A C++11 újdonságai

Smidla József
Operációkutatási Laboratórium
2014. október 9.

Tartalom

- nyers string
- auto, decltype
- nullptr
- constexpr
- foreach
- újdonságok osztályokban
- típusos enum
- lambda kifejezések
- struktúra inicializálás
- inicializáló lista
- smart pointerek
- szálkezelés

A C++11 szabvány

- A szabvány 2011 szeptemberében jelent meg
- Ha a gcc-t használjuk, akkor érdemes legalább a 4.8.1-s verziót használni
- A gcc alapesetben még a 2003-as szabványt támogatja
- Adjuk meg a fordítónak a –std=c++11 kapcsolót!
- A nyelv azóta is fejlődik, a legújabb szabvány a 2014-es
- Jelen előadásnak nem célja, hogy teljes áttekintést nyújtson a C++11-ről

Nyers stringek

Hagyományos stringek:

```
cout << "Hello
World";
```

! HIBA

Nyers stringek:

```
cout << R"(Hello
          World \n \n \t!)";
```

Nyers stringek

Hagyományos stringek:

```
cout << "Hello
World";
```

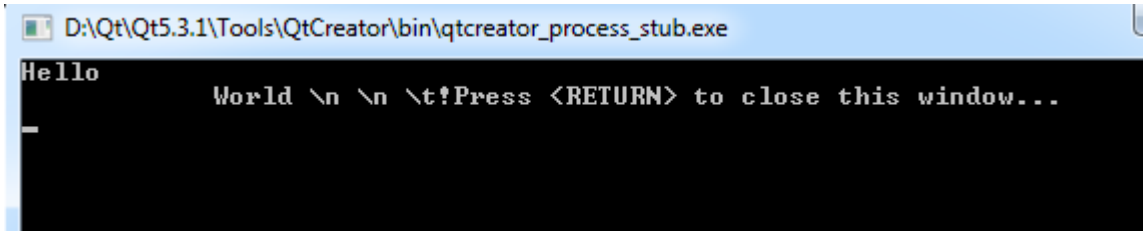
! HIBA

Nyers stringek:

```
cout << R"(Hello
          World \n \n \t!)";
```

Nyers stringek

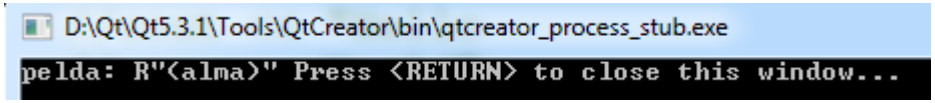
- A nyers stringek úgy tárolódnak, ahogy azokat beírtuk a forráskódba:



- Összetettebb határoló karaktersorozat:

```
cout << R"valami(pelda: R"(alma)" )valami";
```

- A ” és a zárójelek közé tetszőleges határoló karaktersorozatot írhatunk



auto, decltype

- Adott egy összetett tároló:

```
std::map<int, std::vector<string> > m;
```

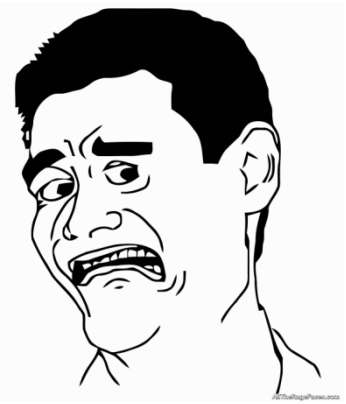
auto, decltype

- Adott egy összetett tároló:

```
std::map<int, std::vector<string> > m;
```

- Járjuk végig egy iterátorral:

```
std::map<int, std::vector<string>
>::iterator iter = m.begin();
```



auto, decltype

- Adott egy összetett tároló:

```
std::map<int, std::vector<string> > m;
```

- Járjuk végig egy iterátorral:

```
std::map<int, std::vector<string>
>::iterator iter = m.begin();
```

Használjuk az auto kulcsszót:

```
auto iter = m.begin();
```

Majd a fordító kitalálja az iter típusát az m.begin() visszatérési értékének típusából.

auto, decltype

- Változó, visszatérési érték típusának lekérdezése:

```
int x;
decltype(x) y = 5;
```

- Főleg templatek használatakor lehet nagyon hasznos.

nullptr

- A nullpointer hagyományosan: 0 vagy NULL

```
#ifndef __cplusplus
#define NULL ((void *)0)
#else /* C++ */
#define NULL 0
#endif /* C++ */
```

Azaz, C++-ban a NULL típusa int!

