5. fejezet

Konstruktorok, move szemantika, jobb-érték referenciák

Haladó Alkalmazott Programozás
ELTE, 2019

C++ inicializálás

C++ inicializálás

- C++ változókat sok féle képpen inicializálhatunk, például:

```
int i(2);
int i{2};
int i = 2;
```

Egyszerű, beépített típusoknál kb. mindegy, néhány esetben a () nem megy, az = eset extra másolással járhat.

Konstruktorok

Konstruktorok

Minden objektumhoz írhatunk több *konstruktor*t is, ezek azok a függvények, amik az objektum létrehozásakor elsőként lefutnak, és inicializálják az elemeket.

A konstruktorok jellemzője, hogy a nevük azonos az objektum nevével, és nincs visszatérési értékük!

Konstruktorok

- Default konstruktor:

```
struct Matrix2
{
  double a11, a12, a21, a22;

  Matrix2() : a11{0.0}, a12{0.0}, a21{0.0}, a22{0.0}
  {
  }
};
```

Konstruktorok

- Default konstruktor:

```
struct Matrix2
{
  double a11, a12, a21, a22;

  Matrix2() : a11{0.0}, a12{0.0}, a21{0.0}, a22{0.0}
  {
  }
};
```

A default konstruktornak nincs argumentuma

Konstruktorok

- Default konstruktor:

Most csak annyit tesz, hogy minden elemet nullára inicializál.

```
struct Matrix2
{
  double a11, a12, a21, a22;

  Matrix2() : a11{0.0}, a12{0.0}, a21{0.0}, a22{0.0}
  {
  }
};
```

Konstruktorok

- A Default konstruktor például akkor hívódik meg, ha semmit nem írunk ki létrehozáskor:

```
Matrix2 m;
```

Azaz, objektumok sose maradnak inicializálatlanok
(kivéve, ha elrontottuk a default konstruktort!)

Konstruktorok

- Copy konstruktor:

```
struct Matrix2
{
  double a11, a12, a21, a22;

  Matrix2( Matrix2 const& cpy ) : a11{cpy.a11}, a12{cpy.a12},
                                  a21{cpy.a21}, a22{cpy.a22}
  {
  }
};
```

Konstruktorok

- Copy konstruktor:

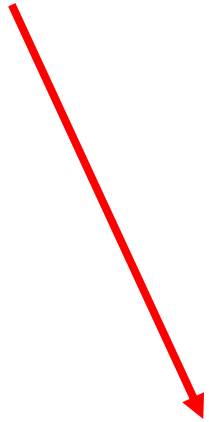
A copy konstruktor másolatot készít egy már létező ugyan ilyen objektumról

```
struct Matrix2
{
  double a11, a12, a21, a22;

  Matrix2( Matrix2 const& cpy ) : a11{cpy.a11}, a12{cpy.a12},
                                  a21{cpy.a21}, a22{cpy.a22}
  {
  }
};
```

Konstruktorok

- Copy konstruktor:

Most egyesével inicializálja az elemeket a másolatból



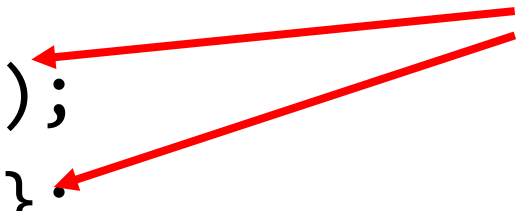
```
struct Matrix2
{
  double a11, a12, a21, a22;

  Matrix2( Matrix2 const& cpy ) : a11{cpy.a11}, a12{cpy.a12},
                                  a21{cpy.a21}, a22{cpy.a22}
  {
  }
};
```

Konstruktorok

- Copy konstruktor:

```
Matrix2 m1;
Matrix2 m2(m1);
Matrix2 m3{m1};
```

Ekkor a copy konstruktor hívódik meg.



Konstruktorok

- További konstruktorokat is írhatunk, pl. megadhatjuk a 4 elemet külön-külön:

```
struct Matrix2
{
  double a11, a12, a21, a22;

  Matrix2( double b11, double b12, double b21, double b22 )
   : a11{b11}, a12{b12}, a21{b21}, a22{b22}
  {
  }
};
```

Konstruktorok

- Ekkor írhatunk ilyet:

```
Matrix2 m1(1.0, 2.0, 3.0, 4.0);
Matrix2 m2{1.0, 2.0, 3.0, 4.0};
```

Konstruktorok

Figyelem!

