# Osztályok

## OOP – Objektum Orientált Programozás (Object Oriented Programming)

Négy dolog szükséges ahhoz, hogy egy programozási nyelvről azt mondhassuk, hogy támogatja az OOP paradigmát.

1. **Adatrejtés (data hiding)**
2. **Egységbezárás (encapsulation)**
3. **Öröklődés (inheritance)**
4. **Polimorfizmus (Polymorphism)**

C++ nyelvben speciális típusok, az osztályok szolgálnak arra, hogy az adatokat és a hozzájuk rendelt függvényeket egy egységként, egy objektumként kezeljük.

A C++ nyelvben a **class** típus a C nyelv struktúratípusának kiterjesztése. Mindkét struktúra tartalmazhat adatmezőket (adattag – data members), azonban a C++-ban ezen adattagokhoz különféle műveleteket, ún. tagfüggvényeket (member function) is megadhatunk. A C++-ban egy osztály típusú tárolási egység függvényeket kombinál adatokkal, és az így létrejött kombinációt elrejtjük, elzárjuk a külvilág elől. Ezt értjük egységbezárás alatt. Egy osztály deklarációja sokban hasonlít a jól ismert struktúra deklarációjához.

```
class osztálynév
{
     adattagok
public:
     tagfüggvények
};
```

A hagyományos C-s struktúrák (**struct**) és a C++ osztályok (**class**) között a fő különbség a tagokhoz való hozzáférésben keresendő. A C-ben egy struktúra adattagja (megfelelő érvényességi tartományon belül) szabadon elérhetők, míg C++-ban egy **struct** vagy **class** minden egyes tagjához való hozzáférés önállóan kézben tartható azáltal, hogy nyilvánosnak, privátnak vagy védettnek deklaráljuk a **public**, **private** és **protected** kulcsszavakkal. Ezeket nevezzük angolul **access specifiers**-öknek.

A **class** kulcsszóval definiált osztályban az adatmezők alapértelmezés szerint **private** hozzáférésűek, míg egy struct esetén **public**.

Az OOP-re jellemző, hogy az adattagokat privátnak (**private**), a tagfüggvényeket pedig publikusnak (**public**) deklaráljuk. Fontos megjegyezni, hogy a C++ osztály önállóan is típusértékű nem kell a **typedef** segítségével típusazonosítót hozzárendelni.

Összefoglalva: A C++-ban az objektumoknak megfelelő tárolási egység típusát osztálynak (**class**) nevezzük. Az objektum tehát valamely osztálytípussal definiált változó, amelyet más szóhasználattal az osztály példányának (**objektumpéldánynak**) nevezünk (**instance**).

Kitérő: **friend** mechanizmus

A friend (barát) mechanizmus lehetővé teszi, hogy az osztály **private** és **protected** tagjait nem saját tagfüggvényből is elérjük. A friend deklarációt az osztály deklarációjában belül kell elhelyeznünk tetszőleges elérésű részben. „barát" lehet egy külső függvény, egy másik osztály adott tagfüggvénye, de akár egy egész osztály is (vagyis annak minden tagfüggvénye)

```
class MyClass
{
private:
      int counter;
public:
      friend int getCounter();
      //  friend  int  getCounter()  const;  hiba  ->  non-member
      function   cannot have a 'const' qualifier.
};

int getCounter() { return MyClass::counter; }
```

Minden tagfüggvény, még a paraméter nélküliek (**void**) is rendelkeznek egy nem látható (implicit) paraméterrel: ez a **this**, amelyben a hívás során az aktuális objektumpéldányra mutató pointert ad át a C++, és minden adattag hivatkozás automatikusan **this->adattag** kifejezésként kerül a kódba.

Kitérő: **inline** tagfüggvények – implicit módon is írhatunk inline tagfüggvényeket.

```
class MyClass
{
private:
      int i;
public:
     int getNumber() const { return i; }
};
```

A fordító az ilyen tagfüggvényeket automatikusan **inline** függvénynek tekinti. A megoldás nagy előnye, hogy a teljes osztályt egyetlen fej állományban tárolhatjuk, és az osztály tagjait könnyen át tekinthetjük. Általában kisebb méretű osztályok esetén alkalmazható hatékonyan a megoldás. Ilyenkor nem függvény hívás történik, hanem magának a függvénynek a kódja beillesztésre kerül a hívások helyein. Nagyobb méretű kód: a függvény törzse több helyen szerepel, több optimalizálási lehetőség miatt lehet mégis rövidebb. Megspórolja a függvényhívás idejét. Explicit módon is megtehetjük, hogy inline-nak jelölünk egy függvényt, mégpedig az **inline** kulcsszó beillesztésével a függvény neve elé.

```
class MyClass
{
private:
      int i;
public:
     inline int getNumber() const { return i; }
      // int inline getNumber() const { return i; } mindkét
      használat helyes
};
```

# Konstruktorok

A C++ programokban az objektumok inicializálását speciális tagfüggvények, a konstruktorok végzik. A konstruktor neve azonos az osztályéval, azonban nincs visszatérési értéke, még **void** se. Feladata a dinamikus adattagok létrehozása és az adattagok kezdőértékkel való inicializálása.

