# Multiplatform szoftverfejlesztés

## Új C++ nyelvi funkciók

# Automatikus memóriakezelés

## RAII – Resource Acquisition Is Initialization

# Automatikus memóriakezelés

- Pointerek használata veszélyes
- Könnyű memory leaket okozni

```
void use_gadget(int x)
{
  Gadget* gadget = new Gadget();
  ...
  if(x > 100) throw std::runtime_error("error!");
  ...
  delete gadget;
}
```

# Automatikus memóriakezelés

- Scoped variable
- Egyszerű, biztonságos
- Modern C++ *stílus*

```
void use_gadget(int x)
{
    Gadget gadget(x);
    ...
    if(x > 100) throw std::runtime_error("error!");
    ...
}
```

# Automatikus memóriakezelés

- Smart pointer: automatikus felszabadítás
- Referencia számlált
- Shared ownership

```
void use_gadget(int x)
{
    shared_ptr<Gadget> gadget =
        make_shared<Gadget>(x);
    ...
    if(x > 100) throw runtime_error("error!");
    ...
}
```

# Automatikus memóriakezelés

- unique_ptr – csak egy példány lehet
  - = operátor elpusztítja a régit
- weak_ptr – nem tartja életben az objektumot
  - Körkörös hivatkozásoknál
  - Például fa struktúra, ahol a szülő mutató weak

- allocator
  - Minden std osztály használja
  - Lehet saját

# Erőforrások kezelése

- C++ fontos tulajdonsága: minden erőforrást kézzel kezelünk
- Nincs garbage collector
  - Memória felszabadítása kézzel
  - Gyors, vagy lassú?
  - Produktív, vagy nem?
  - Ez jó, vagy rossz?

# Foglalás és felszabadítás

- Ezek a műveletek mindig párban vannak
- Szabványos fájl API:
  - fopen(): fájl megnyitása
  - fclose(): fájl handle bezárása
- POSIX:
  - socket(): TCP socket létrehozása
  - close(): socket lezárása
- Win32 Mutex API:
  - WaitForSingleObject(): mutex lockolása
  - ReleaseMutex(): mutex elengedése

# Hibalehetőség

- Nehéz helyesen használni

```
void readFile()
{
    FILE* file = fopen("test.txt", "r");
    // feldolgozás...
    fclose(file);
}
```

- A fájl nem fog felszabadulni, ha
  - A feldolgozás során kivétel dobódik => try-catch
  - return, (goto) => át kell nézni a kódot, vagy __try-__finally (MS specifikus)
  - Lelőjük a szálat

# Megoldás: RAII

- Resource Acquisition Is Initialization
- Az „erőforrást” egy lokális objektummal reprezentáljuk
  - A konstruktorban lefoglaljuk
  - A destruktorban felszabadítjuk
- A lokális objektum mindenképp felszabadul

# Példa: fájl handle

- Fájlt reprezentáló osztály

```
class FileHandle {
public:
    FileHandle(const char* n, const char* rw) {
        f = fopen(n, rw);
        if (!f)
            throw "error";
    }
    ~FileHandle() { fclose(f); }
    // ...
private:
    FILE* f;
};
```

# Példa: fájl handle

- Egyszerűen lokális objektumként hozzuk létre
- Blokk/függvény végén felszabadul

```
void readFile()
{
    FileHandle("test.txt", "r");
    // Feldolgozás...
    // Nem kell kézzel felszabadítani.
}
```

# Memória, mint erőforrás

- Ebből a szempontból a memória nem más, mint egy erőforrás, csak ezt sokkal gyakrabban használjuk
  - malloc/free
  - new/delete
- Az erőforráskezeléshez hasonlóan a memóriakezelést se végezzük kézzel
  - A shared_ptr és unique_ptr típusok pont a RAII mintát valósítják meg a memóriára

