„Tárolók és metódusaik” jellegű feladatok megoldásokkal

A feladatgyűjtemény 2006 és 2015 között ZH feladatként kiadott C++ feladatokat és megoldásukat tartalmazza. A megoldásokat sárga kiemeléssel jelöltük.

Tartalom

1. Feladat

Tételezze fel, hogy rendelkezésére áll az *std::vector* osztályhoz hasonló tulajdonságokkal rendelkező generikus sorozattároló. **Készítsen** olyan generikus adapter osztályt (*MyVector*), ami ehhez a következő publikus hozzáférést biztosítja:

- legyen *paraméter nélküli konstruktora*, ami üres tárolót hoz létre;
- legyen *iterátoros konstruktora*, ami a tárolót feltölti az iterátorokkal megadott sorozattal;
- legyen *másoló konstruktora*, *értékadó operátora* és *destruktora*;
- legyen *írható random access iterátora*, *begin* és *end* tagfüggvénye (az std::vector tárolónak is van ilyen iterátora);
- legyen *shift* tagfüggvénye, ami minden elemet N hellyel balra tol; N a tagfüggvény paramétere legyen; a belépő elemek a generikus adat paraméter nélkül hívott konstruktorával létrehozott objektumok legyenek; balra tolás azt jelenti, hogy a sorozat első helyére kerül a második elem, a második helyére a harmadik elem, a harmadik helyére a negyedik elem, stb;

A felsorolt függvények és az implicit tagfüggvények kivételével más hozzáférést az adatper ne tegyen lehetővé!

Mutassa be az adapter használatát egy rövid kódrészlettel, melyben létrehoz egy 6 egész értéket tartalmazó C tömböt, majd az elkészített adapter segítségével létrehoz egy tárolót, amit feltölt a C tömb elemeivel. Ezután írja ki a tároló elemeit a standard kimenetre, tolja el az elemeket a tárolóban, majd írja ki újra az elemeket! Végül STL algoritmust felhasználva számolja meg, hány darab 0 értékű elem van a tárolóban!

Egy lehetséges megoldás:

```
template<class T, class C>
class MyVector : private C {
public:
    MyVector() {};
    template <class I>
    MyVector(I first, I last) :C(first, last) {}
    typedef typename C::iterator iterator;
    iterator begin() { return C::begin(); }
    iterator end() { return C::end(); }
    void cshift(unsigned int n) {
        while (n--) {
            size_t i = 1;
            for (; i < C::size() ; i++)
                (*this)[i-1] = (*this)[i];
            (*this)[i-1] = T();
        }
    }
};
```

```
    int t[] = { 1, 2, 3, 4, 5 };
    MyVector<int, std::vector<int> >v(t, t+5);
    std::ostream_iterator<int> oit(std::cout, ",");
    std::copy(v.begin(), v.end(), oit);
    v.cshift(1);
    std::copy(v.begin(), v.end(), oit);
    std::count(v.begin(), v.end(), 0);
```

2. Feladat

Készítsen egy LIFO (*Stack*) adapter sablont, ami a sablonparaméterként átadott tárolóból verem működésű tárolót hoz létre.

a. A verem rendelkezzen a következő műveletekkel:
   - *konstruktor* – csak paraméter nélküli konstruktora van, ami létrehoz egy üres vermet
   - *empty()* – igaz, ha a verem üres;
   - *size()* – visszaadja, hogy hány elem van a veremben;
   - *top()* – verem legfelső elemének elérése;
   - *push()* – elem betétele a verembe;
   - *pop()* – verem legfelső elemének eldobása;

Tételezze fel, hogy a sablonparaméterként átadott tároló megvalósítja a következő műveleteket: *back(), push_back(), pop_back(), empty(), size()*. Ezek funkciója és működése megegyezik az STL tárolóknál megismert azonos nevű metódusok funkcióival.
A sablon használatát az alábbi kódrészlet szemlélteti:

```
Stack<int, std::dequeue<int> > st;
st.push(123);
```

b. Készítsen egy olyan **fix méretű** generikus tárolót (*FixTar*), ami rendelkezik a korábban megadott tulajdonságokkal, azaz alkalmazható rá az előző pontban elkészített adapter!
c. Ezen felül legyen a tárolónak előre haladó iterátora, ami rendelkezik preinkremens (++a), egyenlőtlenség vizsgáló (!=), valamint indirekció/dereferáló (*a) operátorokkal! A tároló méretét sablonparaméterként vegye át, melynek alapértelmezett értéke legyen 1000! A *push_back* metódus dobjon *std::out_of_range* kivételt, ha a tároló megtelik, vagy ha üres tároló pop tagfüggvényét hívják!

Valósítsa meg a push_back, valamint az iterátor használatához szükséges összes tagfüggvényt!

Egy lehetséges megoldás
a.)

```
template<class T, class S>
class Stack {
    S store;
public:
    bool empty() const { return store.empty(); }
    size_t size() const { return store.size(); }
    T& top() { return store.back(); }
    void push (const T& x) { store.push_back(x); }
    void pop() { store.pop_back(); }
};
```

b.) és c.)

A feladatban az iterátorral egy tömböt kell bejárni, és az iterátor működésére nincs semmilyen speciális kikötés (pl. dobjon kivételt, stb.), ezért egy pointerrel helyettesíthető. Ezt kihasználó alternatív, teljes (minden tagfüggvényt tartalmazó) megoldás:

```
template<class T, size_t siz = 1000>
class FixTar {
    T tar[siz];
    size_t count;
public:
    typedef T* iterator;
    FixTar():count(0) {}
    iterator begin() { return tar; }
    iterator end() { return tar+count; }
    void push_back(const T& x) {
        if (count == siz) throw
std::out_of_range("FixTar::push_back");
        tar[count++]=x;
    }
```

```
    void pop_back() {
        if (!count) throw std::runtime_error("FixTar::pop_back ");
        count--;
    }
    bool empty() const { return count == 0; }
    size_t size() const { return count; }
};
```

3. Feladat

Tételezze fel, hogy rendelkezésére áll egy generikus tároló osztály (*Tarolo*), ami pontosan olyan publikus interfésszel rendelkezik, mint az *std::list!*

- A *Tarolo* osztály felhasználásával **hozzon** létre egy olyan új generikus tárolót (*UjTarolo*), ami pontosan úgy viselkedik, mint a *Tarolo* osztály, és van olyan tagfüggvénye (*kiir*), ami képes kiírni a tároló taralmát a paraméterként kapott *std::ostream* objektumra! A kiírás sorrendjét és szerkezetét (soronként, egy sorba) Ön határozhatja meg. A kiírás után a tároló tartalma ugyanaz maradjon, mint a kiírás előtt volt!
- Ügyeljen arra, hogy a *Tarolo* minden tagfüggvénye (a konstruktorok is) elérhető legyen!
- Határozza meg, hogy megoldása milyen új követelményeket támaszt a generikus adattal szemben!
- Az elkészített *UjTarolo* sablon felhasználásával hozzon létre egy *double* elemeket tartalmazó tárolót! Olvasson be a szabványos bemenetről fájl végéig értékeket ebbe a tárolóba, majd törölje ki az átlagnál kisebb értékeket, végül írja ki a tároló tartalmát a szabványos kimenetre!

