# Elemi Alkalmazások Fejlesztése II.

2. Osztályok

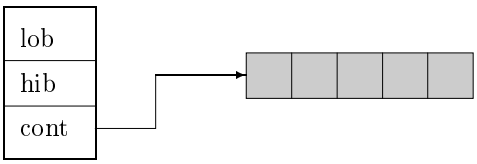
2002.09.26.

## 1. Feladat

Készítsünk egy olyan egészekkel indexelt egészeket tartalmazó vektor típust, amelynek indexhatárai a létrehozásakor adhatók meg! Elvárás, hogy az új típust úgy lehessen használni, mint a beépített tömb típust (indexelés), és kapjunk hibajelzést, ha a vektor használata során valamilyen hiba lép fel. Ezen kívül legyen lehetőség a vektor méretének, alsó és felső indexhatárának lekérdezésére!

## 2. Reprezentáció

Ábrázoljuk a vektorokat (`IntArray`) egy hármassal, amelynek komponensei: az alsóhatár (`lob`), a felsőhatár (`hib`) és egy beépített tömb (`cont`), amely tartalmazza az elemeket (1. ábra).




1. ábra: Az (`IntArray`) típus reprezentációja.

A tömb nullától indexelt, ezért a nulla felel meg az alsóhatárnak. Ha a tömb mérete megfelelő, akkor az utolsó index felel meg a felsőhatárnak. Az ábrázolandó vektorban az elemek száma $hib - lob + 1$, így egy ekkora tömbre van szükségünk. Ezt dinamikusan kell lefoglalni, mert az indexhatárok előre nem ismertek, ezeket a vektor definiálásakor adjuk meg.

## 3. Naív C++ megoldás

Az eddigiek alapján el tudjuk készíteni a `IntArray` osztály első változatát. Szükség lesz egy konstruktorra, amelynek paraméterei az indexhatárok, és lefoglalja a megfelelő méretű tömböt. A destruktor ezt a tömböt szabadítja fel. Ezen kívül kell három lekérdező művelet (`Size`, `Lob`, `Hib`) a feladatban szereplő lekérdezések miatt, és biztosítani kell még az indexelés lehetőségét. Ennek érdekében be kell

vezetni az indexelés operátort ([]), ami egy hivatkozást szolgáltat a megfelelő indexű elemre. Így az osztály definíciója a következő.

```
// IntArray.h: interface for the IntArray class.

#ifndef _INTARRAY_H
#define _INTARRAY_H__

class IntArray
{
public:
    IntArray(int low_bound, int high_bound);
    ~IntArray();
    int Lob() const   { return(lob); }
    int Hib() const   { return(hib); }
    int Size() const  { return(hib - lob + 1); }
    int &operator[](int index);
private:
    int    lob;
    int    hib;
    int    *cont;
};

#endif
```

Az osztálydefinícióban nem implementált műveletek megvalósítása a következő.

```
// IntArray.cpp: implementation of the IntArray class.

#include "IntArray.h"

IntArray::IntArray(int low_bound, int high_bound)
{
    lob = low_bound; hib = high_bound;
    if ( hib < lob )   ; // hiba!
    cont = new int[hib - lob + 1];
}

IntArray::~IntArray()
{
    delete [] cont;
}

int &IntArray::operator [](int index)
{
    if ( index < lob || index > hib )   ; // hiba!
    return(cont[index - lob]);
}
```

Használjuk az elkészült osztályt egy maximumkiválasztásban.

```
#include <iostream.h>
#include "IntArray.h"

void maxker(IntArray x, int &h, int &m)
{
    int i = x.Lob();
    h = i; m = x[i];
    for ( i = x.Lob() + 1; i <= x.Hib(); i++ )
    {
        if ( x[i] > m )
        {
            h = i; m = x[i];
        }
    }
}

int main()
{
    IntArray v(-3, 8);
    int      hely, ertek;

  // A vektor feltöltése elemekkel

    maxker(v, hely, ertek);
    cout << "A maximum: " << ertek << endl;
    cout << "Indexe: " << hely << endl;
    return(0);
}
```

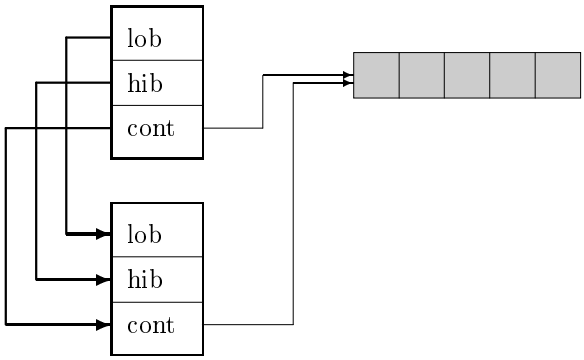
# 4. Copy konstruktor

Az előző megoldásban a paraméterátadás során értékszerinti paraméterátadást használtunk a vektor esetében. Ennek megfelelően egy másolat készül a maximumkiválasztás hívásakor.

