

ADATSZERKEZETEK ÉS ALGORITMUSOK

C++ OOP

Osztálydefiníció

- A C++ **struct** és **class** sok szempontból hasonló
 - OOP elvárások tekintetében a **class** használata szükséges
- Példa egy egyszerű osztály definíciójára
 - Osztálydefiníciók egymásba ágyazhatók

```
class Rectangle {  
    double a_side, b_side;  
public:  
    void print() const { }  
};
```

- Példányosítás

```
Rectangle rect(10, 2);  
Rectangle *rectpt = &rect;
```

Konstruktorok

- Konstruktorok: „nincs objektum konstruktor nélkül”, ha kell, implicit hívódnak
 - Neve megegyezik az osztály nevével
 - Ha már megadtunk egy konstruktort, akkor default konstruktor nem definiálódik
 - A default konstruktor meghívja az attribútumok konstruktorát, de a beépített típusokat nem inicializálja (konzisztensen a C-vel)
 - A konstruktornak nem lehet visszatérési értéke
 - C++11 óta lehetséges
 - Mezők inicializálása
 - Konstruktor delegáció
 - Szülő osztály konstruktorának örökítése

Konstruktorok

```
class Rectangle {  
    double a_side, b_side;  
public:  
    Rectangle(double a, double b): a_side(a), b_side(b) {};  
    Rectangle() : Rectangle(1.0, 1.0) {};  
  
    void print() const {  
        std::cout << "Rectangle " << a_side << " " <<  
                    b_side << std::endl;  
    }  
  
    double calculateArea() {  
        return a_side * b_side;  
    }  
};
```

Destruktor

- Téglalap esetén nincs kifejezett haszna, de egy általános sokszögnél (illetve példa az inicializációra)

```
class Polygon {  
    double *sides {nullptr};  
    int nsides {0};  
  
public:  
    Polygon(double *data, int ndata): nsides(ndata) {  
        sides=new double[nsides];  
        std::memcpy(sides, data, sizeof sides);  
    }  
    ~Polygon() { delete [] sides; }  
  
    void print() const {  
        std::cout << "Polygon, number of sides " << nsides  
                    << std::endl;  
    }  
};
```

Osztályváltozó, osztálymetódus

```
class Circle {  
    double radius;  
    static double PI;  
public:  
    Circle(double r): radius(r) {};  
    Circle() : Circle(1.0) {};  
  
    void print() const {  
        std::cout << "Circle " << radius << std::endl;  
    }  
};
```

```
double Circle::PI = 3.1415926535;
```

- Az osztályon kívül definiálni kell!

Osztályváltozó, osztálymetódus

```
class Circle {  
    //...  
    double calculateArea() {  
        return radius * radius * PI;  
    }  
    //...  
};
```

- Az osztályon belül elérhető közvetlenül, itt a külső használathoz publikus metódus kell!

```
static double getPi() { return Circle::PI; }
```

Osztályváltozó, osztálymetódus

- Használat:

```
Circle::PI; // HIBA  
Circle::getPi();  
Circle c1(10);  
c1.getPi();
```

- Szemantikailag helytelen példányon keresztül elérni, azaz:

- ~~c1.getPi();~~
- ~~return radius * radius * PI;~~

- Helyette

- **return** radius * radius * Circle::PI;

Láthatóság

- Az adattagok és metódusok elrejtése megoldott
- A láthatóság minősítője C++-ban
 - **public**
 - a külső felhasználók elérik
 - **protected**
 - csak a leszármazottak érhetik el
 - **private**
 - csak az adott osztály és a barátok (**friend**) számára elérhető – alapértelmezés

Öröklődés

- A téglalap egy specializációja a négyzet:

```
class Square: public Rectangle {  
public:  
    Square(double a): Rectangle(a, a) { };  
    Square(): Square(1) { };  
  
    void print() const {  
        std::cout << "Square " << a_side << std::endl;  
    }  
};
```

- Azonban ez nem működik, mert ugyan van saját `a_side` attribútuma a négyzetnek, de azt nem éri el közvetlenül, mivel az ősből ez **private**.

Öröklődés

Módosítsuk!

```
class Rectangle {  
protected:  
    double a_side, b_side;  
    //...  
};
```

Ekkor

```
rect.print();    // Rectangle 10 2  
Square s(5);  
s.print();       // Square 5
```

Konstruktor öröklés

- Lehetséges C++11 óta

```
class BaseClass {  
public:  
    BaseClass(int value) {};  
};
```

```
class DerivedClass : public BaseClass {  
public:  
    using BaseClass::BaseClass;  
};
```

Leszármazás (felüldefiniálás)

A osztály, benne két publikus példánymetódus.

B osztály az A leszármazottja, a metódust felüldefiniálja.

A B osztályban további publikus függvényt vezetünk be.

