1. Öröklés

Az objektum orientált tervezés fontos sarkköve, az osztályok viszonyainak a megtervezése.

1.1. Rétegelés

```
class Address { ..... };
class Name { ..... };
class Person {
   Name name;
   Address addr;
   .....
};
```

A rétegelésnél egy osztály adattagként más osztályú objektumot tartalmaz. A szakzsargonban azt mondják, hogy a `Person` osztály a `Name`, `Address` osztály fölé lett rétegelve, mert a feljebb levő osztály adattagként tartalmaz alatta lévő osztály típusú objektumot. A rétegelésre mondanak kompozíciót, tartalmazást, beágyazást. A rétegelés *"vanegy"*, vagy *keresztül implementált* relációt modellez.

1.2. Nyilvános öröklés - isa reláció

A nyilvános (publikus) öröklődés *(isa) "azegy"* relációt jelent. Még helyesebben az *"úgy működik mint"* kapcsolatot jelenti.

```
class Person { ..... };
class Student : public Person { ..... }
```

A fenti példában a diák `(Student)` *"azegy"* személy `(Person)` reláció igaz, tehát használhatjuk a publikus öröklődést.

```
void dance( const Person& p);    // Minden személy táncolhat
void study( const Student& s ); // Csak a diákok tanulnak.

Person p;
Student s;

dance(p);          // OK. p egy személy
dance(s);          // rendben s egy diák, "azegy"  személy

study(s);          // OK.
study(p);          // Hiba p nem diák.
```

A kódrészletben a `Student` típusú objektum úgy viselkedik, mint a `Person` típusú objektum, fordítva azonban ez nem igaz. A nyilvános öröklés lényege nem feltétlenül az alaposztály újrahasznosítása, hanem olyan új osztály létrehozása, ami minden helyen használható, a meglévő kód átírása nélkül, ahol az alaposztály is (pl. a `p` a `dance` függvény paramétereként).

1.3. Korlátozó öröklődés - has-a reláció

Az előadáson elhangzott, hogy bár matematikailag a négyzet *"azegy"* speciális téglalap, amelynek minden oldala egyenlő, mégsem igaz az *"azegy"* reláció, mivel a négyzet nem *úgy viselkedik, mint* a téglalap (nem változtathatjuk meg csak egyik irányban az oldala hosszát). Ugyanakkor a téglalap osztály segítségével egyszerűen implementálhatjuk a négyzet osztályt.

```
class Rectangle { ..... };
class Square : private Rectangle { ..... };
```

1.4. Korlátozó öröklődés vagy rétegelés?

Feladat egy olyan általános, stack osztálynak a létrehozása, ami többféle adatot is képes tárolni. Ezt legjobban sablon (template) segítségével oldhatjuk meg, de többfajta adattípus példányosítása esetén ez *code bloat*-hoz (kódduzzadáshoz) vezethet. Ezért hozzunk létre egy Stack osztályt, ami általános `void*` pointereket tárol.

```
class GenericStack {
public:
    GenericStack() : top(0) {}
    ~GenericStack();
    void push(void *object);
    void * pop();
    bool empty() const;
private:
    struct StackNode {
        void* data;
        StackNode *next;
        StackNode(void *newdata, StackNode *nextnode)
         : data(newdata), next(nextnode) {}
    };
    StackNode *top;
    GenericStack(const GenericStack&); // másolás letíltása
    GenericStack& operator=(const GenericStack& ); // értékadás letíltás
};
```

Sajnos ezt az osztályt könnyű rosszul használni, például ha vegyesen `int*` és Person * mutatókat tárolunk benne. Ilyenkor a `pop` nem tudhatja, hogy milyen típusú adatot tettünk be, ez könnyen programhibához vezethet. Ezért definiálni kell egy típushelyességet biztosító felületosztályt.

```
class IntStack {
public:
    void push(int* intptr){ s.push(intptr); }
    int* pop(){ return static_cast<int*>( s.pop()); }
    bool empty() const { return s.empty(); }
private:
    GenericStack s;
};
```

A `GenericStack` felhasználását rétegeléssel oldottuk meg. Mindig ezt a megoldást válasszuk, ha valami más szempont nem merül fel.

