Dinamikus polimorfizmus

OOP és DB - OOP 5. óra

Osztályhierarchiák

Származtatott osztály maga is lehet további osztályok szülője

Az így létrejött osztályhierarchia általában fa struktúrájú, de előfordulhatnak más gráf-típusok is. Az öröklések láncolata felírható irányított, aciklikus gráfként

Például:

```
class Employee {/*...*/};
class Manager : public Empoloyee {/*...*/};
class Director : public Manager {/*...*/};
class Temporary {/*...*/};
class Assistant : public Employee {/*...*/};
class Temp : public Temporary, public Assistant {/*...*/};
class Consultant : public Temporary, public Manager {/*...*/};
```

A mai óra két fő kérdése

Ha egy függvény / metódus kap egy *Base** típusú változót, hogyan tudja eldönteni, hogy azon a címen valójában milyen konkrét osztály példánya található? És ha meghívja a pointer által hivatkozott objektum egy metódusát, hogyan lehet elérni, hogy a megfelelő konkrét osztály metódusa hívódjon meg?

Erre a kérdésre 4-féle választ adhatunk:

1. Elkerüljük ennek a lehetőségét. Minden függvény argumentuma csak egyféle típusra mutató pointer lehet (ez OOP szempontból nem praktikus)
2. Hozzunk létre egy típusmezőt a szülőosztályban, amit minden gyermek írhat / és megvizsgálhatunk (ronda és könnyen hibához vezet)

A mai óra fő kérdése

Ha egy függvény / metódus kap egy *Base** típusú változót, hogyan tudja eldönteni, hogy azon a címen valójában milyen konkrét osztály példánya található? És ha meghívja a pointer által hivatkozott objektum egy metódusát, hogyan lehet elérni, hogy a megfelelő konkrét osztály metódusa hívódjon meg?

Erre a kérdésre 4-féle választ adhatunk:

3. Használjunk **virtuális függvényeket**

4. Használjunk **dynamic castot**

Az 1. megoldás ál-megoldás

Egy lehetséges megoldásként említettük: “*Minden függvény argumentuma csak egyféle típusra mutató pointer lehet*”

Az objektum-orientált programozás pont arról szól, hogy a világban levő dolgokat és a köztük levő hierarchikus viszonyokat hatékonyan ábrázoljuk / kihasználjuk. Ha van egy *hangfájl* és egy *képfájl*, mindkettő egy fájl, ezért az alapvető fájlműveletek mindegyikre alkalmazhatóak kell, hogy legyenek. Ha van egy *open(File*)* függvény, annak el kell tudnia döntenie, hogy milyen fájl-típusról van szó (vagy a megfelelő osztály *open()* metódusát meg kell tudni hívnia).

-> az nem hatékony megoldás, hogy 5-féle open() függvényt valósítunk meg - mindegyiket más fajta fájlra specializálva.

2. megoldás: típusmezők

Kerüljük a típusmezőket! Ugyanis mi a probléma ezzel?

```
struct Employee{
  enum EmplType {man, empl};
  EmplType type;
  string first_name, last_name, middle_initial;
  short department;

  Employee() : type{empl} {}
  //...
};
```

```
struct Manager : public Employee{
  list<Employee*> group; //kiket menedzsel
  short level;

  Manager() {
    type = man;
  }
  //...
};
```

2. megoldás: típusmezők

Mostantól a fordító ellenőrzése nélkül lavírozunk a típusok között!

```
void print_employee(const Employee* e)
{
  switch(e->type)
  {
    case Employee::empl:
      std::cout << e->family_name << '\t' << e->department << std::endl;
      break;
    case Employee::man:
      std::cout << e->family_name << '\t' << e->department << std::endl;
      //...
      const Manager* p = static_cast<const Manager*>(e);
      std::cout << "level " << p->level << std::endl;
      break;
  }
}
```

2. megoldás: típusmezők

Ráadásul ha új típusú alkalmazottat hozunk létre, minden ilyen helyen frissíteni / bővíteni kell a kódot + bővíteni kell az Employee osztályban az EmplType enumot is :-O

Ez ellentmond az ún. **“open-closed” elv**nek: az osztályok nyitottak a kiterjesztésre (öröklésre), és zártak a módosításra.

> Régen rossz, ha egy A osztályt módosítani kell csak azért, mert valahol máshol (egy távoli leszármazottként) létrehoztunk egy B osztályt.

