C++
Gyakorlat jegyzet
11. óra.

A jegyzetet Umann Kristóf készítette Brunner Tibor gyakorlatán. (2017. június 11.)

1. Objektum orientált programozás

Manapság az egyik legnépszerűbb programozási paradigma a objektum orientált programozás (*object oriented programming*, röviden *OOP*). Bár sokféleképpen definiálhatjuk, e három dolgot azonban általában megköveteljük egy objektum orientált kódtól:

- enkapszuláció, azaz az adatok egybefoglalása
- kód újrafelhasználás
- adat elrejtés

Fontos, hogy az objektum orientáltságot ne kössük feltétlenül az osztályokhoz, ezeket a követelményeket ugyanis ha körülményesen is, de nélkülük is meg tudjuk oldani. Példaképp, C-ben létrehozhatunk egy egyszerű rekordot, melynek deklarációját és a hozzá tartozó műveleteket eltároljuk egy header fájlban, magát a rekordot és a függvényeket csak egy fordítási egységben definiáljuk. Ezzel elértük az enkapszulációt, az adatok rejtve vannak, és egy ilyen rekordot el tudunk tárolni egy másik rekordban is könnyedén, ezzel megvalósítva a kód újrafelhasználást.
Így bátran kijelenthetjük, hogy a C nyelv egy olyan nyelv, mely támogatja az objektum orientált programozást. Azonban az is megállapítható, hogy ez nagyon nehézkes, és a legtöbb nyelv, melyre objektum orientáltként hivatkozunk (Pl. Java) sokkal erősebb eszközökkel rendelkeznek ennek támogatására.
Az objektum orientált C++ kódolásról nagyon sokat lehet mesélni, az elkövetkező néhány szekcióban áttekintjük valamennyi fontosabb részét, és hogy a fenti elveket hogyan tudjuk megvalósítani vele.

1.1. Öröklődés (*inheritance*)

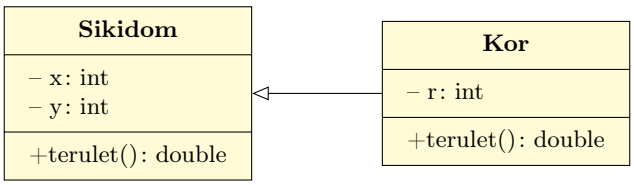
Próbáljunk megalkotni egy osztályt, mellyel egy általános síkidomot tudunk jellemezni!

```
class Sikidom
{
    int x, y; // a síkidom koordinátái
public:
    double terulet() const
    {
        // ???
    }
};
```

Az egyik legelső dolgunk legyen az, hogy írunk egy területszámító függvényt. Azonban egy általános síkidomnak nem tudjuk megadni csak így a területét. Esetleg ennek az típusnak létrehozhatnánk egy *altípusát*, mondjuk egy kört, amely rendelkezik a síkidom minden tulajdonságával, és még a körre vonatkozó adatokkal. Ehhez **öröklődést** fogunk alkalmazni:

```
class Kor : public Sikidom
{
    double r;
public:
    double terulet() const
    {
        return r * r * 3.14;
    }
};
```

Egy adott `struct`-ból illetve `class`-ból a fent látható módon tudunk örökölni: az új osztály neve után kettőspontot írunk, opcionálisan megadjuk a hozzáférés típusát, mely ebben az esetben public (erről nemsokára bővebben lesz szó), és az osztály nevét, melyből örökölni szeretnénk.
Az öröklődés hatására a `Kor` osztály `Sikidom` összes adattagját és metódusát megörökli. Ilyenkor azt mondjuk, hogy a `Kor` osztály `Sikidom` **leszármazottja** (*derived class*), és a `Sikidom` osztály `Kor` **őse** (*base class*).
Példaképp, így tudunk hivatkozni `Kor`-ön belül `Sikidom` egy adattagjára:



A `Sikidom` és a `Kor` osztály kapcsolata, melyet a fent látható nyíllal fejezünk ki: a nyíl a leszármazott osztályból az ősosztályba mutat.

```
class Kor : public Sikidom
{
    double r;
public:
    // ...
    void f()
    {
        std::cout << Sikidom::x << std::endl;
        std::cout << x << std::endl;
    }
};
```

A fenti kóddal kapcsolatban két dolog állapítható meg: Mivel `x` nevű objektum csak egy darab van az osztályban, ezért nem kell explicit módon `Sikidom`-on keresztül hivatkoznunk `x`-re. A másik dolog, hogy ez a kódrészlet sajnos nem helyes, fordítási hibát okoz: `x` ugyanis privát adattagja `Sikidom`-nak, és privát adattaghoz csak és kizárólag az adott osztály fér hozzá, még a leszármazott sem.
Ez probléma hisz mi `Kor`-t azért hoztuk létre, hogy speciálisabb feladatokat tudjuk végrehajtani `Sikidom` tulajdonságaival. Írjuk át ezeknek az adattagoknak a védettségét protectedre:

```
class Sikidom
{
protected:
    int x;
    int y;
public:
    double terulet() const {}
};
```

A `protected` tagokhoz csak az adott osztály fér hozzá, és azon osztályok, melyek ebből örökölnek.