A feladat megoldásához használhat STL algoritmust és/vagy tárolót.

Egy lehetséges megoldás

```
template <class T>
class UjTarolo : public Tarolo<T> {
public:
  UjTarolo(size_t n = 0, const T& v = T()) : Tarolo<T>(n, v) {}
  template <class I>
  UjTarolo(I first, I last) : Tarolo<T>(first, last) {}
  void kiir(std::ostream& os) {
    ostream_iterator<T> out(os, ",");
    copy(this->begin(), this->end(), out);
  }
};

int main() {
  UjTarolo<double> t;
  double x;

  while (cin >> x) t.push_back(x);
  x=0;
  x = accumulate(t.begin(), t.end(), x);
  x = x / t.size();
  t.erase(remove_if(t.begin(), t.end(), bind2nd(less<double>(), x)),
t.end());
  t.kiir(cout);
  return 0;
}
```

Alternatív megoldás a törlésre:

```
typedef UjTarolo<double>::iterator iter;
```

```
for (iter it = t.begin(); it != t.end(); ) {
  if ((*it) < x)
      it = t.erase(it); // Törléskor az iterátorok érvénytelenné
válnak!
                        // Az erase visszaad egy érvényeset a köv.
elemre.
  else
      ++it;
  }
```

4. Feladat

Egy generikus halmaz (*Set*) tárolót kell készíteni, ami tetszőleges számú adatot képes tárolni. A tároló rendelkezzen a következő műveletekkel:

- *isElement()* – annak eldöntése, hogy egy adott elem benne van-e a halmazban;
- *insert()* – elem hozzáadása a halmazhoz;
- *size()* – visszaadja, hogy hány elem van a halmazban;
- *empty()* – igaz, ha a halmaz üres;
- *clear()* – törli az összes elemet

Az osztályból példányosított objektum:

- legyen átadható érték szerint függvényparaméterként;
- kezelje helyesen a többszörös értékadást (*s1*=*s2*=*s3*);

a. **Deklarálja** a *Set* osztályt!
b. **Valósítsa** meg az osztály konstruktorát, másoló konstruktorát, az = operátort, a clear(), insert() valamint a size() tagfüggvényeket.
c. Hozza létre a {10.0, 10.3, 10.8} halmazt!
d. Specializálja ennek a halmaznak (double) az *isElement* tagfüggvényét úgy, hogy két elemet akkor tekintsen azonosnak, ha különbségük abszolútértéke kisebb, mint 1e-8!

Egy lehetséges megoldás:

a.

```
template <class T>
class Set {
   T *t;
   size_t siz;
public:
   Set() :t(NULL), siz(0) { }
   Set(const Set& rhs);
   Set& operator=(const Set& rhs);
   bool isElement(const T& val) const;
   void insert(const T& val);
   size_t size() const;
   bool empty() const;
   void clear();
   ~Set();
};
```

b.

```
template <class T>
Set<T>::Set(const Set<T>& rhs) :t(NULL) {
    siz = rhs.siz;
    t = new T[siz];
    for (size_t i = 0; i < siz; i++)
        t[i] = rhs.t[i];
}
```

```
template <class T>
Set<T>& Set<T>::operator=(const Set<T>& rhs) {
    if (this != &rhs) {
        delete [] t;
        siz = rhs.siz;
        t = new T[siz];
        for (size_t i = 0; i < siz; i++)
          t[i] = rhs.t[i];
    }
    return *this;
}

template <class T>
void Set<T>::clear() {
    siz = 0;
    delete [] t;
    t = NULL;
}

template <class T>
void Set<T>::insert(const T& val) {
    if (isElement(val)) return;
    T* tuj = new T[siz+1];
    for (size_t i = 0; i < siz; i++)
        tuj[i] = t[i];
    tuj[siz++] = val;
    delete [] t;
    t = tuj;
}

template <class T>
size_t Set<T>::size() const { return siz; }
```

c. Létrehozás:

```
Set<double> s;    s.insert(10.0);    s.insert(10.3);
s.insert(10.8);
```

d. Specializáció:

```
template <>
bool Set<double>::isElement(const double& val) const {
    for (size_t i = 0; i != siz; i++)
        if (abs(t[i]-val) < 1e-8) return true;
    return false;
}
```

5. Feladat

a. **Deklaráljon** egy verem (*Stack*) szervezésű generikus tárolót, ami tetszőleges számú adatot képes tárolni. A tároló működését úgy alakítsa ki, hogy amennyiben a tároló dinamikus memóriaterületét növelni kell, úgy ne csak 1 adatnak megfelelő mérettel növeljen, hanem a sablon paramétereként megadott konstansnak megfelelő mérettel. Ezen sablonparaméter alapértelmezett értéke 100 legyen! A tároló rendelkezzen a következő műveletekkel:

- konstruktorok/destruktor
  - a paraméter nélküli konstruktor üres tárolót hoz létre;
  - a kétparaméteres konstruktor a paraméterként kapott 2 iterátornak megfelelő elemekkel tölti fel a tárolót;

- *top()* – a verem legfelső elemének referenciáját adja; dobjon *std::out_of_range* kivételt, ha a verem üres;
- *push()* – elemet tesz a verembe;
- *pop()* – eldobja a verem legfelső elemét; dobjon *std::out_of_range* kivételt, ha üres a verem;
- *swap()* – a verem legfelső 2 elemét felcseréli; dobjon *std::out_of_range* kivételt, ha a veremben 2-nél kevesebb adat van;
- *size()* – a tárolóban levő elemek számát adja;
- *empty()* – igaz értéket ad, ha a tároló üres.

Az osztályból példányosított objektum legyen átadható érték szerint függvényparaméterként, kezelje helyesen a többszörös értékadást (*s1=s2=s2*)! A *top* tagfüggvény konstans objektumra is működjön!

b. **Valósítsa** meg az osztály konstruktorait, valamint a *push(), swap(), empty(), top() és pop()* metódusokat!