3. megoldás: virtuális függvények

A OOP-ben egy alapvető technika, hogy a szülőosztály metódusait a gyermek osztály felülírhatja, és a gyermek osztály azonos nevű metódusát a szülő osztály interfészén keresztül érhetjük el. C++-ban ezt a **virtual** kulcsszó segítségével tehetjük meg.

Ha egy metódust *virtual*-ként jelölünk meg, az azt jelenti, hogy "*ezt a metódust a további leszármazottak felüldefiniálhatják*"

Nem kötelező nekik feltétlenül, de lehetőségük van rá.

```
class Employee
{
public:
  Employee(const string& name, int dept);
  virtual void print() const;
  //...
private:
  string first_name, family name;
  short department;
  //...
};
```

Mindez az eredeti problémára is megoldást jelent, mert ha egy mezei függvény kap egy *Employee** pointert, és meghívja annak *print()* metódusát, a futtatási környezet gondoskodik róla, hogy a megfelelő metódus fusson le.

3. megoldás: virtuális függvények

A OOP-ben egy alapvető technika, hogy a szülőosztály metódusait a gyermek osztály felülírhatja, és a gyermek osztály azonos nevű metódusát a szülő osztály interfészén keresztül érhetjük el. C++-ban ezt a **virtual** kulcsszó segítségével tehetjük meg.

Ha egy metódust *virtual*-ként jelölünk meg, az azt jelenti, hogy "*ezt a metódust a további leszármazottak felüldefiniálhatják*"

Nem kötelező nekik feltétlenül, de lehetőségük van rá.

```
class Employee
{
public:
  Employee(const string& name, int dept);
  virtual void print() const;
  //...
private:
  string first_name, family name;
  short department;
  //...
};
```

Ugyanígy, ha egy fp nevű File* pointerre meghívjuk az fp->open() metódust, akkor ez működni fog akkor is, ha a pointer egy hangfájlra, meg akkor is, ha egy képfájlra mutat.

3. megoldás: virtuális függvények

Megj.: *virtual* kulcsszó nélkül is létrehozhatjuk ugyanazt a nevű és ugyanolyan paramétereket váró metódust a gyermek osztályban. Az lesz a különbség, hogy *Szulo** pointeren keresztül nem tudjuk elérni a *Gyermek* osztály adott metódusát - így elesünk attól a lehetőségtől, hogy egyazon függvény sokféle típusra működjön.

Természetesen *virtual* kulcsszó esetén a gyermek osztályban megvalósított (újradefiniált) metódus argumentumainak száma és típusa a szülőben levő virtual metódusával meg kell, hogy egyezzen

> Visszatérési érték tekintetében lehet eltérés abban az értelemben, hogy ha pl. *Derived* osztály *Base* osztály leszármazottja, akkor egy virtuális függvény a két osztályban adhat vissza *Derived** illetve *Base** típust (vagy *Derived&* illetve *Base&* típust)

3. megoldás: virtuális függvények

Adott szülőosztályban dönthetünk úgy is, hogy egy vagy több virtuális metódust nem valósítunk meg, azok megvalósítását a gyermekosztály(ok)ra hagyjuk

Ezt úgy tehetjük meg, hogy a deklarációban a metódust *tisztán virtuálissá* (**pure virtual**) tesszük, amit konkrétan úgy érünk el, hogy a végére azt írjuk, hogy = 0;

Az olyan osztályt, amelyik legalább egy pure virtual metódust tartalmaz, **absztrakt osztály**nak nevezzük.

Absztrakt osztály nem példányosítható.

> Ettől függetlenül tagváltozói és konstruktora (meg más metódusai is) lehetnek, hiszen ezeket öröklik / inicializáláskor meghívhatják a gyermekosztályok

3. megoldás: virtuális függvények

Példa:

```
class Employee {
    std::string first_name, last_name, middle_initial;
    short department;
public:
    Employee(std::string fn, std::string ln, std::string mi, short d) :
        first_name(fn), last_name(ln), middle_initial(mi), department(d) {}
    virtual void print() = 0;
    std::string getName() {
        std::string s(first_name);
        return s + " " + middle_initial + " " + last_name;
    }
};
```

3. megoldás: virtuális függvények

Példa, folyt.:

```
class Manager : public Employee {
    std::list<Employee*> group;
public:
    Manager(std::string fn, std::string ln, std::string mi, short d) :
        Employee(fn, ln, mi, d) {}
    void addToGroup(Employee* e) { group.push_back(e); }
    void print() {
        std::cout << "Manager called " << getName() << std::endl;
    }
    void printGroup() {
        for (Employee* ep : group) {
            ep->print(); // mindegy milyen employee
        }
    }
};
```