1.2. Publikus öröklés

Öröklődésnél 3 különöző hozzáférési típust különítünk meg: `public, protected, private`. Ezeket a kulcsszavak az öröklődés kontextusában *access specifier*-nek nevezzük, és mind arra van kihatással, hogy az ősosztályból örökölt adattagok és metódusok milyen láthatóságot kapnak a leszármazottban, a következőféleképpen:

Öröklődés típusa:	public	protected	private
`public` örökölt adatok/metódusok	public	protected	private
`protected` örökölt adatok/metódusok	protected	protected	private
`private` örökölt adatok/metódusok	private	private	private

Az alábbi táblázat mutatja, hogy öröklődés után a bázisosztályban lévő adattagok/metódusok láthatóak-e a leszármazott osztályban:

Öröklődés típusa:	public	protected	private
Saját adatok/metódusok	igen	igen	igen
Örökölt adatok/metódusok (public)	igen	igen	igen
Örökölt adatok/metódusok (protected)	igen	igen	igen
Örökölt adatok/metódusok (private)	nem	nem	nem

A következőkben tekintsük a publikus öröklés tulajdonságait.

Az, hogy publikusan örököltünk, azt is jelenti, hogy egy altípus-t (*subtype*) hoztunk létre, és mint egy altípus, minden helyre, ahol `Sikidom`-ot szeretnénk használni, `Kor`-t is használhatjuk.

```
Kor k;
Sikidom s;
s = k; //ok
```

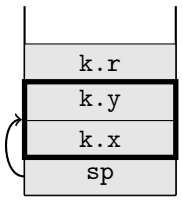
Azt már letisztáztuk, hogy `Kor` `Sikidom` altípusa, de azonban ez akkor is két különbötő típus, hogyan lehet, hogy nem kapunk fordítási idejű hibát?
A válasz az, hogy a fordító meg tudja oldani e két típus közötti értékadást. Mivel `Kor`-ben benne van maga `Sikidom` is, ezért meghívja az ahhoz tartozó értékadó operátort, és a `Kor` osztály `Sikidom`-ból örökölt adattagjait adja majd értékül, ebben az esetben (lévén nincs speciális értékadó operátor megírva `Sikidom`-ban) ez fog történni:

```
s.x = k.x;
s.y = k.y;
```

Ez azonban magával hordozza azt is, hogy minden információ, ami `Kor`-re specifikus (pl. sugár (`r`)) elveszik. Ez a folyamatot *slicing*.
A slicing láthatóan olyan jelenség, mely az esetek többségében elkerülendő. Szerencsére van lehetőség arra, hogy egy `Kor` típusú objektumra egy `Sikidom` típusú objektumon keresztül hivatkozzunk:

```
Kor k;
Sikidom *sp = &k;
Sikidom &sr = k; //a referencia ugyanúgy működik, mint a pointer
```

A fenti példában `sp` *statikus típusa* `Sikidom*`, a dinamikus típusa pedig `Kor`. Ez azt jelenti, hogy a fordító biztosan tudja már fordítási időben hogy `sp`-nek milyen típusa lesz, azonban azt nem tudhatja, hogy éppen milyen típusú objektumra mutat a `Sikidom`-hoz tartozó öröklődési fában, ez csak futási időben derülhet ki.



`sp` pointer csak a `k` objektum `Sikidom` részére tud hivatkozni.

Ez meg is felel az eredeti célunknak – amennyiben csak egy `Sikidom` objektummal szeretnénk dolgozni, addig mindegy is, hogy van-e a azon túl extra adat. Ha például egy olyan barát függvényt írunk, mely `Sikidom`-ok koordinátáit íratja ki, fölösleges információ ahhoz a sugár, vagy bármilyen egyéb tulajdonság.
Mi fog azonban történni, ha meghívjuk a `terulet` metódust?

```
Kor k;
Sikidom *sp = &k;
sp->terulet();
```

Ilyenkor a `Sikidom`-hoz tartozó `terulet` függvény kerül meghívásra. Ennek az oka az, hogy C++ban alapértelmezetten mindig a pointer/referencia statikus típusa szerint kerülnek a tagfüggvények meghívásra. Ennek hatékonyságai okai vannak, ugyanis így azt, hogy melyik függvényt kell meghívni (`Sikidom::terulet` vagy `Kor::terulet`) már fordítási időben meg lehet határozni. Ahhoz, hogy kikényszerítsük, hogy a fordító a dinamikus típusnak megfelelő függvényt hívja meg, a `virtual` kulcsszóra lesz szükségünk.

1.3. Virtuális függvények

Azokat az osztályokat, melyek legalább egy virtuális függvénnyel rendelkeznek, polimorfikus osztálynak (*polymorphic class*) nevezzük.

```
class Sikidom
{
    //...
```

```
public:
    virtual double terulet() const //virtuális!
    {
        //...
    }
};

class Kor : public Sikidom
{
    double r;
public:
    double terulet() const
    {
        return r * r * 3.14;
    }
};
```

Minden metódus, mely öröklődésnél a bázisosztály egy virtuális metódusával megegyező névvel, visszatérési értékkel, paraméterlistával és konstansséggal rendelkezik, implicit módon virtuális lesz. Konstruktor sose lehet virtuális. (pl.: `Kor`-ben `terulet` implicit módon virtuális lesz, mert `Sikidom`-ban is az volt).

