Konstruktorok és destruktorok. Inicializálás módjai bővebben: inicializálás másolással és mozgatással.

C++ programozás – 4. óra
Széchenyi István Egyetem

`http://dropbox.com/...`

2017

Konstruktorok típusai

- Ahogy beszéltük, objektumok létrezohása / inicializálása célszerűen **konstruktorokkal** történik
  - Amint definiáltunk konstruktort, kötelező használni (ha többet definiáltunk, akkor valamelyiket)
  - Argumentum nélkül is csak akkor hozhatunk létre objektumot, ha létezik ún. default (argumentum nélküli) konstruktor
- A konstruktoroknak:
  - neve ugyanaz, mint az osztályé
  - nincs visszatérési értékük
- Konstruktorok / destruktorok működését életbevágóan fontos, hogy megértsük!
  - Stroustrup papa szerint aki ezt a témát érti, az tud C++-ban programozni

Konstruktorok típusai

- Mi történik a példában?

```
string ident(string arg) //string ertek szerinti masolasa
{
  return arg; //eredmeny mozgatasa a valaszba
}

int main()
{
  string s1 {"Adams"}; //string inicializalasa
  s1 = ident(s1); //s1 masolasa ident() fv-hez
  //eredmeny mozgatasa s1-be

  string s2 {"Pratchett"}; //string inicializalasa
  s1 = s2; //s2 ertekenek atmasolasa s1-be
}
```

Konstruktorok típusai II

- A fedél alatt többféle konstruktort hívunk meg:
  - "sima" **konstruktor** string inicializálására (kétszer)
  - **copy constructor**, amely bemásolja a string-et a fv argumentumába
  - **move constructor**, amely kimozdítja a string-et az ident fv kimenetére
  - **move assignment** az ident fv kimenetét tároló átmeneti változóból s1-be
  - **copy assignment** s1 s2-re állítására
  - **destruktor** az s1, s2 illetve ident fv arg változójának felszabadítására.
- Annyira hozzászoktunk, hogy fel sem tűnik!
- A string osztály esetén ez nem gond, mert minderről az osztály automatikusan gondoskodik!
- Saját osztályok tervezésekor azonban nekünk (a programozónak) kell mindenről gondoskodnunk!

Konstruktorok

- Sok szabály és technikai részlet van... Legtöbb programozó ezeket példákból kiindulva sajátítja el.
- Konstruktor feladata az objektum inicializálása olyan módon, hogy a szükséges invariancia-tulajdonságok teljesüljenek:

```
class Vector
{
public:
  Vector(int s);
private:
  double* elem; //elem egy sz darab double-t tartalmazo tombre mutat
  int sz; //sz nem negativ
};
```

- Itt a kommentek = invariancia-tulajdonságok
- A konstruktor garantálja, hogy *sz* nemnegatív, és *elem* nem nullptr

Konstruktorok II

```
Vector::Vector(int s)
{
  if(s < 0)
  {
    throw Bad_size(s);
  }

  sz = s;
  elem = new double[s];
}
```

- Az invariánsok fontosak!
  - Behatárolják, milyen eseteket kell megvalósítania az osztálynak
  - Világossá teszik, mi az osztály funkciója
  - Egyszerűbbé teszik/tehetik a metódusok definícióját
  - Világossá teszik az osztály erőforrás-felhasználását
  - Egyszerűbbé teszik az osztály dokumentálását

Destruktorok

- Előfordul, hogy egy osztály olyan erőforrásokat hoz létre és használ, amelyeket az objektum megszűnésekor el kell engedni
  - például handle file-eléréshez, zárak változók konkurrens eléréséhez, specifikus memória-területek
- Ezek felszabadítására való a destruktor, melynek neve '~'+osztálynévből tevődik össze (pl. ~*Vector*())
- Destruktornak sosincs paramétere, és osztályonként csak egy lehet!

```
class Vector
{
public:
  Vector(int s) : elem(new double[s]), sz(s){} //konstruktor: memoria lefoglalasa
  ~Vector(){ //destruktor: memoria felszabaditasa
    delete[] elem;
  }
private:
  double* elem;
  int sz;
};
```