# Memória foglalás

- Nem RAII – nincs automatizmus
  - malloc, free
  - new, delete
  - new[], delete[]
  - Globális változók és TLS (thread local storage)
    - Ettől még ezek jók lehetnek, de nem szabadulnak fel
- RAII – automatikusan felszabadulnak
  - Globálisan: shared_ptr, unique_ptr, weak_ptr
  - Vermen változó és _alloca

# Garbage collection

- Magasabb szintű nyelvekben van GC
  - Automatikusan kezeli a memóriát és szabadítja fel a már nem használt objektumokat
- C++-ban alapból nincs, RAII helyettesíti
  - Third-party GC-implementációk használhatók
- Vannak hibrid nyelvek
  - D
  - Managed C++

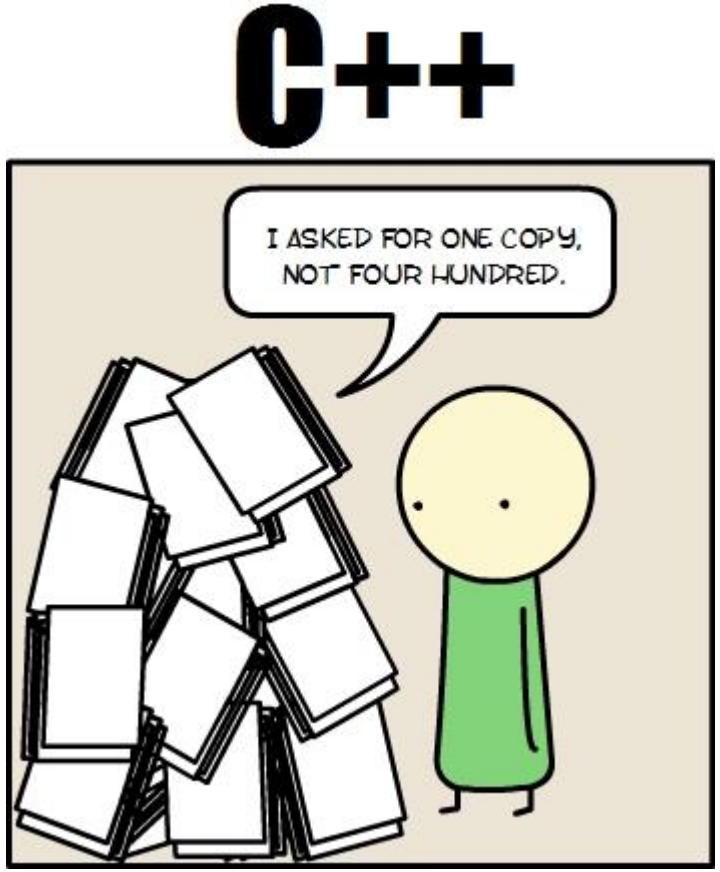
# RAII vs. GC

- Mindkettőre igaz
  - Kényelmes – nem kell felszabadítani kézzel
  - Produktív – nehéz hibázni, ha követjük a szabályokat
- Különbségek
  - Hibázási lehetőség kevesebb GC-vel
    - Nem kell figyelni körkörös hivatkozásra
  - RAII determinisztikus, jobban kézben tartható
  - Sebesség más (GC gyorsabb lehet általános esetben)

# Move konstruktor

# Copy konstruktor elkerülése

- Alapprobléma
  - C++-ban könnyen készül másolat egy objektumról
  - Nagy objektumok vagy sok objektum esetén a másolás drága



# Példa

- Hogyan adjunk vissza egy nagyméretű tömböt?

```
? createData();
```

| Megoldás | Hátrány |
|---|---|
| std::vector<int> createData(); | Másolás |
| std::vector<int>* createData(); | Memóriakezelés |
| void createData(std::vector<int>& result); | Nehezen olvasható, operátorokra nem működik |

- Az elsőt szeretnénk hátrány nélkül

# lvalue, rvalue

- objektum: egy folytonos terület a memóriában
- lvalue: egy objektumra referáló kifejezés, ami tovább él a kifejezésnél (állhat egyenlőség bal oldalán)
  - int i;
- rvalue: minden más (ideiglenes)
  - i + 7;