Melyik függvény hívódik meg?

```
void foo(int) {}
void foo(char *) {}
[...]
foo(NULL);
```

nullptr

- Megoldás: A C++11-ben bevezették a nullptr kulcsszót.
- Implicit kasztolható más pointerre és boolra.
- Saját típusa van:
  - **`decltype`**(**`nullptr`**), vagy
  - `std::nullptr_t`
- Írhatunk olyan függvényt, amely csak nullptr paramétert fogad el:

```
void foo(std::nullptr_t) {}
```

constexpr

- Számítsuk ki egy egész szám kettes alapú logaritmusát metaprogramozással:

constexpr

- Számítsuk ki egy egész szám kettes alapú logaritmusát metaprogramozással:

```
template <unsigned n>
struct Logarithm {
  enum { value = 1 + Logarithm<n / 2>::value };
};
template <>
struct Logarithm<1> {
  enum { value = 0 };
};
cout << Logarithm<512>::value << endl;
```

constexpr

- Számítsuk ki egy egész szám kettes alapú logaritmusát metaprogramozással:

```
template <unsigned n>
struct Logarithm {
  enum { value = 1 + Logarithm<n / 2>::value };
};
template <>
struct Logarithm<1> {
  enum { value = 0 };
};
cout << Logarithm<512>::value << endl;
```

- Használjunk konstans kifejezést:

```
constexpr unsigned int log2(unsigned int n) {
  return n == 1 ? 0 : 1 + log2(n / 2);
}
cout << log2(512) << endl;
```

foreach

- Írassuk ki egy vector minden elemét:

```
vector<int> v;
[…]
auto iter = v.begin();
auto iterEnd = v.end();
for (; iter != iterEnd; iter++) {
  cout << *iter << endl;
}
```

foreach

- Írassuk ki egy vector minden elemét:

```
vector<int> v;
[…]
auto iter = v.begin();
auto iterEnd = v.end();
for (; iter != iterEnd; iter++) {
  cout << *iter << endl;
}
```

- Ugyanez tömörebben:

```
for (auto i: v) {
  cout << i << endl;
}
```

foreach

- Növeljük egy vector minden elemét:

```
vector<int> v;
[…]
auto iter = v.begin();
auto iterEnd = v.end();
for (; iter != iterEnd; iter++) {
  cout << iter++ << endl;
}
```

foreach

- Növeljük egy vector minden elemét:

```
vector<int> v;
[…]
auto iter = v.begin();
auto iterEnd = v.end();
for (; iter != iterEnd; iter++) {
  cout << iter++ << endl;
}
```

- Ugyanez tömörebben:

```
for (auto & i: v) {
  cout << i++ << endl;
}
```

foreach

- A foreach saját osztályra is működik, ha:
- Az osztálynak van begin() és end() függvénye
- Ezen függvények valamilyen iterátort adnak vissza
- Az iterátor rendelkezik ++, != és * (indirekció) operátorokkal

Újdonságok osztályokban: értékadás deklarációban

- Adattagnak értékadás deklarációban:

```
class Person {
private:
   int age = 20;
   string name = "Dave";
};
```

- Amennyiben a konstruktorban nem adunk értéket valamely változónak, a deklarációban megadott értékeket veszi fel.

Újdonságok osztályokban: delete függvény

- Bizonyos függvények meghívása letiltható:

```
class Person {
public:
  void foo() = delete;
  void foo(int) {}
  template <class T> void foo(T) =
    delete;
};
```

- A fenti példában csak int típusú paraméterrel hívhatjuk meg a foo függvényt!
- További példa: Letiltható a másoló konstruktor, értékadó operátor is.

