Feladat: Olvassunk be egy fájlból középpontosan szimmetrikus síkidomokat, majd egy másik fájlból beolvasott pontokról döntsük el, hogy mely síkidomok tartalmazzák azokat.

Lehetséges síkidomok: kör, négyzet, szabályos háromszög, szabályos hatszög és téglalap. A sokszögek esetében az egyik oldal párhuzamos a vízszintes tengellyel. A szövegfájlban először a síkidomok száma adott, majd az egyes alakzatok adatai: az alakzat típusa az angol név kezdőbetűjével (C, S, T, H, R), a középpont és a további adatok (sugár, vagy oldalhossz).

A pontok koordinátáit tartalmazza a másik szövegfájl számpárok formájában. Feltesszük, hogy mindkét szövegfájl eleget tesz a fenti megszorításoknak.

Két lehetséges bemeneti fájl:

```
6
C 0.0 0.0 1.0
C 1.0 0.0 2.0
S 0.0 0.0 3.0
T 0.0 0.0 1.0
H 0.0 1.0 1.0
R 1.0 2.0 2.0 1.0
```

```
0.0 0.0
1.0 1.0
2.0 0.0
0.0 2.0
```

Megoldás: Egy lehetséges megközelítés lenne, hogy létrehozzuk a szóban forgó síkidomok típusának unióját, és ebből készítünk egy dinamikus vektort. Minden alakzathoz megadhatjuk azt a műveletet, amely eldönti, hogy tartalmaz-e egy pontot. Ebben az esetben minden pont esetében végig kell menni a vektor elemein és a típuson alapuló esetszétválasztással (`switch`) a vektor minden elemére végrehajtani a megfelelő műveletet.

```
for ( int i = 0; i < n; i++ )
{
    switch ( v[i].type )
    {
        case Circle : ...
                      break;
        case Square : ...
                      break;
        ...
    }
}
```

Öröklődés

Szerencsére C++-ban van erre egy jobb lehetőség, az *öröklődés* használata. Ennek segítségével egy létező osztályból származtathatunk egy új osztályt úgy, hogy az eredeti osztályt kiegészítjük, esetleg bizonyos műveleteit módosítjuk.

A kiegészítés lehet lehet

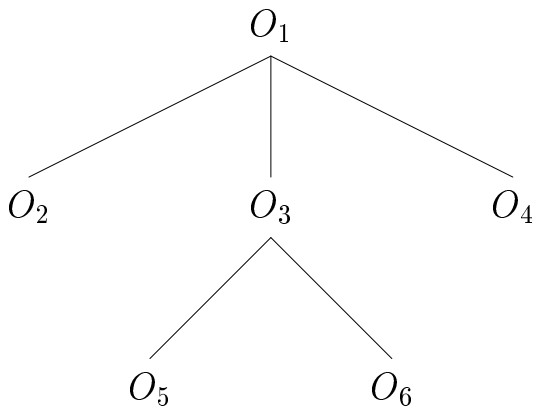
- reprezentációs jellegű, amikor is új adattagokat (változókat) vezetünk be, illetve
- műveleti jellegű, amikor új műveletekkel egészítjük ki az eredeti osztályt.

Lehetőség van *absztrakt osztály* létrehozására is, amelyben

- a reprezentáció nem teljes, és/vagy
- műveletek nincsenek megvalósítva.

Absztrakt osztályhoz nem tartozhat objektum, de pointer típusa lehet. Absztrakt osztályból is származtathatunk osztályt. Ha ennek során a hiányosságok pótlásra kerülnek, akkor ehhez az osztályhoz már tartozhat objektum.

A származtatatott osztályból újabb osztály származtathatunk. Ez az új osztály leszármazottja lesz mindkét osztálynak. Így öröklődési hierarchia hozható létre, amely fával szemléltethető.



Ebben egy osztály, minden a gyökérből hozzá vezető úton található osztály leszármazottja. Fordítva, minden ilyen osztály ezen osztály *őse*.

Származtatás jelölése C++-ban:

```
class Alap
{
...
}

class Uj : public Alap
{
...
}
```

Újdonságok:

- elérhetőség
- polimorfizmus
- dinamikus összekapcsolás

Osztályok komponenseinek (adat, művelet) elérhetősége:

public: minden objektum hozzáférhet a komponenshez

private: csak az adott osztály objektumai férhetnek hozzá a komponenshez

protected: az adott osztály és az abból származtatott osztályok objektumai férhetnek hozzá a komponenshez

Ezért *private* minősítőt a továbbiakban csak akkor használjunk, ha biztosak vagyunk benne, hogy az osztályból nem akarunk újabb osztályt származtatni.

Polimorfizmus:

Egy ős osztályba tartozó objektum felvehet leszármazott osztálybeli értéket. Ezért minden változónak két típusa van:

- statikus: a deklaráció során kapja,
- dinamikus: futás során, ha ős osztályba tartozó objektumnak egy leszármazottbeli értéket adunk.

Legyen O_1 leszármazottja O_2, és $x \in O_1$, $y \in O_2$. $x = y$ értékadás után x statikus típusa O_1, dinamikus típusa O_2.

Dinamikus összekapcsolás:

A dinamikus típusnak megfelelő művelet, adattag kiválasztás a programban.

A fenti példában, ha f műveletet átdefiniáltuk a származtatás során, akkor az értékadás után az O_2-ben átdefiniált műveletet jelenti $x.f()$.

