9. labor (Többszörös öröklés)

2011. április 8.

Kivonat

Az első feladat célja bemutatni a technikát, a szintaxist, hogy miként kell tbbszörös öröklést csinálni. A második és harmadik feladat adja meg a motivációt: a kész osztálykönyvtárhoz történő illesztést. **Figyelem: akinek az fstream-es móka kínai, az nézze meg a vonatkozó korábbi, nagyházihoz kijelölt gyakorlófeladatokat, ezeket mi sem fogjuk órán venni!**

1. Órai feladat

Írjunk egy olyen személy (`Person`) osztályt, amely képes elmenteni/betölteni magát illetve összehasonlítani más `Person` típusú objektumokkal!

- Az elmentést/betöltést és az összehasonlítást támogató muveleteket két speciális absztrakt osztályon keresztül örököljük: a kiadott `Serializable` és a `Comparable` osztályokon keresztül. A < operátor implementálásakor az életkort vegyük alapul.
- Az elmentést és a betöltést a már megszokott `ostream` és `istream` osztályokon keresztül végezzük! A `Person` osztály deklarációi rendelkezésre állnak.
- Futtassuk le a kiadott main() függvény tesztjeit!

1.1. A kiadott `Serializable` interfész

Ez egy absztrakt alaposztály (gyakorlatilag egy teljesen üres osztály), ami csak két tisztán virtuális tagfüggvényt tartalmaz. A leszármaztatott osztályoknak ezért azokat felül kell bírálni (definiálni kell).

```
#ifndef SERIALIZABLE_H
#define SERIALIZABLE_H

#include <iostream>

class Serializable
{
public:
  virtual void Save(std::ostream &)=0;
  virtual void Load(std::istream &)=0;
};

#endif // SERIALIZABLE_H
```

1.2. A kiadott `Comparable` interfész

Szintén absztrakt alaposztály, két `Comparable` (*vagy abból származtatott*) objektum között definiálja az egyenlőség operátorát és a kisebb operátort (utóbbi a rendezéshez szükséges).

```
#ifndef COMPARABLE_H
#define COMPARABLE_H


class Comparable
{
public:
  virtual bool operator ==(const Comparable& )=0;
  virtual bool operator <(const Comparable& )=0;
};

#endif //COMPARABLE_H
```

1.3. A `Person` osztály kiadott deklarációja

Adattagok: életkor, súly, magasság. A hozzá való lekérdező („getter”) függvények definiálva vannak, a beállító („setter”) fügvényeknek csak a deklarációja van meg. A súly 0 és 400 kg között, a magasság 10 és 300 cm között lehet, visszatérési értékkel jelezzük, hogy stimmelnek-e. (A konstruktorban nem ellenőrzünk, arra még nem ismerünk jó módszert, hogy a konstruktor hibát tudjon jelezni. Esetleg assert() lehetne benne.)

Az osztály nyilvánosan öröklődik a `Comparable` és a `Comparable` osztályokból (ennek a célját még nem látjuk, de ez a parancs); ez itt egy többszörös öröklődés.

Kénytelenek vagyunk definiálni a fentieken kívül a két ősosztály virtuális függvényeit. Ezzel a Person osztály deklarációja:

```
#ifndef PERSON_H
#define PERSON_H

#include "Comparable.h"
#include "Serializable.h"

class Person: public Serializable, public Comparable
{

   double height;
   double weight;
public:
  unsigned char age;

  Person(unsigned char age,double height, double weight);
  bool SetHeight(double height);
  double GetHeight(){return height;}

  bool SetWeight(double weight);
  double GetWeight(){return weight;}


  bool operator ==(const Comparable& theOther);
  bool operator <(const Comparable& theOther);
  void Save(std::ostream &);
  void Load(std::istream &);
};
#endif PERSON_H
```

