# C++ 11-től kezdődően bevezetett kiegészítések

Darvay Zsolt

Babeş–Bolyai Tudományegyetem, Kolozsvár

# Tartalomjegyzék

# Automatikus típusmeghatározás

- Az automatikus típusmeghatározást az **auto** kulcsszó használatával valósítjuk meg.
- Az **auto** kulcsszó az ANSI C előtti időszakban más jelentéssel bírt. Egy automatikus objektumot (helyi, de nem statikus változót) határozott meg. Ezt a jelentését az **auto** kulcsszónak törölték a C++ 11-ből.
- A **decltype** segítségével egy adott kifejezés típusát lekérhetjük és használhatjuk a kapott típust a kódunkban új változók deklarálására.

# Példa az auto kulcsszó használatára

```
#include <iostream>
#include <typeinfo>
#include <vector>

using namespace std;

template <typename T>
T negyzetreEmel(T x)
{
  return x * x;
}
```

# Példa az auto kulcsszó használatára

```
int main() {
  auto x = 10;
  cout << typeid(x).name() << endl;

  auto y = 0.5;
  cout << typeid(y).name() << endl;

  auto z = negyzetreEmel(y);
  cout << typeid(z).name() << endl;

  vector<int> v = { 1, 2, 3 };
  auto i = v.begin();
  cout << typeid(i).name() << endl;
  cout << *i << endl;
}
```

# Példa az auto kulcsszó használatára

Kimenet:

```
int
double
double
class std::_Vector_iterator<
  class std::_Vector_val<
    struct std::_Simple_types<int> > >
1
```

# Példa a *decltype* operátor használatára

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
   int x = 50;
   decltype(x) y;
   cout << typeid(y).name() << " y = ";
   y = 60;
   cout << y << endl;
}  // Kimenet: int y = 60
```

# 2. Példa (decltype)

```
#include <iostream>
#include <typeinfo>
using namespace std;

template<typename T1, typename T2>
auto maximum(T1 a, T2 b)
{
  return (a>b) ? a : b;
  //if (a > b)
  //  return a;
  //return b; // fordítási hiba,
}                 //ha T1 és T2 különböző
```

# 2. Példa (decltype)

```
template<typename T1, typename T2>
auto nagyobb(T1 a, T2 b) -> decltype(a + b)
{
  //return (a>b)? a : b;  // helyes
  if (a > b)
    return a;
  return b;
}
```

# 2. Példa (decltype)

```
template<typename T1, typename T2>
void kiir(T1 a, T2 b) {
  auto z1 = maximum(a, b);
  cout << typeid(z1).name() << " z1 = " << z1
     << endl;
  auto z2 = maximum(b, a);
  cout << typeid(z2).name() << " z2 = " << z2
     << endl;
  auto w1 = nagyobb(a, b);
  cout << typeid(w1).name() << " w1 = " << w1
     << endl;
  auto w2 = nagyobb(b, a);
  cout << typeid(w2).name() << " w2 = " << w2
     << endl;
}
```

# 2. Példa (decltype)

```
int main() {
   kiir(1, 2);

   kiir(3, 4.5);

   kiir(3, 2.5);
}
```

# 2. Példa (decltype)

Kimenet:

```
int z1 = 2
int z2 = 2
int w1 = 2
int w2 = 2
double z1 = 4.5
double z2 = 4.5
double w1 = 4.5
double w2 = 4.5
double z1 = 3
double z2 = 3
double w1 = 3
double w2 = 3
```

# Lambda kifejezések

*[elfogás] <típusparaméterek> (paraméterek) specifikátorok kivétel attribútum -> visszatérítés igénylés { test }*

Egy osztályt hoz létre, amelyben a () operátor túl van terhelve.

- elfogás: a lambda kifejezés környezetéből használhatunk bizonyos változókat (az elfogás történhet másolással vagy referencia szerint)
- típusparaméterek, igénylés: C++ 20-tól
- paraméterek: a () operátor paraméterei
- specifikátorok: pl. mutable (a másolással elfogott paraméterek is módosíthatóak)
- kivétel: kivétel specifikátor
- attribútum: attribútum specifikátor (pl. C++ kiterjesztések)
- test: a függvény teste

# Elfogás

- `[&]` Az összes helyi változó referencia szerinti elfogása (cím szerinti paraméterátadáshoz hasonló).
- `[&név]` Egy adott változó referencia szerinti elfogása.
- `[=]` Az összes helyi változó érték szerinti elfogása (érték szerinti paraméterátadáshoz hasonló).
- `[=név]` Egy adott változó érték szerinti elfogása.

# Példa lambda kifejezésre I