# lvalue, példák

```
int i;
i = 5;

const int ci = 5;
ci = 6; // ERROR

int a[5];
a[3] = 2;

std::string& getStr() { return globalStr; }
getStr() = "modifiy"; // Meglepő szintaktika
```

- a példákban ezek az lvalue-k
  - i
  - ci
  - getstr()
  - a[3]

# rvalue

```
std::string createStr() { return "alma"; }

auto addr1 = &(createStr()); // ERROR
auto addr2 = &(std::string("alma")); // ERROR
int i = 2;
(i + 1) = 5; // ERROR
std::string("alma") = "narancs"; // OK (!)
createStr() = "narancs"; // OK (!)
```

- rvalue-k
  - createStr() – értékadás bal oldalán áll, mégsem lvalue
  - std::string("alma")
  - (i + 1)
- Az „Állhat értékadás baloldalán” nem teljeskörű definíció

# lvalue, rvalue

```
std::vector<int> createData()
{
    std::vector<int> temp;
    // adatok előállítása...
    return temp;
}
```

- C++ 98: a fordító tudta, hogy mi lvalue, vagy rvalue
  - De nem volt rá mód a nyelvben, hogy megkülönböztessük azokat a kifejezéseket, amik olyan objektumot reprezentálnak, ami épp megszűnőben van

# rvalue reference

- C++11: kiegészült a nyelv egy új referenciatípussal: **T&&**
- csak rvalue-hoz köt, módosítható
- Egy megszűnőben lévő objektumot reprezentál, „kilophatjuk” a tartalmát
- Ezzel elkerülhetjük a másolást

# Példa: doubleVector pszeudokód

```
class doubleVector
{
    int count;
    int capacity;
    double* elements;
    doubleVector() : capacity(5), count(0), elements(new double[5]){ }
    ~doubleVector() { delete[] elements; }
    void add(double d);
    void remove(int index);
    double get(int index);
};
```

# doubleVector COPY ctor

```
class doubleVector
{
    doubleVector(const doubleVector& other) :
        capacity(other.capacity),
        count(other.count),
        elements(new double[other.capacity])
    {
        // Klasszikus COPY ctor, lemásoljuk a teljes tömböt.
        for (int i = 0; i < count; i++)
            elements[i] = other.elements[i];
    }
}
```

# doubleVector MOVE ctor

```
class doubleVector
{
    doubleVector(doubleVector&& other) :
        capacity(other.capacity), count(other.count)
    {
        // MOVE: kilopjuk a reprezentációt (memóriát).
        // Nincs másolás.
        elements = other.elements;

        // Töröljük a forrás objektumban a pointert.
        // Fontos, különben a destruktorban felszabadítanánk.
        other.elements = nullptr;
    }
}
```

# Mikor hívódik MOVE?

```
doubleVector createVec() { /* ... */ }

void useVec(doubleVector vec) { /* ... */ }

void foo() {
    doubleVector vec;

    useVec(vec); // COPY, mert vec lvalue
    useVec(doubleVector()); // MOVE
    useVec(createVec()); // MOVE
}
```

