Többszörös öröklés és annak nyűgjei. Virtuális öröklés. Kasztolások (különös tekintettel a dinamikus és statikus kasztolásra).

C++ programozás – 6. óra

Széchenyi István Egyetem
©Csapó Ádám

`http://dropbox.com/...`

2017

Mai étlap

- Osztályhierarchiák
  - Többszörös öröklés
- Futásidejű identifikáció: hogyan navigálhatunk explicit módon osztályhierarchiákban?
  - *static_cast*
  - *dynamic_cast*

Többszörös öröklés – Névütközések

- Ahogy korábban mondtuk, az öröklés kétféle célt szolgálhat:
  - **Megosztott interfészek**: futásidejű polimorfizmus ("*A származtatott típus egyfajta szülő*")
  - **Megosztott implementáció**: implementáció öröklése ("*A származtatott típus implementációjában felhasználhatja a szülő implementációját*")
- "Tiszta" interfészt tipikusan absztrakt osztállyal valósítunk meg.
- Igen gyakori, hogy egy osztály egyszerre több interfészt valósít meg (több osztályból származik).
- Annak sincs akadálya, hogy egy osztály többféle másik osztály interfészeként szolgáljon.
  - Pl. ha van egy *Satellite* osztályunk, az jelölhet bármit, ami valami körül kering (műholdat, követ, üstököst, . . .)

Többszörös öröklés – Névütközések II

- Mindegyik származtatott típus (más szóval: *alosztály*) felüldefiniálhat tagokat meg virtuális fv-eket, miközben egy részüket megtarthatja (pl. orbitális pálya, alakzat, stb.)
- Tfh grafikusan is ábrázolni szeretnénk a szatelliteket. Ehhez létrehozunk egy alaposztályt, ami a grafikai jellemzőket reprezentálja, és abból is örököltetjük az ábrázolni kívánt szatelliteket. Pl itt egy kommunikációra is képes szatellit:

```
class CommSat : public Satellite, public Displayed
{
public:
  //...
};

void f(CommSat& s){
  s.draw(); //Displayed::draw()
  Pos p = s.center(); //Satellite::center()
  s.transmit(); //CommSat::transmit()
}
```

Többszörös öröklés – Névütközések III

- Mivel az öröklés publikus, az interfészt származtattuk tovább.
  - Ezért *CommSat* típusú objektumot (pontosabban arra való hivatkozást) átadhatunk minden olyan függvénynek, ami névelegesen *Satellite* vagy *Displayed* típusú objektumot vár:

```
void highlight(Displayed*);
Pos centerOfGravity(const Satellite*);

void g(CommSat* p)
{
  highlight(p);
  Pos x = centerOfGravity(p);
}
```

- De mi történik névütközés esetén? Simán lehetne, hogy két különböző szülőosztályban egyazon nevű és típusú függvény szerepel!

Többszörös öröklés – Névütközések IV

- A válasz: inkább minden örökölt osztályban definiáljuk felül a szülőosztályok közös nevű és típusú függvényeit!

```
class Satellite
{
public:
  virtual DebugInfo getDebug();
  //...
};

class Displayed
{
public:
  virtual DebugInfo getDebug();
  //...
};
```

```
class CommSat : public Satellite, public Displayed
{
public:
  DebugInfo getDebug()
  {
    DebugInfo d1 = Satellite::getDebug();
    DebugInfo d2 = Displayed::getDebug();
    return mergeInfo(d1, d2);
  }
  //...
};
```

- Ezzel azonban még mindig nem oldottunk meg minden problémát!

Többszörös öröklés – Többszörösen előforduló osztály

- Nézzük meg, mi történik ha egy osztály többször is szerepel egy hierarchiában!
- Tfh fájlokban szeretnénk tárolni objektumok állapot-információit (perzisztencia, breakpointolás, stb.)
- Ezért létrehozunk egy *Storable* nevű absztrakt osztályt:

```
class Storable
{
public:
  virtual string getFile() = 0;
  virtual void read() = 0;
  virtual void write() = 0;
  virtual ~Storable(){}
};
```

- Ez nagyon hasznos, sok osztály örökölhet belőle.

