

# 5. fejezet

C++ nyelv alapjai és a standard library példákon keresztül

Grafikus Processzorok Tudományos Célú Programozása

# A C++ nyelv

Tervezési irányelvek:

([olvasnivaló](#))

- Valódi alkalmazásokhoz hasznos nyelv legyen
- Ne erőltesse a fejlesztőket egyetlen programozási stílusba
- Közvetlenül átképezhető legyen a hardverre
- Ne sérüljön implicit módon a típus rendszer
- Kompatibilis legyen a C-vel



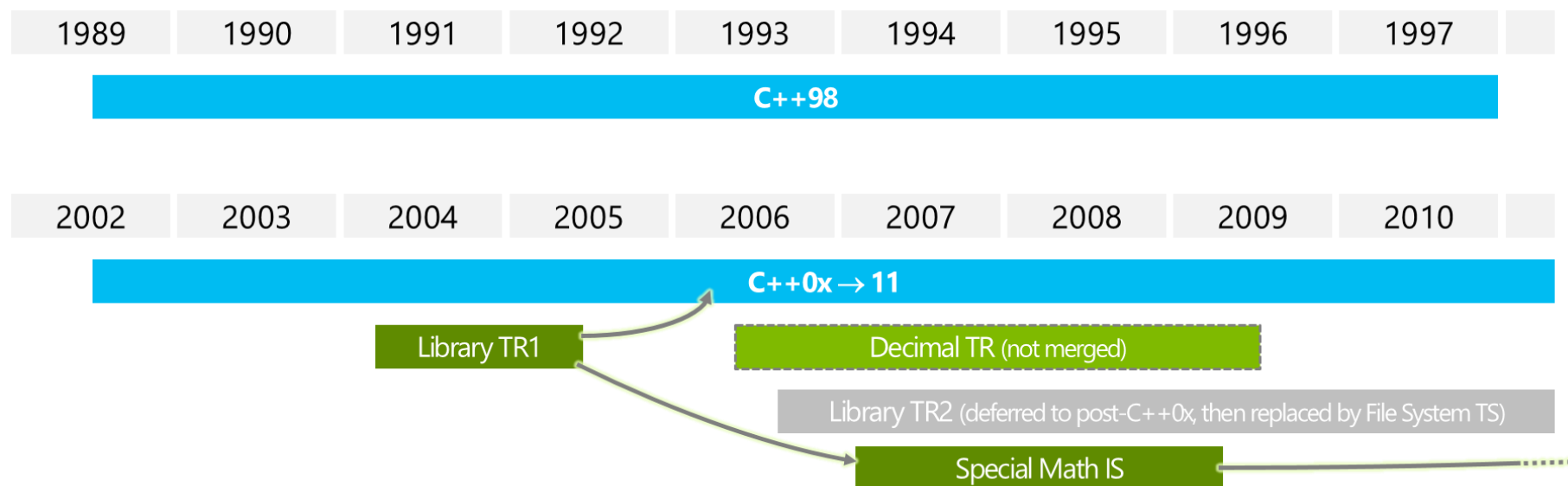
Bjarne Stroustrup  
Előadása [az ELTE-n, 2014-ben](#)  
és a [BME-n 2016-ban](#).

- **Ne fizessünk azért, amit nem használunk (zero-overhead rule)**

# A C++ nyelv

A nyelv kb. 1983-ban különült el végleg a C-től,  
Az első hivatalos könyv és fordító 1985-ben jelent meg

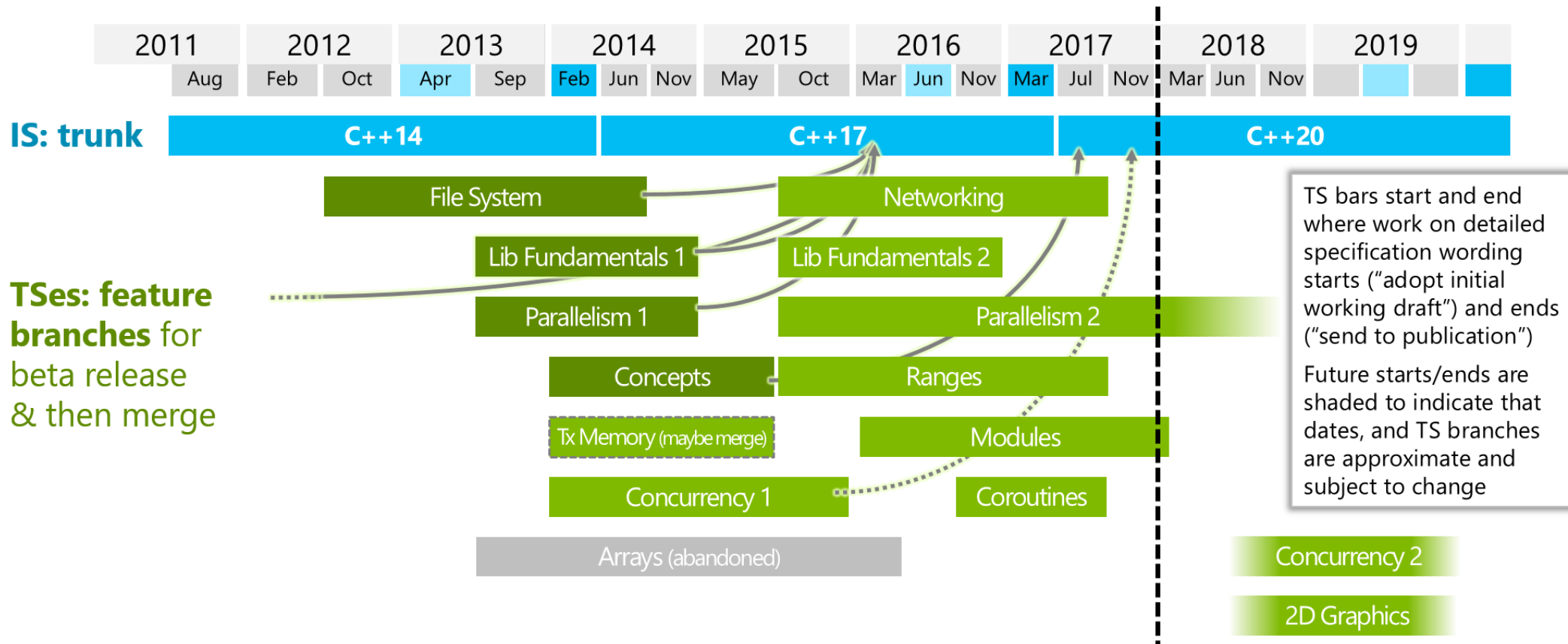
A standardizáció folyamata:



Forrás: [isocpp.org](http://isocpp.org)

# A C++ nyelv

## A standardizáció folyamata:



Forrás: [isocpp.org](http://isocpp.org)

## Programozási paradigmák:

- Imperatív
- Objektum-orientált (osztályok, öröklődés)
- Funkcionális (kifejezetten since C++11-óta: lambda függvények)
- Generikus programozás (sablonok, templatek)

# A C++ nyelv - fordítók

## Ingyenes fordítók:

- [Gnu Compiler Collection \(GCC\)](#):

Leginkább a Linux/Mac platformon használt,  
Aktuális verzió 7.3, C++17 majdnem teljesen támogatott.

- [Clang \(LLVM\)](#):

Az egyik leggyorsabban fejlődő és leginkább standard kompatibilis platform

Könnyű fejlesztői eszközöket építeni vele

Aktuális verzió: 6.0, C++17 majdnem teljesen támogatott

Kompatibilis a GCC-vel. Megy Linux-on, Mac-en, szinte tökéletesen [windows](#)-on is

- [Visual Studio \(IDE\) MSVC \(Visual C++ fordító\)](#):

A leglassabban fejlődő (még mindig nem C++14 kompatibilis teljesen),

Windows specifikus, ezért sok nem standard viselkedése van

## Fizetős fordító:

- [Intel C++ compiler](#): a legjobb teljesítmény, különösen Intel processzorokon, még csak C++11-et támogat maradéktalanul

## Online fordítók:

Hasznosak kipróbálni 1-1 új nyelvi elemet, vagy rövidebb programokat ellenőrizni

- [godbolt.org](http://godbolt.org) szinte minden létező fordítót támogat, több verzióban is, nem futtatja a programot, csak megmutatja a generált assembly kódot
- [Ideone](http://ideone.com) gcc 6.3/clang 4.0, futtatja is a programot
- [Tutorialspoint](http://tutorialspoint.com) gcc 7.1.1, teljes shell, akár több file-t is összefordít
- [coliru](http://coliru.stacked-collapsible.com) gcc 7.2, clang 5.0, web API

## Online fordítók:

Hasznosak kipróbálni 1-1 új nyelvi elemet, vagy rövidebb programokat ellenőrizni

- [Rexter](#) gcc 5.3, clang 3.8
- [cpp.sh](#) gcc 4.9.2 + boost, interaktív input
- [vc++ webcompiler](#) a legfrissebb msvc fordító verzió



## Ajánlások:

- **Használjuk a clang-ot!**

Az egyik legstandardabb fordító, ennek a legérthetőbbek a hibaüzenetei

- Mindig teszteljük le a kódot több fordítóval  
**miért? Fordító hibák, nem standard viselkedések**

Linux/Mac alatt gcc + clang, windows alatt msvc + clang

- **Használjunk egy integrált fejlesztői rendszert!**

## Integrált fejlesztői rendszerek:

Jelentősen megkönnyítik a bonyolultabb programok összerakását, fejlesztését, de minimum a kódolást jelentősen támogatják

- Szintaxis kiemelés / színezés
- Kódkiegészítés, átalakítás
- Statikus analízis/fordítási hibák, már beírásakor
- Integrált debuggolás
- Profilozás
- ...

# A C++ nyelv - fejlesztői rendszerek

Integrált fejlesztői rendszerek:

- Windows: [Visual Studio](#)  
Ipari standard fejlesztői eszköz, kis méretben még kereskedelmi célra is ingyenes a [Community](#) változat
- Platform független:
  - [Visual Studio Code](#)
  - [Kdevelop](#)
  - [Qt creator](#)
  - [Code::Blocks](#)
  - [Eclipse](#)

C++ egy bonyolult nyelv, nagyon mélyen el lehet benne merülni, és sokáig tart teljesen kitapasztalni

Ennek ellenére, számos dolgot egyszerűen is meg lehet oldani benne.

A továbbiakban példákon keresztül mutatjuk be a nyelvi elemeket és felhasználásukat  
bővebb információk mindig a hivatkozásokban érhetőek el

Ha kérdés merül fel:

1. [Google](#)
2. A google valószínűleg ide mutat: [stackoverflow.com](https://stackoverflow.com)
3. Ha nincs találat, akkor [cppreference.com](https://cppreference.com)
4. Kérdezzetek minket 😊

A kötelező hello world:

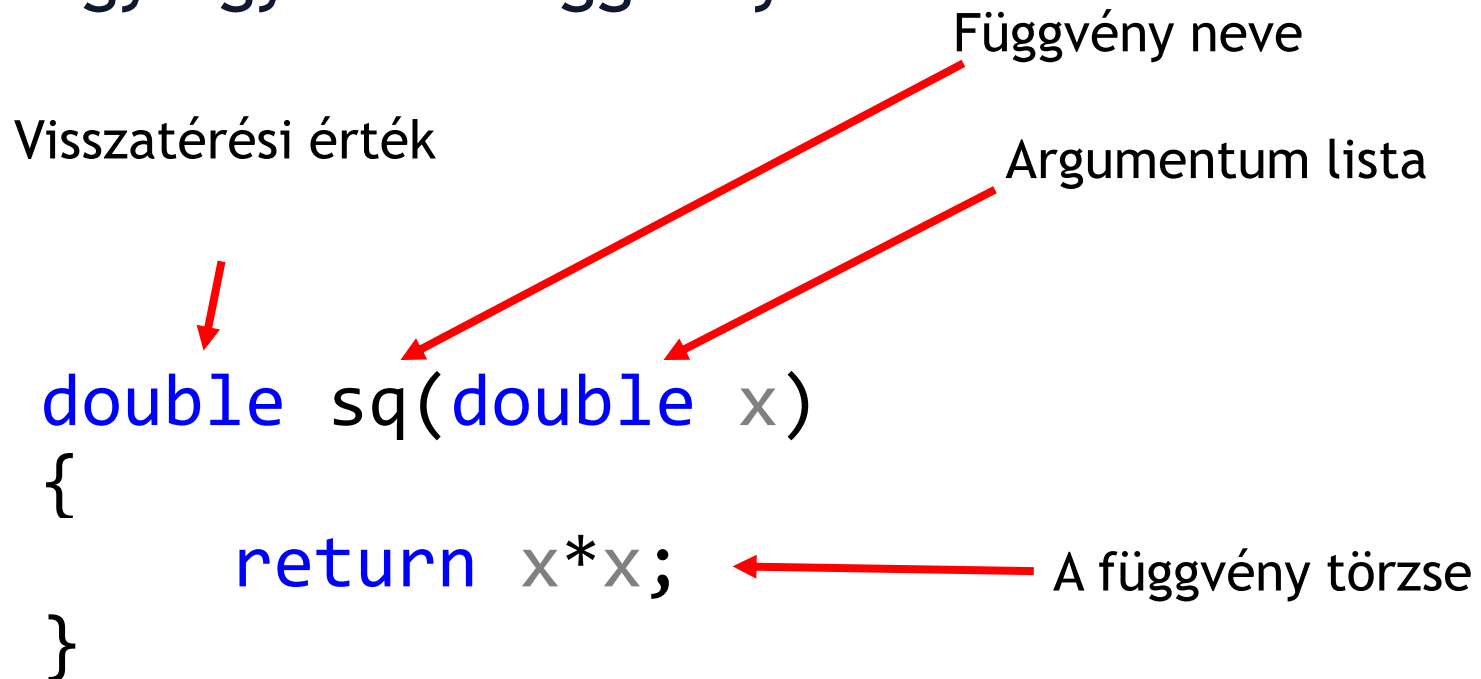
```
#include <iostream>

int main()
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

Egy egyszerű függvény:

```
double sq(double x)
{
    return x*x;
}
```

## Egy egyszerű függvény:





## Egy egyszerű függvény:

A C++11 [lehetővé tette](#), hogy a visszatérési értéket a függvény argumentum listája után adjuk meg, ha előre `auto`-t írunk

Hogy miért, arra a templatek-nél kitérünk

```
auto sq(double x) -> double
{
    return x*x;
}
```

## Egy egyszerű függvény:

A C++14-ben pedig teljesen elhagyható, mert a fordító kitalálja a `return` statementekből.

