# 9. előadás

## Jó hír: Működik a Neptun.

De a tanár úr nem tudja használni.

## Beadandó (opcionális)

Sudoku megoldás ellenőrzése

<http://patakino.web.elte.hu/pny2/beadando.html>

# Folytatjuk a múlt órai anyagot

## Beolvasó és kiíró operátor

Header:

std**::**istream**&** **operator>>(** std**::**istream**&** is**,** date**&** d**);**

std**::**ostream**&** **operator<<(** std**::**ostream**&** os**,** const date **&**d**);**

Source:

istream**&** **operator>>(** istream**&** is**,** date**&** d**){**

 d**.**get**(** is**);**

 **return** is**;**

**}**

ostream**&** **operator<<(** ostream**&** os**,** const date **&**d**){**

 d**.**put**(** os**);**

 **return** os**;**

**}**

### Magyarázat

Ez az $ostream$ típus operátora lenne, (std**::**cout**.operator<<(**d**);**), de ezzel az a gond, hogy másik gépre átvisszük, a szabványkönyvtárat is át kell vinni. Nem akarjuk beírni a szabványkönyvtárba.

Tehát ez nem jó megoldás tagfüggvényként. Helyette globális művelet lesz.

Beolvasás nem konstans referencia szerint veszi át az objektumot, mert meg akarjuk változtatni.

## Mi az a this paraméter

date**&** set\_year**(**int y**)** **{** year **=** y**;** **return** **\*this;** **}**

Meghívjuk d-n, (d**.**set\_year**(**2012**);**), akkor ezzé alakul: set\_year**(&**d**,**int y**);**

Típusa: date**\*** **this**

Vagy: const date**\*** **this**

Megjegyzés: Túlterhelhetőek a függvények const-on keresztül.

## Implementáció

|  |
| --- |
| Implementáció |
| #include <iostream>#include <cstring>#include <cstdlib>#include "date.h"**using** **namespace** std**;**/\* konstruktor \*/date**::**date**(** const char **\***s**){** /\* strchr és strrchr a <cstring>-bõl \*/ char **\***p1 **=** strchr**(** s**,** '.'**);** char **\***p2 **=** strrchr**(** s**,** '.'**);** **if** **(** p1 **&&** p2 **&&** p1 **!=** p2 **)** **{** /\* atoi a <cstdlib>-bõl> \*/ int y **=** atoi**(**s**);** int m **=** atoi**(**p1**+**1**);** int d **=** atoi**(**p2**+**1**);** set**(** y**,** m**,** d**);** **}****}**/\* módosítók \*/date**&** date**::**next**(){** static int day\_in\_month**[]** **=** **{** 31**,** 28**,** 31**,** 30**,** 31**,** 30**,** 31**,** 31**,** 30**,** 31**,** 30**,** 31 **};** **++**day**;** /\* TODO: szökõév \*/ **if** **(** day**-**1 **==** day\_in\_month**[**month**-**1**])** **{** day **=** 1**;** **++**month**;** **}** **if** **(** 13 **==** month **)** **{** month **=** 1**;** **++**year**;** **}** **return** **\*this;****}**date**&** date**::**add**(** int n**){** **for** **(**int i **=** 0**;** i **<** n**;** **++**i**)** next**();** **return** **\*this;****}**/\* input-output \*/void date**::**get**(** std**::**istream**&** is**){** // nem lenne igazán jó: // is >> year >> month >> day; int y**,** m**,** d**;** **if** **(** is **>>** y **>>** m **>>** d **)** **{** set**(** y**,** m**,** d**);** **}****}**void date**::**put**(** std**::**ostream**&** os**)** const**{** // nem lenne igazán jó: os << year << month << day; os **<<** "[ " **<<** get\_year**()** **<<** "," **<<** get\_month**() <<** "," **<<** get\_day**()** **<<** " ]"**;****}**/\* globális mûveletek \*/bool **operator<(** date d1**,** date d2**){** **return** d1**.**get\_year**()** **<** d2**.**get\_year**()** **||** d1**.**get\_year**()** **==** d2**.**get\_year**()** **&&** d1**.**get\_month**()** **<** d2**.**get\_month**()** **||** d1**.**get\_year**()** **==** d2**.**get\_year**()** **&&** d1**.**get\_month**()** **==** d2**.**get\_month**()** **&&** d1**.**get\_day**()** **<** d2**.**get\_day**();****}**istream**&** **operator>>(** istream**&** is**,** date**&** d**){** d**.**get**(** is**);** **return** is**;****}**ostream**&** **operator<<(** ostream**&** os**,** const date **&**d**){** d**.**put**(** os**);** **return** os**;****}** |