# move manuálisan

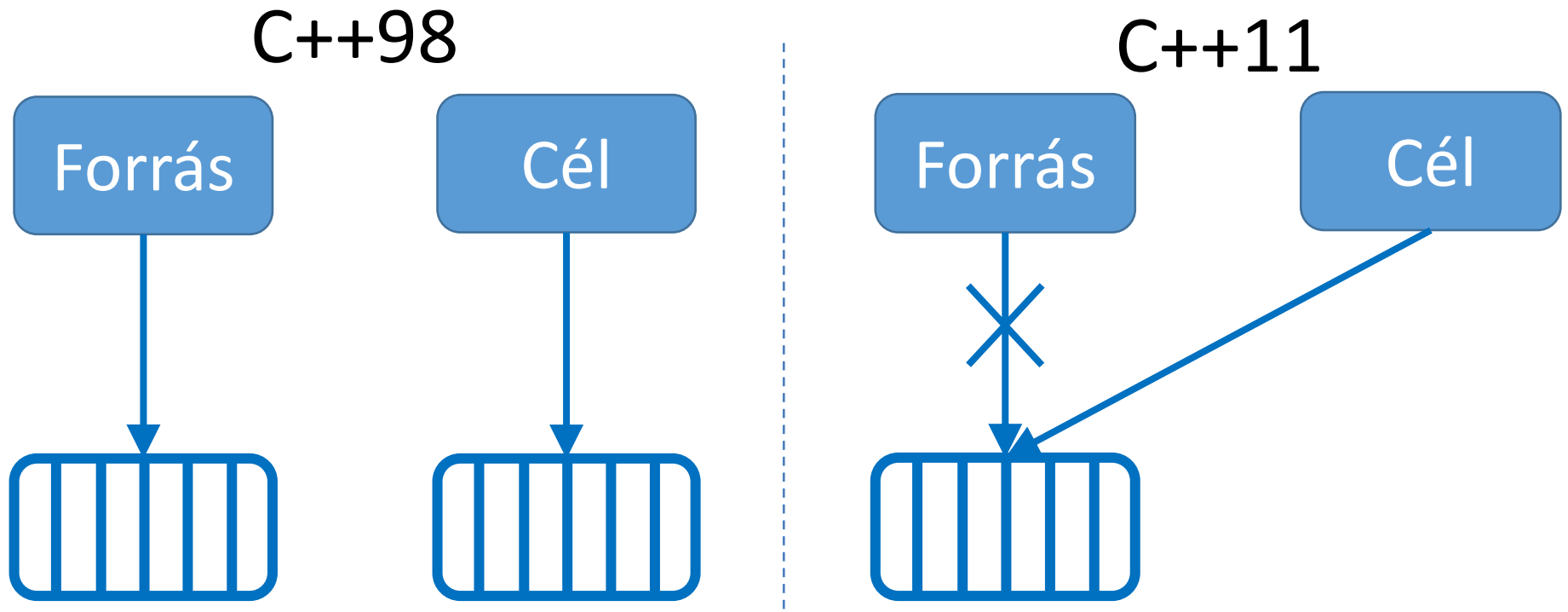
- Ha tudom, hogy egy egyébként lvalue kifejezés által reprezentált objektumot később már nem használok

```
doubleVector vec;
useVec(std::move(vec)); // MOVE
// Ezután már nem szabad vec-
et használni, lehet, hogy a useVec fv. kilopta a tartalmát.
```

- std::move: igazából nem több, mint kasztolás rvalue reference-re
- Így érték szemantikával átadhatunk nem másolható objektumokat is, pl. thread

# Visszaadás érték szerint

- Nincs másolás, hatékony
- Az összes STL-konténer támogatja
- Jelentős gyorsulást jelenthet a fordító frissítése



# Lambda kifejezések

# Logika átadása

- Gyakran szükséges logikát paraméterként átadni, például:
  - algoritmusok
  - callback függvények
  - eseménykezelés
  - strategy pattern (futásidőben választható algoritmus, például plugin)

# Logika átadása: függvénypointer

```
void execFunc(void (*func)()) { func(); }
void myFunc() { std::cout << "alma"; }
void foo3()
{
    execFunc(&myFunc);
}
```

- (Lehet referencia is)
- Specifikus típus
- Nehezen olvasható szintaktika
- Tagfüggvények típusa más, nehézkes használni

# Példa: sort algoritmus

```
#include <algorithm>
bool cmpStrLength(const std::string& a, const std::string& b)
{
    return a.length() < b.length();
}
std::vector<std::string> strings=...;

std::sort(strings.begin(), strings.end(), cmpStrLength);
```