A virtualitás következtében ha egy `Sikdiom` statikus típusú pointeren vagy referencián szeretnénk meghívni a `terulet` függvényt, akkora fordító futási időben leellenőrzi, hogy mi a mutatott terület típusa (dinamikus típusa) és az annak megfelelő függvényt hívja meg. Ez azt is jelenti, hogy ha egy öröklődési fában egy adott gyökérnek van egy virtuális függvénye, akkor minden belőle származó gyereknek az az adott virtuális függvénye leárnyékolja azt az adott függvényt.
A bázisosztály függvényének leárnyékolása azonban nem azt jelenti, hogy azt a függvényt nem lehet meghívni. Ha explicit módon jelezzük a fordítónak, hogy a bázisosztályban definiált metódust szeretnénk meghívni, akkor minden virtualitás ellenére is az lesz meghívva:

```
int main()
{
    Kor k;
    std::cout << k.Sikidom::terulet();
}
```

Ezt explicit névfeloldásnak (*explicit scope resolution*) nevezzük.

1.4. Tisztán virtuális függvények

Matematikailag, ha egy síkidomot háromszögekre felvágunk, meg tudjuk mondani a területét, ez azonban a jelenlegi példánkban ennyi adat segítségével aligha lehetséges. Ezért gyakran úgy dönthetünk, hogy egy adott függvénynek csak a fejlécét (*interface*) örököltetjük, de implementációját nem.
A fenti osztálynál azonban probléma, hogyha valaki létrehoz egy `Sikidom` típusú objektumot, és meghívja a `terulet` függvényt, jogosan várja, hogy az ki is számítsa neki. Ennek megkerülésére elérhetjük, hogy fordítási hibát kapjon, aki ilyennel próbálkozna: `Sikidom`ban `terulet`-et tisztán virtuálissá tesszük:

```
class Sikidom
{
    //...
public:
    virtual double terulet() const = 0; //tisztán virtuális!
};
```

Mostmár `terulet` tisztán virtuális (*pure virtual*) `Sikidom`-ban. Az olyan osztályokat, melyekből nem lehet objektumot létrehozni, absztraktnak nevezzük. Az olyan osztályokból, melyek tartalmaznak tisztán virtuális metódusokat, nem tudunk objektumot létrehozni.

1.4.1. Megjegyzés. Egy osztály úgy is lehet absztrakt, hogy `protected`-dé tesszük a konstruktorait (erről majd később). Az egyszerűség kedvéért, az elkövetkezendő szekciókban ha absztrakt osztályról beszélünk, gondoljunk egy tisztán virtuális metódussal rendelkező osztályra.

Egy olyan osztály, mely egy absztrakt osztályból örököl, ahhoz, hogy példányosítható legyen (lehessen olyan típusú objektumokat létrehozni), felül kell írnia az összes abban található tisztán virtuális függvényt.

```
int main()
{
    Sikidom s; //nem ok, fordítási idejű hiba: Sikidom absztrakt
    Kor k; //ok
    Sikidom *sp = &k;
    Sikidom &sr = k;

    sp->terulet(); //ok, Kor::terulet()-et hívja meg
    sr.terulet(); //szintén
}
```

Mint láthatjuk, absztrakt típusú pointert és referenciát lehet létrehozni, így a polimorfizmus összes előnyével élhetünk, továbbá mindenkit rá tudunk kényszeríteni arra, hogy a `terulet` függvényt implementálja, ha `Sikidom`-ból örököl és példányosítani szeretné azt az osztályt.
Persze, ettől mi még írhatunk definíciót egy tisztán virtuális függvényhez:

```
class Sikidom
{
    //...
public:
    virtual double terulet() const = 0;
};

double Sikidom::terulet()
{
    return -1; //negatív számmal jelezzük a számolás sikertelenségét
}
```

Figyeljük meg, hogy egy tisztán virtuális függvényt csak a deklarációtól külön tudunk definiálni.
Ez a trükk például akkor hasznosítható, ha egy van egy absztrakt bázisosztályunk, melynek szeretnénk, hogy öröklés után egy függvényét mindenki definiáljon felül, de akarunk adni hozzá egy alap implementációt.

```
class Sokszog : public Sikidom
{
    public:
    double terulet() const
    {
        return Sikidom::terulet();
    }
};
```

1.4.2. Megjegyzés. Nyilván ebben az esetben nem célszerű definiálnunk `Sikidom::terulet`-et, hisz ezzel csak lehetőséget adunk arra magunknak és mindenkinek, aki a `Sikidom` osztályt használja, hogy lábon lője magát.

A későbbiekben tekintsünk `Sikidom`-ra úgy, mintha nem lenne tisztán virtuális függvénye.