## Stringből konstruktor

Megnézzük két különböző pont karakter szerepel a paraméterben…

## Next

Statikus tömbbe felvesszük, melyik hónapban hány nap van.

Napot növeljük, megnézzük átléptünk-e a következő hónapba.

Ha igen, hónapot növeljük és napot állítjuk egyre.

Majd megnézzük hónapot léptünk-e, ekkor évet növelünk és hónapot 1-re.

## Add​(int n)

Ciklusban annyiszor hívjuk a $Next\left(​\right)$-et, ahánnyal növelni akarjuk.

Nem túl optimális megoldás, de működik. Sőt, ha megváltozik a specifikáció, nem kell módosítani. Átírás nélkül működik.

## Get

Beolvas egy dátumot, három szám alakjában.

Nem célszerű közvetlen a belső adattagokba beolvasni. Inkább változóba olvassuk, ellenőrizzük sikerült-e beolvasni mindhármat, majd set**(**y**,**m**,**d**);**. Így csak akkor módosítja az objektumot, ha minden beolvasás sikerült.

Példa: Félúton kapunk egy $EOF$-ot. Ekkor elromlik az eredeti dátum, de az új dátum nem tudott létrejönni adathiány miatt.

## Put

Kiírás, közte vesszőkkel. Itt sem jó az adattagokat direktbe kiírni. Mert mi van, ha később $unix time stamp$ formában tároljuk dátumunkat, hogy kevesebb memóriát használjunk (1 long vs. 3 int).

// nem lenne igazán jó: os << year << month << day;

os **<<** "[ " **<<** get\_year**()** **<<** "," **<<** get\_month**()** **<<** "," **<<** get\_day**()** **<<** " ]"**;**

Itt sem kell **this**-en keresztül meghívni a get\_month**()** stb. függvényeket.

## Kisebb

Itt nem is tudjuk az adattagokat közvetlen elérni, muszáj volt meghívni a get\_month**()** stb. függvényeket.

bool **operator<(** date d1**,** date d2**){**

 **return** d1**.**get\_year**()** **<** d2**.**get\_year**()** **||**

 d1**.**get\_year**()** **==** d2**.**get\_year**()** **&&** d1**.**get\_month**()** **<** d2**.**get\_month**()** **||**

 d1**.**get\_year**()** **==** d2**.**get\_year**()** **&&** d1**.**get\_month**()** **==** d2**.**get\_month**()**

 **&&** d1**.**get\_day**()** **<** d2**.**get\_day**();**

**}**

## Getter-ek nem rosszak

Mert inline módon fordulnak le.

# Verem adatszerkezet

## Header

|  |
| --- |
| Header |
| #ifndef DSTACK\_H#define DSTACK\_H#include <iostream>class dstack**{** friend std**::**ostream **&operator<<(** std**::**ostream **&**os**,** dstack ds**);**public**:** dstack**(** int size **=** 128**);** dstack**(** const dstack **&**other**);** **~**dstack**();** dstack**&** **operator=(** const dstack **&**other**);** void push**(** double d**);** double pop**();** bool is\_empty**()** const**;** bool is\_full**()** const**;**private**:** int capacity**;** int sp**;** double **\***v**;** void copy**(** const dstack **&**other**);****};**std**::**ostream **&operator<<(** std**::**ostream **&**os**,** dstack ds**);**#endif /\* DSTACK\_H \*/ |