- Ha egy osztálynak egyetlen konstruktora sincsen,
  akkor a fordító automatikusan generál copy, default konstruktorokat.
- Ekkor működik az aggregált inicializálás is alapból:

```
Matrix2 m2{1.0, 2.0, 3.0, 4.0};
```

- De amint írunk akár 1 darab konstruktort is, akkor ezeket már nem generálja le, nekünk kell megírni őket!

Move szemantika és jobb érték referenciák

Move szemantika és jobb érték referenciák

A C++-ban az értékeknek három féle kategóriájuk van:

Move szemantika és jobb érték referenciák

Wigner GPU Lab

Érték kategóriák: (C++11 óta)

- L-érték: bármilyen érték, ami névhez van kötve,
  értékül lehet adni neki (bal oldalán állhat az =-jelnek), van címe a memóriában
- Tiszta r-érték: átmeneti, még ***nem névhez kötött***!
  Pl.: egy leírt érték a kódban, vagy egy visszatérési érték függvény hívásból
  Nincs címe a memóriában, csak az = jel jobb oldalán állhat
- X-érték: megszűnő (eXpiring) érték, ***nincs névhez kötve***!
  Nincs címe a memóriában, csak az = jel job oldalán állhat
  Az erőforrásai újrahasználhatóak, mert mindjárt meg fog szűnni.
  Pl.: függvények, amik r-érték referenciát adnak vissza, `static_cast` (explicit átalakítás) r-érték referenciává, x-érték objektumok elemei
- Gl-érték: L-érték vagy X-érték → a `const`&-t váró függvényeket preferálják
- R-érték: X-érték vagy Tiszta R-érték → inkább a &&-t váró függvényeket preferálják a `const`& helyett

Move szemantika és jobb érték referenciák

A C++11 bevezetett egy újfajta referenciát, a jobb érték referenciát.

```
auto f(BigObject&& x)
```

Ez azt fejezi ki, hogy a függvény átveszi a tulajdonjogot az objektum felett, a hívó oldalon már nem használhatja / tulajdonolhatja senki.
ez általában csak egy pointer cserét jelent

Hívó oldal:

```
BigObject bo{...};

auto result1 = f( std::move(bo) );
auto result2 = f( makeBigObject(...) );
```

Itt expliciten átadjuk a tulajdonjogot a függvénynek

Ugyan ez történik, ha az objektum egyik fv hívásból a másikba kerül át

Move szemantika és jobb érték referenciák

Miért fontos ez?

A jobb érték referenciákkal elkerülhetőek felesleges memória foglalások és felszabadítások!

Move szemantika és jobb érték referenciák

Ha írunk egy nagy Vektor osztályt:

```
template<typename T>
struct Vector
{
  std::vector<T> data;
};
```

A fordító most generál nekünk mindenféle alapértelmezett konstruktorokat!

És hozzá egy összeadás operátort:

```
template<typename T>
Vector<T> operator+( Vector<T> const& v1, Vector<T> const& v2 );
```

Move szemantika és jobb érték referenciák

Összeadás operátor:

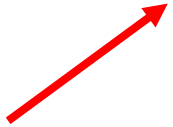
```
template<typename T>
Vector<T> operator+( Vector<T> const& v1, Vector<T> const& v2 );
```

Látjuk a szignatúrából, hogy mind a két bemenő vektort `const`& szerint vesszük át, tehát nem módosítjuk őket, ezért Vector-t visszaadni csak úgy tudunk, ha egy újat foglalunk le.
(megint képzeljünk el 10 GB adatot!)

Move szemantika és jobb érték referenciák

Összeadás operátor:

```
template<typename T>
Vector<T>&& operator+( Vector<T>&& v1, Vector<T> const& v2 );
```



Ebben az esetben viszont az első Vector egy jobb érték referencia!
Ez azt jelenti, hogy ez a Vector a hívó oldalon éppen fel fog szabadulni, nem fog rá hivatkozni senki, ezért az ő memóriáját újra felhasználhatjuk arra, hogy az eredmény Vector elemeit tárolja!

