Multiplatform szoftverfejlesztés

C++ ismétlés, új nyelvi elemek

Mikor használunk C++-t?

- Régi kódbázis (legacy kód)
- Bizonyos célplatformokon csak ez van
  - Tipikusan beágyazott rendszerek
- Összetett GUI alkalmazások
  - Amikor a vezérlők száma több ezer, azt kevés keretrendszer bírja
  - Törekszünk az egyszerű UI-ra, ha lehet
- CAD/multimédia alkalmazások
- Más nyelvek értelmezője/fordítója

Mikor használunk C++-t?

- Nagy adatmennyiség kezelése
  - Adatbázis motor, folyamatirányítás, videó enkódolás/dekódolás
- Operációs rendszerek
- Algoritmusok
- Játékok
  - Sok játék motor C#/egyéb felületet ad

Gyors, de mégis mennyire?

```
int x = 0;
for (int i = 0; i < 1000000; i++)
{
  int y = i % 1000 == 0 ? 1 : 0;
  for (int j = 0; j < i / 10; j++)
    x += y;
}
```

- C++: 1 ms (x += y * y => 2s)
- C#: 13s
- JS (Chrome): 51s
- JS (régi Edge): 41s
- JS (Firefox): 59s

Sebesség teszt

- Ha az adattípus csak double lehet, akkor JS gyors (mint C++)
  - Vagy használhatunk WebAssembly-t
- C++ akkor tud gyorsabb lenni, ha
  - Adatformátum kedvező
    - int: 32 bitre fordítva
    - long long: 64 bitre fordítva
  - A probléma megoldható tömör adattal
    - Uniók, bitműveletek, változó igazítás
  - Speciális utasításkészlet
    - SSE2, AVX2
  - Párhuzamos algoritmus, GPU

C++ ismétlés, alapok

Történelem

- Bjarne Stroustrup
- 1979: C with Classes
  - Cél: mechanizmusok kialakítása bonyolultság kezelésére nagy alkalmazások fejlesztésében
  - Alap: C, gyors, portable, széleskörben elterjedt
  - Elképzelés: C-t kiegészíteni osztályokkal, leszármazással
- 1983: C++, virtuális függvények, overloading, referencia
- 1989: C++ 2.0, static, többszörös öröklés, absztrakt osztályok, template-ek – ez terjedt el széles körben, majd szabvány 1998
- 2011: C++11 (eredeti nevén C++0x), számos új feature
- 2014: főleg javítások, apróságok
- 2017: fő verzió
- 2020: fő verzió, rengeteg nyelvi változás ismét

C++

- Definíció: Vékony absztrakciós rétegű programozási nyelv
  - Ma már egyre kevésbé igaz ez
- Fontosabb jelzők
  - Általános célú
  - Közvetlen hardver leképezés
  - Nulla overhead
  - Osztályok
  - Öröklés
  - Paraméteres típusok (template)

Függvények

Deklaráció (több is lehet egy függvényhez):

```
int add(int a, int b);
void print(std::string msg);
int add(int foo, int bar);
int add(int, int);
```

Definíció (Pontosan egynek kell lennie):

```
int add(int a, int b) {
    return a + b;
}
void print(std::string msg) {
    cout << msg;
}
```

Osztályok

```
class Stack {
public:
    void Push(int value);
    int Pop() {
        return stack[top];
    }
private:
    int top;
    int stack[10];
};

void Stack::Push(int
value) {
    stack[top++] = value;
}
```

- public látható kívülről, private nem
- class és struct kulcsszó majdnem ugyanaz, struct alapból public
- definíció lehet külön a deklarációtól (tipikusan deklaráció headerben, definíció a forrásfájlban)

Objektumok létrehozása

```
void foo()
{
    Stack s1; // Automatikus objektum
    s1.Push(10);
    Stack* s2 = new Stack(); // Dinamikus objektum
    s2->Push(10);
    delete s2; // Dinamikus objektum felszabadítása
               // ha nem volt exception
    // s1 automatikusan felszabadul,
    // amikor a scope-ból kilépünk
}
```

Objektumok átadása

```
void useStack1(Stack s) { s.Push(5); }
void useStack2(Stack* s) { s->Push(5); }
void useStack3(Stack& s) { s.Push(5); }

void foo2() {
    Stack s1;
    Stack s2 = s1; // Másolás
    Stack* s3 = &s2; // Address-of operátor, memóriacím
lekérése

    useStack1(s1); // Másolás (átadás érték szerint)
    useStack2(&s1); // Nincs másolás (átadás cím szerint)
    useStack3(s1); // Nincs másolás (átadás cím szerint)
}
```

