A C++ programozási nyelv

A C++ nem objektum orientált újdonságai

- *Struktúraváltozók megadásához elegendő a* struct *nélküli típusnév*
- *Referencia típus*
- *Dinamikus memóriakezelés* new *és* delete *operátorral*
- *Függvényargumentumok alapértelmezett bemenőértékkel*
- *Függvényátdefiniálás: azonos függvénynév, más bemenet és működés*
- *Inline függvények*
- *Utasítások közötti változódefiniálás*
- *Egysoros megjegyzések*

- ***Struktúraváltozók megadásához elegendő a* struct *nélküli típusnév***

```
struct hallgatorektip {char nev[20];  char tankor[5]; } ;
hallgatorektip hallgatovekt[120] ;
void main()
{
 typedef struct bolygotip {char nev[15] ;
                           unsigned int atmero ;
                           long int tavolsag ;
                           float kering_ido ;
                           char evvagynap[4] ;
                          } ;
 bolygotip bolygok[9] ;
 struct komplextip {double Re ; double Im ;} ;
 komplextip cmp, *cmpmut ;
 //...
}
```

Referencia típus

A referenciaváltozó egy létező változó helyettesítő neve, mely ugyanannak a tárterületnek ad nevet.

```
int     valt;
int&  valtref = valt;
valt = 11;   printf("%d", valtref );        //  11
```

A referenciaváltozó és a helyettesített változó a műveletekben felcserélhető.
Önálló változóként nem definiálható, definiálásakor azonnal meg kell mondani, mely változót helyettesíti, azaz értéket kell adni neki.

Ha egyenértékű a helyettesített változóval, akkor mire jó mégis?

Címszerinti értékátadás függvénynek referenciaváltozóval

Amennyiben függvény paraméterében módosult értéket szeretnénk visszakapni, eddig csak a mutatóval történő megoldást alkalmazhattuk:

```
void szelsoertekek(int a, int b, int* max, int* min)
{ if  (a>b) { *max = a; *min= b; } else { *max = b; *min = a;} ;
}
// a függvény alkalmazása:
int c=3, d=5, nagy, kicsi ;
szelsoertekek(c, d, &nagy, &kicsi );
```

*A függvény törzsében a * indirekció operátort, a függvényhívásnál a & címe operátort kellett használni, ami körülményessé teszi a C nyelvben hiányzó címszerinti átadás miatt a paraméterben való értékvisszaadást.*

- *A C++ megoldása: címszerinti értékátadás referenciaváltozóval!*

```
void szelsoertekek(int a, int b, int& max, int& min)
 { if  (a>b) { max = a; min= b; } else { max = b; min = a;} ;
 }
 // a függvény alkalmazása:
 int c=3, d=5, nagy, kicsi ;
 szelsoertekek(c, d, nagy, kicsi ); //max megváltozása a helyettesített
                                    // külső nagy-ban is érzékelhető
```

A megoldás emlékeztet a Pascal var szócskával történő címszerinti paraméterátadására.

A referenciaváltozó alkalmazásával megkülönböztethetjük a módosításra átadott paramétereket a továbbra is célszerűen mutatóval átadott tömböktől, pl. string-ek átadásától.

Előnyös a referenciaváltozóval történő átadás nagyméretű, csak bemenő paraméter esetén is, mivel nem kell azt átmásolni a verembe.

Dinamikus memóriakezelés* new *és* delete *operátorral

A C nyelvben a dinamikus memória foglalására az malloc() *és a* calloc() *függvények használatosak. Hátrányuk, hogy a típuskonverziót és a foglalandó adat méretét is a programozónak kell megadnia. Emlékeztetőül egy 200 elemű valós vektor lefoglalása a Heap-en:*

```
#include <alloc.h>
float * vektmut ;
vektmut = (float*) calloc(200, sizeof(float));
//...
free( vektmut );
```

A C++ nyelvnek részét képezik a dinamikus memóriakezelést végző **new** *és* **delete** *operátorok.*

A **new** *és* **delete** *operátor használata*

Példa egy skalár és egy vektor dinamikus foglalására:

```
float * valosmut;
float * vektmut;
valosmut = new float;
vektmut  = new float[200];
if (!valosmut || !vektmut)      // A mutatók értéke NULL, ha sikertelen
 {puts("Memóriafoglalás sikertelen! "); exit(1);}
// A létrehozott dinamikus változók használata itt
delete[ ] vektmut;
delete valosmut;
```

- *További tudnivalók a* **new** *és* **delete** *operátor használatához*
  - *Fontos: minden lefoglalt memóriát egyszer fel is kell szabadítani, azaz a* **new** *és* **delete** *operátorok mindig párban legyenek!*
  - *Kisméretű adatokat lokális változókként adjunk meg, azaz a rendszer kezelje a veremben történő automatikus helyfoglalásukat és felszabadításukat! A nagyméretű adatoknak viszont a* Heap*-en foglaljunk dinamikusan helyet a* **new** *és* **delete** *operátorokkal!*
  - *A definiáláskor azonnal nem inicializált mutatókat* NULL*-ázzuk le!*
  - *Ne töröljünk egy nem* NULL *értékű mutatót kétszer!*
  - *Állítsuk* NULL*-ára a mutatókat a* **delete***-vel való törlés után!*
  - *Dinamikusan foglalt tömböket a* **delete[ ]** *operátorral töröljünk!*

Függvényargumentumok alapértelmezett bemenőértékkel

A függvényparaméterek kezdeti értékkel való ellátásának kettős célja:

- *A C-ben és a C++-ban is elhagyható függvényparaméterek hiányából eredő, nehezen észrevehető hibalehetőségek megszüntetése*
- *A C++ által kiemelten kezelt kezdőérték adás kiterjesztése a függvényparaméterekre is.*

A függvényparaméter számára a függvény definiálásakor megadott kezdőérték olyankor lép életbe, amikor az adott paramétert függvényhíváskor nem adjuk meg. Paraméterek híváskor csak az aktuális paraméterlista végéről kezdve hagyhatók el.
Alapértelmezett értékkel olyan paramétert célszerű ellátni, amely ugyanazt az értéket kapja a függvény legtöbb hívásakor.

- *Példák alapértelmezett paramétereket használó függvényekre*

  - *Készítsünk olyan kitöltött ellipszist rajzoló függvényt, amely körlapot rajzol, ha a negyedik paraméterként szereplő függőleges féltengely értékét nem adjuk meg!*

```
void MyFillellipse(int x, int y, int xradius, int yradius= 0)
{if (yradius)  fillellipse(x, y, xradius, yradius) ;
 else  fillellipse(x, y, xradius, xradius);
}
// Függvényhívás:
MyFillellipse(100,100,50);        // 50 sugarú kitöltött kört rajzol
MyFillellipse(200,200,70,30);     // kitöltött ellipszist rajzol
```

Megjegyzés: elegáns lett volna, de csak konstans kezdőérték adható meg:

void MyFillellipse(int x, int y, int xradius, int yradius= xradius) {...}

Függvényátdefiniálás

Pascal nyelv után a C-t tanulva idegesítőnek tűnik a nyelv típusérzékenysége, pl. az abszolutérték függvénynek más-más neve van attól függően, milyen típusú argumentummal működik:

int abs(**int** *x*); **double** fabs(**double** *x*);

long int labs(**long int** *x*); **double** cabs(struct complex *z*);

*A C++ kiküszöböli a C nyelv ezen gyengeségét és bevezeti a függvényátdefiniálás (**function overloading**) lehetőségét. Az eltérő paraméterszámú és/vagy eltérő típusú paramétereket használó (**eltérő paraméterszignatúrájú**) azonos nevű függvények eltérő működést jelentő, más-más függvénydefiníciót kaphatnak.*

Jól egyezik ez a lehetőség a valós világban előforduló többjelentésű tevékenységnevekkel. Más tevékenységet takar például:

megkapni egy levelet, megkapni a magáét, megkapni a náthát.