Move szemantika és jobb érték referenciák


```
template<typename T>
Vector<T> operator+( Vector<T> const& v1, Vector<T> const& v2 )
{
  Vector<T> result; result.data.resize(v1.data.size());
  std::transform(v1.data.begin(),
                 v1.data.end(),
                 v2.data.begin(),
                 result.data.begin(),
                 [](T const& x, T const& y){ return x+y; });
  return result;
}
```

Move szemantika és jobb érték referenciák


```
template<typename T>
Vector<T> operator+( Vector<T> const& v1, Vector<T> const& v2 )
{
  Vector<T> result; result.data.resize(v1.data.size());
  std::transform(v1.data.begin(),
                 v1.data.end(),
                 v2.data.begin(),
                 result.data.begin(),
                 [](T const& x, T const& y){ return x+y; });
  return result;
}
```

Új Vector jön létre.

Move szemantika és jobb érték referenciák

```
template<typename T>
Vector<T>&& operator+( Vector<T>&& v1, Vector<T> const& v2 )
{
  std::transform(v1.data.begin(),
                 v1.data.end(),
                 v2.data.begin(),
                 v1.data.begin(),
                 [](T const& x, T const& y){ return x + y; } );
  return std::move(v1);
}
```

Move szemantika és jobb érték referenciák


```
template<typename T>
Vector<T>&& operator+( Vector<T>&& v1, Vector<T> const& v2 )
{
  std::transform(v1.data.begin(),
                 v1.data.end(),
                 v2.data.begin(),
                 v1.data.begin(),
                 [](T const& x, T const& y){ return x + y; } );
  return std::move(v1);
}
```

A korábbi v1-et használjuk újra!

Move szemantika és jobb érték referenciák


```
template<typename T>
Vector<T>&& operator+( Vector<T>&& v1, Vector<T> const& v2 )
{
   std::transform(v1.data.begin(),
                  v1.data.end(),
                  v2.data.begin(),
                  v1.data.begin(),
                  [](T const& x, T const& y){ return x += y; }
);
   return std::move(v1);
}
```

Ebben a függvényben v1 már bal érték!
Ahhoz, hogy visszaadáskor ne másolat adódjon vissza, hanem ez, ki kell move-olni innen!

Move szemantika és jobb érték referenciák

Példák:

```
auto make_vct(){ return Vector<int>{{3, 4, 5}}; }

Vector<int> v1{{1, 2, 3}};
Vector<int> v2{{10, 20, 30}};

auto w1 = v1 + v2;
auto w2 = make_vct() + v2;
```

Move szemantika és jobb érték referenciák

Példák:

```
auto make_vct(){ return Vector<int>{{3, 4, 5}}; }

Vector<int> v1{{1, 2, 3}};
Vector<int> v2{{10, 20, 30}};

auto w1 = v1 + v2;
auto w2 = make_vct() + v2;
```

Ez az első változatot fogja hívni,
új allokáció történik, mert v1, v2
bal érték referenciák (van címük)!

Move szemantika és jobb érték referenciák

Példák:

```
auto make_vct(){ return Vector<int>{{3, 4, 5}}; }

Vector<int> v1{{1, 2, 3}};
Vector<int> v2{{10, 20, 30}};

auto w1 = v1 + v2;
auto w2 = make_vct() + v2;
```

Ez viszont a másodikat hívja, mert a függvényből vissza adott értéknek nincs neve, nem tudjuk a címét venni, az csak egy átmeneti változó, itt rögtön meg fog szűnni, amint belemegy a + operátorba!

Nem történik tehát új allokáció!

Move szemantika és jobb érték referenciák

Lehetőségünk van arra, hogy kézzel kényszerítsünk egy objektumot arra, hogy jobb értékké váljon, így tartalma újra felhasználódjon:

```
Vector<int> v1{{1, 2, 3}};
Vector<int> v2{{10, 20, 30}};

auto w1 = std::move(v1) + v2;
```

Move szemantika és jobb érték referenciák

Lehetőségünk van arra, hogy kézzel kényszerítsünk egy objektumot arra, hogy jobb értékké váljon, így tartalma újra felhasználódjon:

```
Vector<int> v1{{1, 2, 3}};
Vector<int> v2{{10, 20, 30}};

auto w1 = std::move(v1) + v2;
```

Viszont innentől kezdve `v1` egy „üres” állapotban van!
Nem lehet használni, amíg újra nem inicializáltuk!

