

Programozás alapjai II.

(7. ea) C++

bejárók és egy tervezési példa

Szeberényi Imre

BME IIT

<szebi@iit.bme.hu>



M U E G Y E T E M 1 7 8 2

Előző óra összefoglalása /1

- A C-ben megtanult preprocessor trükkökkel általánosíthatók az osztályok
- Nem biztonságos, és nem ad mindenre megoldást.
- → Nyelvi elem bevezetése: template
- A preprocessoros trükköt csak a működés jobb megértéséhez néztük meg, ma már nem illik használni.

Előző óra összefoglalása /2

- Generikus osztályokkal és függvényekkel általános szerkezetekhez jutunk:
 - Típust paraméterként adhatunk meg.
 - A generikus osztály v. függvény később a típusnak megfelelően példányosítható.
 - A specializáció során a sablonból az általánostól eltérő példány hozható létre.
 - A függvényparaméterekből a konkrét sablonpéldány levezethető.
 - Függvénytípus sablon átdefiniálható.

Generikus tömb és rendezés (ism)

```
template <class T, int s>
class Array {
    T t[s];
public:
    T& operator[](int i);
};
```

```
Array<int, 10> i10;
Array<double, 5> d5;
Array<char*, 20> cp20;
```

```
template <class T>
void rendez (T a[], int n) {
    for (int i = 1; i < n; i++) {
        T tmp = a[i]; int j = i-1;
        while (j >= 0 && a[j] > tmp) {
            a[j+1] = a[j]; j--;
        }
        a[j+1] = tmp;
    }
}
```

```
int t[100];
rendez<int>(t, 100);
```

Predikátum ism.

- Logikai függvények, vagy függvény-objektumok, amelyek befolyásolják az algoritmus működését
- Predikátum megadása
 - Sablonparaméterként:

```
template <class T, bool cmp(T, T)>
void rendez (T a[], int n);
```
 - Függvényparaméterként:

```
template <class T, class S>
void rendez (T a[], int n, S cmp);
```

Predikátum átadás 1.a (ism)

```
template <class T, class S>
void rendez (T a[], int n) {
    for (int i = 1; i < n; i++) {
        T tmp = a[i]; int j = i-1;
        while (j >= 0 && S::cmp(a[j], tmp)) {
            a[j+1] = a[j]; j--;
        }
        a[j+1] = tmp;
    }
}
```

```
template<class T> class less { // kisebb művelet
public:
    static int cmp(T a, T b) { return a < b; }
};
```

```
rendez<int, less<int> >(t, 6);
```

Predikátum átadás 1.b (ism)

```
template <class T, bool cmp(T, T) >
void rendez (T a[], int n) {
    for (int i = 1; i < n; i++) {
        T tmp = a[i]; int j = i-1;
        while (j >= 0 && cmp(a[j], tmp)) {
            a[j+1] = a[j]; j--;
        }
        a[j+1] = tmp;
    }
}
```

```
template<class T> bool fv(T a, T b) { return a > b; }
template<>
bool fv<char*>(char* a, char* b) { //specializáció
    return strcmp(a, b) >= 1;
}
```

```
rendez<int, fv<int> >(t, 6);
rendez<char*, fv<char*> >(txt, 3);
```

Predikátum átadás 2.a (ism)

```
template<class T, class S>
T kivalaszt(T a[], int n, S cmp) {
    T tmp = a[0];
    for (int i = 1; i < n; i++)
        if (cmp(a[i], tmp)) tmp = a[i];
    return tmp;
}
template<class T> bool fv(T a, T b) {
    return a > b;
}
```

ilyen fv. poi

```
int t[6] = { 1, 2, -8, 0, 12, 3 };
kivalaszt<int, bool (*)(int,int)>(t, 6, fv<int>);
kivalaszt(t, 6, fv<int>);
```


Predikátum átadás 2.b (ism)

Objektum függvényhívás operátorával:

```
template<class T> bool hasonlitFv(T a, T b) {  
    return a > b;  
}  
template<class T>  
struct HasonlitObj {  
    bool operator() (T a, T b) { return a > b; }  
};
```

Funktor

```
int t[6] = { 1, 2, -8, 0, 12, 3 };  
kivalaszt(t, 6, hasonlitFv<int>);  
kivalaszt(t, 6, hasonlitObj<int>());
```