Újdonságok osztályokban: move konstruktor

- Balérték: Minden olyan kifejezés, amely megcímezhető, és értéket adhatunk neki

```
a = 1;  // a itt most balérték
```

- Jobbérték: Ideiglenes objektum

```
string getName() {
   return "Joe";
}
string name = getName();
```

- Itt a name balérték, viszont a visszatérési érték jobbérték

Újdonságok osztályokban: move konstruktor

- Balérték referencia: típus & név
- Jobbérték referencia: típus && név

```
string name1 = getName(); ✓
string & name2 = getName(); ✗
const string & name3 = getName(); ✓
string && name4 = getName(); ✓
const string && name5 = getName(); ✓
```

- A jobbérték referenciával hivatkozhatunk az ideiglenes objektumra
- Balérték referenciából jobbérték referencia:

```
Vector v;
Vector && vr = std::move(v);
```

Újdonságok osztályokban: move konstruktor

- Mi történik ekkor?

```
string getName() {
   return "Joe";
}
string name = getName();
```

Újdonságok osztályokban: move konstruktor

- Mi történik ekkor?

```
string getName() {
   return "Joe"; ①
}
string name = getName();
```

- ① Létrejön egy ideiglenes, string típusú objektum (konstruktor, memóriafoglalás, másolás)

Újdonságok osztályokban: move konstruktor

- Mi történik ekkor?

```
string getName() {
   return "Joe";
}
string name = getName(); ②
```

- ② Meghívódik a copy konstruktor, és létrejön a name objektum: újabb memóriafoglalás, másolás
- Végül az ideiglenes objektum felszabadul

Újdonságok osztályokban: move konstruktor

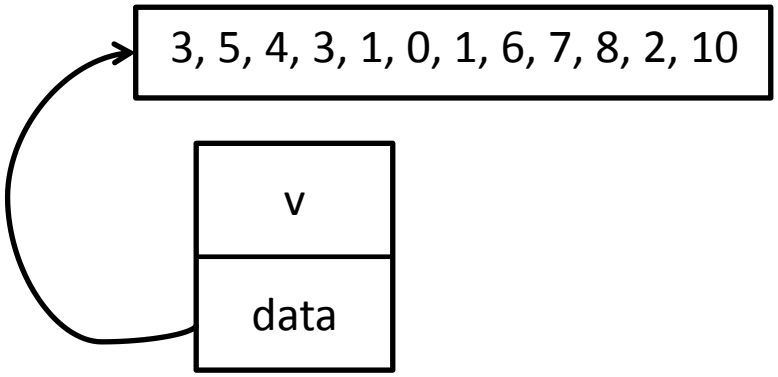
- Csak egyszer foglaljunk memóriát!
- Nem lemásoljuk a visszatérési értéket, hanem átmozgatjuk -> move konstruktor
- Paramétere: egy jobbérték referencia
- Tipikus használata: Lemásoljuk az adattagokat (a pointereket is), majd az eredeti objektum változóit kinullázzuk
- Az objektumot átmozgattuk, az eredeti objektumnál a destruktor nem csinál semmit

Újdonságok osztályokban: move konstruktor

```
class Vector {
public:
  Vector() {
    data = new double[10];
  }
  Vector(Vector && right) {
    data = right.data;
    right.data = nullptr;
  }
  ~Vector() {
    delete [] data;
    data = nullptr;
  }
private:
  double * data;
};
```

Újdonságok osztályokban: move konstruktor

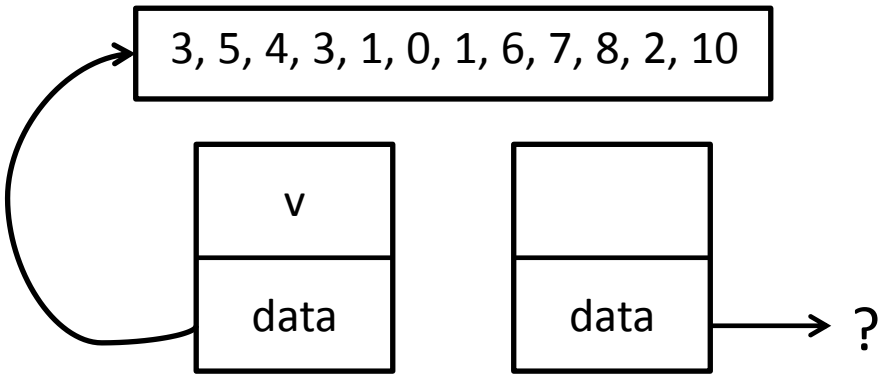
```
Vector getVector() {
  Vector v;
  return v;
}

int main() {
  Vector v0 = getVector();
}
```



Lefut a Vector konstruktora, az pedig lefoglalja a dinamikus tömböt.