Move konstruktor

A jobb érték referenciát váró konstruktor a move konstruktor:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

  Vector( Vector<T>&& mv );
};
```

Move konstruktor

A jobb érték referenciát váró konstruktor a move konstruktor:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

  Vector( Vector<T>&& mv );
};
```

Ez fut le mindig, ha nem jobb értékből inicializálunk (pl. egy függvény hozza létre a Vector-t, és azzal inicializálunk egy másikat)

Ekkor mv memóriáját újra használhatjuk!

Move konstruktor

A jobb érték referenciát váró konstruktor a move konstruktor:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

  Vector( Vector<T>&& mv )
  {
    std::swap(data, mv.data);
  }
};
```

A move konstruktortól általában azt várjuk, hogy vegye át másolás nélkül az adatokat a másik objektumból, és hagyja azt üresen.

Ezt sokszor könnyen megvalósíthatjuk egy swap-al, ami csak kicseréli a pointereket a háttérben.

operator=

Ha írunk copy és/vagy move konstruktorokat,
akkor általában írnunk kell értékadás operátort is:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

  Vector<T>& operator=(Vector<T> const& cpy );
  Vector<T>& operator=(Vector<T> &&     mv  );
};
```

operator=

Ha írunk copy és/vagy move konstruktorokat,
akkor általában írnunk kell értékadás operátort is:

```
Vector<T>& operator=(Vector<T> const& cpy );
Vector<T>& operator=(Vector<T> &&     mv  );
```

A legfontosabb dolog, amit ne felejtsünk el, hogy ne történjen váratlan dolog, ha a felhasználó véletlenül azt írja, hogy:

```
v = v;
```

operator=

Ezért általában ezek az operátorok azzal kezdődnek, hogy kizárják a saját magukkal való egyezést:

```
Vector<T>& operator=( Vector<T> const& cpy )
{
  if(&cpy == this){ return *this; }
  data = cpy.data;
  return *this;
}
```

operator=

Ezért általában ezek az operátorok azzal kezdődnek, hogy kizárják a saját magukkal való egyezést:

```
Vector<T>& operator=( Vector<T>&& mv )
{
  if(&mv == this){ return *this; }
  data = std::move(mv.data);
  return *this;
}
```

Destruktor

Ha valamilyen műveletet kell végeznünk felszabadításkor, akkor azt a destruktorban lehet megtenni:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

  ~Vector();
};
```

Destruktor

Ha valamilyen műveletet kell végeznünk felszabadításkor, akkor azt a destruktorban lehet megtenni:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

  ~Vector();
};
```

A destruktornak nincs argumentuma, ez az utolsó függvény, ami végrehajtódik az objektumon.

Ha allokáltunk memóriát kézzel, vagy valamit le kell zárnunk, itt érdemes megtenni.

Érdemes ismerni a három/öt/nulla szabályt:

- Ha valamiért kell írnunk saját copy konstruktort, copy=-t, vagy destruktort, akkor valószínűleg mind a háromra szükségünk van.
- Ha ezek vannak, akkor a fordító nem generál move konstruktort, és move= operátort, ezért ezeket is meg kell írnunk.
- A copy-move logikát válasszuk le másik objektumba, ha lehet, és akkor azokban a másik objektumokban, ahol csak használni akarjuk, nem kell implementálnunk őket (ha nem kell más konstruktor)

Lehet kérni a fordítót, hogyha egyértelmű, akkor generálja nekünk le az adott konstruktort, destruktort, operator=-t, stb-t a `default` kulcsszóval:

```
struct Vector
{
  std::vector<T> data;

  Vector(Vector<T> const& ) = default;
};
```

Egyéb operátorok:

operator[]

Indexelés:

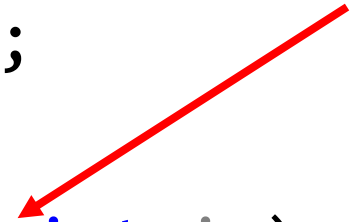
```
template<typename T>
struct Vector
{
  std::vector<T> data;

 T&       operator[]( int i )       { return data[i]; }
 T const& operator[]( int i ) const { return data[i]; }
};
```

operator[]

Indexelés:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

 T&       operator[]( int i )       { return data[i]; }
 T const& operator[]( int i ) const { return data[i]; }
};
```

Az indexelő operátornak kötelezően csak 1 argumentuma lehet!
(annak akármi lehet a típusa)



operator[]

Indexelés:

```
template<typename T>
struct Vector
{
  std::vector<T> data;