Destruktorok II

- Az alábbi példában kétszer kerül meghívásra a destruktor

- Ha a konstruktor nem tud memóriát foglalni, *std* :: *bad_alloc* típusú kivételt dob és a már lefoglalt memóriát felszabadítja (RAII)
- Általában véve: ha a programozó definiálja egy osztály destruktorát, azt is el kell döntenie, hogy másolható-e, illetve elmozdítható-e az objektum (rule of 3)

```
Vector* f(int s)
{
  Vector v1(s);
  //...
  return new Vector(s+s);
}

ing g(int ss)
{
  Vector* p = f(ss);
  //...
  delete p;
}
```

Destruktorok III

- Destruktorokat az esetek túlnyomó többségében nem mi hívjuk meg, a futtatási környezet magától megoldja (ha kikerül a scope-ból, vagy meghívjuk a delete operátort).
- Nagyon ritkán előfordul, hogy mégis mi hívjuk meg: pl. ha az ún. placement new operátort használjuk

```
void* operator new(size_t, void* p){return p;} //explicit placement operator

void* buf = reinterpret_cast<void*>(0xF00F); //valamiert fontos ez a cim
X* ptr = new(buf) X; //oda rakjuk az uj X tipusu valtozot

void C::push_back(const X& a){
  //...
  new(ptr) X(a); //copy konstruktor altal a erteku X-et teszunk p cimre
  //...
}

void C::pop_back(){
  ptr->~X();
}
```

Virtuális destruktorok

- Később lesz szó dinamikus polimorfizmusról
- Destruktorok lehetnek virtuálisak, és virtuális fv-nyel rendelkező osztály esetében általában annak kell lenniük.

```
class Shape
{
public:
  //...
  virtual void draw() = 0;
  virtual ~Shape();
};
```

```
class Circle : public Shape
{
public:
  //...
  void draw();
  ~Circle(); //felultolti ~Shape()-et
};
```

- Ha az interfészen keresztül törlünk, jó hogy a destruktor virtuális

```
void user(Shape* p)
{
  p->draw();
  //...
  delete p; //a megfelelo destruktort (a konkret osztalyet) hivjuk meg!
}
```

Osztály példányainak inicializálása

- Konstruktor nélküli inicializáció: beépített típusokhoz nem adhatunk konstruktort, mégis megfelelő típussal inicializálhatjuk:

```
int a = 1;
int x {2};
char* p {nullptr};
```

- Hasonlóan, konstruktor nélkül is rendelkezésünkre állnak a következő módszerek:
  - **Tagonkénti inicializálás** – akkor, ha egyébként semmilyen konstruktort nem definiáltunk
  - **Copy inicializálás** – akkor, ha nem definiáltunk ún. copy constructort, vagy nem tiltottuk meg a másolást (ld. később)
  - **Default inicializálás** – akkor, ha egyébként semmilyen konstruktort nem definiáltunk, vagy ha a definiált konstruktorok között van olyan is, melynek argumentum-listája üres

Osztály példányainak inicializálása II

- Például:

```
struct Work
{
  string author;
  string name;
  int year;
};

Work s9 {"Beethoven", "Symphony no 9 in d op. 125", 1824}; //tagonkenti inicializalas
Work now_playing {s9}; //copy inicializalas
Work none {}; //default inicializalas
```

- *none* változóinak értéke “”, “” és 0
- Mindenesetre biztonságosabb, ha nem támaszkodunk erre
  - nem statikusan allokált objektumok (tehát stack vagy heap változók) esetén default inicializálás csak osztály-típusokra van, beépített típusoknál definiálatlan!

Osztály példányainak inicializálása III

- Ennek egyébként hatékonysági okai vannak. Az alábbi esetben pl. felesleges inicializálni a hatalmas karaktertömböt, ha úgyis beolvasunk oda valamit:

```
struct Buf{
  int count;
  char buf[16*1024];
};
```

- Mindezt nem kell feltétlenül megjegyeznünk, ha inkább a jól bevált módon konstruktorokat használunk.
  - úgy legalább a kódunk is jobban olvasható lesz!
- Viszont: **ha akár egy konstruktort is definiálunk, utána már csak konstruktorral inicializálhatunk!**

Osztály példányainak inicializálása IV

```
struct X{
  X(int);
};
x x0; //hiba: nincs inicializalo
X x1 {}; //hiba: nincs parameter
X x2 {2}; //OK
X x3 {"ketto"}; //hiba: rossz tipusu parameter
X x4 {x2}; //OK, mert a copy konstruktor implicite letezik, ld. kesobb
```