Egy lehetséges megoldás:

a.

```
template<class T, size_t inc = 100>
class Stack {
    T *tar;
    size_t msiz;
    size_t siz;
public:
    Stack();
    Stack(const Stack&);
    template <typename Iter>                    // template paraméter az
iterátor típusa
    Stack(Iter first, Iter last);
    Stack& operator=(const Stack&);
    T& top();
    const T& top() const;
    void push(const T&);
    void pop();
    void swap();
    int size() const { return siz; }
    bool empty() const { return siz == 0; }
    ~Stack() { delete[] tar; }
};
```

b.

```
// lehetne az osztály dekl.val együtt, de a megodás menetének így jobban megfelel

template<class T, size_t inc>
Stack<T, inc>::Stack(): msiz(0), siz(0) { // konstruktor
    tar = new T[msiz];
}
template<class T, size_t inc>
Stack<T, inc>::Stack(const Stack<T, inc>& s2) {  // másoló
konstruktor
    tar = NULL;
    *this = s2;
}
template<class T, size_t inc>
template <typename Iter>
```

```
Stack<T, inc>::Stack(Iter first, Iter last) :msiz(0), siz(0) { //
konstruktor 2 iterátorral
    tar = new T[msiz];
    while(first != last)
        push(*first++);
}
template<class T, size_t inc>
void Stack<T, inc>::push(const T& e) {
    if (siz == msiz) {
        T *tmp = new T[msiz+inc];
        for (size_t i = 0; i < siz; i++)
            tmp[i] = tar[i];
        delete[] tar;
        tar = tmp;
        msiz += inc;
    }
    tar[siz++] = e;
}
template<class T, size_t inc>
void Stack<T, inc>::swap() {
    if (siz = < 2) throw std::out_of_range("Sor::swap: siz<2!");
    T tmp = tar[siz-1];
    tar[siz-1] = tar[siz-2];
    tar[siz-2] = tmp;
}
template<class T, size_t inc>
T& Stack<T, inc>::top() {
    if (siz == 0) throw std::out_of_range("Stack::top: empty!");
    return tar[siz-1];
}
template<class T, size_t inc>
const T& Stack<T, inc>::top() const {
    if (siz == 0) throw std::out_of_range("Stack::top: empty!");
    return tar[siz-1];
}
template<class T, size_t inc>
void Stack<T, inc>::pop() {
    if (siz == 0) throw std::out_of_range("Stack::pop: empty!");
    siz--;
}
```

6. Feladat

a. Tételezze fel, hogy az előző feladat *Stack* osztálya elkészült, és hibátlanul működik! Ezen osztály **felhasználásával deklaráljon** *Calculator* osztályt, ami aritmetikai műveleteket tud végezni double típusú adatokkal! A *Calculator* osztály rendelkezzen a következő műveletekkel:

- *store()* – adatot ír a *Calculator* veremmemóriájába;
- *print()* – kiveszi a veremből a legfelső elemet, és kiírja paraméterként kapott *std::ostream*-re;
- *add()* – összeadja a verem két legfelső elemét, és az eredményt a verem tetejére teszi;
- *sub()* – kivonja a verem legfelső elemét az alatta levő elemből, és az eredményt a verem tetejére teszi;

Amennyiben a kétparaméteres műveletek esetében kevesebb, mint 2 elem van a veremben, dobjon *std::out_of_range* kivételt!

Működjön az elvárásoknak megfelelően az alábbi kódrészlet:

```
Calculator cal9;
std::cin >> cal9;  // adatot olvas a Calculator veremmemóriájába
std::cout << cal9; // kiveszi a Calculator veremmemóriájának legfelső elemét és kiírja
```

Ügyeljen arra, hogy a *Stack* osztály metódusai ne legyenek publikusan elérhetők!

b. **Valósítsa** meg a Calculator osztály store() és print() műveleteit, a beolvasást a >> operátorral, a kiírást a << operátorral, valamint az add() és sub() metódusokat!

Egy lehetséges megoldás:

a.

```
class Calculator {
    Stack<double> st;
public:
    void store(double x);
    void print(ostream& os);
    void add();
    void sub();
};
```

b.

```
void Calculator::store(double x) {
    st.push(x);
}
void Calculator::print(ostream& os) {
    os << st.top();
    st.pop();
}

istream& operator>>(istream& is, Calculator& c) {
    double d;
    is >> d;
    c.store(d);
    return is;
}
ostream& operator<<(ostream& os, Calculator& c) {
    c.print(os);
    return os;
}

void Calculator::add() {
    double d = st.top();
    st.pop();
    st.top() += d;
}
void Calculator::sub() {
    double d = st.top();
    st.pop();
    st.top() -= d;
}
```

7. Feladat

Egy korlátos méretű generikus adatsor *(Sor)* osztályt kell készíteni. A tároló maximális méretét az osztály egyparaméteres konstruktora kapja meg. A default konstruktor állítsa a méretet 160-ra! Az osztály rendelkezzen a következő műveletekkel:

- *front()* – a sor első elemének referenciáját adja; dobjon *std::out_of_range* kivételt, ha üres a sor;
- *back()* – a sor utolsó elemének referenciáját adja; dobjon *std::out_of_range* kivételt, ha üres a sor;
- *push()* – elemet tesz a sor végére; amennyiben a tárolt elemek száma meghaladná a tároló méretét, úgy a sor elejéről dobjon el egy adatot;
- *pop()* – eldob egy elemet a sor elejéről; dobjon *std::out_of_range* kivételt, ha üres a sor;
- *size()* – a tárolóban levő elemek számát adja;
- *maxsize()* – a tároló maximális méretét adja.

Az osztályból példányosított objektum:

- legyen átadható érték szerint függvényparaméterként;
- kezelje helyesen a többszörös értékadást (*q1=q2=q3*);
- a front és a back tagfüggvényeknek nem kell konstans objektumra működni!

a. **Deklarálja** a *Sor* osztályt!
b. **Valósítsa** meg az osztály default és másoló konstruktorát, a *front()*, *back()*, *pop()*, valamint a *push()* metódusokat!

Egy lehetséges megoldás:

a.

```
template<class T>
class Sor {
    T *tar;           // pointer az adatra
    const int msiz;   //maximális méret
    int siz;          // aktuális méret
    int first;        //első adat helye
public:
    Sor(int msiz = 160);
    Sor(const Sor&);
    Sor& operator=(const Sor&);
    T& front();
    T& back();
    void push(const T&);
    void pop();
    int size() const;
    int maxsize() const;
    ~Sor();
};
```

b.

```
// A tagfüggvények megvalósítása lehetne az osztály deklarációjával együtt, de külön írva a megoldás menetének jobban megfelel.
template<class T>
Sor<T>::Sor(int msiz = 160): msiz(msiz), siz(0), first(0) {
    tar = new T[msiz];      // tárterület lefoglalása
}

template<class T>
Sor<T>::Sor(const Sor& s2) {
    tar = NULL;
    *this = s2;
}
```

```
template<class T>
T& Sor<T>::front() {
    if (siz == 0) throw std::out_of_range("Sor::front: empty!");
    return tar[first];
}

template<class T>
T& Sor<T>::back() {
    if (siz == 0) throw std::out_of_range("Sor::back: empty!");
    return tar[(first+siz-1) % msiz];
}

template<class T>
void Sor<T>::push(const T& e) {
    if (siz == msiz) pop();
    tar[(first+siz) % msiz] = e;
    siz++;
}
template<class T>
void Sor<T>::pop() {
    if (siz == 0) throw std::out_of_range("Sor::pop: empty!");
    first = (first+1) % msiz;
    siz--;
}
```

