Multiplatform szoftverfejlesztés

Interop

Interop

- Fogalmak
  - Natív kód, natív réteg
    - (Natív) C++-ban megírt kód
  - Platform kód
    - C#, Java, Swift, … ami az adott platformon használt
  - Adapter
    - A két réteget összekötő kód, wrapper
- A platformszintű technológia és a natív C++ közötti együttműködés nem triviális
  - A két réteg között interop technológiára van szükség

Cél

- Miért akarunk C++-t hívni platform kódból?
  - Létező kódot kell bekapcsolni a rendszerbe
    - Algoritmusok
    - Integráció más rendszerrel
  - Adott funkcionalitás csak így érhető el
    - Például Unity natív plugin, .NET C#-ból elérhetetlen része
  - Multiplatform közös kód elérése
    - Rossz példa: Xamarin .NET Standard Library
      - Mert mindkét oldal C#
  - Optimalizáció
    - Legkevésbé gyakori

Interop, miért van rá szükség?

- Hívási konvenciók
  - Verem kezelése
- Típusrendszer
  - Ez jelentősen eltérő lehet, ami nagyon megnehezíti az áthívást
- Kivételkezelés
  - A hasonlóság általában kicsi a két réteg között
- Adatok bináris reprezentációja
  - Tipikus probléma a string
  - Igazítás (layout) is kérdéses

Interop, miért van rá szükség?

- Memóriamodell különbségei
  - C++-ban kézzel kezeljük a memóriát, memóriacímeket
  - Java-ban, .NET-ben szemétgyűjtő kezeli a heap-et, átmozgathatja az objektumokat, nem használhatunk natív pointereket (vagy csak nehezen)
- OO koncepciókat (objektum, adattagok, virtuális függvények) máshogy reprezentáljuk a memóriában

Interop megoldások

- A problémák elkerülése végett klasszikusan alacsony szintű
  - Nem kell foglalkozni a legtöbb eltéréssel
  - string és társai így is gond, azt kezelni kell
  - Csak primitív adatokat tudunk átadni
  - Kivételkezelés nincs – meg kell írni
- C-stílusú globális függvények meghívására van lehetőség
- Például: .NET P/Invoke

.NET – C interop, P/Invoke példa

- C++:

```
extern "C" {
    __declspec(dllexport) void nativeFunc(int i) {
        printf("Number passed is %d.\r\n", i);
    }
}
```

- .NET, C#:

