Ejercicio 1

```
using namespace std;

void mira(Foo* foo) {
    foo->sil();
    cout << foo->meta() << endl;
}

int main() {
    mira(new Foo(2.0));      // 1.0
    mira(new Bar(4.0));      // 1.5
}

template<typename T>
void show(const T& t) {
    cout<<t.mitzvah()<<endl;
}
int main() {
    show(Foo());                  // 1
    show(bar(Foo()));             // 2
    show(bar(bar(Foo())));        // 3
    show(bar(bar(bar(Foo()))));   // 4
}
```

Ejercicio 2

Apartado a)
Un tipo de dato genérico es aquel que se especifica mediante un tipo desconocido (paramétro) en su definición. Este tipo se concretará después en su instanciación.

Apartado b)
Las fases de la utilización de un tipo genérico de datos son:
- Definición: se implementa el tipo genérico indicando una serie de elementos parámetro desconocidos a priori
- Instanciación: se utiliza el tipo genérico especificando valores concretos para los elementos parámetro

Apartado c)
```
#include <iostream>
using namespace std;

template <typename T>
class A {
    T dato1, dato2;
public:
    A(T d1, T d2) : dato1(d1), dato2(d2) {}

    T metodo1() const {
        return dato1 * dato2;
    }
};

int main(){
    A<int> a(5, 6);
    A<double> b(6.3, 5.2);
    A<string> c("a", "pablo");
    c.metodo1();  Apartado e)
}
```

Apartado d)
La clase A se especifica mediante el tipo de dato T, que es desconocido. Posteriormente, en el main, se instancian los objetos a, b y c, en los que se reemplaza el tipo T por int, double y string respectivamente.

Apartado f)
El error se produce porque no se puede ejecutar la acción de metodo1 con el tipo de dato string especificado en la instanciación de c, ya que no soporta la operación que tiene lugar en el metodo1.

Ejercicio 3

```
maximo ::  [Int] -> Int
maximo [x] = x
maximo (x:xs)
    | x > maximo xs = x
    | otherwise = maximo xs

parteCreciente :: [Int] -> [Int]
parteCreciente xs = takeWhile (< maximo xs) xs

comprobarParteCreciente :: [Int] -> Bool
comprobarParteCreciente [] = False
comprobarParteCreciente [_] = False
comprobarParteCreciente xs
    | x1 > x2 = False
    | x1 < x2 && (length xs == 2) = True
    | otherwise = comprobarParteCreciente (tail xs)
        where
            [x1,x2] = take 2 xs

parteDecreciente :: [Int] -> [Int]
parteDecreciente xs = dropWhile (< maximo xs) xs

comprobarParteDecreciente :: [Int] -> Bool
comprobarParteDecreciente [_] = False
comprobarParteDecreciente xs
    | x1 > x2 && (length xs == 2) = True
    | x1 > x2 = comprobarParteDecreciente (tail xs)
    | otherwise = False
        where
            [x1,x2] = take 2 xs

esMonticulo :: [Int] -> Bool
esMonticulo xs = comprobarParteCreciente (parteCreciente xs) &&
comprobarParteDecreciente (parteDecreciente xs)
```

Ejercicio 4

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

template <typename T>
class Operador{
    protected:
    T operando1, operando2;
    public:
    Operador(const T& op1, const T& op2) : operando1(op1),
operando2(op2){}

    virtual T operacion() const = 0;

    virtual string mostrar() const = 0;

    virtual T operacionConOperando(const T& operando) const = 0;
};

template <typename T>
class Suma : public Operador<T>{
    protected:
    using Operador<T>::operando1;
    using Operador<T>::operando2;
    public:
    Suma(const T& op1, const T& op2) : Operador<T>(op1,op2){}

    T operacion() const override {
        return operando1 + operando2;
    }

    string mostrar() const override{
        return to_string(operando1) + " + " + to_string(operando2);
    }

    T operacionConOperando(const T& operando) const override{
        return operando + operando1;
    }
};

template <typename T>
class Resta : public Operador<T>{
    protected:
    using Operador<T>::operando1;
    using Operador<T>::operando2;
    public:
    Resta(const T& op1, const T& op2) : Operador<T>(op1,op2){}
```