8. Feladat

Csapatának egy Verem osztályt kell létrehoznia egész számok tárolására. A csapattagokkal megállapodtak az osztály publikus interfészében, valamint abban, hogy a verembe tehető adatok számát nem korlátozzák, a tárolás dinamikus. A megállapodást az alábbi kommentezett deklaráció rögzíti. A deklarációt **a pontozott részen** kiegészítheti, de a **publikus** interfészt nem változtathatja meg! Követelmény, hogy az osztály legyen átadható érték szerint függvényparaméterként, és működjön helyesen a többszörös értékadás is.

```
class Verem: private deque<int> {
public:
  Verem(int n = 0, int val = 0); // Létrehoz egy vermet n db val értékkel (minden
eleme val)
  void push(int);                // Verembe tesz egy egész értéket.
  int  pop() throw(range_error); // Kiveszi a verembe utoljára betett értéket;
                                 // üres verem esetén range_error kivételt dob.
  bool empty() const;            // Igaz, ha üres a verem.
  void dup() throw(range_error); // A verem tetején levő elemet megduplázza (újból
beteszi);
                                 // üres verem esetén range_error kivételt dob
  void swap()throw(range_error); // A verem tetején levő két elemet megcseréli;
                                 // üres verem esetén range_error kivételt dob.
  int size() const;              // Visszaadja a veremben levő elemek számát.
 ~Verem();
    ........................
    ........................
    ........................
};
```

a. Egészítse ki a fenti deklarációt úgy, hogy az az elvárásoknak megfelelően működjön! A megoldáshoz felhasználhatja az STL bármelyik tárolóját ill. algoritmusát! Ügyeljen arra, ha származtatással oldja meg a feladatot, akkor is csak a jelenlegi interfész legyen publikusan elérhető!
b. **Valósítsa meg** a következő tagfüggvényeket: *destruktor, konstruktorok, pop, empty, push, dup, size*!
c. Tételezze fel, hogy az előző pontok *Verem* osztálya elkészült és hibátlanul működik! Ezen osztály **felhasználásával deklaráljon** egy *Szamologep* osztályt elemi aritmetikai műveletek (*osszead, kivon, szoroz, oszt*) végzésére. A műveletvégző tagfüggvények a verem két legfelső elemét kiveszik, elvégzik a műveletet, majd az eredményt a verem tetejére teszik! Legyen az osztálynak egy *kiir* tagfüggvénye is, ami kiírja a legfelső elemet a standard kimenetre, de a kiírt érték a veremben is maradjon meg! Egy *istream* típusú objektumból legyen lehetőség egész számokat beolvasni a szokásos >> operátorral!
d. **Valósítsa** meg az osztály *osszead, kiir, kivon, szoroz, oszt* tagfüggvényét, valamint a >> *operátort*! Működjön helyesen a következő kódrészlet:

```
Szamologep gep;
std::cin >> gep >> gep;        // input: 4 2
gep.osszead();
gep.kiir();                    // output: 6
```

Lehetséges megoldás:

a. és b. Célszerűen egy STL tárolóból származtatunk private vagy protected öröklléssel:

```
Verem::~Verem() {}
Verem::Verem(int n, int val) {
  for (int i = 0; i < n; i++)
    push(val);
}
int Verem::pop() throw(range_error) {
  if (empty()) throw range_error("Verem::pop - empty");
  int tmp = back();
  pop_back();
  return tmp;
}
inline bool Verem::empty() const {
  return deque<int>::empty();
}
```

Tartalmazott STL, vagy saját tárolóval is megoldható. Ez utóbbival többet kellett írni.

```
inline void Verem::push(int val) {
    push_back(val);
}
void Verem::dup() throw(range_error) {
  if (empty()) throw range_error("Verem::dup - empty");
  push(back());
}
void Verem::swap() throw(range_error) {
  if (size() < 2) throw range_error("Verem::swap - empty");
  int tmp1 = pop();
  int tmp2 = pop();
  push(tmp1);
  push(tmp2);
}
inline int Verem::size() const {
  return deque<int>::size();
}
```

c. Tartalmazott objektum alkalmazása a célszerű:

```
class Szamologep {
  Verem v;
public:
  void osszead();
  void kivon();
  void szoroz();
  void oszt();
  void kiir();
  friend istream& operator>>(istream&, Szamologep&);
};
void Szamologep::osszead() {
  int op1 = v.pop();
  int op2 = v.pop();
  v.push(op2+op1);
}
void Szamologep::kiir() {
  v.dup();
  cout << v.pop();
}
istream& operator>>(istream& is, Szamologep& szg) {
  int i;
  is >> i;
  szg.v.push(i);
  return is;
}
```

Privát örökléssel is megoldható.

```
void Szamologep::kivon() {
  int op1 = v.pop();
  int op2 = v.pop();
  v.push(op2-op1);
}
void Szamologep::szoroz() {
  int op1 = v.pop();
  int op2 = v.pop();
  v.push(op2*op1);
}
void Szamologep::oszt() {
  int op1 = v.pop();
int op2 = v.pop();
if (op1 == 0) then throw range_error("Szamologep::oszt - nullával
oszt");
  v.push(op2/op1);
}
```

A hibakezelés hiányát nem vettük itt hibának.

9. Feladat

Egy olyan osztályt (*Vec4)* kell létrehozni, ami dinamikusan változó méretű, valós elemek tárolására alkalmas tömböt valósít meg. A tömb kezdeti mérete és elemeinek kezdőértéke a konstruktorban adható meg. Meg kell valósítani az alábbi műveleteket:

- *at()* – elem direkt elérése (indexelés). Hibás indexelés esetén `range_error` kivételt dob.
- *insert()* – elemet szúr be a paraméterként megadott hely elé
- *erase()* – kitörli a paraméterként megadott helyen levő elemet
- *size()* – tárolt elemek számát adja
- *count()* – megszámolja, hogy hány olyan elem van a tárolóban, ami azonos a paraméterként kapott értékkel

Az osztály megvalósításán többen dolgoznak egyszerre, akikkel megállapodtak az osztály belső adatszerkezetében és a tagfüggvények funkcióiban. A megállapodást az alábbi kommentezett deklaráció rögzíti, amin **nem lehet változtatni.**
Követelmény, hogy az osztály legyen átadható érték szerint függvényparaméterként, de úgy döntöttek, hogy az **értékadást nem fogják megvalósítani.**

```
class Vec4 {
  Vec4& operator=(const Vec4); // Az értékadó operátor nincs megvalósítva
protected:
  double *v;       // Pointer a dinamikus adatterületre. Ezen a területen tároljuk az adatokat.
  int siz;         // Tárolt elemek száma, azaz ekkora dinamikus területet foglalunk.
public:
  Vec4 (int n = 0, double iv = 0);
       // Létrehoz egy n elemű vektort, feltölti iv értékkel, inicializál
       // nulla elemszámhoz is foglal területet, hogy később ne legyen vele baj.
   Vec4 (const Vec4&);          // Másoló konstruktor.
   void insert(double d, int i = 0); // A d értéket beszúrja az i. adat elé.
   void erase(int i = 0);       // Törli az i. helyen levő adatot.
   int size() const;    // Visszaadja a vektorban tárolt elemek számát
   double& at(int);     // Indexelés művelete, ami ellenőrzi, hogy van-e az adott elem,
                        // ha nincs, hibát dob.
   int count(double d);// Megszámolja, hogy hány d-vel azonos elem van a tárolóban.
   ~Vec4();             // Megszünteti az objektumot
};
```

a. Valósítsa meg a tagfüggvényeket! Vegye figyelembe, hogy a mások által írt függvények belső megoldásait nem ismeri, azaz nem használhat ki olyan működést, ami a fenti kódrészletből, vagy annak megjegyzéseiből nem olvasható ki.

