3. fejezet

Constness

Referenciák

Tömb jellegű tárolók

Haladó Alkalmazott Programozás
ELTE, 2019

CMake

- Ajánlott CMake fájl minta a fordítási figyelmeztetések bekapcsolásához, amely a három legfontosabb fordítóval működik:

- https://gist.github.com/MathiasMagnus/1e8bc78cc68e8977063207c16bfa1ce6

CMake

https://gist.github.com/MathiasMagnus/1e8bc78cc68e8977063207c16bfa1ce6

```
cmake_minimum_required(VERSION 3.8) # CXX_STANDARD 17
project(template LANGUAGES CXX)
if (MSVC)
  string(REGEX REPLACE "/W[0-9]" "" CMAKE_CXX_FLAGS ${CMAKE_CXX_FLAGS})
endif (MSVC)

add_executable(${PROJECT_NAME} Main.cpp)
set_target_properties(${PROJECT_NAME} PROPERTIES CXX_STANDARD 17
                                                 CXX_STANDARD_REQUIRED ON
                                                 CXX_EXTENSIONS OFF)

target_compile_options(${PROJECT_NAME} PRIVATE
$<$<OR:$<CXX_COMPILER_ID:GNU>,$<CXX_COMPILER_ID:Clang>>:-Wall -Wextra -pedantic>
$<$<CXX_COMPILER_ID:MSVC>:/W4 /permissive->)
```

Constness

Constness

C++-ban és sok más nyelvben a változók alapból változhatnak.

Ha ezt nem akarjuk
(márpedig ez sokszor felesleges szabadsági fok és ezáltal hibalehetőség),
akkor ki kell írnunk a `const` minősítőt!

Constness

```
double Pi = 3.1415926535897932;

Pi += 10.0;
```

Constness

```
double Pi = 3.14159265358979932;

Pi += 10.0;
```

Constness

```
const double Pi = 3.1415926535897932;

Pi += 10.0;
```

Ez most már hiba!

clang:

```
error: assignment of read-only variable 'Pi'
     Pi += 10.0;
           ^~~~
```

Constness

A `const` változók különösen függvények argumentumaiként lesznek érdekesek a következőek fényében...

Referenciák

Referenciák

Számos esetben akarunk egy értékre közvetett módon hivatkozni.

A legfontosabb alkalmazás:

Azt akarjuk, hogy egy érték 1x szerepeljen a memóriában, de több helyről, pl.: másik függvény(ek)ből tudjunk rá hivatkozni.

Referenciák

Példa:

```
double A = 5.0;
```


Az 5.0 érték 1x szerepel a memóriában

```
double& refA = A;
```

Referenciák

Példa:

```
double A = 5.0;

double& refA = A;
```

`refA` egy hivatkozás `A`-ra.
(pointer méretű)

Tömb jellegű tárolók

Tömb jellegű tárolók

A két legfontosabb tömb jellegű tároló C++-ban:

- std::array
- std::vector

Tömb jellegű tárolók

A két legfontosabb tömb jellegű tároló C++-ban:

- std::array ← Fordítás időben ismert méretű tömb a stack-en
  (alap esetben kevés elemre praktikus)
- std::vector

Tömb jellegű tárolók

A két legfontosabb tömb jellegű tároló C++-ban:

- std::array
- std::vector ← Futás időben változó méretű tömb a heap-en
  (ebbe bármennyit pakolhatunk, amíg bírjuk memóriával)

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <array>

int main()
{
  std::array<int, 3> A = {1, 2, 3};
  int x = A[0];
  A[1] = 7;
  std::cout << A[0] << ", " << A[1] << ", " << A[2] << "\n";
}
```

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <array>

int main()
{
  std::array<int, 3> A = {1, 2, 3};
  int x = A[0];
  A[1] = 7;
  std::cout << A[0] << ", " << A[1] << ", " << A[2] << "\n";
}
```

Az `std::array`-hoz ez az include kell.