- Persze ilyenkor ha default konstruktor nincs is, copy konstruktor még mindig marad, hiszen az objektum tagjai másolhatóak.
- Olyan esetekben, amikor ez problémát okoz, letilthatjuk a másolást. Pl. amikor egy objektumban erőforrás handle-ek vannak, tipikusan katasztrófához vezet a copy konstrukció

```
void bad_copy(Vector& v1){
  Vector v2 = v1; //atmasolunk mindent v2-be
  v1[0] = 2; //v2[0] is 2, mivel a belso ptr ugyanoda mutat...
  v2[1] = 3; //v1[1] is 3!
}
```

Osztály példányainak inicializálása V

- Alapesetben definiált függvény tiltható a delete kulcsszóval:

```
class Base
{
  //...
  Base& operator=(const Base&) = delete; //nem engedjuk a masolast
  Base(const Base&) = delete;

  Base& operator=(Base&&) = delete; //nem engedjuk a mozgatast
  Base(Base&&) = delete;
};

Base x1;
Base x2 {x1}; //hiba: nincs copy konstruktor
```

- C++11-es szabvány előtt nem volt tagonkénti inicializálás inicializáló-listával, csak () és = operátorok voltak használhatóak. Ezek viszont nem univerzális jelentésűek.

Osztály példányainak inicializálása VI

```
struct Y : X
{
  X m;
  Y(int a) : X(a), m=a {} //szintaxis hiba: nem hasznalhato = taginicializalashoz
};

X g(1); //globalis valtozo inicializalasa

void f(int a)
{
  X def(); //fv ami X-et ad vissza
  X* p {nullptr};
  X var = 2; //lokalis valtozo inicializalasa
  p = new X = 4; //szintaxis hiba: new-val nem hasznalhato =-jel
  X a[](1,2,3); //hiba: nem hasznalhato () tomb-inicializalashoz
  vector<X> v(1,2,3,4); //hiba: vektorelemek inicializalashoz nem jo ()
}
```

- Régebben csak a ()-operátor használatával garantálhattuk, hogy konstruktor hívódjon meg. Ma már ez nem szükséges.

Osztály példányainak inicializálása VII

```
struct S1{
  int a,b; //nincs konstruktor
};

struct S2{
  int a,b;
  //konstruktor
  S2(int a=0, int b=0) : a(aa), b(bb){}
};

S1 x11(1,2); //hiba: nincs konstruktor
S1 x12 {1,2}; //OK: tagonkenti
      inicializacio
```

```
S1 x13(1); //hiba: nincs konstruktor
X1 x14 {1}; //OK. b erteke 0 lesz

S2 x21(1,2); //OK
S2 x22 {1,2}; //OK

S2 x23(1); //OK. Masodik parameter erteke default
S2 x24 {1}; //OK. Masodik parameter erteke default
```

- Ha van megfelelő konstruktor, a kapcsos zárójel mindig ugyanúgy jó
  - (nagyon ritka kivételektől eltekintve, ellenpéldára ld. következő fólia)
  - csak ráadásul megóv az implicit szűkítéstől (ahogy mondtuk).

Osztály példányainak inicializálása VIII

```
vector<int> v1 {77}; //egy elemu vektor, az elem erteke 77
vector<int> v2(77); //77 elemu vektor 0 default ertekkel
```

- Vektor osztálynál tehát van konstruktor és tagonkénti inicializálásra is lehetőség, de az utóbbi is lényegében egy konstruktor
- Argumentum nélküli default konstruktor gyakori:

```
class Vector{
public:
  Vector();
  //...
};
class String{
public:
  String(const char* p = ""); //default argumentum is OK
};

Vector v1; //OK
Vector v2 {}; //OK
String s1; //OK
String s2 {}; //OK
```

Osztály példányainak inicializálása IX

- Beépített típusokhoz van default és copy konstruktor.
  - Viszont a default konstruktor nem-statikus és inicializálatlan változóknál nem hívódik meg (ld. korábban *char*[] buf példáját).

```
void f()
{
  int a0; //inicializalatlan
  int a1(); //fv deklaracio. Tenyleg ezt akartuk?

  int a {}; //a erteke 0
  double d {}; //d erteke 0.0
  char* p {}; //p erteke nullptr

  int* p1 = new int; //nem inicializalt int
  int* p2 = new int{}; //int nullara inicializalva
}
```