```
#include <iostream>

#include <algorithm>

using namespace std;

int main()
{
```

# Példa lambda kifejezésre II

```
int t[] = { 1, 2, 3, 4, 5, 6 };
int n = sizeof(t) / sizeof(int);
int paratlan = 0;
for_each(t, t + n,
   [&paratlan](int x) {
   if (x % 2)
     paratlan++;
});
cout << "Paratlanok szama: " <<
   paratlan << endl; // 3
}
```

# Általánosított for

- Ha egy tárolót vagy tömböt teljes egészében be szeretnénk járni, akkor a C++11-től kezdődően használhatjuk a **for** utasítás tartományra alapozott változatát.
- A C++20-tól a **for** utasításon belül egy külön inicializáló rész is elhelyezhető.

# Példa általánosított for utasításra

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  int t[] = { 1, 2, 3 };
  vector<int> v{ 10, 20, 30 };
  cout << "t =";
  for (int elem : t)
    cout << " " << elem;
  cout << "\nv =";
  for (int e : v)
    cout << " " << e;
  cout << endl;
}
```

# Példa általánosított for utasításra

Kimenet:

```
t = 1 2 3
v = 10 20 30
```

# Az inicializálás egységesítése

- A C++ nyelvben többféle inicializálási lehetőséggel találkozunk. Például:
  - egyenlőség jel: `int x = 5;`
  - kerek zárójel (konstruktor): `string s("abc");`
  - kapcsos zárójel (tömb): `int t[] = {1, 2};`
  - alapértelmezett érték: `int y = int();`
- A C++11-től kezdődően egységesen használható a kapcsos zárójel, de a többi lehetőség is megmarad.

# Példa inicializálásra

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

template<typename T>
void kiir(string s, T x)
{
  cout << s << x << endl;
}
```

# Példa inicializálásra

```
template <typename T>
void tkiir(string s, const T& t) {
   cout << s;
   for (auto p : t) {
      cout << p << "␣";
   }
   cout << endl;
}
```

# Példa inicializálásra

```
template <typename T>
void tkiir(string s, T* p, T* q) {
   cout << s;
   for (T* r = p; r != q; ++r) {
      cout << *r << " ";
   }
   cout << endl;
}
```

# Példa inicializálásra

```
class Oszt {
   double mx;
   double my;
   int mz[3];
   int mt = 5;
public:
   Oszt(double, double);
   void kiir();
};

Oszt::Oszt(double a, double b) :
   mx(a), my{ b }, mz{ -1,-2,-3 }
{
}
```

# Példa inicializálásra

```
void Oszt::kiir() {
   ::kiir("mx = ", mx);
   ::kiir("my = ", my);
   tkiir("mz = ", mz);
   ::kiir("mt = ", mt);
}

int main() {
   string s1("egyetem");
   kiir("s1 = ", s1);
   string s2 = "c++";
   kiir("s2 = ", s2);
   int x(3);
   kiir("x = ", x);
```

# Példa inicializálásra

```
int y = 5;
kiir("y = ", y);
int z = int(); //alapértelmezett
               //érték
kiir("z = ", z);
int t1[] = { 1,2,3 };
tkiir("t1 = ", t1);
int t2[]{ 10, 20, 30 };
tkiir("t2 = ", t2);
int t3[4]{ 5 };
tkiir("t3 = ", t3);
int *p = new int[3]{ 11, 22, 33 };
tkiir("p = ", p, p + 3);
```

# Példa inicializálásra

