Öröklés és osztályhierarchiák. Jogosultságok public, private és protected minősítőkkel. Barátok, beágyazott osztályok. Típusmezők helyett virtuális függvények, dinamikus polimorfizmus.

C++ programozás – 5. óra

Széchenyi István Egyetem
©Csapó Ádám

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

2017

Osztályok és származtatás

- A C++ a Simula nyelvből vette át:
  - az osztályok és osztály-hierarchiák gondolatát
  - azt a gondolatot, hogy az osztályokat a programozó és az alkalmazás világában előforduló fogalmak modellezésére használjuk
- Ha a C++ nyelvi funkcióit ennek elérésére használjuk, az jó
- Ha viszont "hagyományos" programozást valósítunk meg ezen funkciók, mint jelölési eszközök használatával, az nem célszerű
- Semmilyen fogalom sem önmagában létezik. Például ha el kell magyaráznunk, mi az, hogy *személyautó*, kisvártatva szükségünk lesz egyre több és több fogalomra (*kerék*, *motor*, *vezető*, *gyalogos*, *kamion*, ...)

Osztályok és származtatás II

- A C++-ban a **származtatás azt a célt szolgálja, hogy bizonyos tulajdonságokat megosszunk osztályok között**.
- Pl. egy körben és háromszögben közös, hogy az *alakzat* fogalma mindkettőre vonatkozik.
  - Ez esetben a *Kor* és *Haromszog* osztályokat úgy definiáljuk, hogy az *Alakzat* osztállyal közös tulajdonságaik lehessenek
  - *Alakzat* a szülőosztály, *Kor* és *Haromszog* pedig származtatott osztályok.
  - A származtatás mechanizmusát öröklésnek is nevezzük.
- Ha viszont a *Kor* és *Haromszog* osztályokat *Alakzat* osztály nélkül hozzuk létre, valami fontosat kihagytunk!
- Gyakran objektum-orientált programozásnak nevezzük, amikor származtatás útján osztályhierarchiákkal operálunk.

Osztályok és származtatás III

- A C++ nagyjából kétféle célból támogatja a származtatást:
  - **Implementáció származtatása**: csökkenti az implementációs költségeket, ha a szülőosztály megvalósítását újrahasznosíthatjuk
  - **Interfész származtatása** (***futásidejű polimorfizmus***, másszóval ***dinamikus polimorfizmus***): különböző gyermekosztályokat együtt használhatunk, ha a közös szülőosztály interfészét használjuk
    - Pl. egy *Alakzat* pointereket tartalmazó vektor vegyesen tartalmazhat *Kor* és *Haromszog* objektumokra mutató pointereket is
- A történetnek itt nincs vége: ha két osztálynak semmilyen származtatásbeli köze nincs egymáshoz, akkor template-eken keresztül még mindig létrehozhatunk kapcsolatot közöttük.
  - Ezt sokszor ***fordításidejú polimorfizmusnak***, vagy ***statikus polimorfizmusnak*** nevezzük.

Tematika

- A következő két órán három témakör fog szóba kerülni:
  - *Alapvető nyelvi funkciók objektum-orientált programozáshoz*
    - szülő és származtatott osztályok
    - virtuális függvények
    - hozzáférési privilégiumok
    - barátok és beágyazott osztályok
  - *Osztály-hierarchiák*
    - hatékony kódszervezés, többszörös öröklés
  - *Futásidejű identifikáció*: hogyan navigálhatunk explicit módon osztály-hierarchiákban
    - *static_cast*
    - *dynamic_cast*
    - *typeid* objektum típusának meghatározására (nem ajánlott!)

Miért van szükség származtatásra? Példa

- Tfh egy cég által alkalmazott dolgozók nyilvántartására készítünk programot. Szükségünk lehet egy ilyen adattípusra:

```
struct Employee
{
  string first_name, family_name;
  char middle_initial;
  Date hiring_date;
  short department;
  //...
};
```

- Ezt követően definiálunk egy menedzser-típust is:

```
struct Manager
{
  Employee emp; //sajat alkalmazotti nyilvantartas
  list<Employee*> group; //akiket menedzsel
  short level;
  //...
};
```

