# 10. előadás

## Sablonok (template)

ZH-n lesz.

Más nyelvekben generic ("generikus") és nem feltétlen pont ilyen funkciókkal.

Típussal történő paraméterezés.

C-ben ez hiányzott, workaround kellett rá. (#define makró, void**\*** pointertípus, int printf**(**char**\*** s**,...);**)

## Swap

C-ben nem lehetett, mert preprocesszor makró paraméterének típusát nem tudjuk lekérdezni.

template **<**class T**>** void swap **(**T**&** a**,** T**&** b**){**

 T tmp**=**a**;** a**=**b**;** b**=**tmp**;**

**}**

Paraméter dedukció, → fordító generál egy swap függvényt a megfelelő típussal. (Példányosítás.)

Működni fog felhasználói típusokra is.

## Példányosítás típusok

'03: Függvények és típus sablonok

Osztályok

'11: typedef (szinonimák)

## Osztályokra nincs automatikus dedukció, van helyette List<int>

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

class List**{**

 //...

**}**

Példányosítás:

List**<**int**>** a**;**

Lazy instantiation: csak azt példányosít, amit muszáj.

## Template hibaüzenetek néha nehezen érthetőek

$STL$-es $list$ rendezése például ilyen, mert csak előző/következő iterátor van rá, nem lehet pl. 5-tel előre ugrani.

## Specializációk

Általános template-et néha érdemes bizonyos típusokra megváltoztatni, testre szabni.

Osztály specializációk leginkább. Van részleges és teljes specializáció.

Alap mátrix template: template **<**class T**>** class Matrix**{**/\*...\*/**}**

Részleges specializáció: template **<**class T**>** class Matrix **<**T**\*>{**/\*...\*/**}**

Teljes specializáció: template **<**class T**>** class Matrix **<**bool**>{**/\*...\*/**}**

Lehet rekurzív template-eket is írni.

## Még egy osztály template feature

Az std**::**set rendezetten tárolja az adatokat.

template **<**class T**,** class Comp **=** std**::**less**<**T**>>** class set**{**/\*...\*/**}**

Most már lehet szóköz nélkül template-et lezárni. (Régen **>** **>** kellett, ma már nem téveszti össze a **>>**-t beolvasás operátorral a preprocesszor.

A második paraméter használata:

std**::**set**<**int**>** a**;**

std**::**set**<**int**,** std**::**greater**<**int**> >** b**;**

## Template paraméterek

* Definiálunk egy típus paramétert.
* Integrális konstansok szerepelhetnek: int, bool, stb. ($float$ már nem integrális! De két $int$ helyette működik.)
* Külső szerkesztésű függvényre, tagfüggvényre, objektumra mutató pointerek

## Fordítási időben faktoriálist számolni is lehet

template **<**int N**>** struct Factorial **{**

 enum**{**value **=** N **\*** Factorial**<**N **-** 1**>::**value**};**

**};**

template**<>** struct Factorial**<**0**>** **{**

 enum**{**value **=** 1**};**

**};**

//Használat:

std**::**cout**<<**Factorial**<**5**>::**value**;**

## Template paraméter nem olyan, mint a konstruktor paraméter

template **<**class T**,** int N**>** class Array **{**

 T v**[**N**];** // Fordításidőben ismert

 //Nem kell HEAP-pel foglalkozni, megszabadulunk a destruktor, copy konstruktor, és értékadó operátor megadásától

**}**

template **<**class T**>** class Vector **{**

 T**\*** p**;**

public**:**

 Vector**(**int n**){**

 p**=new** T**[**n**];**

 **}**

 //+kell destruktor, copy konstruktor, értékadó operátor...

**}**

Természetesen nem játszhatjuk ki az eddig ismert limitációkat: A méretnek fordításidőben ismertnek kell lennie.

(TODO: "limitáció" nem szó MS Word szerint?)

Továbbá Array**<**int**,**10**>** és Array**<**int**,**12**>** típusa különböző, nem másolhatunk egyikből másikba.

De Vector**<**int**>** v**(**10**)** és Vector**<**int**>** w**(**12**)** megy, csak nekünk kell megírni. Template copy konstruktor működhet, de figyeljünk a tömbméretekre.

Ha felveszünk egy harmadik template paraméter (inicializálás érték), akkor nem lesz implicit másolás, hiába int és 12 méretű mindkettő. Ekkor már jobb a $Vector$-t használni.

Előadás vége.