Informatika 3
7. előadás

Tóth Dávid

Budapesti Műszaki és Gazdaságtudományi Egyetem

2026.03.31.

Projektív sík modellezése

- A K test feletti projektív síkot $K\mathbb{P}^2$ jelöli.
- Egy lehetséges modell: a pontok a K^3 tér 1 dimenziós alterei, az egyenesek a 2 dimenziós alterek, egy pont pontosan akkor illeszkedik egy egyenesre, ha az egyenesnek megfelelő altér tartalmazza a pontnak megfelelő alteret.
- Minden $(x, y, z) \neq (0, 0, 0)$ vektor egy pontot ír le, ahol két vektor pontosan akkor ekvivalens (azaz pontosan akkor adja ugyanazt a pontot), ha egyik a másiknak skalárszorosa.
- Minden ekvivalenciaosztályban, ahol $z \neq 0$, pontosan egy $(x, y, 1)$ alakú pont van, ezek a pontok azonosíthatók a K^2 affin síkkal.
- A többi pontot az $(x, 1, 0)$ osztályai ill. az $(1, 0, 0)$ osztálya reprezentálja (ezek a "végtelen távoli" egyenes pontjai).

Projektív sík modellezése

- Az egyenesek az

$$\{(x, y, z) \in K^3 : \ ax + by + cz = 0\}$$

  alakú ponthalmazok, ahol a, b és c legalább egyike nem 0.
- Egy egyenest tehát leír egy $[a, b, c] \neq [0, 0, 0]$ vektor.
- Két vektor pontosan akkor adja ugyanazt az egyenest, ha egyik a másiknak skalárszorosa.
- Tehát az egyeneseket és a pontokat ugyanúgy reprezentálhatjuk!
- Illeszkedés: a

$$\{(\lambda x, \lambda y, \lambda z) : \ \lambda \in K\}$$

  pontok pontosan akkor vannak az $[a, b, c]$ vektor által reprezentált síkban, ha $[a, b, c] \cdot (x, y, z) = 0$ (itt a szorzás a skalárszorzat).

Pontok és egyenesek implementálása

- Mivel tetszőleges test felett szeretnénk dolgozni, a projektív sík pontjait ill. egyeneseit reprezentáló `PPoint` ill. `PLine` osztályok template osztályok lesznek.
- Az objektumokat leíró adat mindkét esetben egy ponthármas lesz.
- Hogy különböző tárolókban tárolhatók legyenek, mindkét osztály elemeire definiálni szeretnénk a < operátort.
- Bármely két pontra szeretnénk meghatározni azt az egyetlen egyenest, ami tartalmazza őket; bármely két egyenesre szeretnénk meghatározni azok egyetlen metszéspontját.
- Szeretnénk bármely három pontról eldönteni, hogy egy egyenesen vannak-e, ill. bármely három egyenesről, hogy egy ponton mennek-e át.
- Szeretnénk kiírni a képernyőre az objektumokat meghatározó adatot.

Pontok és egyenesek implementálása

- A két objektumot leíró adatok és metódusok, sőt, az azokat megvalósító algoritmusok is szinte teljesen azonosak.
- Hogy ezeket csak egyszer kelljen megvalósítani, az azonos részeket csak egyetlen `PObject` *ősosztályban* valósítjuk meg, a pontokat és egyeneseket pedig ebből fogjuk *származtatni*.
- A megspórolt munkán túl további nyereség a *polimorfizmus:* a pontok és egyenesek projektív objektumként tudnak viselkedni, ha olyan műveletet kell elvégezni, amely értelmes mindkettőjükre.

Öröklés

- Mikor használ(hat)unk öröklést?
- Akkor, amikor egy „minden micsoda micsoda” jellegű relációt szeretnénk kifejezni az objektumok között, tehát ha az A típusú objektumok halmaza tartalmazza a B típusú objektumok halmazát (azaz minden B egyben A is).
- Példák:
  - minen projektív pont/egyenes projektív objektum,
  - minden téglalap alakzat, minden négyzet téglalap (és így alakzat),
- Vannak olyan esetek, amikor a fenti reláció fennáll, de nem érdemes öröklést használni: pl. minden kör ellipszis, de ha az ellipszis osztálynak van olyan tagfüggvénye, ami csak az egyik tengely irányába nyújtja az objektumot, az a kör osztálynak nem lehet tagfüggvénye (hiszen a kör nem tud így nyúlni).