3. megoldás: virtuális függvények

Példa, folyt.:

```
class CleaningStaff : public Employee {
public:
    CleaningStaff() : Employee("NoName", "Jose", "R.", 10) {}
    void print() { std::cout << "Cleaning staff member X" << std::endl; }
};
```

```
int main()
{
    CleaningStaff c1;
    CleaningStaff c2;
    Manager pw("Paul", "Wolowitz", "J.", 55);
    Manager sc("Sheldon", "Cooper", "M.", 50);

    sc.addToGroup(&pw);
    sc.addToGroup(&c1);
    sc.addToGroup(&c2);

    std::cout << "Sheldon Cooper manages the following group:" << std::endl;
    sc.printGroup();
```

3. megoldás: virtuális függvények

Mindez lehetővé teszi a futásidejű (dinamikus) polimorfizmust.

Ez pointereken vagy referenciákon keresztül működik (konkrét objektumon nem, mert az már fordításidőben eldől, hogy micsoda). Vegyük észre, hogy a lenti példa akkor is működik, ha *printList()* függvényt még azelőtt lefordítottuk, hogy a *Manager* osztály egyáltalán létezett! (ha a *printList()* egy külön .cpp fájlban van, azt már lehet, hogy 2 éve lefordítottuk)

```
void printList(const list<Employee*>& s)
{
  for(auto x : s){
    x->print;
  }
}
```

```
int main()
{
  Employee e {"Brown", 1234};
  Manager m {"Smith", 1234, 2};

  printList({&e, &m});
}
```

3. megoldás: virtuális függvények

Fontos: ha egy osztálynak van legalább egy virtuális metódusa, jól tesszük, ha a destruktort is virtuálisként definiáljuk.

> Ha egy függvény kap egy pointert a szülő típusra, és meghívja rá a *delete*-et, fontos, hogy a megfelelő gyermek osztály destruktora fusson le!

Összességében: A virtuális függvények előfeltételei annak, hogy egyazon interfészen keresztül többféle objektum-típusnak ugyanazt a nevű metódusát transzparens módon el tudjuk érni. A *virtual* kulcsszó nélkül is definiálhatunk ugyanolyan nevű metódust a gyermekben, de a **fordító és a futtatási környezet nem fogja tudni**, hogy a származtatott metódus használható a szülő metódus *helyett*.

4. megoldás: dinamikus kasztolás

Ha egy osztályhierarchiában több szinten ugyanazok a nevű és paraméter-listájú metódusok fordulnak elő, akkor a virtuális metódusok OOP-szempontból remek megoldást jelentenek.

De mi történik, ha egy függvénynek attól függően más és más további függvényeket / metódusokat kell meghívnia, hogy éppen milyen konkrét típussal dolgozik?

> Mint mondtuk, a típusmező nem nyújt megoldást. Részben megoldást jelenthetne egy *typeof()* virtuális metódus, de csak látszólag: ügyelni kéne arra, hogy az idők végezetéig minden származtatott osztály egyedi azonosítót adjon vissza, és tesztelni kéne a kimenetre, ami “*error-prone*”

4. megoldás: dinamikus kasztolás

Amikor meg kell tudnunk, hogy milyen konkrét típussal van dolgunk, használjunk inkább dinamikus kasztolást!

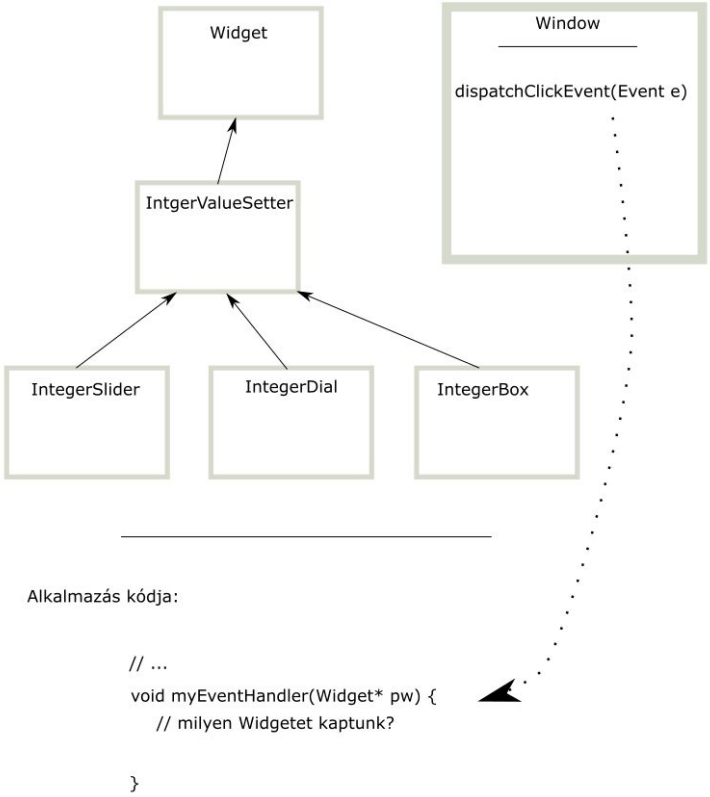
Példa: Tegyük fel, hogy egy GUI-t fejlesztünk. Egész számok megadására a GUI sokféle interfészt kínál: van benne csúszka, tekerentyű, pöcök, stb.

