

C++11

Beállítások c++11-hez

gcc, clang:

`-std=c++11`

mac:

`-stdlib=libc++`

Érdemes clang++-t használni, sokkal jobbak a hibaüzenetei:

Makefile-ban:

`CXX=clang++`

nullptr

Régebben NULL: `#define NULL 0`

```
int valami(int i) ;  
int valami(char* s) ;  
// ...  
valami(NULL) ;
```

Melyik hívódik meg?

nullptr

Egyértelmű

Gyakorlat NULLok kicserélése nullptr-re

auto

auto

- Változó típusaként használhatjuk a deklarációban
- **!!!** Volt egy régi jelentése is, de az megszűnt.
- Egyértelműen kikövetkeztethetőnek kell lennie a típusnak a kifejezésből:
 - `auto d = 3.14;`
 - `std::vector<int> v; auto iter = v.begin();`
 - `const auto& cRefD = d; // ++cRefD - compile error;`

auto - folytatás

- Nem használható:

- `class A`
 {
 // A(auto i) : m_a(i) {} // compiler error
 private:
 // auto m_a = 1; // compile error!
 int m_a;
 };
- `// std::vector<auto> v {1, 2, 3, 4, 5};`
- `// auto returnAuto() // compile error`
 // {
 // return 1;
 // }
- `// void f(auto) compiler error`

Range-based for loops

range base for

- Elemek egy sorozatát lehet bejárni vele
- A háttérben az osztályhoz tartozó iterátoron dolgozik
- Használható minden típusra, ha az:
 - rendelkezik **begin()** és **end()** metódusokkal
 - vagy léteznek és elérhetőek a típust paraméterként váró **begin** és **end** függvények
 - **begin** visszatérési értéke az első elemre mutató mutató
 - az **end**nek az utolsó elem utánira mutató mutató
 - a mutatóra elérhetőnek kell lennie a **++** (prefix), **!=** és ***** operátoroknak

range base for - folytatás

```
std::vector<int> v;  
for (int i : v) {  
    // Érték szerinti bejárás  
}
```

```
for (auto& i : v)  
{  
    i--; //Módosíthatjuk is  
}
```

Sokmindenen működik

inicializáló listán:

```
for (auto i : {2, 4, 6, 8, 10}) {}
```

stringen:

```
string kacska("kacska");  
for (auto ch : kacska) {  
}
```

Gyakorlat

A RelayServer.cpp-ben van pár ciklus, amit át lehet erre írni.

In-class member initializers

Pont mint javában

Mikor jó?

- Több konstruktor ugyanazokat állítaná be
- Csak pár érték miatt nem akarjuk megírni a konstruktort.
- Nincs meg az a sorrendiség nyűg vele, mint a konstruktor utáni init listákkal

Eddig is lehetett statikus mezőket inicializálni

```
class KisKutya {  
    int i = 7;  
    int j = 8;  
};
```

==

```
KisKutya() : i(7), j(8) {  
}
```

Gyakorlat

Ki kell cserélgetni, amit lehet

```
boost::thread* pingThread = nullptr;  
boost::thread* maintenanceThread = nullptr;  
bool running = false;
```


Smart pointerek

std::shared_ptr

```
shared_ptr<Peer> newPeer( new Peer(...) );
```

Referencia számlált

Akkor szabadítja fel a tárolt objektumot, ha nem mutat rá több referencia.

A RelayServer destruktorában pl. nem kell felszabadítani a peereket

shared_ptr::reset : Új objektumra állítja a pointert

std::shared_ptr

```
for (Peer* peer : peerList) {  
    delete peer;  
}
```

Na, ez mind nem kell

std::unique_ptr

Nem másolható, ha kikerül a scope-ból, akkor felszabadít.

Felhasználás pl függvényen belül, függvény végén mindenképpen felszabadul.

std::weak_ptr

- A shared_ptr-hez van köze.
- Egy shared pointeres objektumra mutathat, de nem használható közvetlenül.
- Nem számít bele a referenciaszámlálásba
- Akkor használható, ha shared_ptr-ré lesz alakítva a lock() metódussal.

std::weak_ptr példa

```
shared_ptr<User> sPtr(new User());  
weak_ptr<User> weakPtr;  
  
weakPtr = sPtr; // Rámutat, de nem  
növeli a referenciaszámlálót  
...
```

weak_ptr példa

```
if (auto sharedPtr = weakPtr.lock()) {  
    // Lehet használni a sharedPtr-t  
    // Biztos nem engedik el alólunk  
} else {  
    // Már el lett engedve az objektum  
}
```

Gyakorlat

Peer* -ek kitakarítása a RelayServer-ből

Lehet minden shared ptr

Uniform initialization

C++98

```
int ertekek[] = {10, 20, 30}; // Ok
//Vektorra alakítás:
const std::vector<int> vektor(
    ertekek,
    ertekek+sizeof(ertekek) /
        sizeof(ertekek[0])
);
```

C++98

```
int ertekek[] = {10, 20, 30}; // Ok
//Vektorra alakítás:
const std::vector<int> vektor(
    ertekek.begin(),
    ertekek.end()
);
```

