1. Heterogén kollekció

Heterogén kollekciónak azt nevezzük, amikor azonos alaposztályból (példánkban a `Alakzat`) származtatott (`Vonal`, `Teglalap`, `Négyzet`, `Poligon`) objektumokat, alaposztályukra mutató pointereket (`Alakzat*`) tartalmazó tömbben tárolunk.

1.1. Rajztábla osztály

Készítsünk egy `Rajztabla` osztályt, ami minden `Alakzat` osztályból származtatott objektum tárolására alkalmas. Az adatokat pointeren keresztül tároljuk, különben fellép a szeletelődés (lásd előző anyag).

```
class Rajztabla {
   Alakzat* tab[100];
   int      db;
   Rajztabla( const Rajztabla& );
   Rajztabla& operator=( const Rajztabla& other );
public:
   Rajztabla() :db(0) { };
   void Felrak(Alakzat *a) {tab[db++] = a; }
   void Rajzol() {
      for (int i = 0; i < db; i++) tab[i]->Rajzol();
   }
   void Torol() {
       for (int i = 0; i < db; i++) delete tab[i];
   }
   ~Rajztabla() { Torol(); }
};
```

1.2. Poligon osztály

A poligon több törésponttal leírható kapcsolódó szakaszokat tartalmazó mértani alakzat. Feltételezzük, hogy a poligon nem záródik, ezért nincs területe.

```
class Poligon:public Alakzat {
   Point *pt;
   int   db;
public:
   Poligon(int n = 0, const Point& o= Point::defaultPoint, Szin sz = WHITE)
       :Alakzat( o, sz), db(n), pt( new Point[n] ) {};
   void Rajzol() const;
   void SetPoint( const Point& p, int n );
   const Point GetPoint( int n) const;
   int nPoint() const { return db; }
   ~Poligon() { delete[] pt; }
};
```

A poligon töréspontjainak száma előre nem ismert, ezért dinamikusan lefoglalt vektorban tároljuk, amit a végén a destruktor szabadít fel.

1.3. Alaposztály destruktora legyen virtuális

A `Poligon` osztály megadásánál láttuk, hogy meg kell adni a destruktort is. A `Rajztábla` osztály `Alakzat*` pointereket tartalmaz, és a `Torol()` függvény az alakzatokra mutató objektumokat szabadítja fel. Ha az alaposztály destruktora nem virtuális, akkor nem a származtatott osztály, hanem csak az alaposztály destruktora fog meghívódni. Minden alaposztálynak szánt osztály destrukturának ezért virtuálisnak kell lenni.

```
class Alakzat {
   .....
public:   .....
   virtual ~Alakzat() {}
};
```

Az alaposztály destruktora azért nem tisztán virtuális, hogy ott ahol a destruktornak nincs teendője (pl. `Vonal`), azt nem kelljen megvalósítani. Felmerül a kérdés, hogy a destruktor miért nem mindig virtuális.

```
class Point {
public:
   short x, y;
   ......
};
```

Ha a `Point` osztályban virtuális a destruktor, akkor az osztályba bekerül egy virtuális függvénytáblára mutató pointer is. Ez egyrészt méretnövekedést okoz, másrészt nem tudunk C-vel megegyező struktúra adattípust létrehozni. Kevert nyelvű programozásnál, ha C++-ból C vagy Fortran függvényeket hívunk, akkor az adatok átadásához szükség van bitre megegyező adatszerkezet definiálására.

2. Látogató szoftverminta

Szeretnénk a rajztábla objektumait típusonként leszámolni, vagy a területüket kiszámítani. Modellezési kérdést vet fel, hogy miért kell a nemzáródó alakzatokhoz is `virtual double Terulet()` tagfüggvényt létrehozni.

Objektumok leszámolása dinamikus típuskonverzióval:

```
void Rajztabla::sz( int& v, int& t, int& n, int& p ) {
    v = t = n = p = 0;
    for( int i=0; i<db; i++) {
        if( Vonal* pv = dynamic_cast<Vonal*>( tab[i] ) ) v++;
        else if( Negyzet* pn = dynamic_cast<Negyzet*>( tab[i] ) ) n++;
        else if( Teglalap* pt = dynamic_cast<Teglalap*>( tab[i] ) ) t++;
        else if( Poligon* pp = dynamic_cast<Poligon*>( tab[i] ) ) p++;
    }
}
```

Kérdések:

- Mit kell csinálni egy új származtatott típus bevezetése esetén?
- Mi történt volna, ha a téglalapokat előbb számoljuk meg, mint a négyzeteket?