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <array>

int main()
{
  std::array<int, 3> A = {1, 2, 3};
  int x = A[0];
  A[1] = 7;
  std::cout << A[0] << ", " << A[1] << ", " << A[2] << "\n";
}
```

Milyen típust tárolunk (most `int`)

Hány darabot (most 3)

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <array>

int main()
{
  std::array<int, 3> A = {1, 2, 3};
  int x = A[0];
  A[1] = 7;
  std::cout << A[0] << ", " << A[1] << ", " << A[2] << "\n";
}
```

Inicializáció: {} között ,-vel elválasztva

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <array>

int main()
{
  std::array<int, 3> A = {1, 2, 3};
  int x = A[0];
  A[1] = 7;
  std::cout << A[0] << ", " << A[1] << ", " << A[2] << "\n";
}
```

Indexelés (olvasás, írás)

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> V = {1, 2, 3};
  int x = V[0];
  V[1] = 7;
  std::cout << V[0] << ", " << V[1] << ", " << V[2] << "\n";
}
```

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> V = {1, 2, 3};
  int x = V[0];
  V[1] = 7;
  std::cout << V[0] << ", " << V[1] << ", " << V[2] << "\n";
}
```

Az `std::vector`-hoz ez az include kell.

Tömb jellegű tárolók

Használat:

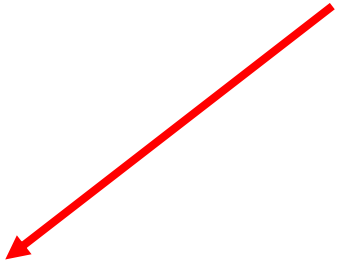
```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> V = {1, 2, 3};
  int x = V[0];
  V[1] = 7;
  std::cout << V[0] << ", " << V[1] << ", " << V[2] << "\n";
}
```

Tömb jellegű tárolók

Használat:

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> V = {1, 2, 3};
  int x = V[0];
  V[1] = 7;
  std::cout << V[0] << ", " << V[1] << ", " << V[2] << "\n";
}
```

Az elemszám az inicializációból jön (ebben az esetben)



Tömb jellegű tárolók

C++-ban minden tárolónak van mérete:

```
std::vector<int> B = {1, 2, 3};
size_t n = B.size(); //n == 3
```

Viszont a visszaadott érték egy spéci előjel nélküli egész (nem `int`!)

Ha át akarjuk alakítani:

```
int m = static_cast<int>(B.size());
```

const + ref + tömbök

const + ref + tömbök

Ez a négy függvény teljesen más jelentéssel bír!

```
int f(std::vector<int>         v);

int f(std::vector<int> const   v);

int f(std::vector<int> &       v);

int f(std::vector<int> const& v);
```

const + ref + tömbök

Ez a négy függvény teljesen más jelentéssel bír!

```
int f(std::vector<int>       v);
```
A vector le lesz másolva és belül változhat

```
int f(std::vector<int> const v);
```
A vector le lesz másolva,
de belül nem változhat

```
int f(std::vector<int> &     v);
```
A vector nem lesz lemásolva, de a referencián keresztül módosulhat a hívó oldali tartalom!

```
int f(std::vector<int> const& v);
```
A vector nem lesz lemásolva
és nem is módosulhat

const + ref + tömbök

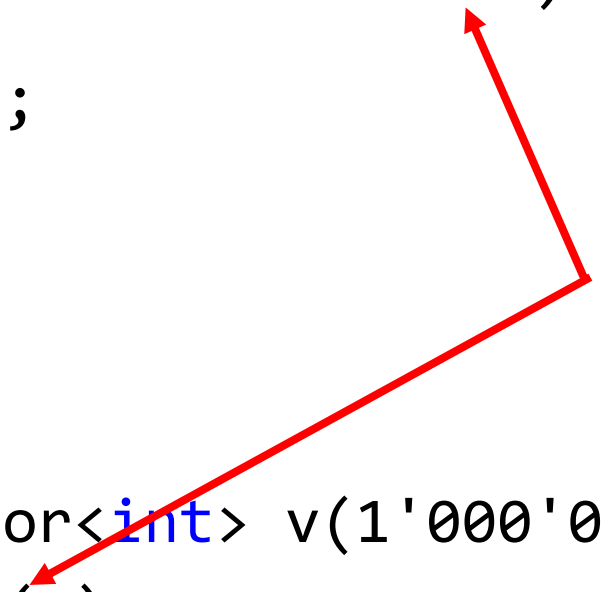
Illusztráció: nagy méretű vector átadásának hatása a memóriára!

const + ref + tömbök

```
#include <vector>
int first(std::vector<int> v)
{
  return v[0];
}
int main()
{
  int n = 0;
  {
    std::vector<int> v(1'000'000'000);
    n = first(v);
  }
  return (int)n;
}
```

const + ref + tömbök

```
#include <vector>
int first(std::vector<int> v)
{
  return v[0];
}
int main()
{
  int n = 0;
  {
    std::vector<int> v(1'000'000'000);
    n = first(v);
  }
  return (int)n;
}
```