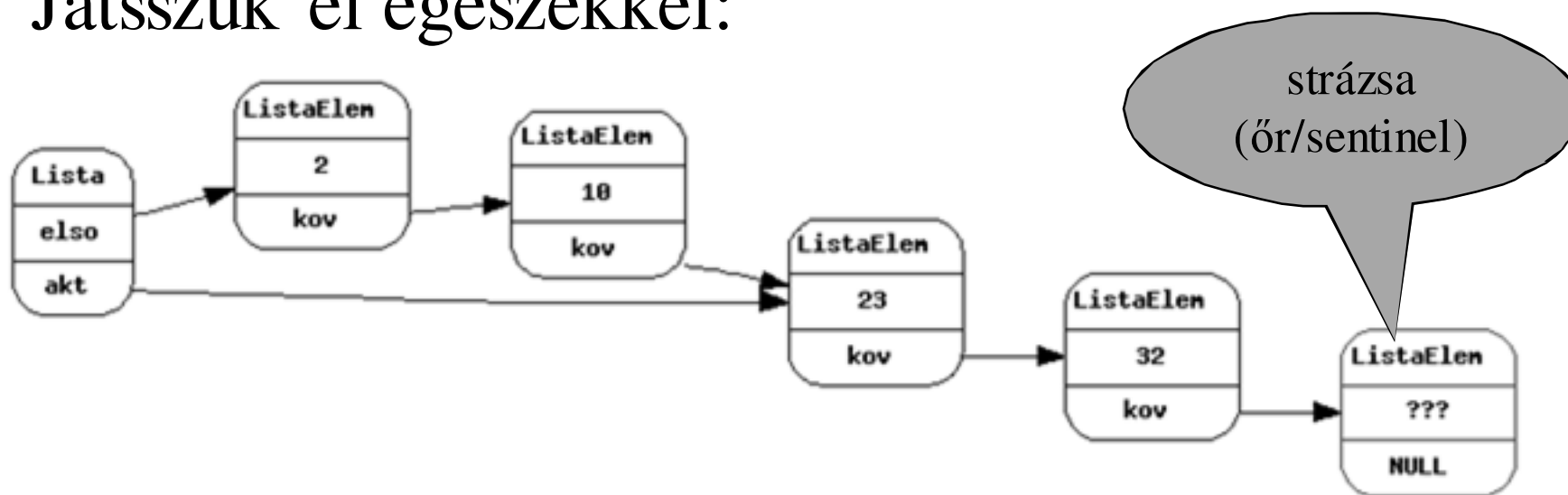
Objektum fv.
operátora.
Példány kell.

Összetettebb példa: Lista sablon

Műveletek:

- beszur() – új elem felvétele
- kovetkezo() – soron következő elem kiolvasása
 - jelzi, ha elérte a végét és újra az elejére áll.

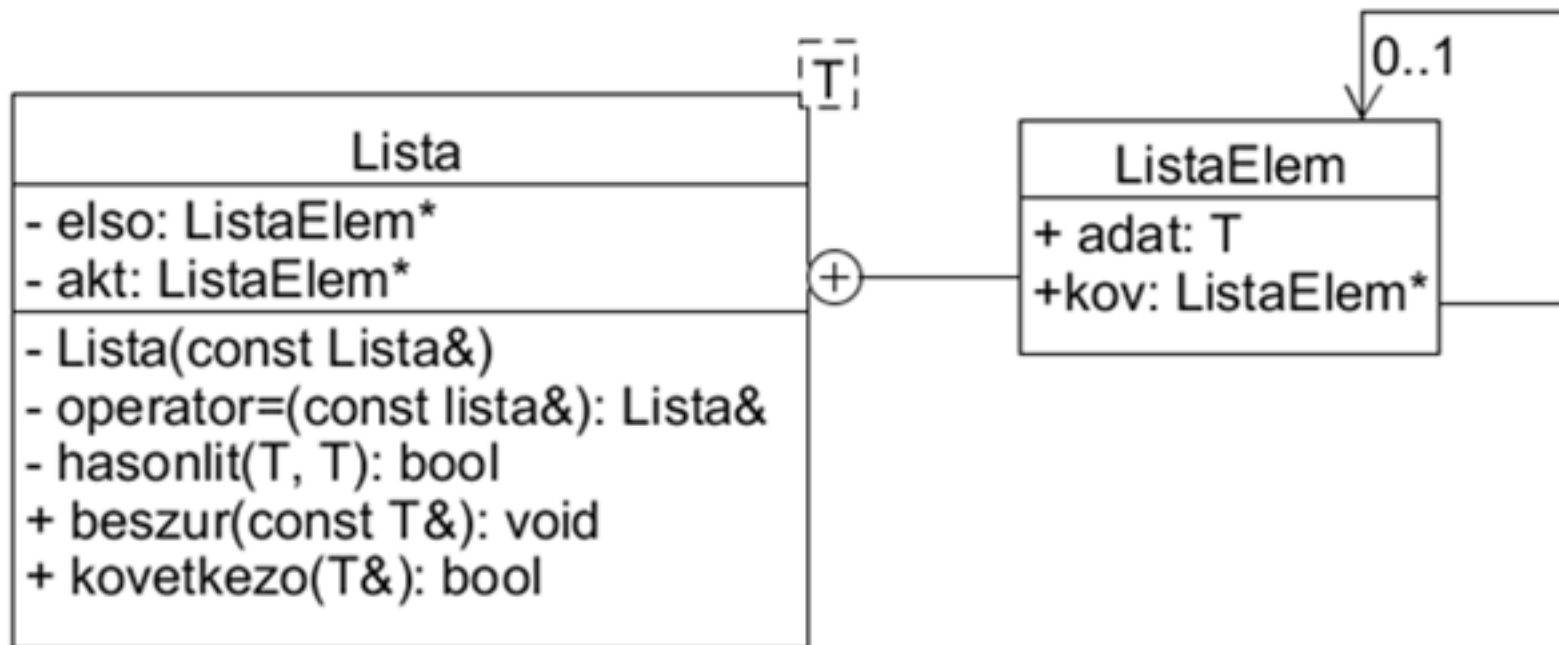
Játsszuk el egészekkel:



Lista tervezése

- Két osztály:
 - Lista
 - pointer az első elemre (első elem)
 - pointer az akt elemre
 - Művelet: beszur(), kovetkezo()
 - ListaElem
 - adat
 - pointer önmagára
 - Művelet: másol, létrehoz

A két osztály kapcsolata



Lista osztály sablonja

```
template <class T> class Lista { // Lista osztálysablon
    struct ListaElem { // privát struktúra
        T adat; // adat
        ListaElem *kov; // következő elem
        ListaElem(ListaElem *p = NULL) :kov(p) {}
    };
    ListaElem *elso, *akt; // első + akt pointer
    bool hasonlit(T d1, T d2) { return(d1<d2); }
public:
    Lista() { akt = elso = new ListaElem; } // első + akt.
    void beszur(const T& dat); // elem beszúrása
    bool kovetkezo(T& dat); // következő elem
    ~Lista() { /* házi feladat */ };
};
```

Tagfüggvények sablonja

```
template <class T> // tagfüggvénytípus sablon
void Lista<T>::beszur(const T& dat) {
    ListaElem *p; // futó pointer
    for (p = elso; p->kov != NULL &&
         hasonlit(p->adat, dat); p = p->kov);
    ListaElem *uj = new ListaElem(*p); //régit ámásolja
    p->adat = dat; p->kov = uj; // új adat beírása
}
template <class T> // tagfüggvénytípus sablon
bool Lista<T>::kovetkezo(T& dat) { // következő elem
    if (akt->kov == NULL) { akt = elso; return(false); }
    dat = akt->adat; akt = akt->kov;
    return(true);
}
```

Lista sablon használata

```
#include "generikus_lista.hpp" // sablonok
int main()
{
    Lista<int> L;           // int lista
    Lista<double> Ld;      // double lista
    Lista<const char*> Ls; // const char* lista

    L.beszur(1); L.beszur(19); L.beszur(-41);
    Ls.beszur("Alma"); Ls.beszur("Hello"); Ls.beszur("Aladar");

    int x; while (L.kovetkezo(x))
        std::cout << x << std::endl;
    const char *s; while (Ls.kovetkezo(s))
        std::cout << s << std::endl;
    return 0;
}
```

sablon példányosítása

Lista<int> L; // int lista

Lista<double> Ld; // double lista

Lista<const char*> Ls; // const char* lista

L.beszur(1); L.beszur(19); L.beszur(-41);

Ls.beszur("Alma"); Ls.beszur("Hello"); Ls.beszur("Aladar");

int x; while (L.kovetkezo(x))

std::cout << x << std::endl;

const char *s; while (Ls.kovetkezo(s))

std::cout << s << std::endl;

return 0;

}

Jól fog működni ?

```
bool hasonlit(T d1, T d2) {
    return(d1<d2);
} // char* < char*
```

Specializációval

```
#include "generikus_lista.hpp" // sablonok
#include <cstring>

template<>
bool Lista<const char*>::hasonlit(const char *s1, const char *s2) { // spec.
    return(std::strcmp(s1, s2) < 0);
}

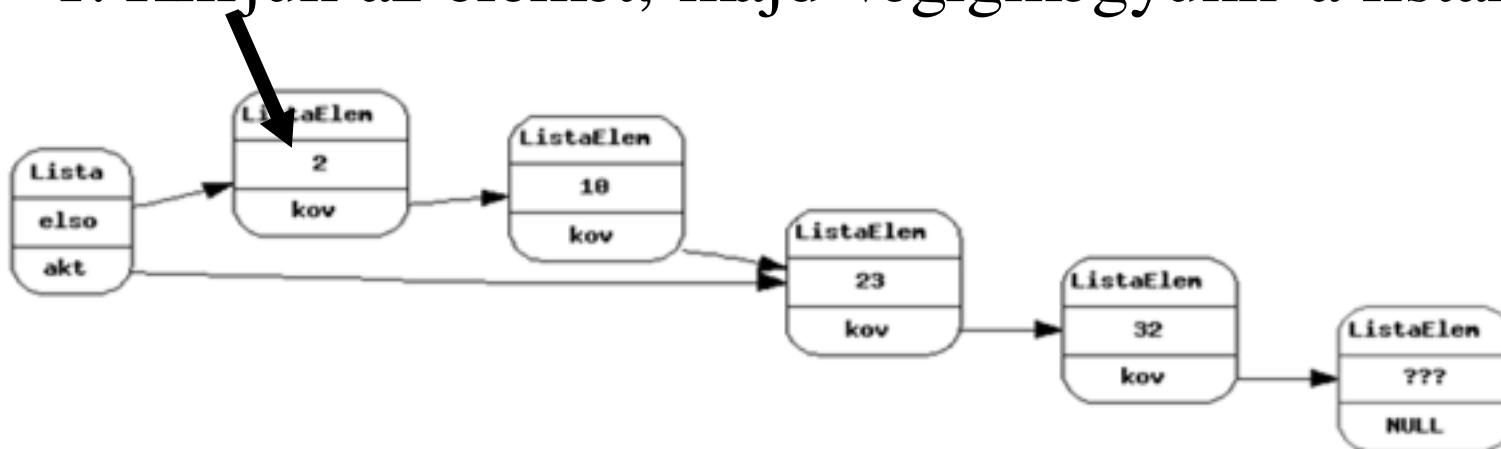
int main() {
    Lista<int> L;           // int lista
    Lista<double> Ld;      // double lista
    Lista<const char*> Ls; // char* lista
    L.beszur(1); L.beszur(19); L.beszur(-41);
    Ls.beszur("Alma"); Ls.beszur("Hello"); Ls.beszur("Aladar");
    int x; while (L.kovetkezo(x)) std::cout << x << std::endl;
    const char *s; while (Ls.kovetkezo(s)) std::cout << s << std::endl;
    return 0;
}
```