**4 féle konstruktort készíthetünk:**

1. Paraméter nélküli, alapértelmezett (default) konstruktor törzsében az adattagoknak konstans kezdő értéket adunk.
   `MyClass() { x = 1; y = 1; }`
2. Paraméteres konstruktornak az a feladata, hogy azoknak az adattagoknak adjon értéket, amelyek a feladat végrehajtásához szükségesek. Ehhez megfelelő darabszámú és típusú paraméter listát kell kialakítanunk.
   `MyClass(int x, int y) { this->x = x; this->y = y; }`
3. Másoló konstruktor (copy constructor) használatával egy objektumpéldánynak kezdőrétéket adhatunk egy már létező és inicializált objektumpéldánnyal.
   `MyClass(const MyClass& mc) { x = mc.x; y = mc.y; }` (részletesebben a 3. gyakorlaton lesz erről szó)
4. Move konstruktor (MSc. multiparadigma programozás, most nem foglalkozunk vele)

Az osztálynak több konstruktora is lehet, és mindig az argumentumlista alapján dől el, hogy melyik változatot kell aktivizálni. Ha mi magunk nem definiálunk konstruktort akkor a C++ fordító biztosít egy alapértelmezett és egy másoló konstruktort. Ha valamilyen saját konstruktort készítünk, ahhoz az alapértelmezett (paraméter nélküli) konstruktort is definiálnunk kell, amennyiben szükségünk van rá.

Függvények bemeneti paramétereinek adhatunk alapértelmezett (default) értéket.

```
int f(int i = 0) { return i };
```

Ilyenkor, ha meghívjuk az f() függvényt és nem adunk neki bemeneti értéket, akkor a default értéket fogja használni.

```
int main()
{
	std::cout << f();  // 0
	std::cout << f(1); // 1
}
```

Megjegyzés: ha nem adtunk volna az f()-nek default bemeneti paraméter értéket az első hívás szabálytalan lenne, mert nem találna a fordító olyan f() függvényt aminek a szignatúrája megfelel annak a hívásnak.

## Implicit- és explicit konstruktor

**Explicit** kulcsszó azt mondja ki, hogy a konstruktor explicit, nem használható implicit átalakítás és copy-initialization.

```
class A
{
public:
	A(int) { }
	A(int, int) { }
};

class B
{
public :
	explicit B(int) { }
	explicit B(int, int) { }
};

int main()
{
	A a1 = 1;	// OK, copy-initialization A::A(int)
	A a2(2);	// OK, direct-initialization A::A(int)
	A a3 {4, 5};	// OK, direct-list-initialization A::A(int, int)
```

```
	A a4 = {4, 5};   // OK, copy-list-initialization A::A(int, int)
	A a5 = (A)1;     // OK, explicit cast

	B b1 = 1;        // Error, copy-initialization figyelmen kívül
hagyja a B::B(int)
	B b2(2);         // OK, direct-initialization B::B(int)
	B b3 {4, 5};     // OK, direct-list-initialization B::B(int, int)
	B b4 = {4, 5};   // Error, copy-list-initialization nem veszi
figyelembe B::B(int, int)
	B b5 = (B)1;     // OK, explicit cast
}
```

### Member initialization list

Segítségével elkerülhető a default konstruktor felesleges hívása. Tekintsük az alábbi példát:

```
class A
{
public:
	A() { x = 0; }
	A(int x_) { x = x_; }
	int x;
};

class B
{
public:
	B() { a.x = 3; }
private:
	A a;
};
```

Ebben az esetben a B osztály konstruktora először az A osztály default konstruktorát fogja meghívni, és utána állítja be az a.x értékét 3-ra. Egy jobb megoldás, ha a B osztály direktben meghívja az A osztály konstruktorát egy inicializációs listában

```
B() : a(3) { }
```

Ebben az esetben az A-nak már csak az int paraméteres konstruktorát fogja meghívni. Megspórolunk egy felesleges konstruktor hívást, ami abban az esetben

nagy előny, ha pl. az A osztály default konstruktorában még egy dinamikus memóriafoglalás is lenne.

