0. Megoldó Manó 0.

Programozás alapjai 2. (inf.) pót zárthelyi 2014.05.22. gyak. hiányzás: 2	n/kZHp: n/11,5
ABCDEF **IB.028/2.**	Hftest: 4 (2,4p)

F.	Max.	Elért
1	3	
2	3	
3	4	
4	4	
5	6	
Σ	**20**	

Minden beadandó megoldását a feladatlapra, a feladat után írja! Készíthet piszkozatot, de csak a feladatlapra írt megoldásokat értékeljük! *A megoldások során feltételezheti, hogy minden szükséges input adat az előírt formátumban rendelkezésre áll.*
*A feladatok megoldásához csak a letölthető C, C++ és STL összefoglaló használható. Elektronikus eszköz (pl. kalkulátor, számítógép, notebook, mobiltelefon, Kindle) nem használható. A feladatokat **figyelmesen olvassa** el, megoldásukhoz **ne használjon fel STL** tárolót, kivéve, ha a feladat ezt külön engedi/kéri! **Ne írjon felesleges** függvényeket ill. kódot!*

A feladatra koncentráljon! Jó munkát! *Minimálisan elérendő pontszám: 8*

1. Feladat **6*0.5=3 pont**

Mit ír ki a szabványos kimenetre az alábbi program? Válaszához használja a négyzetrácsos területet! Abba a sorba írjon, ahol a kiírás keletkezik! Figyeljen a változók élettartamára!

```
#include <iostream>
struct Pr {
  static int sor;
  int kell;
  Pr(bool kell = 0) :kell(kell)    {}
  std::ostream& operator()()       { return std::cout; }
  std::ostream& operator[](int)    { return std::cout << sor << std::endl; }
  ~Pr() {         if (kell){ std::cout << std::endl; sor++; }          }
};
int Pr::sor = 0;

class Adat {
  int a;
public:
  Adat(int i = 3) :a(i)             { Pr()() << a << 'k'; }
  Adat(const Adat&) :a(2)           { Pr()() << a << 'c'; }
  Adat& operator=(const Adat& m)    { Pr()() << m.a << 'e'; return *this; }
  ~Adat()                           { Pr()() << 'd'; }
};
class Ma {
  Adat a;
public:
  Ma(const char *s = "A", int al = 2) :a(al) { Pr()() << s << "K"; }
  ~Ma()                             { Pr()() << "D"; }
};
int main() {
  static Pr end(3);                    Pr(1);
  Ma icC("ABCDEF", 3);                 Pr(1);
  Ma b1 = icC;                         Pr(1);
  b1 = icC;                            Pr(1);
  Ma *ma = new Ma[1];                  Pr(1);
  Pr()[2];
  return 0;
}
```

3	k	A	B	C	D	E	F	K						
2	c													
3	e													
2	k	A	K											
5														
D	d	D	d											

2. Feladat **Σ 3 pont**

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

Egy lehetséges megoldás:

```
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(); }
};
```

3. Feladat Σ 4 pont

Készítsen egy olyan **fix méretű** generikus tárolót (*FixTar*), ami rendelkezik a 2. feladatban megadott tulajdonságokkal, azaz alkalmazható rá a 2. feladatban elkészített adapter! 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 legetséges megoldás:

```
template<class T, size_t siz = 1000>
class FixTar {
    T tar[siz];
    size_t count;
public:
    FixTar():count(0) {}
    class iterator {
        T* p;
    public:
        iterator(FixTar* e, size_t i=0):p(e->tar+i) {}
        const iterator& operator++() { p++; return *this; }
        bool operator!=(const iterator& i2) {return p != i2.p;}
        T& operator*() { return *p; }
    };
    iterator begin() { return iterator(this); }
    iterator end() { return iterator(this,count);}
    void push_back(const T& x) {
        if (count == siz) throw std::out_of_range("FixTar::push_back");
        tar[count++]=x;
    }
    void pop_back();
    bool empty();
    size_t size();
};
```

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 tagfüggvény) megoldás:

```
template<class T, size_t siz = 1000>
class FixTar2 {
    T tar[siz];
    size_t count;
public:
    typedef T* iterator;
    FixTar2():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; }
};
```

4. Feladat Σ 4 pont

Klónozhatónak nevezünk egy objektumot, ha létezik a *clone()* tagfüggvénye, ami dinamikusan létrehoz egy eredetivel megegyező típusú és tartalmú objektumot, és visszaadja annak címét. Tételezze fel, hogy rendelkezésére áll a *ClonableEvent* osztály, ami klónozható.

a) A 3. feladatban elkészített Tar osztály felhasználásával **készítsen** egy olyan klónozható tárolót (*ClonableCollection*), ami *ClonableEvent* pointereket tárol és klónozható, azaz a *clone()* tagfüggvénye úgy másolja le önmagát, hogy a pointerekkel hivatkozott objektumokat is lemásolja. Ügyeljen arra, hogy a tároló megszüntetésekor a tárolt objektumok is megszűnjenek! Az alábbi kódrészlet az tároló használatát mutatja:

```
ClonableCollection store;
store.push_back(new ClonableEvent);
store.push_back(new ClonableEvent);
ClonableCollection* masolat = store.clone();
delete masolat;
```

b) Valósítsa meg a ClonableEvent osztályt úgy, hogy az alapját képezhesse (alaposztálya lehessen) egy klónozható heterogén kollekciónak, azaz a ClonableEvent osztályból származtatott osztály példányai tárolhatók legyenek a store-ban!