Így már ábécé szerint rendez.

Lista sablon felülvizsgálata

- Írjuk ki minden elemhez, hogy mely további elemet oszt maradék nélkül!

1. Kiírjuk az elemet, majd végigmegyünk a listán.



2. Kiírjuk a következő elemet, de melyik a következő ?

Lista sablon felülvizsgálata /2

- Tegyük bele újabb pointert?
 - Mégis hányat?
- Adjuk ki valahogy az adat pointerét?
 - Ekkor ismernünk kell a belső szerkezetet.
- Megoldás:

Olyan általánosított mutató, ami nem ad ki felesleges információt a belső szerkezetéről.

→ Bejáró (iterátor)

Bejárók (iterátorok)

- Általánosított adatsorozat elemeire hivatkozó elvont mutatóobjektum.
- Legfontosabb műveletei:
 - éppen akt. elem elérése (* ->)
 - következő elemre lépés (++)
 - mutatók összehasonlítása (==, !=)
 - mutatóobjektum létrehozása az első elemre (begin())
 - mutatóobj. létrehozása az utolsó utáni elemre(end())

```
Lista<int> li;  
Lista<int>::iterátor i1, i2;  
for (i1 = li.begin(); i1 != li.end(); i1++)  
    int x = *i1;
```

Tároló objektum (lista)

Újabb absztrakciós eszköz

- Általánosan kezelhetjük a tárolókat, azok belső megvalósításának ismerete nélkül.
- Példa: előző feladatot ismeretlen szerkezetű tárolóban tárolt elemekkel akarjuk elvégezni:

```
Tarolo<int> t; Tarolo<int>::iterator i1, i2;  
for (i1 = t.begin(); i1 != t.end(); ++i1) {  
    cout << *i1 << " osztja a kovetkezoet:";  
    i2 = i1;  
    for (++i2; i2 != t.end(); ++i2)  
        if (*i2 % *i1 == 0) cout << " " << *i2;  
    cout << endl;  
}
```

Generikus tömb iterátorral

```
template <class T, int siz = 6>
class Array {
    T t[siz]; // elemek tömje (statikus)
public:
    class iterator; // elődeklaráció, hogy már itt ismert legyen
    iterator begin() { // létrehoz egy iterátort és az elejére állítja
        return iterator(*this);
    }
    iterator end() { // létrehozza és az utolsó elem után állítja
        return iterator(*this, siz);
    }
    class iterator { osztályon belüli osztály a következő dián .....
```

Generikus tömb iterátorral /2

Az osztályon belül van!

```
class iterator {  
    T *p, *pe; // pointer az akt elemre, és az utolsó utánira  
public:  
    iterator() :p(0), pe(0) {}  
    iterator(Array& a, int ix = 0) :p(a.t+ix), pe(a.t+siz) {}  
    iterator& operator++() { // növeli az iterátort (pre)  
        if (p != pe) ++p;  
        return *this;  
    }  
    bool operator!=(const iterator &i) { // összehasonlítja  
        return(p != i.p);  
    }  
    T& operator*() { // indirekció  
        if (p != pe) return *p;  
        else throw out_of_range ("Hibas indirekcio");  
    }  
}; // iterátor belső osztály vége  
}; // Array template osztály vége
```

Generikus tömb használata

```
int main() {  
    Array<int> a1, a2;  
  
    int i = 1;  
    for (Array<int>::iterator i1 = a1.begin(); i1 != a1.end(); ++i1)  
        *i1 = i++;  
  
    return 0;  
}
```

elejére áll!

utolsó utáni elemre

1 2 3 4 5 6

```
int& operator*() {  
    if (p != pe) return *p;  
    else throw out_of_range (...);  
}
```

```
iterator& operator++() {  
    if (p != pe) ++p;  
    return *this;  
}
```