```
auto sq(double x)
{
    return x*x;
}
```

A diagram illustrating return type inference. A red arrow points from the word "return" in the code above to a rounded rectangular box containing the word "semmi". Another red arrow points from the word "return" in the text above to the same box. This indicates that the return type is inferred to be "semmi" (nothing) because the function does not have an explicit return type and only returns a value.

semmi

Lambda függvények (C++11): függvény kifejezések

```
double sq(double x)
{
    return x*x;
}
```

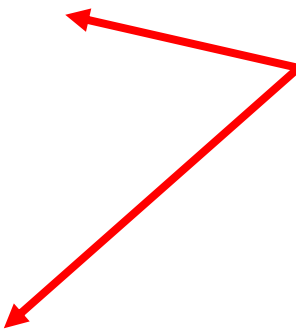
```
[] (double x) -> double { return x*x; }
```

# A C++ nyelv elemei

## Lambda függvények (C++11): függvény kifejezések

```
double sq(double x)
{
    return x*x;
}
```

A fenti függvény egy statement block, szemben az alsóval, ami egy kifejezés. Ennek ellenére, ugyan azt fejezik ki.



```
[](double x)-> double{ return x*x; }
```

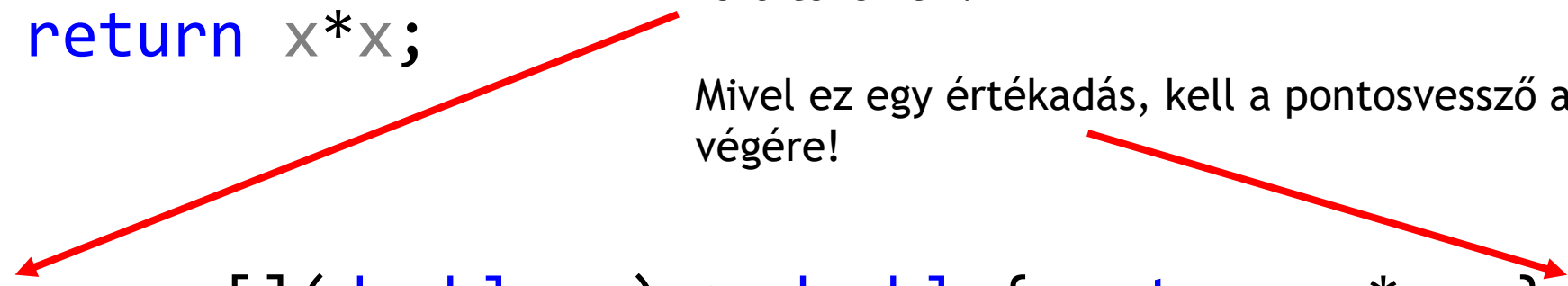
## Lambda függvények (C++11): függvény kifejezések

```
double sq(double x)
{
    return x*x;
}
```

Ha névhez akarjuk kötni, akkor csak `auto`-t használhatunk. A lambda függvény típusát csak a fordító ismeri.

Mivel ez egy értékadás, kell a pontosvessző a végére!

```
auto sq = [](double x) -> double { return x*x; };
```



## Lambda függvények (C++11): függvény kifejezések

```
auto sq(double x)
{
    return x*x;
}
```

Ugyan úgy, mint a függvényeknél, itt is ki tudja találni a visszatérési típust a fordító, nem kell kiírni:

```
auto sq = [](double x) semmi { return x*x; };
```

Generikus lambda függvények (C++14):

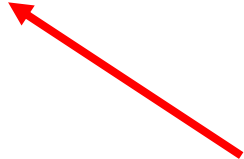
Ha ugyan az a működés kell több típusra:

```
auto sq = [](auto x){ return x*x; };
```

Generikus lambda függvények (C++14):

Ha ugyan az a működés kell több típusra:

```
auto sq = [](auto x){ return x*x; };
```



x típusa majd akkor derül ki, ha meghívjuk a függvényt, és akkor a fordító ellenőrzi, hogy a bemenő típusokra helyesek-e a törzsben leírtak



# A C++ nyelv elemei

## Függvények túlterhelése (ad-hoc polimorfizmus)

Azonos névvel bevezethetünk több függvényt, amíg az argumentumlistájukon keresztül egyértelműen elkülöníthetőek

```
double sq( double x){ return x*x; } //OK
```

```
int sq( int x){ return x*x; } //OK
```

```
double sq( int x){ return (double)x*x; } //Hiba!
```



Csak a visszatérési érték alapján nem lehet túlterhelni!  
Ez a változat nem különíthető el a 2. változattól.

## Tömbök

Ha sok egyforma elemet akarunk tárolni, akkor:

- Ha fordítási időben ismert, hogy hány elem kell, elérnek a stack-en és ez nem változik futásidőben se: [std::array](#) (C++11)
- Ha valamelyik nem teljesül: [std::vector](#)
- Utóbbihoz hasonló, kicsit több funkcionalitással: [std::valarray](#)

Műveletek tömbökön (illetve általánosságban: tárolókon):

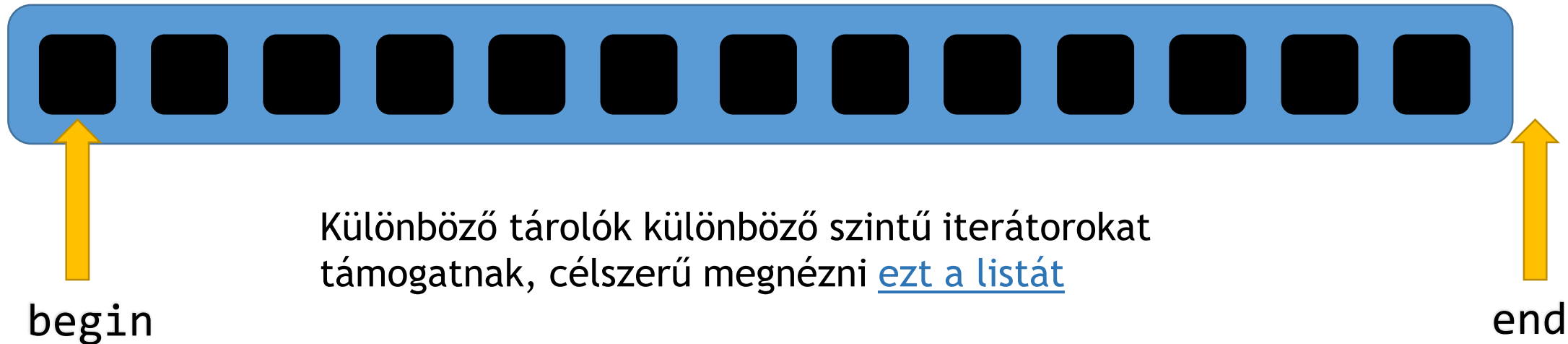
Algoritmusok!

A C++ rengeteg előre megírt algoritmussal érkezik, amik működése sok esetben testreszabható!

# A C++ nyelv elemei

<a href="#"><u>all_of, any_of, none_of (C++11)</u></a>	<a href="#"><u>copy, copy_if (C++11)</u></a>	<a href="#"><u>replace_copy, replace_copy_if</u></a>
<a href="#"><u>for_each</u></a>	<a href="#"><u>copy_n (C++11)</u></a>	<a href="#"><u>swap</u></a>
<a href="#"><u>for_each_n (C++17)</u></a>	<a href="#"><u>copy_backward</u></a>	<a href="#"><u>swap_ranges</u></a>
<a href="#"><u>count, count_if</u></a>	<a href="#"><u>move (C++11)</u></a>	<a href="#"><u>iter_swap</u></a>
<a href="#"><u>mismatch</u></a>	<a href="#"><u>move_backward (C++11)</u></a>	<a href="#"><u>reverse</u></a>
<a href="#"><u>equal</u></a>	<a href="#"><u>fill</u></a>	<a href="#"><u>reverse_copy</u></a>
<a href="#"><u>find, find_if, find_if_not (C++11)</u></a>	<a href="#"><u>fill_n</u></a>	<a href="#"><u>rotate</u></a>
<a href="#"><u>find_end</u></a>	<a href="#"><u>transform</u></a>	<a href="#"><u>rotate_copy</u></a>
<a href="#"><u>find_first_of</u></a>	<a href="#"><u>generate</u></a>	<a href="#"><u>random_shuffle, shuffle (until</u></a>
<a href="#"><u>adjacent_find</u></a>	<a href="#"><u>generate_n</u></a>	<a href="#"><u>C++17)(C++11)</u></a>
<a href="#"><u>search</u></a>	<a href="#"><u>remove, remove_if</u></a>	<a href="#"><u>sample (C++17)</u></a>
<a href="#"><u>search_n</u></a>	<a href="#"><u>remove_copy,</u></a>	<a href="#"><u>unique</u></a>
<a href="#"><u>iota (C++11)</u></a>	<a href="#"><u>remove_copy_if</u></a>	<a href="#"><u>unique_copy</u></a>
<a href="#"><u>accumulate</u></a>	<a href="#"><u>replace, replace_if</u></a>	<a href="#"><u>exclusive_scan (C++17)</u></a>
<a href="#"><u>inner_product</u></a>		<a href="#"><u>inclusive_scan (C++17)</u></a>
<a href="#"><u>adjacent_difference</u></a>		<a href="#"><u>transform_reduce (C++17)</u></a>
<a href="#"><u>partial_sum</u></a>		<a href="#"><u>transform_exclusive_scan</u></a>
<a href="#"><u>reduce (C++17)</u></a>		<a href="#"><u>(C++17)</u></a>
		<a href="#"><u>transform_inclusive_scan</u></a>
		<a href="#"><u>(C++17)</u></a>

A tömbök egyes elemeire iterátorokkal hivatkozunk. Ezek az indexek/pointererek általánosításai



Különböző tárolók különböző szintű iterátorokat támogatnak, célszerű megnézni [ezt a listát](#)

A konstrukció célja, hogy elválassza a tárolókat és az algoritmusokat egymástól! Így nem kell minden tárolóhoz minden algoritmust megírni, külön-külön.

# C++ példák - négyzetösszeg számítás

```
#include <array> // std::array
#include <numeric> // std::iota, std::accumulate
#include <iostream> // std::cout

int main()
{
    std::array<int, 5> values;

    std::iota( values.begin(), values.end(), 1);
    auto sum = std::accumulate( values.begin(),
                                values.end(),
                                0,
                                [](int a, int b){ return a+b*b; } );

    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

# C++ példák - négyzetösszeg számítás

```
#include <array>      // std::array
#include <numeric>    // std::iota, std::accumulate
#include <iostream>   // std::cout

int main()
{
    std::array<int, 5> values;

    std::iota( values.begin(), values.end(), 1);
    auto sum = std::accumulate( values.begin(),
                                values.end(),
                                0,
                                [](int a, int b){ return a+b*b; } );

    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

5 db `int`-et (előjeles egészt) tároló tömb a stack-en

# C++ példák - négyzetösszeg számítás

```
#include <array> // std::array
#include <numeric> // std::iota, std::accumulate
#include <iostream> // std::cout

int main()
{
    std::array<int, 5> values;

    std::iota( values.begin(), values.end(), 1);
    auto sum = std::accumulate( values.begin(),
                                values.end(),
                                0,
                                [](int a, int b){ return a+b*b; } );

    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

std::iota: feltölt a megadott értéktől egyesével növekvő sorozattal egy tárolót.  
Jelen esetben: 1, 2, 3, 4, 5



# C++ példák - négyzetösszeg számítás


```
#include <array> // std::array
#include <numeric> // std::iota, std::accumulate
#include <iostream> // std::cout

int main()
{
    std::array<int, 5> values;

    std::iota( values.begin(), values.end(), 1);
    auto sum = std::accumulate( values.begin(),
                                values.end(),
                                0,
                                [](int a, int b){ return a+b*b; } );

    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

A begin, end a tömb elejét és utolsó utáni elemét jelenti, 1-el kezdődik a feltöltés.



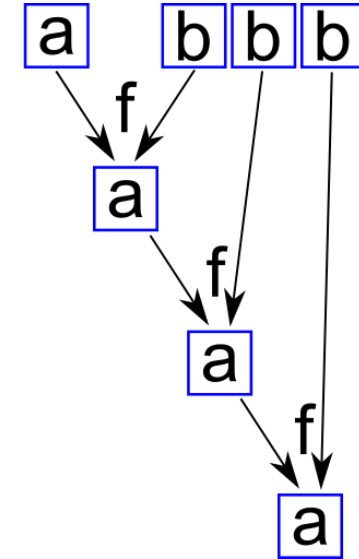
# C++ példák - négyzetösszeg számítás

```
#include <array> // std::array
#include <numeric> // std::iota, std::accumulate
#include <iostream> // std::cout

int main()
{
    std::array<int, 5> values;

    std::iota( values.begin(), values.end(), 1);
    auto sum = std::accumulate( values.begin(),
                                values.end(),
                                0,
                                [](int a, int b){ return a+b*b; } );

    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```



Az `std::accumulate` valamilyen összegzést, redukciót hajt végre elemeken.

