Ejercicio 1

```
int main() {
    Foo* foo = new Foo(1);         // i = 1
    cout<<foo->sil()<<" | " <<foo->yokey()<<endl;   // 4 | 2
    Bar* bar = new Bar(2);         // i = 5
    cout<<bar->sil()<<" | " <<bar->yokey()<<endl;   // 3 | -3
    Foo* foobar = new Bar(3);      // i = 6
    cout<<foobar->sil()<<" | " <<foobar->yokey()<<endl; // -8 | -4
}

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

template<typename F>
void templ(F* f) {
    cout<<f->meta()<<endl;
}
int main() {
    Foo foo;
    Bar bar;
    inher(&foo);       // “FOO”
    inher(&bar);       // “FOO”
    templ(&foo);       // “FOO”
    templ(&bar);       // “BAR”
}
```

Ejercicio 2

Apartado a)
La programación genérica es un estilo de programación que permite escribir código que opera de la misma forma con cualquier tipo de datos, sin que estos tipos tengan que estar relacionados por la herencia. Asimismo permite escribir código reutilizable independientemente de los datos sobre los que se va a aplicar.

Apartado b)
La programación genérica se puede realizar mediante el polimorfismo paramétrico.

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

template <typename T>
T minimo1(T a, T b) {
    if (a < b) {
        return a;
    }
    else {
        return b;
    }
}

int main() {
    cout << minimo1(3, 5) << endl;
    cout << minimo1(4.7, 3.2) << endl;
    cout << minimo2(3, 5) << endl;
    cout << minimo2(4.7, 3.2) << endl;
}
```

Apartado d)
En el ejemplo se puede observar que el metodo1 se puede realizar con cualquier tipo de dato compatible con la operacion que se realiza en él (el mínimo entre dos números).

Apartado e)

```
int minimo2(int a, int b) {
    if (a < b) {
        return a;
    }
    else {
        return b;
    }
}

double minimo2(double a, double b) {
    if (a < b) {
        return a;
    }
    else {
        return b;
    }
}
```

Apartado f)

Si no existiera la programación genérica se debería realizar un método por cada tipo de dato al cual se quiera aplicar la operación de dicho método.

Ejercicio 4

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

class Componente {
protected:
    uint64_t dato;
public:
    Componente(uint64_t d) : dato(d) {}

    uint64_t suDato() const {
        return dato;
    }
};

class Teclado : public virtual Componente {
public:
    Teclado(uint64_t d) : Componente(d) {}

    uint64_t leer() const {
        return dato;
    }
};

class Pantalla : public virtual Componente {
public:
    Pantalla() : Componente(0) {}

    void escribir(uint64_t d) {
        dato = d;
    }
};

class Registro : public Teclado, public Pantalla {
    int num;
public:
    Registro(int n, uint64_t d) : num(n), Teclado(d), Pantalla(),
Componente(d) {}

    int suNum() const {
        return num;
    }
};
```

```
class Maquina {
    list<Registro> regs;
    Pantalla pan;
    Teclado tec;
public:
    Maquina(list<Registro> r, Pantalla p, Teclado t) : regs(r), pan(p),
tec(t) {}

    void mov() {
        pan.escribir(tec.leer());
    }

    void inc(Registro& r) {
        for (auto i : regs) {
            if (i.suNum() == r.suNum()) {
                i.escribir(i.leer()+1);
                cout << i.suDato() << endl;
            }
        }
    }
};

int main() {
    Teclado t(10);
    Pantalla p;
    Registro r1(1, 0), r2(2, 0), r3(3, 0), r4(4, 0), r5(5, 0);
    list<Registro> regs = {r1, r2, r3, r4, r5};
    Maquina m(regs, p, t);
    m.mov();
    m.inc(r4);
}
```