**Static** kulcsszó.

Statikus változókat, függvényeket a **static** kulcsszó segítségével hozhatunk létre. Statikus változók a statikus globális tárterületen jönnek létre. Élettartamuk a program kezdetétől a program futásának végéig tart. Nem szükséges kezdőértékkel el látni őket csak abban az esetben, ha konstansok.

```
int main()
{
	static int i; // fordul
	static const int j; // hiba, konstans-oknak kezdőértéket kell adnunk
}
```

**Static az osztályon belül.**
C++-ban az osztály szintű változókat és függvényeket **static** prefixszel kell ellátni. Ezek nem fognak létrejönni minden egyes objektum példánynál, mint az adattagok, hanem csak egyszer. Statikus adattagok kezdőértékkel való inicializálása csak az osztályon kívül megengedett, kivétel az az eset, amikor konstans statikus adattagokról van szó. Ebben az esetben az osztályon belül tudunk nekik kezdő értéket adni. Konstruktor és destruktor soha nem lehet static.

```
class A
{
	int k = 0;
	static int i = 0; // hiba, nem engedélyezett az inline inicializálás statikus adattagok esetében, helyette az osztályon kivül tudunk nekik kezdő értéket adni.
	static int x; // helyes
	const static int j = 0; // lefordul

	// statikus metódus
	static int f() { return i; } // nem statikus függvényből elérhetőek statikus és nem statikus adattagok is
	static int g() { return k; } // hiba, mivel statikus metódusból csak statikus adattagokat tudunk elérni, ennek oka, hogy ezek már fordítási időben létrejönnek, míg a nem statikus adattagok csak akkor, amikor példányosítjuk az osztályt. A 'return k;' az implicit 'return this->k;' hívásnak felelne meg, ám osztályszintű metódus
```

```
      esetében nincs értelmezve a 'this' pointer, hiszen nincs objektum.
      Statikus metódus eléréséhez használjuk az osztály nevét és a scope
      operátort (::) pl. A::g();
};

int A::x = 0; // helyes, forduló kód
```

## Destruktorok

Gyakran előfordul, hogy egy objektum létrehozása során erőforrásokat (memória, állomány stb.) foglalunk le, amelyeket az objektum megszűnésekor fel kell szabadítanunk. Ellenkező esetben ezek az erőforrások elvesznek a programunk számára.

A C++ nyelv biztosít egy speciális tagfüggvényt – a destruktort – amelyben gondoskodhatunk a lefoglalt erőforrások felszabadításáról. A destruktor nevét a hullám karakterrel (~) egybeépített osztály névként kell megadni. A destruktor a konstruktor-hoz hasonlóan nem rendelkezik visszatérési típussal.

```
class A
{
public:
      A();  // konstruktor
      ~A(); // destruktor
};
```

# Bevezetés az STL-be (Standard Template Library)

A C++ nyelv Szabványos Sablonkönyvtára osztály- és függvénysablonokat tartalmaz, amelyekkel elterjedt adatstruktúrákat (vektor, sor, lista, halmaz stb.) és algoritmusokat (rendezés, keresés, összefésülés stb.) építhetünk be a programunkba.

A sablonos megoldás lehetővé teszi, hogy az adott néven szereplő osztályokat és függvényeket (majdnem) minden típushoz felhasználhatjuk, a program igényeinek megfelelően.

Az STL alapvetően három csoportra épül, a konténerekre (tárolókra), az algoritmusokra és az iterátorokra (bejárókra). Az algoritmusokat a konténerekben tárolt adatokon hajtjuk végre az iterátorok felhasználásával.

## STL konténerek

A konténerek olyan objektumok, amelyekben más, azonos típusú objektumokat (elemeket) tárolhatunk. A tárolás módja szerint a konténereket három csoportba sorolhatjuk. **Szekvenciális (sequence)** tárolóról beszélünk, amikor az elemek sorrendjét a tárolás sorrendje határozza meg. Ezzel szemben az adatokat egy kulccsal azonosítva tárolják az **asszociatív (associative)** konténerek, melyek tovább csoportosíthatjuk a kulcs alapján rendezett **(ordered)**, illetve a nem rendezett **(unordered)** tárolókra. Rendezett asszociatív tárolók esetében a default rendezés a növekvően rendezett.