Itt most 0-tól kezdve négyzetösszeget számol.

Tipikusan C-ben a négyzetösszeget, valahogy így írná az ember:

```
int arr[5] = {1, 2, 3, 4, 5};  
int sqsum = 0;  
for(int i=0; i<4; i++)  
{  
    sqsum += arr[i]*arr[i];  
}
```

Azonban sok a hibalehetőség!

Tipikusan C-ben a négyzetösszeget, valahogy így írná az ember:

```
int arr[5] = {1, 2, 3, 4, 5};  
int sqsum = 0;  
for(int i=0; i<4; i++)  
{  
    sqsum += arr[i]*arr[i];  
}
```

Jó a ciklus változó típusa?  
Jó értékről indul a ciklus?  
És a vége is jó? Meg a léptetés is?

Mindenütt jó indexet használtunk a jó tömbhöz?  
Hova kapjuk az eredményt?  
Minden jól lett inicializálva?

Az előzőeken kicsit segít az új ciklus konstrukció C++11 óta:  
[range-based for loop](#):

```
int arr[5] = {1, 2, 3, 4, 5};  
int sqsum = 0;  
for(auto const& x : arr)  
{  
    sqsum += x*x;  
}
```

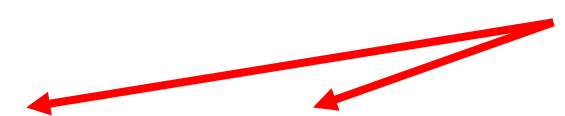
Ez a konstrukció a beépített tömbökkel és minden olyan típusal is működik, aminek van `begin()` és `end()` függvénye.

C++17-ben további fejlesztések érkeznek ide is...

A függvényeknél a legegyszerűbb paraméter átadás, ha nem írunk ki semmi extrát, csak a típust, ilyenkor lokális másolat jön létre az argumentumokból:

```
auto addSq(double x, double y){ ... }
```

Lokális másolatok!

Two red arrows originate from the text "Lokális másolatok!". One arrow points to the first "double" type in the function signature, and the other points to the second "double" type.

Ha valami nagy adatunk van, nem szeretnénk lemásolni...

Több lehetőségünk is van:

- |   |   |
|---|---|
| <code>auto f(BigObject* x)</code>                       | Átadás "csupasz" hivatkozással <b>Nem ajánlott!</b> |
| <code>auto f(BigObject&amp; x)</code>                   | Átadás referenciával <i>ajánlott!</i>               |
| <code>auto f(std::shared_ptr&lt;BigObject&gt; x)</code> | "Okos" pointer (C++11) (van némi költsége)          |

A referencia igazából egy mutató, pointer, de szemantikailag sokkal megszorítottabb nem lehet null, vagy inicializálatlan, nem lehet aritmetikát végezni rajta, vagy módosítani, stb.

Constness, megváltoztathatlanság:

Ha egy változót csak olvasni akarunk, de módosítani nem, akkor jelöljük meg `const`-al

```
auto f(const BigObject& x)
```

Nem módosítható referencia *ajánlott!*

```
auto f(BigObject * x)
```

Módosítható az objektum is és a pointer is

```
auto f(BigObject const* x)
```

Nem módosítható az objektum, de a pointer igen

```
auto f(BigObject * const x)
```

Módosítható az objektum, de a pointer nem

```
auto f(BigObject const* const x)
```

Egyik sem módosítható



## Érték kategóriák: (C++11 óta)

- L-érték: bármilyen érték, ami névhez van kötve, értékül lehet adni neki (baloldalán állhat az =-jelnek), van címe a memóriában
- Tiszta r-érték: átmeneti, még ***nem névhez kötött!***  
Pl.: egy leírt érték a kódban, vagy egy visszatérési érték függvény hívásból  
Nincs címe a memóriában, csak az = jel jobboldalán állhat
- X-érték: megszűnő (eXpiring) érték, ***nincs névhez kötve!***  
Nincs címe a memóriában, csak az = jel jobboldalán állhat  
Pl.: függvények, amik r-érték referenciát adnak vissza, static cast (explicit átalakítás) r-érték referenciává, x-érték objektumok elemei
- Gl-value: L-value + X-value → a `const&`-t váró függvényeket preferálják
- R-value: X-value + Pure R-value → inkább a `&&`-t váró függvényeket preferálják a `const&` helyett:

A C++11 bevezetett egy újfajta referenciát, az r-érték referenciát.

`auto f(BigObject&& x)`

Ez azt fejezi ki, hogy a függvény átveszi a tulajdonjogot az objektum felett, a hívó oldalon már nem használhatja / tulajdonolhatja senki.  
ez általában csak egy pointer cserét jelent

Hívó oldal:

`BigObject bo = ...;`

`auto result1 = f( std::move(bo) );`

`auto result2 = f( makeBigObject(...) );`

Itt expliciten átadjuk a tulajdonjogot a függvénynek

Ugyan ez történik, ha az objektum egyik fv hívásból a másikba kerül át

Ökölszabályok, hogy mikor milyen átadást érdemes használni:

- `auto f(Object x)` Ha az objektum kicsi (mondjuk  $\leq$  regiszter méret), vagy kifejezetten másolat kell
- `auto f(Object* x)` Soha  
kivéve ha könyvtár fejlesztők vagyunk, és tisztában vagyunk azzal, mit csinálunk
- `auto f(std::shared_ptr<Object> x)` Ha meg akarunk osztani egy objektumot a program többi részével, ami esetleg null is lehet
- `auto f(BigObject & x)` Ha mindenképp módosítani akarjuk a külső példányt
- `auto f(BigObject const& x)` Ha nem akarjuk módosítani a külső példányt
- `auto f(BigObject && x)` Ha át akarjuk venni a tulajdonjogot a megkapott példányra

Névterek: a típusok, függvények csoportosításának eszközei

```
namespace math
{
    int f(int x);
}
```

```
int g(int x)
{
    using namespace math;
    return f(x);
}
```

```
int g(int x)
{
    return math::f(x);
}
```

# A C++ nyelv elemei

Névterek: a típusok, függvények csoportosításának eszközei

```
namespace math
```

```
{  
    int f(int x);  
}
```

Új névtér bevezetése / folytatása  
elemei a {}-ben levő azonosítók, most pl. f

```
int g(int x)  
{  
    using namespace math;  
    return f(x);  
}
```

```
int g(int x)  
{  
    return math::f(x);  
}
```

Névterek: a típusok, függvények csoportosításának eszközei

```
namespace math
```

```
{  
    int f(int x);  
}
```

Egy másik helyen úgy használhatjuk, hogy vagy a teljes { } blokkba `using`-al láthatóvá tesszük

```
int g(int x)  
{  
    using namespace math;  
    return f(x);  
}
```

```
int g(int x)  
{  
    return math::f(x);  
}
```

Névterek: a típusok, függvények csoportosításának eszközei

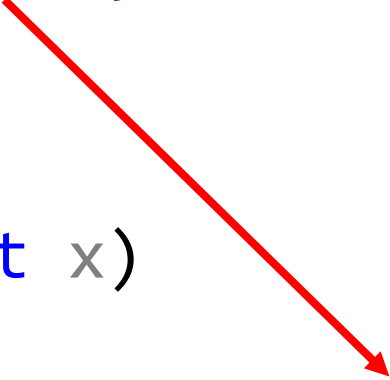
```
namespace math
```

```
{  
    int f(int x);  
}
```

```
int g(int x)  
{  
    using namespace math;  
    return f(x);  
}
```

Vagy expliciten kiírjuk

```
int g(int x)  
{  
    return math::f(x);  
}
```



Egymásba ágyazott névtérekét egyszerűbbekkel helyettesíthetünk:

```
namespace fs = std::experimental::filesystem;
```



## Struktúrák, az objektum orientált programozási építőkövei:

Összemosnak pár dolgot

- Új típust vezetnek be
- Változók és az őket gyakran használó függvények csoportja
- Kombinálhatóság öröklésen keresztül  
([subtyping](#) polimorfizmus, akár dinamikusan is: virtuális függvények)
- Inicializálási és felszabadítási logika  
([RAII](#), [konstruktorok](#), [destruktorok](#))
- Névtér

Az osztályok (classok) erre csupán [hozzáférési korlátozásokat](#) raknak rá.

Objektum:

Egy új típus adattagokkal, tagfüggvényekkel, tagtípusokkal...

```
struct Circle
{
    double r;

    double area(){ return r*r*3.1415; }
};
```

# A C++ nyelv elemei

Objektum:

Egy új típus adattagokkal, tagfüggvényekkel, tagtípusokkal...

kulcsszó: [struct or class](#)

```
struct Circle  
{  
    double r;  
    double area(){ return r*r*3.1415; }  
};
```

Az új típus neve

Adattagok deklarációi

Tagfüggvények

Ennek az objektumnak egy példánya a *stack*-en létrehozva az új C++11-es [inicializálással](#):

```
Circle c{2.0};
```

A tagváltozók, tagfüggvények ekkor "."-al érhetőek el:

```
auto R = c.r;
```

```
auto A = c.area();
```

Ugyan ez a típus a *heap* -en létrehozva:

```
#include <memory>
std::shared_ptr<Circle> c =
    std::make_shared<Circle>(Circle{2.});
```

Ekkor a tagok így érhetőek el:

```
auto R = c->r;
auto A = c->area();
```

Bővebben lásd:

[std::shared\\_ptr](#) (C++11)  
[std::make\\_shared](#) (C++11)

Ne használjunk csupasz `new`-t, mert könnyű elrontani!

## Konstruktorok, Destruktorok:

Speciális függvények, melyek az objektumok létrehozását és felszabadítását kezelik le:

- Konstruktorok: új objektum létrehozása
  - Alapértelmezett (default) konstruktor (nincs argumentuma)
  - Másoló (copy) konstruktor
  - Move konstruktor
- Destruktor: az objektum megsemmisítése (nincs argumentuma)

A konstruktorok és destruktorkok fontos sarokpontjai a nyelvnek

A nyelv erős garanciákat ad ezeknek a jól definiált sorrendben törénő mindenkori végrehajtására

Erre épül a [RAII](#) technika, amit az STL számos helyen használ:

- Dinamikus memória, file kezelés, okos pointerok, mutex lockok (l. később)

Ezért nem kell pl. [garbage collection](#) alapú memória kezelés, ami hozzájárul a nyelv gyorsaságához...

## Konstruktorok és destruktorkok:

Amíg csak primitív típusokat és könyvtárakból származó objektumokat használunk, addig általában nem kell ezekkel foglalkozni, a fordító legenerálja a megfelelő függvényeket:

```
struct Circle
```

```
{  
    std::array<double, 2> position;  
    double r;  
};
```

```
Circle c{ {{2., 5.}}, 2.5 };  
Circle c2; c2 = c;
```

Továbbra is [list initialization](#) (C++11)



Konstruktorok és destruktorkok:

Ha valami összetettebb működés kell, pl. ha az objektum birtoklást fejez ki (pl. egy memóriára) akkor a [három/öt/nulla](#) szabályt érdemes ismerni.

```
struct Vector
{
    size_t n;
    float* data;
};
```

Konstruktorok és destruktorkok:

A legáltalánosabb esetben ezeket kellhet megírni:

```
struct Vector
```

```
{
```

```
    Vector()
```

Default konstruktor

```
    Vector( Vector const& cpy );
```

Copy konstruktor

```
    Vector( Vector && mv );
```

Move konstruktor (C++11)

```
    Vector& operator=( Vector const& cpy );
```

Értékadás másolással

```
    Vector& operator=( Vector && mv );
```

Értékadás mozgatással  
(C++11)

```
    ~Vector();
```

Destruktor

```
}
```

Új típus nevek bevezetése már meglévő típusokra:

```
typedef Vector Vec;
```

Szinonima, nem lehet generikus

```
using Vec = Vector;
```

Szinonima, C++11 óta  
ez lehet generikus is (l. később)

# A C++ nyelv elemei

Egy beágyazott típust `::`-al lehet elérni:

```
struct Vector
{
    size_t n;
    float* data;
    using elem_type = float;
};
```

```
Vector::elem_type x = 1.4142f; //equivalent to:
                               //float x = 1.4142;
```

# Vektor példa

Elemezzük, hogy hogyan épül fel egy 2 elemű vektor osztály:

```
struct Vector2
{
    std::array<float, 2> elems;
    float& operator[](int i) { return elems[i]; }
    const float& operator[](int i) const { return elems[i]; }

    Vector2& operator+=( const Vector2 & v )
    {
        elems[0] += v.elems[0]; elems[1] += v.elems[1];
        return *this;
    }
};
```


# Vektor példa

Elemezzük, hogy hogyan épül fel egy 2 elemű vektor osztály:

Fordítási időben ismert számú elem, tehát `std::array`

```
struct Vector2
{
    std::array<float, 2> elems;
    float& operator[](int i) { return elems[i]; }
    const float& operator[](int i) const { return elems[i]; }

    Vector2& operator+=( const Vector2& v )
    {
        elems[0] += v.elems[0]; elems[1] += v.elems[1];
        return *this;
    }
};
```

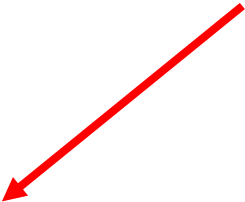


Elemezzük, hogy hogyan épül fel egy 2 elemű vektor osztály:

```
struct Vector2
```

```
{  
    std::array<float, 2> elems;  
    float& operator[](int i) { return elems[i]; }  
    const float& operator[](int i) const { return elems[i]; }  
  
    Vector2& operator+=( const Vector2& v )  
    {  
        elems[0] += v.elems[0]; elems[1] += v.elems[1];  
        return *this;  
    }  
};
```

Hozzáférés az elemekhez: a [] operátor  
(kötelezően csak 1 intet várhat)