Öröklés

- Az öröklés során egy ősosztályból indulunk ki:

```
class Parent {
private:
    double x, y;
public:
    Parent(double a, double b) { x = a; y = b; }
    double sum()const { return x+y; }
    void print()const {
        cout << "(" << x << "," << y << ")";
    }
};
```

- Ennek van egy egyszerű `public` konstruktora, egy tagfüggvény, ami kiszámolja az adattagok összegét, és egy tagfüggvény, ami ezek értékét kiírja.

Öröklés

- Tekintsük ennek az osztálynak egy leszármazottját:

```
class Child : public Parent {
private:
    int k;
public:
    Child(double a, double b, int n) : Parent(a,b) {
        k = n;
    }
    double value() const { return sum()*k; }
    void print()const {
        Parent::print(); cout << "*" << k;
    }
};
```

- Itt van egy új adattag, egy új `value` függvény, ill. definiáltunk egy új `print` függvényt.

Öröklés

- Az öröklést a következő sor jelzi:

  ```
  class Child : public Parent {
  ```

- A : jelzi, hogy a `Child` osztály leszármazottja egy másik osztálynak, az ősosztály neve a `public` kulcsszó után következik.
- A `public` kulcsszó azt jelenti, hogy az ősosztály publikus részei a leszármazottnak is publikus részei lesznek: egy konstruktor ill. a `sum` és `print` tagfüggvények.
- Ha `private` kulcsszót írnánk, akkor az ősosztály publikus részei a leszármazottnak `private` részei lennének.
- Az ősosztály `private` részei a leszármazott számára nem elérhetők közvetlenül egyik esetben sem.

Öröklés

- Van rá mód, hogy az ősosztály egyes tagváltozóit vagy tagfüggvényeit elrejtsük a külső felhasználók elől, de elérhetővé tegyük a leszármazottak számára.
- Ezeket az ősosztályban `protected` kulcsszó után írjuk:

```
class Parent{
private:
    ...
protected:
    ...
public:
    ...
};
```

Öröklés

- Ha az öröklés `public`, akkor a gyerek számára (az osztályon belülről):
  - az ős private részei közvetlenül nem elérhetők,
  - az ős protected részei protected részek lesznek,
  - az ős public részei public részek lesznek.
- Ha az öröklés `protected`, akkor a gyerek számára (az osztályon belülről):
  - az ős private részei közvetlenül nem elérhetők,
  - az ős protected és public részei protected részek lesznek.
- Ha az öröklés `private`, akkor a gyerek számára (az osztályon belülről):
  - az ős private részei közvetlenül nem elérhetők,
  - az ős protected és public részei private részek lesznek.

Öröklés

- A leszármazott konstruktorában az inicializáló listában meghívhatjuk az ős egy konstruktorát:

  ```
  Child(double a, double b, int n) : Parent(a,b) {...}
  ```

- Az ős konstruktora után vesszővel elválasztva folytatódhat az inicializáló lista:

  ```
  Child() : Parent(), k(0) { }
  ```

Öröklés

```
class Child : public Parent {
private:
    int k;
public:
    Child(double a, double b, int n) : Parent(a,b) {
        k = n;
    }
    double value() const { return sum()*k; }
    void print()const {
        Parent::print(); cout << "*" << k;
    }
};
```