Újdonságok osztályokban: move konstruktor

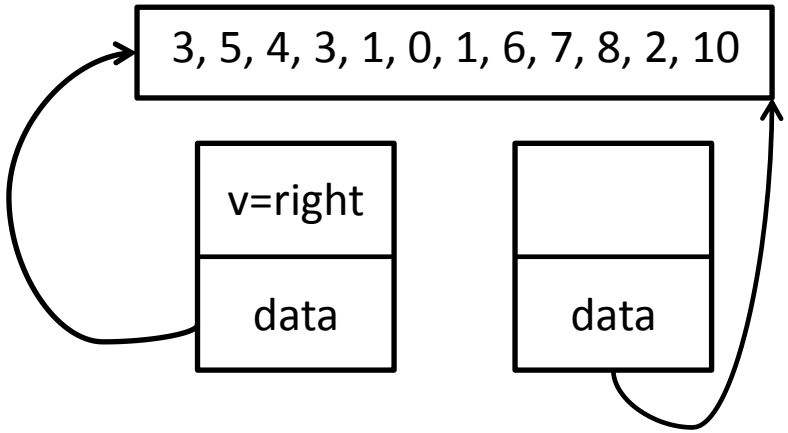
```
Vector getVector() {
  Vector v;
  return v;
}

int main() {
  Vector v0 = getVector();
}
```



Létrejön egy ideiglenes objektum, ez a move konstruktor által jön létre.

Újdonságok osztályokban: move konstruktor

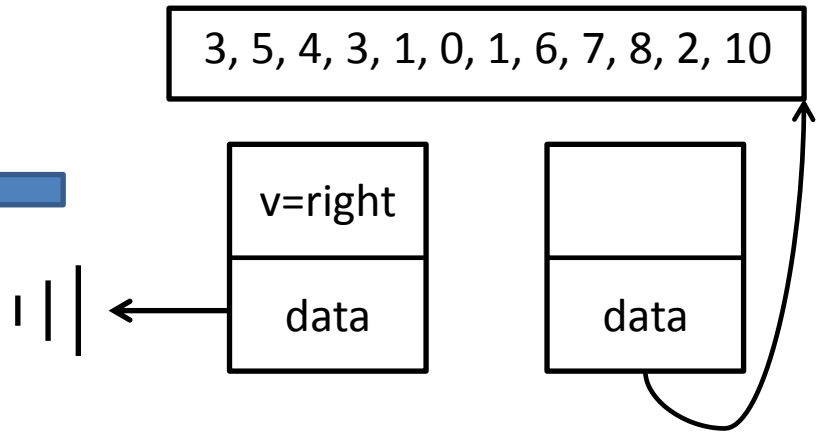
```
Vector(Vector && right) {
  data = right.data;
  right.data = nullptr;
}
```



Az újonnan létrejövő objektum lemásolja a pointert.

Újdonságok osztályokban: move konstruktor

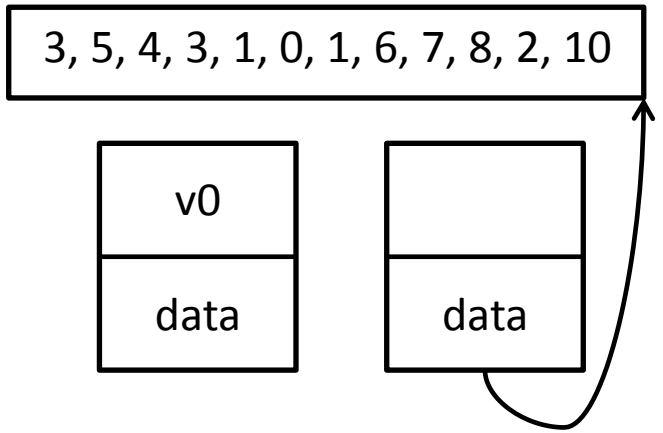
```
Vector(Vector && right) {
  data = right.data;
  right.data = nullptr;
}
```



A régi objektum pointerét nullázzuk, hiszen az úgyis felszabadul.

