# 11. előadás – STL

## Vizsgára lehet jelentkezni

Vizsgalapon nem volt fent egy csoport péntekig, a többinek működött. Most már mindenkinek jó.

## ZH-hoz

ZH könnyebb lesz, ha ismerjük az STL adatszerkezeteit és műveleteit. (Van másolás, felszabadítás, és nem kell pointereket debuggolni.)

Referenciaoldalak használata szabad. Sőt, teljes internetelérés lesz. De ne csaljunk!

Javasolt referencia: <http://www.sgi.com/tech/stl/>

## Standard Template Library – STL

Előre megírt szabványkönyvtárként több-kevesebb dolgot adnak a programozási nyelvek.

C++ nem ad operációs rendszertől függő könyvtárat, adatbázist sem. Természetesen adatbázis motorhoz lehet kapcsolódni, GUI-s alkalmazást is lehet írni, de operációs rendszerhez kapcsolódó könyvtárat kell használni.

Az új '11-es szabvány kibővíti ezt a könyvtárat. De még most sincs GUI, socket, stb. könyvtár.

Az STL nem implementáció, hanem leírás. Többféle implementációja is van. (pl. Microsoftos STL verzió)

Nagyrészt template-ekből van felépítve.

## STL komponensei

* Adatszerkezetek (konténerek)
  + Kétszeresen láncolt lista (Már egyirányú lista is van az új szabványban)
  + Vektor Memória-kiosztása ugyan olyan, mint egy C tömb, de bővíthető, kényelmesebben használható
* Algoritmusok (Gyakori feladatokra ad megoldást, konténer-függetlenül működik)

## Az STL mindkét irányba bővíthető, párhuzamosan

* Írhatunk új algoritmust, ami működik a már meglevő konténerekkel
* Vagy új konténert, amin működik az összes eddigi meglévő algoritmus

## Példa: Tömbben keressünk meg egy x értéket, adjunk rá vissza pointert

int**\*** find**(**int**\*** first**,** int**\*** last**,** int x**){**

**while(**first**!=**last**){**

**if(\***first**==**x**){**

**return** first**;**

**}**

**++**first**;**

**}**

**return** 0**;**

**}**

Ez nem elég általános. Javítsuk:

## Template: tömbben keressünk meg egy x értéket, adjunk rá vissza pointert

A nem kell, hogy osztály legyen, lehet típus is. C++-ban a kulcsszó ezzel ekvivalens, és rövidebb.

template **<**typename T**>** //typename === class

T**\*** find**(**T**\*** first**,** T**\*** last**,** const T**&** t**){** //Nem akarjuk lemásolni, ezért konstans referencia szerinti paraméterátadást választunk

**while(**first **!=** last**){**

**if(\***first **==** t**){**

**return** first**;**

**}**

**++**first**;**

**}**

**return** 0**;**

**}**

Ez még mindig tömbökön dolgozik. Vezessünk be még egy template paramétert

## Template: konténerben keressünk meg egy x értéket, adjunk rá vissza pointert

template **<**typename T**,** typename Iterator**>**

Iterator find**(**Iterator first**,** Iterator last**,** const T**&** t**){**

**while(**first **!=** last**){**

**if(\***first **==** t**){**

**return** first**;**

**}**

**++**first**;**

**}**

**return** last**;** //Már nem egy null iterátort adunk vissza, hanem a last objektumot, ami extremális, "utolsó utáni" eleme a lekérdezésnek

**}**

## Fenti kódrészlet használata

std**::**vector**<**int**>** v**;**

std**::**vector**<**int**>::**iterator i **=** std**::**find**(**v**.**begin**(),** v**.**end**(),** 5**;**

## Intervallumok az STL-ben

Mindig balról zárt jobbról nyílt:

Az mindig az utolsó utáni elemre vonatkozik.

## Általánosítsuk még a fenti kódrészletet!

El lehet vonatkoztatni még egy paramétertől: Ne egy -et vagy -t keressünk, hanem az első tulajdonságú elemet!

template **<**class Iterator**,** class Pred**>**

Iterator find\_if**(**Iterator first**,** Iterator last**,** Pred p**){** //Azért más a neve, mert nem lehet ugyanolyannak tűnő template paraméterekkel túlterhelni a find-ot. Ha lennének conceptek a nyelveben, akkor lehetne.

**while(**first **!=** last**){**

**if(** p**(\***first**)** **){**

**return** first**;**

**}**

**++**first**;**

**}**

**return** last**;**

**}**

Nem vár el a típustól egyenlőségvizsgálatot. Viszont -től elvár egy operátort. (Tehát vagy funkció pointer, vagy Functor.)

STL-ben gyakori a functor.