A egy példánya a A* a = new A()

B egy példánya b B* b = new B()

Ekkor:

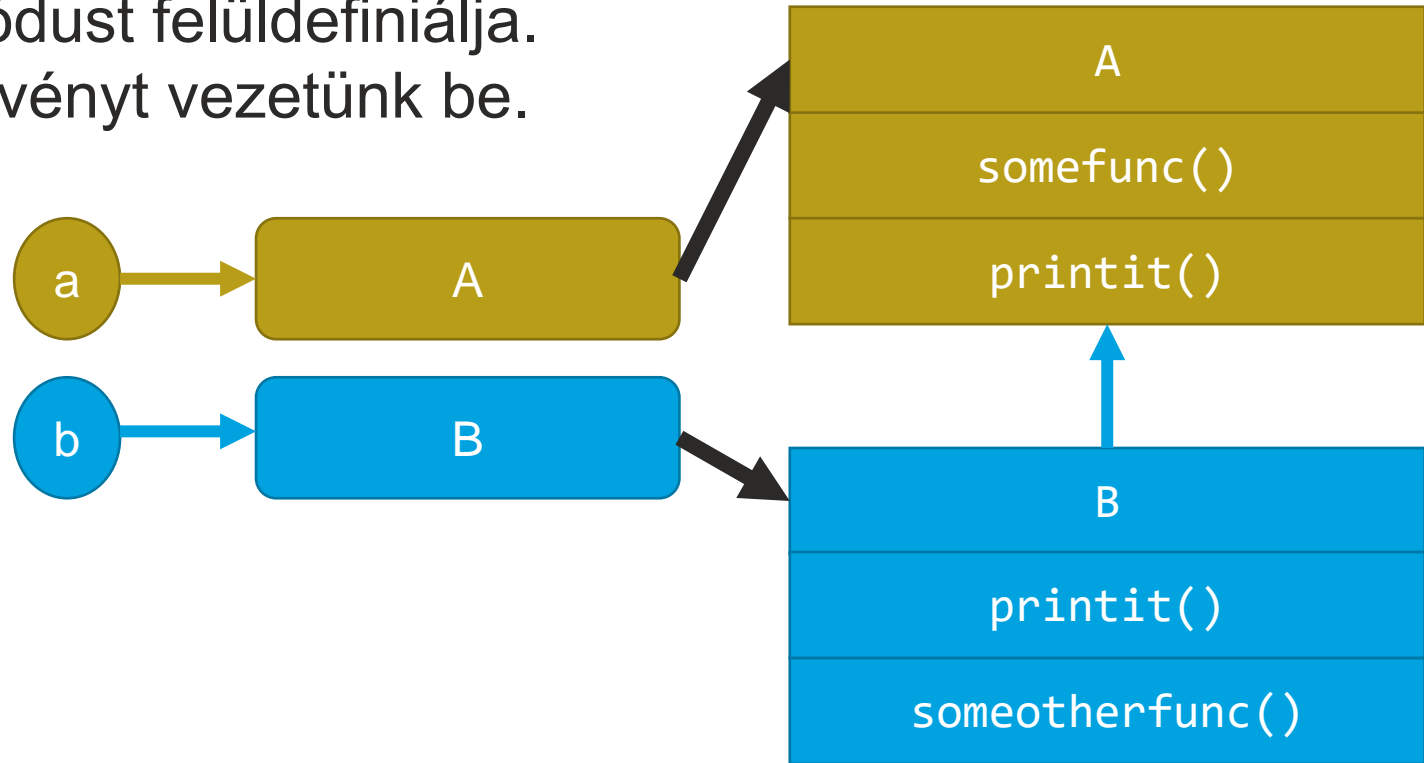
`a.printit()`

`b.printit()`

`a.somefunc()`

`b.somefunc()`

~~`a.someotherfunc()`~~



Altípusos polimorfizmus

- Tekintsük az alábbi kódot

```
Rectangle rect(10, 2);
```

```
Square s(5);
```

```
rect = s;
```

- Mi történik?

```
rect.print();
```

- Fontos, hogy itt még nincsen dinamikus kötés!

Altípusos polimorfizmus

- Ha B (pl. `Square`) altípusa az A (pl. `Rectangle`) típusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciáinak.
- Azaz:

```
Rectangle* prect = new Rectangle(4, 3);  
Square* psqu = new Square(2);
```

```
prect->print();  
psqu->print();  
prect = psqu;
```

Mi történik?

```
prect->print();
```

Altípusos polizmorfizmus

Altípusos polimorfizmus esetén az a változóval lehetséges hivatkozni a B példányára
Itt a polimorfizmusra láttunk példát

Azaz A típusú változó B típusú objektumra hivatkozik.
Dinamikus kötés nem jött létre.