Újdonságok osztályokban: move konstruktor

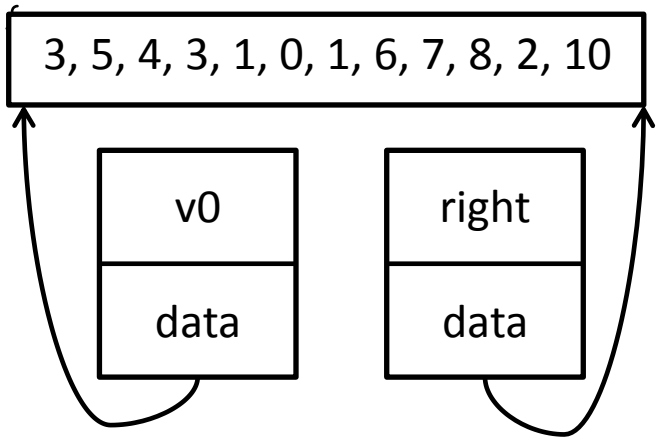
```
Vector getVector() {
  Vector v;
  return v;
}

int main() {
  Vector v0 = getVector();
}
```



Létrejön a v0 változó, mégpedig szintén egy move konstruktor hívással, ahol most az előbb létrejött vektor lesz a konstruktor paramétere

Újdonságok osztályokban: move konstruktor

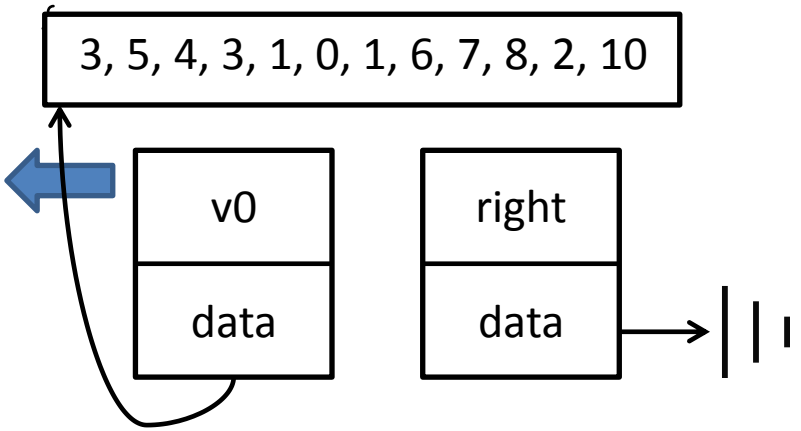
```
Vector(Vector && right) {
  data = right.data;
  right.data = nullptr;
}

int main() {
  Vector v0 = getVector();
}
```



Lemásoljuk a pointert.

Újdonságok osztályokban: move konstruktor

```
Vector(Vector && right) {
  data = right.data;
  right.data = nullptr;
}

int main() {
  Vector v0 = getVector();
}
```



Majd nullázzuk az ideiglenes objektum pointerét, hiszen úgyis felszabadul.

Újdonságok osztályokban: move = operátor

- Ha a visszatérési értéket olyan változóban tároljuk, amely már létezik, akkor az = operátor fut le
- Létezik move = operátor, amely szintén jobbérték referenciát kap paraméterként
- A move konstruktornál annyiban több, hogy az aktuális változó tartalmát felszabadítja

Újdonságok osztályokban: move = operátor

```
class Vector {
public:
   […]
   Vector & operator=(Vector && right) {
     if (&right == this) {
     return *this;
   }
   delete [] data;
   data = right.data;
   right.data = nullptr;
   […]
}
```

Eközben a gcc-ben…

- Gcc-t alapértelmezett paraméterekkel használva az alábbi kód esetében csak egy konstruktor és egy destruktor fut le:

```
Vector getVec() {
  Vector v;
  return v;
}
int main() {
  Vector vec = getVec();
```

- A fordító optimalizál (copy elision)
- Kikapcsolható a -fno-elide-constructors kapcsolóval (de nem érdemes ☺)

Típusos enum

- Probléma:

```
enum GYUMOLCSOK {ALMA, KORTE, NARANCS};
enum SZINEK {PIROS, KEK, NARANCS};
```