A *titánokat* semmi sem tartja vissza, hogy típushelyességet biztosító felületosztály nélkül használják a `GenericStack` osztályt, majd a `(int *)s.pop()` cast-olást ők elvégzik. Ezt meg kell akadályozni, úgy hogy a `GenericStack` osztályt ne lehessen önmagában használni.

```
class GenericStack {
protected:          // Eldugjuk a konstruktorokat
    GenericStack() : top(0) {}
    virtual ~GenericStack();
    void push(void *object);
    void * pop();
    bool empty() const;
private:
    struct StackNode { .... } // ugyanaz mint előbb
    StackNode *top;
    GenericStack(const GenericStack&); // másolás letíltása
    GenericStack& operator=(const GenericStack& ); // értékadás letíltás
};
```

```
GenericStack s;    // Hiba! a konstruktor védett.
```

Így már nem lehet rétegelt megoldást használni, csak a privát öröklődés felhasználásával készíthetünk típushelyességet biztosító felületosztályt.

```
class IntStack : private GenericStack {
public:
   void push(int* intptr){ GenericStack::push(intptr); }
   int* pop(){ return static_cast<int*>( GenericStack::pop()); }
   bool empty() const { return GenericStack::empty(); }
};
```

Tanulság, hogy ha öröklődéssel lehet elérni a védett tagokat, vagy az osztály virtuális függvényt tartalmaz, akkor a *keresztül implementálást* privát (korlátozó) öröklődéssel oldjuk meg, különben az egyszerűbb és biztonságosabb rétegelési technikát válasszuk.

1.5. Származtatott osztály értékadó operátora

Fontos, hogy az alaposztály értékadásáról se feledkezzünk meg.

```
class Base {
   int x;
public:
   Base(int value=0) : x(value) {}
};

class Derived : public Base {
   int y;
public:
   Derived(int v1=0, int v2=0) : Base(v1), y(v2) {}
   Derived(const Derived& d) : Base(d), y(d.y) {} // Laboron volt.
   Derived& operator=( const Derived& rhs ) {
      if( this != &rhs ) {
         Base::operator=(rhs); // Alaposztály értékadása
         y = rhs.y;            // Tagváltozó értékadása
      }
      return *this;
   }
};
```

1.6. Szeletelődés

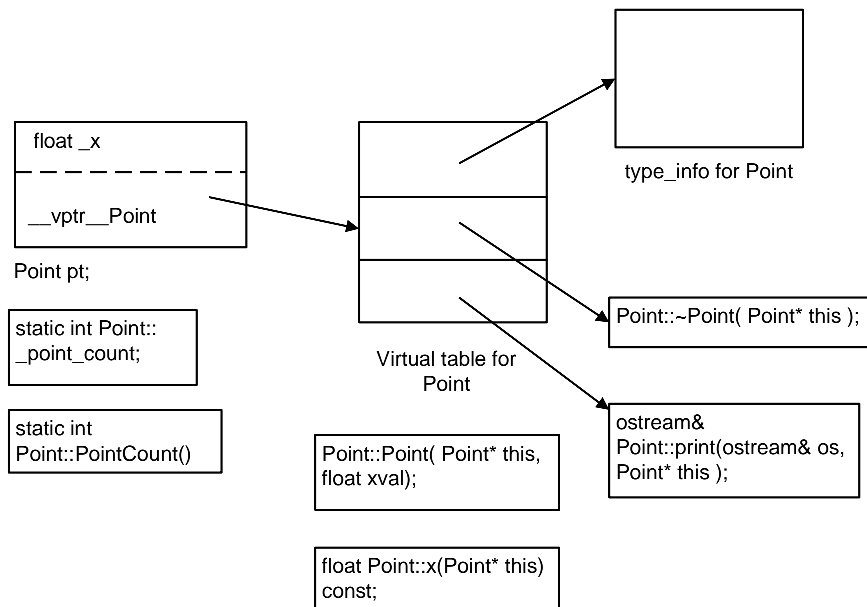
```
class A {
public:
   virtual void Kiir() { cout << "A kiir" << endl; }
};

class B : public A {
public:
   void Kiir() { cout << "B kiir" << endl; }
};

void f( A& x ) { cout << "f: "; x.Kiir(); }
void g( A x ) { cout <<"g: "; x.Kiir(); }
```

```
int main() {
   A a;    B b;
   a.Kiir();                          // OK. "A kiir"
   b.Kiir();                          // OK. "B kiir"
   A* p1 = &a;    p1->Kiir();         // OK. "A kiir"
   A* p2 = &b;    p2->Kiir();         // OK. "B kiir"
   f( b );                            // OK. "f: B kiir"
   g( b );                            // Szeletelődés "g: A kiir"
   A v = b;
   v.Kiir();                          // Szeletelődés "A kiir"
   return 0;
}
```