b. A Vec4 osztály **módosítása nélkül** készítsen egy *read* függvényt, ami a paraméterként kapott *std::isteram* típusú adatfolyamról **fájl végéig** érkező valós számokat beolvassa egy szintén **paraméterként kapott** *Vec4* típusú objektumba! Ügyeljen a helyes paraméterezésre! Feltételezheti, hogy az inputon a valós számok whitespace karakterekkel elválasztva követik egymást.

c. Az **STL vector** sablonjának, **vagy az előző pontok** sablonjának felhasználásával **készítsen** valós értékeket tároló verem osztályt! Valósítsa meg következő műveleteket:

   push – elemet tesz a verem tetejére
   pop – elemet vesz le a verem tetejéről. A függvény visszatérési értéke a kivett elem legyen!
   empty – *true* értéket ad, ha a sor üres.

Az osztály legyen átadható érték szerint függvényparaméterként, és kezelje helyesen a többszörös értékadást (*s1=s2=s3*)! (Az STL vector fontosabb műveletei: at, front, back, push_back, pop_back, insert, empty, resize, =, [])

Egy lehetséges megoldás:

a.

Konstruktorok:

```
Vec4::Vec4(int n, double iv) :siz(n){
  v = new double[siz];
  for (int i = 0; i < siz; i++)
    v[i] = iv;
}
Vec4::Vec4(const Vec4& f) {
  v = new double[siz = f.siz];
  for (int i = 0; i < siz; i++)
    v[i] = f.v[i];
}
```

Insert:

```
void Vec4::insert(double d, int i) {
  double *tmp = new double[siz + 1];
  int j;
  for (j = 0; j < i; j++)
    tmp[j] = v[j];
  while (j < siz)
    tmp[j+1] = v[j];
  tmp[i] = d;
  delete[] v;
    v = tmp;
    siz++;
}
```

Size:

```
int Vec4::size() const {
  return siz;
}
```

At + count:

```
double& Vec4::at(int ix) {
  if (ix >= siz)
    throw range_error("indexeles");
  return v[ix];
}
int Vec4::count(double d) {
  int n = 0;
  for (int i = 0; i < siz; i++)
     if (v[i] == d) n++;
  return n;
}
```

Erase:

```
void Vec4::erase(int i) {
  double *tmp = new double[siz - 1];
  int j;
  for (j = 0; j < i; j++)
    tmp[j] = v[j];
  for (;j < siz; j++)
    tmp[j] = v[j+1];
  delete[] v;
  v = tmp;
  siz--;
}
```

b.

```
void read(std::istream& is, Vec4& vec) {
  int i;
  while (is >> i)
     vec.insert(i, vec.size());
  vagy:
     vec.insert(i);
}
```

c.

```
class Verem {
    Vec4<double> adat;        // STL vector sablonnal is hasonló a megoldás
public:
    void push(double d) {
        adat.insert(d, adat.size());
    }
    double pop() {
        double d = adat.at(adat.size()-1);
        adat.erase(adat.size()-1);
        return d;
    }
    bool empty() {
        return adat.size() == 0;
    }
};
```

Akik felüldefiniálták a másoló konstruktort és/vagy az értékadást, azok feleslegesen dolgoztak, hiszen a Verem osztály nem kezel dinamikus adattagot, csak a tartalmazott objektum (adat), ami viszont a feltételezésünk szerint jól működik.
Csak akkor kell a másoló konstruktort és/vagy az értékadást felüldefiniálni, ha az alapértelmezés szerinti működés nem jó valamiért!

A feladat szerint vagy az STL vector sablonját, vagy az a. feladat Vec4 sablonját fel kellett használni!

10. Feladat

Valósítson meg C++ nyelven egy olyan osztályt (***KomplexTar***), ami tetszőleges számú *Komplex* típusú objektumot képes tárolni! Elvárás az osztállyal szemben, hogy másolható legyen, és értékadás bal és jobb oldalán is szerepelhessen. Legyen az osztálynak egy ***keres()*** tagfüggvénye, ami a tárolóból kikeres egy, a paramétereként kapott valós értékkel egyező abszolút értékű komplex számot. Ha talál, akkor az első megtaláltat adja vissza, ha nincs ilyen, úgy egy komplex nullát (*Komplex(0,0)*) adjon. Tételezze fel, hogy a *Komplex* osztály létezik, és helyesen működik!

Megadtuk a ***KomplexTar*** osztály deklarációját és a tagfüggvények definícióját, de ez több helyen hiányos. A **pontozott** részekre írva **egészítse ki** az alábbi **kódrészletet** úgy, hogy a fenti elvárásoknak megfelelően működjön, és ne lépjen fel memóriakezelési hiba! Nem kell minden pontozott részre írnia! Tételezze fel, hogy az **értékadás operátor,** amely külön állományban van megvalósítva, **helyesen működik,** ezért azt **nem kell** megírnia!

A feladatlap hátoldalán **adja meg**, hogy **megoldása** minimálisan **milyen elvárásokat támaszt** a *Komplex* osztállyal szemben! (pl: van olyan konstruktora, ami két valós számot vesz át; van … operátora; stb.)

Egy lehetséges megoldás:

```
class KomplexTar {
    Komplex* pKompl;  // dinamikus adatterület kezdőcíme, ahol
tárolunk
    int db;           // tárolt számok száma (pontosan ennyi komplex
van)
public:
    // Konstruktor: n darab valós számpárt (2*n darab számot) vesz
át, amiből
    // n darab komplex értéket készít és letárolja
    KomplexTar(const double v[], int n = 0) :db(n) {
        pKompl = new Komplex[db];
        for (int i = 0; i < 2*db; i += 2)
            pKompl[i/2] = Komplex(v[i], v[i+1]);
    }
    KomplexTar(const KomplexTar&);
    KomplexTar& operator=(const KomplexTar&);
    Komplex keres(double) const;
    ........~KomplexTar() { delete [] pKompl; }
};
KomplexTar::KomplexTar(const KomplexTar& k) :db(k.db) {
        pKompl = new Komplex[db];
        for (int i = 0; i < db; i++)
            pKompl[i] = k.pKompl[i];
}// Keres: az első megtalált abs abszolút értékű komplex számot adja
vissza.
Komplex KomplexTar::keres(double abs) const {
     for (int i = 0; i < db; i++)
          if (pKompl[i].abs() == abs)
               return pKompl[i];
     return Komplex(0,0);
}// Példa a használatra:
int main() {
    double dv[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    KomplexTar k1(dv, 4);  // négy darab valós számpár (8 darab
szám)
    KomplexTar k2 = k1;
    k1 = k2 = k2;
    cout << k2.keres(1.0); // az 1 abs. értékű számot keressük
}
```