Miért van szükség származtatásra? Példa II

- A menedzser alkalmazott is egyben. Az alkalmazotti szerepre vonatkozó adatokat az *emp* tagváltozóban tárolja.
- Figyelmes olvasó számára ez érthető lehet, de semmi sem mondja meg a fordítónak, vagy más elemzőprogramnak, hogy a kettő között ilyen kapcsolat áll fenn.
  - Egy *Manager*$*$ nem *Employee*$*$ is egyben, így nem lehet pl. egyiket a másik helyén használni.
  - Például ha Employee-kat (pointereket) tartalmazó konténerbe akarnánk Managereket (pointereket) rakni, akkor vagy explicit módon kellene konvertálnunk, vagy pl. az *emp* tagváltozóra mutató pointert kellene használnunk
    - mindkettő nagyon obskúrus, sok hibalehetőséggel

Miért van szükség származtatásra? Példa III

- Inkább csináljuk így:

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

- A *Manager* osztály az *Employee* osztályból származik. *Employee* a *Manager* osztály szülőosztálya.
- A *Manager* osztály tartalmazza az *Employee* osztály összes tagváltozóját és alapesetben annak összes tagfüggvényét, plusz még amit az osztály maga definiál (*group*, *level*, ...).
- A gyermek osztály (memóriaterület tekintetében) sosem lehet kisebb, mint a szülőosztály

Memóriafelhasználás öröklésekor

- A motorháztető alatt a származtatott osztály összetevődik:
  - egy érintetlenül maradt szülőosztálynak megfelelő memóriaterületből
  - a fenti területhez hozzávett további adatokból
  - Ezt követően a gyermekosztály könnyen felhasználható azokon a helyeken, ahol igazából szülőosztályt várnánk (mivel a memóriaterületre mutató pointer direktben értelmezhető szülő típusú objektumra mutató pointerként is)

```
void f(Manager m1, Employee e1)
{
  list<Employee*> elist {&m1, &e1};
  //...
}
```

Memóriafelhasználás öröklésekor II

- *Menedzser* egyfajta *Employee*, ezért *Manager*$*$ és *Manager*& használható *Employee*$*$ és *Employee*& helyett.
  - Fordítva viszont nem igaz! Hiszen nem minden *Employee*-ra igaz, hogy egyben *Manageris*, így csúnyán beletrafálhatnánk olyan memóriaterületbe, ami már egy másik objektumhoz tartozik!
- Viszont ahhoz, hogy ezt megtehessük, a szülőosztálynak is definiálva kell lennie (nemcsak delkarálva)
  - ugyanis amikor helyette használjuk a származtatott osztályt, olyan, mintha egy név nélküli objektumot hoznánk létre a szülő típusból!

```
class Employee; //csak deklaracio, itt nincs definialva

class Manager : public Employee //hiba! Nem tudjuk, mekkora helyet foglal el egy Employee
{
  //...
};
```

Osztályok függvényei és származtatás

- Általában nem elég adattípust definiálni, műveletek is kellenek hozzá. Erre valók a metódusok (tagfüggvények). Pl.

```
class Employee
{
public:
  void print() const;
  string full_name() const {return first_name + ' ' + middle_initial + ' ' + family_name; }
  //...
private:
  string first_name, family_name;
  char middle_initial;
  //...
};

class Manager : public Employee
{
public:
  void print() const;
};
```

Osztályok függvényei és származtatás II

- Származtatott osztály ugyanúgy használhatja a szülőosztályban levő public és protected kulcsszóval definiált függvényeket is, mintha azok a származtatott osztályhoz magához tartoznának. Pl.

```
void Manager::print() const
{
  std::cout << "name is " << full_name() << std::endl; // full_name() a szuloben public, tehat OK!
}
```

- *private* tagváltozók viszont nem érhetőek el:

```
void Manager::print() const
{
  std::cout << "name is " << family_name << std::endl; //hiba, nem fordul le!
}
```