- *A probléma megoldása a C++ függvényátdefiniálási lehetőségével*

A C++ nyelvben az összes abszolutérték számítást abs() *nevű függvénnyel végezhetjük, miután megadtuk az eltérő paraméterszignatúrájú függvények definícióit:*

```
int        abs(int x);          // ez adott
double  abs(double x)     // paraméterszignatúra: double
  {return fabs(x);}
long int abs(long int x)    // paraméterszignatúra: long int
  {return labs(x);}
double  abs(complex z)    // parameterszignatúra: complex
  {return cabs(z);}
```

Valójában a fordító más-más nevű függvényeket hoz létre, a nevek rejtett eltérését a paraméterszignatúra felhasználásával éri el.

Az átdefiniálással kapott névrokonfüggvények használata

```
double r, v=23.456;
int i, j;
r = abs(v);    //valójában az fabs( ) függvény működik
i = abs(-12); //ERROR!! A fordító nem tud választani a lehetséges
    //függvények közül!! Megoldás: az érték változóba helyezése, vagy
    //explicit típuskonvertálás alkalmazása:
j = -12; i = abs( j );  //szignatúra egyértelmű: int
i = abs( (int)-12 );   //a típusátalakítás egyértelmű szignatúrát ad: int
```

Az explicit típusátalakítás gyakran használható, ha típuseltérési probléma miatt nem megfelelő a szignatúra.

A függvény nevének típusa nem része a szignatúrának! A következő két prototípus fordítási hibát generál:

```
int      fuggv( );
double fuggv( );   //ERROR!!
```

Inline függvények

Az inline függvények olyan rövid függvénytörzsű függvények, amelyek lefordított alakja az alkalmazás helyén a programkódba illeszthető a program méretének jelentős megnövekedése nélkül.

Pl.: **inline long int** negyzet(**int** *x*) {**return** *x***x*;}

A C++ nyelvben történő bevezetésének két indoka:

- *Az objektum orientált programozás fő építőkövei az objektumok, melyek osztályainak tagfüggvényei között sok kis méretű, elkülönülő prototípust és definíciót nem igénylő függvény van.*
- *A C nyelv hasonló szerepű* függvény makroi *veszélyes hibákat tudnak produkálni, különösen mellékhatással bíró léptető-operátoros argumentumokkal történő aktualizálás esetén.*

Pl.: #**define** negyzet(*x*) (*x*)*(*x*)

```
//alkalmazása:
int x= 2, y;
y= negyzet(++x);   // várt y == 9, ehelyett: y ==16
```

Utasítások közötti változódefiniálás

- *A C nyelv lokális változóit a blokkok elején, az utasításokat megelőzően kellett definiálni. A C++ nyelv ezen túl megengedi a változódefiníciók elhelyezését az utasítások között. Minden olyan helyen, ahol utasítás állhat, állhat változódefiníció is. Az ilyen változó akkor jön létre a veremben, amikor a definiálása megtörtént és ettől a ponttól kezdve a blokk végéig hivatkozható érvényesen.*
- *Előnye ennek a lehetőségnek, hogy a változók nem válnak el az alkalmazási területüktől és inicializálásuk is magától értetődő a felhasználás helyén. Ezáltal kisebb az esélye annak, hogy a jóval korábban definiált, inicializálatlan változóra hivatkozzunk.*

Tipikus példa a for ciklusok változóinak definiálása magában a for ciklusban:

```
for (int i =1, int gyujto=0; i<=100; i++) gyujto += i; //szummázás 100-ig
```

Egysoros megjegyzések

A C nyelv a forrásszövegben elhelyezett kommentek, megjegyzések megadására a

```
/*   megjegyzés
*/
```

formát ismeri. A C++ ezen felül bevezette az egysoros megjegyzéseket is a következő alakkal:

```
// ez a szöveg a sor végéig megjegyzésnek minősül
```

A régi és az új forma keverten is alkalmazható:

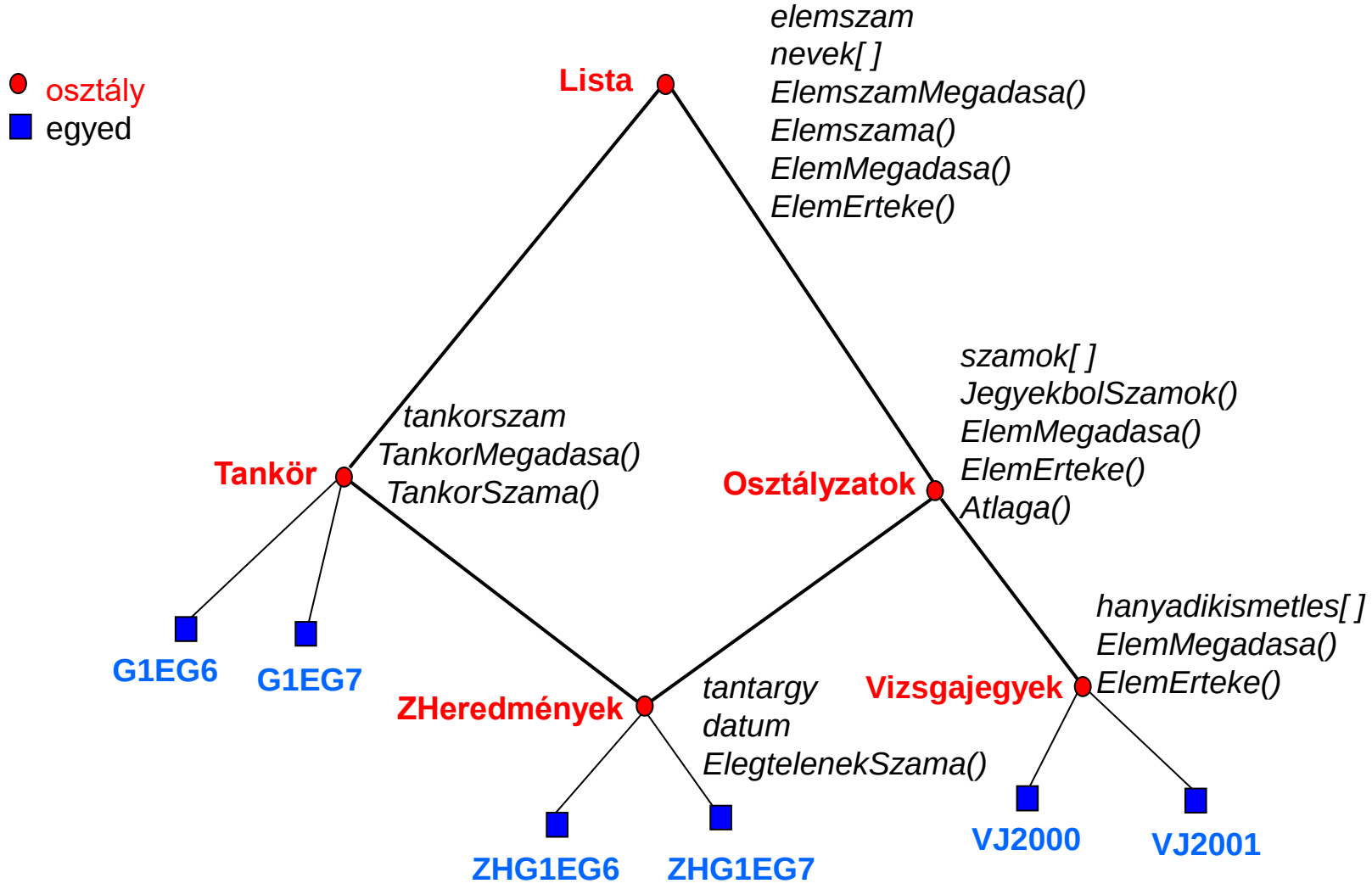
```
// új új új új új új /* régi régi régi */ új új új új új új új új új új új
/* régirégirégirégirégi // új új új új új új új új új új új új új új új új új
   régirégirégirégirégirégirégi
   régirégirégirégi */
```