Pointerek és referenciák

- Ajánlás: mindig használjunk referenciát. Pointert csak akkor, ha muszáj. Referencia biztonságosabb, könnyebben olvasható

	Pointer	Referencia
típus	`Stack* s`	`Stack& s`
member elérése	`s->Push(5)`	`s.Push(5)`
default érték	`nullptr`	nincs! `Stack& s; // ERROR, kötelező értéket adni`
átadás	Memóriacím	memóriacím
aritmetika	`++, --, +, -`	nincs
módosítás	Tetszőleges	nem lehet módosítani
többszörös indirekció	igen, pl.: `Stack** s`	nincs

Leszármazás

```
class Base {
private:
    void privFoo() { /* ... */ }
protected:
    void protFoo() { /* ... */ }
public:
    virtual void Foo();
};

class Derived : Base {
    void Foo() // Felülimplementálja Base::Foo-t.
    {
        privFoo(); // ERROR
        protFoo(); // OK
    }
};
```

Többszörös öröklés

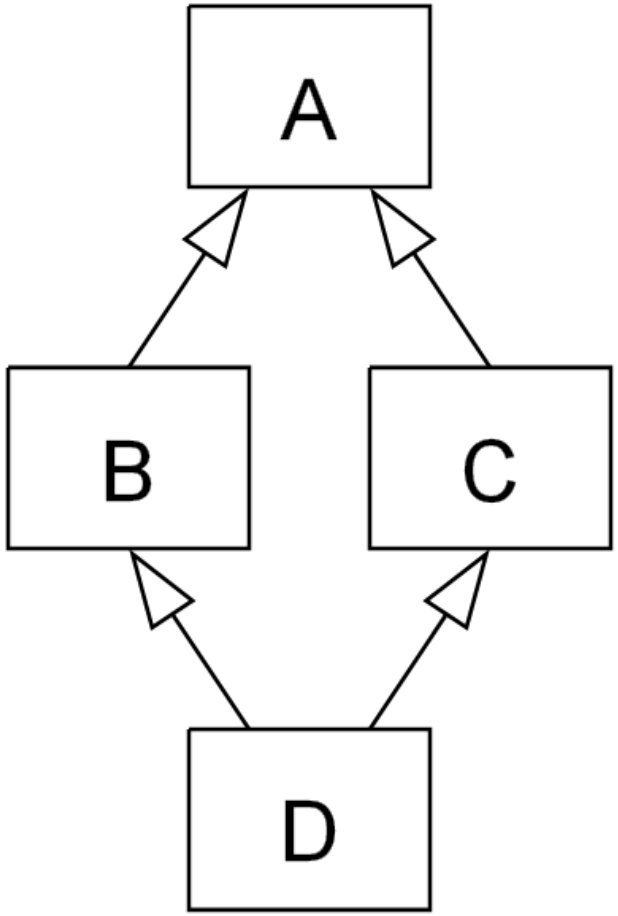
```
struct CommonBase {
    int foo();
};

struct BaseA : CommonBase {
    int foo() { return 1; }
};

struct BaseB : CommonBase {
    int foo() { return 2; }
};

struct Derived : BaseA, BaseB {
};
```

```
int main()
{
    Derived d;
    cout << d.foo(); // ERROR,
ambigous access.
    // Nem egyértelmű, hogy
melyik örökölt
    // függvényt szeretnénk hívni

    cout << d.BaseA::foo(); // 1
    cout << d.BaseB::foo(); // 2
}
```

Virtuális öröklés

- Több öröklési úton keresztül örököljük ugyanazt a tagot
- Hányszor tároljuk az ős adattagjait?

```
struct A {
    int foo();
};

struct B : public virtual A {
    int foo() { return 1; }
};
```



Többszörös öröklés, interfészek

- Interfészek: „pure virtual” osztályok
- pure virtual: nincs adattagja és implementációja
- pure virtual osztályokból nyugodtan örökölhetünk többszörösen
- Egy szerződést írnak le, amit a leszármazottnak meg kell valósítania

Többszörös öröklés, interfészek

```
class Drawable {
    virtual void Draw() = 0; // Pure virtual function
};
class SupportsInteraction {
    virtual void Click() = 0; // Pure virtual function
};
class Button : Drawable, SupportsInteraction {
    void Draw();
    void Click();
};
class Rectangle : Drawable {
    void Draw();
};
```

Generikus programozás

- Cél: hasznos algoritmusok és adatstruktúrák általánosítása paraméterezéssel
- Típusok más típusokkal

```
vector<int> intVect;
vector<Shape> shapeVect;
```