1.5. Konstruktorok öröklődésnél

Öröklődésnél a konstruktorok és destruktorok működése is újabb kihívásokat jelent. Lévén értelmetlen úgy létrehozni egy síkidomot, hogy nem adunk meg neki pozíciót (azaz, nem inicializáljuk az `x` és `y` változót), hozzunk létre e célra egy konstruktort. Így azt is el tudjuk érni, hogy a fordító nem generáljon default konstruktort:

```
class Sikidom
{
protected:
    int x, y;
public:
    Sikidom(int x, int y) : x(x), y(y) {}
    virtual double terulet() const {}
```

```
};

class Kor : public Sikidom
{
    double r;
public:
    double terulet() const {/* ... */}
};

int main()
{
    Sikidom s(1,2); //ok
    Kor k; //fordítási hiba
    Kor k2(1,2); //fordítási hiba
}
```

Mivel konstruktort nem írtunk `Kor`-nek, így a fordító generál nekünk egyet, melyben megpróbálja `Sikidom`-hoz tartozó adattagokat `Sikidom` paraméter nélküli konstruktorával meghívni. Azonban mivel nincs neki ilyen, fordítási idejű hibát kapunk. Ilyenkor csak egy lehetőségünk van a `Sikidom` osztály módosítása nélkül, írnunk kell `Kor`-nek egy konstruktort, melynek inicializációs listájában meghívjuk `Sikidom` egyetlen, két `int`-et váró konstruktorát.

```
class Sikidom
{
protected:
    int x, y;
public:
    Sikidom(int x, int y) : x(x), y(y) {}
    virtual double terulet() const {}
};

class Kor : public Sikidom
{
    double r;
public:
    Kor(int x, int y, int r) : Sikidom(x,y), r(r) {}
    double terulet() const {/* ... */}
};

int main()
{
    Sikidom s(1,2); //ok
    Kor k2(1,2); //ok
}
```

Mi történni a copy konstruktorokkal? Hasonló helyzetben, ha egy kört másolunk, a síkidom default copy konstrukora fog meghívódni, kivéve, ha az felül van definiálva. Azaz alapértelmezetten rendre másolásnál először meghívódik a `Sikidom` copy konstruktora, melyet a `Kor`-é követ.

```
class A {};
class B : public A {};
class C : public B {};
class D : public C {};
```

Konstruktorok hívási sorrendje: `A -> B -> C -> D`

Előfordulhat olyan helyzet, mikor nem akarjuk, hogy az adott osztályt példányosítsák. Például a fent erre egy módszer volt a tisztán virtuális metódusok használata. Egy másik módszer az, ha az összes konstruktort védetté tesszük.

```
class Base
{
```

```
protected:
    Base() {}
};

class Derived : public Base {};
```

Ebben az esetben `Derived` meg tudja hívni `Base` konstruktorát, így példányosítható lesz, de önmagában `Base` nem.

1.6. Destruktorok öröklődésnél

Ha a megfelelő `terulet` függvény csak akkor hívódott meg, ha a `terulet` függvényt `Sikidom`-ban virtuálissá tettük, hogyan fognak viselkedni a destruktorok? A válasz az, hogy azoknak a működése is közel azonos e téren, azaz nem a megfelelő destruktor fog lefutni, egy bázisosztályon keresztül próbálunk egy leszármazottat megsemmisíteni.

```
Sikidom *s = new Kor(1,2);
delete s; //undefined behaviour
```

A nem definiált viselkedés ellenére azért lehet nagyjából tudni, mi fog történni: mivel `s` statikus típusa `Sikidom`, az objektumon annak a destruktora fog lefutni. Ez az esetek többségében azt jelenti, hogy minden extra adat ezen kívül elszivárog.
Javítsuk is ezt gyorsan:

```
class Sikidom
{
protected:
    int x;
    int y;
public:
    Sikidom(int x, int y) : x(x), y(y) {}
    virtual ~Sikidom() {}
    virtual double terulet() const;
};
```

Bár örökölni közel minden osztályból lehet (ezalól kivétel lehet egy osztályon belüli privátként deklarált inline class-ok), ettől még nem mindig bölcs döntés az. Példaképp örökölni STL konténerekből nagy galibát okozhat, hisz azoknak a destruktora nem virtuális.

Ha biztosra akarunk menni, deklaráljunk minden destruktort virtuálisnak. Azonban tartsuk észben, hogy csakúgy mint minden virtuális függvény, ez futási idejű költséggel párosul. Ha 100%, hogy az osztályunkból nem fognak örökölni, bátran hagyjuk annak destruktorát `virtual` kulcsszó nélkül.

A fenti `A, B, C, D` osztályokat tekintvén, a destruktorok lefutási sorrendje ellentétes: `D -> C -> B -> A`.

1.7. Bővebben a slicing veszélyeiről

A slicing-ról már volt szó, azonban veszélyesebb az talán, mint elsőre gondolnánk. Emlékeztetőként, akkor fordul ez elő, ha egy objektumot egy, a bázisosztály objektumának adjuk **értékül**. Ilyenkor minden speciális adat elveszik, nem fogunk tudni hivatkozni specifikus metódusokra, de ami még veszélyesebb, hogy a virtuális függvényeket ugyanúgy meg fogjuk tudni hívni, azonban azok mást fognak csinálni, mint amire számítunk.

```
struct Base
{
    virtual void print() const
    {
        std::cout << "Base!" << std::endl;
    }
};

struct Derived : public Base
{
    void print() const
```

```
    {
        std::cout << "Derived!" << std::endl;
    }
};

void printRef(const Base& b)
{
    b.print();
}

void printVal(const Base b)
{
    b.print();
}

int main()
{
    Base b;
    Derived d;
    printRef(b); //Base!
    printRef(d); //Derived!
    printVal(b); //Base!
    printVal(d); //Base!
}
```

