Öröklődés

A problémák objektum-orientált feldolgozása során általában egy új programépítési módszert, a származtatást (öröklés, *inheritance*) alkalmazzuk. Az öröklődés lehetővé teszi, hogy már meglévő osztályok adatait és műveleteit új megközelítésben alkalmazzuk, illetve a feladat igényeinek megfelelően módosítsuk, bővítsük. A problémákat így nem egyetlen (nagy) osztállyal, hanem osztályok egymásra épülő rendszerével (hierarchiával) oldjuk meg.

Az öröklés az OO C++ nyelv egyik legfőbb sajátossága. Ez a mechanizmus lehetővé teszi, hogy meglévő osztályokból kiindulva, új osztályt hozzunk létre. Az öröklés során az új osztály örököli a meglévő osztály(ok) nyilvános (**public**) és védett (**protected**) tulajdonságait (adattagjait) és viselkedését (tagfüggvényeit), amelyek aztán annak sajátjaként használhatunk. Az új osztállyal bővíthetjük is a meglévő osztály(oka)t, új adattagokat és tagfüggvényeket definiálhatunk, illetve újra értelmezhetjük (override) az örökölt, de működésükben elavult tagfüggvényeket (polimorfizmus).

ősosztály – base class
öröklés – inheritance
gyermekosztály – derived class

A C++ támogatja a többszörös öröklődést (multiple inheriance), mely során valamely új osztályt több alaposztályból származtatunk. Nem összekeverendő a több osztályon keresztüli öröklés.

```
class Base1
{
     //...
};

class Base2
{
     //...
};

class Derived: public Base1, public Base2
{
     //...
};
```

A származtatott osztály olyan osztály, amely adattagjait és függvényeit egy vagy több előzőleg definiált osztálytól örököli. A származtatott osztály szintén lehet alaposztálya további osztályoknak, lehetővé téve ezzel az osztályhierarchia kialakítását.

A származtatott osztály az alaposztály minden tagját örököli, azonban az alaposztályból csak a **public** és **protected** tagokat éri el sajátjaként. A tagfüggvényeket általában **public** vagy **protected** hozzáféréssel adjuk meg, míg az adattagok esetén a **protected** vagy **private** elérést alkalmazzuk. Öröklés során megadhatjuk a származtatás módját (**public, protected, private**).

Az alaposztálybeli elérhetőségüktől függetlenül **nem öröklődnek** a konstruktorok, destruktor, az értékadó operátor valamint a **friend** viszonyok.

Az öröklés módja	Alaposztálybeli elérés	Hozzáférés származtatott osztályban
public	public protected	public protected
protected	public protected	protected protected
private	public protected	private private

A public származtatás során az örökölt tagok megtartják az alaposztálybeli elérhetőségüket, míg a private származtatás során az örökölt tagok a származtatott osztály privát tagjaivá válnak. Protected öröklés esetén az örökölt tagok védettek lesznek az új osztályban.

Virtuális alaposztályok a többszörös öröklésnél

A többszörös öröklés során problémát jelenthet, ha ugyanazon alaposztály több példányban jelenik meg a származtatott osztályban. A virtuális használatával az ilyen jellegű problémák kiküszöbölhetők. A virtuális alaposztály az öröklés során egyetlen példányban lesz jelen a származtatott osztályokban, független attól, hogy hányszor fordul elő az öröklődési láncban.

Polimorfizmus

Amikor a származtatott osztály felüldefiniálja az ősosztály valamelyik metódusát, ami annyit jelent, hogy az ősosztályban definiált metódusnak meghatározunk egy új működést.

Ad-hoc polimorfizmus, ismertebb nevén a függvénynevek túlterhelése (overloading). Ekkor a fordítóprogram a típusok alapján választja ki az elkészített függvényváltozatok közül a megfelelőt. Ennek kiterjesztése a parametrikus vagy fordítási idejű polimorfizmus, ami lehetővé teszi, hogy ugyanazt a kódot bármilyen típussal végre tudjuk hajtani. Mivel öröklés során gyakran speciálizáljuk a leszármazott osztályt, szükséges lehet, hogy bizonyos örökölt műveletek másképp működjenek. Ezt az igényt a virtuális (**virtual**) tagfüggvények bevezetésével tehetjük meg. A futásidejű polimorfizmusnak köszönhetően egy objektum attól

függően, hogy az osztály-hierarchia mely szintjén lévő osztály példánya, ugyanarra az üzenetre másképp reagál. Az pedig, hogy az üzenet hatására melyik tagfüggvény hívódik meg az öröklési láncból, csak a program futása közben derül ki (késői kötés).