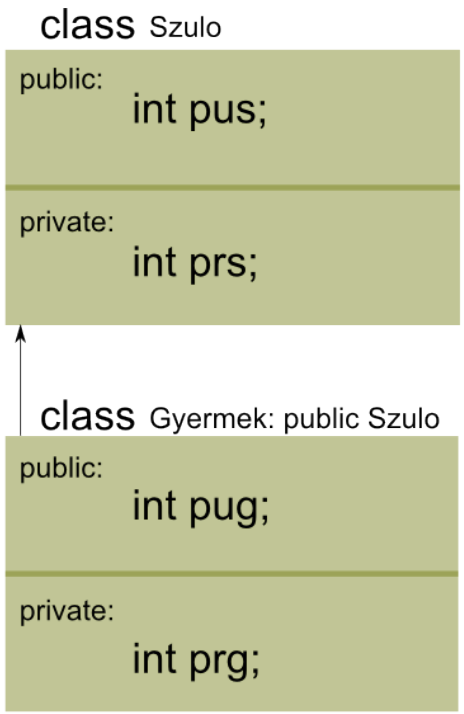
Osztályok függvényei és származtatás III

- Ellenkező esetben nem lenne túl sok értelme, hiszen akkor:
  - Bármilyen privát változó elérhető lenne, ha csak származtatnánk egy osztályból
  - Ezen kívül ha kíváncsiak lennénk, hogy mit csinál egy *private* tag, az összes forrásfájlt át kellene nézni, nemcsak az osztályt magát (és annak barát függvényeit).
    - Ez nyilvánvalóan nem praktikus, és ellentmond az adatrejtés elvének!
- Tehát az eddigiek alapján:
  - osztály privát adattagja sosem érhető el, sem az abból származtatott objektumból, sem pedig kívülről (hanem csak az adott osztályon belülről)
  - osztály publikus adattagja elérhető nemcsak az adott osztályon belülről, hanem származtatott objektumból, továbbá kívülről is
- De ez még mind semmi: C++-ban magához az öröklődéshez is társítunk jogosultságra vonatkozó módosítót!

Publikus és privát öröklés

- **Publikus (public) származtatás jelentése**: *Manager* egyfajta *Employee*. Ekkor:
  - Az *Employee* publikus tagjaihoz hozzáférhet:
    - *Manager* bármely függvénye (implementációja)
    - *Manager* bármely barát (friend) függvénye (ld. később)
    - *Manager* objektumon keresztül bármely mezei függvény
- De ehhez muszáj öröklésekor kiírni, hogy public!
- Alapesetben az öröklés privát. Ekkor *Employee* változó publikus tagjaihoz hozzáférhet:
  - *Manager* bármely függvénye (implementációja)
  - *Manager* bármely barát (friend) függvénye
  - de *NEM* férhet hozzá bármely mezei függvény
  - Pont ezekért: **Privát öröklés implementációs részletek elrejtéséhez hasznos** (ilyenkor az interfészt korlátozzuk / átalakítjuk).

Publikus és privát öröklés II.

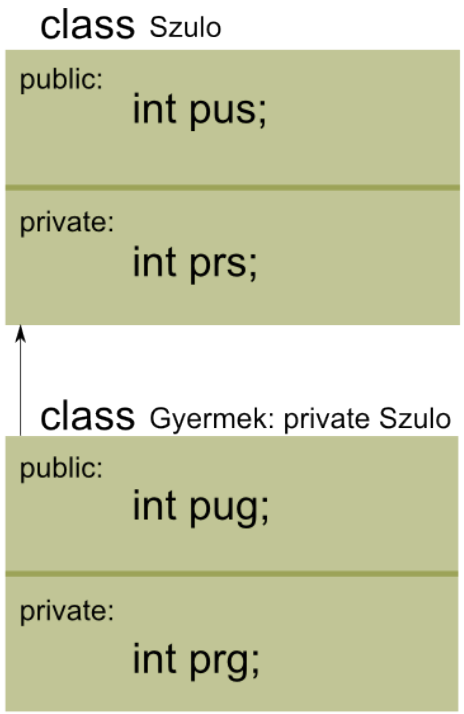



```
void Szulo::fuggveny1(Szulo& sz){
    //Szulo scope-ja: pus és prs is elérhető!
    pus = 5;
    prs = 2;
    sz.pus = 5; //nincs kulonbseg, Szulo scope-ja!
    sz.prs = 2;
}
void Gyermek::fuggveny2(Gyermek& gy){
    //Gyermek scope-ja: pus publikus, prs nem elérhető!
    pus = 5;
    prs = 2; //hiba
    pug = 15;
    prg = 6;
    gy.pus = 5; //nincs kulonbseg, Gyermek scope-ja!
    gy.prs = 5; //hiba: privat es nem Szulo scope!!
    gy.pug = 5;
    gy.prg = 5;
}
int main(){
    Gyermek g;
    g.pus = 15; g.pug = 20; //OK
    g.prs = 12; g.prg = 14; //hiba!!
    return 0;
}
```