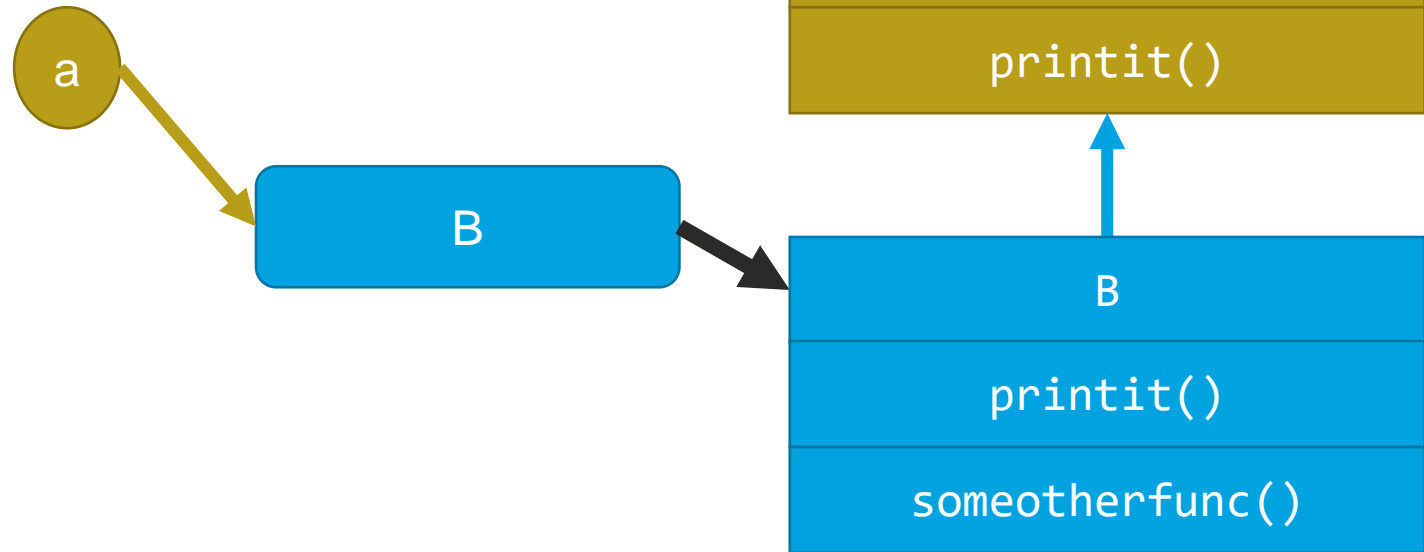
Polimorfizmus: A* a = new B()

Ekkor:

`a.printit()`

`a.somefunc()`

~~`a.someotherfunc()`~~



Dinamikus összekapcsolás

- C++ nyelven a dinamikus kötéshez:
 - Az **őstípus** **virtual**-nak deklarálja a metódust, amire megengedi a felüldefiniálást
 - Ellenkező esetben elfedés lesz!
 - Tipikusan az ősbeli **virtual**-t a leszármazottban is **virtual**-nak jelöljük.

Dinamikus összekapcsolás

- Módosítás a `Rectangle` osztályban

```
virtual void print() const {  
    std::cout << "Rectangle " << a_side << " "  
    << b_side << std::endl;  
}
```

- Ekkor az előző példa

```
prect->print();  
// Square 2
```

Felüldefiniálás jelzése

- Expliciten jelezhetjük, hogy felüldefiniálást szeretnénk

```
class Square: public Rectangle {  
    //...  
    virtual void print() const override {//...  
};
```

- Ellenőrizni fogja a fordító ekkor
- Hibát kapunk ha felüldefiniálandó függvény
 - nem **virtual**
 - **final**
- Felüldefinálás (és leszármazás) megakadályozása
virtual void print() **const final** {//...

Dinamikus kötés

Dinamikus kötés megvalósulása esetén az objektum típusa (nem a rá hivatkozó változó típusa) szerinti implementáció hajtódik végre.

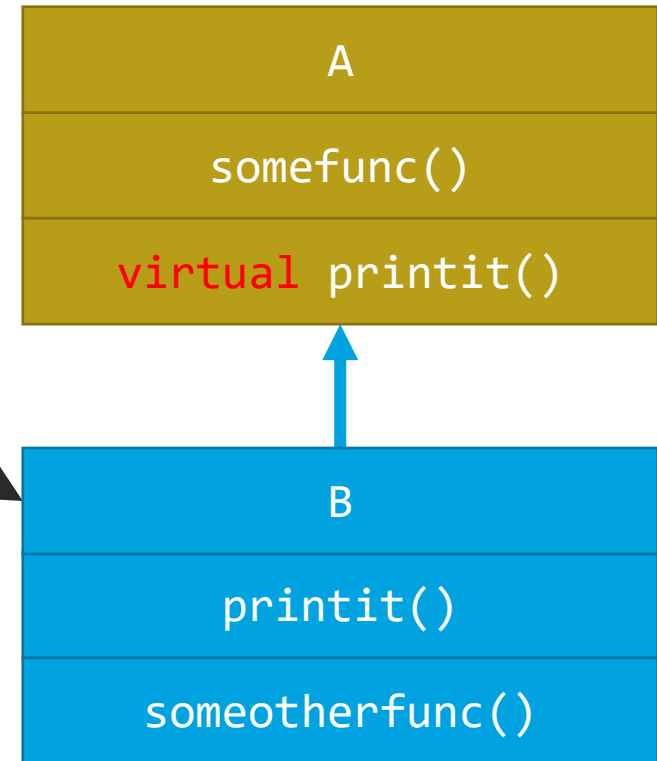
C++ esetén a **virtual** kulcsszót kell használni!