- Algoritmusok más algoritmusokkal

```
bool compareShapes(const Shape& s1, const Shape& s2)
{
   return s1.X > s2.X;
}
sort(shapeVect, compareShapes);
```

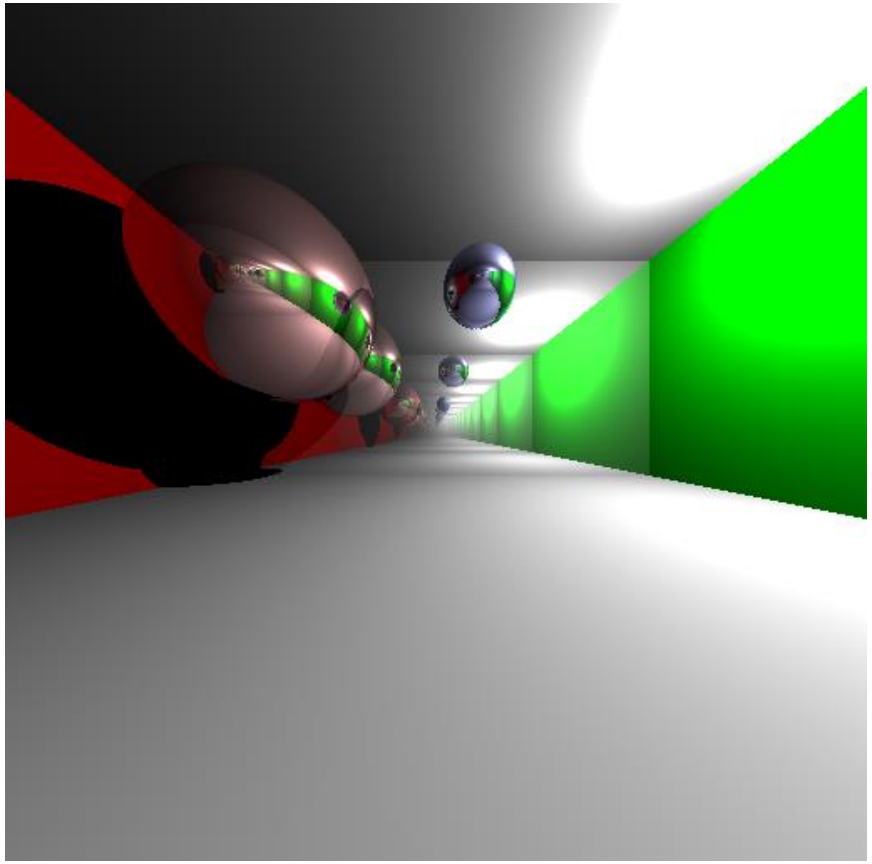
(Template metaprogramozás)

- Logika implementálása template-ekkel
- Kiértékelés **fordítási** időben

```
template <int n>
struct factorial {
    enum { value = n * factorial<n - 1>::value };
};

template <>
struct factorial<0> {
    enum { value = 1 };
};
```

metatrace

- Ray tracer fordítási időben



```
private:
        typedef typename whitted_mirror<
                intersection,
                ray,
                whitted_style::raytrace,
                depth_max
        >::color mirror_color;

        typedef typename intersection::material::reflection reflection;
        typedef typename intersection::material::color      surface_color;
public:
        typedef ift<
                intersection::does_intersect,
                color::add_rgbf<
                        color::mul_rgbf<
                                diffuse_color,
                                scalar::sub<scalar::c1,reflection>
                        >,
                        color::mul_rgbf<
                                color::filter_rgbf<
                                        surface_color,
                                        mirror_color
                                >,
                                reflection
                        >
                >,
                background_color
        > color;
```

C++ újdonságok

C++11, 14, 17, 20

Szabványosítás

- C++98: A már elterjedt programnyelv szabványosítása
- C++03: Hibajavítások
- C++11: Eredetileg C++0x volt a neve, mert évekig azt hitték, hogy 2010 előtt kijön
- C++14: Hibajavítások
- C++17: Fő cél az osztálykönyvtárak bővítése
- C++20: Rengeteg újdonság (nyelvi is)

C++ 11, a megkésett óriás

- Az első jelentős újítás 1989 (1998) óta.
- Számos új nyelvi és STL feature
- Sokat implementáltak már korábban a fordítók, de a szabványtól kezdve lehet megbízhatóan használni
- Összességében: koherensebb, könnyebben használható
- Jobb teljesítmény, biztosabb memóriakezelés
- Több feature, egyszerűbb használat

auto

- C++-ban sok esetben hosszú típusnevek

```
std::map<int, my_namespace::Gadget> gadgets;
std::map<int, my_namespace::Gadget>::const_iterator
        it = gadgets.begin();
```