A lényeg publikus öröklés esetén:

Szülő		Gyermek
public	→	public
private	→	nem elérhető

Publikus és privát öröklés III.




```
void Szulo::fuggveny1(Szulo& sz){
    //Szulo scope-ja: pus és prs is elérhető!
    pus = 5;
    prs = 2;
    sz.pus = 5; //nincs kulonbseg, Szulo scope-ja!
    sz.prs = 2;
}
void Gyermek::fuggveny2(Gyermek& gy){
    //Gyermek scope-ja: pus private, prs nem elérhető!
    pus = 5;
    prs = 2; //hiba
    pug = 15;
    prg = 6;
    gy.pus = 5; //nincs kulonbseg, Gyermek scope-ja!
    gy.prs = 5; //hiba: privat es nem Szulo scope!!
    gy.pug = 5;
    gy.prg = 5;
}
int main(){
    Gyermek g;
    g.pus = 15; //hiba!!
    g.pug = 20; //OK
    g.prs = 12; g.prg = 14; //hiba!!
    return 0;}
```

A lényeg privát öröklés esetén:

Szülő		Gyermek
public	→	private!!!
private	→	nem elérhető

Protected adattagok és függvények

- A C++-ban léteznek azonban *protected* tagok is. Ezek a privát és publikus "keverékei":
  - kívülről sosem érhetőek el (mint privát), de
  - elérhetőek az adott osztályból (mint publikus), illetve
  - az abból származott osztályból (mint publikus), illetve
  - a származtatott osztály barát függvényeiből (mint publikus)
- Ezek a szabályok a nevekre uniform módon vonatkoznak: teljesen mindegy, hogy egy név milyen típusú dologra utal (beépített típusra, UDT-ra, pointerre, stb.)

Protected adattagok és függvények II

- Mikor lehetnek jók a protected tagok? Időnként lehetővé akarjuk tenni, hogy az osztályt kiterjesztő osztályok felhasználhassanak bizonyos adatokat, vagy fv-eket, de azokat nem szeretnénk a felhasználó orrára kötni.
- Például: programozók számára hatékony (nem ellenőrzött) elérést tehetünk lehetővé, míg más esetben ellenőrzött elérést nyújthatunk.

```
class Buffer
{
public:
  char& operator[](int i); //ellenorzott hozzaferes
protected:
  char& access(int j); //nem ellenorzott (de gyors!) hozzaferes
};
```

Protected adattagok és függvények III

- Folytatás:

```
class CircularBuffer : public Buffer
{
public:
  void reallocate(char* p, int s); //athelyezes a memoriaban
  //...
};

void CircularBuffer::reallocate(char* p, int s)
{
  //...
  for(int i=0; i!=old_sz; ++i)
  {
    p[i] = access(i); //OK, mert szarmaztatott osztalyon belulrol a protected elerheto
  }
  //...
}

void f(Buffer& b)
{
  b[3] = 'b'; //OK: ellenorzott
  b.access(3) = 'c'; //hiba: protected fv!
}
```

Protected adattagok és függvények IV

- Függvények esetén a *protected* OK, de adattagok esetén Stroustrup papa célszerűen nem ajánlja!
- A nyelv szabályai nem tiltják ugyan, de: protected adattagok (vagyis nem függvények, hanem adattagok!) használata nem tiszta kódhoz vezethet
  - Ha valaki olvassa a kódunkat, nem fogja érteni, hogy az adott adattag miért protected!
  - Ugyanis gondolhatja: mivel protected $\rightarrow$ valaki módosítani fogja, de ki? hol?
  - Ha meg senkinek sem kell módosítania, inkább tartsuk meg privátnak
    - a protected adattag valójában felhívás a “rosszalkodásra”