Osztály példányainak inicializálása X

- Mikor használunk beépített típushoz konstruktort?
- Leginkább template-eknél

```
template<class T>
struct Handle{
  T* p;
  Handle(T* pp = new T{}) : p{pp} {}
};

Handle<int> px; //letre fog hozni egy int{}-et
```

Osztály példányainak inicializálása XI

- Invariánsok és erőforrások létrehozásához leggyakrabban tagok és szülők inicializálása szükséges. A tagok konstruktorait tag-inicializáló listában szokás megadni:

```
class Club{
  string name;
  vector<string> members;
  vector<string> officers;
  Date founded;
  //...
  Club(const string& n, Date fd);
};

Club::Club(const string& n, Date fd)
  : name{n}, members{}, officers{}, founded{fd}
{//...}
```

- Ha egy tagváltozó *const* vagy referencia, akkor ráadásul nincs is más lehetőségünk, mint hogy inicializáló-listában adjunk neki értéket (ugyanis kötelező értékkel inicializálni)

Osztály példányainak inicializálása XII

- A tag-inicializáló lista elemei a törzs meghívása előtt kerülnek feldolgozásra, abban a sorrendben, ahogy megjelennek.
- A destruktorok fordított sorrendben hívódnak meg.
- Tag-inicializáló lista elkerülhetetlen, ha egy tag inicializálása nem ugyanazt jelenti, mint assignmentje (=-operátor).
  - Pl. referenciát illetve const-ot kötelező inicializálni, ezért ha nincs tag-inicializáló lista, a fordító panaszkodik:

```
class X
{
  const int i;
  Club c1;
  Club& rc;
  //...
  X(int ii, const string& n, Date d, Club& c)
    : i{ii}, c1{n,d}, rc{c}
  {}
};
```

Tagosztályok és szülők inicializálása

- Öröklést még nem vettünk, de nagyjából sejtjük, hogyan működik.
- Szülők inicializálása a tagváltozókkal analóg módon történik:

```
class B1{ B1();}; //van default konstruktora
class B2{ B2(int); }; //nincs default konstruktora

struct D1 : B1, B2 {
  D1(int i) : B1{}, B2{i} {}
};

struct D2 : B1, B2 {
  D2(int i) : B2{i} {} //B1{} implicit
};

struct D3 : B1, B2 {
  D3(int i) {} //hiba: B2-t int-tel kell inicializalni
};
```

Tagosztályok és szülők inicializálása II

- Tag-inicializáló lista a felesleges ismétlések elkerülésében is segíthet: ún. **konstruktor-delegációt** tesz lehetővé

```
class X{
  int a;
public:
  X(int x) { if (0<x && x<=max) a=x; else throw Bad_x(x);}
  X() : X{42} {} //konstruktor masik konstruktort hiv meg
  X(string s) : X{to<int>(s)} {} //konverzioval
};
```

- Egyidőben ugyanakkor nem delegálhatunk konstruktort és iniciaizálhatunk tagot:

```
class X{
  int a;
public:
  X(int x) {if (0<x && x<=max) a=x; else throw Bad_x(x);}
  X() : X{42}, a{56} {} //hiba
};
```

Tagosztályok és szülők inicializálása III

- Konstruktor-delegáció és törzsön belüli hívás merőben mást jelent

```
class X{
  int a;
public:
  X(int x) {if (0<x && x<=max) a=x; else throw Bad_x(x);}
  X(){ X{42}; } //hiba, valoszinuleg...
};
```

- A 2. esetben egyszerűen létrehoztunk egy teljesen új objektumot

Rule of 3

- Ha egy osztálynak definiálunk destruktort, az általában azt jelenti, hogy a megszűntetése nem triviális
  - pl. mert dinamikusan allokál területet, amit fel kell szabadítani.
- Ilyenkor igen nagy valószínűséggel szükségünk lesz:
  - Copy constructor-ra is, és
  - Copy assign constructor-ra is
- C++-11 óta szintén szükségünk lesz (rule of 5??):
  - Move constructor-ra is, és
  - Move assign constructor-ra is
- **Rule of 3 (5)**: Ha a 3 (5) közül bármelyiket definiáltuk, készítsünk a többire is valamilyen fv-t!
  - Ha nem tesszük, a fordító automatikusan létrehozza, de nem biztos hogy azt fogja csinálni, amit szeretnénk!