Művelet átdefiniálhatóságának engedélyezése:

Erre szolgál C++-ban a *virtual* kulcsszó. Az így megjelölt függvények definiálhatóak újra a származtatás során. Itt csak a viselkedés változhat, a külső forma (paraméterezés) nem.

```
class C
{
    ...
public:
    virtual int f(...);
    ...
}

class D : public C
{
    ...
    int f(...);
    ...
}
```

Absztrakt osztályok meg nem valósított műveletei:

Mint mondtuk, absztrakt osztályban lehet olyan művelet, amelynek még nem ismert a megvalósítása. Ezt meg lehetne oldani például úgy, hogy a művelet törzse üres lenne. Ez nyilvánvalóan nem felel meg az előbbieknek. A megvalósítás hiányának jelölésére egy speciális lehetőség van:

```
class A
{
    ...
public:
    virtual f(...) = 0;
    ...
}
```

Most már egyszerűbben is megoldható a feladat. Tegyük fel, hogy adott a síkbeli pontokat megvalósító osztály, `Point`. Ezt felhasználva készítünk egy absztrakt osztályt, amely a síkidomoknak felel meg, `Shape`. Ebben az érdekes művelet, `Contains`, amely eldönti, hogy egy síkbeli pontot tartalmaz-e a síkidom. Ez itt még nem valósítható meg. A reprezentáció is csak részelges lehet, csak a középpontot reprezentálhatjuk még.

```
class Shape
{
public:
    virtual ~Shape(void);
    virtual bool Contains(const Point &p) const = 0;
protected:
    Shape(const Point &cp);
    Point center;
};

Shape::Shape(const Point &cp)
{
    center = cp;
}
```

A Shape osztályból származtatjuk a többi osztályt:

```
class Circle : public Shape
{
public:
    Circle(const Point &cp, double r);
    virtual ~Circle(void);
    bool Contains(const Point &p) const;
protected:
    double radius;
};
class Square : public Shape
{
public:
    Square(const Point &cp, double s);
    virtual ~Square(void);
    bool Contains(const Point &p) const;
protected:
    double side;
};
```

A műveletek megvalósításai értelemszerűen:

```
Circle::Circle(const Point &cp, double r) : Shape(cp)
{
    radius = r;
}
bool Circle::Contains(const Point &p) const
{
    Point   tmp = p - center;
    return(tmp.GetX()*tmp.GetX()+tmp.GetY()*tmp.GetY()<=radius*radius);
}

Square::Square(const Point &cp, double s) : Shape(cp)
{
    side = s;
}
bool Square::Contains(const Point &p) const
{
    Point   tmp = p - center;
    return(2.0*fabs(tmp.GetX())<=side && 2.0*fabs(tmp.GetY())<=side);
}
```

Tegyük fel, hogy a feladatban szereplő többi síkidomhoz is elkészítettük a megfelelő osztályt, és van egy olyan műveletünk is (`read`), amely beolvas egy síkidomot egy szövegfájlból. Ennek felhasználásával elkészíthetjük a megoldást.

```
void read(ifstream &inp, Shape *&s)

int main( void )
{
    char     fname[81];
    int      nr_shapes;
    Shape    **s;
    ifstream inp;
    Point    p;
    int      i;
```

```
cout << "Name of the shape defining file: ";
cin >> fname;
inp.open(fname);
inp >> nr_shapes;
s = new Shape *[nr_shapes];
for ( i = 0; i < nr_shapes; i++ )
{
    read(inp, s[i]);
}
inp.close();
```

```
    cout << "Name of the points file: ";
    cin >> fname;
    inp.open(fname);
    inp >> p;
    while ( !inp.eof() )
    {
        cout << "Pont: " << p << endl;
        for ( i = 0; i < nr_shapes; i++ )
        {
            if ( s[i]->Contains(p) )
            {
                cout << i + 1 <<". tartalmazza" << endl;
            }
        }
        inp >> p;
    }
    inp.close();
    return(0);
}
```

```
void read(ifstream &inp, Shape *&s)
{
    char    c;
    Point   p;
    double  dat, dat2;
    inp >> c;
    inp >> p;
    switch ( c )
    {
        case 'C':
            inp >> dat;
            s = new Circle(p, dat);
            break;
        case 'S':
            inp >> dat;
            s = new Square(p, dat);
            break;
        ...
    }
}
```

```
#ifndef __POINT_H
#define __POINT_H
#include <fstream.h>

class Point
{
public:
    Point( void );
    Point( double x, double y );
    virtual ~Point( void );
    Point( const Point &p );
    Point  &operator=( const Point &p );
    void   Set( double x, double y )    { coord_x = x; coord_y = y; }
    void   SetX( double x )    { coord_x = x; }
    void   SetY( double y )    { coord_y = y; }
    double GetX() const    { return coord_x; }
    double GetY() const    { return coord_y; }
    Point  operator+( const Point &p ) const;
    Point  operator-( const Point &p ) const;
    void   Rotate( double angle );
```

```
    Point  operator*( const double factor ) const;
    friend Point    operator*( const double factor, const Point &p );
    friend ostream  &operator<<( ostream &s, const Point &p );
    friend istream  &operator>>( istream &s, Point &p );
    friend ofstream &operator<<( ofstream &f, const Point &p );
    friend ifstream &operator>>( ifstream &f, Point &p );
protected:
    double    coord_x;
    double    coord_y;
};

#endif
```