A C++ programozási nyelv

Az objektum orientált programozás alapjai C++ nyelven

- *A világ objektum alapú szemlélete*
- *Szoftverfejlesztési módszertanok az objektum orientáltság előtt*
- *A C++ objektum orientáltságának jellemvonásai*
- *A class típus*
- *Hivatkozás az osztály elemeire*
- *A tagfüggvények definiálásának másik módja*

A világ objektum alapú szemlélete



A világ modellezése az objektum alapú szemlélet alapján

A világ együttműködő, egymásra ható valós és elvont objektumok rendszere. Az objektumoknak jól megkülönböztethető a külvilággal kapcsolatot tartó felszíne - az **interfész** *- és az objektum lényegét adó* **belseje***. A valóság modellezésénél általában a valós objektumok egyszerűsített modelljével dolgozunk, amely a hasonló objektumok összességének közös tulajdonságait írja le, egy osztályba, típusba fogva azokat. Az osztály megtestesíti az egyedek* **funkcionalitás***át, az egyedek megkülönböztetését lehetővé tévő egyedi állapot jellemzőinek tárolására pedig* **adatszerkezeteke***t nyújt.*

Az objektum orientált programozás (OOP) **módszertana** *eszközt ad a valóság objektumainak absztrahálására, leképezésére, alkalmazása során létrehozzuk a valóságot modellező programegységeket és ezek kapcsolatrendszerét.*

Egy objektum az összes, adott absztrakcióban hozzá köthető funkcionalitás működtetéséért és jellemvonás képviseléséért **felelős***, és csak azért.*

A C++ objektum orientáltságának jellemvonásai

- **Adatok és adatkezelő alprogramok egységben (encapsulation)**

  Az egy objektum jellemzésére használt adatok és az objektum funkcionalitását megvalósító függvények egy egységbe, az objektumba vannak foglalva, elzárva és védve az objektum külvilág számára érdektelen belső részeit a külvilág elől. A hasonló objektumok azonos jellemzőit az objektumok ***osztály****ának leírására használjuk, míg specifikus jellemzőik - adatértékeik - alapján* ***egyed****ekként különböztetjük meg őket.*

- **Az osztálytulajdonságok továbbörökítése alosztályokra, egyedekre**

  Az örökítés (inheritance) az osztályok adattagjainak és függvényeinek megjelenését jelenti az utódosztályban, vagy egyedben. Az utódosztályok az ősöktől örökölt tulajdonságokon felül újabb tulajdonságokat: adattagokat és függvényeket (tagfüggvényeket, metódusokat) is birtokolhatnak. Az utódosztály több szülőosztálytól is örökölhet. Az osztályok az ős-utód kapcsolatok révén osztályhierarchiát alkotnak.

A C++ objektum orientáltságának jellemvonásai, folyt.

- **Azonos nevű függvények eltérő működése (polymorphism)**

  Alapvetően azonos feladatra szánt, azonos nevű függvények működése iránt egy öröklési lánc osztályhierarchia szintjein lévő osztályok többé-kevésbé eltérő igényeket támaszthatnak, amelyet a függvények eltérő függvénytörzzsel elégítenek ki. Ilyen esetben az ősosztály ***virtuális függvény****ének azonos szignatúrájú függvénnyel történő újradefiniálása zajlik.*

  *A C++ meghökkentő rugalmassága érhető el az operátorok feladatának átdefiniálása által. Például elérhető, hogy a * operátor két mátrix összeszorzását végezze el. Ez az* ***Operator overloading*** *is a korábban említett névrokon függvényekhez hasonlóan erősen paraméterszignatúra-függő.*

A class típus

A class (osztály) típus a C nyelv struct típusából fejlődött ki. A class megvalósítja az adattagok és a tagfüggvények egységbezárását.

Példaként definiáljuk a Lista osztályt a korábbi ábrán megadott adattagokkal és tagfüggvényekkel!

```
class Lista
{private : int    elemszam;
           char nevek[300][25];
 public  : void ElemszamMegadasa(int elemszambe){elemszam= elemszambe;}
           int    Elemszama( ){ return elemszam;}
           void ElemMegadasa(int i, char* nevbe) {strcpy(nevek[ i ], nevbe);}
           void ElemErteke(int i, char* nevki ){strcpy(nevki, nevek[ i ]);}
};

Lista listavaltozo;  //létrehoztuk a Lista osztály egy egyedét
```

Hivatkozás az osztály elemeire

Az előző példa annyiban speciális, hogy az összes tagfüggvényt inline módon definiáltuk.

Adjunk értéket a listavaltozo adattagjainak!

```
listavaltozo. ElemszamMegadasa(25);  //25 név lehet benne
listavaltozo. ElemMegadasa( 0, "Kiss Lajos" ) ;
listavaltozo. ElemMegadasa( 1, "Nagy Izidor" ) ; // További nevek megadása
                                                 // hasonlóan
  // Írassuk ki az elemek számát és a második nevet!
printf("A listában %d név van tárolva.", listavaltozo . Elemszama( ) );
char nevki [25];
listavaltozo . ElemErteke( 1, nevki ) ;
puts( nevki );
```

A tagfüggvények definiálásának másik módja

Inline függvényekkel csak rövid, egysoros függvényeket célszerű definiálni.

Ha a tagfüggvény terjedelmesebb, a szokásos deklaráció - definíció *párost alkalmazzuk:*

```
class Lista
{private : int    elemszam;
           char nevek[300][25];
 public  : void ElemszamMegadasa(int elemszambe);        // deklaráció
           int    Elemszama( ){ return elemszam;}
           void ElemMegadasa(int i, char* nevbe) {strcpy(nevek[ i ], nevbe);}
           void ElemErteke(int i, char* nevki ){strcpy(nevki, nevek[ i ]);}
};
void Lista :: ElemszamMegadasa(int elemszambe)          // definíció
 {elemszam= elemszambe;}
```

- **A C++ programozási nyelv**

Az objektum orientált programozás alapjai C++ nyelven

- *Konstruktor és destruktor*
- *Példa a kezdőérték paramétert váró objektum definiálására*
- *Az osztálytulajdonságok továbbörökítése alosztályokra, egyedekre*
- *Mintaprogram egyszerű osztályhierarchiára*
- *A konstruktorok hívási sorrendje*
- *A destruktorok hívási sorrendje*
- *Megjegyzések a példaprogramhoz*

Konstruktor és destruktor

A valós világ objektumai keletkeznek és megszűnnek. Szükség lehet az objektum keletkezésekor bizonyos alapbeállítások, inicializálások elvégzésére. Ugyanígy szükség lehet az objektum megszűnése pillanatában bizonyos tevékenységek elvégzésére, hogy az objektum nyomtalanul eltűnjön.