Elemezzük, hogy hogyan épül fel egy 2 elemű vektor osztály:

```
struct Vector2
```

```
{  
    std::array<float, 2> elems;  
    float& operator[](int i) { return elems[i]; }  
    const float& operator[](int i) const { return elems[i]; }  
  
    Vector2& operator+=( const Vector2& v )  
    {  
        elems[0] += v.elems[0]; elems[1] += v.elems[1];  
        return *this;  
    }  
};
```

Általában két változat kell belőle:  
az elsőn keresztül írni lehet a kiválasztott  
elemet, a másodikon keresztül csak olvasni

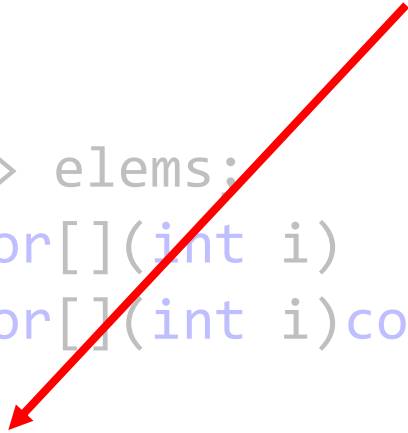


Elemezzük, hogy hogyan épül fel egy 2 elemű vektor osztály:

```
struct Vector2
```

```
{  
    std::array<float, 2> elems;  
    float& operator[](int i) { return elems[i]; }  
    const float& operator[](int i) const { return elems[i]; }  
  
    Vector2& operator+=( const Vector2& v )  
    {  
        elems[0] += v.elems[0]; elems[1] += v.elems[1];  
        return *this;  
    }  
};
```

Hozzáadás operátor: referenciát ad vissza,  
hogy ilyet is írassunk: (u += v1) += v2



Elemezzük, hogy hogyan épül fel egy 2 elemű vektor osztály:

```
struct Vector2
```

```
{
```

```
...
```

```
Vector2& operator*=( const float& c )
```

```
{
```

```
    elems[0] *= c; elems[1] *= c;
```

```
    return *this;
```

```
}
```

```
};
```

Hasonlóan megírhatjuk a szorzás műveletet is



Elemezzük, hogy hogyan épül fel egy 2 elemű vektor osztály:

```
struct Vector2
```

```
{
```

```
    ...
```

```
    float length() const
```

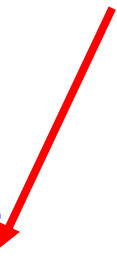
```
{
```

```
    return std::sqrt(elems[0]*elems[0]+  
                     elems[1]*elems[1]);
```

```
}
```

```
};
```

Ha olyan tagfüggvényt írunk, ami nem változtat meg semmit, csak olvas, akkor az mindig legyen `const`!



Néhány hasznos külső függvény:

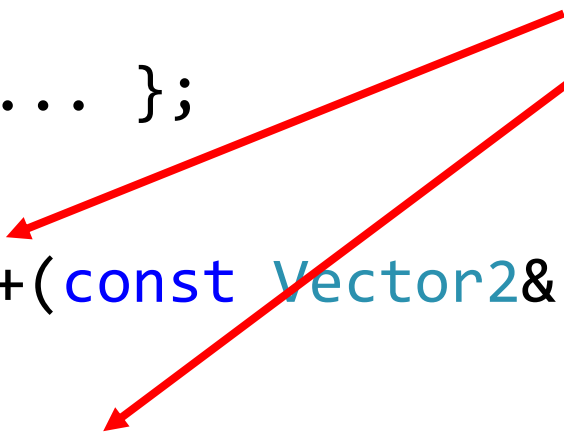
```
struct Vector2{ ... };
```

```
Vector2 operator+(const Vector2& v1, const Vector2& v2)  
{  
    return Vector2{ { { v1[0]+v2[0], v1[1]+v2[1] } } };}
```

Összeadás operátor:

Ez értéket ad vissza!

Az inicializáláskor a belső `std::array`-ban levő nyelvi tömböt inicializáljuk, ezért kell tripla `{{{}}}`



Néhány hasznos külső függvény:

```
struct Vector2{ ... };
```

Skalárral szorzás operátor a két oldalról:  
Mind két lehetőséget külön be kell vezetni!

```
Vector2 operator*(const Vector2& v, const float& c) ←  
{  
    return Vector2{ { { v[0]*c, v[1]*c } } };}
```

```
Vector2 operator*(const float& c, const Vector2& v,)  
{  
    return Vector2{ { { c*v[0], c*v[1] } } };}
```

Megjegyzés a C++ beépített matematikai funkcióiról:

Ami van:

- Aritmetika a beépített típusokon ([int](#), [float](#), [double](#), ...)
- Néhány gyakori és speciális függvény  
(a teljes lista [itt](#), C++11-ben bővült, és C++17-ben [tovább bővül majd](#))
- Van [complex](#) adattípus, aritmetikával és spec függvényekkel (ezt is kiterjesztették C++11-ben)
- Fordítási idejű [racionális tört aritmetika](#) (C++11-óta)
- [Véletlenszám-generátorok](#) (C++11-óta)  
Egy fizikus tervezte☺: több különböző generátor, amik számos gyakori eloszlás mintavételezésére használhatóak.

Megjegyzés a C++ beépített matematikai funkcióiról:

Ami nincs:

- Normális vektor/mátrix/több dimenziós tömb/tenzor struktúrák

Ilyesmiket a boost-ban ([math](#) alkönyvtárak, [multi-array](#)), vagy más könyvtárakban érdemes keresni.

- Lineáris algebra rutinok

Ezekre is külső könyvtárak, pl.: [eigen](#), [armadillo](#)

A C++ egyik legfontosabb nyelvi szolgáltatása a templatek (sablonok)



A templatek sablonok, bennük helyőrzőkkel azaz kitöltendő paraméterekkel

Amikor használjuk ezeket, meg kell adnunk, hogy mivel szeretnénk kitölteni a helyőrzőket.  
Ez a lépés a példányosítás (*instantiation*)

Miután ez megtörtént, a keletkező kód átmegy a típusellenőrzésen és fordításon pont úgy, mintha kézzel írta volna a fejlesztő.

Ez egy osztály sablon (class template):  
(annak ellenére, hogy ez struct, de kb. ugyan azok)

```
template<typename T>
struct Circle
{
    T r;
    T area() const { return r*r*3.14; }
};
```

Ez egy osztály sablon (class template):  
(annak ellenére, hogy ez struct, de kb. ugyan azok)

```
template<typename T>
struct Circle
{
    T r;
    T area() const { return r*r*3.14; }
};
```

Két példányosítása pedig:

```
Circle<float> c1;
Circle<double> c2;
```

# C++ templatek

Ez egy osztály sablon (class template):  
(annak ellenére, hogy ez struct, de kb. ugyan azok)

```
template<typename T>
struct Circle
{
    T r;
    T area() const
    {
        return r*r*3.14;
    }
};
```

```
struct Circle
{
    float r;
    float area() const
    {
        return r*r*3.14;
    }
};
```

```
struct Circle
{
    double r;
    double area() const
    {
        return r*r*3.14;
    }
};
```

Két példányosítása pedig:

```
Circle<float> c1;
Circle<double> c2;
```

Ezek teljesen ekvivalens kódot eredményeznek  
ezekkel a fenti "kézzel írt" változatokkal

Függvények is lehetnek templatek:

```
template<typename T>  
T sq(T const& x){ return x*x; }
```

Amelyek használatánál nem kell kiírni a template paramétereket, ha egyértelműen kitalálhatóak ([template argument deduction](#)):

```
double sq10 = sq(10.0);
```

A függvény templatek és a referenciák együtt egy további hasznos elemet képeznek:

Sok esetben kellene függvény kódot duplikálni csak azért, mert `const&` és `&&` esetben is működnie kellene.

A legegyszerűbb példa a perfect forwarding probléma:


Hogyan írjunk olyan függvényt, ami megtartja az argumentum érték kategóriáját, amikor továbbadja?

```
template<typename T>
auto foo( T ??? t )
{
    return bar( t ??? );
}
```

Hogyan írjunk olyan függvényt, ami megtartja az argumentum érték kategóriáját?

```
template<typename T>
auto foo( T&& t )
{
    return bar( std::forward<T>(t) );
}
```

A bar függvény most úgy kapja meg t-t, ahogy foo:  
Ha `const&` volt, akkor `bar(const&)`-et hívja,  
Ha `&&` volt, akkor `bar(T&&)`-t.  
Bővebben: [itt](#)



C++14-óta változók is lehetnek templatek:

```
template<typename T>  
const T pi = T(3.1415926535897932385);
```

Itt meg kell adni a paramétert:

```
double sqpi = sq( pi<double> );
```



# C++ templatek

A templateknek nem csak típus paraméterei lehetnek, hanem nem-típus paraméterek is: pl.: egész számok, amire példa az `std::array`:

```
template<typename T, int size> struct array  
{  
    T data[size];  
};
```

```
array<float, 4> A;
```

## A templatek specializálhatóak is

(újabb ad-hoc polimorfizmus, de ez most fordítás idejű!)

### Alap (base) template

```
template<typename T>
struct Circle
{
    T r;
    T area() const
    { return r*r*3.14; }
};
```

### Specializáció `int`-re:

```
template<>
struct Circle<int>
{
    int r;
    double area() const
    { return (r*r)*3.14; }
};
```

Példa a template specializáció alkalmazására:

```
template<bool B, typename T = void>  
struct enable_if {};
```

```
template<class T>  
struct enable_if<true, T> { using type = T; };
```



Ennek a struktúrának csak akkor van belső `type` típusa, ha az első paraméter `true`.

SFINAE

SFINAE - A történelem legrosszabb betűszava...

SFINAE - Substitution Failure Is Not An Error

## SFINAE - [Substitution Failure Is Not An Error](#)

Amikor template példányosítás történik, a lehetséges eseteket végig nézi a fordító, hogy melyik illik a megadott paraméterekre.

Ha bizonyos behelyettesítések érvénytelen kódot eredményeznének, akkor nem fordítási hiba lesz, csak csöndben eldobódik az adott eset.

## Példa a template specializáció alkalmazására:

```
template <typename T,  
         typename Checker = void>  
struct Matrix  
{...};
```

```
template <typename T>  
struct Matrix<T, typename enable_if<is_noncommutative<T>::value>::type>  
{...};
```




## Példa a template specializáció alkalmazására:

```
template <typename T,  
         typename Checker = void>  
struct Matrix  
{...};
```

`is_noncommutative` lehetne egy ugyan olyan template, mint az `enable_if`, de ez egy `bool` value tagváltozót tartalmaz, ha a `T` argumentum egy nem-kommutatív algebrát jelöl. A `Matrix` ekkor specializálható erre az esetre, de továbbra is csak az első paramétert kell megadni.

```
template <typename T>  
struct Matrix<T, typename enable_if<is_noncommutative<T>::value>::type>  
{...};
```



Kifejezés (expression) SFINAE (C++11)

```
template<typename T> struct Circle
{
    T r;
    T area() const { return r*r*3.14; }
};
```

```
struct SomeShape
{
    double area;
};
```

```
template<typename T> struct Circle  
{  
    T r;  
    T area() const { return r*r*3.14; }  
};
```

Itt area egy függvény!

```
struct SomeShape  
{  
    double area;  
};
```

Itt area egy változó!

```
template<typename S>  
auto getArea( S shape )->decltype( shape.area() )  
{  
    return shape.area();  
}
```

```
template<typename S>  
auto getArea( S shape )->decltype( +shape.area )  
{  
    return shape.area;  
}
```

```
template<typename S>  
auto getArea( S shape )->decltype( shape.area() )  
{  
    return shape.area();  
}
```

`decltype` egy C++11-ben bevezetett operátor, ami meghatározza a zárójelben álló kifejezés típusát.

```
template<typename S>  
auto getArea( S shape )->decltype( +shape.area )  
{  
    return shape.area;  
}
```


```
template<typename S>
auto getArea( S shape )->decltype( shape.area() )
{
    return shape.area();
}
```

Ez csak függvényekre érvényes kifejezés



```
template<typename S>
auto getArea( S shape )->decltype( +shape.area )
{
    return shape.area;
}
```

Ez csak változókra érvényes kifejezés



```
template<typename S>
auto getArea( S shape )->decltype( shape.area() )
{
    return shape.area();
}
```

getArea(Circle<float>{1.0})  
ezt hívja, mert Circle-ben area() függvény van

```
template<typename S>
auto getArea( S shape )->decltype( +shape.area )
{
    return shape.area;
}
```

getArea(SomeShape{1.0})  
ezt hívja, mert SomeShape area egy változó



Ez (expression) SFINAE segítségével általános függvényeket/struktúrákat specializálhatunk az alapján, hogy adott műveletek értelmesek-e a típus paramétereken

## Paramétercsomagok

## Paramétercsomagok

Számos esetben szeretnénk általánosítani műveleteket tetszőleges számú argumentumra.