Polimorfizmus +
dinamikus kötés:

`A* a = new B()`

a

B



Ekkor:

`a.printit()`

`a.somefunc()`

~~`a.someotherfunc()`~~

A változó típusa (statikus típus) határozza meg, hogy milyen műveletek hívhatók!
Az objektum típusa (dinamikus típus) határozza meg a végrehajtandó implementációt!

Virtuális destruktork

- A destruktork esetén is fontos a **virtual** használata
 - Amennyiben az altípusos polimorfizmus használjuk.
- Nézzük az alakzatot

```
class Shape {  
public:  
    virtual void print() const {  
        std::cout << "Generic Shape" << std::endl;  
    }  
};
```

Virtuális destruktork

- Ha a **Polygon** ebből származik le

```
class Polygon: public Shape {  
    //...  
    ~Polygon() {  
        delete [] sides;  
        std::cout<< "Deleted" << std::endl;  
    }  
};
```

- Akkor a következő eset memóriaszivárgás

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);  
delete sh;  
std::cout << "^^ Memory leak ^^ " << std::endl;
```

Virtuális destruktork

- Helyesen

```
class Shape {  
public:  
    virtual ~Shape() {};  
    virtual void print() const {  
        std::cout << "Generic Shape" << std::endl;  
    }  
};
```

- Így már rendben lesz

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);  
delete sh;  
std::cout << "^^ No memory leak ^^ " << std::endl;
```

Automatikusan létrehozott függvények

- A fordító több tagfüggvényt létrehoz, ha nem definiáljuk azokat.
Dönthetünk az automatikusan létrejövő tagfüggvények

- létrehozásáról (**default**)
- elutasításáról (**delete**)

```
class Square {  
public:  
    Square() = default;  
    Square(const Square&) = default;  
    Square& operator=(const Square&) = delete;  
    virtual ~Square() = default;  
};
```

- Az utolsó sornál fontos a **virtual**.
 - Ezzel jelezzük, hogy automatikusan létrejövő destruktork virtuális is legyen!

Absztrakt osztály

Pure virtual függvények bevezetése

```
class Shape {  
    public:  
        virtual ~Shape() {};  
        virtual void print() = 0;  
        virtual double calculateArea() = 0;  
};
```

Absztrakt osztály

- Ilyenkor a leszármazott adja meg a definíciót:

```
class Circle : public Shape {  
    double radius;  
public:  
    Circle(double r): radius(r) {};  
  
    virtual void print() const override {  
        std::cout << "Circle radius: " << radius << std::endl;  
    }  
    virtual double calculateArea() const override {  
        return radius * radius * PI;  
    }  
};
```

Absztrakt osztály típusként

- Természetesen nem példányosítható:

```
Shape s;
```

- De statikus típusa egy változónak lehet:

```
Circle c;
```

```
Shape * pa = &c;
```