Idézzük ide korábbi osztály példánkat:

```
class Lista
{private : int    elemszam;
           char nevek[300][25];      // rögzített méretű férőhely!?
 public  : ...
};
```

Az osztály deklarációjából látszik, hogy egy ilyen típussal létrehozott objektum mindig 300 név számára biztosít férőhelyet, függetlenül attól, hogy többre, vagy éppen kevesebbre lenne szükség. A problémát könnyen megoldhatjuk, ha az elemszam *tényleges értékét és ezáltal a szükséges férőhelyet is csak az objektum létrehozásakor, azaz az objektumváltozó definiálásakor adjuk meg.*

Ehhez a következőképpen kell az osztályt deklarálni:

```
typedef char nevtip[25]; //ez a névtípus megkönnyíti a program későbbi módosítását
class Lista
  {private : int        elemszam;
             nevtip * nevek;       // csak egy nevekre mutató pointert adunk meg
   public  :            Lista( int elemszam0 = 0 );  // konstruktor függvény deklarációja.
                                  // elemszam0 paraméterrel fogjuk megadni a szükséges
                                  // elemszámot amikor ilyen osztályba tartozó objektumot
                                  // hozunk létre. Ha nem adjuk meg, 0 lesz az elemszam.
                        ~Lista( ); // destruktor függvény deklarációja, nincs soha paramétere!
                         …          // korábban említett további tagfüggvények vannak itt még
  };
// A konstruktor és a destruktor definíciója:

Lista :: Lista( int elemszam0 )     // itt már nem szabad megadni a 0 default értéket!
{ elemszam = elemszam0;
  nevek = new nevtip[ elemszam0 ];   // dinamikusan lefoglaljuk a szükséges helyet
  for ( int i = 0; i<elemszam; i++)  nevek[ i ][0]= '\0';  // üres stringek lesznek a nevek
}

Lista :: ~Lista( )
{ if ( nevek ) delete[ ] nevek;  }      // a dinamikusan lefoglalt helyet felszabadítjuk
```

Példa a kezdőérték paramétert váró Lista osztály objektum egyedének (objektum változónak) a definiálására:

```
Lista vezerek(7);  // Hét vezér nevének tárolására alkalmas egyetlen Lista típusú
                   // objektum létrehozása inicializálással
```

Ha a konstruktor összes paramétere kap alapértelmezett értéket, akkor objektumokból álló tömb definiálása esetén nem okoz gondot az aktuális paraméterértékek hiánya (BCB):

```
Lista listak[25];  // 25 darab Lista típusú objektumból álló vektor
```

Ha akár csak egy paraméter is alapértelmezett kezdőérték nélküli, akkor egy további, paramétert nem váró konstruktort is alkalmazni kell az osztályban:

```
class Lista
{ …
  public: Lista( int elemszam0 ) ; // Default paraméterérték nélküli konstruktor.
          Lista( ){ }              // Paramétert nem váró második konstruktor.
  …
};
```

```
Lista nevsorok[25];  // Ilyenkor az objektumok inicializálásáról külön kell gondoskodni.
```

Megjegyzések a konstruktorokkal kapcsolatban:

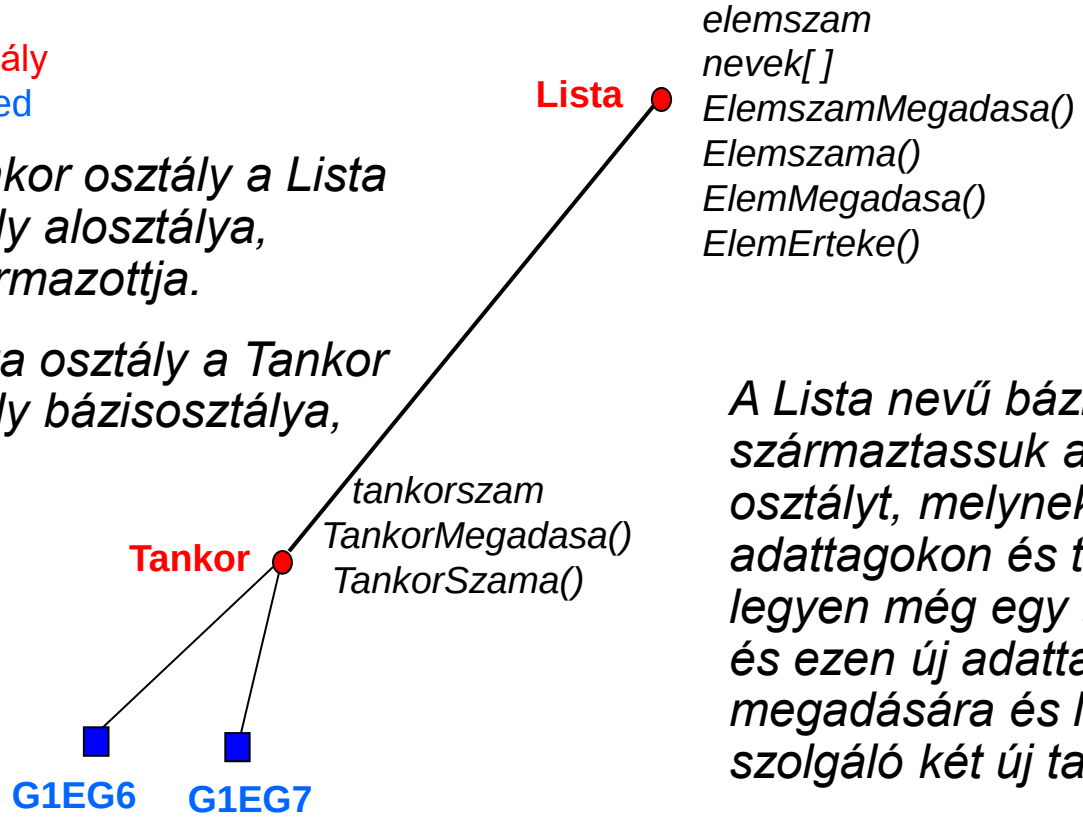
- *Ha nem adunk meg konstruktort, akkor a fordító egy rejtett* ***automatikus konstruktort*** *kreál.*
- *A programozó által megadott konstruktor megoldja az objektumnak való kezdőértékadás problémáját.*
- *A konstruktor nem ad vissza a nevében értéket, de még csak void típusa sincs.*
- *Ha a programozott konstruktorunknak nincs kezdőérték paramétere, akkor az objektumdefiníciónál az üres, kezdőérték nélküli zárójelpár elmaradhat:*
  pl.: *Lista2* :: Lista2( ) {...} *konstruktordefiníció mellett a*
  Lista2 listaobjektum( ), *masiklistaobjektum*; *objektumdefiníciók helyesek.*
- *Egy osztályban több konstruktor is lehet, más-más paraméterszignatúrával.*
- *Ha az osztály típussal tömböt is szeretnénk létrehozni, kötelező a default konstruktor, mely lehet automatikus, vagy paraméter nélküli, ill. csak default paraméteres programozott kivitelű.*
- *Objektumok dinamikus foglalása:*
  *Lista * vezerekp* = **new** *Lista*(7); *Lista * nevsorokp* = **new** *Lista*[25];
  megszüntetése: **delete** *vezerekp*; **delete**[ ] *nevsorokp*;

Az osztálytulajdonságok továbbörökítése alosztályokra, egyedekre

osztály
egyed

A Tankor osztály a Lista osztály alosztálya, leszármazottja.

A Lista osztály a Tankor osztály bázisosztálya, őse.



Hozzuk létre a Tankor osztályba tartozó két objektumot, azaz egyedet:

a G1EG6 és G1EG7 tanköröket!

A Lista nevű bázisosztályból származtassuk a Tankor nevű osztályt, melynek az öröklött adattagokon és tagfüggvényeken túl legyen még egy tankorszam adattagja és ezen új adattag értékének megadására és lekérdezésére szolgáló két új tagfüggvénye, a

TankorMegadasa()

és a

TankorSzama().