1.4. A kiadott tesztprogram (`MultiInh.cpp`)

Érdemes a tesztprogramot végignézni. Teszteli önmagában a fájlba mentést/visszatöltést, majd az összehasonlító operátorokat. A tesztesetek úgy vannak kialakítva, hogy csak helyes működés esetén éri el a program a main() végét. Ebben a programban még semmi jelentőségét nem látjuk a két ősosztálynak, csak azt látjuk, hogy működik a fájlba mentés és az összehasonlítás.

```
#include "Person.h"
#include <fstream>

using namespace std;



//Teszteljük az osztályokat
int main()
{
  // Egy tömb létrehozása
  const unsigned maxPeople=4;
  Person* people[maxPeople];

  people[0]=new Person(12,100,50);
  people[1]=new Person(30,180,85);
  people[2]=new Person(40,182,90);
  people[3]=new Person(40,182,90);

  // Kimentjük a tömböt egy fájlba.
  // !!! Lehet, hogy a HSZK-ban módosítani kell az elérési utat,
  // hogy legyen írási jog. !!!
  char* fname="d:\\user\\data.txt";
  ofstream os(fname);
  if(!os)
  {
    cerr<<"Error opening output file: "<<fname<<endl;
    return -1;
  }

  for(unsigned int i=0;i<maxPeople;i++)
  {
    // A Save függvény tesztje
    people[i]->Save(os);

    // Összezavarjuk a tartalmat, hogy
    // tesztelhessük a Load függvényt
    people[i]->age=0;
    people[i]->SetHeight(20);
    people[i]->SetWeight(10);
  }

  os.close();
  if(!os)
  {
    cerr<<"Error writing output file: "<<fname<<endl;
    return -1;
  }
```

```
  ifstream is(fname);
  if(!is)
  {
    cerr<<"Error opening input file: "<<fname<<endl;
    return -1;
  }

  for(unsigned int j=0;j<maxPeople;j++)
  {
    // A Load függvény tesztje
    people[j]->Load(is);
  }

  is.close();
  if(!is)
  {
    cerr<<"Error reading input file: "<<fname<<endl;
    return -1;
  }

  // Az összehasonlítás tesztje:
  // ezek nem egyenlok
  if(*people[0]==*people[1])
    return -1;

  // Az összehasonlítás tesztje:
  // ezek egyenlok
  if(!(*people[2]==*people[3]))
    return -1;

  // Felszabadítás
  for(unsigned int k=0;k<maxPeople;k++)
  {
    delete people[k];
  }

  // Ide helyezzük a töréspontot.
  // Ha ideáig eljutottunk, a tesztek sikeresen lefutottak.
  return 0;
}
```

1.5. Az általunk készített `Person.cpp`

A megoldás néhány fontos eleme: az összehasonlításnál `Comparable` objektumokat kell átvennünk, de Person-t kell csinálni belőlük. „Lefelé” mozgunk az osztályhierarchiában (downcast), emiatt explicit konverzió előírása szükséges. Lásd az összehasonlító függvényekhez fűzött magyarázatot.

A szerializálás/de-szerializálás egyszerűen úgy történik, hogy pontosvesszővel elválasztva írunk, majd olvasásnál a pontosvesszőlet „lenyeljük” (egyben ellenőrizzük, hogy megvannak-e, vagy helytelen a bemenő fájl formátuma.)

```
#include "Person.h"

using namespace std;
```

```
Person::Person(unsigned char age,double height, double weight)
{
  this->age=age;
  this->height=height;
  this->weight=weight;
}

bool Person::SetHeight(double height)
{
  if(height<10||height>300)
    return false;
  this->height=height;
    return true;
}


bool Person::SetWeight(double weight)
{
  if(weight<0||weight>400)
    return false;
  this->weight=weight;
    return true;
}


bool Person::operator ==(const Comparable& theOther)
{
  Person* pPerson=(Person*)&theOther;

  if(age==pPerson->age && height==pPerson->height && weight==pPerson->weight)
    return true;
  else
    return false;
}

bool Person::operator <(const Comparable& theOther)
{
  Person* pPerson=(Person*)&theOther;

  if(age<pPerson->age)
    return true;
  else
    return false;
}

void Person::Save(ostream & os)
{
    os<<(int)age<<';'<<height<<';'<<weight<<endl;
}

void Person::Load(istream &is)
{
```

```
  double height;
  double weight;
  int age;
  char c;

  is>>age;
  is>>c;
  if(c!=';')is.clear(ios::failbit);
  is>>height;
  is>>c;
  if(c!=';')is.clear(ios::failbit);
  is>>weight;

  if(is)
  {
    this->age=age;
    this->height=height;
    this->weight=weight;
  }
  else
  {
    cerr<<"Error in input format."<<endl;
  }
}
```