```
    T operacion() const override {
        return operando1 - operando2;
    }

    string mostrar() const override{
        return to_string(operando1) + " - " + to_string(operando2);
    }

    T operacionConOperando(const T& operando) const override{
        return operando - operando1;
    }
};

template <typename T>
class Multiplicacion : public Operador<T>{
    protected:
    using Operador<T>::operando1;
    using Operador<T>::operando2;
    public:
    Multiplicacion(const T& op1, const T& op2) : Operador<T>(op1,op2){}

    T operacion() const override {
        return operando1 * operando2;
    }

    string mostrar() const override{
        return to_string(operando1) + " * " + to_string(operando2);
    }

    T operacionConOperando(const T& operando) const override{
        return operando * operando1;
    }
};

template <typename T>
class Division : public Operador<T>{
    protected:
    using Operador<T>::operando1;
    using Operador<T>::operando2;
    public:
    Division(const T& op1, const T& op2) : Operador<T>(op1,op2){}

    T operacion() const override {
        return operando1 / operando2;
    }

    T operacionConOperando(const T& operando) const override {
        return operando / operando1;
    }
```

```
    string mostrar() const override{
        return to_string(operando1) + " / " + to_string(operando2);
    }
};

template <typename T>
class Parentesis : public Operador<T>{
    string oper;
    public:
    Parentesis(const T& op1, const T& op2, const string& oper) :
Operador<T>(op1,op2), oper(oper){}
    string suOper()const{
        return oper;
    }
};

template <typename T>
class Evaluacion{
    list<Operador<T>*> ev;
    public:
    Evaluacion(list<Operador<T>*> ev) : ev(ev){}

    string mostrar() const {
        string cadena = "";
        for(auto i : ev){
            cadena = cadena + i->mostrar() + " ";
        }
        return cadena;
    }

    T evaluacion(){
        T total = 0;
        bool primeraOperacion = true;
        for(auto i : ev){
            if(dynamic_cast<Multiplicacion<T>*>(i) != nullptr &&
primeraOperacion){
                Multiplicacion<T>* m =
dynamic_cast<Multiplicacion<T>*>(i);
                total = m->operacion();
                primeraOperacion = false;
                // ev.remove(i);
            }
            else if(dynamic_cast<Multiplicacion<T>*>(i) != nullptr){
                Multiplicacion<T>* m =
dynamic_cast<Multiplicacion<T>*>(i);
                total = m->operacionConOperando(total);
            }
        }
```

```
        for(auto i : ev){
            if(dynamic_cast<Division<T>*>(i) != nullptr &&
primeraOperacion){
                Division<T>* m = dynamic_cast<Division<T>*>(i);
                total = m->operacion();
                primeraOperacion = false;
            }
            else if(dynamic_cast<Division<T>*>(i) != nullptr){
                Division<T>* m = dynamic_cast<Division<T>*>(i);
                total = m->operacionConOperando(total);
            }
        }
        for(auto i : ev){
            if(dynamic_cast<Suma<T>*>(i) != nullptr &&
primeraOperacion){
                Suma<T>* m = dynamic_cast<Suma<T>*>(i);
                total = m->operacion();
                primeraOperacion = false;
            }

            else if(dynamic_cast<Suma<T>*>(i) != nullptr){
                Suma<T>* m = dynamic_cast<Suma<T>*>(i);
                total = m->operacionConOperando(total);
            }
        }
        for(auto i : ev){
            if(dynamic_cast<Resta<T>*>(i) != nullptr &&
primeraOperacion){
                Resta<T>* m = dynamic_cast<Resta<T>*>(i);
                total = m->operacion();
                primeraOperacion = false;
            }

            else if(dynamic_cast<Resta<T>*>(i) != nullptr){
                Resta<T>* m = dynamic_cast<Resta<T>*>(i);
                total = m->operacionConOperando(total);
            }
        }
        return total;
    }
};

int main(){
    Evaluacion<int> e({
        new Suma<int>(3,3),
        new Multiplicacion<int>(2,5),
    });

    cout << e.evaluacion();
}
```