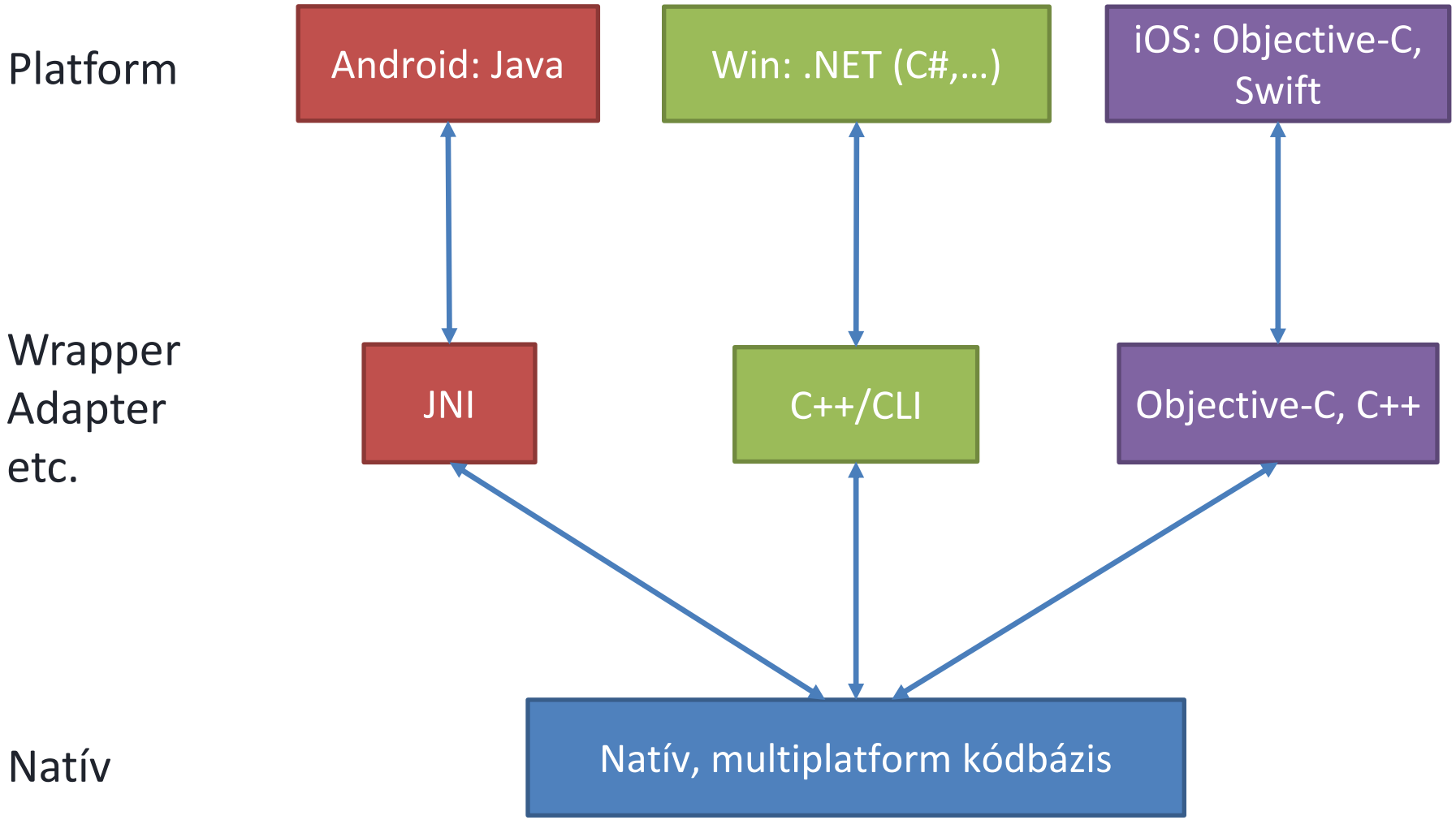
```
[DllImport("NativeLib.dll")]
public static extern void nativeFunc(int i);

static void Main(string[] args) {
    nativeFunc(5);
}
```

Interop cél

- Bár működik, ennél többet szeretnénk
  - Minél kényelmesebben elérhető legyen a natív kódbázis a platform oldaláról
    - Bónusz, ha a natív oldal el tudja érni a platform kódot
  - Minél kisebb legyen az áthívás overheadje
    - Marshaling: adatok másolása/konvertálása
  - Minél kisebb munka legyen implementálni a köztes réteget
- A következő technológiák többet tudnak
  - JNI, Objective-C++, C++/CLI

Interop, technológia



Java Native Interfaces

JNI – Java Native Interface

- Infrastruktúra, ami ahhoz kell, hogy Javából elérjünk natív technológiákat (pl. C++), és fordítva
  - Többet tud egy sima C függvény meghívásánál
- Szükséges, ha
  - Platformszintű API-t nem lehet elérni Javából, csak mondjuk C++-ból
  - Már létező natív könyvtárat szeretnénk egy Java alkalmazásban használni
- https://developer.android.com/training/articles/perf-jni.html

JNI alapok

- Java kódban deklarálhatunk **natív** metódusokat
  - Ezek implementációja a natív rétegben van
- A natív metódus meghívására a JNI framework továbbítja a hívást a natív komponenshez
- Így lehet áthívni Javából egy C vagy C++-os osztálykönyvtárba
- Ezzel közvetlenül globális függvényeket hívhatunk meg

JNI Hello World!

Natív függvény

```
JNIEXPORT void JNICALL
Java_mypackage_HelloWorld_print
( JNIEnv *env, jobject obj )
{
    printf("Hello world\n");
}
```

Java osztály

```
package mypackage;

class HelloWorld {
    public native void print(); // Natív fv
    static // Statikus inicializáció.
    {
        // Natív library betöltése.
        // CLibHelloWorld.dll
        System.loadLibrary("CLibHelloWorld");
    }
    public static void main(String[] args) {
            HelloWorld hw = new HelloWorld();
            // Natív függvény meghívása.
            hw.print();
        }
    }
}
```

Natív függvény argumentumai

```
JNIEXPORT void JNICALL Java_mypackage_HelloWorld_print
(JNIEnv* env, jobject obj)
{
   printf("Hello world\n");
}
```