## Functor példa

struct X**{**

bool **operator()(**int x**)** const**{**

**return** x**<**5**;**

**}**

**};**

Ez használható a fenti template paraméternek:

std**::**list**<**int**>** c**;**

std**::**list**<**int**>::**iterator it **=** std**::**find\_if**(**c**.**begin**(),** c**.**end**(),** X**()** **);**

//X**()** Névtelen default konstruált objektum X-ből

"Funktor típus"

Functor még a rendezett adatszerkezetek azon template paramétere is, hogy milyen rendezéssel rendezzük.

Függvénypointert ugyanúgy kezeljük, mint a functort. De STL-ben a functort általában nem váltja ki a függvénypointer.

## Adatszerkezetek

* Szekvenciális konténerek (Egységes interfészt ad a szekvencialitásuk: push\_back, pop\_back)
  + (Kétszeresen láncolt lista)
  + (Itt van indexelés is, egybefüggő tárterületet használ, újat foglal le, ha betelik.)
  + (Két végén levő nyolcas tömbben gyors.)
  + (Új szabvány: array, egyszeresen láncolt lista)
* Asszociatív adatszerkezetek ()
  + Rendezett:
    - set, multiset
    - map, multimap
  + Rendezetlen:
    - unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap
* Adatpterek / adaptorok (Szekvenciális adatszerkezetet "butít le". Template paraméter, hogy melyik szekvenciális konténer segítségével legyen. Van alapértelmezett értéke a paraméternek, természetesen. deque?)
  + stack (Vektor, ami nem indexelhető és csak a teteje hozzáférhető)
  + queue, priority\_queue

## Hatékonyság: Lista méret lekérdezése vs. splice

Splice: Átláncolás. Egyik listarészt a másikba.

Ha a méretet tároljuk, a splice lesz lassú, mert minden átfűzésnél újra kell számolni a méretet.

Ha a méretet nem tároljuk, a splice gyors, de a méretlekérdezés lesz lassú. (Végig kell menni egyesével.)

## Iterátor hierarchia

Különböző típusú, képességű bejárók.

Listának kétirányú (bidirectional) bejárói vannak. Aktuális elemtől egyet lehet lépni: ++, --

Vektornak és deque-nek random access iterátora van. Indexelni is tudunk.

Algoritmusok tehetnek megszorításokat, hogy milyen képességű iterátort akarnak.

Gyakori hibaüzenet: Listát nem tudunk rendezni, mert a random access iterátort akar, de listán csak bidirectional iterátor van.

Van még input, output, és forward iterátor is.

Ezek az iterátor absztrakciók csak megegyezések, konvenciók. Nem nyelvi eszköz.

## Konténer által tárolt elem típusának lekérdezése, ha nincs hozzáférésünk -hez a template-ben

template **<**class Iterator**>**

void f**(**Iterator first**,** Iterator last**){**

typename std**::**iterator\_traits**<**Iterator**>::**value\_type a **=** **\***first**;**

...

**}**

Ekkor a konténer belső adatainak típusa.

## Template fordítás kétlépéses ( fontos)

int x**;**

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

//...

Foo**<**T**>::**N**\***x**;**

//...

**}**

A fenti kódrészletet szorzásnak veszi.

typename Foo**<**T**>::**N**\***x**;**

Így már típus.

## Konstans iterátor, fordított iterátor, konstans fordított iterátor

Rendes iterátor esetén -en keresztül felül lehet írni az elemet. Konstans iterátoron keresztül nem.

std**::**list**<**int**>::**const\_iterator

Fordított iterátor az utolsó egyezést fogja megtalálni.

std**::**set**<**int**>::**reverse\_iterator

A kettő tranzitív lezártja:.

std**::**deque**<**double**>::**const\_reverse\_iterator

Van még stream és másoló iterátor is.

## Map

Asszociatív tömb.

Első paramétere egy kulcs típus, ami szerint rendezetten lehet benne keresni és e szerint indexelni.

std**::**map**<**std**::**string**,**int**>** s**;**

Természetesen a string is az STL része. Közel áll a vektorhoz.

s**[**"blablabla"**]** **=** S**;** //Hozzávesz vagy módosít, nincs is konstans verziója

Van tagfüggvénye: s**.**find**(**"abc"**);**, ami ki tudja használni a rendezettséget. (string esetén lexikografikusan.)

Érdemes a tagfüggvényt használni, nem a globálisat. Például listának is van rendező tagfüggvénye, ami nem adja a fenti hibát.

## Generikus programozás

Öröklődés lehet nem lesz, majd megtanuljuk Java-ból.

(Generikus programozás be volt ígérve, de nem került rá sor csak 1 mondat az elején.)

Előadás vége.