Öröklés típusai: public, protected, private

- Ahogy a tagok, az öröklés módja is lehet public, protected vagy private
  - Structok esetén az öröklés alapesetben public
  - Classok esetén az öröklés alapesetben private
- *class D : public B{...}* jelentése: D egyfajta B
- *class D : private B{...}* jelentése: D hozzáférhet mindenhez B-ben (ami nem private), de csak a saját implementációján belül – kívülálló nem is definiálhat olyan fv-t amely $B*$-ot vár de $D*$-gal is meghívható, mert a kettő nem konvertálható egymásba!
  - ilyenkor egy interfész korlátozásával valamilyen konkrét megvalósítást rejtünk el (*B* egy implementációs részlet)
- *class D : protected B{...}* jelentése: szintén implementációs részletet rejtünk el, de további öröklés lehetséges. Ha *D* osztályból tovább származtatunk, ugyanúgy felhasználhatóak lesznek *B* protected tagjai.

Öröklés típusai: public, protected, private II

- Mindezt az alábbi kódrészlet jól szemlélteti (érdemes ezen megtanulni az egész öröklés témakört, mert a további öröklést is jól leírja!)

```
class A
{
public:
  int x;
protected:
  int y;
private:
  int z;
};

class B : public A
{
  //x public
  //y protected
  //z nem elerheto B-bol
};
```

```
class C : protected A
{
  //x protected
  //y protected
  //z nem elerheto C-bol
};

class D : private A
{
  //x private
  //y private
  //z nem elerheto D-bol
};
```

Öröklés típusai: public, protected, private III

- Fontos: x, y és z mindhármom származtatott osztályban ugyanúgy jelen van! Itt csak hozzáférésről van szó.
- A további öröklésnél van mindennek jelentőssége. Pl. ha először protected öröklés van, majd az új osztályból publikus öröklés, az eredetileg publikus tagok továbbra is protectedek maradnak.
- Ugyanígy, ha private öröklés után tovább származtatunk, ami kezdetekkor public volt, később ugyanúgy nem lesz elérhető (mivel az első származtatás után private lett)

Barátok

- Mezei tagfüggvény három logikailag elkülönülő dolgot határoz meg:
  - A függvény hozzáférhet az osztály privát tagváltozóihoz (akkor is ha argumentumként kap egy példányt az adott osztályból)
  - A függvény az osztály hatóköréhez (scope) tartozik
  - A függvényt kötelező az osztály objektumára meghívni (értelmezi a this pointert)
- Ha egy tagfüggvény statikus, csak az első kettő igaz
- Ha egy (külső) függvényt barátként (*friend*) deklarálunk, csak az első lesz igaz.
- Vagyis: **barát függvény olyan függvény, amelyik független az osztálytól, de mégis hozzáférhet a belsejéhez.**

Barátok II

- Mire lehet ez jó? Tfh definiálni szeretnénk egy operátort, amely mátrixot vektorral szoroz össze.
  - Természetesen a *Matrix* és *Vector* osztályok elrejtik saját implementációjukat.
  - A szorzás-operátor viszont nem lehet mindkét osztály tagja, miért rakjuk egyikbe vagy másikba?
  - Közben nem szeretnénk alacsonyszintű hozzáférést lehetővé tevő fv-eket sem hozzáadni a két osztályhoz (az operátoron kívül sose lenne jó alkalmazása).

```
constexpr rc_max {4}; //sor es oszlop merete
class Matrix;

class Vector{
  float v[rc_max];
  //...
  friend Vector operator*(const Matrix&, const
        Vector&);
};
```

```
class Matrix
{
  Vector v[rc_max];
  //...
  friend Vector operator*(const Matrix&, const
        Vector&);
};
```

Barátok III

- Ezek után az operátor függvény mindkét implementációhoz hozzáférhet:

```
Vector operator*(const Matrix& m, const Vector& v)
{
  Vector r;
  for(int i = 0; i!=rc; ++i)
  {
    r.v[i] = 0;
    for(int j=0; j!=rc_max; j++)
    {
      r.v[i] += m.v[i].v[j] * v.v[j];
    }
  }
  return r;
}
```

Barátok IV

- Egy osztály barát-függvénye nemcsak mezei függvény, hanem egy másik osztály tagfüggvénye is lehet:

```
class ListIterator
{
  //...
  int* next();
};

class List
{
  friend int* ListIterator::next(); //most mar ListIterator adott fv-e is hozzaferhet List
        belsejehez!
  //...
};
```

- Az ilyen megoldást mértékkel használjuk: innentől kezdve nem tudjuk pl. felsorolni (csak a *List* osztály alapján) hogy milyen függvények férnek majd hozzá a reprezentációhoz (minden, ami meghívja *next*()-et??).