2. 1. házi feladat

(forráskódok a szokásos könyvtárban, 2Feladat néven, de a meglevők kiegészítésével egyszerűbb)

Tegyük fel, hogy készítettünk egy kb. 15 rendezőalgoritmusból álló osztályt (`Sorter`), amely tetszőleges típusú objektumokat álló tömböt képes rendezni, és amely forrását nem szeretnénk kiadni, csak a lefordított kódot (statikus programkönyvtár [.lib] formájában), és a meggazdagodás ezek után már csak idő kérdése. Mit írjunk elő a rendezendő objektumoknak? A `Comparable` osztályból való származást! Másként nem tudnám átvenni, hiszen a `void *` által mutatott objektumokra nem hívhatok semmit, nem hogy operátorokat. Konvertálnom kellene, de mire? Ezért van szükség az absztrakt `Comparable` osztályra!

- Írjuk meg a `Sorter` osztály egy rendezőfüggvényét egy tetszőleges rendezőalgoritmussal (pl. buborékrendezés)!
- Készítsük el a rendezendő tömböt, és hívjuk meg rá a rendezőfüggvényt! Figyeljünk, hogy a tömb típusa `Comparable *` legyen (polimorf kezelés a `Comparable *` interfésznél „fogva”)

A Sorter osztály és az egyetlen rendezőfüggvény kerülhet a tesztprogramba, a main() elé, nagyjából a következő formája lehetne:

```
// Ennek a forráskódját nem kapjuk meg, csak a .h állományt
// (.o, .obj., lib., .a, .dll,.so formátumban)
class Sorter
{
public:
  static void Sort(Comparable **pItems, unsigned itemCount)
  {
    // ide jön a (pl. buborék) rendezés
    // használhatjuk a < operátort (ezért definiáltuk Person-ra)
    // de itt Person nem fog megjelenni, csak Comparable objektumok
  }
  // Idejön a további 14 algoritmus
```

```
};
```

Tehát – bár most nem úgy látszik, de – a Sorter definícióját nem látjuk, nem módosíthatjuk, csak azt tudjuk, hogy a vásárolt könyvtárban van egy ilyen függvény.

```
class Sorter
{
public:
  static void Sort(Comparable **pItems, unsigned itemCount);
  // Idejön a további 14 algoritmus deklarációja
};
```

Ha a `Sort` függvény prototípusát elemezzük, akkor látjuk, hogy `Comparable` pointereket tartalmazó tömböt vesz át (plusz a tömb méretét). Gondoljuk végig, hogy miért így választotta a gyártó (azaz mi) az interfészt, és ne felejtsük el a rendezésnél, hogy nem a pointereket (címeket) kell rendezni, hanem az általuk mutatott objektumokat.

A tesztprogramban az alábbi formában tesztelhetjük:

```
Comparable *people[maxPeople]; // !!! most a Comparable interfészt használjuk!!!

Sorter::Sort(people, maxPeople);

for(unsigned int j=0;j < maxPeople;j++)
{
  Person *pPerson = (Person *) people[j]; // ismét downcast szükséges a Person-sághoz
  // ezt cseréljük static_cast-ra vagy dynamic_cast-ra
  cout << (int) pPerson->age<< ' ';
}
```

2.1. Mi az értelme?

Ha kigyógyultunk a skizofréniából (úgy tettünk, mintha nem férnénk hozzá a Sort definíciójához), akkor azt látjuk, hogy a mások által készített osztálykönyvtárhoz (itt a rendezéshez) illesztésnek az az egyetlen módja, hogy a gyártó által megadott interfészt (itt a Comparable) implementálja az osztályunk, és annál az interfésznél fogva végzünk polimorf kezelést.