Komplex osztállyal szemben támasztott elvárások:

- legyen létrehozható két valós számmal (valós és képzetes rész)
- legyen default konstruktora
- legyen másoló konstruktora
- legyen értékadás (operator=) operátora
- legyen egy abs() tagfüggvénye, ami visszaadja a szám abszolút értékét
- kiírható legyen egy ostream objektumra

11. Feladat

Valósítson meg C++ nyelven egy olyan osztályt (***Stringek***), ami tetszőleges számú *String* típusú objektumot képes tárolni! Elvárás az osztállyal szemben, hogy másolható legyen, és értékadás bal és jobb oldalán is szerepelhessen. Legyen az osztálynak egy ***keres()*** tagfüggvénye, ami a tárolóban megkeres egy, a paraméterként megkapott betűvel kezdődő stringet. Ha több ilyen van, akkor az első

találatot adja, ha nincs ilyen, úgy üres stringet (`String("")`) adjon vissza! (Feltételezzük, hogy üres stringet nem tárolunk a *Stringek* osztályban.) Tételezze fel, hogy a *String* osztály létezik és helyesen működik!

Megadtuk a ***Stringek*** osztály deklarációját és a tagfüggvények definícióját, de ez több helyen hiányos. A **pontozott** részekre írva **egészítse ki** az alábbi **kódrészletet** úgy, hogy a megadott főprogram az elvárásoknak megfelelően működjön, és ne lépjen fel memóriakezelési hiba! Nem kell minden pontozott részre írnia! Feltételezheti, hogy az **értékadás operátor,** amely külön állományban van megvalósítva, **helyesen működik,** ezért azt **nem kell** megvalósítania!

A feladatlap hátoldalán **adja meg**, hogy **megoldása** minimálisan **milyen elvárásokat támaszt** a *String* osztállyal szemben! (pl: van olyan konstruktora, ami C stringet kap paraméterként; van … operátora; stb.)

Egy lehetséges megoldás:

```
class Stringek {
    String* pStr;       // dinamikus adatterület kezdőcíme, ahol tárolunk
    int db;             // tárolt Stringek száma (pontosan ennyi string van)
public:
// Konstruktor: n darab C stringet kap, amiből n darab String-et készít és eltárolja
    Stringek(const char *sv[], int n = 0) :db(n) {
        pStr = new String[db];
        for (int i = 0; i < db; i++)
            pStr[i] = String(sv[i]);
    }
    Stringek(const Stringek&) .................;
    Stringek& operator=(const Stringek&);
    String keres(char) const;
    ........~Stringek() { delete [] pStr; }
};
Stringek::Stringek(const Stringek& s) {
    db = s.db;
    pStr = new String[db];
    for (int i = 0; i < db; i++)
      pStr[i] = s.pStr[i];
}
// Adott betűvel kezdődő stringet keres.
String Stringek::keres(char ch) const {
    for (int i = 0; i < db; i++)
        if (pStr[i][0] == ch)
            return pStr[i];
    return String("");
}
int main(int argc, const char *argv[]) {
    Stringek s1(argv, argc);   // indítási paraméterként kapott Stringek
    Stringek s2 = s1;
    s2 = s1 = s1;
    cout << s1.keres('C');     // 'C' betűvel kezdődő stringet keresünk.
}
```

String osztállyal szemben támasztott elvárások:

- legyen létrehozható C stringből
- legyen default konstruktora
- legyen másoló konstruktora
- legyen értékadás (operator=) operátora
- legyen index (operator[]) operátora, ami visszaadja az adott indexű karaktert
- kiírható legyen egy ostream objektumra

12. Feladat

Valósítson meg C++ nyelven egy olyan osztályt (***Pontok***), ami tetszőleges számú *Pont* típusú objektumot képes tárolni! Tételezze fel, hogy a *Pont* osztály létezik, és síkbeli pontok valós koordinátáit tárolja! A *Pontok* osztályt úgy valósítsa meg, hogy másolható legyen, és értékadás bal és jobb oldalán is szerepelhessen. Legyen összehasonlító operátora, ami logikai **igaz** értékkel tér vissza, ha a két objektum azonos számú pontot tartalmaz, és minden tárolt pont a tárolás sorrendjében megegyezik.

Megadtuk a ***Pontok*** osztály deklarációját és a tagfüggvények definícióját, de ez több helyen hiányos. A **pontozott** részekre írva **egészítse ki** az alábbi **kódrészletet** úgy, hogy a megadott főprogram az elvárásoknak megfelelően működjön, és ne lépjen fel memóriakezelési hiba! Nem kell minden pontozott részre írnia! Feltételezheti, hogy a **másoló konstruktor,** amely külön állományban van megvalósítva, **helyesen működik,** ezért azt **nem kell** megírnia!

A feladatlap hátoldalán **adja meg**, hogy **megoldása** minimálisan **milyen elvárásokat támaszt** a *Pont* osztállyal szemben! (pl: van olyan konstruktora, ami 2 valós számot kap paraméterként; van … operátora; stb.)

Egy lehetséges megoldás:

```
class Pontok {
    Pont* pPnt;       // dinamikus adatterület kezdőcíme, ahol tárolunk
    int db;           // tárolt Pontok száma (pontosan ennyi pont van)
public:
      // Konstruktor: n darab valós számpárt (2*n darab számot) kap, amiből
      // n darab Pont objektumot készít és letárolja
    Pontok(const double pv[], int n = 0) ........ :db(n)
...........{
        pPnt = new Pont[db];
        for (int i = 0; i < 2*db; i += 2)
            pPnt[i/2] = Pont(pv[i], pv[i+1]);
    }
    Pontok(const Pontok&);
    Pontok& operator=(const Pontok&) ............;
    bool operator==(const Pontok&) const;
    ........~Pontok() { delete [] pPnt;  }
};
Pontok& Pontok::operator=(const Pontok& p) ................ {
    if (this != &p) {
        delete[] pPnt;
        db = p.db;
        pPnt = new Pont[db];
        for (int i = 0; i < db; i++)
            pPnt[i] = p.pPnt[i];
    }
    return *this;
}
bool Pontok::operator==(const Pontok& p) const {
    if (db != p.db)
        return false;
    for (int i = 0; i < db; i++)
        if (pPnt[i] != p.pPnt[i])
            return false;
    return true;
}
int main() {
    double dv[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    Pontok p1(dv, 4); // négy darab valós számpár (8 darab szám)
```

```
    Pontok p2(dv, 3);
    cout << (p1 == p2); // ezek nem azonosak
    p2 = p1 = p1;
    cout << (p1 == p2); // ezek azonosak
}
```

Pont osztállyal szemben támasztott elvárások:

- legyen létrehozható 2 db valós számból
- legyen default konstruktora
- legyen értékadás (operator=) operátora
- legyen egyenlőtlenség (operator!=) operátora, ami hamis értékkel tér vissza, ha a tárolt koordináták azonosak.