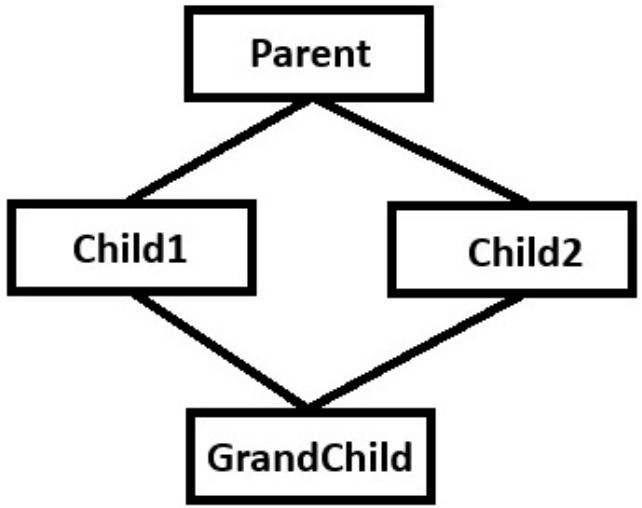
Öröklés

- A `Parent` osztálynak volt egy `print` tagfüggvénye, de a `Child` osztályban *felüldefiniáltuk* ezt.
- Ettől a `Child` osztályban is elérhető a `Parent` osztály `print` függvénye a `Parent::print` névvel.
- Ha template osztályokat írunk, akkor az ős tagfüggvényére az osztály definíciójában a `Parent::nev` nevet akkor is használni kell, ha nem definiáljuk felül a függvényt (ebben az esetben a `this->nev` használata is OK).

Öröklés

- Egy osztályt több osztályból is származtathatunk, ilyen módon létrejöhet pl. a következő „négyszög”:

```
class Parent {...};
class Child1 : protected Parent { ... };
class Child2 : private Parent { ... };
class GrandChild : public Child1, public Child2 {
    ...
};
```



Problémák:

1. Többértelműség:
   - Esetenként a fordító nem képes eldönteni, hogy melyik osztály tagfüggvényét hívjuk, még akkor sem, ha egyébként az egyértelmű $\rightsquigarrow$ fordítási hiba.
   - Pl. ha a `Parent` osztályban definiált az `f()` függvény, és az semelyik leszármazottban sincs felüldefiniálva, akkor azt a `GrandChild` osztályból nem tudjuk szimplán az `f();` szintaxissal meghívni, mert a `Child1` és `Child2` osztálynak is van `f()` függvénye (a `Parent` ősben definiálva), és hiába azonosak, a fordító ezt nem látja
     $\rightsquigarrow$ a jó szintaxis a `Parent::f();`

Problémák:

2. Nehezen követhető konstruktorhívások:
   - A `GrandChild` osztályból nem hívható közvetlenül a `Parent` osztály konstruktora, az a `Child1` és `Child2` osztályokon keresztül hívódik meg (többször!) azok konstruktorában (az öröklés sorrendjében).
   - A `Parent` osztály különböző konstruktorok hívódhatnak meg, nehezen követhető az inicializálás.

   Ha lehetséges, kerüljük a hasonló helyzeteket.

Polimorfizmus

- Tegyük fel, hogy különböző alakzatokat (köröket, téglalapokat stb.) reprezentáló osztályokat szeretnénk megvalósítani.
- Ősosztályként definiálhatunk egy `Alakzat` osztályt, melyből a `Kor`, `Teglalap` stb. osztályokat származtatjuk.
- Ha egy tömbben szeretnénk *különböző* típusú alakzatokat tárolni, akkor a több elemeinek nem lehet kör vagy téglalap a típusa, hiszen itt heterogén kollekcióról van szó.
- Alakzat sem lehet a tömb elemeinek a típusa, mert akkor elveszítjük a körökre és téglalapokra nézve specifikus adatokat.
- A megoldás az alakzatokra mutató pointer, az minden lehetséges típusra nézve közös.

Polimorfizmus

- A következő kód tehát fordul:

```
Kor k;
Teglalap t;
Alakzat* tomb[2];
tomb[0] = &k;
tomb[1] = &t;
```

- Tegyük fel, hogy minden konkrét alakzatra elvárjuk, hogy legyen egy `terulet()` tagfüggvénye.
- Az egyik probléma: a fenti kódban a `tomb[0]->terulet()` függvényhívás esetén honnan tudjuk, hogy nem az `Alakzat` objektum tagfüggvénye hívódik meg, hanem a `Kor` objektumé?
- Az másik probléma: egy általános alakzatnak nem tudjuk kiszámolni a területét, így az `Alakzat` objektumnak nem tudunk `terulet` tagfüggvényt definiálni, de a függvényhíváshoz a nevet deklarálni kell az `Alakzat` osztályban is.