Mintaprogram egyszerű osztályhierarchiára

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
typedef char nevtip[25];   // felhasználói típus név tárolására

class Lista
{ protected:   int    elemszam;
               nevtip * nevek;
  public:    Lista(int elemszam0 = 0);        // konstruktor, default 0 elemszámmal
            ~Lista( );                        // destruktor
        void ElemszamMegadasa(int elemszambe) {elemszam = elemszambe; }
        int    Elemszama( ) { return elemszam; }
        void ElemMegadasa(int i, char* nevbe) {strcpy(nevek[ i ], nevbe); }
        void ElemErteke(int i, char* nevki ) { strcpy(nevki, nevek[ i ]); }
};

// A konstruktor és destruktor definícióját lásd a program végén
```

```
class Tankor : public Lista   // Tankor osztály származtatása a Lista osztályról
{ protected: char   tankorszam[ 6 ];   // pl.: "G1EG6"
  public:    Tankor(int elemszam0 = 0, char * tankorszam0 = " " ); // konstruktor,
                        // default 0 elemszámmal és üres sztringgel tankörszám helyett
             ~Tankor( ){ };          // destruktor
        void TankorMegadasa(char* tankorbe) {strcpy(tankorszam, tankorbe); }
        void TankorSzama(char* tankorki ) {strcpy(tankorki, tankorszam); }
};

// A konstruktor definícióját lásd a program végén

// Objektumváltozók definiálása:

Tankor G1EG6(25, "G1EG6"), G1EG7(28, "G1EG7"); // a konstruktor kétszer futott

void main( )

{ clrscr();
  puts("Tankör objektumok létrehozása és használata\n\n");

  puts("Adja meg a G1EG6 tankör hallgatóinak nevét! " );

        //folytatódik
```

```
  nevtip nevStr;

  for ( int k=0; k<G1EG6.Elemszama( ); k++)
  { printf("A %d. hallgató neve: ", k);
    gets(nevStr ); G1EG6. ElemMegadasa( k, nevStr );
  }

  char tankorki [7];

  G1EG6.TankorSzama( tankorki ); // megkérdezzük a tankörszámot

  printf("A %s tankör hallgatói a következők: \n", tankorki );

  for ( k=0; k<G1EG6.Elemszama(); k++)
  { printf("A %d. hallgató neve: ", k );
    G1EG6.ElemErteke(k, nevStr );  puts(nevStr );
  }

  printf("Létszáma= %d fő", G1EG6.Elemszama( ) );

  getch( );

} // main függvény vége

        //folytatódik
```

```
// A konstruktor és destruktor definiciók:

Lista :: Lista( int elemszam0 )                // a Lista osztály konstruktora
{ elemszam= elemszam0;
  nevek = new nevtip[ elemszam0 ];      //dinamikusan foglaljuk a vektort
  for (int i = 0; i < elemszam0; i++ )  nevek[ i ][0]= '\0' ;
}

Lista :: ~Lista( )   // a Lista osztály destruktora
{ // a dinamikusan foglalt nevek vektort felszabadítjuk
  if ( nevek ) delete[ ] nevek;
}

Tankor :: Tankor( int elemszam0, char * tankorszam0 ) : Lista( elemszam0 )
{ strcpy( tankorszam, tankorszam0 );
}
// A Lista szülőosztály konstruktora csak ilyen módon kaphat kezdőértéket,
// hiszen előbb a Lista konstruktora hívódik és létrehozza a nevek tömböt,
// majd ez a Tankor konstruktor.
```

A konstruktorok hívási sorrendje

```
// Az alábbi objektumdefiníció végrehajtásakor, azaz az objektum létrehozásakor a Lista
// osztály konstruktorát a Tankor osztály konstruktora rögtön meghívja, átadva neki az
// elemszam0 parametert. A Lista osztály konstruktorának lefutása után visszaugrik a
// program a Tankor osztály blokkjára és az abban levő utasítások is végrehajtódnak.

Tankor G1EG6(25, "G1EG6");      // objektum definiálása, létrehozása

Lista :: Lista( int elemszam0 )      // a Lista osztály konstruktora
{ elemszam= elemszam0;
  nevek = new nevtip[ elemszam0 ];    //dinamikusan foglaljuk a vektort
  for (int i = 0; i < elemszam0; i++ )  nevek[ i ][0]= '\0' ;
}
                                                          // paraméter átadás
Tankor :: Tankor( int elemszam0, char * tankorszam0 ): Lista( elemszam0 )
{ strcpy( tankorszam, tankorszam0 );
}
```

1

2

A destruktorok hívási sorrendje

// A destruktorok végrehajtási sorrendje fordított, előbb a megszüntetendő objektum
// osztályának konstruktora hajtódik végre, majd felfelé haladva a hierarchiában, sorra a
// szülő osztályok destruktorai.

//a *G1EG6* objektum a program végének elérésekor szűnik meg (a mostani példában).

```
Lista :: ~Lista( )   // a Lista osztály destruktora
{   // a dinamikusan foglalt nevek vektort felszabadítjuk
  if ( nevek ) delete[ ] nevek;
}
```

2

1