Többszörös öröklés – Többszörösen előforduló osztály II

- De amikor ún. *gyémánt-alakzat* (diamond shape) jön létre az öröklési struktúrában, akkor jön a bibi!

```
class Transmitter : public Storable{
public:
  void write() override;
  //...
};

class Receiver : public Storable{
public:
  void write() override;
  //...
};
```

```
class Radio : public Transmitter, public Receiver
{
public:
  string getFile() override;
  void read() override;
  void write() override;
};
```

- Miért baj ez? Mit is mondtunk a memóriabeli tárolásról?
  - *Transmitter* tartalmazza *Storable* struktúráját is
  - *Receiver* tartalmazza *Storable* struktúráját is
  - *Radio* tartalmazza *Transmitter* és *Receiver* struktúráját is, vagyis tranzitívan **nem egy, hanem két** *Storable*-t is tartalmaz!

Többszörös öröklés – Többszörösen előforduló osztály III

- Egy virtuális fv-t viszont csak egyszer kell felüldefiniálni (ilyenkor explicite megmondjuk, mikor melyik szülő fv-ét hívjuk meg):

```
void Radio::write()
{
  Transmitter::write();
  Receiver::write();
  // ... irjunk ki mindent, ami Radio-specifikus
}
```

- De mit tehetünk, ha mégis fontos számunkra, hogy ne replikáljunk? Jobb lenne, ha a *Receiver* és *Transmitter* objektumok ugyanazon a *Storable* objektumon osztoznának!
  - Ugyanis: absztrakt osztállyal könnyű volt, de mi történik ha a *Storable* osztály tárol is valami információt, mint pl. a fájl nevét amibe írunk? Ilyenkor nincs értelme replikálni!

Többszörös öröklés – Többszörösen előforduló osztály IV

- Például ez egy agyrém:

```
class Storable{
public:
  Storable(const string& s);
  virtual void read() = 0;
  //...
private:
  string fileName;
  Storable(const Storable&) = delete;
  Storable& operator=(const Storable&) = delete;
};
```

- Ilyen esetben nem lenne helyes, ha a *Storable* objektumot replikálnánk, hiszen akkor a *Radio* típusú objektum különböző részei különböző fájlokban kerülnének eltárolásra (vagy személyesen kéne garantálnunk, hogy ne így legyen).

Többszörös öröklés – Többszörösen előforduló osztály V

- **A megoldás**: származtassunk *virtual* kulcsszóval!
- Jelentése: amikor ebből az osztályból tovább származtatunk, a szülőjéből csak egy közös példány lehet a memóriában

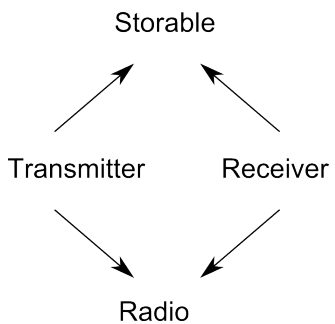
```
class Transmitter : public virtual Storable{
public:
  void write() override;
  //...
};

class Receiver : public virtual Storable{
public:
  void write() override;
  //...
};
```

```
class Radio : public Transmitter, public Receiver
{
public:
  string getFile() override;
  void read() override;
  void write() override;
};
```

- Ez igen hasznos: gondoljunk bele, mi egyebet csinálhatnánk ha több osztály között kell adatot megosztani:
  - Közös szülőosztály: nem jó, eleve ezt csináltuk!
  - Pointer ugyanarra az objektumra mindkét osztályból: rémálom!

Többszörös öröklés – Többszörösen előforduló osztály VI

- Legjobb tehát a virtuális szülőosztály használata.
- Fontos tudni, hogy ennek csak a *Radio* osztály példányosításakor van szerepe.
  - Ha csak egy *Transmitter*-t vagy *Receiver*-t példányosítunk, ugyanúgy mehet minden ahogy eddig
  - De ha egy *Radio* objektumot hozunk létre, akkor kell tudnia a futtatási környezetnek, hogy mekkora mem.területet foglaljon le, hány darab *Storable* legyen az adatszerkezetben...



Hasznos tanácsok Stroustruptól

- Ha szülőosztályt interfészként definiálunk:
  - Használjunk absztrakt osztályt, és
  - kerüljük az adattagok használatát
- Implementáció megosztását viszont adattagokat is tartalmaz(hat)ó szülőosztályokkal valósítsunk meg, protected örökléssel.
- Implementáció és interfész szétválasztásához használjunk többszörös public-protected öröklést.
- Tulajdonságok uniójának kifejezéséhez használjunk sima public többszörös öröklést.
- Ha olyan dolgot reprezentálunk, ami a hierarchia néhány objektumára közös (de nem az összesre), használjunk virtuális öröklést!