- Ehelyett

```
auto it = gadgets.begin(); // Honnan tudja a típust?
```

Kollekción iterálás

```
for(auto it = shapes.begin(); it != shapes.end(); it++)
        it->Foo();

for(auto& shape : shapes)
        shape.Foo();
```

auto iterációs paraméter

```
vector<string> v; // Melyik a gyorsabb?
for (const auto& s1 : v) { }
for (const string& s2 : v) { }
// Egyforma.

map<string, int> m; // Melyik a gyorsabb?
for (const auto& p1 : m) { }
for (const pair<string, int>& p2 : m) { }
// const auto& a gyorsabb, mert m értékeinek típusa pair<const string, int>.
// Mivel a típusuk különbözik, ezért egy másolat jön létre.
```

enum problémák

- Az enumerátorok láthatók az egész scope-ban

```
// Fordítási hiba
enum Animals { Bear, Cat, Chicken };
enum Birds { Eagle, Duck, Chicken };
```

- Automatikus int-konverzió

```
int i = Chicken;
bool b = Eagle && Duck; // wat?
```

- A mögötte lévő típus nem szabályozható

enum class

```
// Működik
enum class Birds { Eagle, Duck, Chicken };
enum class Animals : unsigned char { Bear, Cat, Chicken };
auto i = Chicken; // Fordítási hiba
int i = Birds::Chicken; // Fordítási hiba
struct MyStruct
{
	Birds BirdsField; // 4 byte
	Animals AnimalsField; // 1 byte
};
```

Üres pointer, C++ 98

```
#define NULL 0
```

- Két jelentés: egész szám és pointer konstans

```
void f(char const *ptr);
void f(int v);
int n = NULL; // Lefordul.
f(NULL); // A második overload hívódik meg.
```

- Könnyen okozhat hibákat

nullptr

- nullptr: pointer literal, a típusa std::nullptr_t, az értéke 0.
- NULL konvertálható std::nullptr_t típusra
- nullptr nem kasztolható implicit módon integerré

```
void f(char const *ptr);
void f(int v);
int n = nullptr; // Fordítási hiba.
f(nullptr); // Az első overload hívódik meg.
```

override leszármazott osztályban

```
struct B {
	virtual void f();
	virtual void h(char);
	virtual void g() const;
	void k(); // Nem virtuális
};
struct D : B {
	void f(); // B::f()-et implementálja felül
	virtual void h(char); // B::h()-t implementálja felül
	void g(); // Nem implementál felül (rossz típus)
	void k(); // Nem implementál felül (B::k() nem virtuális)
};
```

Függvény-felüldefiniálást nem kötelező kulcsszóval jelölni

override kulcsszó

```
struct B {
	virtual void f();
	virtual void h(char);
	virtual void g() const;
	void k(); // Nem virtuális
};
struct D : B
{
	void f() override; // OK: B::f()-et implementálja felül
	void g() override; // Hiba: rossz típus
	virtual void h(char) override; // B::h()-t implementálja felül
	void k() override; // Hiba: B::k() nem virtuális
};
```

Függvény törlése

- Előfordul, hogy egy osztály másolását szeretnénk tiltani kívülről

```
class nonCopyable {
public:
    nonCopyable() { }
private:
    nonCopyable(const nonCopyable& other);
};
nonCopyable instance;
nonCopyable copy = instance; // ERROR
// C2248: 'nonCopyable::nonCopyable' : cannot access private member declared in class 'nonCopyable'
```

Függvény törlése

- C++11-ben explicit „törölhető” egy függvény

```
class nonCopyable
{
public:
    nonCopyable() { }
    nonCopyable(const nonCopyable& other) = delete;
};
nonCopyable instance;
nonCopyable copy = instance; // ERROR
// C2280: 'nonCopyable::nonCopyable(const nonCopyable &)' : attempting to
reference a deleted function
```

Függvény törlése

- Nem kívánt konverziók is letilthatók

```
struct Z
{
    // ...
    Z(long long);     // long longgal inicializálható
    Z(long) = delete; // de kisebb egésszel nem
};
```

Alapértelmezett implementáció

- Ha elveszítjük az alapértelmezett konstruktor implementációt
  - Mert például létrehozunk egy paraméteres konstruktort
- Hasonlító operátorhoz is C++20-tól

```
class A
{
    A(int);
    A() = default;
};
```

long long

- Eddig fordító specifikus megoldások
  - __int64
- Fontos a támogatása
  - 64 bites művelet atomi a megfelelő processzoron

```
long long a; // 64 bit
```