- Első paraméterben megkapjuk a JNI környezetre mutató pointert
- Második paraméterben azt a Java objektumot, amiből a hívás érkezett
- Ha definiáltunk volna még paramétert a metódusnak, az ezek után szerepelne

JVM elérése C++-ból

- JNIEnv* pointeren keresztül elérhetőek a Java futtatókörnyezet (JVM) szolgáltatásai
  - JVM típusok felderítése
  - Típusok mezőinek, metódusainak lekérése
  - Java objektumok létrehozása
  - Mezők beállítása
  - Metódusok meghívása
  - Egyéb: stringek létrehozása, tömbök kezelése, kivételkezelés, szálkezelés

Metaprogramozás

- A JNI használata egyfajta string alapú metaprogramozás
  - Mint .NET Reflection
- Objektumok fogják reprezentálni a Javás típusokat, a típusok mezőit, metódusait
- Ezeket a nevük és típusuk stringként való megadásával tudjuk lekérdezni

Java típusok elérése

- Egy osztály elérhető a neve alapján
- Az osztállyal kapcsolatos többi funkció az így visszakapott objektumon keresztül érhető el

```
package hu.jnitest;
public class Vector {
    public double X;
    public double Y;
    public double Length() { ... }
    public Vector Multiply(double d) { ... }
    public Vector Add(Vector v) { ... }
}
```

Java osztály lekérése

- Az osztály reprezentáló objektum lekérhető a környezettől a nevével azonosítva

```
void Foo(JNIEnv env*) {
    jclass vectorClass = env->FindClass("hu/jnitest/Vector");
}
```

- Működik saját és beépített típusokra is

Mezők, metódusok lekérése

```
jclass vectorClass = env->FindClass("hu/jnitest/Vector");
jfieldID xId = env->GetFieldID(vectorClass, "X", "D");
jmethodID ctorId = env->GetMethodID(vectorClass, "<init>", "()V");
jmethodID lengthId = env->GetMethodID(vectorClass, "Length",
    "()D");
jmethodID multiplyId = env->GetMethodID(vectorClass, "Multiply",
    "(D)Lhu/jnitest/Vector;");
```

- Mező lekérése név és típus megadásával
- Metódus lekérése név és szignatúra megadásával
- Konstruktor lekérése speciális <init> névvel

Metódusszignatúrák

- A név nem elég egy metódus azonosítására
  - Szükség van a szignatúrára is (polimorfizmus miatt)
- Formátum (…)…
  - (argumentum-típusok)visszatérési érték típusa
- Primitív típusok leírója egy karakter
  - Pl.: boolean – „Z”, char – „C”, void – „V”
- Osztályok leírója L karakterrel kezdődik, utána a teljes package és az osztály neve, a végén egy pontosvesszővel
  - “Ljava/lang/String;”, “Lhu/jnitest/Vector;”
- Tömbök leírója [karakterrel kezdődik
  - Pl. egy boolean tömb: „[Z”

Metódusszignatúrák – példa

```
Vector[] f (String s, int[] arr);
```

(Ljava/lang/String;[I)[Lhu/jnitest/Vector;

- Sok a hibalehetőség, könnyű elrontani
- Nincs fordításidejű ellenőrzés

Java objektum létrehozása, beállítása

- A korábban lekért jclass, jfieldid és jmethodid objektumokkal hozható létre új példány

```
jobject CreateJavaVector(double x, double y, JNIEnv* env)
{
    jobject javaObj = env->NewObject(vectorClass, ctorId);
    env->SetDoubleField(javaObj, xId, x);
    env->SetDoubleField(javaObj, yId, y);
    return javaObj;
}
```

- Mező értéke lekérdezhető a GetXField függvényekkel

Metódus meghívása

- Metódus meghívható a CallXMethod függvényekkel, ahol X a visszatérési érték típusára utal. Át kell adni
  - az objektumot, amin a metódust hívjuk
  - a metódus azonosítóját (jmethodid)
  - az argumentumokat

```
double AddAndMultiply(jobject vec1, jobject vec2, double x, JNIEnv* env) {
    jobject sum = env->CallObjectMethod(vec1, addId, vec2);
    jobject multiplied = env->CallObjectMethod(sum, multiplyId, x);
    jdouble result = env->CallDoubleMethod(multiplied, lengthId);
    return result;
}
```