Virtuális függvények

- Megoldás: virtuális függvény.
- Ha a `virtual` kulcsszóval deklaráljuk az ősosztályban a függvényt, az biztosítja, hogy mindig a megfelelő osztály tagfüggvénye hívódjon meg akkor is, ha egy ősre mutató pointeren keresztül hívjuk meg.
- Ha az `Alakzat` osztályban nem akarjuk definiálni a terület tagfüggvényt, azt a következő kóddal tehetjük meg:

```
virtual double terulet() = 0;
```

- Ez jelzi, hogy a leszármazottakban meg kell valósítani a `terulet` tagfüggvényt, másrészt hogy az adott osztályban nincs megvalósítva.
- Ezeket tisztán virtuális függvénynek nevezzük, az ilyeneket tartalmazó osztályok ún. *absztrakt* osztályok, nem példányosíthatók.

Polimorfizmus

- A polimorfizmus referenciáknál is működik: a következő függvény az `Alakzat` osztály tetszőleges leszármazottját megkaphatja paraméterül:

```
void f(const Alakzat& x) {...}
//az Alakzat leszármazottai átadhatók,
//a leszármazott virtuális függvényei meghívhatók
```

- Ezzel szemben ha érték szerint adjuk át a paramétert, akkor `Alakzat` objektummá konvertálódik a leszármazott, elveszik a leszármazottra jellemző adat, más a virtuális tagfüggvények működése!

```
void f(const Alakzat x) {...}
//x Alakzat típusú,
//a leszármazott virtuális függvényei nem hívhatók
```

Öröklés és memóriakezelés

- Tegyük fel, hogy az az `Alakzat` osztályból egy sokszög osztályt akarunk származtatni:

```
class Sokszog : public Alakzat {
private:
    int ncsucs;
    bme::Point* p;
public:
    Sokszog(paraméterek) {
        ...
        p = new bme::Point[ncsucs];
    }
    ...
    ~Sokszog() {
        delete[] p;
    }
}
```

Öröklés és memóriakezelés

- Megjegyzendő, hogy a dinamikus memóriakezelés miatt az = operátort és a másoló konstruktort is felül kell definiálni a `Sokszog` osztályban.
- Tegyük fel, hogy különböző típusú alakzatokat egy tömbben tárolunk:

```
Alakzat* tomb[10];
tomb[1] = new Sokszog(paraméterek);
...
delete tomb[1];
```

- Gond van: a `delete` utasításnál az `Alakzat` osztály destruktora hívódik meg, ami nem tud a `Sokszog` osztályban foglalt memóriáról.

Öröklés és memóriakezelés

- Megoldás: virtuális destruktor.
- Amennyiben van virtuális függvénye egy osztálynak, akkor a fenti jellegű problémák kezeléséhez általában is érdemes virtuális destruktort írni.
- Megjegyzés: (másoló) konstruktor nem lehet virtuális, és bár az értékadó = operátor lehet virtuális, de az ilyen megoldások kerülendők (részletek később).

Kivételkezelés

- Egy program futása során váratlan („kivételes”) események fordulhatnak elő: pl. nem invertálható elemet próbálunk invertálni, rosszul indexelünk egy tömböt stb.
- Az első esetben a hibát ez idáig egy invalid objektummal való visszatéréssel kezeltük.
- Tegyük fel, hogy egy `[]` operátor egy tároló elemére mutató `int&` referenciát ad vissza.
- Milyen értéket adjunk vissza rossz index esetén?
- Nem lefoglalt memóriacímet visszaadni semmi esetre sem tanácsos, bármilyen más címmel pedig a függvény hívóját csapjuk be.

Kivételkezelés

- A C++ a következő megoldást kínálja erre: bármely függvény számára megengedett, hogy a szokásos visszatérési értéke helyett – ha valamilyen hibát szeretne jelezni – egy kivételobjektummal „térjen vissza”.
- A kivételobjektumokat a hiba észlelésének helyén a `throw` kulcsszóval kell eldobni.
- Ennek hatására a kivételt dobó függvény futása megszakad, a kivételobjektum pedig a hívási sorban a dobáshoz legközelebbi olyan `catch(...){}` utasításblokkhoz kerül, amely az adott típusú kivételt el tudja kapni, és ez a blokk lefut.
- Ehhez a kivételt dobó függvényt egy `try{}` blokkba kell írni.
- A `catch`-nek meg kell adni egyetlen paramétert, ilyen típusú objektumot tud elkapni.

