Ejercicio 1

```
#include <vector>
#include <iomanip>
#include <iostream>

class Suma {
    float operator() (const float& a, const float& b) {
        return a + b;
    }
};

class Producto {
    float operator() (const float& a, const float& b) {
        return a * b;
    }
};

template <typename A, typename B, typename C,  typename F>
std::vector<C> zipWith (F f, std::vector<A> v, std::vector<B> w) {
    std::vector<C> aux;
    auto ia = begin(v);
    auto ib = begin(w);
    for (; (ia!=end(v)) and (ib!=end(w)); ++ia,++ib) {
        aux.push_back(f(ia*, ib*));
    }
    return aux;
}

template <typename A, typename B, typename C, typename F>
A foldl (F f, A param1, std::vector<B> w) {
    A res = param1;
    for (auto i : w) {
        res = f(res,w);
    }
    return res;
}

template <typename T>
float dot(std::vector<float> a, std::vector<float> b) {
    return foldl(Suma(), 0.0, zipWith(Producto(), a, b));
}
```

Ejercicio 2

Apartado a)

```
#include <iostream>
using namespace std;
class Padre {
    protected:
        int atr1;
        int atr2;
    public:
        Padre(int atr1, int atr2) : atr1(atr1), atr2(atr2) {}
        virtual int metodo1() const {
            return atr1 + atr2;
        }
};
class Hija : public Padre {
    public:
        Hija(int atr1, int atr2) : Padre(atr1,atr2) {}
        int metodo1() const override{
            return atr1 * atr2;
        }
};
int funcion(const Padre& p) {
    return p.metodo1();
}

int main() {
    Padre p(5,6);
    Hija h(3,4);
    cout << funcion(p) << endl;
    cout << funcion(h) << endl;
}
```

Apartado b)

El polimorfismo por inclusión se consigue en este código a través de dos mecanismos de la herencia. Por un lado, en la clase Hija redefine el comportamiento definido en el método "metodo1" de la clase Padre.

Por otro lado, en la función creada se permite como parámetro cualquiera de las dos clases, dado que un puntero o referencia a una clase padre puede estar representando a cualquiera de sus clases hijas, como ocurre en este caso. Por ello al invocar a la función en el main, se puede pasar como parámetro tanto un objeto de la clase Padre como un objeto de la clase Hija.

Apartado c)

Se aplica la asociación dinámica, ya que de lo contrario no se podría redefinir el comportamiento de los métodos de la clase padre ni acceder a la clase hija a través de la clase padre.

Ejercicio 3

Apartado a)

```
type ReglaBasica = (Double -> Double) -> (Double, Double) -> Double

trapecio :: ReglaBasica
trapecio f (a, b) = if a==b then 0.0
                    else (b-a) / 2.0 * (f a + f b)

simpson :: ReglaBasica
simpson f (a, b) = if a==b then 0.0
                   else ((b-a) / 6.0) * (f a + 4.0 * f((a+b)/2.0) + f
```

Apartado b)

```
integralCompuesta :: ReglaBasica -> Integer -> (Double -> Double) ->
(Double, Double) -> Double
integralCompuesta regla n f (a, b) =
    sum $ map (regla f) $ zip l (tail l)
    where
        dx = (b-a) / fromInteger n
        l = [a, a+dx..b]
```

Apartado c)

```
integralAdaptativa :: ReglaBasica -> ReglaBasica -> Double -> (Double
-> Double) -> (Double, Double) -> Double
integralAdaptativa r1 r2 t f (a, b)
    | abs (ra - rb) > t = minf + msup
    | otherwise = ra
    where
        ra = r1 f (a, b)
        rb = r2 f (a, b)
        c = (a+b)/2
        minf = integralAdaptativa r1 r2 t f (a, c)
        msup = integralAdaptativa r1 r2 t f (b, c)
```

Ejercicio 4

```
#include <iostream>
#include <list>
using namespace std;

class Residuo {
protected:
    string NIF;
public:
    Residuo(const string& NIF) : NIF(NIF) {}

    virtual ~Residuo() {}

    string suNIF() const {
        return NIF;
    }
};

class Organico : public virtual Residuo {
protected:
    int tiempo;
public:
    Organico(const string& NIF, int t) : Residuo(NIF), tiempo(t) {}

    int suTiempo() const {
        return tiempo;
    }
};

class Inorganico : public virtual Residuo {
public:
    Inorganico(const string& NIF) : Residuo(NIF) {}
};

class Liquido : public virtual Residuo {
protected:
    int volumen;
public:
    Liquido(const string& NIF, int v) : Residuo(NIF), volumen(v) {}

    int suVolumen() const {
        return volumen;
    }
};
```

```
class Solido : public virtual Residuo {
protected:
    int peso;
public:
    Solido(const string& NIF, int p) : Residuo(NIF), peso(p) {}

    int suPeso() const {
        return peso;
    }
};

class Aceite : public Organico, public Liquido {
public:
    Aceite(const string& NIF, int t, int v) : Organico(NIF, t),
Liquido(NIF, v), Residuo(NIF) {}
};

class Pintura : public Inorganico, public Liquido {
public:
    Pintura(const string& NIF, int v) : Inorganico(NIF), Liquido(NIF,
v), Residuo(NIF) {}
};

class Basura : public Organico, public Solido {
public:
    Basura(const string& NIF, int t, int p) : Organico(NIF, t),
Solido(NIF, p), Residuo(NIF) {}
};

class Vidrio : public Inorganico, public Solido {
public:
    Vidrio(const string& NIF, int p) : Inorganico(NIF), Solido(NIF, p),
Residuo(NIF) {}
};
```

```
class Recogida {
    list<Residuo*> pendientes;
public:
    Recogida(list<Residuo*> p) : pendientes(p) {}

    int volumen(string NIF) {
        int total = 0;
        for (auto i : pendientes) {
            if (dynamic_cast<Liquido*>(i) != nullptr and i->suNIF() ==
NIF) {
                Liquido* l = dynamic_cast<Liquido*>(i);
                total += l->suVolumen();
            }
        }
        return total;
    }

    list<string> recogidasPendientes(int dias) {
        list<string> aux;
        for (auto i : pendientes) {
            if (dynamic_cast<Organico*>(i) != nullptr) {
                Organico* o = dynamic_cast<Organico*>(i);
                if (o->suTiempo() < dias){
                    aux.push_back(o->suNIF());
                }
            }
        }
        return aux;
    }
};

int main() {
    Aceite* a = new Aceite("Oscar", 8, 30);
    Pintura* p = new Pintura("Pablo", 200);
    list<Residuo*> recogida = {
        new Aceite("Oscar", 8, 30),
        new Pintura("Pablo", 200),
        new Basura("Pablo", 2, 3),
        new Vidrio("Oscar", 8)
    };
    Recogida r(recogida);

    cout << r.volumen("Oscar") << endl;
    cout << r.volumen("Pablo") << endl;

    list<string> NIFS = r.recogidasPendientes(4);

    for (auto i : NIFS) {
        cout << i << endl;
    }
}
```