Láthatjuk, ebbe a problémába belefutni könnyebb mint gondolnánk. Ez is plusz egy indok, miért preferáljuk a referencia szerinti átvételt, és érték szerint csak akkor veszünk át, ha nagyon indokolt. E téren a kivételezésnél is jobb ha figyelünk!

```
int main()
{
    try
    {
        throw Derived();
    }
    catch(const Base& b) //referencia!
    {
        b.print(); //Derived!
    }
}
```

1.8. Kód módosítása polimorfizmusnál

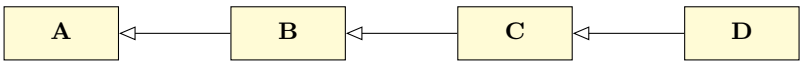
A polimorfikus osztályok használatának egyik nagy előnye, hogy már megírt függvényeket módosíthatunk anélkül, fordítása hiba keletkezzék bárhol, ahol az adott függvény már alkalmazásban van.

```
void f(Kor* k)
{
    //...
}
```

Lehet, az implementáció egy pontján úgy döntünk, hogy az `f` függvény várjon inkább `Sikdiom`-okat, mert nem használ fel semmilyen `Kor`-re specifikus adatot.

```
void f(Sikidom* k)
{
    //...
}
```

Ez azért nem okozhat problémát, mert minden olyan objektum, melynek típusa `Kor` vagy `Kor`-ből örökölt, ugyanúgy átkonvertálható `Sikidom*` típusra, mint `Kor*` típusra.



Függvény...	Eredeti típus	Módosított típus		
...paraméterei	A*	B*	C*	D*
	B*	A*	C*	D*
	C*	A*	B*	D*
	D*	A*	B*	C*
...visszatérési érte	A*	B*	C*	D*
	B*	A*	C*	D*
	C*	A*	B*	D*
	D*	A*	B*	C*

Az alábbi táblázat összefoglalja az itt megállapítottakat.
A fenti osztályhierarchiában `A` a legáltalánosabb, `D` a legspeciálisabb.
(piros: a módosítás nem megfelelő, zöld: a módosítás jó lesz):

Ennek a tanulsága az, hogy függvény paramétereknél **felfelé** az öröklődési láncon lehet haladni. Értelemszerű okokból `Sikidom*`-ról nem válthatunk `Kor*`-ra a fordítási hiba veszélye nélkül, hisz egy `Sikidom` típusú objektum nem adható át egy olyan függvénynek.

Tekintsük most a függvények visszatérési értékét:

```
Sikidom* makeObject() {}
```

Hasonlóan, döntsünk úgy hogy az `makeObject` függvény `Kor*`-t adjon vissza.

```
Kor* makeObject() {}
```

A fentiekhez hasonló okokból, ez sem okoz majd problémát. Megállapítható, hogy egy függvény visszatérési értéke egy adott öröklődési fán **lefelé** bátran módosulhat.

1.8.1. Megjegyzés. Ez mutatja jól a publikus öröklődésnek az *az egy* relációját, hisz minden kör egy síkidom, de nem minden síkidom kör.

1.9. Többszörös öröklődés

Viszonylag kevés nyelv engedi meg, hogy egyszerre több osztályból is örököljünk. A C++ azonban nem ilyen:

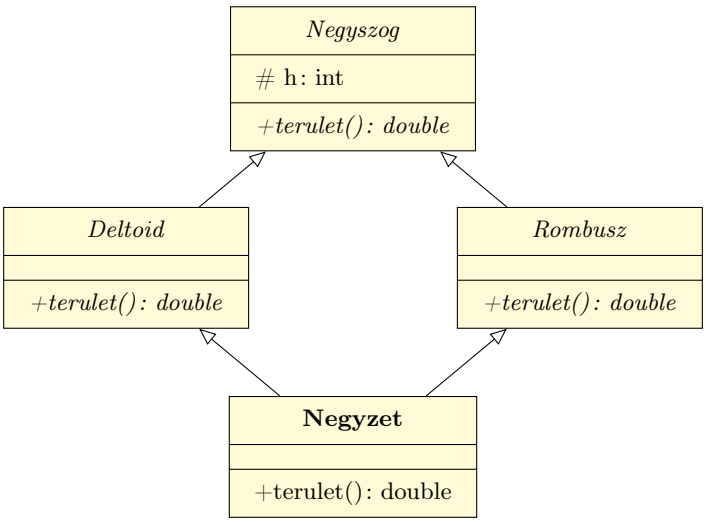
```
class BaseOne {};
class BaseTwo {};

class Derived : public BaseOne, public BaseTwo {};
```

A többszörös öröklés egy nagyon erős eszköze a nyelvnek, hisz az *az egy* relációt több másik osztállyal is fel tudjuk állítani. Példaképp, ha adott egy `Bird` és egy `NonFlyingAnimal` osztály, a `Penguin` örökölhetne mindkettőből, hisz minden pingvin *az egy* madár, és minden pingvin *az egy* röpképtelen állat.
Azonban általában igaz az, hogy a többszörös öröklődésnél több a galiba mint a haszon, így hacsak nincs nagyon jó okunk rá, ne erőltessük. Nehezen lesz velük átlátható, hogy milyen lesz egy pointer dinamikus típusa, rászorulhatunk side cast-okra (erről később), és egyéb nem elegáns módszerekre.