- A NARANCS két enumban is szerepel
- Megoldás:

```
enum class GYUMOLCSOK {ALMA, KORTE, NARANCS};
enum class SZINEK {PIROS, KEK, NARANCS};
enum class SZINEK szin = SZINEK::NARANCS;
```

- Egyértelműen kifejezhetjük, hogy az egyik enum típus nem kompatibilis a másikkal

Az enum mérete

- Az enum mérete alapértelmezetten az int méretével egyenlő
- A méret lecserélhető bármely egész típus méretére:

```
enum class GYUMOLCSOK : char {ALMA, KORTE};
```

Lambda kifejezések

- Írjunk függvényt, amely megszámlálja, hogy hány olyan elem van egy tömbben, amelyekre igaz egy bizonyos tulajdonság!
- A függvény legyen általános célú.
- Régi módszerek:
  - Függvény pointer átadása
  - Funktorok alkalmazása

Lambda kifejezések

Új módszer:

```
#include <functional>
int szamol(int * v, int meret,
                    std::function<bool (int)> talalt) {
  int i;
  int mennyi = 0;
  for (i = 0; i < meret; i++) {
    if (talalt(v[i])) {
      mennyi++;
    }
  }
  return mennyi;
}
int main() {
  int vec[10] = {3, 5, 1, 3, 5, 1, 7, 8, 4, 1};
  auto paros = [](int num)->bool{return num % 2 == 0;};
  int parosak = szamol(vec, 10, paros);
  […]
```

Lambda kifejezések

Másképp:

```
#include <functional>
int szamol(int * v, int meret,
                    std::function<bool (int)> talalt) {
  int i;
  int mennyi = 0;
  for (i = 0; i < meret; i++) {
    if (talalt(v[i])) {
      mennyi++;
    }
  }
  return mennyi;
}
int main() {
  int vec[10] = {3, 5, 1, 3, 5, 1, 7, 8, 4, 1};
  int parosak = szamol(vec, 10, [](int num)->bool{
    return num % 2 == 0;
  });
  […]
```

Lambda kifejezések

- Szintaktika:
- [<változók elfogása>] (<paraméterek>) -><visszatérési érték típusa>{<függvény törzs>}
- Változók elfogása: Segítségével a függvényből hivatkozhatunk olyan változókra, amelyek ott léteznek, ahol megírtuk a lambda kifejezést

Lambda kifejezések

- Példa változók elfogására:

```
int alma = 4;
int vec[10] = {3, 5, 1, 3, 5, 1, 7, 8, 4, 1};
int nagyok = szamol(vec, 10, [=](int num)->bool {
  return num > 3+alma;
});
```

- Az [=] jelentése: a függvény megkapja minden, a lambda kifejezés környezetében elérhető változó másolatát

Lambda kifejezések

- További lehetőségek változók elfogására:
- [] semmit nem kapunk el
- [&] a változókat referencia szerint kapjuk el
- [<változónév>] egy konkrét változók kapunk el érték szerint
- [=, &<változónév>] minden változót érték szerint kapunk el, kivéve egyet, amelyet referencia szerint
- [this] a befoglaló objektum címét fogjuk el

Struktúra inicializálás

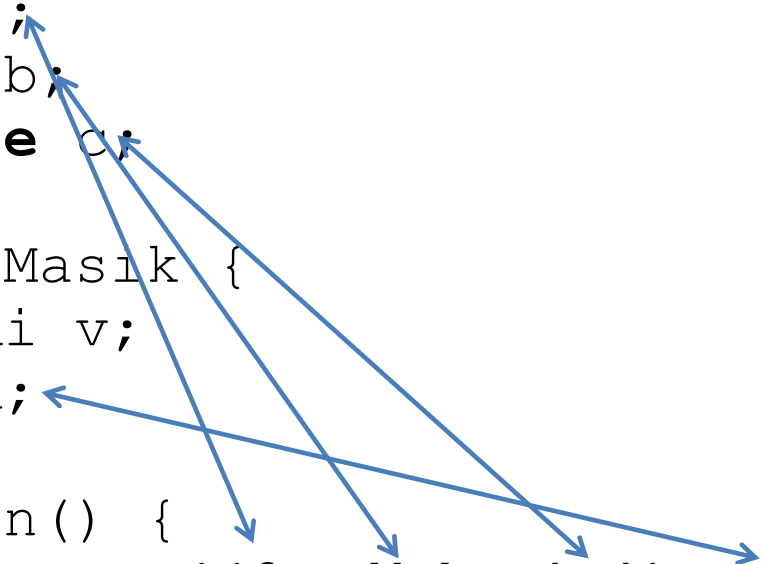
A struktúrákat mostantól tömörebb formában is inicializálhatjuk:

```
struct Valami {
  int a;
  char b;
  double c;
};
struct Masik {
  Valami v;
  int d;
};
int main() {
  Masik m = {{2, 'b', 4.4}, 4};
  […]
```

