# 10. gyakorlat

Késtem 15 percet

MyVector.cpp template verzióban át van írva, a kód állítólag fent lesz tanár úr honlapján.

## Még 3 (szerintem csak 2) gyakorlat

Utána ZH

## Friend + template

template**<** typename T **>**

class MyVector**{**

 friend std**::**ostream**&** **operator<<** **<>** **(**std**::**ostream**&** os**,** const MyVector**<**T**>&** v**);**

//...

**}**

Van egy üres $<>$, ahonnan elhagytuk a $T$-t.

Előtte deklarálni kell **operator<<**-t, amihez pedig deklarálni kell MyVector-t. És csak ez után fogja fel a fordító mit jelent az üres $<>$.

Tehát a teljes kód a megfelelő sorrendben:

template**<** typename T **>** class MyVector**;**

template**<**class T **>**

std**::**ostream**&** **operator<<(**std**::**ostream**&** os**,** const MyVector**<**T**>&** v**);**

template**<** typename T **>**

class MyVector**{**

 friend std**::**ostream**&** **operator<<** **<>** **(**std**::**ostream**&** os**,** const MyVector**<**T**>&** v**);**

//...

**}**

ZH-n valószínűleg nem lesz friend, de lesz template.

## Alap érték megadható template típusnévnek is

template**<** typename T **=** int **>** class MyVector**;**

…

MyVector**<>** a**;** // működik, <int> lesz

MyVector b**;** // <> nem hagyható el.

## Standard Template Library (STL)

Céljai:

* Általános, hatékony szolgáltatások
* Jól használható konténerek, algoritmusok
* Kiterjeszthető és újrafelhasználható

Részei:

* Konténerek
* Algoritmusok
* Iterátorok
* C-s könyvtárak
* Matematikai segédfüggvények
* String/dátum/stb. kezelés
* Típusinformációk
* Numerikus értékhatárokat megadó konstansok
* I/O típus/függvények

## Konténerek

Tárolók tipikus adatszerkezetekre.

## Algoritmusok

Gyakori feladat: Adatok felvétele konténerbe, majd egy algoritmus alkalmazása rajta. (Rendezés, keresés, csere…)

## Iterátorok

Hogy működjön a bejárás olyan konténereken is, ahol nincs például $\left[​\right]$ indexelő művelet. (ZH-n ezzel fogunk szenvedni az ötösért.)

## For\_each

template**<**class InputIterator**,** class Function**>**

Function for\_each**(**InputIterator first**,** InputIterator last**,** Function f**){**

 **for** **(** **;** first**!=**last**;** **++**first **)** f**(\***first**);**

 **return** f**;**

**}**

//void myfunction (int i) { cout << " " << i;}

for\_each **(**v**.**begin**(),** v**.**end**(),** myfunction**);**

## Functor

Olyan objektum, aminek típusán meg van írva az **operator()**.

Fent láttuk első functorunkat. (myfunction)

Még egy példa:

struct myclass **{**

 int counter**;**

 myclass**(**int n**=**0**):**counter**(**n**){}**

 void **operator()** **(**int i**)** **{++**counter**;** cout **<<** " " **<<** i**;}**

**}** myobject**;**

Ebből létre is hoztunk egy myobject nevű változót. Használata:

cout**<<(**for\_each **(**myvector**.**begin**(),** myvector**.**end**(),** myclass**())).**counter**;** //vagy myobject a myclass() helyett

## Tárolók között különbség van

* Szekvenciális konténerek
	+ <vector> (dinamikus tömb)
	+ <deque> (Kétvégű sor)
	+ <list> (Kétirányú láncolt lista)
* Asszociatív konténerek
	+ <set>, multiset (halmaz, multi-halmaz/zsák)
	+ <map>, multimap (asszociatív tömb tetszőleges kulccsal indexelve / szótár / kulcs-érték párok halmaza / STL-ben bináris faként van implementálva)
* Konténer adapterek
	+ <stack> (Verem, vektorra vagy kétvégű sorra épül)
	+ <queue>, priority\_queue (Sor, prioritásos sor)

ZH-n ilyen konténer adapter írása lesz a feladat. Felveszünk egy már létező konténert és kifelé úgy implementáljuk, hogy (…) befelé a megfelelő műveleteket hívjuk.

## Tárolók elve

Van egy közös művelet halmaz, ami a legtöbb konténerre megvan. (pl. Méret lekérdezése.) Konvenció, hogy mindben benne van.

De a konténerek maguk nem közös ősből származnak, nincs is közös ős.

### Például a vektor

template **<** class T**,** class Allocator **=** allocator**<**T**>** **>** class vector**{**

//méret

 size**();**

 max\_size**();**

 clear**();**

 resize**();**

//elemlekérdezés

 **operator[](**index**);** //const verzió is

 at**(**index**);** //const verzió is

 back**();** //legelső elem

 front**();** //utolsó elem

//módosító

 push\_back**(**item**);**

 pop\_back**();**

 insert**(**iterator/\*, iterator\*/**);**

 erase**(**iterator/\*, iterator\*/**);**

**}**

### Bejáráshoz

iterator begin**();**

const iterator begin**()** const**;**

iterator end**();**

const iterator end**()** const**;**

Míg a $begin$ simán az elejére mutat, az $end$ eleje után eggyel. Mert a legtöbb algoritmusnak így jó.

A front az első elem, a back az utolsó elem.

Visszafelé bejárás, ha szövegben egy betű utolsó előfordulására vagyunk kíváncsiak: reverse\_iterator rbegin**();** rend**();**

De ezzel vigyázzunk, becsapós lehet. Nem ---olni kell az iterátort, hanem ++-olni.

Viszont ha vissza akarjuk alakítani sima iterátorrá: it**=**rb**.**base**();** it**--;**//Lesz a talált elem

### Régen az iterátor csak egy pointer volt vektorban

**typedef** T**\*** iterator**;**

Minden konténer maga implementálja az iterátorait.

Konténer iterátorának használata:

typename vector**<**int**>::**iterator it**=**v**.**begin**();** //typename nélkül nem működik.

## Iterátorok származása

random access → bidirectional → forward → input / output

Az input/output iterátor már meglehetősen korlátozott. Van rá ++ művelet, de csak lineárisan tud dolgozni.

Forward tud mindent, amit az i/o. Csak előre haladhat.

Bidirectional már tud ---olni is.

Random mindent tud, amit az előzőek, plusz pointer-arithmetika-szerű műveleteket.

Mostantól a ZH-ra készülünk.

Gyakorlat vége.

## Gyakorlat után még elhangzott

Eddig csak két pluszminuszt írtunk. ⇒ Pluszminusz lesz a következő mindkét gyakorlaton.

A ma kivetített kódok állítólag fent lesznek tanár úr honlapján.