Ez igen macerás

c++11

```
const std::vector<int> vektor {10, 20, 30};
```

c++11

```
const std::vector<int> vektor = {10, 20, 30};
```

Opcionális

Nem csak vektor

```
map<char, vector<MorseText::Signal> >
MorseText::signalMap = {
    {'a', {SHORT, LONG} },
    {'b', {LONG, SHORT, SHORT, SHORT} },
    {'c', {LONG, SHORT, LONG, SHORT} },
```

Nem csak vektor

```
map<char, vector<MorseText::Signal> >  
MorseText::signalMap = {
```

Ide jön a map inicializációs listája

```
};
```

Nem csak vektor

```
map<char, vector<MorseText::Signal> >  
MorseText::signalMap = {  
    {Entry inicializációs lista},  
    {Entry inicializációs lista},  
    {Entry inicializációs lista},  
};
```


Nem csak vektor

```
map<char, vector<MorseText::Signal> >  
MorseText::signalMap = {  
    {'a', {vector inicializációs lista}},  
    {'b', {vector inicializációs lista}},  
    {'c', {vector inicializációs lista}},  
};
```

Nem csak vektor

```
map<char, vector<MorseText::Signal> >  
MorseText::signalMap = {  
    {'a', {SHORT, LONG}},  
    {'b', {LONG, SHORT, SHORT, SHORT}},  
    {'c', {LONG, SHORT, LONG, SHORT}},  
};
```

Hogyan használhatunk ilyet saját osztályban?

```
#include <initializer_list>

class Valami {
public:
    Valami(std::initializer_list<int> ertekek) ;
};
```

Ezek támogatottak:

```
size()
begin()
end()
```

λ függvények

Mire jó?

Helyben lehet függvényeket írni.

Egy osztályt gyárt belőle, aminek meg van írva a () operátora

Szerkezet

```
[] (int a) { return a*2; }
```

Szerkezet

```
[] (int a) { return a*2; }
```

->

```
class Valami {  
    int operator(int a) {return a*2};  
};
```

Closure

A lambda függvény létrehozásakor fel lehet használni az elérhető változókat.

Ez történhet érték vagy referencia szerint.

```
int a = 0;
```

```
int b = 2;
```

```
auto f = [a,b]() { return a + b; }; // érték szerint
```

Az `f()`-et meghívva mindig 2-vel tér vissza.

Closure - referencia

```
int a = 0;  
int b = 2;  
auto f = [=,&b]() { return a + b; };  
           // a érték, b referencia szerint  
  
a=1;  
b=4;  
  
f() visszatérési értéke:
```

Closure - referencia

```
int a = 0;  
int b = 2;  
auto f = [=,&b]() { return a + b; };  
           // a érték, b referencia szerint  
  
a=1;  
b=4;  
  
f() visszatérési értéke: 4
```

Closure - osztály

Nem az osztály egyes adatait, hanem a this pointert kell a closureba menteni.

```
class AB
{
public:
    AB(int a, int b): m_a(a), m_b(b) {}
    std::function<int()> f()
    {
        return [this]() { return m_a + m_b; }; // capture by value
    }
private:
    int m_a;
    int m_b;
};
```

Closure - osztály

```
class AB
{
public:
    AB(int a, int b): m_a(a), m_b(b) {}
    std::function<int()> f()
    {
        return [this]() { return m_a + m_b; }; // capture by value
    }
private:
    int m_a;
    int m_b;
};
```

Az f() visszatérése egy int visszatérési típusú paraméter nélküli függvény.

Closure - osztály

```
class AB
{
public:
    AB(int a, int b): m_a(a), m_b(b) {}
    std::function<int()> f()
    {
        return [this]() { return m_a + m_b; }; // capture by value
    }
private:
    int m_a;
    int m_b;
};
```

A lambda függvény a két adattagot használja fel.

Closure - osztály

```
class AB
{
public:
    AB(int a, int b): m_a(a), m_b(b) {}
    std::function<int()> f()
    {
        return [this]() { return m_a + m_b; }; // capture by value
    }
private:
    int m_a;
    int m_b;
};
```

Ehhez érték szerint menti az objektumot.

Lamba visszatérési típus (trailing)

```
[]() { return 3.14;} // double
```

```
[]() -> int { return 3.14;} // int + compiler warning
```

Hagyományos függvénydeklarációnál is használható:

```
int f();
```

```
auto f() -> int;
```

Jobbérték referencia

Bal- és jobbérték

Balérték (állhat az értékadás bal oldalán):

- `int i;`
- `int& f();`
- `i = 1;`
- `f() = 2;`

Jobbérték (csak az értékadás jobb oldalán állhat, nem lehet rá hivatkozni):

- `int g();`
- `int j = 3;`
- `j = (j+1)`

Move

- `X x;`
- `X getX();`
- `x = getX();`

Hagyományos értékadás: `(X& X::operator=(X const& other))`

- elengedi az x-ben fogott erőforrásokat
- másolatot készít az új értékről
- elmenti x-ben az új erőforrásokat
- a `getX` visszatérési értéke felszabadítódik

Szerencsésebb lenne a másolás helyett inkább kicserélni, belemozgatni az új értéket, de közben ne okozzunk bajt.