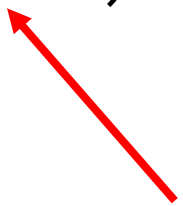
Ebben az esetben a first hívásnál lemásolódik ~4 GB adat, csak azért, hogy az első elemet kiválasszuk...

const + ref + tömbök

```
#include <vector>
int first(std::vector<int> const& v)
{
  return v[0];
}
int main()
{
  int n = 0;
  {
    std::vector<int> v(1'000'000'000);
    n = first(v);
  }
  return (int)n;
}
```

const + ref + tömbök


```
#include <vector>
int first(std::vector<int> const& v)
{
  return v[0];
}
int main()
{
  int n = 0;
  {
    std::vector<int> v(1'000'000'000);
    n = first(v);
  }
  return (int)n;
}
```



Ebben az esetben nincs másolás hála a referenciának!

const + ref + tömbök

Ajánlás: ha nincs alapos indokunk, hogy másképp járjunk el, akkor
1) Minden legyen const, aminek nem muszáj változnia
2) Minden nagy méretű dolgot referenciaként adjunk át

Modern C++-ban, pláne kezdőként nem javasolt a pointerek és a C-stílusú tömbök használata!

Tömbök adatainak feldolgozása

Tömbök adatainak feldolgozása

Ha a tömbökön akarunk műveleteket végezni, akkor.

- Írhatunk szokásos for-ciklust (nem ajánlott, ha van más megoldás!)
- Írhatunk range-based for-ciklust (C++11 új szintaxis)
- Használhatjuk a standard library-ban levő algoritmusokat!

Tömbök adatainak feldolgozása

Tipikus példa, amit csak for ciklussal lehet megcsinálni: ha egyszerre kell a tömb eleme és az indexe is, pl.: Numerikus integrátor

```
template<typename F>
auto integrate(std::vector<double> const& w, F const& f,
          const int n, const double x0, const double x1)
{
  const double h = (x1-x0)/n;
  double sum = 0.0;
  for(int i=0; i<n; ++i)
  {
    sum += f(x0 + i*h) * w[i];
  }
  return sum;
}
```

Tömbök adatainak feldolgozása

Range-based for-loop: sorban végrehajtja a ciklus törzsét minden elemen, egymás után, úgy, hogy mindig a ciklus elején bevezetett változó (most e) jelöli az aktuális elemet a tárolóból:

```
std::vector<double> v;

for (auto e : v)
{
  std::cout << e << " ";
}
std::cout << '\n';
```

Tömbök adatainak feldolgozása

Algoritmusok: rengeteg van, nézzünk körül

Legfontosabbak:

- transform – átalakít egy elemenkénti függvénnyel egyik tárolóból a másikba.
- accumulate – felösszegez egy tömböt egy függvénnyel
- find – megkeresi az első elemet, amire egy függvény igazat ad
- generate – feltölt egy tárolót egy függvény segítségével
- sort – sorbarendez egy tárolót egy relációs függvény segítségével

Tömbök adatainak feldolgozása

Példa: vector elemeinek összege:

```
std::vector<int> B = {1, 2, 3};

int sum = std::accumulate(B.begin(), B.end(), 0);
```

$$\sum_i b_i$$

Tömbök adatainak feldolgozása

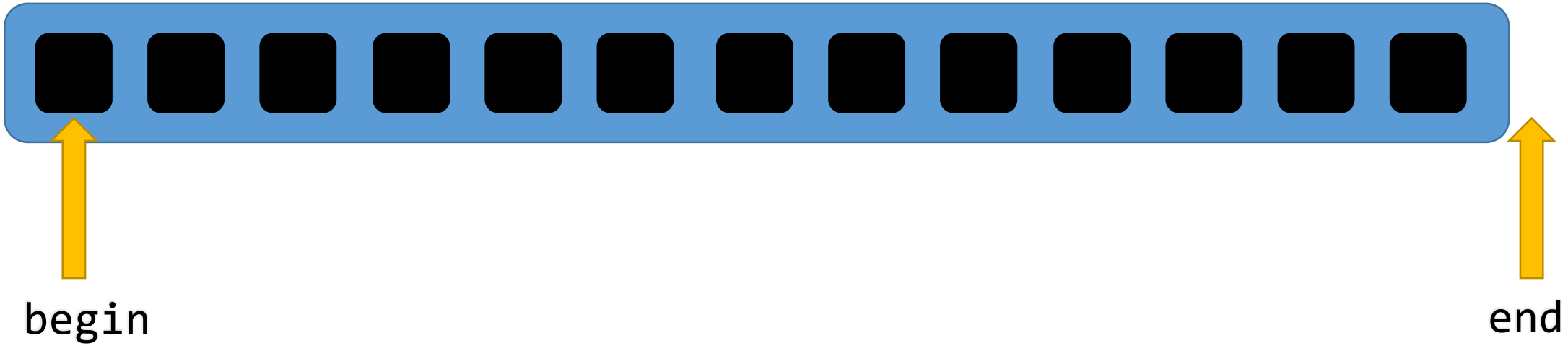
Példa: vector elemeinek összege:

```
std::vector<int> B = {1, 2, 3};

int sum = std::accumulate(B.begin(), B.end(), 0);
```