Pl.: hogyan írjunk olyan függvényt, ami összeadja az összes argumentumát?

```
auto x = sum(1, 2, 3, 4);
```

C++11 óta lehetséges template paraméter csomagokat bevezetni:

```
template<typename... Ts> auto func(Ts... ts);
```

C++11 óta lehetséges template paraméter csomagokat bevezetni:

```
template<typename... Ts> auto func(Ts... ts);
```

Itt a ... miatt nem egyetlen paramétert vezetünk be, hanem **Ts** lehet 0, 1, 2, 3, ... darab típus

És ennek megfelelően **ts** lehet 0, 1, 2, 3, ... darab argumentum

Pl.: hogyan írjunk olyan függvényt, ami összeadja az összes argumentumát?

```
template<typename T>  
auto sum(T x){ return x; }
```

```
template<typename T1,  
        typename T2, typename... Ts>  
auto sum(T1 x1, T2 x2, Ts... xs)  
{  
    return sum(x1+x2, xs...);  
}
```

Pl.: hogyan írjunk olyan függvényt, ami összeadja az összes argumentumát?

```
template<typename T>
```

```
auto sum(T x){ return x; }
```



Ha egyetlen elemet akarunk összegezni,  
akkor az maga az elem

```
template<typename T1,  
         typename T2, typename... Ts>
```

```
auto sum(T1 x1, T2 x2, Ts... xs)
```

```
{
```

```
    return sum(x1+x2, xs...);
```

```
}
```

Pl.: hogyan írjunk olyan függvényt, ami összeadja az összes argumentumát?

```
template<typename T>  
auto sum(T x){ return x; }
```

Ha legalább 2 elemünk van, akkor azt a kettőt összeadhatjuk, majd az eredményt és a maradékot újra szummázzuk

```
template<typename T1,  
        typename T2, typename... Ts>  
auto sum(T1 x1, T2 x2, Ts... xs)  
{  
    return sum(x1+x2, xs...);  
}
```

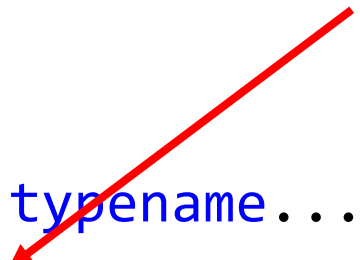


Pl.: hogyan írjunk olyan függvényt, ami összeadja az összes argumentumát?

```
template<typename T>  
auto sum(T x){ return x; }
```

Ez a szétválasztása az argumentumoknak kijelölt és csomagban levőkre csak akkor működik, ha a paraméter csomag az utolsó argumentuma a függvénynek!!!

```
template<typename T1,  
         typename T2, typename... Ts>  
auto sum(T1 x1, T2 x2, Ts... xs)  
{  
    return sum(x1+x2, xs...);  
}
```



C++17-ben külön nyelvi elem lesz tetszőleges beépített operátorral való redukcióra. Az előző feladat akkor csupán ennyi lesz:

```
template<typename... Ts>
auto sum(Ts... xs)
{
    return (xs + ...);
}
```

A paraméter csomagokat osztály sablonokban is használhatjuk.

Típuspélda az [std::tuple](#) (C++11), amely akárhány, akármilyen típust képes tárolni:

```
template<typename... Ts>  
struct tuple{ ... };
```

A függvények csak 1 típust tudnak visszaadni, de az lehet összetett is, pl. egy [std::tuple](#):

```
std::tuple<int, int, double> f(int a, int b)
{
    return std::make_tuple(a, b, (double)a/(double)b);
}
```

Később egy [std::tie](#) segítségével értékül adhatjuk az eredményeket balérték referenciáknak...

A függvények csak 1 típust tudnak visszaadni, de az lehet összetett is, pl. egy [std::tuple](#):

```
std::tuple<int, int, double> f(int a, int b)
{
    return std::make_tuple(a, b, (double)a/(double)b);
}
```

Később egy [std::tie](#) segítségével értékül adhatjuk az eredményeket balérték referenciáknak:

```
int a, b;
double ratio;
std::tie(a, b, ratio) = f(14, 7);
```

C++17-ben már inicializációs listát is használhatunk ilyen esetekben:

```
std::tuple<int, int, double> f(int a, int b)
{
    return { a, b, (double)a/(double)b };
}
```

A template-k bevezetése számos esetben nagyon hosszadalmas. Azonban érdemes tudni, hogy a generikus lambda függvények (C++14) álruhás osztály template-k.

Ez a lambda: `[](auto x){ return x*x; }`

Ezzel a struktúra példánnyal ekvivalens:

```
struct Unnamed
{
    template<typename T>
    auto operator()(T x) const { return x*x; }
} unnamed;
```

A capture clause pedig igazából a struktúra tagjait listázza:

```
int x = 42; int y = 137;  
[x, p = &y](auto z){ return x*z + (*p); }; //C++14
```

Ekvivalens struct:

```
struct Unnamed  
{  
    int x; int* p;  
  
    template<typename T>  
    auto operator()(T z) const{ return x*z + (*p); }  
} unnamed{x, &y};
```



Egy általános n elemű vektor osztály kb. így festhet:

```
template<typename T, int n>
struct Vector
{
    std::array<T, n> elems;
    T& operator[](int i) { return elems[i]; }
    const T& operator[](int i) const { return elems[i]; }

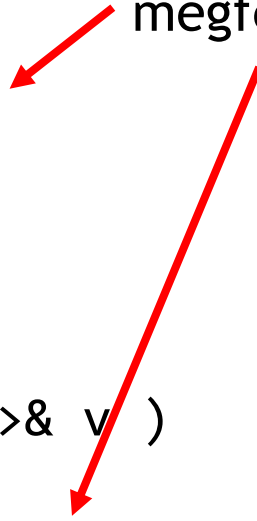
    Vector<T, n>& operator*=( const T& c );
    Vector<T, n>& operator+=( const Vector<T, n>& v );
};
```

↑

És ezeket az operátorokat hogyan implementáljuk?

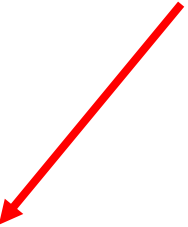
```
template<typename T, int n> struct Vector {  
    ...  
    Vector<T, n>& operator*=( const T& c )  
    {  
        for(int i=0; i<n; ++i){ elems[i] *= c; }  
        return *this;  
    }  
  
    Vector<T, n>& operator+=( const Vector<T, n>& v )  
    {  
        for(int i=0; i<n; ++i){ elems[i] += v[i]; }  
        return *this;  
    }  
};
```

Sokan írnának egy for-ciklust,  
majd copy-pastelnék,  
átírogatnák mindenhol a  
megfelelő műveletre...



```
template<typename T, int n> struct Vector {  
    ...  
    Vector<T, n>& operator*=( const T& c )  
    {  
        //for(int i=0; i<n; ++i){ elems[i] *= c; }  
        std::for_each(elems.begin(), elems.end(), [&](T& l){ l *= c; });  
        return *this;  
    }  
};
```

Ahol lehet, használjunk algoritmusokat:



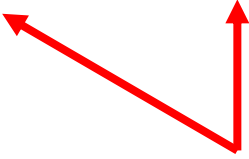
```
template<typename T, int n> struct Vector {  
    ...  
    Vector<T, n>& operator+=( const Vector<T, n>& v )  
    {  
        //for(int i=0; i<n; ++i){ elems[i] += v[i]; }  
        ???  
        return *this;  
    }  
};
```



Ha nincs megfelelő, ne féljünk írni egyet!

```
template<typename I1, typename I2, typename F>
void zipassign2(I1 it1begin, I1 it1end, I2 it2begin, F f)
{
    auto it1 = it1begin;
    auto it2 = it2begin;
    while(it1 != it1end)
    {
        *it1 = f(*it1, *it2);
        ++it1;
        ++it2;
    }
}
```

```
template<typename I1, typename I2, typename F>
void zipassign2(I1 it1begin, I1 it1end, I2 it2begin, F f)
{
    auto it1 = it1begin;
    auto it2 = it2begin;
    while(it1 != it1end)
    {
        *it1 = f(*it1, *it2);
        ++it1;
        ++it2;
    }
}
```



Első adatsor kezdő és vég iterátora

```
template<typename I1, typename I2, typename F>
void zipassign2(I1 it1begin, I1 it1end, I2 it2begin, F f)
{
    auto it1 = it1begin;
    auto it2 = it2begin;
    while(it1 != it1end)
    {
        *it1 = f(*it1, *it2);
        ++it1;
        ++it2;
    }
}
```



Másik adatsor kezdő iterátora

```
template<typename I1, typename I2, typename F>
void zipassign2(I1 it1begin, I1 it1end, I2 it2begin, F f)
{
    auto it1 = it1begin;
    auto it2 = it2begin;
    while(it1 != it1end)
    {
        *it1 = f(*it1, *it2);
        ++it1;
        ++it2;
    }
}
```

A kétváltozós függvény, amit a párokra hattaní akarunk



```
template<typename T, int n> struct Vector {  
    ...  
    Vector<T, n>& operator+=( const Vector<T, n>& v )  
    {  
        //for(int i=0; i<n; ++i){ elems[i] += v[i]; }  
        zipassign2(elems.begin(), elems.end(),  
                  v.elems.begin(),  
                  [](auto x, auto y){ return x+y; });  
        return *this;  
    }  
};
```

```
template<typename T, int n> struct Vector {  
    ...  
    Vector<T, n>& operator+=( const Vector<T, n>& v )  
    {  
        //for(int i=0; i<n; ++i){ elems[i] += v[i]; }  
        zipassign2(elems.begin(), elems.end(),  
                  v.elems.begin(),  
                  std::plus<T>()//[(auto x, auto y){ return x+y; }]);  
        return *this;  
    }  
};
```

A gyakori műveletekre van rövidítés

A jelenlegi iterátor alapú megközelítés addig működik, amíg mindig megvan, hogy hova akarunk írni! (Ez az STL2-ben javulni fog a [range](#)-kkel)

Amíg egy fix méretű vektornál a globális operátorok esetén lista inicializációt írhatunk:

```
template<typename T>  
Vector2<T> operator+( Vector2<T> const& u,  
                      Vector2<T> const& v )  
{ return Vector2<T>{ { {u.x+v.x, u.y+v.y } } }; }
```

...addig, ha algoritmust kell használnunk, akkor előre létre kell hoznunk az eredményt:

```
template<typename T, int n>
Vector<T, n> operator+( Vector<T, n> const& u,
                       Vector<T, n> const& v )
{
    Vector<T, n> res;
    std::transform(u.elems.begin(), u.elems.end(),
                  v.elems.begin(), res.elems.begin(),
                  std::plus<T>());
    return res;
}
```

...addig, ha algoritmust kell használnunk, akkor előre létre kell hoznunk az eredményt:

```
template<typename T, int n>
Vector<T, n> operator+( Vector<T, n> const& u,
                       Vector<T, n> const& v )
{
    Vector<T, n> res;
    std::transform(u.elems.begin(), u.elems.end(),
                  v.elems.begin(), res.elems.begin(),
                  std::plus<T>());
    return res;
}
```

Ez akkor hátrányos, ha itt dinamikus allokáció történik,

továbbá ha  $T$  bonyolult default konstrukciót hajt végre feleslegesen, mert a következő lépésben felülírjuk az eredményét a transformban...

Hogyan célszerű megírni a szorzás operátort?

```
template<typename T, int n, typename C>  
auto operator*( C const& c, Vector<T, n> const& v );
```

Hogyan célszerű megírni a szorzás operátort?

```
template<typename T, int n, typename C>  
auto operator*( C const& c, Vector<T, n> const& v );
```



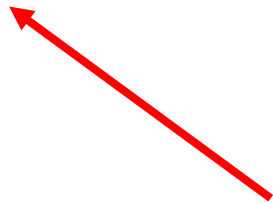
Ha külön típusnak hagyjuk a skalárt, akkor...



Hogyan célszerű megírni a szorzás operátort?

```
Vector<Vector<int, 2>, 2> q{ {-1, 1}, {3, 6} };  
Vector<Vector<int, 2>, 2> r{ {1, -2}, {5, 2} };
```

3 \* q + r



...akkor ezek a kifejezések is automatikusan értelmesek lesznek

Speciális vektorműveletek: nem javasolt az operátor forma  
mert félreértésekhez vezethet mind a felhasználó, mind a fordító szemszögéből

```
template<typename T, int n>
```

```
T dot( Vector<T, n> const& u, Vector<T, n> const& v )  
{ ... }
```

```
template<typename T>
```

```
Vector<T, 3> cross( Vector<T, 3> const& u, Vector<T, 3> const& v )  
{ ... }
```

```
template<typename T, int n>
```

```
Matrix<T, n, n> dyadic( Vector<T, n> const& u, Vector<T, n> const& v )  
{ ... }
```

## Mátrix osztály:

Az indexeléshez nem használhatunk []-t, marad a ().

```
template<typename T, int m, int n>
```

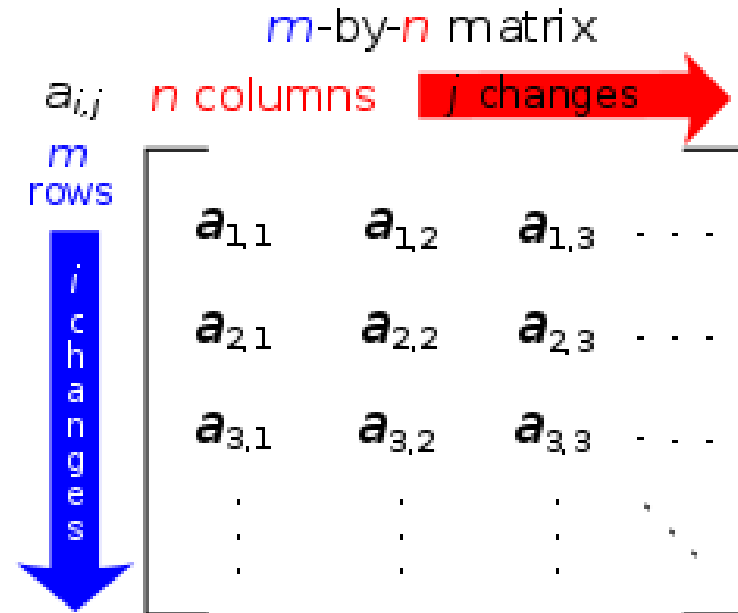
```
struct Matrix  
{
```

```
    std::array<T, m*n> elems;
```

```
    T& operator()(int row, int col) { return elems[row*n+col]; }
```

```
    T const& operator()(int row, int col) const { return elems[row*n+col]; }
```

```
};
```