Generikus lista iterátorral

```
template<class T> class Lista {
    struct ListaElem {          // privát struktúra
        T adat;                 // adat
        ListaElem *kov;        // pointer a következőre
        ListaElem(ListaElem *p = NULL) :kov(p) {}
    };
    ListaElem *első;           // pointer az elsőre
    bool hasonlit(T d1, T d2) { return(d1<d2); }
public:
    Lista() { első = new ListaElem;} // strázsa létrehozása
    void beszur(const T& dat);       // elem beszúrása
    class iterator;                  // elődeklaráció
    iterator begin() {               // létrehoz egy iterátort és az elejére állítja
        return(iterator(*this)); }
    iterator end() {                 // létrehozza és az utolsó elem után állítja
        return(iterator()); }
};
```

nincs akt

Generikus lista iterátorral /2

// Lista osztály deklarációjában vagyunk...

```
class iterator {           // belső osztály
    ListaElem *akt;       // mutató az aktuális elemre
public:
    iterator() : akt(NULL) {}; // végére állítja az iterátort
    iterator(const Lista& l) : akt(l.első) { // elejére állítja
        if (akt->kov == NULL) akt = NULL; // strázsa mi
    }
    iterator& operator++() { // növeli az iterátort (pre)
        if (akt != NULL) {
            akt = akt->kov; // következőre
            if (akt->kov == NULL) akt = NULL; // strázsa miatti trükk
        }
        return(*this);
    }
}
```

Itt az akt

Így egyszerűbb,
mint a végéig menni.

Generikus lista iterátorral /3

Nem referencia. Miért ?

```
iterator operator++(int) { // növeli az iterátort (post)
    iterator tmp = *this;    // előző érték
    operator++();           // növel
    return(tmp);            // előzővel kell visszatérni
}
bool operator!=(const iterator &i) const { // összehasonlít
    return(akt != i.akt);
}
T& operator*() {           // indirekció
    if (akt != NULL ) return(akt->adat);
    else throw out_of_range("Hibás");
}
T* operator->() {          // indirekció
    if (akt != NULL) return(&akt->adat);
    else throw out_of_range("Hibás");
} };
```

Címet kell, hogy adjon

Lista használata

```
#include "generikus_lista_iter.hpp"
```

```
int main()    {  
    Lista<int> L;  
    Lista<int>::iterator i;  
    for (i = L.begin(); i != L.end(); i++)  
        int x = *i;  
  
    Lista<Komplex> Lk;  
    Lista<Komplex>::iterator ik(Lk);  
    ik->Abs();  
    Komplex k1 = ik->;    // hibás  
    return(0);  
}
```

A végéig megy, nem kell tudni, hogy valóban milyen adat.

növel

aktuális elem elérése

-> egyoperandusú utótag operátor.
A formai köv. miatt a tagnevet ki kell írni !

Bejárók - összefoglalás

- Tárolók -> adatsorozatok tárolása
 - adatsorozat elemeit el kell érni
 - tipikus művelet: *"add a következőt"*
- Iterátor: általánosított adatsorozat elemeire hivatkozó elvont mutatóobjektum.
- Legfontosabb műveletei:
 - éppen akt. elem elérése (* ->)
 - következő elemre lépés (++)
 - mutatók összehasonlítása (==, !=)
 - mutatóobjektum létrehozása az első elemre (**begin()**)
 - mutatóobjektum létrehozása az utolsó utáni elemre (**end()**)

Bejárók – összefoglalás /2

- Nem kell ismerni a tároló belső adatszerkezetét.
- Tároló könnyen változtatható.
- Generikus algoritmusok fel tudják használni.
- Indexelés nem mindig alkalmazható, de iterátor..
- A pointer az iterátor egy speciális fajtája.

```
template<class Iter>
void PrintFv(Iter first, Iter last){
    while (first != last) cout << *first++ << endl;
}
int tarolo[5] = { 1, 2, 3, 4, 5 };
PrintFv<int*>(tarolo, tarolo+5);
Array<double, 10> d10;
PrintFv<>(d10.begin(), d10.end());
```

Bejárók – még egy példa

```
int szamok[] = { 2, 6, 72, 12, 3, 50, 25, 100, 0 };
std::cout << "szamok a tombbol: ";
printfv(szamok, szamok+8);
Lista<int> li;
int *p = szamok; while (*p != 0) li.beszur(*p++);
std::cout << "szamok a listabol: ";
printfv(li.begin(), li.end());
Lista<int>::iterator i1, i2;
for (i1 = li.begin(); i1 != li.end(); ++i1) {
    std::cout << std::setw(3) << *i1 << " osztja: ";
    i2 = i1;
    for (++i2; i2 != li.end(); ++i2)
        if (*i2 % *i1 == 0) std::cout << " " << *i2;
    std::cout << std::endl;
}
```

Egy tervezési példa

Sharks & Fishes



Példa: Cápák és halak /1

- Modellezzük halak és cápák viselkedését az óceánban.
- Óceán: 2d rács. Cella: szabad, lehet benne hal vagy cápa.
- Kezdetben halak és cápák véletlen-szerűen helyezkednek el.
- Diszkrét időpillanatokban megvizsgáljuk a populációt és a viselkedésüknek megfelelően változtatjuk azt.

Cápák és halak /2 – szabályok

- Hal:
 - Átúszik a szomszédos szabad cellába, ha van ilyen.
 - Ha elérte a szaporodási kort, akkor a másik cellába történő úszás közben szaporodik: Eredeti helyén hagy egy 0 éves halat.
 - Ha nincs szabad cella, nem úszik és nem szaporodik.
 - Sohasem döglük meg.