Az algoritmusok iterátorokat várnak (részletesen később), a tömb elejére és vége utánra (!) mutató iterátorokat a begin és end tagfüggvényekkel érhetjük el.

Kezdő érték

Tömbök adatainak feldolgozása



Tömbök adatainak feldolgozása

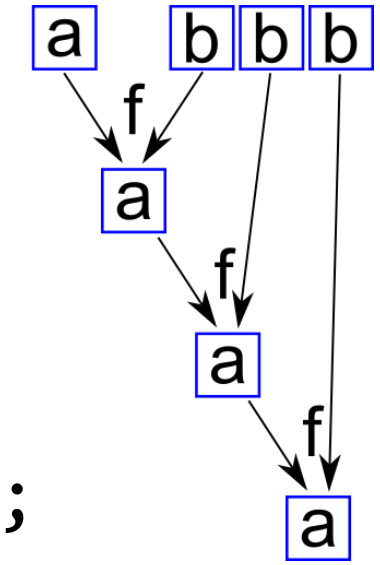
Példa: vector elemeinek a négyzetének az összege:

```
std::vector<int> B = {1, 2, 3};

auto f = [](int a, int b){ return a + b*b; };

int sqsum = std::accumulate(B.begin(), B.end(), 0, f);
```

$$\sum_i b_i^2$$

Tömbök adatainak feldolgozása

Példa: vector elemeinek a négyzetének az összege:




```
std::vector<int> B = {1, 2, 3};

auto f = [](int a, int b){ return a + b*b; };

int sqsum = std::accumulate(B.begin(), B.end(), 0, f);
```

Az f függvény lesz hattatva a kezdő értékre (most 0) és az első elemre, majd az így kapott eredményre és a 2. elemre, stb…, amíg a teljes tömb el nem fogyott.

Tömbök adatainak feldolgozása


```cpp
#include <iostream>
#include <vector>
#include <numeric>

int main()
{
  std::vector<int> B = {1, 2, 3};
  auto f = [](int acc, int x){ return acc + x*x; };
  auto sqsum = std::accumulate(B.begin(), B.end(), 0, f);
  std::cout << "sqsum: " << sqsum << "\n";
  return 0;
}
```

Tömbök adatainak feldolgozása

Példa: a vector adott hatványra emelt elemeinek az összege:

```
std::vector<double> B = {1.0, 2.0, 3.0};

double n = 4.0;

auto f = [n](double a, double b)
{
  return a + std::pow(b, n);
};

double s = std::accumulate(B.begin(), B.end(), 0.0, f);
```

$$\sum_i b_i^n$$

Tömbök adatainak feldolgozása

Példa: a vector adott hatványra emelt elemeinek az összege:

```
std::vector<int> B = {1.0, 2.0, 3.0};

double n = 4.0;

auto f = [n](double a, double b)
{
  return a + std::pow(b, n);
};

double s = std::accumulate(B.begin(), B.end(), 0.0, f);
```

Ha a lambdában akarunk használni egy értéket a külső scope-ból, akkor írjuk be a [ ] közé!

Feladat

Házi feladat:

Írjunk egyenes illesztést, ha adott két `std::vector<double>` (az x-ek és az y-ok), használjunk algoritmusokat (accumulate), figyeljünk a paraméterek átadására.

A visszatérési érték legyen `std::array<double, 2>`, aminek első eleme a meredekség, a második a tengelymetszet!

Legyen teszt eset!

Feladat

Emlékeztetőül:

$$m = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$

$$b = \bar{y} - m\bar{x}$$

ahol a felülvonás átlagot jelöl.

Szorgalmi: számoljuk ki algoritmusokkal r^2-et is. Link

Írjunk tételes listát, hogy hány féle képpen romolhat el az egész egyenes illesztő függvény a bemenő paraméterek függvényében.