A jelenlegi változatban az alapértelmezett másoló művelet (copy konstruktor) használható. Ez azonban nem megfelelő, mert az adattagokat átmásolja, de a tömb esetében ez csak a tömb címének másolását jelenti (2. ábra), ami nyilvánvalóan nem megfelelő.

Ezt a problémát úgy oldhatjuk meg, ha megírjuk az osztály saját copy konstruktorát, ami a megfelelő másolást hajtja végre. Ezután ugyanis már ez a másoló művelet kerül végrehajtásra a paraméterátadáskor.

A másolás során indexhatárok átírását követően létre kell hozni egy megfelelő tömböt majd oda elemenként át kell másolni az értékeket.




2. ábra: A default copy konstruktor viselkedése.

```
// IntArray.h: interface for the IntArray class.

#ifndef _INTARRAY_H
#define _INTARRAY_H__

class IntArray
{
public:
    IntArray(int low_bound, int high_bound);
    ~IntArray();
    IntArray(const IntArray &src);         // copy konstruktor
    int Lob() const   { return(lob); }
    int Hib() const   { return(hib); }
    int Size() const  { return(hib - lob + 1); }
    int &operator[](int index);
private:
    int    lob;
    int    hib;
    int    *cont;
};

#endif

// IntArray.cpp: implementation of the IntArray class.

IntArray::IntArray(const IntArray &src)
{
    lob = src.lob; hib = src.hib;
    cont = new int[hib - lob + 1];
    for ( int i = 0; i < hib - lob + 1; i++ )
        cont[i] = src.cont[i];
}
```

## 5. Az indexelés operátor konstans változata

Ha a maximumkiválasztás során hivatkozás szerint adtuk volna át a vektort, akkor egy másik problémába ütköztünk volna. Miután az eljárás nem változtatja a vektor, ezérta paraméterlistában ezt jelölnünk illene a `const` minősítéssel. Azaz az eljárás a következőképpen nézne ki.

```
void maxker(const IntArray &x, int &h, int &m);
```

Ebben az esetben a program nem fordul le, mert konstans vektorra nem alkalmazható az általunk definiált indexelés operátor, ugyanis az megengedné egy elem átírását, azaz a vektor megváltoztatását.

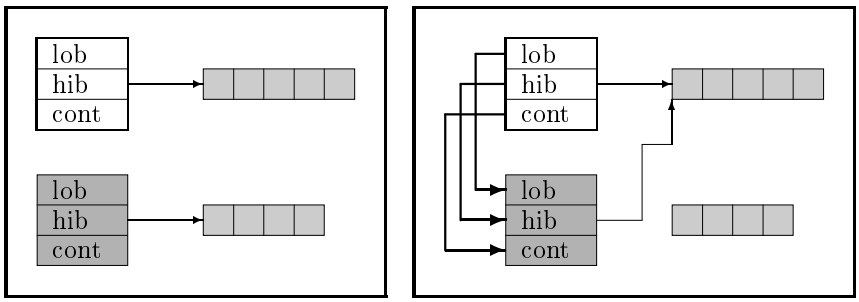
Ezt úgy oldhatjuk fel, ha bevezetünk még egy indexelés operátort (túlterhelés), amelyben már nem írhatjuk át az értéket csak lekérdezhetjük. Ennek módja a következő.

```
class IntArray
{
public:
 ...
    int &operator[](int index);
    int operator[](int index) const;
 ...
};

int IntArray::operator [](int index) const
{
    if ( index < lob || index > hib )  ; // hiba
    return(cont[index - lob]);
}
```

## 6. Értékadás

Vizsgáljuk meg mi történik, ha egy vektort egy másiknak adunk értékül! Az alapértelmezett értékadó operátor kerül végrehajtásra, ami az adattagokat átmásolja (3. ábra). Ez ugyanúgy elfogadhatatlan, mint a másolás esetén volt.




3. ábra: Az alapértelmezett értékadás.

A megoldás most is egy saját értékadó operátor elkészítése. Itt egy kicsit bonyolultabb a helyzet, mint a másolás esetén volt. Akkor ugyanis biztosan egy új objektumot hoztunk létre. Most ez nem áll fenn, ezért fel kell szabadítani a jelenlegi tömb területét. Ezt azonban óvatosan tehetjük csak meg, mert lehet, hogy valamit önmagának akarunk értékül adni (`x=x`). Ha ebben az esetben felszabadítanánk a tömb területét nem lenne honnan másolnunk. Szerencsére ebben az esetben semmit sem kell tennünk, csak meg kell vizsgálnunk, hogy fennáll-e. Erre szolgál a `this` operátor, ami az aktuális objektumra mutat. Ha ez megegyezik a paraméterrel, akkor semmit sem kell tennünk.