Struktúra inicializálás

A struktúrákat mostantól tömörebb formában is inicializálhatjuk:

```
struct Valami {
   int a;
   char b;
   double c;
};
struct Masik {
   Valami v;
   int d;
};
int main() {
   Masik m = {{2, 'b', 4.4}, 4};
   […]
```



Inicializáló lista

- Gyorsan töltsünk fel egy vectort néhány elemmel, régi módszer:

```
std::vector<int> v;
v.push_back(3);
v.push_back(2);
v.push_back(7);
```

- Új módszer:

```
std::vector<int> v = {3, 2, 7};
```

Inicializáló lista

- A saját osztályunkat is felkészíthetjük arra, hogy inicializáló listát kap
- A fordító felismeri az inicializáló listát
- A listát beágyazza egy std::initializer_list típusú objektumba
- Az objektumot pedig megkapja a függvényünk paraméterként

Inicializáló lista

```
Vector(const std::initializer_list<double> & list) {
  data = new double[ list.size() ];
  auto iter = list.begin();
  auto iterEnd = list.end();
  unsigned int index = 0;
  for (; iter != iterEnd; iter++, index++) {
    data[index] = *iter;
  }
}
```

Vagy:

```
Vector(const std::initializer_list<double> & list) {
  data = new double[ list.size() ];
  unsigned int index = 0;
  for (auto value: list) {
    data[index] = value; }
  }
}
```

Smart pointerek

- Header: memory
- Okos pointerek: korábban std::auto_ptr
- Eltárol egy pointert, és ha a változó megszűnik, automatikusan felszabadítja a pointer által mutatott objektumot
- Ha lemásoljuk az objektumot, akkor a másolat tárolja az eredeti pointert, az eredeti objektum nullpointert tartalmaz

Smart pointerek

```
std::auto_ptr<Vector> ptr1(new Vector);
std::auto_ptr<Vector> ptr2;

cout << ptr1.get() << endl;
cout << ptr2.get() << endl;

ptr2 = ptr1;

cout << ptr1.get() << endl;
cout << ptr2.get() << endl;
```

Kimenet:

```
konstruktor
0x661600
0
0
0x661600
destruktor
```

Smart pointerek

- std::unique_ptr: ugyanaz, mint az std::auto_ptr, csupán nem azt mondjuk hogy lemásoljuk a pointereket, hanem átmozgatjuk
- A C++11-ben érvénytelenítettnek számít az std::auto_ptr
- Az std::unique_ptr copy constructora és a = operátorok deleted függvények

Smart pointerek

```
std::unique_ptr<Vector> ptr1(new Vector);
std::unique_ptr<Vector> ptr2;

cout << ptr1.get() << endl;
cout << ptr2.get() << endl;

ptr2 = ptr1; ✘
ptr2 = std::move(ptr1); ✓

cout << ptr1.get() << endl;
cout << ptr2.get() << endl;
```

Smart pointerek

- std::shared_ptr: Több shared_ptr képes tárolni ugyanazt a pointert, ám referencia számlálást alkalmaz
- Ha megszűnik minden olyan shared_ptr, amely az adott objektumra hivatkozik, csak akkor szabadul fel a dinamikus objektum
- Probléma lehet abból ha az objektumok körkörösen hivatkoznak egymásra

Smart pointerek

```
std::shared_ptr<Vector> ptr1(new Vector);
std::shared_ptr<Vector> ptr2;

cout << ptr1.get() << endl;
cout << ptr2.get() << endl;

ptr2 = ptr1;

cout << ptr1.get() << endl;
cout << ptr2.get() << endl;
```