 T&       operator[]( int i )       { return data[i]; }
 T const& operator[]( int i ) const { return data[i]; }
};
```

Azért kell két változat,
mert az elsőn keresztül írni is lehet, a
másodikon keresztül viszont csak olvasni,
csak ez utóbbi működik `const` Vectorokon!

operator()

Gömbölyű zárójel operátor:

```
struct X
{
  int operator()(int x, int y, int z){ return x+y+z; }
};
```

Segítségével függvény szerűen működő objektumokat lehet létrehozni. Bármi és bármennyi argumentuma lehet, és a vissza térési értéke is bármi lehet.

operator()

A lambda függvény nem más, mint egy inline definiált struct, aminek a lambdában megadott függvény a gömbölyű zárójel operátora:

Ez a lambda: `[](int x){ return x*x; }`

Ezzel a struktúra példánnyal ekvivalens:

```
struct Unnamed
{
  auto operator()(int x)const{ return x*x; }
} unnamed;
```

operator()

A lambda capture része tulajdon képpen a struktúra elemeit definiálja, és egyben a konstruktor is:

```
[y = 5](int x){ return x + y; }
```

Ezzel a struktúra példánnyal ekvivalens:

```
struct Unnamed
{
  int y;
  auto operator()(int x)const{ return x + y; }
} unnamed{ 5 };
```

operator()

A lambda capture része tulajdon képpen a struktúra elemeit definiálja, és egyben a konstruktor is:

```
[y = 5](int x){ return x + y; }
```

y típusa az = jel jobb oldalából számolódik ki!

Ezzel a struktúra példánnyal ekvivalens:

```
struct Unnamed
{
  int y;
  auto operator()(int x)const{ return x + y; }
} unnamed{ 5 };
```

y itt kap értéket!

operator()

A generikus lambda függvények (C++14) álruhás osztály template-k.

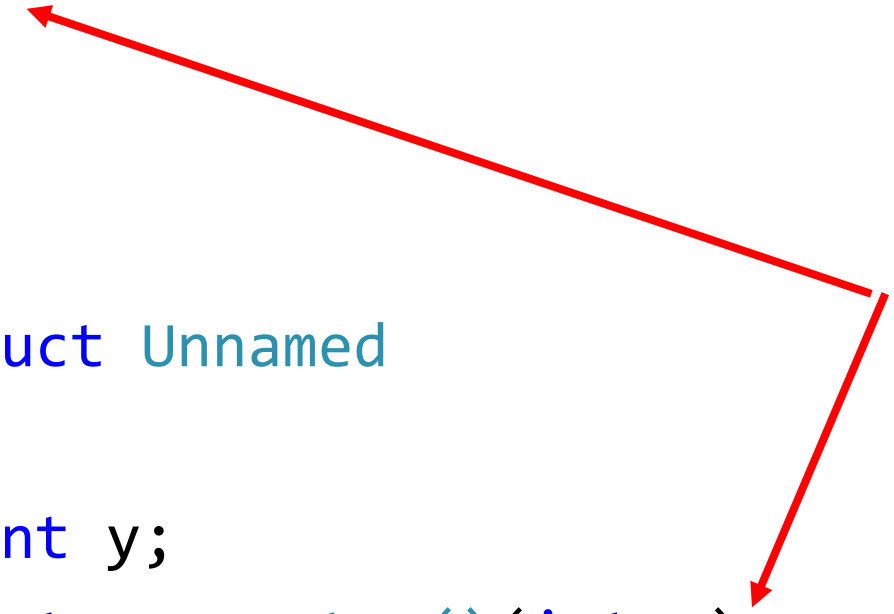
Ez a lambda: `[](auto x){ return x*x; }`

Ezzel a struktúra példánnyal ekvivalens:

```
struct Unnamed
{
   template<typename T>
   auto operator()(T x)const{ return x*x; }
} unnamed;
```

operator()

```
[y = 5](int x){ return x + y; }
```

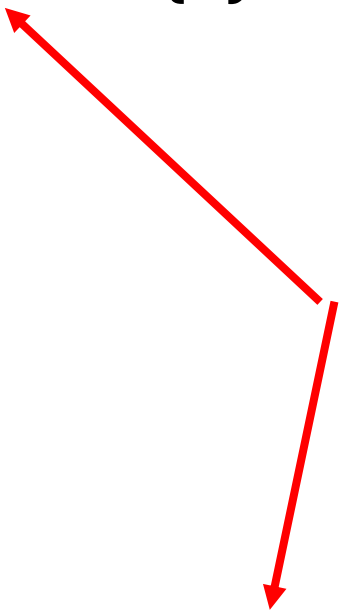


y most nem módosítható!
A const miatt!

