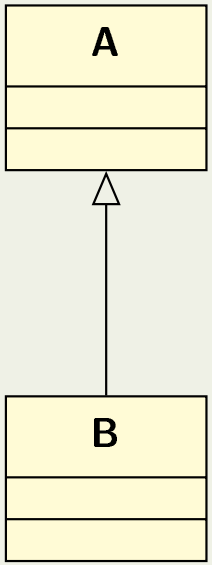



```
struct A
{
    virtual char fun(void) const  { return 'a'; }
};

struct B : public A
{
    char fun(void) const override { 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
b

tworzenie i niszczenie obiektów

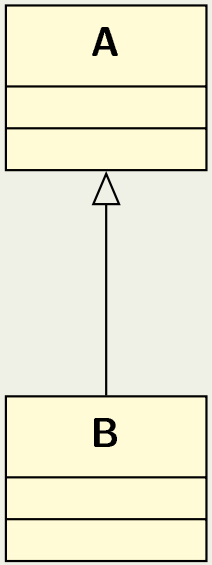



```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

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

tworzenie i niszczenie obiektów



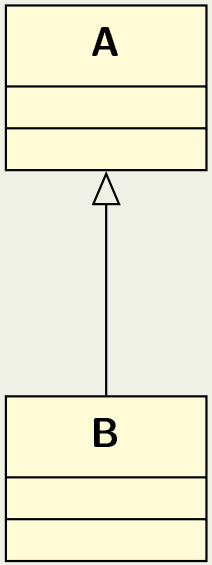

```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

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;
}
```

W jaki sposób prawidłowo usuwać obiekty klas pochodnych?

tworzenie i niszczenie obiektów



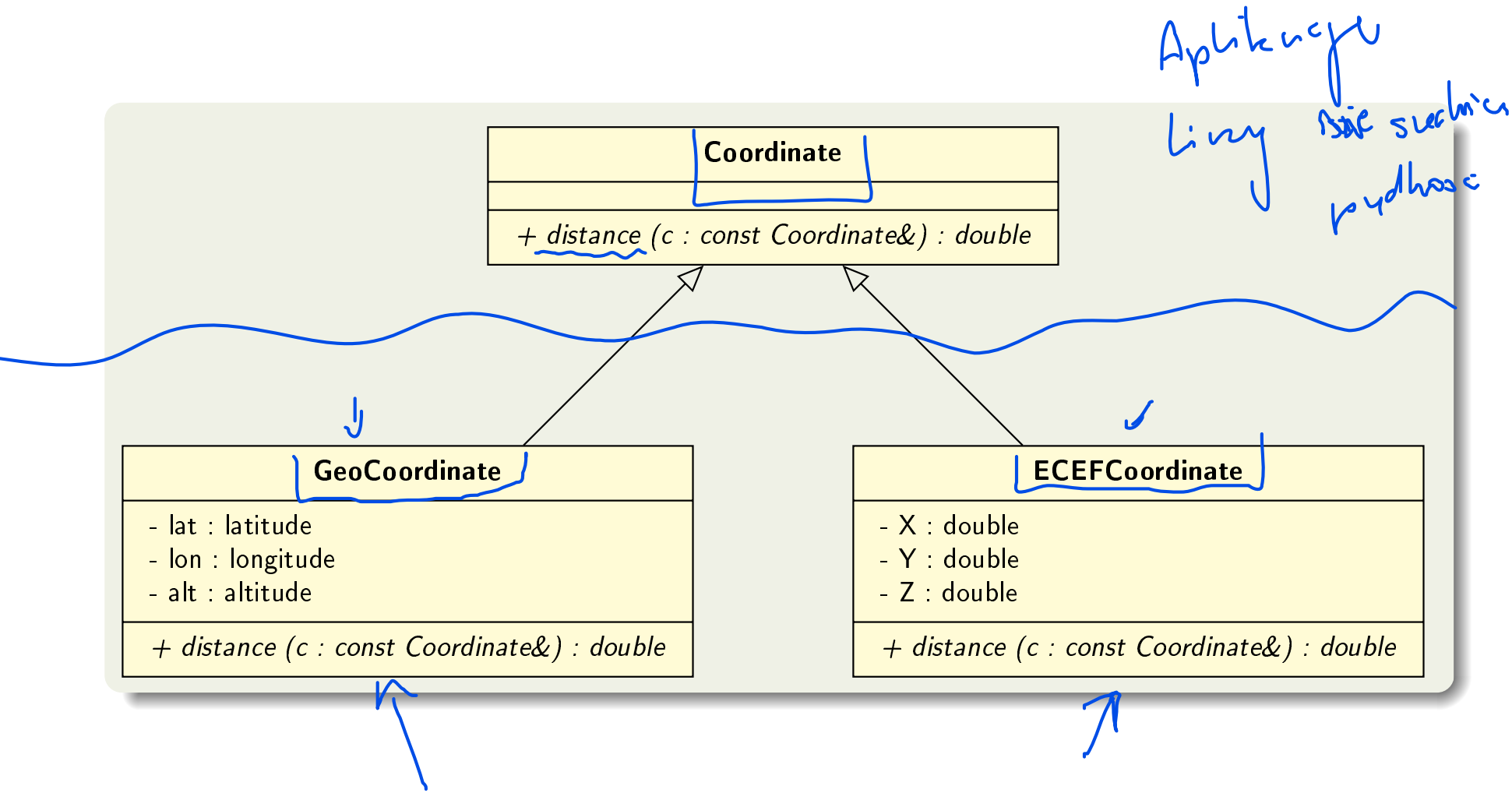

```
void print_info(const char* info) {
    std::cout << info << std::endl;
}

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;
}
```

```
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;

    double distance(const Coordinate& c) const override;
};

struct ECEFCoordinate : public Coordinate
{
    double x;
    double y;
    double z;

    double distance(const Coordinate& c) const override;
};
```