```
    Oszt ob1(1, 2);
    ob1.kiir();
    Oszt ob2{ 3, 4 };
    ob2.kiir();
    vector<int> v{ 1, 2, 3 };
    tkiir("v = ", v);
    cout << "v merete = " << v.size() << endl;
    vector<double> w{ 1.1, 2.2, 3.3 };
    tkiir("w = ", w);
    cout << "w merete = " << w.size() << endl;
}
```

# Példa inicializálásra

Kimenet:

```
s1 = egyetem
s2 = c++
x = 3
y = 5
z = 0
t1 = 1 2 3
t2 = 10 20 30
t3 = 5 0 0 0
p = 11 22 33
mx = 1
my = 2
mz = -1 -2 -3
mt = 5
mx = 3
my = 4
mz = -1 -2 -3
mt = 5
v = 1 2 3
v merete = 3
w = 1.1 2.2 3.3
w merete = 3
```

# Típusok jellemzői

- Egy sablonokon alapuló fordítási időben elérhető felületet biztosít az egyes típusok jellemző vonásainak lekérdezése, illetve módosítása érdekében.
- A type_traits fejállományra van szükség.
- Példák: `is_void`, `is_class`, `is_arithmetic`, `is_const`, `has_virtual_destructor`

# Példa: is_void

```
#include <iostream>
#include <type_traits>

using namespace std;

int main() {
   cout << boolalpha;
   cout << is_void<void>::value
        << endl;  // true
   cout << is_void<float>::value
        << endl; // false
}
```

# Virtuális destruktor

```
#include <iostream>
#include <type_traits>
using namespace std;

class Film {
protected:
   char *cim;
public:
   Film(char* cim);
   /*virtual*/ ~Film();
};
```

# Virtuális destruktor

```
Film::Film(char *cim)
{
  this->cim = new char[strlen(cim) + 1];
  strcpy(this->cim, cim);
}

Film::~Film()
{
  cout << "Felszabaditva: cim ( "
       << cim << " )\n";
  delete[]cim;
}
```

# Virtuális destruktor

```
class Krimi : public Film {
protected:
   char *nyomozo;
public:
   Krimi(char *cim, char *nyomozo);
   ~Krimi();
};
```

# Virtuális destruktor

```
Krimi::Krimi(char * cim, char * nyomozo) :
    Film(cim)
{
  this->nyomozo =
    new char[strlen(nyomozo) + 1];
  strcpy(this->nyomozo, nyomozo);
}

Krimi::~Krimi()
{
  cout << "Felszabaditva: nyomozo ( "
       << nyomozo << " )\n";
  delete[] nyomozo;
}
```

# Virtuális destruktor

```
void virtualis_destruktor()
{
  if (has_virtual_destructor<Krimi>::value)
    cout << "Krimi destruktora virtualis."
         << endl;
  else
    cout << "Krimi destruktora nem virtualis."
         << endl;
}
```

# Virtuális destruktor

```
int main()
{
  Film *a = new Film("Forrest Gump");
  delete a;
  Krimi *b =
    new Krimi("Castle", "Kate Beckett");
  delete b;
  Film *c =
    new Krimi("Dr. Csont", "Seeley Booth");
  delete c;
  virtualis_destruktor();
}
```

# Virtuális destruktor

Kimenet:

```
Felszabaditva: cim ( Forrest Gump )
Felszabaditva: nyomozo ( Kate Beckett )
Felszabaditva: cim ( Castle )
Felszabaditva: cim ( Dr. Csont )
Krimi destruktora nem virtualis.
```

# Virtuális destruktor

Kimenet (ha a destruktort virtuálisnak deklaráljuk):

```
Felszabaditva: cim ( Forrest Gump )
Felszabaditva: nyomozo ( Kate Beckett )
Felszabaditva: cim ( Castle )
Felszabaditva: nyomozo ( Seeley Booth )
Felszabaditva: cim ( Dr. Csont )
Krimi destruktora virtualis.
```

# Áthelyező konstruktor és értékadó operátor

- A C++11-ben bevezették az **rvalue referencia** fogalmát, melyet a `&&` segítségével adunk meg.
- Ily módon különbséget lehet tenni a balérték (lvalue) és jobbérték (rvalue) referencia között.
- Az lvalue egy olyan objektummal van megadva, amely rendelkezik egy névvel, míg az rvalue általában egy névvel nem rendelkező temporális objektumot jelöl.
- Az áthelyező (move) konstruktor és értékadó operátor az rvalue módosítását teszi lehetővé.

# Áthelyező konstruktor