2.1. Látogató interface

Hozzuk létre az `AlakzatVisitor` absztrakt osztályt. Minden látogatónak ez lesz a felülete;

```
class Vonal;
class Teglalap;
class Negyzet;
class Poligon;

class AlakzatVisitor {
public:
   virtual void Visit( Vonal& )= 0;
   virtual void Visit( Teglalap& )= 0;
   virtual void Visit( Negyzet& )= 0;
   virtual void Visit( Poligon& )= 0;
};
```

2.2. Látogatott osztály kibővítése

Az `AlakzatVisitor`-ból származtatott osztálynak ismerni kell az alakzat osztály tagfüggvényeit. A származtatott alakzat osztály fogad `Accept()` nevű tagfüggvényének viszont látni kell az `AlakzatVisitor` osztály felületét. Ez körkörös függőséget jelent, ami nem feloldhatatlan probléma, de valamelyik osztály változtatása esetén a *látogató* és a *látogatott* osztályt is le kell fordítani.

```
class Alakzat {
public:
    .......
    virtual void Accept(AlakzatVisitor&)=0;
};

class Vonal : public Alakzat {
   Point kp, vp;                          // vonal végpontja
public:     .....
   void Accept(AlakzatVisitor& v) { v.Visit( *this ); }
 };

class Teglalap:public Alakzat {
   Point lt, rb;                          // ellentétes csúcspont
public:     .....
   void Accept(AlakzatVisitor& v) { v.Visit( *this ); }
};

class Negyzet : private Teglalap {
public:     .....
   void Accept(AlakzatVisitor& v) { v.Visit( *this ); }
};

class Poligon:public Alakzat {
   Point *pt; int    db;
public:     .....
    void Accept(AlakzatVisitor& v) { v.Visit( *this ); }
};
```

Minden osztályba vezessük be a fogadó `Accept` tagfüggvényt, amely `*this` paraméterrel meghívja a látogató osztály látogat `Visit` tagfüggvényét.

Készítsük el az alakzatokat típusonként megszámláló `SzamlaloVisitor` osztályunkat.

```
class SzamlaloVisitor : public AlakzatVisitor {
    int dbVonal;
    int dbTeglalap;
    int dbNegyzet;
    int dbPoligon;
public:
    SzamlaloVisitor(): dbVonal(0),dbTeglalap(0),dbNegyzet(0),dbPoligon(0)
      {}
    void Visit( Vonal& l) { dbVonal++; }
    void Visit( Teglalap& t ) { dbTeglalap++; }
    void Visit( Negyzet& n ) {  dbNegyzet++; }
    void Visit( Poligon& p) { dbPoligon++; }

    int VonalSzam() const { return dbVonal; }
    int TeglalapSzam() const { return dbTeglalap; }
    int NegyzetSzam() const { return dbNegyzet; }
    int PoligonSzam() const { return dbPoligon; }
};
```

2.3. Látogató hívása

```
class Rajztabla {
   Alakzat* tab[100];
   int      db;
public:
   .....
   void Szamlalo(SzamlaloVisitor& tv) {
       for (int i = 0; i < db; i++) tab[i]->Accept( tv );
   }
};
```

2.4. Látogató működése

1. A `Rajztabla` meghívja az `Alakzat*` pointeren keresztül az `Accept()` függvényt a `SzamlaloVisitor&` referenciával (`tab[i]->Accept( tv )`).
2. Az `Alakzat::Accept()` virtuális függvény, ezért a virtuális függvénytábla segítségével a származtatott osztály, például a `void Polinom::Accept(AlakzatVisitor& v)` függvény hívódik meg.
3. A származtatott osztály *fogadó* függvénye visszahívja a látogató osztályban definiált *látogató* Visit() függvényt a `*this` paraméterrel. A `Visit()` függvénynév túlterhelt, minden származtatott alakzat paraméterrel meg lett valósítva. A `*this` biztosítja, hogy például a `Poligon` osztály `Accept()` függvénye a megfelelő `Visit( Poligon&)` függvényt hívja meg. Ez itt a kulcsmomentum. Ez valósítja meg a statikus típus szerinti függvényhívást (static type switching).

2.5. Előnyök

- Fejleszthetőség szempontból előny, hogy a látogatóban megadott `Visit()` tagfüggvények egy helyen találhatók, és nem a származtatott objektumokban, amelyek egy nagyobb programban különböző fájlban vannak implementálva.
- A `Visit()` függvényeknek a működése eltérő lehet.
- Nem kell `dynamic_cast()`-ot használni. `dynamic_cast()` használata esetén egy újabb származtatott osztályt bevezetésénél, mindenhol, ahol `dynamic_cast()`-ot használtunk, bővíteni kell a programot. A fordító automatikusan nem szól, ha valahol ezt elfelejtjük megtenni.
- Újabb látogató osztály bevezetéséhez már nem kell az `Alakzat` típusból származtatott osztályokat módosítani.
- Újabb képességek bevezetéséhez nem kell minden osztályhoz újabb virtuális függvényt definiálni.