Stringek kezelése

- A javás string egy objektum a managed heapen
  - Nem csak egy sima karakterlánc
- Nem kompatibilis a natív C++ stringekkel
  - Konvertálni kell a két réteg között
  - UTF-8 és UTF-16 enkódolást támogat a JVM
    - C++ is támogatja ezeket, így elképzelhető, hogy maga a kódolás megegyezik
- Új string objektum létrehozása

```
jstring str = env->NewStringUTF("test string");
```

Stringek kezelése

- String hosszának lekérdezése

```
jsize len = env->GetStringLength(str);
```

- String nyers tartalmának lekérdezése (az így megkapott tömb használható megszokott módon natív C++-ban)

```
jchar* charsUTF16 = env->GetStringChars(str, nullptr);
char* charsUTF8 = env->GetStringUTFChars(str, nullptr);
```

- Használat után a karaktereket fel kell szabadítani a ReleaseStringChars vagy a ReleaseStringUTFChars függvényekkel

Tömbök kezelése

- Tömböket saját típusok reprezentálják: jintArray, jdoubleArray, jobjectArray, stb.
- Új tömb létrehozása

```
jintArray arr = env->NewIntArray( 5 );
```

- Primitív-tömb elemeinek beállítása
  - Egyszerre több elemet másol át

```
env->SetIntArrayRegion( arr, startIndex, length, nativeArray );
```

Tömbök kezelése

- Elemek lekérdezése
  - GetIntArrayElements: összes elem
  - GetIntArrayRegion: egy része
- Objektum-tömbök elemeinek beállítása egyenként történik
  - GC-vel együtt kell működni

```
jobject string = env->NewString(text.data(), text.length());
env->SetObjectArrayElement(array, index, string);
```

Lokális referenciák

- GC-vel együtt kell működni
- Amikor kapunk/létrehozunk egy referenciát, akkor a GC nem mozgathatja (pin/lock)
  - Amíg fut a kódunk
- Csak az adott natív hívás kontextusában használható
  - Visszatéréskor automatikusan felszabadul
  - Memory leak biztos
- Kézzel is fel lehet szabadítani hamarabb

```
env->DeleteLocalRef(javaObject);
```

Globális referenciák

- Ha később is szükségünk van egy referenciára
  - Eltároljuk
  - Háttérszálat indítunk, amiben használjuk
- Akkor globális referenciát kell rá létrehoznunk

```
jclass vectorClass = env->FindClass("hu/jnitest/Vector");
vectorClass = env->NewGlobalRef(vectorClass );
```

- Ezt kézzel fel kell szabadítani, magától nem szabadul fel

```
env->DeleteGlobalRef(vectorClass );
```

Szálkezelés, JNIEnv

- Minden natív függvény megkapja a JNIEnv* pointert
- Ha nem egy natív fv. kontextusában vagyunk, elkérhetjük a JVM-től. Erre egyszer el kell tárolnunk egy referenciát, ami sosem változik:

```
JavaVM* jvm;
int ret = env->GetJavaVM(&jvm);
```

- A JavaVM-től bármikor elkérhetjük a környezetet

```
JNIEnv * env;
int ret = jvm->GetEnv((void **)&env, JNI_VERSION_1_6);
```

Szálkezelés, JNIEnv

- Natív oldalról indított szálból
  - A szálat be kell regisztrálnunk

```
ret = jvm->AttachCurrentThread( &env, nullptr);
```

  - Ez nem lassít, csak létrehoz a JVM-ben egy gyűjteményt, ahova tárolni lehet a referenciákat

Referencia natív objektumra

- A natív objektumok nem a felügyelt heap-en vannak, nem tud róluk a JVM
  - Java referenciával nem tudunk natív objektumot kezelni
- Viszont egy natív objektum címe létrehozás után nem változik a memóriában
  - Címét egy longként a Java oldalnak átadva eltárolhatjuk
  - Visszaküldhetjük a natív rétegnek, ahol mutatóvá kell kasztolni

Referencia natív objektumra

- Natív objektum címének felküldése
  - 32 biten

```
MyNativeObj* obj = new MyNativeObj();
long address = reinterpret_cast<long>(obj);
env->CallVoidMethod(javaClass, storeAddressMethod, address);
```