```
#include <string>
#include <iostream>
#include <iomanip>
#include <utility>
using namespace std;
```

# Áthelyező konstruktor

```
class A
{
   string s;
public:
   A() : s("A.s") { }
   A(const A& o) : s(o.s) {
      cout << "Az athelyezes nem sikeres!\n";
   }
   A(A&& o) noexcept : s(move(o.s)) {
      cout << "Athelyezve: A.s\n";
   } // noexcept: nem vált ki kivételt
   string getS() { return s; }
};
```

# Áthelyező konstruktor

```
A f(A a) {
  return a;
}

class B : public A
{
  string s2{ "B.s2" };
  int n = 0;
  // alapértelmezett move konstruktor B::(B&&)
  // meghívja az A move konstruktorát
  // meghívja az s2 move konstruktorát
  // bitenként másolja le az n értékét
};
```

# Áthelyező konstruktor

```
class C : public B
{
public:
   ~C() { }
   // ha van destruktor, akkor nem jön létre
   // az alapértelmezett move konstruktor
   // C::(C&&)
};
```

# Áthelyező konstruktor

```
class D : public B
{
public:
   D() { }
   ~D() { }
   D(D&&) = default;
   // van destruktor, de mégis létrehozza az
   // alapértelmezett move konstruktort
};
```

# Áthelyező konstruktor

```
int main()
{
   cout << "Az \"A\" athelyezese:\n";
   A a1 = f(A());
   // temporális objektum áthelyezése
   cout << "Athelyezes elott: a1.s = "
        << quoted(a1.getS()) << "\n";
   // quoted - idézőjelbe teszi
   // a karakterláncot
   A a2 = move(a1); // az a1 áthelyezése
   cout << "Athelyezes utan: a1.s = "
        << quoted(a1.getS()) << "\n";
```

# Áthelyező konstruktor

```
cout << "A␣\"B\"␣athelyezese:\n";
B b1;
cout << "Athelyezes␣elott:␣b1.s␣=␣"
     << quoted(b1.getS()) << "\n";
B b2 = move(b1);
// az alapértelmezett áthelyező
// konstruktort hívja meg
cout << "Athelyezes␣utan:␣b1.s␣=␣"
     << quoted(b1.getS()) << "\n";
```

# Áthelyező konstruktor

```
    cout << "A␣\"C\"␣athelyezese:\n";
    C c1;
    C c2 = move(c1);
    // másoló konstruktort hív meg

    cout << "A␣\"D\"␣athelyezese:\n";
    D d1;
    D d2 = move(d1);
}
```

# Áthelyező konstruktor

Kimenet:

```
Az "A" athelyezese:
Athelyezve: A.s
Athelyezes elott: a1.s = "A.s"
Athelyezve: A.s
Athelyezes utan: a1.s = ""
A "B" athelyezese:
Athelyezes elott: b1.s = "A.s"
Athelyezve: A.s
Athelyezes utan: b1.s = ""
A "C" athelyezese:
Az athelyezes nem sikeres!
A "D" athelyezese:
Athelyezve: A.s
```

# Alapértelmezett és törölt függvények

- A C++03-ban, amennyiben mi nem adjuk meg, a fordító alapértelmezett konstruktort, másoló konstruktort, értékadó operátort és destruktort hoz létre, azonban nincs lehetőség ennek módosítására.
- A C++11-től lehetőségünk van arra, hogy a fenti függvények közül egyeseket alapértelmezetten létrehozzunk (default), másokat pedig töröljünk (delete), azaz ne hozzuk létre.

# Alapértelmezett és törölt függvények

Alapértelmezett konstruktor és destruktor:

```
class Oszt
{
public:
   Oszt() = default;
   virtual ~Oszt() = default;
};
```

# Nem másolható osztály