## std::initializer\_list

Ha azonos elemek sokaságát kell átadni, pl. vektor vagy mátrix konstruktornak:

```
template<typename T, int n> struct Vector
{
    std::array<T, n> a;

    Vector( std::initializer_list<T> il )
    {
        std::copy(il.begin(), il.end(), a.begin());
    }
};

Vector<float, 2> v({2.0f, 3.0f});
Vector<float, 2> u = {2.0f, 3.0f};
```

Ha valamiért saját dinamikus memória kezelést kell írni, akkor erre egy példa:

```
template<typename T> class Vector
{
    T* data;    size_t sz;
public:
    Vector():data{nullptr}, sz{0}{}
    Vector( Vector const& other);
    Vector( Vector &&      other);
    Vector& operator=( Vector const& other);
    Vector& operator=( Vector &&      other);
    ~Vector(){ delete[] data; sz = 0; }
};
```

Ha valamiért saját dinamikus memória kezelést kell írni, akkor erre egy példa:

```
template<typename T> class Vector
{
    T* data;    size_t sz;
public:
    Vector():data{nullptr}, sz{0}{}
    Vector( Vector const& other);
    Vector( Vector &&      other);
    Vector& operator=( Vector const& other);
    Vector& operator=( Vector &&      other);
    ~Vector(){ delete[] data; sz = 0; }
};
```

Ha megengedünk default  
konstruálást, az legyen mindig  
minden null!  
(pointerekre `nullptr` (C++11))

A fenti konstruktor valid ezzel  
a destruktossal is, ugyanis a  
`delete nullptr` nem csinál  
semmit!

```
template<typename T> class Vector
{
    ...
    Vector( Vector const& other):data{new T[other.sz]},
        sz{other.sz}
    {
        std::copy(other.data, other.data+sz, data);
    }
};
```

Copy konstruktor:  
Új memóriát foglalunk,  
lemásoljuk a méretet,  
majd az összes adatot átmásoljuk

```
template<typename T> class Vector
{
    ...
    Vector(Vector && other):data{nullptr},
                        sz{0}
    {
        std::swap(data, other.data); std::swap(sz, other.sz);
    }
};
```

Move konstruktor:

Nullázzuk a memóriát, és méretet, majd megcseréljük mindkettőt a másik vektorral, így a másik olyan lesz, mintha default konstruáltak volna.



```
template<typename T> class Vector
{
    ...
    Vector(Vector && other) : Vector()
    {
        std::swap(data, other.data); std::swap(sz, other.sz);
    }
};
```

Move konstruktor:

Alternatíva: ismerjük fel, hogy az inicializálást már megírtuk a Default Konstruktorban, és ezt felhasználhatjuk:

```
template<typename T> class Vector {  
    ...  
    Vector& operator=( Vector const& other)  
    {  
        if(data!=other.data) ← Másoló értékadás:  
                                Ki kell védeni, az önmagához adást!!!  
        {  
            delete[] data; sz = other.sz;  
            data = new T[sz];  
            std::copy(other.data, other.data+sz, data);  
        }  
        return *this;  
    }  
};
```

A többi ugyan az, mint a copy konstruktornál.

```
template<typename T> class Vector {  
    ...  
    Vector& operator=( Vector && other )  
    {  
        if(data!=other.data) ← Mozgató értékadás:  
                                Ki kell védeni, az önmagához adást!!!  
        {  
            delete[] data; data = nullptr; sz = 0;  
            std::swap(data, other.data); std::swap(sz, other.sz);  
        }  
        return *this;           A többi ugyan az, mint a move konstruktornál.  
    }  
};
```

## Stream-ek, string-ek

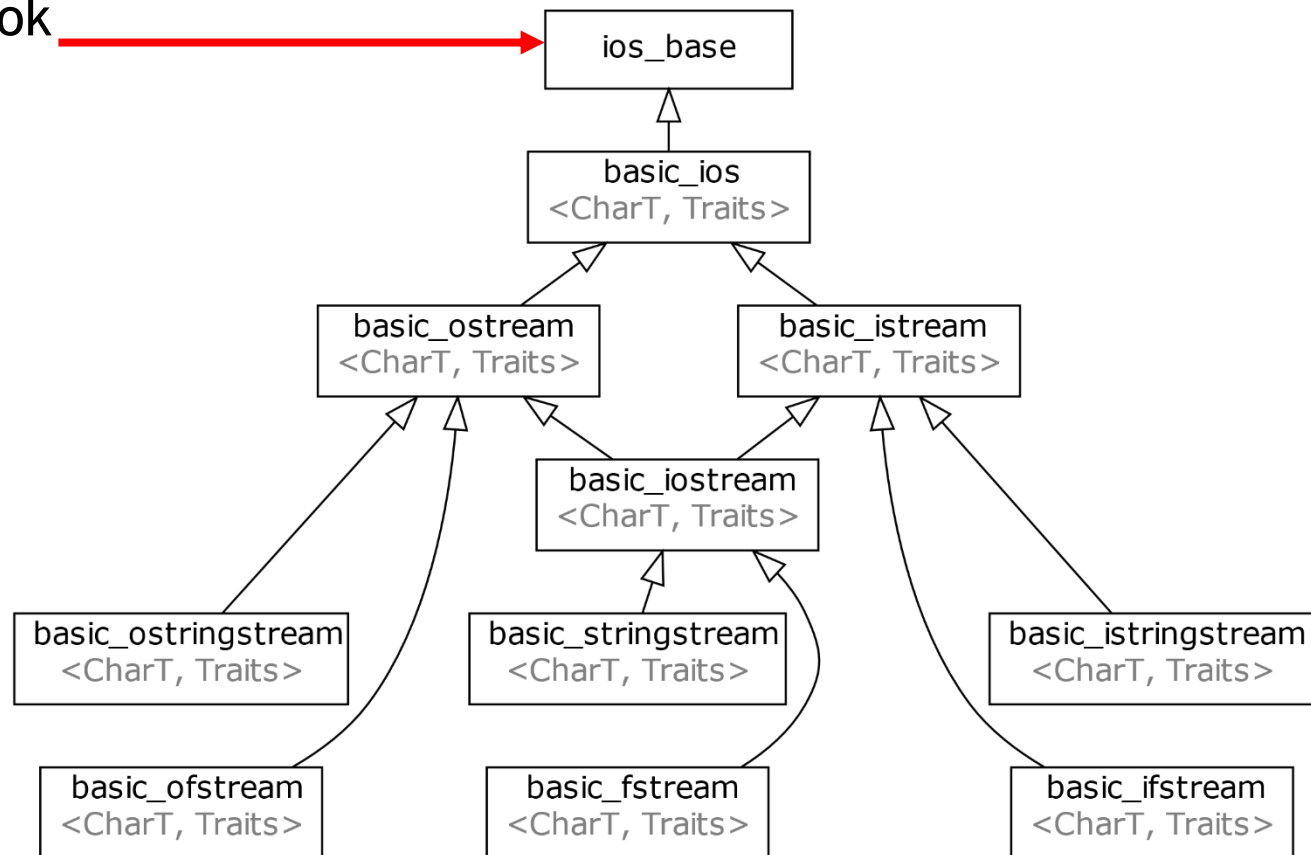
A stream-ek adatfolyamokat reprezentálnak, amik potenciálisan végtelenek is lehetnek és csak egy irányban írhatók és/vagy olvashatóak.

Példák:

- Fájlok
- Standard kimenet, bemenet, hiba
- Külső eszközök
- Hálózat

## A C++ a ki/bemeneteket kezelő osztályok hierarchiája:

Az `ios_base` felelős a beépített típusok formázásáért és a stream hibaállapotainak kezeléséért

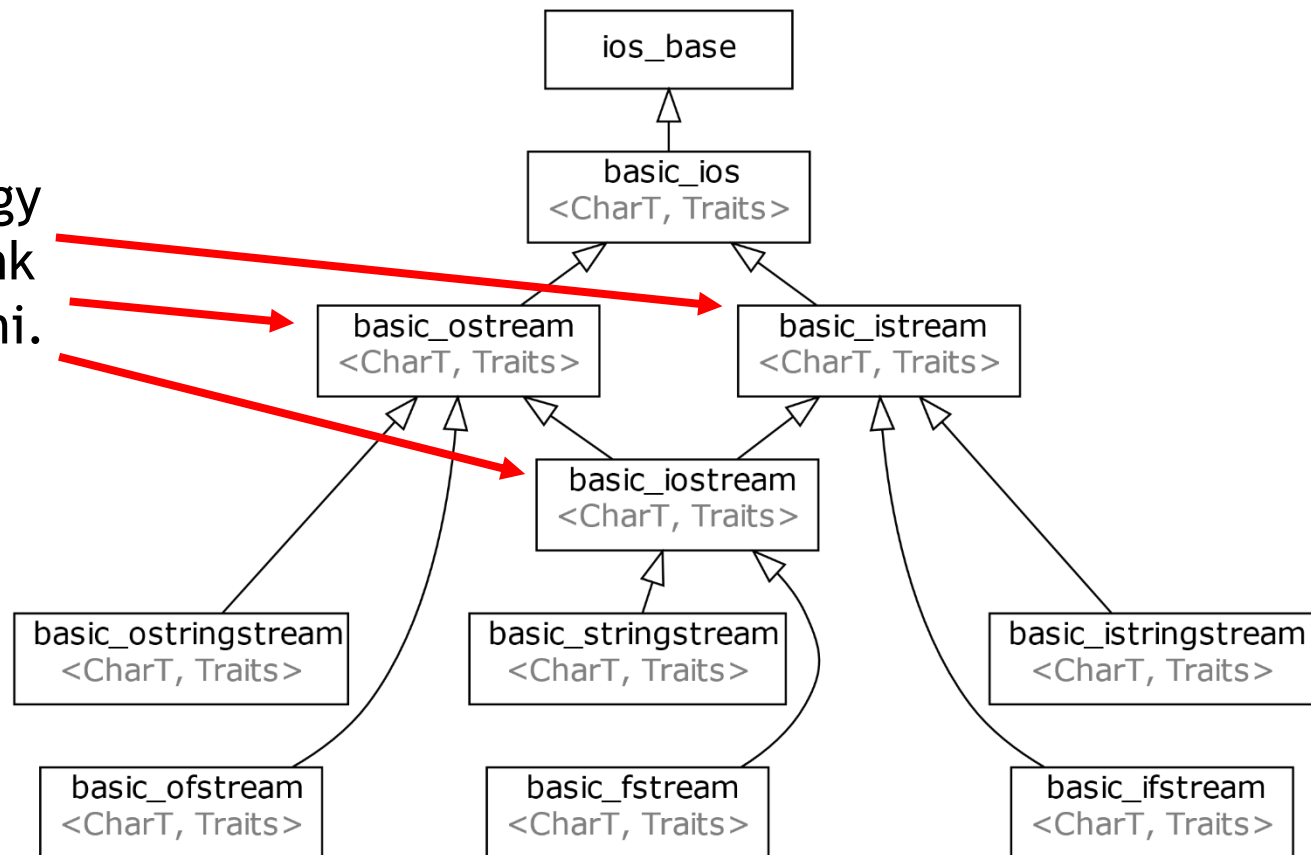




## A C++ a ki/bemeneteket kezelő osztályok hierarchiája:

A `basic_i/o/iostream` objektum teszi elérhetővé azokat a magasszintű műveleteket, amikkel formázott, vagy formázatlan (bináris) adatokat tudunk a streamekbe írni, vagy onnan olvasni.

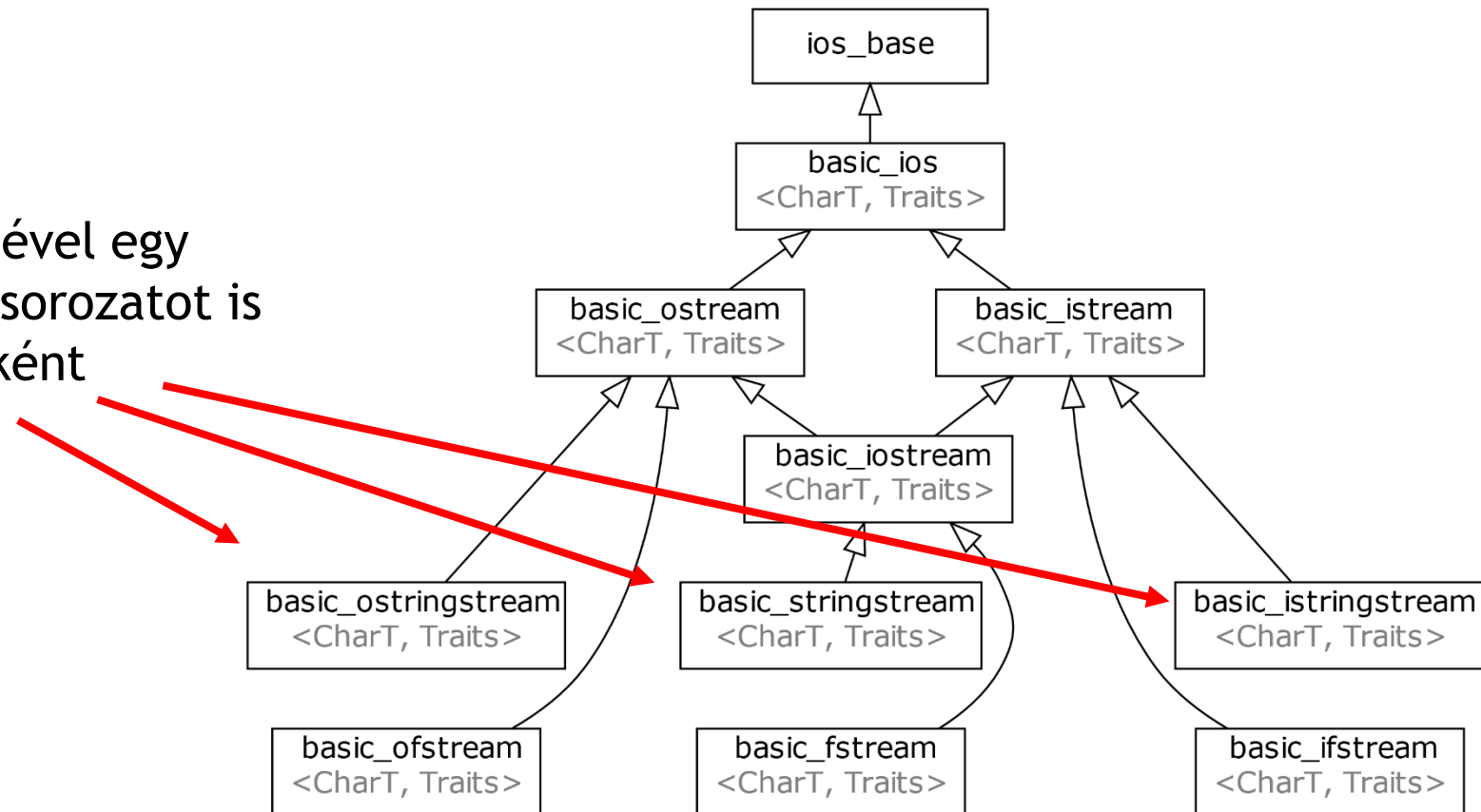
Ez adja az ismerős `>>` és `<<` operátorokat is, illetve az új túlterheléseket ezzel a típussal kell bevezetni (példák később).