  - 64 biten long long kell
- Cím visszaküldése egy natív metódusnak

```
MyNativeObj* obj =
    reinterpret_cast<MyNativeObj*>(nativeEnginePointer);
obj->Foo();
```

C++/CLI

C++/CLI

- Nyelvi kiegészítések a C++ nyelvhez
  - Elérhetőek/létrehozhatóak .NET-es típusok
- CLI: Common Language Infrastructure
  - Nyílt szabvány, aminek egyik megvalósítása a .NET Framework
- Ezekkel a nyelvi kiegészítésekkel használhatók a .NET platform konstrukciói és szolgáltatásai, így Windows-ra írt alkalmazásokkal lehet C++-ból együttműködni
  - Xamarin nem támogatja

C++/CLI céljai

- A .NET és a natív réteg közötti különbségek hasonlóak, mint a Java esetén
  - Más típusrendszer
  - Objektumok máshogy vannak reprezentálva a memóriában
  - .NET oldalon felügyelt heap szemétgyűjtéssel, C++ oldalon natív heap manuális memóriakezeléssel
- Ami Javában nincs: .NET oldalon két alapvetően eltérő user-defined típus van: értéktípus és referenciatípus, a kettőt máshogy kell tárolni, átadni

Windows interop, példa

Natív (C++/CLI)

```
namespace NativeLib
{
    public ref class MyClass sealed
    {
    public:
        Platform::String^ CreateString()
        {
            return L"alma";
        }
    };
}
```

Nyelvi kiegészítések
C++/CLI

Platform (C#)

```
var c =
    new NativeLib.MyClass();

string s = c.CreateString();

Console.WriteLine(s);
```

C++/CLI assembly

- Speciális osztálykönyvtár, amiben keveredhet a C++/CLI kód és a natív C++ kód
- C++/CLI-ben a .NET osztálykönyvtárának (BCL), és más .NET-es szerelvényeknek bármelyik osztálya elérhető és használható
- Lehet include-olni natív headeröket
- Lehet hozzá linkelni natív libraryket
- .NET-ből elérhető
- A benne definiált C++/CLI típusok .NET-ből ugyanúgy használhatók, mintha bármilyen más .NET-es nyelvben lettek volna implementálva (pl. C#-ban)

Nyelvi kiegészítések: tracking handle

- Tracking handle: speciális referencia, ami egy managed heapen lévő objektumra mutat

```
String^ str = "Ez egy .NET-es string";
auto di = gcnew DirectoryInfo("C:/Temp");
auto strArr = gcnew array<String^>(10);
```

- Tagok elérése: pointerhez hasonlóan -> operátorral

```
Console::WriteLine(di->FullName);
```

Nyelvi kiegészítések: tracking handle

- gcnew: Speciális operátor a new helyett, ami a natív heap helyett a felügyelt heapen hoz létre egy .NET-es objektumot
- Az így létrehozott objektumokat nem kell manuálisan felszabadítani
  - Nincs gcdelete, delete nem működik rajta
  - Hasonló a használatuk egy okos pointerhez, de nem referenciaszámlálással működnek, hanem a .NET megszokott szemétgyűjtésével
- Ez a nyelv az egyik ritka esete az opcionális GC-nek

Nyelvi kiegészítések: .NET típusok

- Natív osztályok létrehozása
  - class
  - struct
- .NET-es típusok létrehozása
  - public ref class: .NET-es referencia típus (class)
  - public value struct: .NET-es érték típus (struct)
  - public interface class: .NET-es interfész
  - public enum class: .NET-es enum
    - public nélkül C++11 nyelvi elem

ref class példa

```
public ref class ManagedPerson {
    int age;
    System::String^ name;
public:
    ManagedPerson(int age, System::String^ name) : age(age), name(name) { }
    property int Age {
        int get() { return age; }
        void set(int value) {
            if (value < 0)
                throw gcnew System::ArgumentException("Age must be 0+");
        }
    }
    property System::String^ Name {
        System::String^ get() { return name; }
        void set(System::String^ value) { name = value; }
    }
};
```

ref class példa

- A public kulcsszó azt jelenti, hogy elérhető legyen ez a típus más szerelvényekben is
- A .NET-es nyelvek (pl. C#) funkciói elérhetők C++/CLI-ben és fordítva
- Ez magas szintű, mindkét oldalon egyszerűen kezelhető interopot tesz lehetővé