```
class NemMasolhato
{
public:
   // alapértelmezett konstruktor
   NemMasolhato() = default;
   // masoló konstruktor
   NemMasolhato(const NemMasolhato&) = delete;
   // értékadó operátor
   NemMasolhato&
      operator =(const NemMasolhato&) = delete;
};
```

# Fő függvény

```
int main()
{
  Oszt ob;
  NemMasolhato x;
  //NemMasolhato y{x};
  // hiba: a másoló konstruktor törölt
  NemMasolhato z;
  //z = x;
  // hiba: az értékadó operátor törölt
  return 0;
}
```

# Megbízott (delegating) konstruktor

- A C++11-től lehetőség van arra, hogy egy adott osztály meghívja ugyanannak az osztálynak egy másik konstruktorát.

# Osztálydeklaráció

```
#include <iostream>
using namespace std;

class Oszt {
   int x, y;
public:
   Oszt(int a, int b);
   Oszt();
   void kiir();
};
```

# Konstruktorok

```
Oszt::Oszt(int a, int b) : x{ a }, y{ b } {
  cout << "Parameteres konstruktor.\n";
}

Oszt::Oszt() : Oszt(0, 0) {
  // meghívja a paraméteres konstruktort
  cout << "Alapertelmezett konstruktor.\n";
}
```

# A kiir metódus és a fő függvény

```
void Oszt::kiir()
{
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
}

int main() {
   Oszt ob1(10, 20);
   ob1.kiir();
   Oszt ob2;
   ob2.kiir();
}
```

# Megbízott konstruktor

Kimenet:

```
Parameteres konstruktor.
x = 10
y = 20
Parameteres konstruktor.
Alapertelmezett konstruktor.
x = 0
y = 0
```

# Felülírás és végleges jelleg

- A C++03-ban megtörténhet, hogy a egy virtuális függvényt úgy írunk felül, hogy tévedésből megváltoztatjuk a fejlécét.
- A C++11-től megadható az **override** kulcsszó a származtatott osztály metódusának fejlécében. Ebben az esetben a fordító ellenőrzi, hogy az illető függvény valóban egy ősbeli metódus felülírása-e? Ha nem talál ugyanolyan fejléccel rendelkező függvényt, akkor hibát jelez.
- A C++11-től kezdődően arra is lehetőség van, hogy letiltsuk egy adott függvény felülírhatóságát, illetve azt, hogy egy adott osztályból származtatottat hozzunk létre. Ezt a **final** kulcsszóval kell megadni.

# Lehetséges hiba C++ 03-ban

```
#include <iostream>
using namespace std;
struct Alap
{
  virtual void f(double x) {
    cout << "alap␣f,␣x␣=␣" << x << endl;
  }
};

struct Szarm : Alap
{
  virtual void f(int x) {
    cout << "szarm␣f,␣x␣=␣" << x << endl;
  }
};
```

# Az Alap2 struktúra

```
struct Alap2
{
  virtual void f(double x) {
    cout << "alap2␣f,␣x␣=␣" << x << endl;
  }
};
```

# Az override azonosító

```
struct Szarm2 : Alap2
{
  // hibás, nem lehet felülírni
  //virtual void f(int x) override {
  //  cout << "szarm2 f, x = " << x << endl;
  //}
  virtual void f(double x) override {
    cout << "szarm2 f, x = " << x << endl;
  }
};
```

# Végleges osztály vagy struktúra

```
struct Alap3 final { };

// hiba, nem lehet származtatni
//struct Szarm3 : Alap3 { };
```

# Végleges metódus

```
struct Alap4
{
  virtual void f() final;
};

void Alap4::f() {
  cout << "alap4, f\n";
}

struct Szarm4 : Alap4
{
  // hiba, nem lehet felülírni
  //void f() {}
};
```

# Fő függvény és kimenet

```
int main() {
   Alap* p = new Szarm;
   p->f(10);
   Alap2* q = new Szarm2;
   q->f(10);
}
```

Kimenet:

```
alap f, x = 10
szarm2 f, x = 10
```