A többszörös öröklés igazi fő gonosza talán a gyémántöröklődés, melynek használata közel minden esetben elkerülendő.

```
class Negyszog
{
protected:
    double h; //height
public:
    virtual double terulet() = 0;
};
```



Egy példa a gyémántöröklésre.

```
class Rombusz : public Negyszog
{
public:
    //az egyszerűség kedvéért maradjon tisztán virtuális
    double terulet() = 0;
};

class Deltoid : public Negyszog
{
public:
    double terulet() = 0;
};

class Negyzet : public Deltoid, public Rombusz
{
public:
    double terulet()
    {
        return h * h;
    }
};
```

A legutóbbi osztály természetesnek tűnhet, hisz egy négyzet deltoid is, meg rombusz is, így szeretnénk, ha minden függvény, ami `Deltoid`-ot vagy `Rombusz`-t vár, `Negyzet`-et is elfogadjon. Azonban a fenti kód nem fordul le, hisz mind `Rombusz`-ban, mind `Deltoid`-ban van egy teljes `Negyszog` is. Így `Negyzet` kétszer fogja tartalmazni annak összes adattagját, így `h`-t is, és a fordító nem tudja kitalálni, mi épp melyikre gondolunk. 2 kézenfekvő megoldás lehet:

```
class Negyzet : public Deltoid, public Rombusz
{
public:
    double terulet()
    {
        return Deltoid::x * Deltoid::x;
    }
};
```

Azonban leggyakrabban nem cél, hogy egy leszármazott bármelyik őst kétszer tartalmazza. Erre megoldás lehet a virtuális öröklődés.

```
class Negyszog
{
```

```
protected:
    double x;
public:
    virtual double terulet() = 0;
};

class Rombusz : virtual public Negyszog //virtuális öröklés!
{
public:
    double terulet() = 0;
};

class Deltoid : virtual public Negyszog
{
public:
    double terulet() = 0;
};

class Negyzet : public Deltoid, public Rombusz
{
public:
    double terulet()
    {
        return x * x;
    }
};
```

Ezzel garantáltuk is ezt, de milyen áron? A virtuális öröklődésnek, csak úgy mint a virtuális metódusoknak futási idejű költsége van, így ha egy mód van rá, kerüljük. Az öröklődési gráfot is nehéz átlátni és értelmezni, ha tartalmaz gyémánt öröklést.
Annak ellenére, hogy a gyémánt öröklődés erősen ellenjavallott dolog, van kivétel még a standard könyvtáron belül is: az `iostream`. A beolvasó és kiíratós stream-ek közös tulajdonságait tartalmazza `iosbase`, melyből örököl `istream` és `ostream` is, és mindkettőből az `iostream`.

1.10. Privát és védett öröklődés

Lévén az esetek **erősen** túlnyomó többségében publikusan öröklünk, és igény is nagyon kevés van ettől eltérni, így ez a szekció nem részleteiben fogja tárgyalni e két öröklődési módot.

```
struct Base
{
    int x;
    virtual ~Base() {}
};
struct Derived : private Base {}; //private inheritance
struct DerivedTwo : public Derived {}; //public inheritance

int main()
{
    Derived *d = new DerivedTwo(); //ok
    Base *b = d; //fordítási hiba
}
```

A fenti kódrészlet azzal a hibaüzenettel nem fog lefordulni, hogy a `Base` egy nem elérhető bázisa `Derived`-nak. Ennek oda egyszerű: mivel privátan örököltünk, ezért `x` nem hozzáférhető `DerivedTwo`-ban, azonban hozzá tudnánk férni, ha `Base`-ként tudnánk rá hivatkozni.
Ezt értelmezhetjük úgy is, hogy a a privát/védett öröklés „elvágja” az öröklődési fát olyan értelemben, hogy a „vágáson” keresztül nem lehet leszármazottakat ősosztály típusba konvertálni.

Privát örökléssel szokás például tartalmazást is kifejezni. Bár leggyakrabban ha egy másik osztály adattaggá teszünk, kifizetődőbb, van eset, amikor pl. az interface egy részét elérhetővé akarjuk tenni.

Egy másik példa lehet, amikor egy olyan osztályból öröklünk, melynek nincs virtuális destruktora. Ezzel elérhetjük, hogy véletlenül se tudjunk átkonvertálni az ősosztály típusára, és ezzel nem elkerülhetjük a nem definiált viselkedést.

```
class MyVector : private std::vector<int>
{
public:
    using std::vector<int>::push_back; //C++11
};
```

A védett öröklés szinte univerzálisan sosem használt.

1.10.1. Megjegyzés. Az öröklődés típusa alapértelmezetten (azaz ha nem írjuk ki az access specifiert) ha `class`ból öröklünk `private`, ha `struct`ból akkor `public`.

1.10.2. Megjegyzés. A publikus öröklődést szokás **az egy** (*is a*), privátot **van egy** (*has a*) kapcsolatnak is hívni.