Barátok V

- Osztályból, sőt template argumentumból is csinálhatunk barátot:

```
template<typename T>
class X
{
  friend T;
  friend class T; //mindketto jo
  //...
};
```

- Ilyenkor az adott osztály hozzáférhet ennek az osztálynak a privát tagjaihoz
- Végül: természetesen adott barát fv-t
  - vagy deklarálnunk kell az osztály definíciója előtt (hogy a fordító tudja, miről beszélünk egyáltalán)
  - vagy definiálnunk kell az osztályhoz tartozó namespace-ben

Barátok VI

- Tehát:

```
class C1{ }; //baratja lesz N::C-nek
void f1(); //baratja lesz N::C-nek

namespace N
{
  class C2 {}; //baratja lesz C-nek
  void f2() {} //baratja lesz C-nek

  class C
  {
    int x;
  public:
    friend class C1; //OK, mar deklaraltuk
    friend void f1();

    friend class C3; //OK, mert egyazon
          namespace-ben definialjuk
    friend void f3();

    friend class C4; //feltetelezzuk hogy ezek
          leteznek N nevterben
    friend void f4();
  };
```

```
  class C3 {\ldots};
  void f3() { C x; x.x = 1;} //OK mert C baratja
}//namespace N

class C4 {}; //nem baratja N::C-nek mert N
      nevteren kivuli!
void f4() {N::C x; x.x = 1;} //hiba: x private
      mert f4() nem baratja N::C-nek
```

Beágyazott osztályok

- Barátok lehetséges alternatívája: beágyazott osztályok
  - Legtöbbször esztétikai döntés dolga, hogy melyiket használjuk

```
template<typename T>
class Tree
{
  using value_type = T; //tag alias
  enum Policy {rb, splay}; //tag enum
  class Node
  {
    Node* right;
    Node* left;
    value_type value;
  public:
    void f(Tree*);
  };
  Node* top;
public:
  void g(const T&);
  //...
};
```

Beágyazott osztályok II

- Beágyazott osztály hivatkozhat bármire, ami a szülőosztályban van. Ellenben nem tudhatja, hogy milyen objektumra hívták meg az adott fv-ét. Pl:

```
template<typename T>
void Tree::Node::f(Tree* p)
{
  top = right; //hiba: a fv nem tudhatja, hogy melyik Tree objektumra hivtak meg
  p->top = right; //OK
  value_type v = left->value; //OK, value_type nem objektumhoz kotodik
}
```

- Ezért beágyazott osztály tényleg olyan, mint egy barát függvény. Az is a paramétereivel dolgozhatott, illetve azzal, amit saját maga hozott létre változót (de barát fv-t sem valamilyen objektum kontextusában hívhattunk meg)

Beágyazott osztályok III

- Továbbá: osztály sem férhet hozzá beágyazott osztály tagváltozóihoz, kivéve ha paraméterként kapja meg:

```
template<typename T>
void Tree::g(Tree::Node* p)
{
  value_type val = right->value; //hiba: nincs Tree::Node tipusu objektum, melyik right?
  value_type v = p->right->value; //hiba: Node::right private!
  p->f(this); //OK
}
```

Osztályhierarchiák

- Származtatott osztály maga is lehet maga szülőosztály.
- Az így létrejött osztály-hierarchia általában fa struktúrájú, de előfordulhatnak általánosabb gráf-típusok is. Mindez grafikusan is felrajzolható, aciklikus, irányított gráfként.

```
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 {/*...*/};
```