# Functor objektum

```
struct strLengthComparer
{
    bool operator()(const std::string& a, const std::string& b)
    {
        return a.length() < b.length();
    }
};
std::vector<std::string> strings;
strLengthComparer cmp;

std::sort(strings.begin(), strings.end(), cmp);
```

# Lambda expressions

- Logika definiálása egyetlen kifejezésben
- Functor objektumot generál

```
std::vector<std::string> strings;
// ...

std::sort(
    strings.begin(),
    strings.end(),
    [] (const std::string& a, const std::string& b)
    {
        return a.length() < b.length();
    });
```

# Lambda expressions

- Az átadás helyén van az implementáció, könnyen olvasható
- Érdemes használni, ha
  - egyszerű logikát implementál
  - csak egyetlen helyen van rá szükség
- Egy functor objektum több helyen újra használható
  - Lambát is eltehetjük egy változóba és újra felhasználhatjuk – nem az igazi
- A functor objektum típusának a neve dokumentálja a funkcióját, a lambdának nincs neve

# Lambda expressions: capturing

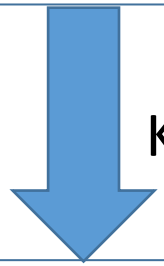
- Ha egy, a lambda scope-jában lévő változót szeretnénk használni a lambda kódjában

```
int count = 0;
std::generate(beg, end, [&count] (){ return count++; });
```

  - [&count]: capture referencia szerint
  - [=count]: capture érték szerint
  - [&]: minden hivatkozott változó referencia szerint
  - [=]: minden hivatkozott változó érték szerint
  - [=, &count]: minden hivatkozott változó érték szerint, kivétel count (ami referencia szerint)
  - []: nincsen capture

# Lambda expressions: fordítás

```
std::vector<int> indices(10);
int count = 0;
std::generate(indices.begin(), indices.end(),
    [&count] (){ return count++; });
```



Kódgenerálás a fordítás során

```
struct generatedFuncClass {
    int& capturedInt; // capture by ref!

    generatedFuncClass(int& i) : capturedInt(i) { }

    int operator()() { return capturedInt++; }
};

std::vector<int> indices(10);
int count = 0;
generatedFuncClass gfc(count);
std::generate(indices.begin(), indices.end(), gfc);
```

# Lambda expression részei

```
string name = "blue";

auto concatNameWithStr = [&] (string str) -> string
{
    string result = name;
    result.append(str);
    return result;
};

string result = concatNameWithStr("green");
cout << result; // prints "bluegreen"
```

capture list

paraméterlista (opcionális)

visszatérési érték (opcionális)

akció

# Lambda expression típusa

```
auto x = [] (int a, int b) { return a < b; }
```

- Változó típusa nem specifikált
  - Gyakorlatban a típusa a fordító által generált functor osztály
- az auto helyére nem tudnánk konkrét típusnevet írni
- std::function: burkoló objektum egy függvényre, functor objektumra vagy lambdára

```
std::function<bool(int, int)> cmpIntsFunc =
    [] (int a, int b) { return a < b; };
```

# Lambda expression típusa

- typeid operátorral kiírathatjuk egy kifejezés típusát (RTTI-nek engedélyezve kell lennie)

```
auto cmpInts = [] (int a, int b) { return a < b; };
std::function<bool(int, int)> cmpIntsFunc =
    [] (int a, int b) { return a < b; };
std::cout << typeid(cmpIntsFunc).name() << std::endl;
std::cout << typeid(cmpInts).name() << std::endl;
```

- Kimenet:

```
class std::function<bool __cdecl(int,int)>
class <lambda_b853351245db9a879f640980a0f46f1d>
```

# std::sort

- Ezekkel hívtuk meg a sort-ot:
  - függvénypointer
  - functor objektum
  - lambda-kifejezés
  - std::function objektum
- Hogyan tudja a sort fv. bámelyiket fogadni?
  - template paraméter!