String literal

- Korábban volt
  - “hello” // const char* (ne használd)
  - L”hello” // const wchar_t*, 16/32 bit/char, Unicode 16/32, (ne használd)
- C++11
  - u8”hello” // UTF8, const char*
  - u”hello” // UTF16 , const char16_t* (ne használd)
  - U”hello” // UTF32, const char32_t*
  - R"(\w\\\w)"; // kombinálható mindegyikkel, raw string
- C++17
  - u8’h’ // UTF8 karakter, char (ne használd)
- Karaktereket általában ne használjunk
  - Például ß vs. SS, vagy i vs. İ

User defined literal

```
101010111000101b // binary
1.2i             // imaginary

constexpr complex<double> operator "" i(double d)
{
    return {0,d};
}
```

Thread local storage

```
thread_local int a;
```

- Minden szálban más az értéke
- Mire jó?
  - Globális változók szálbiztos tárolása
  - Temp változó, nem kell újrafoglalni, itt talán lehetne _alloca

Structured Binding (C++17)

```
auto [a, b] = f();
```

- Ha f függvény olyat ad vissza, amit szét lehet szedni több változóba
  - Tömb
  - Objektum mezőkkel
  - std::Tuple

Egyéb C++17

- Parallel STL:
  - `sort(par, vec.begin(), vec.end());`
- Hex floating point numbers
  - 0x1.2p3 // 9
- Fordításidejű if
  - if constexpr (true) { } // az else ág bele se fordul
- Inicializálás if-ben
  - if (auto p = f(); !p.second) { } // 2 tagja van az if-nek

Elágazás optimalizálás (C++20)

- Melyik ág valószínű

```
if (n > 1) [[likely]]
        return 1;
    else [[unlikely]]
        return 2;
```

Fordításidejű kiértékelés (C++20)

- C++17-hez képest bővült a kör, ahol lehet használni és van consteval kulcsszó

```
consteval int sqr(int n) {
  return n*n;
}
constexpr int r = sqr(100);  // OK

int x = 100;
int r2 = sqr(x);  // Error: Call does not produce a constant
```

C++ 20 range és view

- begin() + end() helyett
- Algoritmusok, amik elfogadtak iterátort, most elfogadnak range-et és view-t
- Hasonló a C# IEnumerable interfészhez
- Tipikusan input range minden – csak olvasható
  - Van output range, ami írható. Például vector input és output is
- | operátorral lehet fűzni őket

```
std::vector<int> ints{0,1,2,3,4,5};
for (int i : ints | std::views::filter([](int i){ return i >2; }))
    std::cout << i;
```

C++ 20 range és view

- Nem feltétlen csak iterálni lehet rajta – ezt mindegyik tudja
  - Címezni is lehet valamelyiket []

	forward_list	list	deque	array	vector
std::ranges::input_range	☑	☑	☑	☑	☑
std::ranges::forward_range	☑	☑	☑	☑	☑
std::ranges::bidirectional_range		☑	☑	☑	☑
std::ranges::random_access_range			☑	☑	☑
std::ranges::contiguous_range				☑	☑

C++ 20 range és view

- view egy lusta kiértékelésű algoritmus, nem kollekció
  - Csak akkor fut le, amikor iterálunk rajta
- Hasznos kollekciók transzformációjára
  - std::views::*
    - filter – csak azokat adja vissza, amiket a predicate igazra értékel
    - reverse – megfordít
    - drop – kihagy X elemet
    - take – első X elem
    - take_while – addig veszi az elemeket, amíg a predicate igaz
    - …

C++20

- Modulok
  - Mint #include, de a fordító izolálja a fordítást, így újra felhasználható a fordítás eredménye – gyors
- Coroutine: co_await a szokásos async-await minta
- <=> operátor sorrendezéshez
  - Ha alapértelmezett implementációt használunk (=default), akkor legenerálja az összes hasonlító operátort
- template kényszerek: lényegesen többet tud, mint pl. C#
- void f(auto x); // template<class T> void f(T x)

Fordítók

- Gyakorlatilag bármilyen platform
- Minden elterjedt fordító C++17 konform
  - GCC: teljes C++17 és 99% C++20
  - Visual C++: teljes C++17 és 99% C++20
  - Clang: közel teljes C++17 és 90% C++20

- Még sok másik: http://www.stroustrup.com/compilers.html

Fejlesztőeszközök

- Platform- és fordítófüggő
- Windows: Visual Studio
  - Esetleg Visual Assist extension
- Bárhol: Eclipse, Netbeans, Qt
- Csak szerkesztők: vim, emacs, sublime, VSCode, stb.

Kérdések?