Ejercicio 1

```
#include <iostream>
int main(int argc, char** argv) {
    // programa A
    Foo<int> ifoo(1);             // t = 1 n = 1
    ifoo.mar(2);                  // t = 3 n = 2
    ifoo.mar(3);                  // t = 6 n = 3
    std::cout<<ifoo.sil()<<std::endl;   // 2

    Foo<double> ffoo(0.1);        // t = 0.1 n = 1
    ffoo.mar(0.2);                // t = 0.3 n = 2
    ffoo.mar(0.3);                // t = 0.6 n = 3
    std::cout<<ffoo.sil()<<std::endl;   // 0.2

    Foo<Bar> bfoo(Bar(1,10));     // t = 1/10 n = 1
    bfoo.mar(Bar(2,10));          // t = 30/100 n = 2
    bfoo.mar(Bar(3,10));          // t = 600/1000 n = 3
    std::cout<<bfoo.sil()<<std::endl;   // t = 600/3000
/
    // programa B

    Foo* foos[5];
    foos[0]=new Foo();                  // 1
    for (int i=1;i<5;++i){
        foos[i]=new Bar(foos[i-1]);
    }
    for (Foo* f : foos){
        std::cout<<f->value()<<" ";     // 1 2 3 4 5
        std::cout<<std::endl;
    }
    for (Foo* f : foos){
        delete f;
    }
}
```

Ejercicio 2

Apartado a)
La herencia es el mecanismo que permite a una clase reutilizar datos y código definidos en otra. El comportamiento común se recoge en la clase base, y múltiples clases derivadas pueden heredarlo.

Apartado b)
Un lenguaje que presenta herencia es por ejemplo C++

Apartado c)
Tres consecuencia de la herencia:
- Reutilizar directamente atributos y métodos definidos para la clase padre
- Definir atributos y/o métodos nuevos
- Redefinir métodos existentes en el padre

Apartado d)
```
class Padre {
protected:
    int atr1, atr2;
public:
    Padre(int atr1, int atr2) : atr1(atr1), atr2(atr2) {}

    int metodo1() const {
        return atr1 + atr2;
    }

    virtual int metodo2() const {
        return atr1 * atr2;
    }
};

class Hija : public Padre {
    int atr3;
public:
    Hija(int atr1, int atr2, int atr3) : Padre(atr1, atr2), atr3(atr3)
{}

    int metodo2() const override {
        return atr1 * atr2 * atr3;
    }

    int metodo3() const {
        return atr1 - atr3;
    }
};
```

Apartado e)

La primera de las consecuencias nombradas era la de reutilizar por parte de la clase hija atributos y métodos definidos en la clase padre, esto en el ejemplo se puede observar en el caso de los atributos con atr1 y atr2, y en el caso de los métodos en el metodo1. El caso de los atributos es más claro ya que se emplea en el constructor de la clase hija el propio constructor de la clase padre. En cuanto a la segunda característica, en la clase hija se define un nuevo atributo (atr3) y un nuevo método (metodo3). La última de las consecuencias se trata de redefinir métodos declarados en la clase padre, como se puede observar, el metodo2 es virtual en la clase padre y por lo tanto se puede redefinir en la clase hija mediante override.