Egy legetséges megoldás:

```
class ClonableCollection : public FixTar<ClonableEvent*> {
public:
    ClonableCollection* clone() {
        ClonableCollection *tmp = new ClonableCollection;
        for (iterator i = begin(); i != end(); ++i)
           tmp->push_back((*i)->clone());
        return tmp;
    }
    ~ClonableCollection() {
        for (iterator i = begin(); i != end(); ++i)
           delete *i;
    }
};
```

```
struct ClonableEvent {
    virtual ClonableEvent* clone() { return new ClonableEvent(*this); }
    virtual ~ClonableEvent() {}
};
```

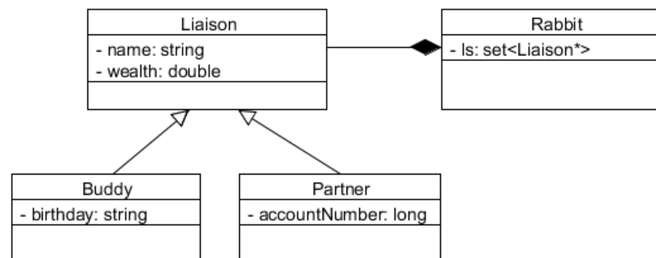
5. Feladat Σ 6 pont

Nyuszinak (*Rabbit*) sok ismerőse (*Liaison*) van, akik vagy barátok (*Buddy*), vagy üzletfelek (*Partner*), de ez később bővülhet. Minden ismerősnek van neve (*name*) és több-kevesebb vagyona (*wealth*), utóbbinak egysége a váltópénz nélküli tallér. Nyuszi nem kispályás, sötét ügyeihez (*doPrank* metódus) időnként mindenkitől kér pénzt (*borrow* metódus). Ekkor a barátok vagyonuk felét adják neki, az üzletfelek 5%-ot, de maximum 10 tallért. A barátoknak születésnapja is van (*birthday*) valamilyen szöveges formátumban tárolva, az üzletfeleknek pedig számlaszáma (*accountNumber*), ami egy nagy egész szám. Nyuszival meg lehet ismertetni új személyeket (*meet* metódus). Ha Nyuszi lebukik (megsemmisül), akkor magával rántja a teljes brancsot. Nyuszi fel tudja sorolni (ostream-re listázni) névsor szerint az ismerőseit (*list* metódus), minden adatukkal együtt.

Feladatok:

- **Tervezzen** a fenti mesére olyan OO modellt, amely könnyen bővíthető további ismerősökkel! **Rajzolja fel** a modell osztálydiagramját! Ne tüntesse fel a tagfüggvényeket az UML ábrán! Használja a dőlt betűs neveket!
- **Deklarálja** és **implementálja** a *Rabbit*, *Liason*, *Buddy*, *Partner* osztályokat és tagfüggvényeit! A *Rabbit* osztály ne legyen másolható, és az értékadás művelet használata is okozzon fordítási hibát! A vagyon legyen privát tag! A *doPrank* metódus térjen vissza a kölcsönkért teljes összeggel! Használja a dőlt betűs neveket!
- **Írjon** egy olyan kódrészletet, amiben Nyuszi megismerkedik néhány illetővel, valami sötét ügyet intéz, majd felsorolja az ismerősöket (névsorban)!

A feladat megoldásához használhat STL algoritmust és/vagy tárolót.




```
class Liaison {
    string name;
    double wealth;
public:
    Liaison(string n, double w) :name(n),  wealth(w) {}
    double getWealth() const { return wealth; }
    string getName() const { return name; }
    void setWealth(double w) { wealth = w; }
    virtual double borrow() = 0;
    virtual ostream& printMe(ostream& os) const {
        return os << name << ":" << wealth;
    }
    virtual ~Liaison(){}
};

class Buddy : public Liaison {
    string birthday;
public :
    Buddy(string n, double w, string bd) : Liaison(n, w), birthday(bd) {}
    double borrow() {
        double w = getWealth()/2;
        setWealth(w);
        return w;
    }
    ostream& printMe(ostream& os) const {
        return Liaison::printMe(os) << ", " << birthday;
    }
};
```

```
class Partner : public Liaison {
    long accountNumber;
public :
    Partner(string n, double w, long a) : Liaison(n, w), accountNumber(a) {}
    double borrow() {
        double w = getWealth()*0.05;
        if (w > 10) w = 10;
        setWealth(getWealth()-w);
        return w;
    }
    ostream& printMe(ostream& os) const {
        return Liaison::printMe(os) << ", " << accountNumber;
    }
};

// segédobjektum (predikátum) a rendezéshez
struct LiaisonCmp {
    bool operator() (Liaison* v1, Liaison* v2) {
        return v1->getName() < v2->getName();
    }
};

// segédfüggvény (funktor) a kiíráshoz
void printIt(const Liaison* lp) { lp->printMe(cout) << endl; }

class Rabbit {
    set<Liaison*, LiaisonCmp> ls;// a set rendezetten tárol, de lehetett rendezni is
    Rabbit(const Rabbit&);
    Rabbit& operator=(const Rabbit&);
public:
    typedef set<Liaison*, LiaisonCmp>::iterator iter;
    Rabbit() {}
    void meet(Liaison* l) {
        ls.insert(l);
    }
    void list() { // nem kell utólag rendezni, mert rendezetten tárolunk
       for_each(ls.begin(), ls.end(), printIt);
    }
    double doPrank() { // itt az accumulate használata kézenfekvő lenne
        double d = 0;
        for (iter it = ls.begin(); it != ls.end(); it++)
            d+= (*it)->borrow();
        return d;
    }
    ~Rabbit() {
        for (iter it = ls.begin(); it != ls.end(); it++)
            delete *it;
    }
};


    Rabbit r;
    r.meet(new Partner("Malacka", 1000, 1234));
    r.meet(new Buddy("Fules", 30, "Majus 5"));
    r.meet(new Buddy("Robert Gida", 50, "Majus 9"));
    cout << r.doPrank() << endl;
    r.list();
```