Nem csak a pointerrel, hanem a referencia alkalmazásával is meg tudjuk gátolni a szeletelődést.

1.7. C++ objektum modell

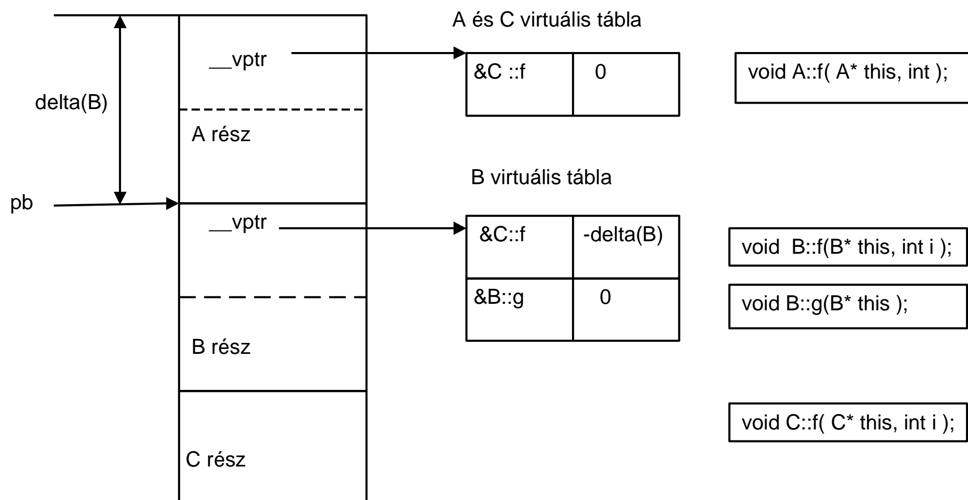
```
class Point {
public:
   Point( float xval );
   virtual ~Point();
   float x() const;
   static int PointCount();
protected:
   virtual ostream& print( ostream &os ) const;
   float _x;
   static int _point_count;
};
```



1. ábra. C++ objektum modell

1.8. Virtuális függvény

```
class A {
public:
   virtual void f(int);
 };
 class B {
 public:
    virtual void f(int);
    virtual void g();
 };
 class C : public A, public B {
 public:
    void f(int);
 };
```



2. ábra. virtuális függvények

Virtuális függvény hívása:

```
void ff(B* pb ) {
   pb->f(2);
}
```

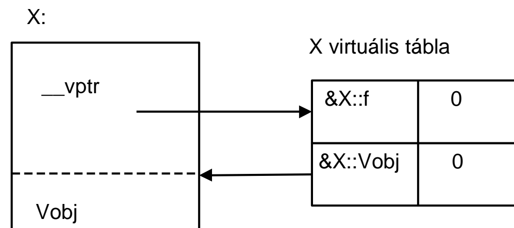
Megvalósítás:

```
 vtbl_entry* vt = &pb->__vptr[index(f)];
 (*vt->fct)((C*)( (char*)pb+vt->delta), 2 );
```

Az `index(f)` értékét a fordító ismeri.

1.9. Virtuális alaposztály

A C++ nyelvben virtuális szó valami mágikus tulajdonságot jelent. Pontosabban azt lehet mondani, hogy a virtuális függvény olyan függvény, amit indirekcióval lehet meghívni. A virtuális alaposztály olyan alaposztály, aminek nincs fix helye az osztályon belül, azaz csak indirekcióval lehet elérni.



3. ábra. virtuális alaposztály