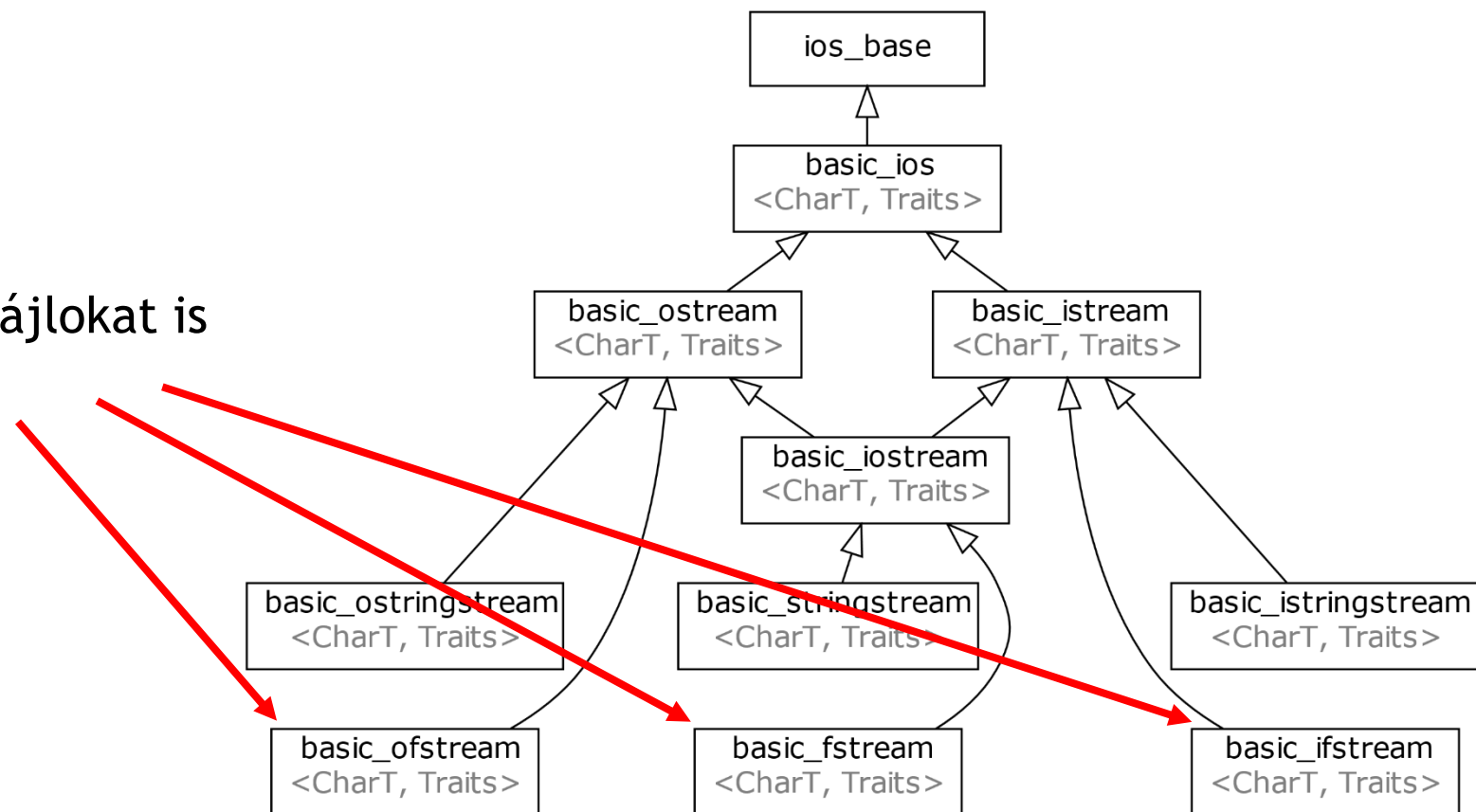
## A C++ a ki/bemeneteket kezelő osztályok hierarchiája:

A standard library segítségével egy memóriában levő karaktersorozatot is kezelhetünk i/o/iostreamként



## A C++ a ki/bemeneteket kezelő osztályok hierarchiája:

Valamint természetesen fájlokat is



A C++ I/O nagyon összetett és kicsit rosszul megtervezett, de ennek ellenére sok mindent meg lehet oldani 1 sorban vele.

Számos interneten fellelhető példakódban a file I/O elég „fapadosan” van megoldva...

A beolvasáshoz a [különböző](#) iterátorok és iterátor adaptorok nagyon hasznosak.

Egy text fájl megnyitása és `int`-ek másolása belőle egy `std::vector`-ba:

```
std::vector<int> data;

std::ifstream input("data.txt");
if( input.is_open() )
{
    std::copy( std::istream_iterator<int>(input),
               std::istream_iterator<int>(),
               std::back_inserter(data) );
}
else{ std::cout << "Could not open input file\n"; }
```

Egy text fájl megnyitása és `int`-ek másolása belőle egy `std::vector`-ba:

```
std::vector<int> data;
```

A fájlt bemeneti streamként  
reprezentáló objektum

```
std::ifstream input("data.txt");
```

```
if( input.is_open() )
```

Ellenőrizni kell, hogy sikerült-e megnyitni a fájlt!

```
{
```

```
    std::copy( std::istream_iterator<int>(input),
```

A fájl eleje iterátor

```
            std::istream_iterator<int>(),
```

A fájl vége iterátor

```
            std::back_inserter(data) );
```

Spec iterátor a vectorra,  
ami a véghez told, ha  
írnak bele

```
}
```

```
else{ std::cout << "Could not open input file\n"; }
```

# C++ példa

Egy text fájl megnyitása és `int`-ek másolása bele egy `std::vector`-ból:

```
std::vector<int> data = ...;

std::ofstream output("data.txt");
if( output.is_open() )
{
    std::copy( data.begin(),
               data.end(),
               std::ostream_iterator<int>(output, " ") );
}
else{ std::cout << "Could not open output file\n"; }
```

Egy text fájl megnyitása és `int`-ek másolása bele egy `std::vector`-ból:

```
std::vector<int> data = ...;
```

A fájlt kimeneti streamként  
reprezentáló objektum

```
std::ofstream output("data.txt");
```

```
if( output.is_open() )
```

Ellenőrzés, hogy nyitva van-e

```
    std::copy( data.begin(),
```

Adatok eleje iterátor

```
              data.end(),
```

Adatok vége iterátor

```
              std::ostream_iterator<int>(output, " ") );
```

```
}
```

```
else{ std::cout << "Could not open output file\n"; }
```

`int`-eket kiíró, és  
azokat szóközzel  
elválasztó iterátor

Még egyszer a különbség:

Beolvasás:

```
std::copy( std::istream_iterator<T>(input),  
          std::istream_iterator<T>(),  
          std::back_inserter(data) );
```

Kiírás:

```
std::copy( data.begin(),  
          data.end(),  
          std::ostream_iterator<T>(output, " ") );
```



A stream-ek formázott beolvasás és kiírás operátorai:

```
std::ifstream ifile("data.txt");
```

```
int i = 0;  
ifile >> i;
```

Átalakítja a streamben aktuálisan levő karaktereket `int`-é ha lehetséges. Ha nem sikerül, `i` nem módosul.

```
std::ofstream ofile("data2.txt");
```

```
ofile << i;
```

Átalakítja `i`-t karakterekké és kiírja a stream-be.

A karaktersorozatokot a C-stílusú `const char*` helyett `std::string`-ekként célszerű kezelni:

```
std::string str("mytext");
```

Az `std::string` ugyan úgy heap-en dinamikusán allokált tároló, mint az `std::vector`, csak némi plusz funkcionalitással.


Az `std::string` egyik konstruktora egy iterátor párt vár, ezzel is beolvashatunk egy fájlt, csak vigyázzunk a formázásra:

```
std::ifstream file("data.txt");
std::string str1{ std::istream_iterator<char>(file),
                 std::istream_iterator<char>()};

file.clear();
file.seekg(0);
std::string str2{ std::istreambuf_iterator<char>(file),
                 std::istreambuf_iterator<char>()};
```

Az `std::string` egyik konstruktora egy iterátor párt vár, ezzel is beolvashatunk egy fájlt, csak vigyázzunk a formázásra:

```
std::ifstream file("data.txt");  
std::string str1{ std::istream_iterator<char>(file),  
                 std::istream_iterator<char>()};  
  
file.clear();  
file.seekg(0);  
std::string str2{ std::istreambuf_iterator<char>(file),  
                 std::istreambuf_iterator<char>()};
```

 Ez az iterátor eldobja a white space karaktereket (space, tab, sortörés)

Az `std::string` egyik konstruktora egy iterátor párt vár, ezzel is beolvashatunk egy fájlt, csak vigyázzunk a formázásra:

```
std::ifstream file("data.txt");  
std::string str1{ std::istream_iterator<char>(file),  
                 std::istream_iterator<char>()};
```

```
file.clear();  
file.seekg(0);
```

```
std::string str2{ std::istreambuf_iterator<char>(file),  
                 std::istreambuf_iterator<char>()};
```

Ez az iterátor nem hagyja el a white space karaktereket



Az `std::string` egyik konstruktora egy iterátor párt vár, ezzel is beolvashatunk egy fájlt, csak vigyázzunk a formázásra:

```
std::ifstream file("data.txt");  
std::string str1{ std::istream_iterator<char>(file),  
                 std::istream_iterator<char>()};
```

```
file.clear();
```

```
file.seekg(0);
```

```
std::string str2{ std::istreambuf_iterator<char>(file),  
                 std::istreambuf_iterator<char>()};
```

A file elejére visszamenéshez először törölni kell a file vége jelet a stream-ből,

Majd vissza lehet állítani a beolvasási pozíciót az elejére

Általában a tárolóknak van konstruktora iterátor párból, így `std::copy` nélkül is beolvashatunk:

```
std::ifstream file("data.txt");
```

```
std::vector<int> data{std::istream_iterator<int>(file),  
                    std::istream_iterator<int>()};
```

Egy karakter stringet is használhatunk stream-ként az [std::stringstream](#) segítségével:

```
std::stringstream ss;
```

```
ss << sqrt(2.0);
```

← Kiírunk egy lebegőpontos számot a streambe

```
std::string s = ss.str();
```

← Le tudjuk kérdezni a stringet

```
float y = 0.0f;
```

```
ss >> y;
```

← Vissza tudjuk olvasni a stream-ből a karaktereket lebegőpontos számként

```
if( ss ){ std::cout << "Conversion to float succeeded.\n"; }  
else    { std::cout << "Conversion to float failed.\n"; }
```



A streamek belső bufferén keresztül hatékonyan tudunk teljes adatfolyamokat másolni (formázatlanul):

```
std::ifstream file("data.txt");  
std::stringstream ss;  
  
if(file)  
{  
    ss << file.rdbuf();  
}
```

Hogyan kell megírni a formázási operátorokat,  
azaz a beolvasót >>, és a kiírókat << saját típusokra?

Vegyünk egy saját típust, pl.:

```
struct Data
{
    int i;
    double x;
    std::string s;
};
```

A kiíró operátor:

```
struct Data{ int i; double x; std::string s; };
```

```
std::ostream& operator<<( std::ostream& s, Data const & d )  
{  
    s << d.i <<" , " << d.x <<" , " << d.s;  
    return s;  
}
```

A kiíró operátor:

A kimeneti stream objektumot referenciaként kötelező átvennünk, és ugyan így visszaadnunk!

```
struct Data{ int i; double x; std::string s; };
```

```
std::ostream& operator<<( std::ostream& s, Data const & d )  
{  
    s << d.i <<"", " << d.x <<"", " << d.s;  
    return s;  
}
```



Beolvasás, a bemeneti stream-et most is referenciaként kapjuk és adjuk vissza:

```
std::istream& operator>>( std::istream& s, Data& d )  
{  
    std::string tmp;  
    std::getline(s, tmp);  
    if(tmp.size() > 0)  
    {  
        std::stringstream ss(tmp);  
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);  
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);  
        std::getline(ss, d.s);  
    }  
    return s; ←  
}
```

```
std::istream& operator>>( std::istream& s, Data& d )
```

```
{
```

```
    std::string tmp;
```

```
    std::getline(s, tmp);
```

```
    if(tmp.size() > 0){
```

```
        std::stringstream ss(tmp);
```

```
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
```

```
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);
```

```
        std::getline(ss, d.s);
```

```
    }
```

```
    return s;
```

```
}
```

Olyan formátumot szeretnénk beolvasni, ahol vesszővel vannak elválasztva az értékek:

2, 3.14, abcd

5, 5.67, xyz

```
std::istream& operator>>( std::istream& s, Data& d )
```

```
{
```

```
    std::string tmp;
```

```
    std::getline(s, tmp); ←
```

```
    if(tmp.size() > 0){
```

```
        std::stringstream ss(tmp);
```

```
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
```

```
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);
```

```
        std::getline(ss, d.s);
```

```
    }
```

```
    return s;
```

```
}
```

Az [std::getline](#) beolvas egy adott határkarakterig mindent egy streamból egy stringbe.