Referenciatípusok használata

- Létrehozhatjuk a felügyelt heapen is, ilyenkor a szemétgyűjtő kezeli és szabadítja fel

```
auto personOnHeap = gcnew ManagedPerson(25, "Peter");
```

- Létrehozhatjuk a stacken is, ilyenkor automatikusan felszabadul, mint a natív C++ objektumok

```
ManagedPerson personOnStack(32, "John");
```

  - C#-ban erre nincs lehetőség, hogy referenciatípusokat a stacken hozzunk létre

Value struct példa

```
public value struct Color
{
    unsigned char A;
    unsigned char R;
    unsigned char G;
    unsigned char B;
};
```

- .NET-es érték típus lesz – nem terheli a GC-t
- Tömbben nagy mennyiségben hatékonyan létrehozható
- Tipikusan a stacken hozzuk őket létre és érték szerint adjuk át
  - A felügyelt heapen is létrehozhatjuk, de akkor nem közvetlenül az érték fog tárolódni, hanem egy csomagoló objektum (boxing)

Natív és felügyelt típusok együtt

- Elég szabadon keverhető, néhány megkötéssel

```
class NativeClass
{
public:
    NativePerson NativeMethod(NativePerson p); // OK
    ManagedPerson^ ManagedWithHandle(ManagedPerson^ mp); // OK
    ManagedPerson ManagedByValue(ManagedPerson mp); // OK
    NativePerson nativePerson; // OK
    ManagedPerson^ managedPerson; // ERROR: natív típusban nem
lehet ref
    ManagedPerson managedPersonByValue; // ERROR: natív típusban
nem lehet ref class érték szerint
    gcroot<ManagedPerson^> ManagedPerson; // OK
    Color valueTypeField; // OK
};
```

gcroot<T>

- A natív kód nem tud a felügyelt referenciákról és a GC-ről
- A szemétgyűjtés miatt a felügyelt heapen lévő objektumok pozíciója a memóriában megváltozhat
  - Nem tárolhatunk felügyelt objektumra mutatót
- gcroot<T>: segédosztály, amivel egy felügyelt objektumra tárolhatunk referenciát natív típusban
  - A System::Runtime::InteropServices::GCHandle segédosztályt használja

Natív és felügyelt típusok együtt

```
public ref class ManagedClass {
public:
    NativePerson NativeType(NativePerson p); // OK, de csak C++-ból lesz hívható, C#-ból nem.
    ManagedPerson^ ManagedWithHandle(ManagedPerson^ mp); // OK
    ManagedPerson ManagedByValue(ManagedPerson mp); // OK, de szokatlan, C#-ban nem is így jelenik meg a visszatérési érték, hanem kimeneti paraméterként, mert referenciatípusokat C#-ban nem lehet érték szerint átadni.
    NativePerson nativePerson; // ERROR: felügyelt típusban nem lehet natív mező.
    NativePerson& npRef; // OK: Referencia lehet.
    NativePerson* npPtr; // OK: És pointer is.
    ManagedPerson^ managedPerson; // OK.
    ManagedPerson managedPersonByValue; // OK.
    Color valueTypeField; // OK
};
```

Natív típusok a felügyelt osztályban

- Az előző példából:

```
public ref class ManagedClass {
    ...
    NativePerson& npRef; // OK: Referencia lehet.
    NativePerson* npPtr; // OK: És pointer is.
    ...
};
```

- Ezek gond nélkül tárolhatók
- Egy natív referencia vagy pointer gyakorlatilag egy memóriacím, mintha csak egy long típusú mező lenne
- C#-ban ezek nem jelennek meg
  - A natív típusokat a .NET típusrendszere nem ismeri

Generikus típusok

- .NET generikus típusok funkciója hasonló, mint a C++ template-eké, de a működési elvük más. A szintaktikájuk is hasonló:

```
generic<typename T> where T : Shape
ref class GeometryManager {
    void Store(T^ shape);
    void Draw(T^ shape);
};
```

where T : Shape: Megkötés a típusparaméterre (C++20-ban van template-re is)