Futásidejű típusinformáció

- Mint láttuk, igen gyakori az osztály-hierarchia. A C++ előnyös tulajdonsága, hogy $f()$ fv meghívásakor nem lényeges, hogy a függvényt ugyanaz az osztály definiálja, mint amelyik deklarálta
- De hogyan szerezhetünk infót egy objektum teljes összetételéről, ha csak valamely szülőosztály interfészének vagyunk birtokában?
  - Például tfh van egy GUI-nk amiben mindenféle widget létezhet integer érték megadására (csúszka, tekerentyű, pöcök, ...). Az interfész egy *IvalBox* nevű osztályban van, amely widget is, tehát származik egy *Widget* osztályból
  - Ezek után tfh egy közelebbről nem meghatározott *IvalBox* objektumot:
    - Atadunk az *ablakozó rendszernek* ("AR", amelyik a képernyőt kezeli).
    - Amikor történik valamilyen esemény (pl. felhasználó megnyomja a *Submit* gombot, vagy állít egy csúszkán, stb.), *AR* visszaküldi az objektumot az alkalmazás felé, pl. *myEventHandler*(*Widget* $*$ *pw*) fv meghívásával. Mit kezdjünk vele az alkalmazásban?

Futásidejű típusinformáció II – dynamic cast

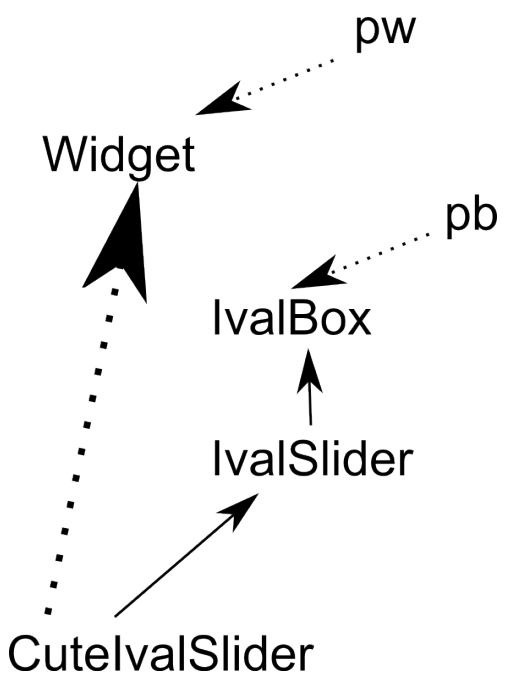
- A gond csak annyi, hogy az alkalmazás nem feltétlenül fogja tudni, hogy pontosan milyen típusú objektumot kapott vissza.
  - Pointert viszont csak akkor tudunk felhasználni, ha tudjuk, hogy milyen típusú, méretű objektumra mutat.
- Ezért jó a *dynamic_cast*() fv, ami pointer esetén nullptr-t ad vissza, ha nem a megfelelő típusú objektumra próbálunk kasztolni:

```
void myEventHandler(Widget* pw)
{
  if(IvalBox* pb = dynamic_cast<IvalBox*>(pw)) //dynamic_cast template-elt fv
  {
    //...
    int x = pb->getValue(); //hasznaljuk az IvalBox-ot
    //...
  }
  else //ha a fenti eredmeny nullptr, ez az ag fut le
  {
    //hupsz! varatlan esemeny
  }
}
```

Futásidejű típusinformáció III – dynamic cast II

- Itt fontos megjegyezni, hogy csak az interfészről van szó: teljesen mindegy, hogy milyen típusú *IvalBox*-ot használunk (slidert, dialt, vagy valami mást). A fv szerepe annyi, hogy *Widget* és *IvalBox* között konvertáljon.

- A típus-információ ilyen felhasználását **run-time type information-nek**, vagy RTTI-nek szokták nevezni.
- Kasztolásban pedig megkülönböztetjük a *downcast, upcast, és crosscast* (mint esetünkben) kasztolást.



