Programozás alapjai II.
(8. ea) C++

bejárók és egy tervezési példa

Szeberényi Imre
BME IIT
<szebi@iit.bme.hu>

Előző óra összefoglalása /1

- A C-ben megtanult preprocesszor trükkökkel általánosíthatók az osztályok
- Nem biztonságos, és nem ad mindenre megolást.
- → Nyelvi elem bevezetése: template
- A preprocesszoros 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.
  - Függvénysablon átdefiniálható.

Előző óra összefoglalása /3

```
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);
```


Előző óra összefoglalása /4

- Az algoritmusok tovább általánosíthatók ún. predikátumokkal.
- Ezek a logikai függvények, vagy függvény-objektumok befolyásolják az algoritmust.

```
template <class T, bool Has(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 && Has(a[j], tmp) {
        a[j+1] = a[j]; j--;
      }
      a[j+1] = tmp;
  }
}
```

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


Előző óra összefoglalása /5

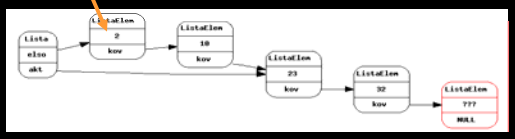
- Maguk a predikátumok is általánosíthatók.

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

```
int t[6] = { 1, 2, -8, 0, 12, 3 };
rendez<int, hasonlitFv<int> >(t, 6);
char *txt[] = { "szilva", "alma", "korte" };
rendez<char*, hasonlitFv<char*> >(txt, 3);
```

Lista sablon felülvizsgálata

- Írjuk ki minden elemhez, hogy mely további elemet oszt maradék nélkül!
  1. Kiírjuk az elemet, majd végig megyünk a listán.



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

Lista sablon felülvizsgálata /2

- Tegyünk bele újabb pointert?
  - Mégis hányat?
- Adjuk ki valahogy a pointert?
  - Legyen a ListaElem publikus?
  - Inkább bele kellene rejtenünk az osztályba!
- Megoldás:
  Olyan általánosított mutató, ami nem ad ki felesleges információt a belső szerkezetről.
  → 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>::iterator 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 kovetkezoket:";
    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) { // összehasonlít
        return(p != i.p);
    }
    T& operator*() {                     // indirekció
        if (p != pe) return *p;
        else throw runtime_error("Hibas indirekcio");
    }
};                                       // iterator 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


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

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


Generikus lista iterátorral

Ezt is bezárjuk a Lista osztályba

```
template<class T> class Lista {
  struct ListaElem {          // privát struktúra (így nem kell friend)
    T  adat;                  // adat
    ListaElem *kov;           // pointer a következőre
    ListaElem(ListaElem *p = 0) :kov(p) {}
  };
  ListaElem *elso;            // pointer az elsőre
public:
  Lista() { elso = new ListaElem;} // strázsa létrehozása
  bool hasonlit(T d1, T d2) { return(d1<d2); }
  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(0) {};     // végére (null) állítja az iterátort
  iterator(const Lista& l) : akt(l.elso) { // elejére állítja
    if (akt->kov == 0) akt = 0; // strázsa miatti trükk
  }
  iterator& operator++() { // növeli az iterátort (pre)
    if (akt) { akt = akt->kov;      // következőre
      if (akt->kov == 0) akt = 0; // 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) return(akt->adat); else throw("Hibás listakezelés");
  }
  T* operator->() {               // indirekció
     if (akt) return(&akt->adat); else throw("Hibás listakezelés");
  }
};
};
```

Címet kell, hogy adjon

Lista használata

```
#include "generic_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());
```


Egy tervezési példa

Sharks & Fishes

Példa: Cápák és halak /1

- Modellezzük halak és cápák viselkedését az óceánban.
- Óceán: 2d rács. Egy cella szabad, vagy 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 döglik meg.

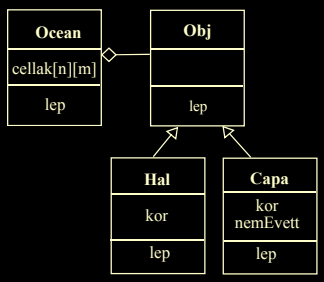
Cápák és halak /3 – szabályok

- Cápa:
  - Ha van olyan szomszédos cella, amiben hal van, akkor átúszik oda és megeszi a halat.
  - Ha nincs hal a szomszédban, de van szabad cella, akkor oda úszik át.
  - Miközben átúszik maga után hagy egy 0 éves, éhes cápát.
  - Ha nincs szabad cella, nem úszik és nem szaporodik.
  - Ha egy adott ideig nem eszik, megdöglik.

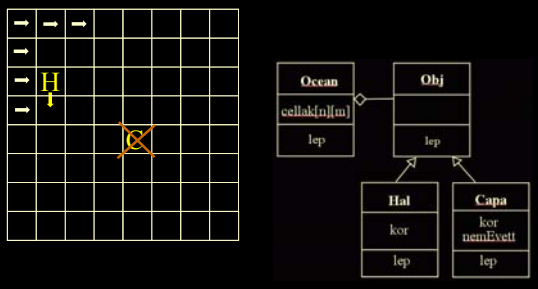
Modellezés: heterogén adatszerk.

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

Első statikus modell



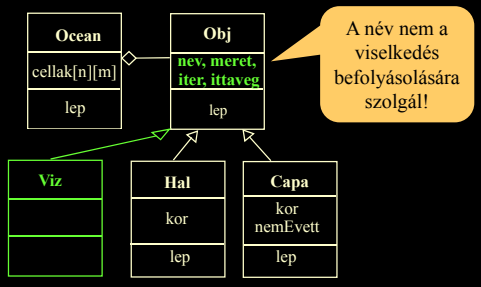
Algoritmusok



Problémák, kérdések

- Egy iterációs ciklusban csak egyszer léptessük.
  - kell egy számláló az ősbe
- 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



Koordináták kezelése

```
/// Cellarács koordinátáinak kezeléséhez

const int MaxN = 10;    /// sorok száma
const int MaxM = 40;    /// oszlopok száma

/// Koord osztály (minden tagja publikus)
struct Koord {
    enum Irany { fel, jobbra, le, balra };
    int i;              /// sor
    int j;              /// oszlop
    Koord(int, int);
    Koord eltol(Irany) const;
    bool ervenyes() const;
};
```


Obj

```
/// Ős.
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&) const;
    virtual void lep(Koord, Ocean&, int);
};
```

Capa

```
/// Cápa
class Capa :public Obj {
    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);
  if (ujPos.ervenyes()) {//van kaja vagy víz
    if (ov.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;
    oc.setObj(pos, o);
  }}
```


Ocean

```
/// Statikus méretű cellarácsot tartalmaz.
/// Minden cella egy objektum mutatóját tárolja.

class Ocean {
    int iter;                 /// Iteráció sz.
    Obj *cellak[MaxN][MaxM]; /// Cellák tárolója
public:
    Ocean();
    Obj* getObj(Koord pos) const;
    void setObj(Koord pos, Obj* o);
    void replObj(Koord pos, Obj* o);
    void rajzol(std::ostream& os) 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;
}
```


Szimuláció (1,5,7)

http://svn.iit.bme.hu/proga2/eloadas_peldak/ea_08/SharksAndFishes