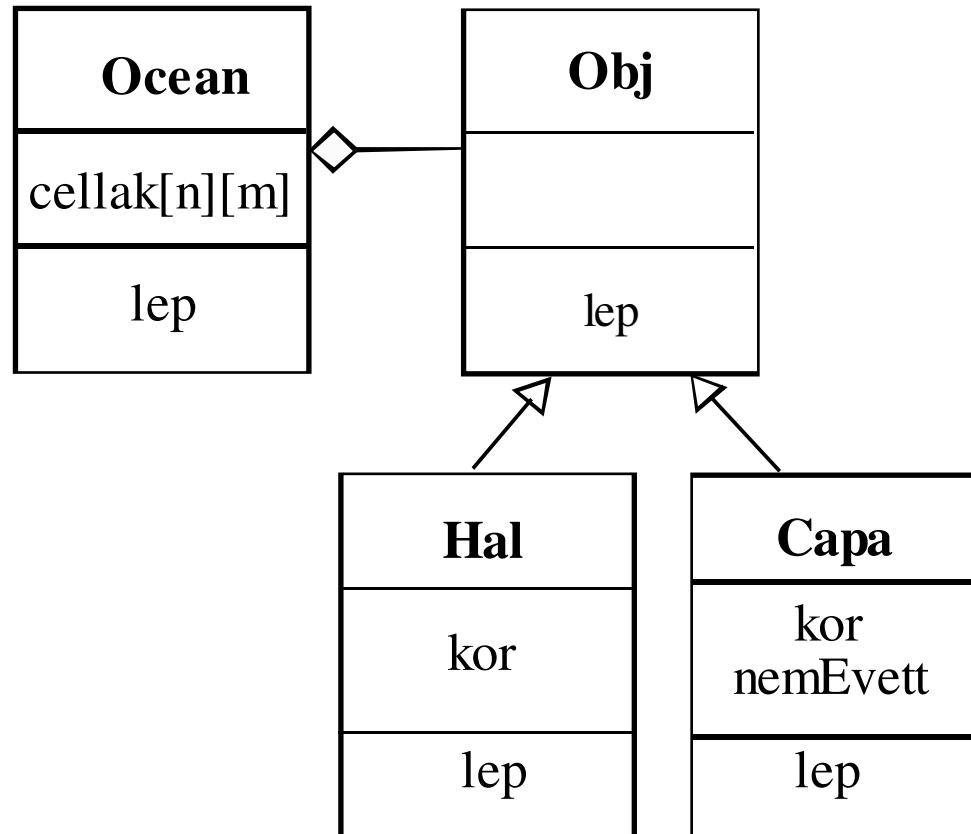
Cápák és halak /3 – szabályok

- Cápák:
 - Ha van olyan szomszédos cella, amiben hal van, akkor átúszik oda és megeszi.
 - Ha nincs hal a szomszédban, de van szabad cella, akkor oda úszik át.
 - Ha elérte a szaporodási kort, akkor a másik cellába történő úszás közben szaporodik: Eredeti helyén hagy egy 0 éves, éhes cápát.
 - Ha nincs szabad cella, nem úszik és nem szaporodik.
 - Ha egy adott ideig nem eszik, megdöglik.

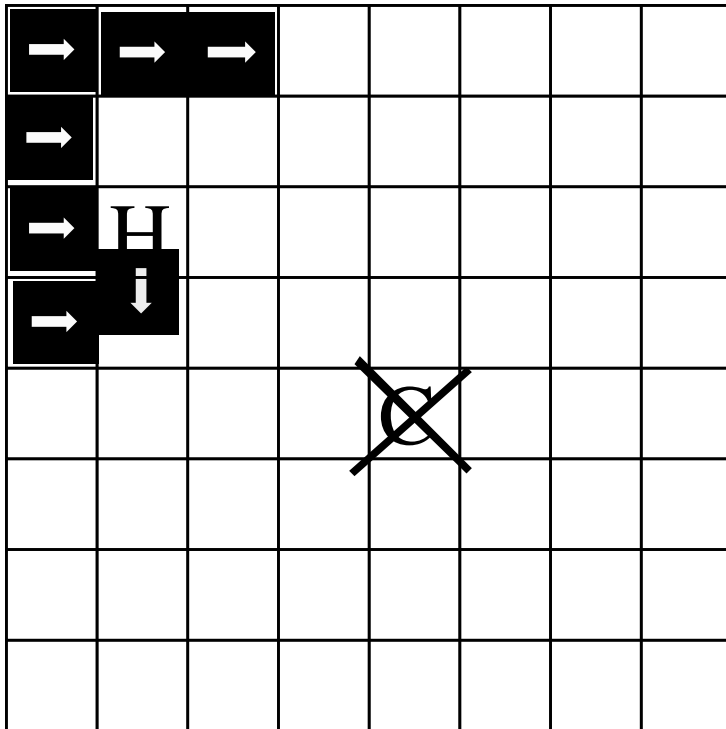
Modellezés: heterogén adatszerk.

- Óceán olyan alapobjektumra mutató pointert tárol mely objektumból származtatható hal, cápa, stb. (Heterogén kollekció.)
- Óceán ciklikusan bejárja a tárolót és a pointerok segítségével minden objektumra meghív egy metódust, ami a viselkedést szimulálja.
- Minden ciklus végén kirajzolja az új populációt.