## A friend operátor

/\* friend deklaráció: nem helyettesíti a függvény deklarációt \*/

friend std**::**ostream **&operator<<(** std**::**ostream **&**os**,** dstack ds**);**

/\* a kiíró mûvelet deklarációja (friend nem elég) \*/

std**::**ostream **&operator<<(** std**::**ostream **&**os**,** dstack ds**);**

## Konstruktor

dstack**::**dstack**(** int size**){**

 v **=** **new** double**[**capacity **=** size**];** \*

 sp **=** 0**;**

**}**

\* Ez ugyanaz, mint:

capacity **=** size**;**

v **=** **new** double**[**capacity**];**

## Pop

double dstack**::**pop**(){** **return** is\_empty**()** **?** 0.0 **:** v**[--**sp**];}**

Hiba: Honnan tudjuk, hogy az utolsó elem 0 vagy üres?

## Üres

// a szabványos könyvtárban: bool empty() const

bool dstack**::**is\_empty**()** const**{** **return** 0 **==** sp**;}**

## Kiírás

std**::**ostream **&operator<<(** std**::**ostream **&**os**,** dstack ds**){**

 os **<<** "[ "**;**

 **for** **(** int i **=** 0**;** i **<** ds**.**sp**-**1**;** **++**i **){**os **<<** ds**.**v**[**i**]** **<<** ", "**;}**

 **if** **(** ds**.**sp **>** 0 **){**os **<<** ds**.**v**[**ds**.**sp**-**1**];}**// hogy ne írjunk vesszőt az utolsó elem után

 os **<<** " ]"**;**

 **return** os**;**

**}**

## Alapértelmezett copy konstruktor nem jó

Tagonként másol, de mivel volt egy pointer a tárterületre (double **\***v**;**), a másolat ugyan azon a tárterületen fog osztozni, mint az eredeti. Ezt kijavíthatjuk:

Felüldefiniálható az értékadás. Ezért jó, hogy az értékadás operátor.

### Copy konstruktor

dstack**(** const dstack **&**other**);**

dstack**::**dstack**(** const dstack **&**other**){** copy**(**other**);}**

void dstack**::**copy**(** const dstack **&**other**){**

 v **=** **new** double**[**capacity **=** other**.**capacity**];**

 sp **=** other**.**sp**;**

 **for** **(** int i **=** 0 **;** i **<** sp**;** **++**i**){**v**[**i**]** **=** other**.**v**[**i**];}**

**}**

Copy konstruktor nem veheti át érték szerint a paraméterét. Végtelen rekurziót okozna, önmagán keresztül próbálná hívni a copy konstruktort, ha érték szerint próbálná átvenni a másolandót.

### Értékadó operátor

dstack**&** **operator=(** const dstack **&**other**);**

dstack**&** dstack**::operator=(** const dstack **&**other**){**

 **if** **(** **this** **!=** **&**other **){** // x = x

 // nem igazán kivétel-biztos

 **delete** **[]** v**;**

 copy**(**other**);**

 **}**

 **return** **\*this;**

**}**

Meg kell vizsgálni, önértékadás történik-e. Mert először kitöröljük, majd átmásoljuk.

dstack d**(**10**);**

d**=**d**;**//önértékadás

dstack**&** f **=** d**;**

d**=**f**;**//ez is önértékadás

Itt egyébként nem lenne végtelen rekurzió, ha érték szerint adjuk át. De azért érdemes referencia szerint.

## Destruktor

**~**dstack**();** //headerbe

dstack**::~**dstack**(){** **delete** **[]** v**;}** //cpp-be

Lokális blokk végén (élettartam végén) automatikusan meghívódik a destruktor. Ekkor nem kell a felhasználónak külön figyelnie, hogy mindig felszabadítsa a memóriát.

Ezért érdemes mindig destruktort írni.

Destruktornak nem lehet paramétere. Nem is lehet túlterhelni.

## Új C++'11szabványban le lehet tiltani explicit módon az értékadást, másolást.

$move$ szemantika.

Temporális változók: Gyakran másoljuk, majd töröljük az eredetit. Ezt gyorsítja a $move$.

Előadás vége.