Generikus típusok vs. template-ek

- Generikus típusok: .NET-es runtime mechanizmus
  - Benne marad az IL kódban
- C++ template-ek: fordításidejű kódgenerálás
  - Fordítás közben eltűnik
- A kettő tud egyszerre működni
  - Template generikus osztály
  - Fordítás után marad a generikus típus minden olyan változatban, amilyen típussal használtuk a template-et

Egyéb

- .NET-es tömb: `array<int>^ ints;`
- Többdimenziós tömb: `array<String^, 2>^ strs;`
- Delegate-ek
- Minden meg van valósítva

C# != .NET

```
public ref class ClassWithIndexer
{
public:
    property int default[int] // Fordul, elérhető C#-ból (indexer)
    {
        int get(int index) { return 0; }
        void set(int index, int value) {}
    }
    property int prop2[int] // Fordul, de nem érhető el C#-ból, reflectionnel igen
    {
        int get(int index) { return 0; }
        void set(int index, int value) {}
    }
};
```

C++/CLI, összefoglalás

- Magas szintű megoldás
  - Egy új nyelvet kellett hozzá létrehozni
- A .NET-es, C#-ban elérhető funkciók leképezése C++-ra nyelvi kiegészítésekkel
- C# oldalról megszokott .NET API-t látunk
- Fontos: A fordító függvényenként más
  - .NET-es függvényekre a .NET-es + JIT
  - Natív C++-os függvényekre az optimalizáló fordító – ami gyorsabb kódot eredményez

Objective-C++, Swift

Objective-C – C++ interop

- Tud C-stílusú függvényeket hívni
  - Ugyanaz a probléma, mint az összes többi platformnál
- A fordító (Clang) mindkét kódot képes fordítani!
  - Egy fájlban is
- Könnyű az átjárás, Objective-C és C++ szabadon keverhető
- Az objektummodellek eltérései miatt van néhány megkötés, például:
  - Objective-C osztály nem származhat C++ osztályból és fordítva
  - Ha C++ típusú mezőt tartalmaz egy Objective-C osztály, annak csak a default konstruktorát tudja meghívni

Objective-C és Objective-C++

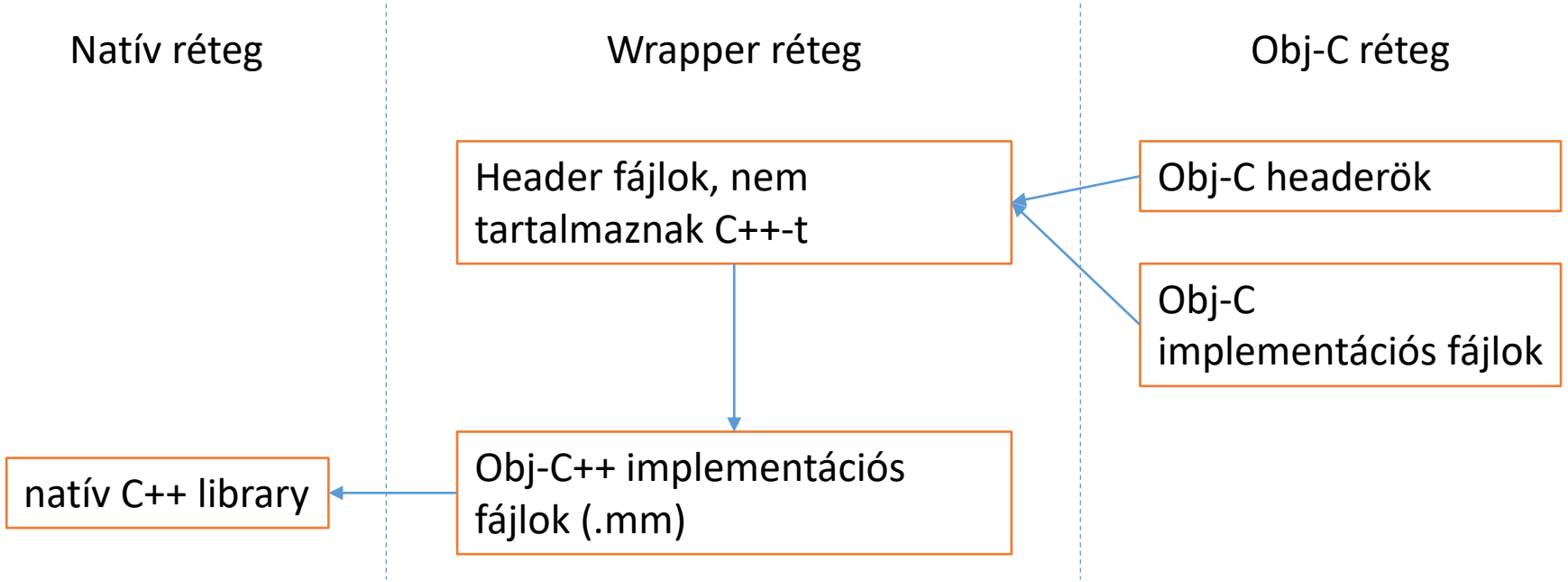
- C++ és Objective-C együttes használatához egy külön fordítási mód (Objective-C++) kell
- Obj-C fájlok kiterjesztése .m, Obj-C++ fájloké .mm
- Ha egy Obj-C headerben include-olunk C++-kódot

```
#include <map>
@interface MyClass : NSObject {
    @private
    std::map<int, id> lookupTable;
}
// ...
@end
```