Rule of 3

- Így néznek ki:

```
class X{
  X(Sometype); //"normal" konstruktor
  X(); //default konstruktor
  X(const X&); //copy constructor
  X(X&&); //move constructor
  X& operator=(const X&); //copy assigment: cel megtisztitasa es masolas
  X& operator=(X&&); //move assignment: cel megtisztitasa es mozgatas
  ~X(); //destruktor
  //...
};
```

- Lényegében 5 szituáció van, amikor objektum másolódik vagy mozgatódik:
  - Értékadás esetén (=)
  - Inicializálás esetén, pl. $Date\ d = Date\{2, 3, 2012\}$
  - Fv-argumentum, illetve fv-kimenet esetén
  - Kivétel létrehozásakor

Másolás és mozgatás

- Másolás, gyakorlatilag x = y konvencionális jelentése
- x = move(y) esetén x y korábbi értékét kapja meg, y pedig valamilyen moved-from állapottal rendelkezik (konténerek esetén ez egy "üres" állapot)
- Egyszerű különbség, csak sajnos a tradíciók és a jelölések hasonlósága zavart keltenek.
- Tipikusan egy mozgatás nem dobhat kivételt, de egy másolás igen, hiszen olyankor újabb erőforrásokat kellhet allokálni.
- Mozgatás esetén a "forrás" objektum nem maradhat invalid állapotban, mert a destruktornak meg kell rá hívódnia. Ezért fontos, hogy ne dobjanak kivételt, és assignmentet valamint destrukciót lehetővé tevő állapotban maradjanak.

Másolás

- Másoláshoz két operáció definiálható:
  - Copy constructor: $X(const X\&)$
  - Copy assignment: $X\&operator=(const X\&)$
- Ilyenkor módosítás nélkül másoljuk az objektumot. Példa:

```
template<class T>
class Matrix{
  array<int,2> dim; //ket dimenzio
  T* elem; //mutato dim[0]*dim[1] T tipusu elemre
public:
  Matrix(int d1, int d2) : dim{d1, d2}, elem{new T{d1*d2}} {} //egyszerusites, nincs hibakezeles
  int size() const {return dim[0]*dim[1];}

  Matrix(const Matrix&); //copy constructor
  Matrix& operator=(const Matrix&); //copy assignment

  Matrix(Matrix&&); //move constructor
  Mattrix& operator=(Matrix&&); //move assignment

  ~Matrix(){delete[] elem; }
};
```

Másolás II

- A default copy (inicializálással) itt katasztrofális lenne: a mátrix elemei nem másolódnának, csak a pointer maga.
- Programozóként ezzel együtt bármit csinálhatunk:

```
template<class T>
Matrix::Matrix(const Matrix& m)
  : dim{m.dim}, elem{new T[m.size()]}
{
  uninitialized_copy(m.elem, m.elem+m.size(), elem); //elemek masolasa
}
```

Másolás III

- A copy constructor és assignment közötti különbség:
  - Copy construktor inicializálatlan memóriaterületet használ
  - Copy assignment olyan objektumot ír felül, amely már létezik, és esetleg már rendelkezhet erőforrásokkal
- Vegyük észre, hogy ha assignmentkor egy elem másolása hibát dob, felemás állapotban lesz a másolás célja. Ez általában nem megengedhető, ezért először használjuk a copy constructor-t:

```
Matrix& Matrix::operator=(const Matrix& m)
{
  Matrix tmp{m}; //erdemes mindig eloszor masolni... ha a masolassal gond van, itt fog elhasalni a
        kod
  swap(tmp, *this); //ezutan a ket objektumot kicsereljuk. swap mar nem dob kivetelt
  return *this; //az eredmeny mindenkeppen konzisztens. ezt hivjak copy-and-swap idiomanak!
}
```

- Ez csak akkor jó, ha swap() nem assignmentet használ (az std::swap move-ot használ)

Másolás IV

- Végezetül: másoláskor fontos, hogy a szülőt és az összes tagot is másoljuk. Sose felejtsük el!

```
class X{
  string s;
  string s2;
  vector<string> v;

  X(const X& a)
    : s{a.s}, v{a.v} //gondatlan es valszeg hibas
  {}
};
```

- s2-ről elfelejtkeztünk...
- Másoláskor egyébként a szülő ugyanolyan, mintha tagváltozó lenne: annak másolásáról is gondoskodni kell!