Osztályhierarchiák II

- Ha adva van egy *Base*∗ típusú változó, honnan tudjuk, hogy melyik származtatott osztályra mutat? Négyféle megoldás:
  - Kerüljük el ennek lehetőségét úgy, hogy garantáljuk hogy adott konténerben csak egyféle pointer lehet
  - Hozzunk létre ún. típus mezőt a szülőosztályban, melyet bármelyik fv megvizsgálhat
  - Használjunk *dynamic_cast*-ot
  - Használjunk virtuális függvényeket
- Az első megoldás önmagában feltételezi, hogy többet tudunk a típusokról, mint a fordító. Ez általában nem jó ötlet.
  - Egy dolog konkrét esetben azt mondani, hogy tudjuk, milyen objektumot várunk (*dynamic_cast*)
  - Más dolog, ha ehhez a képességhez mindig ragaszkodni próbálunk! A rugalmas kódhoz kell, hogy néha ne tudjuk / ne érdekeljen bennünket, milyen adatszerkezet van egy struktúrában

Osztályhierarchiák III – Típusmezők

- Az 1. és 4. megoldás együtt általában nagyon hatékony és tiszta (garantált felhasználási mód és virtuális fv-ek, ld. később). A 3. megoldás a nyelvbe beépített mechanizmust ad. De nézzük előbb a típus mezőket, hogy belássuk, miért érdemes őket elkerülni!
- Korábbi példánk újrafogalmazható:

```
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;
  }
  //...
};
```

Osztályhierarchiák IV – Típusmezők II

- Hogyan használnánk egy ilyen objektumot függvényben? Pl:

```
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;
  }
}
```

- A probléma: mostantól a fordító ellenőrzése nélkül manipulálunk típusokat!

Osztályhierarchiák V – Típusmezők III

- Még kevésbé átlátható, ha meg is próbáljuk kihasználni az osztályok közös részeit. Egy nagy projektnél ember legyen a talpán, aki ezt megérti:

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

- Továbbá: ha Employee szerkezete belül megváltozik, minden előfordulást ellenőriznünk kell!

Osztályhierarchiák VI – Típusmezők IV

- Ráadásul, ha egyszer belekezdtünk, hol álljunk meg?
  - Végül a szülőosztály mindenféle "hasznos információt" tartalmazhat, ami túl szoros kapcsolatot eredményez az osztályok implementációi között.
  - Mindez nem egészséges, nem skálázható.
- Ezért: sose használjunk típus-mezőt!
- Inkább használjunk virtuális függvényeket. Ezekről az előző órán már esett némi szó.

Osztályhierarchiák VII – Virtuális függvények

- Virtuális függvényekkel lehetséges megmondani, hogy szülőosztály egyes fv-ei újradefiniálhatóak.
  - A fordító és linkelő megteremti a megfelelő kapcsolatot objektum és fv között.
  - *virtual* módosító: származtatott osztályok felüldefiniálhatják adott fv-t, és automatikusan a "helyes" fv hívódik majd meg:

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

Osztályhierarchiák VIII – Virtuális függvények II

- Ehhez az is kell, hogy a paraméterek száma és típusa azonos legyen! Továbbá a visszaadott típus megváltoztatására is csak korlátozott lehetőségünk van:
  - Ha B osztály publikus szülőosztálya D-nek, akkor adhatunk vissza pl. $D*$-ot $B*$ helyett. Ugyanez lehetséges referenciákra is.
- Pl. tfh kifejezések kiértékelésére hozunk létre osztály-hierarchiát. A szülőosztály segítségével új kifejezést is létrehozhatunk:

```
class Expr
{
  Expr();
  Expr(const Expr&); //copy constructor
  virtual Expr* newExpr() = 0;
  virtual Expr* clone() = 0;
};
```

- *newExpr*() a kifejezéstől függően létrehoz valamilyen típusú kifejezést, *clone*() pedig létezőt másol.

Osztályhierarchiák IX – Virtuális függvények III

- Ez a megoldás felhasználható pl. így:

```
class Cond : public Expr
{
public:
  Cond();
  Cond(const Cond&);
  Cond* newExpr() override {return new Cond();}
  Cond* clone() override {return new Cond(*this);}
  //...
};
```

```
void user(Expr* p)
{
  Expr* p2 = p->newExpr();
}
```

- Virtuális fv-t kötelező saját osztályában definiálni is, kivéve ha pure virtual fv (ld. később, = 0 jelölés)
- Ha az osztályból nem származik semmi, virtuális fv-e akkor is használható.
- Ha származik is belőle valami, nem kötelező újradefiniálni a virtuális fv-t (kivéve, ha az pure virtuális)

Osztályhierarchiák X – Virtuális függvények IV

- Mindez lehetővé teszi a futásidejű polimorfizmust. Csak pointeren vagy referencián keresztül működik (ha objektumot használnánk, már a fordító ismerné a típusát, nem futásidejű lenne). Pl.

```
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});
}
```

- Vegyük észre, hogy ez akkor is működik, ha *printList* fv-t azelőtt fordítottuk le, mielőtt az alkalmazott származtatott osztályt létrehoztuk volna!!! Ez baromira jól strukturált megoldást jelent!