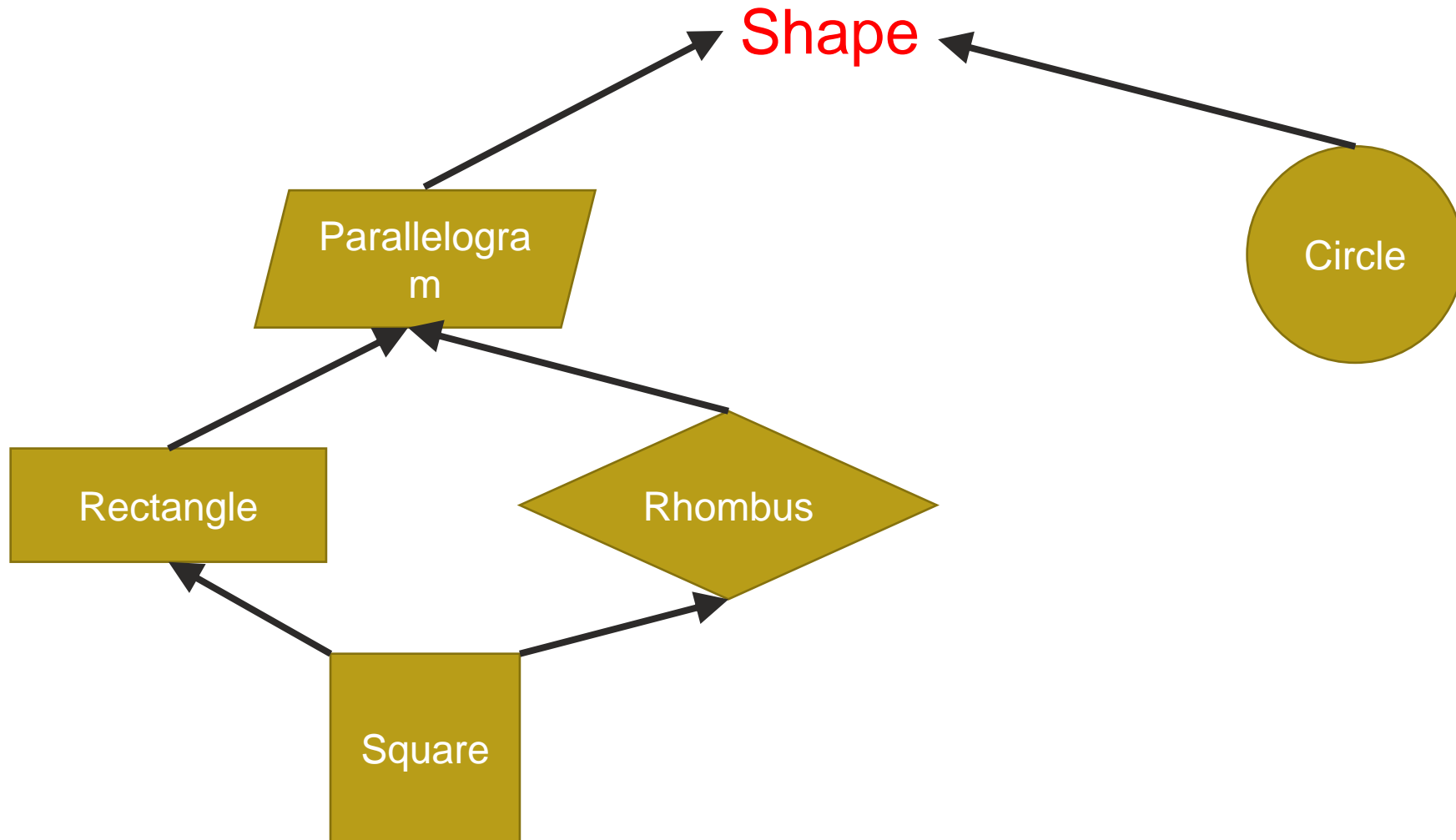
- De statikus típusa egy változónak lehet és dinamikus kötés működik

```
pa->calculateArea(); // Circle::calculateArea()
```

Többszörös öröklés

- Az osztályhierarchia lehet fa, de lehet általánosabb körmentes gráf is:
 - `class Shape`
 - `class Circle: public Shape`
 - `class Parallelogram: public Shape`
 - `class Rhombus: public Parallelogram`
 - `class Rectangle: public Parallelogram`
 - `class Square: public Parallelogram, public Rhombus`
- Problémák:
 - Adattagok hányszor?
 - Melyik metódus?

Többszörös öröklés



Többszörös öröklés

- `double PI = 3.1415926535;`

```
class Shape {
public:
    virtual ~Shape() {};
    virtual void print() const = 0;
    virtual double calculateArea() const = 0;
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {};

    virtual void print() const override {
        std::cout << "Circle radius: " << radius << std::endl;
    }
    virtual double calculateArea() const override {
        return radius * radius * PI;
    }
};
```

Többszörös öröklés

- ```
class Parallelogram : public Shape {
protected:
 double _side_a;
 double _side_b;
 double _delta;
 double _height_a;
public:
 virtual double get_a_side() const final { return _side_a; }
 virtual double get_b_side() const final { return _side_b; }
 virtual double get_angle() const { return _delta; }
 virtual bool is_equilateral() const { return _side_a == _side_b; }
 virtual bool is_right() const { return _delta == 90; }

 Parallelogram(double a, double b, double angle): _side_a(a), _side_b(b), _delta(angle) {
 _height_a = b * std::sin(_delta*PI / 180.0);
 }

 virtual void print() const override {
 std::cout << "Parallelogram a, b, delta: " << _side_a << ", "
 << _side_b << ", "
 << _delta << std::endl;
 }

 virtual double calculateArea() const override {
 return _side_a * _height_a;
 }
};
```

# Többszörös öröklés

```
• class Rhombus : public Parallelogram {
public:
 Rhombus(double a, double angle): Parallelogram(a, a, angle) { };

 virtual void print() const override {
 std::cout << "Rhombus side, angle " << _side_a << ", " << _delta << std::endl;
 }

 virtual double calculateArea() const override {
 return _side_a * _side_a * std::sin(_delta*PI / 180.0);
 }

 virtual bool is_equilateral() const override final { return true; }
};

class Rectangle : public Parallelogram {
public:
 Rectangle(double a, double b): Parallelogram(a, b, 90) { _height_a = b; };

 virtual void print() const override {
 std::cout << "Rectangle " << _side_a << ", " << _side_b << std::endl;
 }

 virtual bool is_right() const override final { return true; }
};
```