  - Innentől kezdve ezt a headert csak Obj-C++ fájlokban lehet include-olni
  - Láncreakció, lehet, hogy a teljes projektet Obj-C++ fordításra kell átalakítani, ez egy nagy projektnél problémás lehet

Wrapper réteg

- Bár keverhető a C++ és az Objective-C, itt is érdemes tudatosan szétválasztani a kettőt
- Csak a kettő közötti interop réteg legyen Objective-C++ módban fordítva



Wrapper réteg, megoldás 1: PIMPL

- Pointer to Implementation
- A headerben csak egy struct forward declaration és egy pointer, ezt Obj-C-ben is szabad
- Header fájl (.h):

```
struct MyClassImpl;

@interface MyClass : NSObject {
  @private
  struct MyClassImpl* impl;
}
// public method declarations...
- (id)lookup:(int)num;
@end
```

PIMPL

- C++ implementáció használata csak az .mm forrásfájlban
- Itt már lehet C++ típusokat használni

```
#import "MyClass.h"
#include <map>
struct MyClassImpl {
  std::map<int, id> lookupTable;
};
@implementation MyClass
- (id)init {
  self = [super init];
  if (self)  {
    impl = new MyClassImpl;
  }
  return self;
}
- (void)dealloc {
  delete impl;
}
- (id)lookup:(int)num {
  std::map<int, id>::const_iterator found =
    impl->lookupTable.find(num);
  if (found == impl->lookupTable.end()) return nil;
  return found->second;
}
@end
```

Wrapper réteg, megoldás 2: class extensions

- Obj-C 2.0 óta közvetlenül az implementációs fájlban deklarálhatók a tagváltozók (instance variable, ivar). Ebben az esetben ezek privátok lesznek, és a headerben nem is szerepelnek. Így itt használhatunk natív C++ típusokat, a headert pedig Obj-C-ben is include-olhatjuk.
- Header:

```
#import <Foundation/Foundation.h>

@interface ObjcObject : NSObject
- (void)exampleMethodWithString:(NSString*)str;
// other wrapped methods and properties
@end
```

Wrapper réteg, megoldás 2: class extensions

- Implementációs fájl (.mm), itt deklaráljuk a C++-ban implementált típussal rendelkező mezőt:

```
#import "ObjcObject.h"
#import "CppObject.h"

@implementation ObjcObject {
  CppObject wrapped; // Ez az objektum C++-ban van implementálva, akik a headert látják, azok erről nem tudnak.
}

- (void)exampleMethodWithString:(NSString*)str
{
  std::string cpp_str([str UTF8String], [str lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
  wrapped.ExampleMethod(cpp_str);
}:
```

Swift – C++ interop

- Tud hívni C-stílusú függvényeket
  - Közvetlenül hívjuk, ahol szükséges
    - Hibalehetőség, az interop kód szét van szórva, nehéz kézben tartani
  - Készíthetünk csomagoló osztályt minden függvényhez
    - C++ oldalon C-stílusú függvények hívnak tovább
    - Swift oldalon Swift csomagoló osztályt készítünk
- Nincs más interop lehetőség C++ hívásra
- Viszont tud hívni Objective-C-t!
  - Dupla csomagolás, Swift – Objective-C – C++
  - Ez macerás, de robosztus megoldás
  - Fordításidőben látjuk a hibákat

Kérdések?