13. Feladat

Valósítson meg C++ nyelven egy olyan osztályt (***Polinomok***), ami tetszőleges számú *Polinom* típusú objektumot képes tárolni! Tételezze fel, hogy a *Polinom* osztály létezik, és másodfokú polinom együtthatóit tárolja! A *Polinomok* osztályt úgy valósítsa meg, hogy másolható legyen, és értékadás bal és jobb oldalán is szerepelhessen. A függvényhívás operátora számítsa ki a tárolt polinomok helyettesítési értékének összegét a paraméterként kapott helyen.

Megadtuk a ***Polinomok*** osztály deklarációját és a tagfüggvények definícióját, de ez több helyen hiányos. A **pontozott** részekre írva **egészítse ki** az alábbi kódrészletet úgy, hogy a megadott főprogram az elvárásoknak megfelelően működjön, és ne lépjen fel memóriakezelési hiba! Nem kell minden pontozott részre írnia! Feltételezheti, hogy a **másoló konstruktor,** amely külön állományban van megvalósítva, **helyesen működik,** ezért azt **nem kell** megírnia!

A feladatlap hátoldalán **adja meg**, hogy **megoldása** minimálisan **milyen elvárásokat támaszt** a *Polinom* osztállyal szemben! (pl: van olyan konstruktora, ami 3 valós számot kap paraméterként; van … operátora; stb.) (1.5p)

Egy lehetséges megoldás:

```
class Polinomok {
    Polinom* pPoli;     // dinamikus adatterület kezdőcíme, ahol tárolunk
    int db;             // tárolt Polinomok száma (pontosan ennyi polinom van )
public:
    // Konstruktor: n darab valós számhármast (3*n darab számot) kap, amiből
    // n darab Polinom objektumot készít és letárolja
    Polinomok(const double a[], int n = 0)
.........:db(n)........... {
        pPoli = new Polinom[db];
        for (int i = 0; i < 3*db; i += 3)
            pPoli[i/3] = Polinom(a[i], a[i+1], a[i+2]);
    }
    ........ ........Polinomok(const Polinomok&).................;
    Polinomok& operator=(const Polinomok&) ......................;
    double operator()(double x) const;
    ........~Polinomok() { delete [] pPoli; }
};
Polinomok& Polinomok::operator=(const Polinomok& p)
.................. {
    if (this != &p) {
        delete[] pPoli;
        db = p.db;
        pPoli = new Polinom[db];
        for (int i = 0; i < db; i++)
            pPoli[i] = p.pPoli[i];
```

```
    }
    return *this;
}
double Polinomok::operator()(double x) const {
    double sum = 0;
    for (int i = 0; i < db; i++)
        sum += pPoli[i](x);
    return sum;
}
int main() {
    double av[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Polinomok p1(av, 3);     // három darab valós számhármas (9 darab szám)
    Polinomok p2 = p1;
    cout << p1(1);           // helyettesítés értékek összege az 1 helyen
    p2 = p1 = p1;
    cout << p2(3.14);        // helyettesítés értékek összege a 3.14 helyen
}
```

Polinom osztállyal szemben támasztott elvárások:

- legyen létrehozható 3 db valós számból
- legyen default konstruktora
- legyen értékadás (operator=) operátora
- legyen függvényhívás operátora, ami a helyettesítési értéket számolja

14. Feladat

Egy **térinformatikai rendszerben** különböző objektumokat akarunk egy tárolóban (*Tarolo*) tárolni. Tudjuk, hogy legfeljebb 300 objektumunk lehet. Minden objektumnak van egy azonosítója (*String*), amit tárolni kell, de a további adatok teljesen eltérőek lehetnek. Jelenleg csak utcát (*Utca*) és benzinkutat (*Kut*) akarunk tárolni, de később bővíteni akarjuk a rendszert. Az utcát a neve azonosítja, de tároljuk a hosszát is. A benzinkút esetében a neve mellett tároljuk azt is, hogy hétvégén nyitva van-e. A rendszerben megvalósítandó műveleteket egy kódrészlettel mutatjuk be:

```
Tarolo proba;                          // Ez lesz a tárolónk
proba.uj(new Utca("Irinyi", 2000));    // Utca hozzáadása (utcanév és hossz)
proba.uj(new Kut("OMV", true));        // Benzinkút (neve és nyitva van)
proba.kiir();                          // Tárolóból minden adatot kiírunk (név, hossz, stb.)
proba.clear();                         // Összes tárolt adatot töröljük
```

Feltételezheti, hogy a *Tarolo* osztályból példányosított objektumot nem akarjuk paraméterként átadni, és értékadás jobb, ill. bal oldalán sem szerepel.

a. **Tervezzen** meg olyan osztályhierarchiát, ami alkalmas az objektumok tárolására, és könnyen bővíthető. Ügyeljen arra, hogy jól határozza meg az objektumok felelősségét! **Deklarálja** a *Tarolo, Utca* és *Kut* osztályokat! **Csak** a *Tarolo* osztály konstruktorát, valamint az *uj()* és *kiir()* metódusát **valósítsa** meg!

b. **Tételezze** fel, hogy a *Tarolo* osztálynak nincs *kiir()* metódusa, de van iterátora, amivel egymás után sorban elérhetők a tárolt elemek. Mutassa be egy kódrészleten, hogy ezzel az iterátorral hogyan tudná a *kiir()* metódus funkcióját kiváltani!

Egy lehetséges megoldás:

a.

```
class Azon {
      String az;
public:
      Azon(String);
      virtual void kiir();
      virtual ~Azon();
```

```
};

class Utca :public Azon {
      double hossz;
public:
      Utca(String, double);
      void kiir();
};

class Kut :public Azon {
      bool nyitva;
public:
      Kut(String, bool);
      void kiir();
};


class Tarolo {
      Azon* azon[300];
      int db;
public:
      Tarolo() :db(0) {}
      void uj(Azon* p) {
            azon[db++] = p;
      }
      void kiir() {
            for (int i = 0; i < db; i++)
                  azon[i]->kiir();
      }
      void clear();
      ~Tarolo();
};
```

b.

```
for (Tarolo::iterator i = proba.begin(); i != proba.end(); ++i)
            (*i)->kiir();
```

15. Feladat

Egy **naplózó rendszerben** (*Log*) különböző eseményeket (*Event*) szeretnénk tárolni. Tudjuk, hogy legfeljebb 100 esemény lehet. Tárolni kell az esemény idejét és további adatokat, melyek teljesen eltérőek lehetnek. Keletkezhet pl. hőfokot (*Temp*) vagy szöveges üzenetet (*Message*) tartalmazó esemény is, de nem zárható ki, hogy újabb események is lesznek. A hőfokot valós számként (*double*), az üzenetet pedig *String*-ként kell tárolni. Az idő tárolására a *Time* osztályt kell használni, melynek az alapértelmezett konstruktora mindig az aktuális időt tárolja el. A rendszerben megvalósítandó műveleteket egy kódrészlettel mutatjuk be:

```
Log events;                          // Ez lesz az eseménytár
events.add(new Temp(12.3));          // Hőmérséklet esemény hozzáadása
events.add(new Message("ZH"));       // Szöveges esemény hozzáadása
events.print();                 // Események kiírása keletkezési sorrendben
events.clear();                      // Összes esemény törlése
```

Feltételezheti, hogy a *Log* osztályból példányosított objektumot nem akarjuk paraméterként átadni, és értékadás jobb, ill. bal oldalán sem szerepel.

a. **Tervezzen** meg olyan osztályhierarchiát, ami alkalmas az események keletkezési sorrendben való tárolására, és könnyen bővíthető. Ügyeljen arra, hogy jól határozza meg az objektumok felelősségét!