```
struct Unnamed
{
  int y;
  auto operator()(int x)const{ return x + y; }
} unnamed{ 5 };
```

operator()


```
[y = 5](int x) mutable { y += 1; return x + y; }
```



Ha azt szeretnénk, hogy a lambda által átvett értékek módosíthatóak legyenek, ki kell írnia `mutable` kulcsszót.

```
struct Unnamed
{
  int y;
  auto operator()(int x)      { y += 1; return x + y; }
} unnamed{ 5 };
```

operator()

Mivel a [] operátornak csak 1 argumentuma lehet, ezért ha mátrixot, vagy más több dimenziós dolgot szeretnénk csinálni, sokszor a () zárójelet használjuk indexelésre.

operator()

Mátrix konvenció, indexelés

```
template<typename T>
struct Matrix
{
  int N, M;
  std::vector<T> data;

  T&       operator()(int i, int j)
  { return data[N*i+j]; }

  T const& operator()(int i, int j) const
  { return data[N*i+j]; }

};
```

Kiegészítés a mátrix osztályhoz

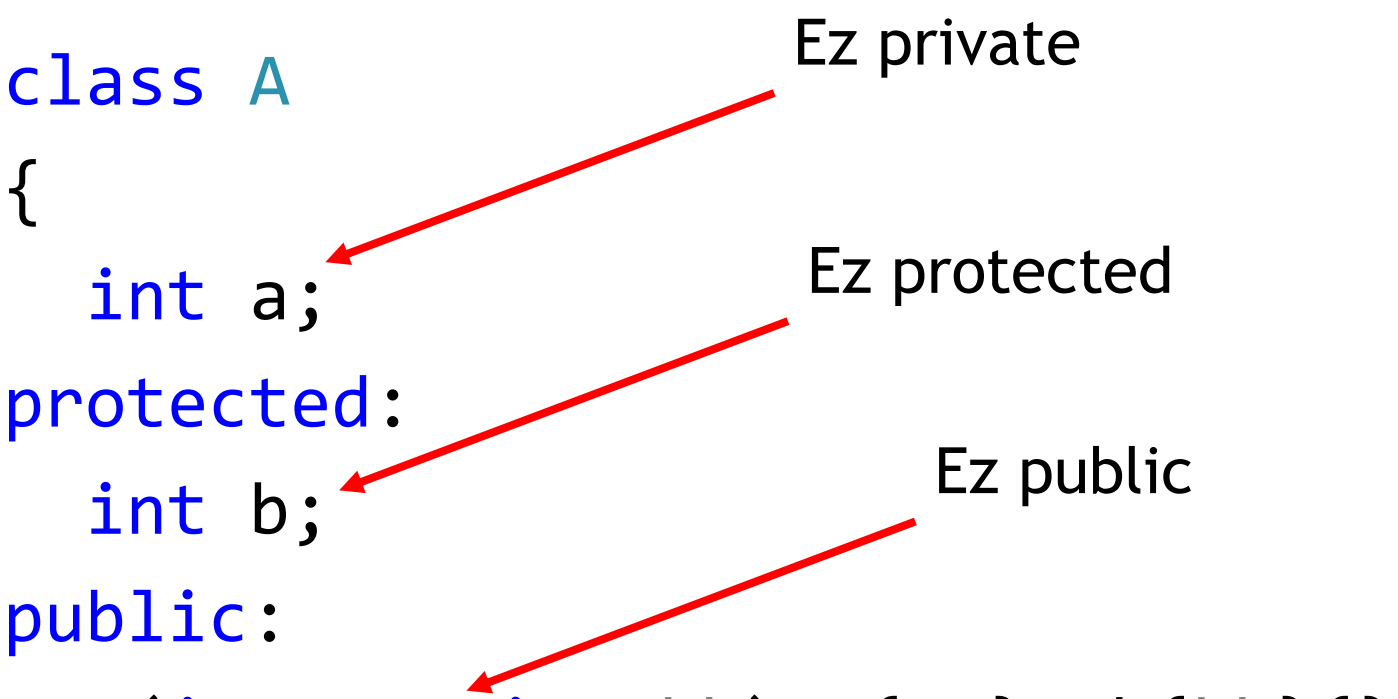
A mátrixnál már van belső összefüggés, amit a felhasználó elronthat, ha módosíthatja a belső változókat -> ezért ez már `class` kell legyen.

A `class` és `struct` elemeinek beállítható a láthatósága:

- `private` - csak saját maga látja a szimbólumokat
- `protected` - ő, és a belőle származtatott osztályok látják a szimbólumokat
- `public` - bárki hozzáférhet a szimbólumokhoz.

A struct elemei ha nem írunk ki semmit, akkor `public`-ok, a class elemei private-k.

Kiegészítés a mátrix osztályhoz