**Szekvenciális konténerek:**

1. **Vektor –** dinamikus tömbben tárolódik folytonos memóriaterületen, amely a végén növekedhet. A elemeket indexelve is elérhetjük konstans *O(1)* idő alatt. Elem eltávolítása ( *pop_back()* ), illetve hozzáadása ( *push_back()* ) a vektor végéhez szintén *O(1)* időt igényel, míg az elején vagy a közepén ezek a műveletek ( *insert(), erase()* ) *O(n),* azaz lineáris végrehajtású idejűek. Rendezetlen vektorban egy adott elem megkeresésének ideje szintén *O(n).* Használatukhoz az **#include <vector>** szükséges. Vektor létrehozása **std::vector<típus amire szeretnénk példányosítani pl. int> vektornév.**
2. **Lista –** kettős láncolt lista, melynek elemei nem érhetőek el az indexelés operátorával. Tetszőleges pozíció esetén a beszúrás ( *insert()* ) és a törlés ( *erase()* ) művelete gyorsan, konstans *O(1)* idő alatt elvégezhető. A lista mindkét végéhez, adhatunk elemeket ( *push_front(), push_back()* ), illetve

törölhetünk onnan ( *pop_front(), pop_back()* ). Használatukhoz az **#include <list>** szükséges. Lista létrehozása **std::list<típus amire szeretnénk példányosítani pl. int> listanév.**

3. **Deque** – kettősvégű sor, amely mindkét végén növelhető, egydimenziós tömböket tartalmazó listában tárolódik. Elemeket mindkét végén konstans idő alatt adhatunk ( *push_front(), push_back()* ) a deque-hez, illetve távolíthatunk el onnan ( *pop_front(), pop_back()* ). Az elemek index segítségével is elérhetők. Használatukhoz az **#include <deque>** szükséges. Kettősvégű sor létrehozása **std::deque<típus amire szeretnénk példányosítani pl. int> dequenév.**

**Asszociatív konténerek:**

1. **Set / Multiset**
   - **Set** - minden elem legfeljebb **egyszer** szerepelhet, **mindig rendezett**, alapértelmezetten növekvő sorrendben. Beszúrás logaritmikus idejű *O(log(n))*, abban az esetben, ha a *mySet.insert(3)*-at használjuk, iterátorral a beszúrás konstans idejű *mySet.insert(iterator, 3)*. Keresés is logaritmikus idejű (műveletek kihasználják a rendezettséget.) Használatukhoz az **#include <set>** szükséges. Set létrehozása **std::set<típus amire szeretnénk példányosítani pl. int> setnév.**
   - **Multiset** - olyan, mint a set, de itt engedve vannak a duplikációk. Használatukhoz az **#include <set>** szükséges. Multiset létrehozása **std::multiset<típus amire szeretnénk példányosítani pl. int> multisetnév.**
2. **Map / Multimap** – olyan, mint a set / multiset azonban itt kulcs-érték párokat tárolunk. Az elemek indexelve vannak, nem feltétlen 0-tól és nem feltétlenül egymás utáni indexek. Kulcs alapján rendezett. A kulcsok értékét nem tudjuk módosítani. Keresés logaritmikus idejű *O(log(n))*.
   - **map** használatához az **#include <map>** szükséges. Map létrehozása **std::map<típus amire szeretnénk példányosítani pl. int> mapnév.**
   - **multimap** használatához az **#include <map>** szükséges. Multimap létrehozása **std::multimap<típus amire szeretnénk példányosítani pl. int> multimap.**
3. **Unordered Set / Unordered Map** – jó hash függvény esetén egy unordered konténerben a keresés konstans időt vesz igénybe. A beszúrás szintúgy. Unordered Set / Unordered Multiset-ben az adatok értéke nem változhat. Unordered Map / Unordered Multimap esetén a kulcsok értéke nem változtatható. Unordered Map-nek és az Unordered Multimap-nek nincs [] operátora.
   - **Unordered Set/Unordered Multiset** használatához az **#include <unordered_set>** szükséges.

    - **Unordered Set** létrehozása **std::unordered_set<típus amire szeretnénk példányosítani pl. int> unordered_setnév**
    - **Unordered Multiset** létrehozása **std::unordered_multiset<típus amire szeretnénk példányosítani> unordered_multisetnév**
- **Unordered Map/Unordered Multimap** használatához az **#include<unordered_map>** szükséges
    - **Unordered Map** létrehozása **std::unordered_map<típus amire szeretnénk példányosítani pl. int> unordered_mapnév**
    - **Unordered Multimap** létrehozása **std::unordered_multimap<típus amire szeretnénk példányosítani> unordered_multimap név**