Ezen interfészek ősosztálya az *IntegerValueSetter*. Ez az ősosztály pedig a *Widget* ős-ősosztálytól örököl, mivel ő egyébként egy widget is.

Van egy ablakozó osztályunk is - *Window* - ami a widgeteket kirajzolja. Ha egy csúszkát (*IntegerSlider*) állítunk, a Window osztály kódja meghívja a *myEventHandler(Widget* pw)* függvényt. A fő kérdés: hogyan döntse el a *myEventHandler* függvény, hogy amit kapott, az milyen értéket állít?

4. megoldás: dinamikus kasztolás

Van egy ablakozó osztályunk is - *Window* - ami a widgeteket kirajzolja. Ha egy csúszkát (*IntegerSlider*) állítunk, a Window osztály kódja meghívja a *myEventHandler(Widget* pw)* függvényt. A fő kérdés: hogyan döntse el a *myEventHandler* függvény, hogy amit kapott, az milyen értéket állít?



4. megoldás: dinamikus kasztolás

Válasz: A C++-ben a *dynamic_cast* segítségével “kibarkochbázhatjuk”, hogy adott változó milyen típusú.

```
void myEventHandler(Widget *pw) {
    IntegerValueSetter* pivs = dynamic_cast<IntegerValueSetter*>(pw);
    if (pivs != nullptr) {
        int x = pivs->getValue();
    }
    else {
        // hupsz! .. varatlan esemeny
    }
}
```

4. megoldás: dinamikus kasztolás

Itt megjegyezzük, hogy a példában teljesen mindegy, hogy konkrétan milyen módon állítottuk az int típust (csúszkával? pöcökkel? mindegy…). Az *IntegerValueSetter* és a *Widget* is két interfész, és közöttük tudunk konvertálni.

```
void myEventHandler(Widget *pw) {
    IntegerValueSetter* pivs = dynamic_cast<IntegerValueSetter*>(pw);
    if (pivs != nullptr) {
        int x = pivs->getValue();
    }
    else {
        // hupsz! .. varatlan esemeny
    }
}
```

4. megoldás: dinamikus kasztolás

A dynamic_cast hívás kacsacsőrök között (template paraméterként) vár egy pointer vagy referencia típust, argumentumként pedig egy konkrét pointert vagy referenciát

A template típus azt mondja meg a fordítónak, hogy milyen típusra szeretnénk konvertálni az adott pointert vagy referenciát.

Ha a konverzió sikertelen, a dynamic_cast:

- pointer-argumentum esetén *nullptr*-t ad vissza
- referencia argumentum esetén *bad_cast* kivételt dob (olyan ugyanis van, hogy pointer nem mutat semmire, de egy referenciának mindig hivatkoznia kell valamire - mivel null referencia nincs, ezért jobb híján kivétel keletkezik)

Összefoglalás

1.) Ha van olyan működés, ami több osztályban azonos, akkor -- amennyiben logikus -- szedjük ki azt egy ősosztályba (előző órai anyag)
2.) Ha van olyan funkció, ami több osztályban megvan, de működhet máshogy, akkor a gyermekekben felül lehet definiálni (ehhez a szülő megfelelő metódusai legyenek virtuálisak!)
3.) Ha uniform interfészen keresztül szeretnénk objektumokat elérni, függetlenül attól, hogy az objektum egy ősosztály, vagy valamilyen gyermek osztály példánya, használjunk virtual metódusokat!
4.) Ha ezen túlmenően osztályspecifikus dolgokat is szeretnénk elérni, típusmezők és virtuális függvények helyett használjunk dynamic_cast-ot!