A harmadik argumentum a határkarakter, alapértelmezésben a sortörés, tehát egy egész sort olvasunk be.



```
std::istream& operator>>( std::istream& s, Data& d )
```

```
{
```

```
    std::string tmp;
```

```
    std::getline(s, tmp);
```

```
    if(tmp.size() > 0){
```

Ellenőrizni kell, hogy sikeres volt-e a beolvasás.



```
        std::stringstream ss(tmp);
```

```
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
```

```
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);
```

```
        std::getline(ss, d.s);
```


```
    }
```

```
    return s;
```

```
}
```

```
std::istream& operator>>( std::istream& s, Data& d )
{
    std::string tmp;
    std::getline(s, tmp);
    if(tmp.size() > 0){
        std::stringstream ss(tmp);
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);
        std::getline(ss, d.s);
    }
    return s;
}
```

Készítsünk egy stream-et a beolvasott stringből



```
std::istream& operator>>( std::istream& s, Data& d )
```

```
{
```

```
    std::string tmp;
```

```
    std::getline(s, tmp);
```

```
    if(tmp.size() > 0){
```

```
        std::stringstream ss(tmp);
```

```
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
```

```
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);
```

```
        std::getline(ss, d.s);
```

```
    }
```

```
    return s;
```

```
}
```

Majd a stringstream-ból a következő vesszőig kiolvassuk a karaktereket és `int`-é alakítjuk

```
std::istream& operator>>( std::istream& s, Data& d )
```

```
{
```

```
    std::string tmp;
```

```
    std::getline(s, tmp);
```

```
    if(tmp.size() > 0){
```

```
        std::stringstream ss(tmp);
```

```
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
```

```
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);
```

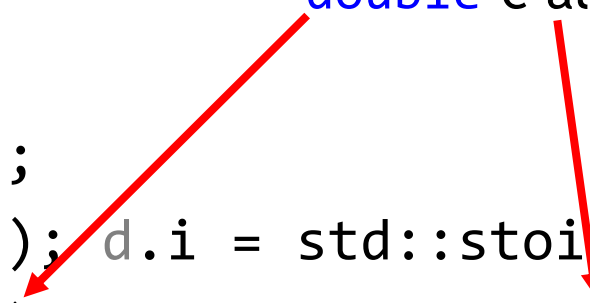
```
        std::getline(ss, d.s);
```

```
    }
```

```
    return s;
```

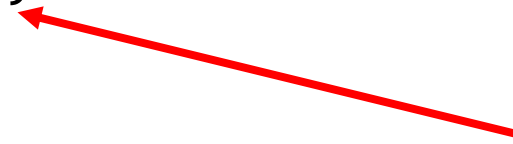
```
}
```

Majd ismét a következő vesszőig  
kiolvassuk a karaktereket és  
**double**-é alakítjuk



```
std::istream& operator>>( std::istream& s, Data& d )
{
    std::string tmp;
    std::getline(s, tmp);
    if(tmp.size() > 0){
        std::stringstream ss(tmp);
        std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
        std::getline(ss, tmp, ','); d.x = std::stod(tmp);
        std::getline(ss, d.s);
    }
    return s;
}
```

Majd ami maradt, azt egyenesen beolvassuk a Data struktúra s nevű string mezőjébe.



Bővebb példa: részecske pályák beolvasása

Formátum:

```
név energia px py pz px py pz px py pz ...
```

[structured\\_io.cpp](#)

A stringek és a streamek eddigi példái sima ASCII karakterekkel dolgoztak (ami elrejtett template paraméter volt), ahol az elemi karakter elfér az általában 8 bit-es `char` típusban.

A szélesebb, pl. Unicode reprezentációhoz a `w_char`, `char16_t`, `char32_t` típusokat és a rájuk specializált stream-eket és stringeket kell használni.

A lokalizáció (pl. ékezetes betűk) és a nem ASCII karakterkezelés, vagy az eltérő kódolások és szélességek közötti átalakítás valódi rémálom, amit sokszor nagyon nehéz hordozhatóan megoldani...

A string konverziókra C++11-óta van standard módszer:

`std::string` vagy `std::wstring` átalakítása adott numerikus típusra,  
pl.: `std::stoi` ha `int`-et akarunk kapni

string-é alakítás: `std::to_string`, `std::to_wstring`

vagy más szemantikával ugyan ezek a funkciók C++17-től:  
`std::to_chars`, `std::from_chars`



További hasznos szolgáltatások a standard library-ben

Standardizált időmérés (C++11, #include <chrono>)

```
auto t1 = std::chrono::high_resolution_clock::now();
```

```
//some lengthy operation...
```

```
auto t2 = std::chrono::high_resolution_clock::now();
```

```
long long duration =  64 bites előjeles egész típus (C++11)
```

```
std::chrono::duration_cast<std::chrono::microseconds>(t2-t1).count();
```

Standardizált időmérés (C++11, #include <chrono>)

Ha lebegőpontosan szeretnénk:

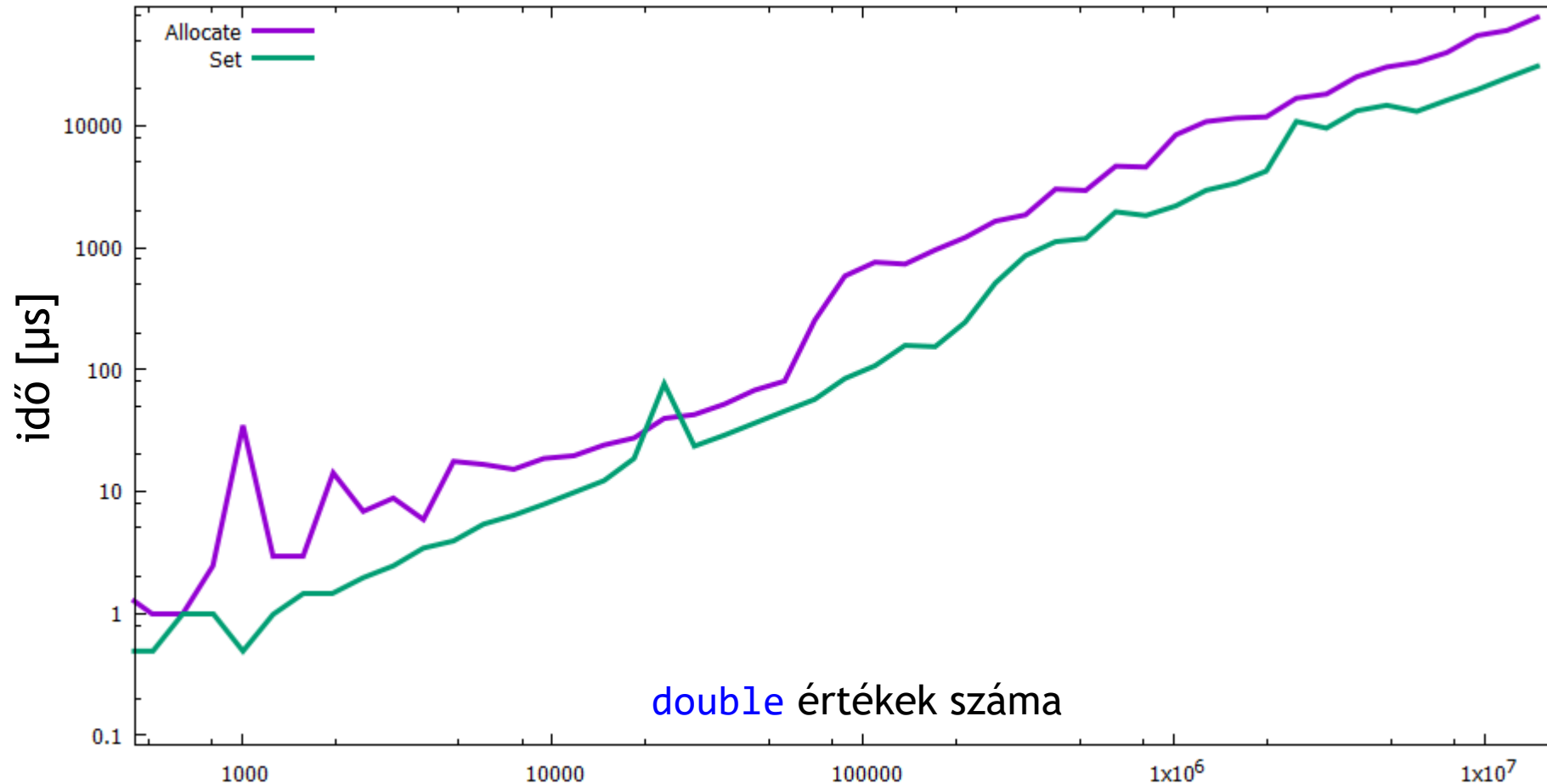
```
auto t1 = std::chrono::high_resolution_clock::now();  
//some lengthy operation...  
auto t2 = std::chrono::high_resolution_clock::now();  
  
using floating_milliseconds  
    = std::chrono::duration<double, std::chrono::milliseconds::period>;  
  
double duration = floating_milliseconds{ t2 - t1 }.count();
```

Példakód: Melyik tart tovább?

- Lefoglalni sok `double`-et
- Vagy kinullázni őket?

[time\\_measurement.cpp](#)

## [time measurement.cpp](#)



## Reguláris kifejezések (C++11, `#include <regex>`):

Ha véletlenül valakinek az awk vagy a grep funkcionalitása kellene adott alakú stringek megtalálásához és átalakításához...

```
std::string text = "Quick brown fox";  
std::regex vowel_re("a|i|e|o|u");  
std::cout << std::regex_replace(text, vowel_re, "$&");
```

Az eredmény: Q[u][i]ck br[o]wn f[o]x

## Véletlenszám generálás (C++11, #include <random>):

```
std::vector<double> data(1500);
```

```
std::random_device rnd_device;
```

```
std::mt19937 mersenne_engine(rnd_device());
```


```
std::normal_distribution<double> dist(1.0, 0.5);
```

```
std::generate(data.begin(),
```

```
                data.end(),
```

```
                [&]{ return dist(mersenne_engine); });
```

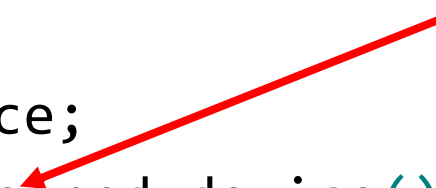
Ez a "seed", ami ideális esetben teljesen véletlen (hw random generátor)

A red arrow points from the text to the `rnd_device` variable in the code above.

## Véletlenszám generálás (C++11, #include <random>):

```
std::vector<double> data(1500);  
  
std::random_device rnd_device;  
std::mt19937 mersenne_engine(rnd_device());  
std::normal_distribution<double> dist(1.0, 0.5);  
  
std::generate(data.begin(),  
              data.end(),  
              [&]{ return dist(mersenne_engine); });
```

Ez egy determinisztikus, de nagyon jó közelítéssel random egészeket adó generátor, amit a random seed-el inicializálunk





## Véletlenszám generálás (C++11, #include <random>):

```
std::vector<double> data(1500);
```

```
std::random_device rnd_device;
```

```
std::mt19937 mersenne_engine(rnd_device());
```

```
std::normal_distribution<double> dist(1.0, 0.5);
```

```
std::generate(data.begin(),  
              data.end(),
```

```
              [&]{ return dist(mersenne_engine); });
```

Ez egy (normál) eloszlás objektum, aminek ha random egészeket adunk a generátorból, akkor ez a nevében foglalt eloszlásúvá alakítja őket

Példakód: normál eloszlású számok generálása, majd átlag és szórás számolása

[simple\\_io\\_statistics.cpp](#)

Fájlrendszer kezelés (C++17, #include <filesystem>):

```
namespace fs = std::filesystem;
fs::create_directories("test/a/b");
std::ofstream("test/file1.txt");
std::ofstream("test/file2.txt");
for(auto& p : fs::directory_iterator("test"))
{
    std::cout << p << '\n';
}
fs::remove_all("test");
```

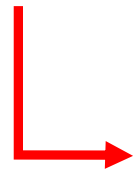
Példa:

A legutóbb módosított fájl megkeresése a mappában:

[filesystem ts.cpp](#)

Párhuzamos algoritmusok (C++17):  
A legtöbb algoritmusnak lesz CPU párhuzamos verziója is!

```
std::sort(___, data.begin(), data.end());
```



3 lehetőség:

- seq - szekvenciális (soros)
- par - párhuzamos
- par\_unseq - párhuzamos és vektorizált

Párhuzamos algoritmusok (C++17):  
A legtöbb algoritmusnak lesz CPU párhuzamos verziója is!

```
std::sort(____, data.begin(), data.end());
```

Sőt! Számos GPU API-ra is készül implementációja a Parallel STL-nek, így ha a kódunk algoritmusok mentén van felépítve, akkor minimális munkával lehet majd GPU gyorsítani.

Számos kisebb nagyobb fejlesztés lett elfogadva még a C++17-el, pl. innen és innen (meg innen) lehet még szemezgetni.