Első statikus modell



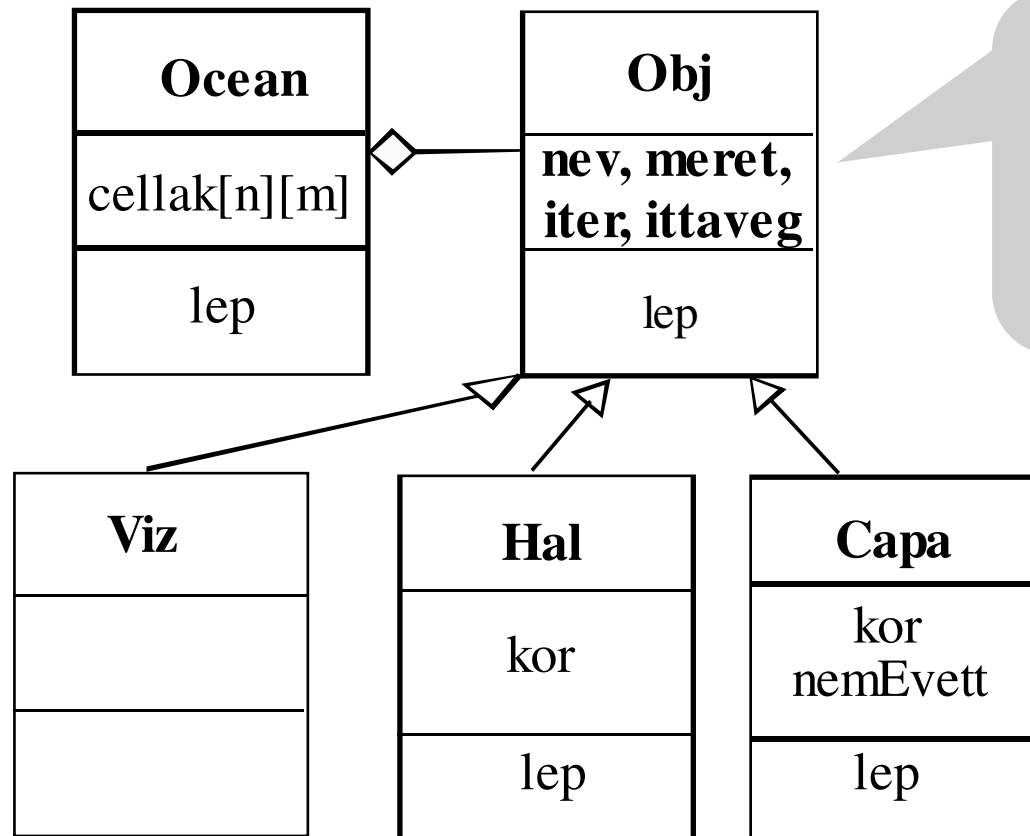
Algoritmusok



Problémák, kérdések

- Egy iterációs ciklusban csak egyszer léptessük.
 - kell egy számláló az ősbé
- A cápa felelőssége önmaga megszüntetése?
 - kell egy hullabegyűjtő (Ocean)
- Mi van az üres cellákban ?
 - víz
- Lehetne sziget is:
 - part objektum
- A cápa honnan tudja, hogy megeheti a halat?
 - nagyobb hal megeszi a kisebbet
 - méret értéke (víz <<< part)

Kiegészített statikus modell



A név nem a viselkedés befolyásolására szolgál!

Koordináták kezelése

```
/// Cellarács koordinátáinak kezeléséhez
/// Koord osztály (minden tagja publikus)
struct Koord {
    enum Irany { fel, jobbra, le, balra };
    int i;           /// sor
    int j;           /// oszlop
    Koord(int, int);

    // Adott iránynak megf. lépve új pozíciót ad
    Koord lep(Irany) const;
};
```


Obj

```
typedef bool cmpf_t(int, int);
inline bool kisebb(int a, int b) { return a < b; }
class Obj {
protected:
    char nev;          /// Objektum neve
    int meret;        /// nagyobb eszik...
    int iter;         /// Iteráció számlálója
    bool ittaveg;     /// kimúlást jelző flag
public:
    Obj(char, int);
    char getnev() const;
    char getmeret() const;
    bool is_vege() const;
    Koord keres(Koord&, Ocean&,
                cmpf_t = kisebb) const;
    virtual void lep(Koord&, Ocean&, int) = 0;
    virtual ~Obj() {}; };
```

Obj::keres()

```
Koord Obj::keres(const Koord& pos, Ocean& oc
                  cmpf_t cmp) const {
    Koord talalt = noPos;
    for (int i = 0; i < 4; i++) { // kih. enum-ot
        Koord ujPos = pos.lep(Koord::Irandy(i));
        if (oc.ervenyes(ujPos)) {
            int m = oc.getObj(ujPos)->getmeret();
            if (m == 0) // legkisebb méret
                if (talalt == noPos) talalt = ujPos;
            else if (cmp(m, meret))
                return ujPos; // van 0-tól eltérő
        }
    }
    return talalt;
}
```