Kimenet:

```
konstruktor
0x6b1600
0
0x6b1600
0x6b1600
destruktor
```

Smart pointerek

x hivatkozik y-ra, y pedig x-re, hiába töröljük egyiket, vagy a másikat, életben tartják egymást

```
struct A {
  ~A() {
    cout << "Destruktor" << endl;
   }
   std::shared_ptr<A> ptr;
};
int main() {
  std::shared_ptr<A> x=std::make_shared<A>();
  std::shared_ptr<A> y=std::make_shared<A>();
  x->ptr = y;
  y->ptr = x;
```

Smart pointerek

- std::weak_ptr: Ha az objektumra már csak egy weak_ptr hivatkozik, akkor az az objektum felszabadul
- Megtörhető a körkörös hivatozás
- A weak_ptr lemásolhat egy shared_ptr-t
- A shared_ptr csak a weak_ptr lock() függvényén keresztül másolhat le egy weak_ptr-t
- Utóbbi esetben mindig ellenőrizni kell, hogy a másolat shared_ptr érvényes memóriaterületre mutat-e

Smart pointerek

A korábbi körkörös hivatkozás feloldása:

```
struct A {
  ~A() {
    cout << "Destruktor" << endl;
   }
   std::shared_ptr<A> ptr;
   std::weak_ptr<A> ptrWeak;
};
int main() {
  std::shared_ptr<A> x=std::make_shared<A>();
  std::shared_ptr<A> y=std::make_shared<A>();
  x->ptr = y;
  y->ptrWeak = x;
```

Szálkezelés

- Új szálakat hozhatunk létre az std::thread osztály segítségével
- Meg kell adnunk azt a függvényt, amelyet az új szál futtat

```
#include <thread>
#include <iostream>

void foo() {
    std::cout << "thread" << std::endl;
}

int main() {
    std::thread t1(foo);
    std::thread t2(foo);
    std::thread t3(foo);
    t1.join();  // megvárjuk, amíg lefut
    t2.join();  // megvárjuk, amíg lefut
    t3.join();  // megvárjuk, amíg lefut
 }
```

Szálkezelés

- Osztott változókat mutexekkel védhetünk
- A mutex is osztott
- A szálak megpróbálják lefoglalni a mutexet, de egyszerre csak egynek sikerül, a többi addig várakozik

```
int a = 0;
std::mutex m;
void foo() {
  m.lock();
  a = rand() % 10;
  m.unlock();
}
```

Szálkezelés

- Egy statikus változó csak egy példányban létezik
- Minden szál ugyanazt a statikus változót látja
- A thread_local kulcsszóval minden szál saját példányt kap az adott statikus változóból
- A thread_local változó a szál létrehozásakor jön létre, és a szál végén megsemmisül

Szálkezelés

thread_local használatával:

```
#include <iostream>
#include <thread>
thread_local int b;
class A {
public:
  static thread_local int a;
};
thread_local int A::a = 10;
std::mutex m;
void foo() {
  m.lock();
  cout << A::a++ << " " << &A::a << endl;
  cout << &b << endl;
  m.unlock();
}
```

Kimenet:

```
10 0x9431a4
0x9431dc
10 0x23c8124
0x23c8134
10 0x23c8144
0x23c8114
```

Szálkezelés

thread_local nélkül:

```
#include <iostream>
#include <thread>
int b;
class A {
public:
  static int a;
};
int A::a = 10;
std::mutex m;
void foo() {
  m.lock();
  cout << A::a++ << " " << &A::a << endl;
  cout << &b << endl;
  m.unlock();
}
```

Kimenet:

```
10 0x404004
0x40903c
11 0x404004
0x40903c
12 0x404004
0x40903c
```

Köszönöm a figyelmet!

A publikáció az Európai Unió, Magyarország és az Európai Szociális Alap társfinanszírozása által biztosított forrásból a TÁMOP-4.2.2.C-11/1/KONV-2012-0004 azonosítójú "Nemzeti kutatóközpont fejlett infokommunikációs technológiák kidolgozására és piaci bevezetésére" című projekt támogatásával jött létre.