Osztályhierarchiák XI – Virtuális függvények V

- A helyes fv meghívásához a fordító ún. virtuális táblát használ
- Mindez elég hatékony, normál fv-híváshoz képest max. 25%-kal több. Használjuk bátran (a típus mező is jelentene overhead-et)
- Végül fontos szót ejteni az *override* és *final* módosítókról.
- Alapból ha származtatott osztályban virtuális fv-nyel megegyező nevű és típusú fv-t hozunk létre, trivi hogy felüldefiniálunk.
- Bonyolultabb esetekben nem biztos, hogy trivi.

Osztályhierarchiák XII – Override és final

- Itt a példában nem tudjuk, hogy mi szerepel B1..B5-ben, de valószínűleg:
  - $B0 :: f()$ nem is virtuális, nem felüldefiniálható, csak elrejthető.
  - $D :: g()$ más típusú, mint B0::g() (valószínűleg csak elrejti azt, nem definiálja felül)
  - Olyan nincs is, hogy $B0 :: h()$. Valószínű, hogy egy vadonatúj virtuális függvényt vezetünk be D-ben.

```
struct B0
{
  void f(int) const;
  virtual void g(double);
};

struct B1 : B0 {/* ... */};
struct B2 : B1 {/* ... */};
struct B3 : B2 {/* ... */};
struct B4 : B3 {/* ... */};
```

```
struct B5 : B4 {/* ... */};

struct D : B5
{
  void f(int) const;
  void g(int);
  virtual int h();
};
```

Osztályhierarchiák XIII – Override és final II

- Fentiek miatt többéle finomhangolásra lehetőséget adó megoldás létezik:
  - *virtual* módosító: a függvény felüldefiniálható
  - *virtualfv*(*params*) = 0; a függvényt kötelezi felüldefiniálni
  - *override* módosító: ez a fv felül szeretné definiálni a szülőosztály azonos nevű fv-ét
  - *final* módosító: ez a fv. semmiképpen sem definiálható felül
- Ezek a kulcsszavak megkönnyítik a fordító dolgát: a fordító szól, ha felrúgjuk a szabályokat. A korábbi dilemmák feloldhatóak:

```
struct D : B5
{
  void f(int) const override; //hiba: B0::f() nem virtualis
  void g(int) override; //hiba: B0::g() double argumentumot var
  virtual int h() override; //hiba: nincs mit overrideolni
};
```

Hasznos tanácsok Stroustruptól I

- Kerüljük a típus mezők alkalmazását
- Polimorfikus objektumokat pointereken vagy referenciákon keresztül érjünk el
- Interfészek tiszta létrehozásához használjunk absztrakt osztályokat
- Használjuk az override kulcsszót, hogy egyértelmű legyen, amit csinálunk
- A final kulcsszót csak ritka esetben alkalmazzuk
- Virtuális fv-nyel rendelkező osztályban legyen a destruktor is virtuális!

Hasznos tanácsok Stroustruptól II

- Implementációs részleteket privát tagokkal oldjuk meg
- Használjunk publikus tagokat az interfészhez
- Csak indokolt esetben használjunk protected tagokat
- Adattagokra (nem fv-ekre) sose használjunk protected-et
  - Ha valami publikus, tudjuk, hogy bármi elérheti
  - Ha valami protected, nem tudjuk, miért protected? Hiba esetén sok keresgélést eredményezhet!

Feladat

- Készítsünk osztályhierarchiát aritmetikai absztrakt szintaxis fák (AST-k) létrehozására!
- Pl. egy ilyen kifejezés parszolásához:
  - 4 + 5 * (3 + 2)
- Az alkalmazást bővíthetjük szimbolikus képességekkel is. Pl.
  - a + 5 * (b + 2) értéke legyen határozatlan mindaddig, amíg a és b értékét nem tudjuk!