```
~Tankor( ) { };        // osztálydeklaráción belül automatikusan inline-ként
                       // definiált destruktor volt
```

Megjegyzések a példaprogramhoz

- *Ha a rövid tagfüggvények **definícióját az osztálydefiníción belül** adjuk meg, akkor az az inline kulcsszó megadása nélkül is **inline** függvény-definíció lesz.*

- *Ha a hosszabb tagfüggvények definícióját az osztály definíción kívül adjuk meg, akkor azt, hogy a függvény melyik osztályhoz tartozik, a*

  *tipus Osztalyneve **::** Tagfuggveny() { }*

  *formában meg kell adni. A **::** a **scope operátor**, segítségével többértelmű-ség esetén egyértelművé tehetjük, hogy melyik osztályhoz tartozik a definiálandó, vagy meghívandó, azonos névvel és paraméterszignatúrával több osztályban is előforduló tagfüggvény.*

- *A tagfüggvényeken belül a **saját** osztálybeli adattagokra és tagfüggvé-nyekre **scope operátor nélkül** hivatkozhatunk.*

- *Minden tagfüggvény paraméterei között szerepel egy **rejtett mutató**, mely mindig arra az objektumra mutat, amelyhez kapcsoltan a függvényt meg-hívtuk. Ez a mutató a függvényben közvetlenül is megadható **this** néven, és ily módon már a függvény definiálásakor is hivatkozhatunk olyan címre, amely csak az objektum keletkezése után válik majd konkréttá.*

A C++ programozási nyelv

Osztályhierarchia, öröklődés, virtuális függvények

- *Az osztály tagjainak elérési szintjei*
- *Függvény többalakúság az öröklési lánc mentén*
- *Virtuális tagfüggvények*
- *A többszörös öröklődés problémája*

Az osztály tagjainak elérési szintjei

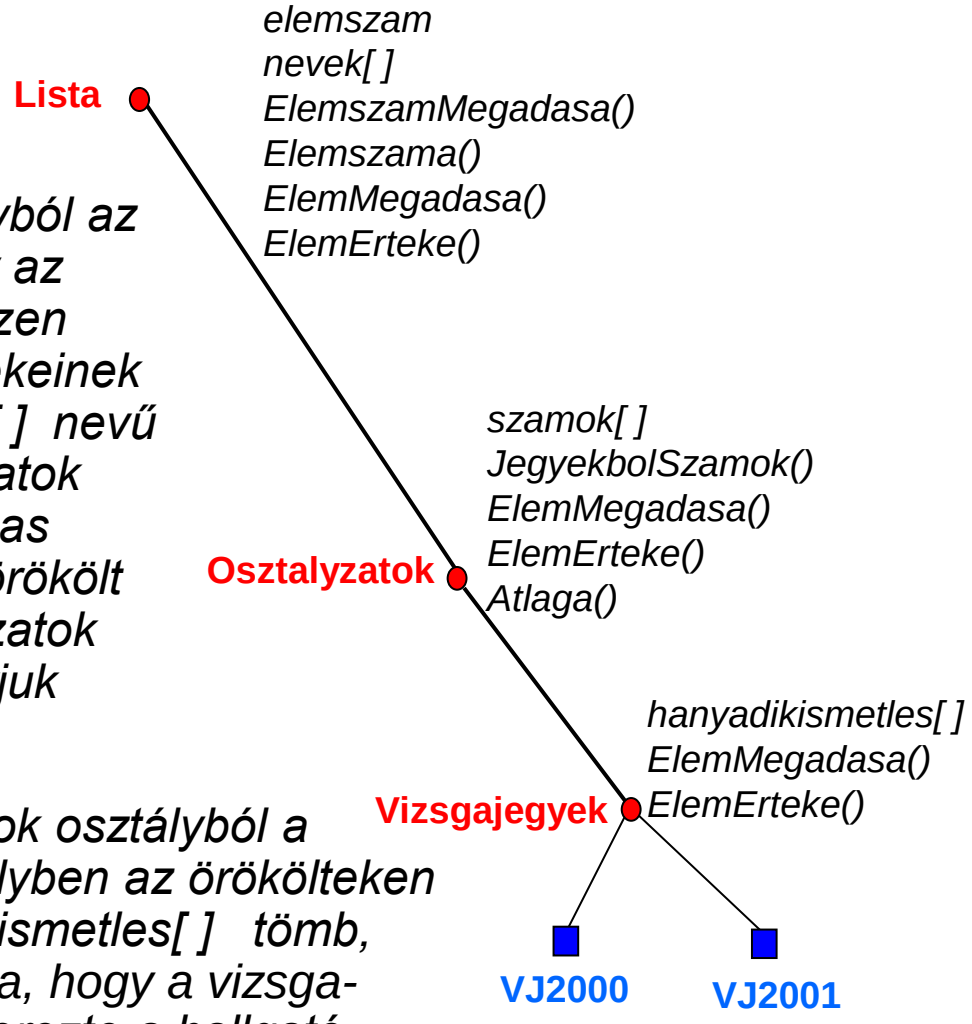
*Egy osztály adattagjai és tagfüggvényei alapértelmezetten **private** elérhetőségűek, ami azt jelenti, hogy csak az osztály tagfüggvényeiben hivatkozhatók közvetlenül. Ez a tulajdonság adattagokra és lokális, csak az osztályban szükséges függvényekre alkalmazva megvalósítja az adat- és működéselrejtés kívánalmát.*

*Az adattagok és a működés elérése az osztály későbbi gondnélküli módosíthatósága érdekében csak a külvilággal kapcsolatot tartó függvényeken keresztül célszerű. Így az osztály, illetve a belőle származtatott objektumok módosítása nem kell, hogy érintse az osztály felületét, interfészét. Természetesen, a kapcsolattartó függvények esetében a bárhonnani elérés kívánatos, hogy az objektumok egymással függvényhívásokkal, üzenetküldésekkel kommunikálhassanak. Ezt a célt a **public** elérhetőség megadásával érhetjük el.*

*Származtatott osztályokra tekintettel kívánatos, hogy az alaposztály privát tagjainak közvetlen elérhetetlenségét megszüntessük, és a származtatott osztályok tagfüggvényeiben is közvetlenül használhassuk azokat. Az elérhetőség körének ilyen tágítását a **protected** elérhetőség megadásával végezzük.*

Függvény többalakúság az öröklési lánc mentén

osztály
egyed



Származtassuk a Lista *osztályból az* **Osztalyzatok** *alosztályt, mely az öröklött tagokon túl rendelkezzen egy, az osztályzatok számértékeinek tárolására alkalmas szamok[ ] nevű tömbbel, valamint az osztályzatok átlagának számítására alkalmas Atlaga( ) tagfüggvénnyel. Az örökölt nevek[ ] vektorban az osztályzatok* jeles, jó, *stb. minősítését akarjuk tárolni.*

Származtassuk az Osztalyzatok osztályból a **Vizsgajegyek** *alosztályt, amelyben az örökölteken túl legyen még egy hanyadikismetles[ ] tömb, melynek egy eleme azt mutatja, hogy a vizsga-jegyet hányadik ismétlésre szerezte a hallgató.*

Az öröklési láncban megfigyelhető, hogy mind a Lista, mind az Osztalyzatok, mind a Vizsgajegyek osztály rendelkezik az ElemMegadasa() és az ElemErteke() függvényekkel. Természetesen ***azt szeretnénk, ha*** *az Osztalyzatok osztályban* ***ezek a hasonló szerepű függvények másképpen működnének****, azaz együtt kezelnék egy osztályzat nevét (jeles) és számalakját (5). Hasonlóan, a Vizsgajegyek osztály függvényeitől elvárjuk, hogy az osztályzat neve és számalakja mellett az ismétlés számát is kezeljék. Vizsgaismétlés nélkül megszerzett jegy esetében ez nulla.*

A megoldás kézenfekvő: eltérő paraméterszignatúrájú, gyakorlatilag az örököltek mellett új függvényeket adunk meg:

```
class Lista
{ …
  public: ...

        void ElemMegadasa(int i, char* nevbe) { strcpy(nevek[ i ], nevbe);}
        void ElemErteke(int i, char* nevki ) { strcpy(nevki, nevek[ i ] );}

};

            //folytatódik
```

```
class Osztalyzatok : public Lista     // mindent örököl a Lista osztálytól
{
  protected: int* szamok;       // ebbe foglalja majd a konstruktor a vektort
  public:        Osztalyzatok (int elemszam0= 0);      // konstruktor deklarációja
                ~Osztalyzatok( );                          // destruktor prototípusa
            void ElemMegadasa(int i, int szambe);   // definíciót lásd külön
            void ElemErteke(int i, int& szamki, char* nevki )
                                    { szamki= szamok[ i ]; strcpy(nevki, nevek[ i ] );}
        double  Atlaga( );   // definíciót lásd külön
};


class Vizsgajegyek : public Osztalyzatok    // mindent örököl az Osztalyzatok-tól
{
  private: int* hanyadikismetles;   // 0,1, 2, 3
  public:        Vizsgajegyek(int elemszam0= 0 );   // konstruktor deklarációja
                ~Vizsgajegyek( );
            void ElemMegadasa(int i, int jegy, int hanyadik)
             {hanyadikismetles[ i ]= hanyadik; Osztalyzatok:: ElemMegadasa(i, jegy);};
            void ElemErteke(int i, int& szamki, char* nevki, int& hanyadik)
                                    {Osztalyzatok:: ElemErteke(i, szamki, nevki );
                                                hanyadik= hanyadikismetles[ i ]; }
};                  //folytatódik
```

```
Vizsgajegyek VJ2000(200), VJ2001(250);  // Vizsgajegyek típusú objektumok

void main( )

{ …  // Itt van az objektumokat használó programrész

}

// függvénydefiníciók:

…
Osztalyzatok :: Osztalyzatok (int elemszam0) : Lista(elemszam0)
{ szamok = new int[ elemszam0 ];
  for (int i =0; i < elemszam0; i++) {szamok[ i ]= 5; strcpy(nevek[ i ], "jeles"); }
}