`X& X::operator=(<varázslat> other)`

Jobbérték referencia

`(X& X::operator=(X&& other))`

Az `X&&` egy `X` típusú jobbérték referenciája.

A jobbértékeknek „nincs szükségük” a belső erőforrásaikra, mert úgyis ideiglenesek. Nyugodtan el lehet tőlük venni.

std::move

Ki lehet kényszeríteni, hogy jobbértékként kezelődjön egy balérték.

- Ennek van, hogy nincs értelme:

```
X x1;
```

```
X x2 = std::move(x1);
```

- Van amikor igen:

```
X& X::move(X&& other)
```

```
{
```

```
    return *this = std::move(other);
```

```
}
```

Implicit konverziók kizárása

- `class enum`
- `explicit cast operator`

Enum class

pl.:

```
enum class Allat {KUTYA, MACSKA, KACSA};
```

Szigorúan típusos, nem fog intként viselkedni.

Explicit cast operator

explicit operator std::string() const;

Nem történnek implicit konverziók.

Használata `static_cast<mire>(mit)`-tal

Normal c++ cast operatorok

`(ValamilyenOsztaly)masikOsztaly;`

Na, de igazából mi lesz belőle?

Normal c++ cast operatorok

`static_cast<mire>(mit)`: Fordítási idejű konverzió

`dynamic_cast<mire>(mit)`: Futási időben, pointerekkel lehet. Kell hozzá RTTI

`const_cast<>()`: Le lehet vele venni a constot

`reinterpret_cast<>()`: Bármit bármire

Szálak

boost::thread vs std::thread

Nagyjából ugyanaz, nagyon hasonló API

'Hívható' valamiket kell neki átadni.
(**Callable**)

Hívható: értelmes a () rajta

Szinkronizálás szálak között

`std::mutex`

Egyszerre egy szál 'tulajdona' lehet

`lock()`: Megszerzi tulajdont, blockol, amíg másnál van a tulajdon

`try_lock()`: Nem blokkol, false-szal tér vissza, ha nem sikerült

`unlock()`: Elengedi

lock_guard

Kényelmetlen kézzel hívogatni a lock/unlockot és előfordulhat, hogy nem hívjuk meg.

```
mtx.lock;  
...  
if (specialis eset)  
return; // Hopp, elfelejtettük  
...  
mtx.unlock();
```

lock_guard

lock_guard: RAII-s implementáció. Kap egy mutexet

A konstruktorban lockolja

A destruktorban unlockolja.

Ha a stacken hozzuk létre, akkor a scope-ból való kilépésnél automatikusan elengedi

lock_guard példa

```
{  
lock_guard<mutex> scopelock (mtx) ;  
  
...  
}
```

Ha return, vagy exception miatt kilép a scopeból, akkor is felszabadul.

std::condition_variable

Signaling esemény, felébreszthet egy vagy több szálát, amit várt erre az objektumra

`notify_one`, `notify_all`: szálakat lehet felébreszteni

Várakozás:

`wait`

`wait_for(timeout)`

`wait_until(időpont)`

std::condition_variable

Mindenképpen kell hozzá egy lock amivel dolgozhat

A lockot meg kell fogni, mielőtt waitelünk vagy notify-olunk.

A wait és a notify automatikusan elengedi a lockot

Unicode support

Eddig

`char*`: 1 byte-os karakterek - "hello"

`std::string`

`w_char*`: 2 bytes karakterek - L"hello"

`std::wstring`

Újdonságok:

`char16_t*` 2 byte-os karakterek - utf-16

`u"valami"` - `std::u16string`

`char32_t*` `char16_t*` 2 byte-os karakterek - utf-32

`U"valami"` - `std::u32string`

Literálként le lehet írni UTF-8 stringet is: `u8"`
`valami"`

HOSSZ!

Új konténerek

`std::array<class T, size_t N>`

Fix méretű vektor

normál c tömböt csomagol be, de vannak hozzá iterátorok

std::forward_list

Egyszeresen láncolt lista

(Az std::list kétszeresen láncolt lista)

Ha nagyon fontos a hely, akkor ezzel lehet spórolni.

`std::unordered_set`

= java HashSet

- Konstans idejű a keresés, hozzáadás, törlés.
Nem kell operator<
- De kell hash függvény
- (Az `std::map` fa adatszerkezettel van megvalósítva: sorrendet tart a bejárás)

`std::unordered_multiset`:

Több egyenlő érték is lehet benne.

`std::unordered_map`

= java HashMap

Konstans idejű a keresés, hozzáadás, törlés

Forrás

C++ Rvalue References Explained - http://thbecker.net/articles/rvalue_references/section_01.html

C++11 Features (Modern C++) - <http://msdn.microsoft.com/en-us/library/vstudio/hh567368.aspx>

C++0x/C++11 Support in GCC
<http://gcc.gnu.org/projects/cxx0x.html>

C++98 and C++11 Support in Clang
http://clang.llvm.org/cxx_status.html

CPP Reference
<http://cppreference.com>