# Többszörös öröklés

```
• class Square: public Rectangle, public Rhombus {
 public:
 Square(double a): Rectangle(a,a), Rhombus(a, 90) { };

 virtual void print() const override final {
 std::cout << "Square " << _side_a << std::endl;
 }

 virtual double calculateArea() const override final {
 return _side_a * _side_a;
 }
};

int main() {
 Rectangle *s = new Square(10);
 std::cout << s->calculateArea() << std::endl;
 s->print();

 return 0;
}
```

# Többszörös öröklés

- Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító
  - A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen
- 1. Lehetséges megadni, például

```
virtual double calculateArea() const override final {
 return Rectangle::_side_a * Rectangle::_side_a;
}
```

  - Ez azonban nem oldja meg a következőt

```
Parallelogram *s = new Square(10);
```

# Többszörös öröklés

- Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító
  - A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen

- 2. Virtuális öröklés

- Ilyenkor a virtuális őszosztály nem két irányban lesz ős, hanem egy közös ős
- Gondoskodni kell a virtuális ősök inicializációjáról is

```
class Rhombus: public virtual Parallelogram
class Rectangle: public virtual Parallelogram
Square(double a): Parallelogram(a, a, 90),
 Rhombus(a, 90), Rectangle(a,a) { };
```

- Így már jó

```
Parallelogram *s = new Square(10);
```

# Többszörös öröklés függvény példa

```
class A{
public:
 virtual void f() { std::cout << "A.f()" << std::endl; };
};

class B : public A {
public:
 void f() { std::cout << "B.f()" << std::endl; };
};

class C : public A{
public:
 void f() { std::cout << "C.f()" << std::endl; };
};

class D : public B, public C {
public:
 void g() { std::cout << "D.g()" << std::endl; };
};
```

# Többszörös öröklés függvény példa

- Használatkor

- `D d;`  
`d.f();`
- Hibaüzenet: request for member „f” is ambiguous

- Itt is döntés kell

- `using C::f;`
- Azonban ilyenkor!
- `B *b = new D();`  
`b->f(); // B.f()`

# Friend

- A barát hozzáfér a privát adatokhoz, függvényekhez
  - Tipikusan olyan operátorok készítésére használjuk, amelyek a modellezés szerint nem részei a típusnak, de szorosan kötődnek hozzá
    - Kiírás

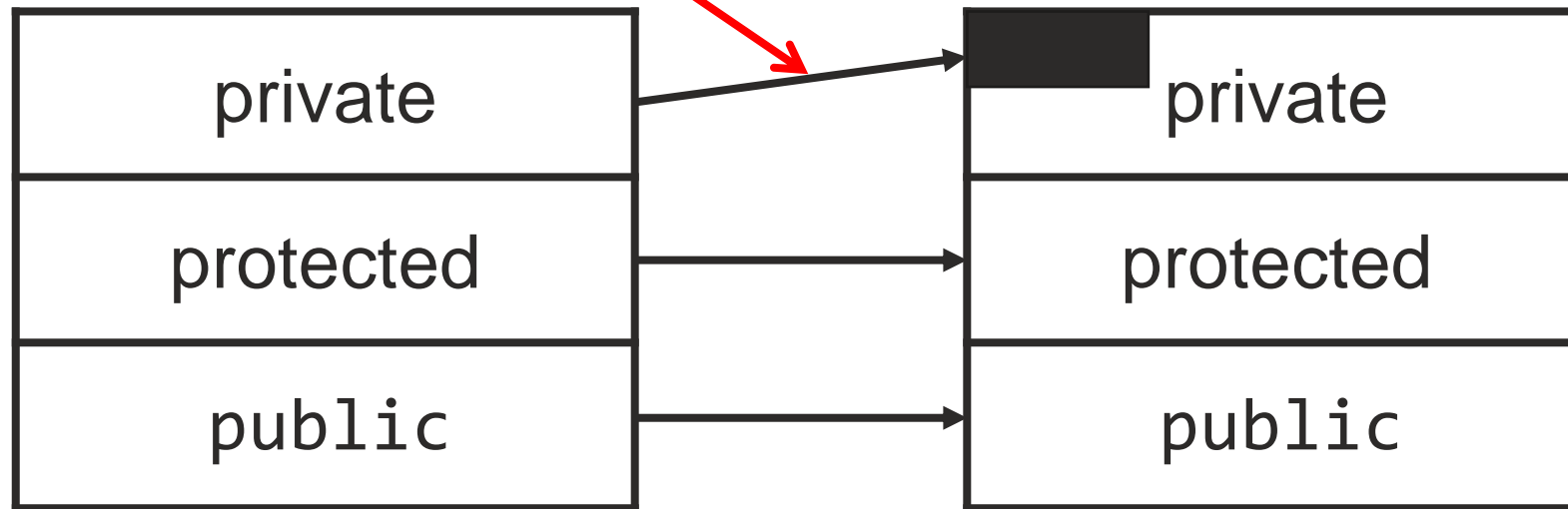
```
class Point {
 friend ostream &operator<<(ostream &, const Point &);
public:
 Point(int = 0, int = 0);
 void setPoint(int, int);
 int getX() const { return x; }
 int getY() const { return y; }
protected:
 int x, y;
};
```

