AAAAAAAAAAAAAAAAAAA rendelkezésre álló idő: 20 **perc**

Programozás alapja 2. 3. ellenőrző dolgozat. 2015.04.15. Kurz/**Terem**: G2/	Elért pontszám:
Név: **Neptun:**	

1. Készítsen C++ nyelven generikus tárolót (*Sor*), ami FIFO (queue) elven működik! A tárolóban maximum 432 adatot akarunk tárolni. A tároló valósítsa meg a következő műveleteket: 7p
   - új adat behelyezése (*push*)
   - legrégebben betett adat elérése jobb- és balértékként egyaránt (*front*)
   - legrégebben betett adat eldobása (*remove_front*)
   - tároló ürítése (*clear*) (minden adatot eldob)
   - tárolt elemek számának lekérdezése (*size*)

   A tároló adatai közvetlenül ne legyenek elérhetők! Hibakezeléssel nem kell foglalkoznia! Használatára az alábbi kódrészlet mutat példát:

```
Sor<int> ti;                  // int tároló létrehozása
std::cout << ti.size();       // az elemek száma most 0
Sor<double> td;               // double tároló létrehozása
td.push(3.2);                 // beteszünk 3.2-t
td.push(123);                 // beteszünk 123-at
td.front() += 1;              // a legrégebbi elemet megnöveljük eggyel
std::cout << td.front();      // kiírjuk a legrégebbi, növelt adatot (4.2)
td.remove_front();            // eldobjuk a legrégebben betett adatot
std::cout << td.front();      // kiírjuk a legrégebben betett adatot (123)
td.clear();                   // a tároló ismét üres
```

2. Az előző feladatban megvalósított sablont specializálja úgy, hogy ha a generikus típus *BudaCrash*, akkor a tároló iratmegsemmisítőként működjön, azaz a betett adatot ne tárolja el! Mindig maradjon üres! 3p

megoldás:

```
template <class T>
class Sor {
    static const int M=432;
    T data[M];
    int db;
    int elso;
public:
    Sor() :db(0),elso(0) {}
    T& front() { return data[elso%M]; }
    void push(T d) { data[(elso+db++)%M] = d; }
    void remove_front() { db--; elso++;}
    size_t size() const { return db; }
    void clear() { db = 0; }
};

template <>
void Sor<BudaCrash>::push(BudaCrash) {}
```

BBBBBBBBBBBBBBBBBBBBB rendelkezésre álló idő:20 **perc**

Programozás alapja 2. 3. ellenőrző dolgozat. 2015.04.15. Kurz/**Terem**: G1/	Elért pontszám:
Név: **Neptun:**	

1. Készítsen C++ nyelven fix méretű generikus tömböt (*Tomb*)! A tömbben maximum 567 adatot akarunk tárolni. A tömb tartsa nyilván az aktuális adatok számát! Valósítsa meg a következő műveleteket: 7p
   - új adat behelyezése a tömb végére (*push_back*), azaz az eddig tárolt legnagyobb indexű adat után; csak ezzel a művelettel nő az adatok száma;
   - tetszőleges indexű adat elérése (*operator[]*) jobb- és balértékként egyaránt; nem kell ellenőrizni, hogy az adat létezik-e
   - legkisebb indexű adat eldobása (*pop_front*); ezzel a művelettel csökken a tárolt adatok száma, és minden tárolt adat indexe eggyel csökken;
   - tároló ürítése (*clear*) (minden adatot eldob);
   - tárolt elemek számának lekérdezése (*size*);

   A tároló adatai közvetlenül ne legyenek elérhetők! Az osztálynak nem kell konstans környezetben működnie és hibakezeléssel sem kell foglalkoznia! Használatára az alábbi kódrészlet mutat példát:

```
Tomb<int> ti;                  // int tároló létrehozása
std::cout << ti.size();        // az elemek száma most 0
ti.push_back(12);              // beteszünk 12-t
ti.push_back(456);             // beteszünk 456-ot
ti[0] = 34;                    // 0. elemet (12) átírjuk 34-re
ti.clear();                    // minden elemet eldobunk
ti.push_back(43);              // beteszünk 43-at
ti.push_back(33);              // beteszünk 33-at
ti.pop_front();                // eldobjuk a 43-at
std::cout << ti[0];            // kiírjuk a 0. adatot (33)
```

2. Az előző feladatban megvalósított sablont specializálja úgy, hogy ha a generikus típus *Quaestor*, akkor a tömb iratmegsemmisítőként működjön, azaz a *push_back* művelettel betett adatot ne tárolja el! 3p

megoldás:

```
template <class T>
class Tomb {
    static const int M=567;
    T data[M];
    int db;
    int elso;
public:
    Tomb() :db(0),elso(0) {}
    T& operator[](size_t i) { return data[(elso+i)%M]; }
    void push_back(T d) { data[(elso+db++)%M] = d; }
    void pop_front() { db--; elso++; }
    size_t size() const { return db; }
    void clear() { db = 0; }
};

template <>
void Tomb<Quaestor>::push_back(Quaestor) {}
```