Annak érdekében, hogy a C++ nyelvben megszokott módon `x=y=z;` megengedett legyen, az értékadás operátornak vissza kell adnia egy hivatkozást az objektumra. Ezt megtehetjük a `return(*this)` utasítással.

```
class IntArray
{
public:
 ...
    IntArray &operator=(const IntArray &src);
 ...
};

IntArray &IntArray::operator=(const IntArray &src)
{
    if ( this == &src ) return(*this);
    delete [] cont;
    lob = src.lob; hib = src.hib;
    cont = new int[hib - lob + 1];
    for ( int i = 0; i < hib - lob + 1; i++ )
        cont[i] = src.cont[i];
    return(*this);
}
```

# 7. Kivételek

Térjünk vissza az esetleges hibák kezelésére, amelyekre eddig csak megjegyzésekkel utaltunk a programban. Vegyük észre, hogy a „szokásos” módszer, miszerint egy logikai visszatérési érték vagy egy extremális érték jelzi a hibát, most nem alkalmazható, ugyanis a konstruktornak nincs visszatérési értéke, az indexelés esetén pedig bármilyen egész szám szerepelhet a tömbben, ezért nincs extremális elem.

Az ilyen helyzetek kezelésére szolgálnak C++-ban a kivételek. A kivételek egy-egy osztály példányának feleltethetők meg, amelyeket a kivételes helyzet bekövetkezésekor lehet kiváltani, dobni (`throw`). Az osztály attribútumai írhatják le a helyzetet.

Egy kiváltott kivételt el lehet kapni (`catch`), ha azokat az utasításokat, amelyek a kivételt kiválthatják egy `try` blokkba helyezzük el.

A példánkban két hibalehetőségre készülhetünk fel:

- a konstruktorban az indexhatárok rosszak (`InvalidBound`),

- az indexelés során az indextartományon kivül eső indexet adunk meg (`IndexOutOfRange`).

Ennek megfelelően két beágyazott osztályt kell definiálnunk. Az első egy üres osztály, mert nem kell semmilyen információ az első esetben, a második egy adattagot tartalmazo osztály, mert ekkor szeretnénk megtudni az indexet. A második esetben a konstruktor biztosítja, hogy a kiváltáskor az adattag a kívánt értéket vegye fel.

Ezek után a teljes osztály és egy egyszerű főprogram a következő.

```
// IntArray.h: interface for the IntArray class.

#ifndef _INTARRAY_H
#define _INTARRAY_H__

class IntArray
{
public:
    IntArray(int low_bound, int high_bound);
    ~IntArray();
    IntArray(const IntArray &src);
    IntArray &operator=(const IntArray &src);
    int Lob() const     { return(lob); }
    int Hib() const     { return(hib); }
    int Size() const    { return(hib - lob + 1); }
    int &operator[](int index);
    int operator[](int index) const;

    class InvalidBound{};
    class IndexOutOfRange
    {
    public:
        int    index;
        IndexOutOfRange(int i)  { index = i; }
    };
private:
    int        lob;
    int        hib;
    int        *cont;
};

#endif
```

```
// IntArray.cpp: implementation of the IntArray class.

#include "IntArray.h"

IntArray::IntArray(int low_bound, int high_bound)
{
    lob = low_bound; hib = high_bound;
    if ( hib < lob )    throw InvalidBound();
    cont = new int[hib - lob + 1];
}

IntArray::~IntArray()
{
    delete [] cont;
}

IntArray::IntArray(const IntArray &src)
{
    lob = src.lob; hib = src.hib;
    cont = new int[hib - lob + 1];
    for ( int i = 0; i < hib - lob + 1; i++ )
        cont[i] = src.cont[i];
}

IntArray &IntArray::operator=(const IntArray &src)
{
    if ( this == &src )    return(*this);
    delete [] cont;
    lob = src.lob; hib = src.hib;
    cont = new int[hib - lob + 1];
    for ( int i = 0; i < hib - lob + 1; i++ )
        cont[i] = src.cont[i];
    return(*this);
}

int &IntArray::operator [](int index)
{
    if ( index < lob || index > hib )
        throw IndexOutOfRange(index);
    return(cont[index - lob]);
}

int IntArray::operator [](int index) const
{
    if ( index < lob || index > hib )
        throw IndexOutOfRange(index);
    return(cont[index - lob]);
}
```

```
// Tomb.cpp: main.

#include <iostream.h>
#include "IntArray.h"

int main( void )
{
    IntArray    x(10, 20);
    cout << "x.lob = " << x.Lob() << endl;
    cout << "x.hib = " << x.Hib() << endl;
    try
    {
        x[11] = 8;
        cout << x[11] << endl;
        x[5] = -1;
        cout << x[5] << endl;
    } catch (IntArray::IndexOutOfRange &ex)
    {
        cout << "Invalid index: " << ex.index << endl;
    }
    return(0);
}
```