Osztalyzatok :: ~Osztalyzatok( ) { delete[ ] szamok; }


void Osztalyzatok :: ElemMegadasa(int i, int szambe)
{ char ss[5][10]= {"elégtelen","elégséges","közepes","jó","jeles"};
  szamok[ i ]= szambe;
  strcpy(nevek[ i ], ss[szambe -1]);
}
```

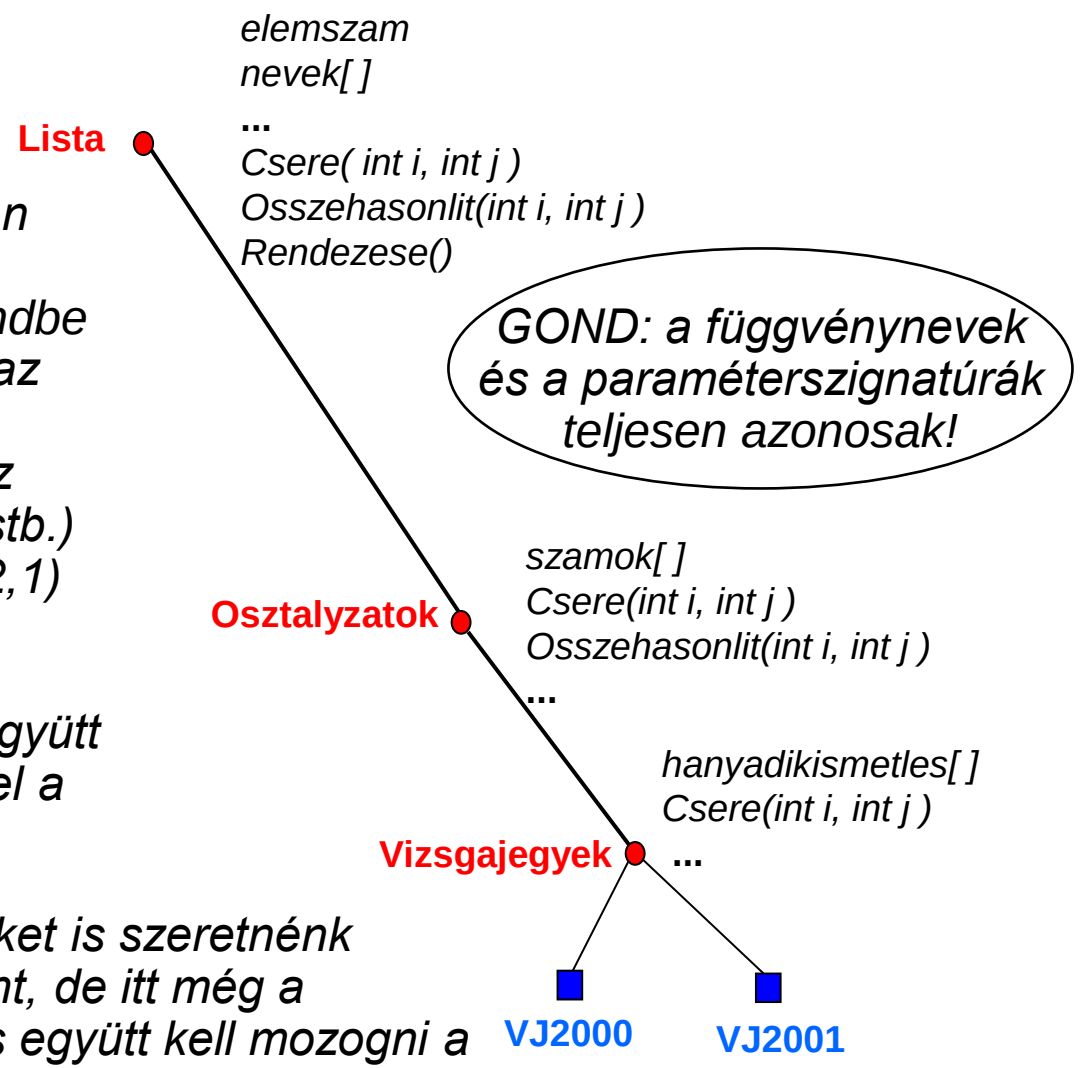
```
double Osztalyzatok :: Atlaga( )
{  double szum= 0.0;
   for (int i =0; i < elemszam; i++) szum += szamok[ i ];
   return szum / elemszam;
}

Vizsgajegyek :: Vizsgajegyek (int elemszam0 ) : Osztalyzatok(elemszam0)
{ hanyadikismetles = new int[ elemszam0 ];
  for ( int i =0; i < elemszam0; i++) hanyadikismetles[ i ]= 0;
}


Vizsgajegyek :: ~ Vizsgajegyek ( ) { delete[ ] hanyadikismetles; }
```

Megjegyzés: *a* Vizsgajegyek VJ2000(200); *objektumdefiníció 200 fő számára hoz létre egy vizsgajegyek tárolására alkalmas objektumot. A Vizsgajegyek osztály konstruktora átadja a 200 elemszámot az Osztalyzatok osztaly konstruktorának, amely feljebb adja a Lista konstruktorának. Lista konstruktora lefoglalja a férőhelyet a* nevek *számára, inicializálja, majd átadja a működést az Osztalyzatok konstruktorának. Az lefoglalja a* szamok *vektort, inicializálja azt és beirja a jeleseket a* nevek *vektorba. Utána a Vizsgajegyek konstruktora következik a* hanyadikismetles *vektor lefoglalásával és nullákkal való feltöltésével.*

Virtuális tagfüggvények

osztály
egyed

Jó lenne, ha a Lista osztályban lenne egy rendező függvény, amellyel a neveket abc sorrendbe rendezhetnénk. Ez öröklődik az Osztalyzatok osztályra, de ott jobban szeretnénk, ha nem az osztályzatok nevei (jeles, jó, stb.) hanem a számértékei (5,4,3,2,1) szerint rendezné az elemeket növekvő sorrendbe. Természetesen a neveknek együtt kell mozogni a számértékekkel a helycserék során.



Természetesen a vizsgajegyeket is szeretnénk rendezni a számértékek szerint, de itt még a vizsgaismétlések számának is együtt kell mozogni a nevekkel és a számokkal a helycserék során.

Honnan fogja tudni a Rendezese( ) függvény**, mely változatlan alakban örök-lődik az Osztalyzatok és a Vizsgajegyek osztályokra, hogy a három eltérő belső működésű, de neve és paraméterszignatúrája alapján nem megkülönböztethető **Csere(int i, int j) függvény közül mikor melyiket kell használnia?

***Válasz**: abból fogja tudni, hogy éppen **melyik osztály egyedéhez csatoltan hívtuk meg**. Ez viszont csak a program futása közben derül ki, ezért a függvények ilyen fajta egymáshoz rendelését futásidejű, **késői, dinamikus hozzárendelésnek** (late binding) nevezik.*

*Az azonos nevű és paraméterszignatúrájú függvények könnyen megkülönböztet-hetők az által, hogy melyik osztályhoz (vagy osztályhoz nem) tartoznak, ha használjuk a **::** scope operátort, de ezzel a módszerrel a program írásakor (fordítási időben) hozzákötnénk valamelyik osztály Csere(int i, int j) függvényét a Rendezese( ) függvény kódjához. Ez **korai, statikus hozzárendelés** lenne.*

Az azonos függvények közüli futásközbeni választás előkészítéséhez a függvény legelső előfordulását az öröklési láncban **virtual** *kulcsszóvel kell ellátni.*

Mitől virtual (látszólagos, nem igazi) a függvény? Képzeljük el, hogy a Lista osztály neveit nem akarjuk rendezni, csak a belőle származtatott Tankor, Osztályzatok és Vizsgajegyek vektorainak elemeit. Ekkor is alkalmazhatunk definíció, tényleges megadás nélküli tisztán virtual Csere(int i, int j) függvényt a Lista osztályon belüli Rendezese() függvény megírásához. Természetesen egy ilyen tisztán virtuális függvénnyel bíró osztálynak nem lehet egyede.

A megoldás C++ nyelven:

```
...

class Lista
{ protected:        int elemszam;
                    nevtip * nevek;
            virtual void Csere(int i, int j )          // neveket cserél
                        { nevtip buff ; strcpy(buff, nevek[ i ]);
                          strcpy(nevek[ i ], nevek[ j ]); strcpy(nevek[ j ], buff );
                        }
              virtual int Osszehasonlit(int i, int j )    // neveket hasonlít össze
                        { return strcmp(nevek[ i ], nevek[ j ]);
                        }
  public:           …
                    void Rendezese();    // a definiciót, amely használja a
                                         // Csere( ) és az
                                         // Osszehasonlit( ) függvényeket, lásd külön
};

                    // folytatódik
```

```
...
class Osztalyzatok : public Lista
{ protected: int * szamok;
                void Csere(int i, int j )
                        {int buff ; buff= szamok[ i ]; szamok[ i ]= szamok[ j ];
                         szamok[ j ]= buff ; nevtip buf ;
                         strcpy(buf, nevek[ i ]); strcpy(nevek[ i ], nevek[ j ]);
                         strcpy(nevek[ j ], buf);
                        }
                  int Osszehasonlit(int i, int j )
                        { if szamok[ i ]<szamok[ j ]) return -1;
                          else if (szamok[ i ]==szamok[ j ]) return 0;
                                else return 1;
                        }
  ...
 };