Futásidejű típusinformáció IV – dynamic cast III

- A *dynamic_cast* hívás kacsacsőrök között egy típust vár, paraméterként pedig pointert vagy referenciát.
- Ha *dynamic_cast* $< Sz* >$ (p) hívásban p Sz típusú pointer, vagy G típusú pointer (ahol Sz szülőosztálya G gyermeknek), akkor p pointer memóriaterülete kezelhető úgy, mint egy Sz típusú objektum. Például:

```
class CuteIvalSlider : public IvalSlider, protected CuteSlider
{
  //...
};

void f(CuteIvalSlider* p)
{
  IvalSlider* pi1 = p; //OK
  IvalSlider* pi2 = dynamic_cast<IvalSlider*>(p); //OK, CuteIvalSlider kezelheto IvalSlider-kent

  CuteSlider* pbb1 = p; //hiba: CuteSlider protected szuloosztaly!
  CuteSlider* pbb2 = dynamic_cast<CuteSlider*>(p); //OK, p erteke nullptr
}
```

Futásidejű típusinformáció V – dynamic cast IV

- Ha a konverzió sikertelen, *dynamic_cast*:
  - pointer esetén *nullptr*-t ad vissza
  - referencia esetén kivételt dob (méghozzá *bad_cast* típusút)!
- Az ugyanis lehetséges, hogy pointer ne mutasson semmire.
- Referenciának viszont mindig hivatkoznia kell valamire!
- Nézzük most meg a dinamikus kasztolás működését kicsit "formálisabban"!

Futásidejű típusinformáció VI – dynamic cast V

- A helyzet a következő: képzeljük el, hogy van egy *Auto* meg egy *Ferrari* osztályunk. Minden *Ferrari* egyben *Auto* is, de fordítva nem igaz!
  - Ha *dynamic_cast*-tal egy *Ferrari*& referenciát akarok *Auto*& referenciára konvertálni, minden OK. Ugyanis: minden *Ferrari* objektum tartalmaz egy *Auto*-t is a saját memóriaterületén. Csak le kell belőle vágni!
  - Fordítva viszont nem biztonságos a dolog!
- **Amikor tehát meghívjuk *dynamic_cast*-ot, egy kérdést teszünk fel a futtatási környezetnek**:
  - *Garantálod, hogy ez az objektum biztonságosan konvertálható a másik típusba?*
- Persze OK lehet a lefele kasztolás is (downcast, pl. *Auto*&-ból *Ferrari*&), **de csak akkor, ha az az autó amire meghívom tényleg Ferrari!**

Futásidejű típusinformáció VII – dynamic cast VI

- Összességében a *dynamic_cast* **akkor nem működik**, ha:
  - A két osztály nem polimorfikus (tehát nem is szerepelhetnének más helyett, nincs bennük virtuális fv.)
  - Az adott példány nem konvertálható biztonságosan a másikba (pl. mert nem kompatibilisek, vagy nincs egyedi konverzió!)
    - Gondoljunk pl. arra az esetre, ha gyémánt-alakzat van virtuális öröklés nélkül!
    - Ha *Radio* referenciát konvertálnánk *Storable* referenciává, a futtatási környezet nem igazán tudná megmondani, hogy melyik *Storable* objektumot válassza, mivel kettő is található lenne a *Radio* objektumon belül.

További kasztolások – static cast

- Milyen más megoldások vannak kasztolásra? Nézzük pl. a *static_cast*-ot! **Ezt a kaszt-típus fordításidőben végzi el a konverziót**.
- Két objektum között akkor tud konverziót elvégezni, ha a kettő között létezik definiált konverzió (jelesül egy konstruktor, ami az egyik alapján a másikat elő tudja állítani)
- Amennyiben nem sima objektumról, hanem pointerről vagy referenciáról van szó, abban az esetben olyankor is működik a kaszt, ha az egyik osztály a másikból származik
  - Ráadásul upcast és downcast esetén is működik... vagyis abban az esetben is, ha a konverzió nem biztonságos!
  - Ezért mondhatjuk, hogy a *static_cast* mindent elhisz (a programozó már fordításidőben megmondja, hogy ez így lesz, és kész)

