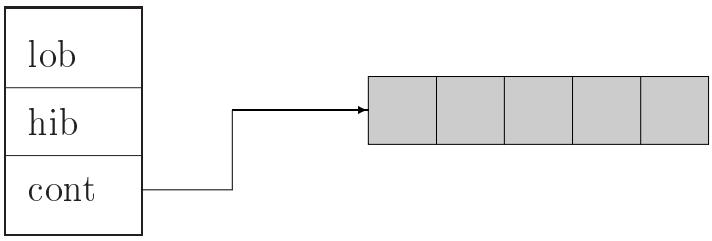
1. Feladat

észítsünk 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.



3. Naív C++ megoldás

```
// 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
```

```
// 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];
        }
    }
}
```

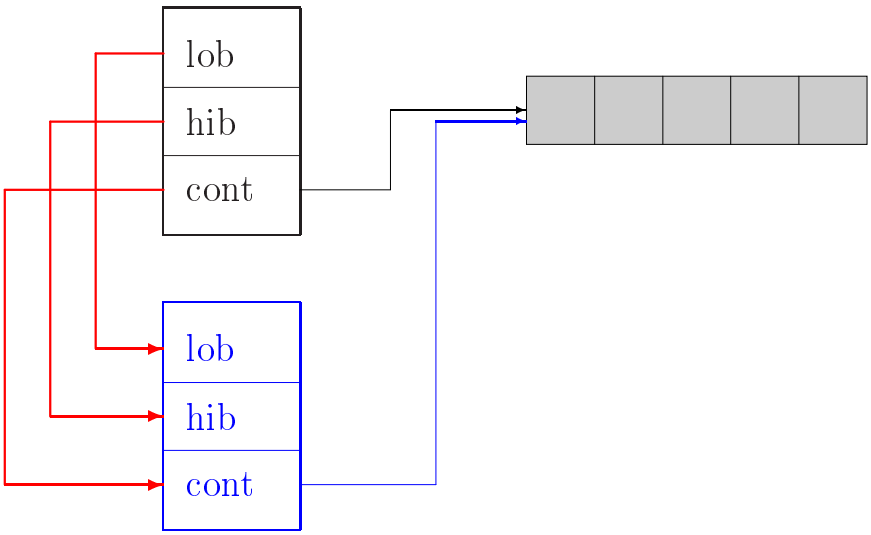
```
int main()
{
    IntArray v(-3, 8);
    int      hely, ertek;

// 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.



```
// 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);
```

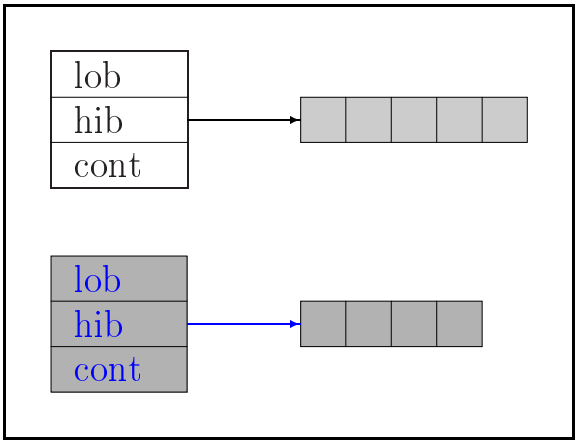
Ebben az esetben nem alkalmazható az általunk definiált indexelés operátor, ugyanis az megengedné egy elem átírását.

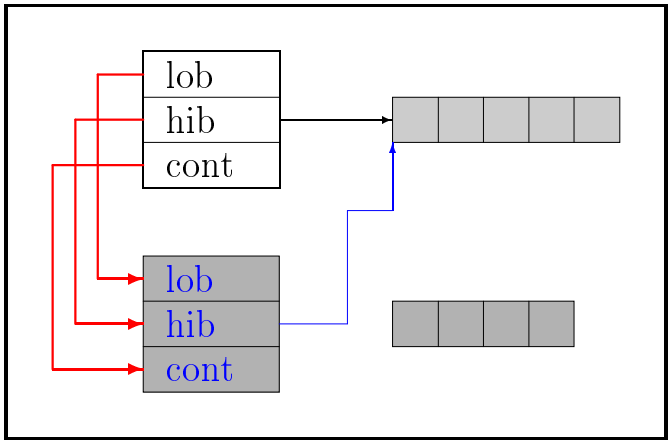
Bevezetünk még egy indexelés operátort (túlterhelés), amelyben már nem írhatjuk át az értéket csak lekérdezhetjük.

```
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





```
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

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`).

```
// IntArray.h: interface for the IntArray class.
#ifndef _INTARRAY_H
#define _INTARRAY_H__

class IntArray
{
public:
...
    class InvalidBound{};
    class IndexOutOfRange
    {
    public:
        int    index;
        IndexOutOfRange(int i)  { index = i; }
    };
private:
...
};

#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];
}
...
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);
}
```