Másolás V

- Ezért osztály-hierarchiában így másolunk:

```
struct B1{
  B1();
  B1(const B1&);
  //...
};

struct B2{
  B2(int);
  B2(const B2&);
  //...
};

struct D : B1, B2{
  D(int i) : B1{}, B2{i}, m1{}, m2{2*i} {}
  D(const D& a) : B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}
  B1 m1;
  B2 m2;
  //...
};

D d {1}; //letrehozas int argumentummal
D dd {d}; //copy construct
```

Mozgatás

- C++-11-es szabványban új funkció a mozgatás. Egyszerű adatok esetén a másolás trivi, de bonyolultabb osztályok esetén költséges lehet. Pl: mi történik ha 1000 karakteres stringeket másolunk?

```
template<class T>
void swap(T& a, T& b)
{
  const T tmp = a;
  a = b;
  b = tmp;
}

void f(string& s1, string& s2, vector<string>& vs1, vector<string>&vs2, Matrix& m1, Matrix& m2)
{
  swap(s1, s2);
  swap(vs1, vs2);
  swap(m1, m2);
}
```

Mozgatás II

- Nézzük meg, Mátrix osztály esetén mindez hogyan valósítható meg?

```
template<class T>
Matrix<T>::Matrix(Matrix&& a)
  : dim{a.dim}, elem{a.elem} //megragadjuk a reprezentaciojat
{
  a.dim = {0,0}; //a reprezentaciojat ezutan toroljuk
  a.elem = nullptr;
}
```

- Move assignment: destruktor végzi el a “piszkos munkát”

```
template<class T>
Matrix<T>& Matrix<T>::operator=(Matrix&& a)
{
  swap(dim, a.dim); //reprezentaciok kicserelese... majd a destruktor torol
  swap(elem, a.elem);
  return *this;
}
```

Mozgatás III

- Ahhoz azonban, hogy a fordító értse, hogy mikor szeretnénk mozgatást használni másolás helyett, explicit módon szükséges rvalue referenciát átadnunk a hívásnak:

```
template<class T>
void swap(T& a, T& b) //majdnem tokeletes swap
{
  T tmp = std::move(a);
  a = std::move(b);
  b = std::move(tmp);
}
```

- *std* :: *move*() jobboldali referenciát ad vissza, így egyértelmű a dolog. std::move(x) annyit jelent, hogy “adj nekem x-re egy jobboldali referenciát”. Valójában tehát nem mozgat semmit.

Hasznos tippek (Stroustruptól)

- Konstruktort, értékadást és destruktort együtt tervezzünk meg
- Konstruktorokkal hozzuk létre az osztály invariánsait
- Ha a konstruktor erőforrást foglal le, destruktorra is szükség van
- Ha egy osztálynak virtuális fv-e van, virtuális destruktorra is szükség van (nem tudhatjuk, hogy a gyermekben milyen erőforrások lesznek lefoglalva)
- Ha egy osztálynak nincs konstruktora, tagonkénti inicializálás lehetséges
- Preferáljuk a {} operátor használatát () és = helyett
- Csak akkor használjunk default konstruktort, ha az osztálynak létezik "természetes" alapeseti értéke

Hasznos tippek (Stroustruptól) II

- Tagokat és szülőosztályokat deklarációjuk sorrendjében inicializáljuk
- Ha egy osztályban van referencia (vagy pointer), igen valószínű hogy copy constructor-ra és copy assignment-re is szükség van
- Preferáljuk a tagváltozó-inicializáló listák használatát értékadás helyett (értékadásnál először inicializálás történik 0-ba, majd értékadás)
- Alapeseti értékek beállításához deklaráción belüli inicializálókat használjunk az oszály belsejében
- Ha egy osztály erőforrás-handle, valószínűleg szüksége