Ejercicio 3

```
internalSplit :: [a] -> Int -> [a] -> [[a]]
internalSplit (first:rest) count firstPart
    | count == 0 = [firstPart, (first:rest)]
    | otherwise  = internalSplit rest (count - 1) (firstPart ++
[first])

split :: [a] -> [[a]]
split myList =
    let listLength = length myList
    in
        if listLength `mod` 2 == 0 then
            internalSplit myList (listLength `div` 2) []
        else
            internalSplit myList ((listLength `div` 2) + 1) []

minimo ::  (Eq a, Ord a) => [a] -> a
minimo [x] = x
minimo (x:xs)
    | x < minimo xs = x
    | otherwise = minimo xs

eliminar :: (Eq a) => a -> [a] -> [a]
eliminar x [] = []
eliminar x [y] = if x == y then []
                 else x : [y]
eliminar x xs = if x == head xs && numVeces x xs == 1 then eliminar x
(tail xs)
                 else head xs : eliminar x (tail xs)

numVeces :: (Eq a) => a -> [a] -> Int
numVeces x [] = 0
numVeces x [y] = if x == y then 1
                 else 0
numVeces x xs = if x == head xs then 1 + numVeces x (tail xs)
                 else numVeces x (tail xs)

ordenarLista :: (Eq a, Ord a) => [a] -> [a]
ordenarLista [] = []
ordenarLista [x] = [x]
ordenarLista xs
    | head xs == minimo xs = minimo xs : ordenarLista (tail xs)
    | otherwise = minimo xs : ordenarLista (eliminar (minimo xs) xs)
```

```
merge ::  (Ord a) => [[a]] -> [a]
merge [[]] = []
merge [[],[x]] = [x]
merge [[x],[]] = [x]
merge xss
    | a < b = head a : merge [tail a, b]
    | otherwise = head b : merge [a, tail b]
        where
            a = ordenarLista x
            b = ordenarLista y
            [x,y] = take 2 xss

msort :: Ord a => [a] -> [a]
msort [] = []
msort xs = merge (split xs)



antecesores :: ArbolSkywalker -> Nombre -> [Nombre]
antecesores Nulo n = []
antecesores (Nodo nom agno []) n
    | nom == n =[nom]
    | otherwise = []
antecesores (Nodo nom agno xs) n
    | nom == n = [nom]
    | buscaNombre (head xs) n = nom : antecesores (head xs) n
    | not (buscaNombre (head xs) n) = nom : antecesores (head (tail
xs)) n
    | otherwise = []

padre :: ArbolSkywalker -> Nombre -> [Nombre]
padre a n = drop (length l - 1) l
    where
        l = take (length lista - 1) lista
        lista = antecesores a n

arbol :: ArbolSkywalker
arbol = Nodo "Shmi" (Nom 72 "BBY")
    [Nodo "Anakin" (Nom 42 "BBY")
        [Nodo "Luke" (Nom 19 "BBY")
            [Nodo "Ben" (Nom 26 "ABY") []],
        Nodo "Leia" (Nom 19 "BBY")
            [Nodo "Jaina" (Nom 9 "ABY") [],Nodo "Jacen" (Nom 9 "ABY")
[],Nodo "Ben" (Nom 1 "ABY") [],Nodo "Anakin" (Nom 10 "ABY") []]]]
```

Ejercicio 4

```
#include <list>

class Objeto {
protected:
    float x, y, z;
public:
    Objeto(const float& x, const float& y, const float& z) : x(x),
y(y), z(z) {}
    virtual ~Objeto() {}
};

class Fijo : public Objeto {
public:
    Fijo(const float& x, const float& y, const float& z) : Objeto(x, y,
z) {}
};

class Movil : public Objeto {
public:
    Movil(const float& x, const float& y, const float& z) : Objeto(x,
y, z) {}

    void update(float t) {
        x += t;
        y += t;
        z += t;
    }
};
```

```
class Grupo{
    std::list<Objeto*> els;
public:
    Grupo(std::list<Objeto*> o) : els(o) {}

    void add(Objeto& o, bool error) {
        if(els.empty()){
            els.push_back(&o);
        }
        else {
            if(dynamic_cast<Fijo*>(*els.begin()) != nullptr &&
dynamic_cast<Fijo*>(&o) != nullptr){
                Fijo* f = dynamic_cast<Fijo*>(&o);
                els.push_back(f);
            }

            else if(dynamic_cast<Movil*>(*els.begin()) != nullptr &&
dynamic_cast<Movil*>(&o) != nullptr){
                Movil* m = dynamic_cast<Movil*>(&o);
                els.push_back(m);
            }
            else {
                error = true;
            }
        }

    }
};

class Escena {
    std::list<Objeto*> objs;
public:
    void sort();

    void draw();

    void update(float t) {
        for (auto i : objs) {
            if (dynamic_cast<Movil*>(i) != nullptr) {
                Movil* m = dynamic_cast<Movil*>(i);
                m->update(t);
            }
        }
    }
};

void main() {
    Grupo g ({new Fijo(1.0, 1.0, 1.0)});

}
```