# Függvény átadása template-tel

- template: kódgenerálás, bármi működik, ha a generált kód fordul

```
template <typename Comparer>
void doubleSort(vector<double>& vec, Comparer cmp)
{
    // sort algoritmus kódja
    int i;
    // ebben valahol használjuk a cmp-t összehasonlításra.
    if (cmp(vec[i], vec[i + 1]))
    {
        // ...
    }
    // ...
}
```

# Függvény átadás template-tel

- Az előző példában így használtuk a cmp paramétert

```
if (cmp(vec[i], vec[i + 1]))
```

- A függvény template-nek bármilyen olyan cmp argumentumot átadhatunk, amire a fenti kódsor fordul
  - függvénypointer bool (double, double) függvényre
  - függvényreferencia bool (double, double) függvényre
  - funktor objektum
  - lambda
  - std::function (ez egy szabványos funktor objektum)
- A template típusú paraméter jó olyan esetben, ha nem tudjuk előre a paraméter típusát, vagy a típust nem lehet kifejezni a nyelvben

# std::sort szabványos szignatúrája

```
template <class RandomAccessIterator, class Compare>
void sort(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp);
```

# Inicializálás

# initializer lists

- C++11-től kezdve nem csak tömbök inicializálhatók {3, 5, 11} szintaktikával

```
vector<double> v = { 1, 2, 3.456, 99.99 };
list<pair<string, string>> languages = {
    { "Nygaard", "Simula" }, { "Richards", "BCPL" }, { "Ritchie", "C" }
};
map<vector<string>, vector<int>> years = {
    { { "Maurice", "Vincent", "Wilkes" }, { 1913, 1945, 1951, 1967, 2000 } },
    { { "Martin", "Ritchards" }, { 1982, 2003, 2007 } },
    { { "David", "John", "Wheeler" }, { 1927, 1947, 1951, 2004 } }
};
```

# initializer list támogatása

- Új konstruktor

```
template<class E> class myVector {
    E* elements;
    int size;
public:
    // initializer-list constructor
    myVector(initializer_list<E> s)
    {
        elements = new E[s.size()];
        copy(s.begin(), s.end(), elements);
        size = s.size();
    }
};
```

# uniform initialization

- C++ 98-ban több szintaktika van objektumok incializációjára:

```
string a[] = { "foo", " bar" }; // OK: tömb
vector<string> v = { "foo", " bar" }; // Hiba: nem tömb
void f12(string a[]);
f12({ "foo", " bar" }); // Hiba: blokk argumentumként

int a = 2;                  // assignment style
int aa[] = { 2, 3 };        // assignment style listával
complex z(1, 2);             // functional style
x = Ptr(y);                 // functional style konverzióhoz/kasztoláshoz

int a(1); // változó definíciója
int b(); // függvény deklarációja
int b(foo); // változódefiníció vagy függvénydeklaráció
```

# uniform initialization

- C++11: egységes szintaktika kapcsos zárójelekkel

```
X x1 = X{1, 2};
X x2 = {1, 2}; // Opcionális egyenlőségjel
X x3{1, 2};
X* p = new X{1, 2};
struct D:X{
    D(int x, int y):X{x, y}{};
};
struct S{
    int a[3];
    // Régi problémára megoldás:
    S(int x, int y, int z):a{x, y, z}{};
};
```

# Tag inicializáció (C++20)

- Mint C# var a=A{m=2};

```
struct A {
  string str;
  int n = 42;
  int m = -1;
};
auto a=A{.m=21}
```

# uniform initialization

- Nem végez szűkítő converziót (narrow cast)

```
int x=2.5; // OK, truncates
int x{2.5}; // ERROR, narrowing conversion
```

- auto + kapcsos zárójelek: initializer_list!

```
auto x1=5; // int
auto x2(5); // int
auto x3{5}; // std::initializer_list<int>
auto x4={5}; // std::initializer_list<int>
```

# Kérdések?