1.11. C++11

C++11-ben lehetőségünk van picit kényelmesebbé tenni az életünket, például az `override` kulcsszóval, mely garantálja, hogy az adott függvény felülír egy másikat.

```
class Base
{
public:
    virtual void f();
};

class Derived : public Base
{
public:
    virtual void f() override {}
};
```

Amennyiben a felülírás mégse sikerült (`f` nem virtuális, nem egyezik a paraméterlista/konstansság), fordítási idejű hibát kapunk, mely mindig sokkal jobb, mint a futási idejű.

Hasonlóan hasonló kulcsszó a `final`, mely megtiltja az öröklődést, így bátran használhatunk nem virtuális destruktorokat.

```
class Base final
{
public:
    virtual void f();
};

class Derived : public Base {}; //fordítási idejű hiba
```

Amennyiben örököltünk egy osztályból, mely privát adattagokat tartalmaz, azokhoz nem fogunk tudni hozzáférni. Ilyenkor rá vagyunk kényszerülve get/set függvények alkalmazására, ha azok meg vannak írva (szempont lehet megírásuk szempontjából ez is).

2. Cast-ok

Implicit konverziókkal számtalanszor találkozunk, például ha egy lebegőpontos számot egy egész számnak adunk értékül, vagy fordítva. Konverzió volt az is, amikor egy konverziós operátor vagy konverziós konstruktor segítségével egy osztályhoz tartozó objektumot egy másik osztály típusúnak adtunk értékül. A pilomorfikus osztályoknál meg számtalanszor éltünk már vele.

2.1. dynamic_cast

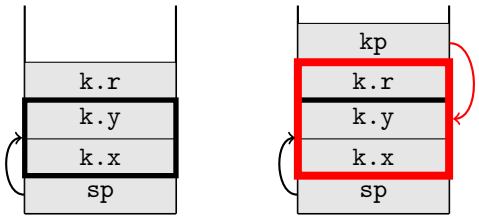
Néha abba a kellemetlen helyzetbe kerülhetünk, hogy van egy adott pointerünk/referenciák, és a dinamikus típusa `Kor`, azonban statikus típusa `Sikidom`, de mégis fontos, hogy `Kor`-re vonatkozó dolgokat használjunk. Ilyenkor segíthet a `dynamic_cast` mely képes ezt a konverziót végrehajtani:

```
void f(Sikidom *sp)
{
    Kor *kp = dynamic_cast<Kor*>(sp);
}
```

Megfigyelhető, hogy kacsacsőrbe írtuk azt a típust, amelyre konvertálunk, és utána gömbölyű zárójelbe azt, melyből konvertálnánk. A `dynamic_cast`-ot **csak és kizárólag** polimorfikus osztályoknál alkalmazhatjuk.

2.1.1. Megjegyzés. A `dynamic_cast` előtt nincs `std`. Ennek oka, az, hogy a `dynamic_cast` egy operátor, és nem egy standard függvény.

Fontos megállapítás, hogy ez a cast egy adott pointer/referencia dinamikus típusát változtatja/változtathatja meg, így értelemszerűen az a típus amelyre konvertálunk mindenképpen referencia vagy pointer kell hogy legyen. Ez leginkább 3 dologra jó, leszármazott típusról bázis típusra konvertálásra (*upcast*), bázisról leszármazottra



A `dynamic_cast` működése.

(*downcast*), valamint többszörös öröklődésnél bázisok közötti váltásra (*sidecast*).
A `dynamic_cast` futási időben végzi a konverziót. Természetesen előfordulhat az, hogy a konverzió nem lehetséges:

```
Sikidom *s = new Sikidom;
Kor *k = dynamic_cast<Kor*>(*s);
```

Ebben az esetben egy hibás downcast-ot csináltunk, hisz `s`-nek a dinamikus típusa is `Sikidom`. Mivel minden `Kor` *az egy* `Sikidom`, de ez fordítva nem igaz, így a konverzió nem lehetséges. Ilyenkor a `dynamic_cast` nullpointert ad vissza. Ha referenciákkal jutunk ilyen helyzetbe, akkor egy `std::bad_cast` típusú kivételt fog dobni.

Tekintsünk példát a `dynamic_cast` működésére:

```
class Base
{
    virtual int f(){} //polimorfizmus szükséges
};
class DerivedOne : virtual public Base {};
class DerivedTwo : virtual public Base {};
class DerivedLast : public DerivedOne, public DerivedTwo {};
//ez a jól ismert gyémántöröklés egy példája

int main()
{
    DerivedLast *dlp = new DerivedLast;
    Base *bp = dynamic_cast<Base*>(dlp); //upcast, dynamic_cast fölösleges
    DerivedOne *dop = dynamic_cast<DerivedOne*>(bp); //downcast
    DerivedTwo *dtp = dynamic_cast<DerivedTwo*>(dop); //sidecast
    delete dlp;

    bp = new Base;
    dop = dynamic_cast<DerivedOne*>(bp);  //dop nullpointer
    delete bp;
```

```
    Base b;
    try
    {
        DerivedOne &dor = dynamic_cast<DerivedOne&>(b);
        //std::bad_cast-ot fog dobni
    }
    catch (std::bad_cast &exc)
    {
        std::cout << exc.what() << std::endl;
    }
}
```

Ez azonban nem hatékony, hisz a `dynamic_cast` futási időben járja végig az öröklődési láncot. Ha nem vagyunk biztosak benne, hogy garantáltan lehetséges a castolás, akkor `dynamic_cast`-ot érdemes használni, ha azonban biztosak vagyunk benne hogy az lehetséges, akkor `static_cast` hatékonyabb.

