AAAAAAAAAAAAAAAAAAA rendelkezésre álló idő: 15 **perc**

Programozás alapja 2. 3. ellenőrző dolgozat. 2015.04.13. Kurz/**Terem**: G1/	Elért pontszám:
Név: **Neptun:**	

1. Készítsen C++ nyelven generikus tárolót (*Tarolo*), ami LIFO (stack) elven működik! A tárolóban maximum 100 adatot akarunk tárolni. A tároló valósítsa meg a következő műveleteket: 7p
   - új adat behelyezése (*push*)
   - legutoljára betett adat elérése (*top*)
   - legutoljára betett adat eldobása (*pop*)
   - 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:

```
Tarolo<int> ti;                 // int tároló létrehozása
std::cout << ti.size();         // az elemek száma most 0
Tarolo<double> td;              // double tároló létrehozása
td.push(3.2);                   // beteszünk 3.2-t
td.push(123);                   // beteszünk 123-at
std::cout << td.top();          // kiírjuk az utoljára betett adatot (123)
td.pop();                       // eldobjuk az utoljára betett adatot
```

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 Tarolo {
    T data[100];
    int db;
public:
    Tarolo() :db(0) {}
    T& top() { return data[db-1]; }
    void push(T d) { data[db++] = d; }
    void pop() { db--;}
    size_t size() const { return db; }
    void clear() { db = 0; }
};

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

BBBBBBBBBBBBBBBBBBBBB rendelkezésre álló idő:15 **perc**

Programozás alapja 2. 1. ellenőrző dolgozat. 2015.04.13. Kurz/**Terem**: G1/	Elért pontszám:
Név: **Neptun:**	

1. Készítsen C++ nyelven fix méretű generikus tömböt (*Tomb*)! A tömbben maximum 500 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
   - legnagyobb indexű adat eldobása (*pop_back*); ezzel a művelettel csökken a tárolt adatok száma;
   - 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 átírjuk 34-re
ti.clear();                   // minden elemet eldobunk
Tomb<double> td;              // double tároló létrehozása
td.push_back(43.2);           // beteszünk 43.2-t
td.pop_back();                // el is dobjuk
```

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 {
    T data[100];
    int db;
public:
    Tomb() :db(0) {}
    T& operator[](size_t i) { return data[i]; }
    void push_back(T d) { data[db++] = d; }
    void pop_back() { db--;}
    size_t size() const { return db; }
    void clear() { db = 0; }
};

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