Capa

```
/// Cápá
class Capa :public Obj {
    static const int capaSzapKor = 5;
    static const int capaEhenhal = 7;
    int kor;          //< kora
    int nemEvett;    //< ennyi ideje nem evett
public:
    Capa() :Obj('C', 100), kor(0), nemEvett(0) {}

    /// Másoló a szaporodáshoz kell.
    /// Nullázza a kor-t
    Capa(const Capa& h)
        :Obj(h), kor(0), nemEvett(h.nemEvett) {}
    void lep(Koord pos, Ocean& oc, int);
};
```

Capa viselkedése

```
void Capa::lep(Koord& pos, Ocean& oc, int i) {
    if (iter >= i) return; // már léptettük
    iter = i; kor++;      // öregszik
    if (nemEvett++ >= capaEhenhal) {
        ittaveg = true; return;} // éhen halt
    Koord ujPos = keres(pos, oc); //gyengébbet keres
    if (oc.ervenyes(ujPos)) { //van kaja vagy víz
        if (oc.getObj(ujpos)->getmeret() > 0)
            nemEvett = 0; // fincsi volt a kaja
        oc.replObj(ujPos, this); // új cellába úszik
        Obj* o;
        if (kor > capaSzapKor)
            o = new Capa(*this); // szaporodik
        else
            o = new Viz; // víz lesz a helyén
        oc.setObj(pos, o);
    }
}
```

Ocean

```
/// Statikus méretű cellarácsot tartalmaz.
/// Minden cella egy objektum mutatóját tárolja.
const int MaxN = 10;    /// sorok száma
const int MaxM = 40;    /// oszlopok száma
class Ocean {
    int iter;            /// Iteráció sz.
    Obj *cellak[MaxN][MaxM]; /// Cellák tárolója
public:
    Ocean();
    bool ervenyes(Koord&) const;
    Obj* getObj(Koord&) const;
    void setObj(Koord&, Obj*);
    void replObj(Koord&, Obj*);
    void rajzol(std::ostream&) const;
    void lep();
    ~Ocean();
};
```

Ocean::lep()

```
/// Egy iterációs lépés
void Ocean::lep() {
    iter++;
    for (int i = 0; i < MaxN; i++)
        for (int j = 0; j < MaxM; j++) {
            Koord pos(i, j);
            cellak[i][j]->lep(pos, *this, iter);
            // hullák begyűjtése
            if (cellak[i][j]->is_vege())
                replObj(pos, new Viz);
        }
}
// Objektum törlése és pointer átírása
void Ocean::replObj(Koord& pos, Obj* o) {
    delete cellak[pos.i][pos.j];
    cellak[pos.i][pos.j] = o;
}
```


Írjuk ki a halak számát!

- Kinek a dolga ?
 - Óceáné ?
 - Halaké ?
- Be kell járni az óceánt -> bejáró
- Számolás: általánosított számoló template



```
szamol (atlanti.begin(),  
        atlanti.end(), HalnevCmp ('H')) ;  
szamol (atlanti.begin(),  
        atlanti.end(), HalnevCmp ('C')) ;
```


Írjuk ki a halak számát!/2

```
template<class Iter, class Pred>
int szamol(Iter elso, Iter utso, Pred pred) {
    int db = 0;
    while (elso != utso)
        if (pred(*elso++)) db++;
    return db;
}

struct HalnevCmp {
    char refnev;           // referencia név
    HalnevCmp(char nev) :refnev(nev) {}
    bool operator()(const Obj* o) const {
        return o->getnev() == refnev;
    }
};

cout << "Hal:" << szamol(atlanti.begin(),
                        atlanti.end(), HalnevCmp('H'));
```



Ocean kiegészítése iterátorral

```
class Ocean {
...
public:
    class Iterator;
    Iterator begin() {
        return Iterator(*this);
    }

    Iterator end() {
        return Iterator(*this, MaxN*MaxM);
    }
...
}
```

Ocean::Iterator

```
Ocean::Iterator begin() { return Iterator(*this); }
```

```
class Iterator {  
    Obj **p;           // aktuális pointer  
    Obj **pe;         // végpointer  
public:  
    Iterator() :p(0), pe(0) {}  
    Iterator(Ocean& o, int n=0) :p(&o.cellak[0][0]+n),  
                                pe(&o.cellak[0][0]+MaxN*MaxM) {}  
    bool operator!=(Iterator&);  
    bool operator==(Iterator&);  
    Iterator& operator++();  
    Iterator& operator++(int);  
    Obj* operator*();  
    Obj** operator->();  
};
```

Ocean::iterator /2

```
// Pre inkremens
Ocean::Iterator& Ocean::Iterator::operator++() {
    if (p == 0 || p == pe)
        throw out_of_range("Iterator++");
    p++;
    return *this;
}
// Post inkremens
Ocean::Iterator Ocean::Iterator::operator++(int) {
    Iterator tmp = *this;
    if (p == 0 || p == pe)
        throw out_of_range("Iterator++");
    p++;
    return tmp;
}
```

Ocean::iterator /3

```
/// Ocean Iterator csillag
Obj* Ocean::Iterator::operator*() {
    if (p == 0 || p == pe)
        throw out_of_range("Iterator*");
    return *p;
}

/// Ocean Iterator nyil operator
Obj** Ocean::Iterator::operator->() {
    if (p == 0 || p == pe)
        throw out_of_range("Iterator->");
    return p;
}
```