# Alosztályképzés

- C++ - implementációs öröklés altípus létrehozása nélkül
  - **private**, **protected** öröklés

# Public öröklés

Tagok láthatóságának változása



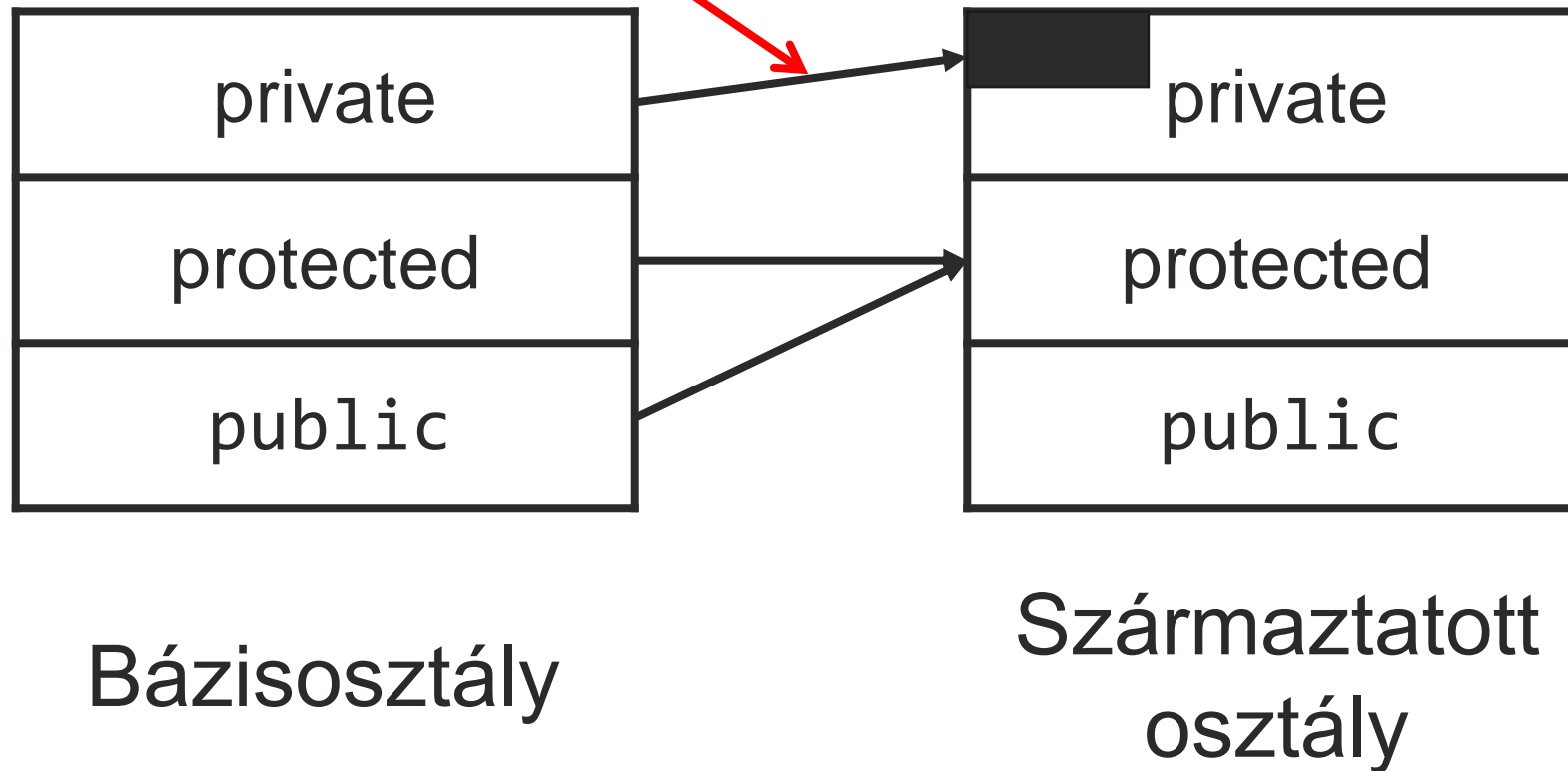
Ősosztály

Leszármazott



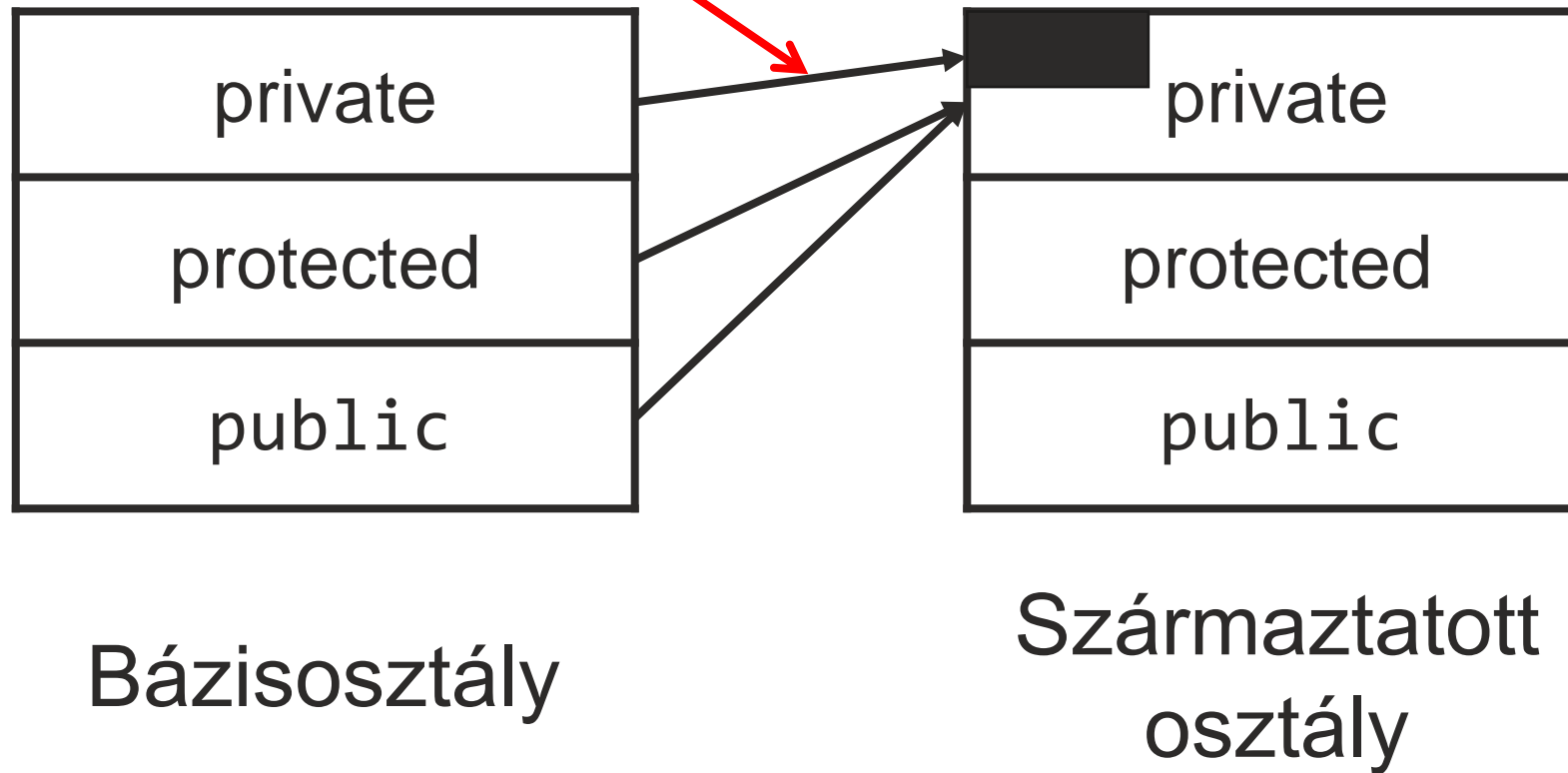
# Protected öröklés

Tagok láthatóságának változása



# Private öröklés

Tagok láthatóságának változása



# Kis kitérő – **struct**

- Struct esetén minden tag **public** alapértelmezés szerint.
  - Ez az információ elrejtést nem segíti, úgy mint az osztályoknál
  - Azonban célszerű „sima” adatrekordok tárolására (rekord típus)
    - Plain Old Data Structure (POD)
    - C-vel kompatibilis

# C++ – Összefoglalás

- Mit örököl a leszármazott?
  - Adattagokat
  - Metódusokat
- Mit nem örököl a leszármazott?
  - Ősosztály konstruktorait, destruktort
    - Konstruktor örökíthető
    - Leszármazottban delegálható rá, használható
  - Ősosztály értékadás operátorát
  - Ősosztály barátait

# C++ – Összefoglalás

- Mit vezethet be a leszármazott osztály?
  - Új adattagokat
  - Új metódusokat
  - Felüldefiniálhat már meglévőket (virtual)
  - Új konstruktorokat és destruktort
  - Új barátokat

# C++

- Egy általános metódus deklarációja a következőket jelenti:
  1. A metódus elérheti a privát mezőket is
  2. Az osztály scope-ját használja
  3. A metódus egy konkrét objektumra hívódik meg, ezért birtokolja a „this” pointert
- Statikus metódus csak az 1, 2 -vel rendelkezik,
- Ha egy függvényt friend-nek deklarálunk, akkor csak az 1. jogunk lesz (friend mechanizmus)

# OOP gyakorlatban

Következő téma