2.2. static_cast

Amennyiben teljesen biztosak vagyunk abban, hogy a cast szabályos, a `static_cast` is használható mindenhol, ahol a `dynamic_cast` is. Azonban ez az operátor nem végez semmilyen futási idejű ellenőrzést, így sikertelen kaszt esetén a kapott pointer/referencia invalid lesz (használata nem definiált viselkedést eredményez).

A `static_cast` fordítási időben konvertál egy típust egy másik típusra. Annak ellenére, hogy nem feltétlen biztonságos a használata polimorfikus osztályoknál, ez a konverzió végez ellenőrzéseket, méghozzá fordítási időben.

```
//a dynamic_cast-nál látott típusok itt is érvényesek
int main()
{
    Base *bp = new DerivedOne;
    DerivedOne *dop = static_cast<DerivedOne*>(bp); //ok
    DerivedLast *dlp = static_cast<DerivedLast*>(bp); //dlp invalid
    *dlp; //undefined behaviour

    int i = 5;
    double d = 5 / static_cast<double>(i);

    //malloc visszatérési típusa void*
    DerivedTwo *dtp = static_cast<DerivedTwo*>(malloc(sizeof(DerivedTwo)));
    free (dtp);

    Base ba[10];
    bp = static_cast<Base*>(ba);
}
```

E kettő kaszttal közel minden konverziót meg tudunk tenni, melyre szükségünk van és nem okoz nem definiált viselkedést.

2.3. const_cast

A `const_cast` az egyik a két veszélyesebb kaszt közül. Használata csak kivételes esetekben elfogadható, ugyanis nagyon sok galibát okozhat.
A `const_cast` egy konstans objektumról „lekasztolja” a konstansságot:

```
void print(int *param) { printf("%i", *param); }

int main()
{
    const int a = 0;
    print(const_cast<int*>(&a));
}
```

A fenti példa jól demonstrál egy tipikus helyzetet, amikor szabályos a `const_cast` használata. Lévén C-ben nem létezett korábban a `const` kulcsszó, ezért mindent vagy érték szerint vagy nem konstans pointerrel vettek át. Ennek egyik következménye az, hogyha egy régi, C-s függvényt kell igénybe vennünk, mely nem módosítja a paramétert azonban nem konstansként várja, akkor le kell „szednünk" a konstans objektumainkról a konstansságot.

Amennyiben egy nem konstans objektumra konstans referenciával vagy pointerrel hivatkozunk, abban az esetben a nem konstanssá castolt referencián vagy pointeren keresztüli módosítás legális. Amennyiben azonban egy adott objektum konstansként lett deklarálva, annak a módosítása nem definiált.

```
int main()
{
    int i = 3;
    const int& cref_i = i;
    const_cast<int&>(cref_i) = 4; // ok, megváltoztatja i értékét

    const int j = 3;
    int* pj = const_cast<int*>(&j); // maga a cast legális
    *pj = 4; // a módosítás már undefined behaviour
}
```

2.4. reinterpret_cast

Mind közül a legveszélyesebb a `reinterpret_cast`: két tetszőleges típus között végez konverziót, ha értelmezhető az, ha nem. Az értelmes konverziók közül egyetlen eset kivételével minden esetben a fenti 3 cast végre tudja hajtani a konverziót helyesen és sokkal biztonságosabban. Ez azt jelenti, hogy a `reinterpret_cast` használata közel kivétel nélkül elkerülendő.

Az egyetlen konverzió, mely értelmes, és a fentiek nem képesek végrehajtani, amikor polimorfikus osztályoknál egy leszármazottat egy privát ősosztály típusra konvertálunk.

```
class MyVector : private std::vector<int> {};

MyVector *mvp = new MyVector;
std::vector<int> *vp = reinterpret_cast<std::vector<int>*>(mvp);
```

Azonban hiába szabályos ez a konverzió, rengeteg veszélyforrást tartalmaz, hisz a privát öröklés általában épp ennek a megakadályozására szolgál.

2.5. C-szerű cast

A C-szerű castok, vagy más néven *C-style casts* olyan konverziók, melyek a fenti konverziókat próbálják meghívni, a `dynamic_cast` kivételével. Amennyiben az egyik nem sikerül, a következőre haladnak. Ennek az értelemszerű hátránya az, hogy ha mind a `static_cast`, mind a `const_cast` sikertelen, akkor a `reinterpret_cast`-ot fogja használni.

```
int main()
{
    float a = (float)10; //static_cast kerül meghívásra
    const int a;
    int &b = (int&)a; //const_cast
    Base *bp = (int*)b; //reinterpret_cast
}
```

A `reinterpret_cast`-tal ellentétben **soha** sincs szükség erre a cast-ra, csak a C-vel való kompatibilitás miatt része még a nyelvnek.