```
class A
{
   int a;
protected:
   int b;
public:
   A(int aa, int bb):a{aa}, b{bb}{}
};
```

Kiegészítés a mátrix osztályhoz

Ha egy private vagy protected memberhez kell hozzáférést adni, akkor az a `friend` kulcsszóval lehet.

De, sok mindent meg lehet oldani kellően okos konstruktorokkal.

Kiegészítés a mátrix osztályhoz

Érdemes írni a mátrix osztályhoz:

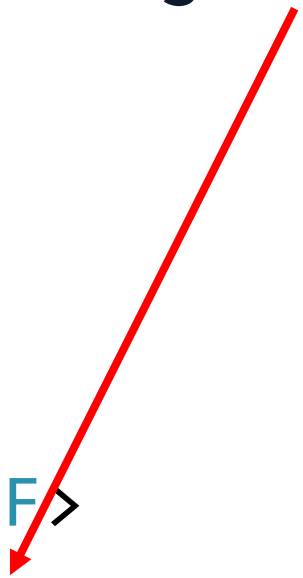
- operator[]-t az egy dimenziós indexeléshez
- operator()-t a két dimenziós indexeléshez

- Olyan méret lekérdező függvényt, ami a sorok/oszlopok számát adja vissza, és olyat, ami az elemek számát

- Olyan konstruktort, ami 1 argumentumot (indexet) váró függvényből és a méretből inicializálja a mátrixot
- És olyat, ami 2 indexből és a méretből…

Kiegészítés a mátrix osztályhoz

Utóbbi kettőt nem lehet könnyen elkülöníteni, mert mind a két konstruktornak ez lenne a szignatúrája:



```
template<typename T>
struct Matrix
{
  template<typename F>
  Matrix(F f, int N);
};
```

Kiegészítés a mátrix osztályhoz

Ilyen esetekben általában megoldás, hogy új, üres structokat vezetünk be, amelyeket átadunk a konstruktornak, csak azért, hogy kiválasszuk, hogy melyiket hívjuk:

```
struct Idx1{};
struct Idx2{};

template<typename T> struct Matrix
{
  template<typename F>
  Matrix(Idx1, F f, int N);

  template<typename F>
  Matrix(Idx2, F f, int N);
};
```

Kiegészítés a mátrix osztályhoz

Példák konstruktorok használatára:

```
template<typename T>
Matrix<T> operator/(Matrix<T> const& A, T const& s)
{
  return Matrix<T>(Idx1{},
                   [&](auto i){ return A[i] / s; },
                   A.size());
}
```

Kiegészítés a mátrix osztályhoz

Példák konstruktorok használatára (négyzetes mátrix):

```
template<typename T>
Matrix<T> operator*(Matrix<T> const& A, Matrix<T> const& B)
{
  auto N = A.size();

  return Matrix<T>(Idx2{}, [&](int i, int j)
  {
    T sum = 0.0;
    for(int k = 0; k<N; ++k){ sum += mat1(i, k) * mat2(k, j); }
    return sum;
  }, N);
}
```

Házi feladat

Írjuk meg a nagy mátrix osztályt (legyen az egyszerűség kedvéért négyzetes), ami std::vector-ban tárolja az elemeket, lehet indexelni, és megvannak rá írva az alapvető műveletek (+, -, skalárral * , skalárral /, egymással szorzás) és ezeknek az operátoroknak az összes &&-es változatai is.

Külön matrix.h header-t írjunk!

Minden műveletet és művelet esetet külön teszteljünk le, hogy helyesen működik-e!

Példa: Vector osztály.

Szorgalmi: írjuk meg, vagy másik kurzusból emeljük át a mátrixot invertáló kódot egy belső tagfüggvénybe (inv), és definiáljuk vele a mátrix-mátrix osztást.