Kivételkezelés

- Mindez egy példán:

```
int& IntTomb::operator[] (size_t idx) {
    if (idx >= meret)
        throw "hiba: tulindexeles";
    return intomb[idx];
}

int main() {
    IntTomb v(...);
    try {
        v[12] = 93;
    } catch (const char* hibaszoveg) {
        std::cerr << hibaszoveg << std::endl;
    }
}
```

Kivételkezelés

- Az előző kódban rossz indexelés esetén az értékadás nem hajtódik végre, hanem a `catch` elkapja a kivételt, és ott folytatódik a program futása.
- Előny: egy komplikált műveletsornál nem kell minden hibát bonyolult kóddal kezelni, elég az egészet egy `try` blokkba tenni, így biztosak lehetünk benne, hogy hiba esetén nem fognak a további műveletek hibás eredményeken végrehajtódni.
- Hátrány: előfordulhat olyan futás, amire a kód alapján nem számítunk.
- ```
  void f(IntTomb& tomb, int i) {
      tomb[12] = i;
      std::cout << "Siker";
  }
  ```
- Biztosan kiírja a függvény, hogy "Siker"? Mi dobhat itt kivételt?
  ```

Kivételkezelés

- Veszély: a kód futásának megszakadása azt eredményezheti, hogy dinamikusan foglalt memória nem szabadul fel!
- Ezt a következőképp orvosolhatjuk:

```
int* t = new int[meret];
try {
    tomb[12] = v;
} catch (...) {
    delete[] t;
    throw;
}
std::cout << "Siker";
delete[] t;
```

Kivételkezelés

- A `catch(...) { ... }` blokk minden kivételt elkap, itt az első `...` írandó a zárójelbe, míg a második `...` a kód helyét jelenti.
- Itt a `catch` blokkban lévő `throw` utasítással tovább dobjuk a kivételt a hívási sorban a következő `catch`-nek.

Kivételkezelés

- Egy másik (jobb) módja a dinamikusan foglalt memória felszabadításának, ha azt egy objektumba becsomagoljuk, a lokális objektumoknak ugyanis még a kivételdobásnál is lefut a destruktora, ami felszabadítja a memóriát.