Virtuális tagfüggvények

A virtuális függvény olyan **public** vagy **protected** tagfüggvénye az alaposztálynak, amelyet a származtatott osztályba újra definiálhatunk az osztály "viselkedésének" megváltoztatása érdekében.

Az osztályon belül virtuálissá tehetünk metódusokat a **virtual** kulcsszó használatával. **Konstruktor nem lehet virtuális!**

```
virtual int vf();
```

Lehetőségünk van **tisztán virtuális** metódusokat is készíteni. Ilyenkor nem definiálunk törzset a metódusnak az alaposztályban, csak egyenlővé tesszük nullával, pl.:

```
virtual int getNumber() const = 0;
```

Ekkor a származtatott osztályban felül tudjuk írni ezt a függvényt és az **override** kulcsszóval jelölhetjük, hogy virtuális metódust definiálunk. Fontos, hogy a szignatúrája megegyezzen az ősosztályban definiált virtuális metóduséval.

```
int getNumber() const override { return _number; }
```

Azokat az osztályokat, amelyek tartalmaznak legalább egy tisztán virtuális metódust **absztrakt** osztályoknak nevezzük. C++-ban nincs külön jelölés az absztrakt osztályokra mint pl. JAVA-ban az **abstract** kulcsszó. Ezekből nem készíthetünk objektumpéldányt. Azok az osztályok amelyek csak és kizárólag virtuális metódusokat tartalmaznak **interface**-eknek nevezzük. Fontos, hogy a származtatott osztályok az interface összes nyilvános metódusát megvalósítsák.

Statikus korai kötés

Korai kötés során a fordító statikusan befordítja a kódba a közvetlen tagfüggvény hívásokat. Az osztályok esetén ez az alapértelmezett működési mód.

Dinamikus késői kötés

Alapvetően változik a helyzet, ha az ősosztályban a tagfüggvényeket virtuálissá tesszük. A virtuális függvények hívását közvetlen módon, memóriában tárolt címre történő ugrással helyezi el a kódban a fordító.

Virtuális destruktor

Ha az alaposztály destruktora virtuális, akkor minden ebből származó osztály destruktora is virtuális lesz. Ezáltal biztosak lehetünk abban, hogy a megfelelő destruktor hívódik meg, amikor az objektum megszűnik, még akkor is, ha valamelyik alaposztály típusú mutatóval vagy referenciával hivatkozunk a leszármazott osztály példányára.

Smart pointerek

A C++11 három új mutatótípust vezetett be, amelyek hivatkozások számlálásra és az elért memóriaterület automatikus felszabadítására is képesek. Használatukhoz a **memory** header fájl meghívása szükséges.

unique_ptr: akkor használjuk, amikor a lefoglalt memóriát nem osztjuk meg, azonban átvihető más unique_ptr-hez (move konstruktor)

```
std::unique_ptr<int> ptr = std::make_unique<int>(1);
```

shared_ptr: akkor alkalmazzuk, amikor a lefoglalt memóriát meg kívánjuk osztani, pl. másolással. A shared_ptr számon tartja, hogy hány referencia van az adott objektumon. Erre a reference count segítségével képes. Ez a heap-en jön létre és mindegyik shared_ptr megnöveli az értékét eggyel, majd amikor megszűnik akkor csökkenti. Amikor ennek az értéke nullára csökken, akkor törlődik az utolsó mutató megszűnésekor a mutatott érték. a `.use_count()`-al tudjuk lekérdezni, hogy mennyin áll a referencia számláló.

```
std::shared_ptr<int> ptr = std::make_shared<int>(1);
```

weak_ptr: a shared_ptr általall kezelt objektumra hivatkozó referenciát tartalmaz, azon nem növeli a hivatkozás számlálót.

Ezekkel az új mutatók bevezetésével az **auto_ptr** elavult lett, így használata nem javasolt.