                  // folytatódik
```

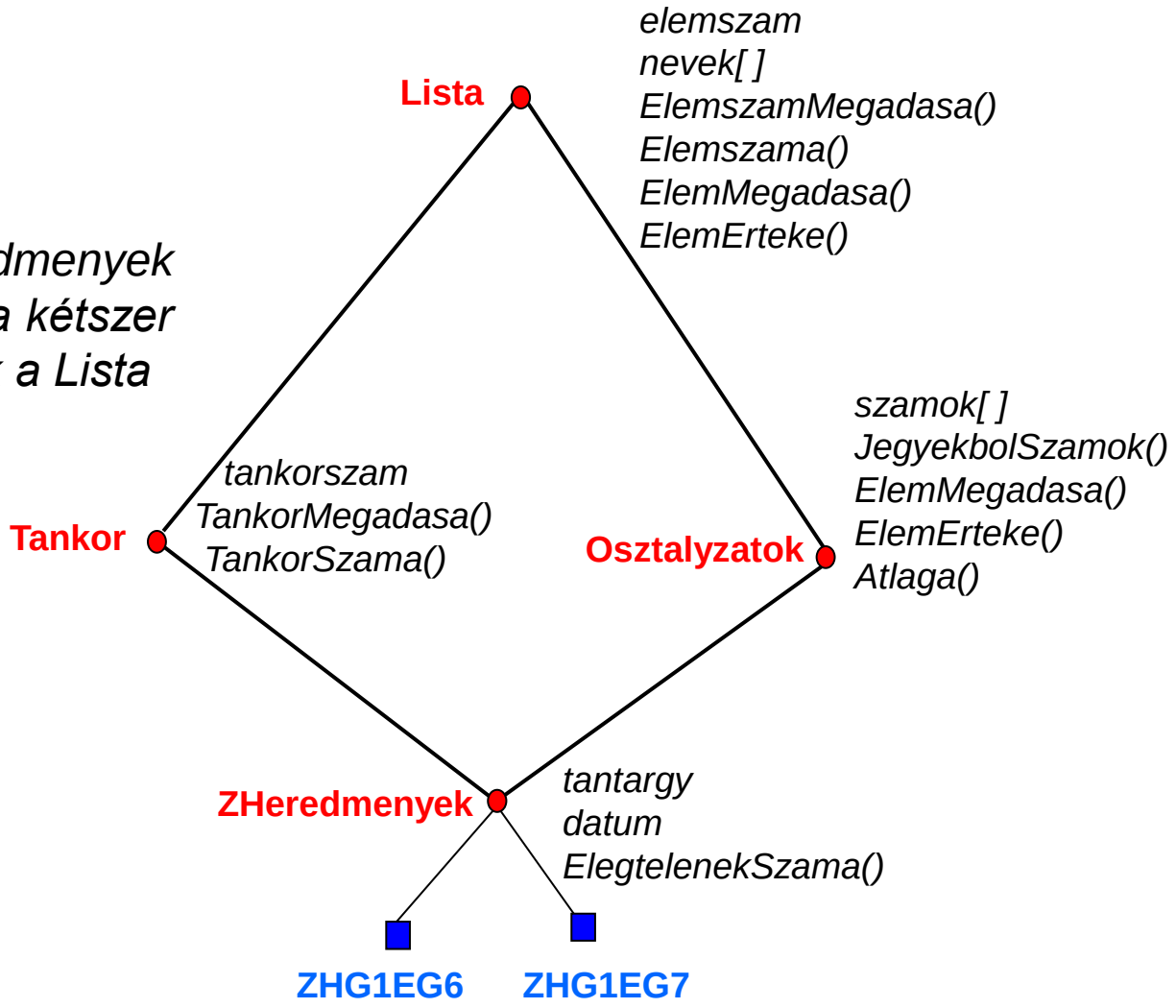
```
...

class Vizsgajegyek : public Osztalyzatok
{ private:      int * hanyadikismetles;
                void Csere(int i, int j )
                        { Osztalyzatok :: Csere( i, j );
                          int b= hanyadikismetles[ i ];
                          hanyadikismetles[ i ]=  hanyadikismetles[ j ];
                          hanyadikismetles[ j ]= b;
                        }
  ...
 };
Lista vezerek(7);                    // objektumváltozók definiálása
Osztalyzatok osztalyzatok(25);
Vizsgajegyek VJ2000(200);
void main( )
{ …
  vezerek.Rendezese(); osztalyzatok.Rendezese(); VJ2000.Rendezese();
...

}               // folytatódik
```

```
...

// A teljesen definiált, de virtuális függvény-függő rendező függvény:
void Lista :: Rendezese()
{ if (elemszam > 1)  // ha van mit rendezni
  for (int i = 0; i< elemszam -1; i++)
     for (int j = i+1; j < elemszam; j++)
     if (Osszehasonlit( i, j ) > 0) Csere( i , j );  // futásidőben helyettesítődnek  !!
  puts("\n Rendezés kész.");
}
```

A többszörös öröklődés problémája

osztály
egyed

A ZHeredmenyek osztályba kétszer öröklődik a Lista osztály



A többszörös öröklődés egyik fajtájánál ***a Lista osztály adattagjai megkettőződnek*** *a ZHeredmenyek osztályban. Ez a* nevek[ ] *tömbök esetén kívánatos, hiszen ezekben a Tankor osztályból a tankör névsorát örökli, az Osztalyzatok osztályból pedig a tankör hallgatóinak ZH-osztályzatait, az* elemszam *esetén viszont zavaró, mert az értékek ugyanazok, így felesleges a duplázás. Jegyezzünk meg annyit, hogy létezik virtuális örökítés is, amellyel az adattagok megduplázása elkerülhető, most viszont a duplázás tűnik kedvezőbb esetnek. Látni fogjuk, hogy a két* elemszam *adattag tartalmának eltérése elkerülhető.*

```
...
class ZHeredmenyek : public Tankor, public Osztalyzatok        // két őse is van
{ private: char tantargy[30];          // amiből a zárthelyit írták
           char datum[12];             // amikor a zárthelyit írták

  public: ZHeredmenyek(int elemszam0= 0,
                       char* tantargy0= "Számítástechnika",
                       char* datum0= "2002.03.28");   // konstruktor deklarációja
        ~ZHeredmenyek( );                             // destruktor deklarációja
      int ElegtelenekSzama( );     // egy tagfüggvény deklaráció

};

            //folytatódik
```

```
...
ZHeredmenyek  ZHG1EG6(28), ZHG1EG7(33);

void main( )
{…
  ZHG1EG6.Osztalyzatok :: Rendezese( );     // Az azonos nevű függvények közül
  ZHG1EG6.Tankor :: Rendezese( );           // a scope operátorral választhatunk
  printf("Átlag= %5.1lf ", ZHG1EG6.Atlaga( ) );          // Egyértelmű eset
  ...

}

// Nincs gond az elemszam duplázódással, mert mindkét ágon ugyanazt az inicializáló értéket
// kapja meg a konstruktortól:

ZHeredmenyek :: Zheredmenyek(int elemszam0, char* tantargy0, char* datum0)
                                        : Osztalyzatok(elemszam0), Tankor(elemszam0)
{ strcpy(tantargy, tantargy0 ); strcpy(datum, datum0);

}

ZHeredmenyek :: ~ZHeredmenyek( ) { }    //destruktor függvény definíciója

int ZHeredmenyek :: ElegtelenekSzama( )
{ int k =0; for (int i =0; i <Osztalyzatok::elemszam; i++)
            if (szamok[ i ]==1) k++; return k; }
```