```
void f(IntTomb& tomb, int i) {
    std::vector<int> t(tomb.meret());
    tomb[12] = v;
    std::cout << "Siker";
}
```

- Ezen elv neve RAII, Resource Acquisition Is Initialization.
- A név arra utal, hogy az erőforrás foglalása az objektum létrejöttéhez kötött, de a hangsúly igazából a destruktoron és a felszabadításon van.

Kivételkezelés

- Ahogy korábban láttuk, `catch` által elkapott kivételt a `throw` utasítással tovább dobhatjuk egy következő `catch`-nek.
- Egy `try` blokk után több `catch` blokk is lehet, amelyek különböző típusú kivételek elkapásáért felelnek.
- A `catch(...)` blokk minden kivételt elkap. Ez sokszor nem túl jó gyakorlat, hiszen így nem áll rendelkezésre a kivételobjektum.
- Figyelni kell a sorrendre, mert egy `try` blokk után legfeljebb egy `catch` blokkba lépünk be. Így pl. a `catch(...)` blokkot (ha van ilyen) utolsónak kell írni.
- Ha egy kivételt nem kapnak el, akkor a program futása leáll.

Kivételkezelés

- A C++ standard library számos kivételt kínál, melyek az `stdexcept` fejállományban találhatók.
- Pl. a `vector` osztály `at()` tagfüggvénye `std::out_of_range` kivételt dob, ha a megadott pozíció érvénytelen.
- Az `std::invalid_argument` kivételt dobhatjuk, ha pl. nem invertálható elemet próbálunk meg invertálni.

Kivételkezelés

- A kivételek közt öröklési hierarchia van, pl. mindkét fenti kivétel a `std::logic_error` kivétel leszármazottja, amely pedig (mint minden `std` kivétel) az `std::exception` leszármazottja.
- Így a `catch(const std::logic_error& exception){...}` blokk elkapja az `out_of_range` és `std::invalid_argument` kivételeket is.
- A fenti kivételek konstruktorában a dobásnál hibaüzenetként megadhatunk egy C string-et vagy `string`-et, amit aztán a `what()` tagfüggvénnyel kérdezhetünk le (ami egyébként az `exception` ősosztály tagfüggvénye).
- További kivételekért és részletekért lásd a dokumentációt.

Kivételkezelés

- A fenti kódban lényeges (és általában jó gyakorlatnak számít), hogy referenciaként kapjuk el a kivételt, érték szerinti átadásnál a leszármazottak az őssé konvertálódnak és másolódnak!
- Másrészt a kivételekről munkamásolat készülhet, pl. ha azt változtatni szeretnénk. Ekkor (ha a kivételt tovább dobjuk) az eredetit nem érdemes változtatni, hiszen onnantól nem azt a hibát írja le, mint eredetileg.
- A másolhatóság biztosítása érdekében csak publikus másolókonstruktorral rendelkező objektum dobható el kivételként.

Kivételkezelés

- A kivételkezelés tipikusan olyan szituáció, amikor egy kivétel több ősből származtatható, hiszen különböző szempontból vizsgálva a kódot egy hiba többféle kategóriába is beleeshet (pl. egy hálózatifájl-hiba tartozhat a hálózati kivételek ill. a fájlrendszeri kivételek közé is).
- De figyelem: egy `catch(T)` blokk (ahol a `T` helyén állhat `T&` vagy `T*`) elkapja a T típusú kivételt, ill. annak minden *egyértelmű* leszármazottját (tehát pl. a korábbi példából egy `GrandChild`-ot nem lehet `catch(Parent exception)` blokkal elkapni, épp a már tárgyalt anomáliák miatt).

Kivételek specifikációja

- Ha egy függvény kivételt dob, lehetőség van ezt a kivételek lehetséges típusaival együtt jelezni a deklarációnál/definíciónál:

```
void f(int i) throw(int,double){
    if (i > 0) { throw 1; }
    else { throw 0.1; }
}
```

- Kivétel-specifikációk nélküli függvényekről azt kell feltételeznünk, hogy bármilyen kivételt kiválthatnak.
- A C++17 szabvány előtt a `throw()` jelezte, hogy a függvény nem dob kivételt.
- A C++17 óta ez egyenértékű a ez a C++11 szabvány óta létező `noexcept` vagy `noexcept(true)` kifejezéssel. Így deklarálva egy függvényt, ha abban kivételdobás történik, akkor a program futása leáll.

Kivételek specifikációja

- Kivételkezelésnél (a `catch` blokkban) csak olyan függvényt érdemes hívni, ami nem dobhat kivételt, azaz `noexcept`-ként van deklarálva. Egy kivételdobás ugyanis a program leállását eredményezhetné ilyen esetben.
- Ilyen például az `std::exception` objektum `what()` tagfüggvénye is.
- Ha egy virtuális függvényt `noexcept`-ként deklarálunk egy osztályban, akkor az összes leszármazottban így kell deklarálni.
- Ezért pl. az `std::exception` objektum `what()` tagfüggvényét a leszármazottakban `noexcept`-ként kell deklarálni.

Kivételek specifikációja

- A `noexcept(függvény)` segítségével lekérdezhető, hogy az adott függvény dobhat-e kivételt.
- Az alábbi `main` függvény a 0 ill. 1 értékeket írja ki a képernyőre.

```
void f() {...}

void g() noexcept {...}

int main() {
 cout << noexcept(f()) << endl;
 cout << noexcept(g()) << endl;
 return 0;
}
```

Kivételek specifikációja

- Ennek segítségével futási időben eldönthető, hogy egy függvény dob-e kivételt, ill. template esetén a template paramétertől függővé tehető ez:

```
template<typename T>
void f() noexcept(T()) {...}
```

- A fenti függvény attól függően nem dobhat vagy dobhat kivételt, hogy a T típus default konstruktora `noexcept`-ként van deklarálva vagy sem.