Deklarálja a *Log*, *Event* és *Message* osztályokat! **Csak** a *Log* osztály konstruktorát, valamint az *add()* és *print()* metódusát **valósítsa** meg!

b. **Tételezze** fel, hogy a *Log* osztálynak nincs *print()* metódusa, de van iterátora, amivel egymás után sorban elérhetők a tárolt események. Mutassa be egy kódrészleten, hogy ezzel az iterátorral hogyan tudná a *print()* metódus funkcióját kiváltani!

Egy lehetséges megoldás:

a.

```
class Event {
      Time time;
public:
      Event();
      virtual void print();
      virtual ~Event();
};

class Message :public Event {
      String msg;
public:
      Message (String);
      void print();
};
class Log {
      Event* events[100];
      int numevents;
public:
      Log() :numevents(0) {}
      void add(Event* ev) {
            events[numevents++] = ev;
      }
      void print() {
            for (int i = 0; i < numevents; i++)
                  events[i]->print();
      }
      void clear();
      ~Log();
};
```

b.

```
for (Log::iterator i = events.begin(); i != events.end(); ++i)
            (*i)->print();
```

16. Feladat

Tételezze fel, hogy rendelkezésére áll egy verem tulajdonságokkal rendelkező generikus tároló (*Stack*)! A tároló a szokásos verem műveletekkel rendelkezik (*push, pop, top, empty*), csak default konstruktora van. Sajnos nem jelez semmilyen hibát, ha üres veremre *pop* műveletet adnak ki. Ezen osztály felhasználásával készítsen egy olyan generikus osztályt (*MyStack),* ami *out_of_range* hibát dob ilyen esetben, egyébként a *Stack* osztállyal teljesen azonos módon működik! Működjön helyesen az alábbi programrészlet:

```
MyStack<int> intSt;                 // int elemeket tartalmazó stack
try {
   intSt.push(1);                   // 1-et teszünk a stack-be
   cout << intSt.top() << endl;     // kiírjuk a legfelső elemet
   intSt.pop();                     // eldobja a legfelső elemet
   intSt.pop();                     // hibát dob
} catch (exception& e) {
```

```
    cerr << "Kivetel:" << e.what() << endl;
}
```

Egy lehetséges megoldás:
Örökléssel egyszerűbb, de tartalmazott objektummal is megoldható volt.
Ez utóbbi esetben a Stack osztály minden függvényét delegálni kellett.

```
template<typename T>
class MyStack :public Stack<T> {
public:
    void pop() {
        if (empty()) throw std::out_of_range("MyStack: pop");
        Stack<T>::pop();
    }
};
```

17. Feladat

Tételezze fel, hogy rendelkezésére áll egy sorozattároló (*Vector*)! A tároló a szokásos műveletekkel rendelkezik (*push_back, pop_back, front, back, size, …*), csak default konstruktora van. Sajnos nem jelez semmilyen hibát, ha egy üres tároló utolsó elemét akarjuk elérni (back). Ezen osztály felhasználásával készítsen egy olyan generikus osztályt (*MyVector)*, ami *out_of_range* hibát dob ilyen esetben, egyébként a *Vector* osztállyal teljesen azonos módon működik! Működjön helyesen az alábbi programrészlet:

```
MyVector<double> vec;              // double elemeket tartalmazó
tároló
try {
    vec.push_back(11.0);           // 11-et teszünk a tárolóba
    cout << vec.back() << endl;    // kiírjuk a legutolsó elemet
    vec.pop_back();                // eldobja a legutolsó elemet
    vec.back();                    // hibát dob
} catch (exception& e) {
    cerr << "Kivetel:" << e.what() << endl;
}
```

Egy lehetséges megoldás:

Örökléssel egyszerűbb, de tartalmazott objektummal is megoldható volt. A size utáni …-tal azt akartuk sugallni, hogy még sok egyéb metódus lehet. Beágyazott objektummal való megvalósításnál ezeket mind delegálni kellett volna.

```
template<typename T>
class MyVector :public Vector<T> {
public:
      T& back() {
         if (size() == 0) throw std::out_of_range("MyVector: back");
            return Vector<T>::back();
      }
};
```

Teljesen precíz a megoldás akkor lenne, ha a

```
      const T& back() const;
```

tagfüggvényt is megvalósítjuk, de ezt nem vártuk el.

18. Feladat

Tételezze fel, hogy rendelkezésére áll egy sorozattároló (*Vektor*)! A tároló a szokásos műveletekkel rendelkezik (*push_back, pop_back, front, back, size, …*), csak default konstruktora van. Szeretnénk, ha lenne egy olyan művelete is (*swap_ends*), ami lehetővé teszi, hogy a sorozat első elemét megcseréljük az utolsó elemével. A *Vektor* osztály felhasználásával készítsen egy olyan generikus osztályt (*Vektorom),* ami pontosan úgy működik, mint a *Vektor*, és van *swap_ends()* tagfüggvénye! Ez utóbbi dobjon *out_of_range* hibát, ha a tároló üres, vagy csak 1 eleme van! Működjön helyesen az alábbi programrészlet:

```
Vektorom<double> vec;              // double elemeket tartalmazó
tároló
try {
   vec.push_back(11.0);            // 11-et teszünk a tárolóba
   vec.push_back(33.0);            // 33-at teszünk a tárolóba
   vec.push_back(21.0);            // 21-et teszünk a tárolóba
   vec.swap_ends();                // megcseréli a 11-et és a 21-et
   vec.pop_back();                 // eldobja a legutolsó elemet
   vec.pop_back();                 // eldobja a legutolsó elemet
   vec.pop_back();                 // eldobja a legutolsó elemet
   vec.swap_ends();                // hibát dob
} catch (exception& e) {
   cerr << "Kivetel:" <<e.what() << endl;
}
```

Egy lehetséges megoldás:

Örökléssel egyszerűbb, de tartalmazott objektummal is megoldható volt. A size utáni …-tal azt akartuk sugallni, hogy még sok egyéb metódus lehet. Beágyazott objektummal való megvalósításnál ezeket mind delegálni kellett volna.

```
template<typename T>
class Vektorom :public Vektor<T> {
public:
   void swap_ends() {
      if (size() <= 1) throw std::out_of_range("MyVector:
swap_ends");
      T tmp = back();
      back() = front();
      front() = tmp;
   }
};
```