További kasztolások – static cast II

```
class Dog{
public:
  Dog(std::string s){name = s;}
  virtual void print(){std::cout << name <<
        std::endl;}
  std::string getName(){return name;}
private:
  std::string name;
};

class YellowDog : public Dog{
public:
  YellowDog(std::string s) : Dog(s){};
  YellowDog(Dog d); //statikus konverzio OK!
  void print() override{std::cout << "yellow: "
        << getName() << std::endl;}
};

int testFunction(Dog* d, Dog& rd){
    YellowDog* yd1 = dynamic_cast<YellowDog*>(d);
    if(yd1){
        cout << "Dcasted pointer" << endl;
    }else{
        cout << "Dcast gave nullptr " << endl;
    }
```

```
    //testFunction folyt.
    bool sentinel = false;
    try{
        YellowDog& yd2 =
              dynamic_cast<YellowDog&>(rd);
    }catch(std::bad_cast e){
        cout << "could not Dcast reference " <<
              endl;
        sentinel = true;
    }
    if(!sentinel){
        cout << "Dcasted reference " << endl;
    }

    //mindig mukodik
    YellowDog* yd3 = static_cast<YellowDog*>(d);
    YellowDog& yd4 = static_cast<YellowDog&>(rd);
    cout << "Scasted pointer and reference" <<
          endl;

    //csak akkor mukodik, ha konverzio definialt
    //ellenkezo esetben le se fordul!
    YellowDog yd4 = static_cast<YellowDog>(*d);
    std::cout << "Scasted Dog to YDog" << endl;
}
```

További kasztolások – static cast III

```
int main(){
    Dog d1("kutya1");
    Dog d2("kutya2");
    YellowDog d3("sargakutya1");
    YellowDog d4("sargakutya2");
    testFunction(&d1, d2);
    testFunction(&d3, d4);
}
```

- Tehát:

```
Dog d1 = static_cast<Dog>(string("Bob")); //mukodik, ha Dog-nak van string-gel mukodo konstruktora!
YellowDog d2 = static_cast<YellowDog>(d1); //mukodik, ha YellowDog-nak van megfelelo konstruktora
Dog d3 = static_cast<Dog>(d2); //mindig mukodik, csak el kell hagyni valamennyit d2 mem.teruletebol
Dog* d2 = static_cast<Dog*>(new YellowDog()); //pointerek eseten mukodik polimorfikus tipusokkal
      is, up es downcast eseten is, de csak kapcsolatban levo osztalyok eseten!
YellowDog* d3 = static_cast<YellowDog*>(new Dog()); //ez a downcast veszelyes, de a static cast
      megoldja... nem vesz hozza mem.teruletet sem a Dog objektumhoz, csak ugy csinal mintha ott
      lenne

//ez utobbi peldara vonatkozik, hogy a static_cast mindent elhisz. Nem nezi, hogy biztonsagos-e!
```

További kasztolások – static cast IV

- Mivel *static_cast* fordításidejű, ezért virtuális örökléssel felépített polimorfizmus esetén downcastra nem működik (ellentétben a *dynamic_cast*-tal).
  - A *static_cast* ugyanis nem tudja megvizsgálni a forrás-objektum (amit konvertál) belső struktúráját, mivel fordításkor kell eldőlnie, hogy miről mire konvertál!
  - Például: nem néz bele a *Storable* pointer által mutatott memóriaterületre, hogy kimazolázza hogy az valójában *Receiver*, *Transmitter* vagy *Radio*. Virtuális öröklésnél ezért nem használható downcast-ra.
  - Az upcast viszont könnyű: a konvertáláshoz annyi szükséges, hogy a *Receiver* mutató által mutatott mem.területhez hozzávegyünk annyit, hogy a nagyobb *Radio* típusú objektumra is mutathasson.

További kasztolások – static cast V

- Tehát:

```
void g(Radio& r)
{
  Receiver* prec = &r; //receiver sima szuloosztaly tehat rendben
  Radio* pr = static_cast<Radio*>(prec); //OK, unchecked: csak nem biztos, hogy az ezt koveto
        mem.terulet jol formazott...
  pr = dynamic_cast<Radio*>(prec); //OK, run-time checked

  Storable *ps = &r; //storable virtualis szuloje r-nek...
  pr = static_cast<Radio*>(ps); //hiba: virtualis osbol nem lehet statikusan kasztolni
  pr = dynamic_cast<Radio*>(ps); //futasidoben ok, mert ps tenyleg radio...
}
```

- Végül, van olyan eset is, amikor csak *static_cast* használható. Pl. *void∗* esetén bármi lehet a mem.területen, így *dynamic_cast* nem is tudna belenézni!