3. 2. házi feladat

Vásároltunk egy Saver és egy Loader osztályt, amelynek a forráskódja nem áll rendelkezésre. Egészítsük ki a tesztelő programot az osztályok használatával!

Az említett osztályok (a látszattal ellentétben) rendelkezésre nem álló forráskódja (a 3Feladat alatti tesztprogramban megvan):

```
class Saver
{
private:
    string outFileName;
    ofstream ofs;
public:
    Saver( string outFileName): outFileName(outFileName), ofs(outFileName.c_str())
    {
        if(!ofs)
        {
            cerr<<"Error opening output file: " << outFileName << endl;
        }
    }
```

```
    // Wir benötigen Referenz (polymorphe Behandlung)
    // Ein Pointer wäre auch gut, aber nicht elegant
    bool Save(Serializable &item)
    {
        item.Save(ofs);
        return !ofs;
    }

    // Sollte nicht vom Benutzer aufgerufen werden, der Destruktor von ofs
    // wird es automatisch freigeben
    void Close()
    {
        ofs.close();
    }

};

class Loader
{
private:
    string inFileName;
    ifstream ifs;
public:
    Loader(string inFileName):inFileName(inFileName),ifs(inFileName.c_str())
    {
        if(!ifs)
        {
            cerr<<"Error opening input file: " << inFileName << endl;
        }
    }

    // Wir benötigen Referenz (polymorphe Behandlung)
    // Ein Pointer wäre auch gut, aber nicht elegant
    bool Load(Serializable& item)
    {
        item.Load(ifs);
        return !ifs;
    }

    // Sollte nicht vom Benutzer aufgerufen werden, der Destruktor von ifs
    // wird es automatisch freigeben
    void Close()
    {
        ifs.close();
    }

};
```

Ez a két osztály kényelmi funkciókat biztosít: a mentés és visszatöltés műveletét néhány egyszerű függvény meghívásával elvégezhetjük, nem nekünk kell pl. az ofstream-et létrehozni és hibákat ellenőrizni, mint az órai példa esetén. A működés feltétele csak annyi, hogy az elmentendő-visszatöltendő osztály megvalósítsa a `Serializable` interfész függvényeit (Load, Save) – a Person teljesen véletlenül pont tudja is ezt.

A (rendkívül bonyolult) házi feladat: a tesztprogramban az elmentést és visszatöltést ezekkel az osztá-

lyokkal elvégezni.

Megjegyzés: a std::string osztály `c_str()` függvénye – ugye – a tárolt string `const char *` (egyszerű C-sztring) reprezentációját adja.

4. Mi értelme ennek az egésznek?

Ha már előre adott osztályok vannak egy keretrendszerben, akkor sokszor azt írják elő, hogy valamiből leszármazzon az általunk megvalósított osztály. Így implementálhatnak pár műveletet, amit a keretrendszer szeretne majd meghívni. Vagyis úgy illesztettük a keretrendszerhez, hogy (itt) több osztályból származtattunk. Ha ez a célunk, akkor ilyenkor semmilyen implementációt nem tartalmazó, tisztán virtuális függvényeket használjunk, mint ahogy a példák mutatják! Mivel így csak a művelet deklarációját adtuk meg, a leszármazott egyfajta hívható felületet, interfészt implementál. A gyakorlat az, hogy maximum egy osztályból öröklünk implementációt, viszont adott esetben több interfészt is implementálhatunk. Általában interfészt sem szoktunk implementálni, csak ha egy keretrendszerhez akarunk illeszkedni. Érdemes végiggondolnunk, hogy az operátorok túlterhelése egyfajta implicit interfészimplementáció. (Az egészhez lásd a többszörös örökléses előadás végét.)