Seminarios-RESUELTOS.pdf

Murphy_

Programación Orientada a Objetos

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

SEMINARIO 2.1

Ejercicio 1. ¿Cómo implementa el lenguaje de programación C++ el principio de encapsulamiento?

El principio de encapsulamiento se implementa mediante las clases.

Ejercicio 2. ¿Cómo implementa el lenguaje de programación C++ el principio de ocultación de información?

Mediante la parte privada de las clases.

Ejercicio 3 ¿Hay algún error en el siguiente programa? Si es así, explique por qué y corríjalo

```
#include <iostream>
class C
{
    public:
        C(int i = 0): n(i) {}

        void mostrar() { std::cout << "i = " << n << std::endl; }

    private:
        int n;
};

int main()
{
    const C c;

    c.mostrar();

    return 0;
}
```


Lo que falla en este código es que estas llamando a un método el cual no está marcado como constante en un objeto creado como `const`.

Solución:

```
#include <iostream>
class C
{
    public:
        C(int i = 0): n(i) {}
//              AQUÍ ESTÁ EL ERROR CORREGIDO
        void mostrar() const { std::cout << "i = " << n << std::endl; }
```

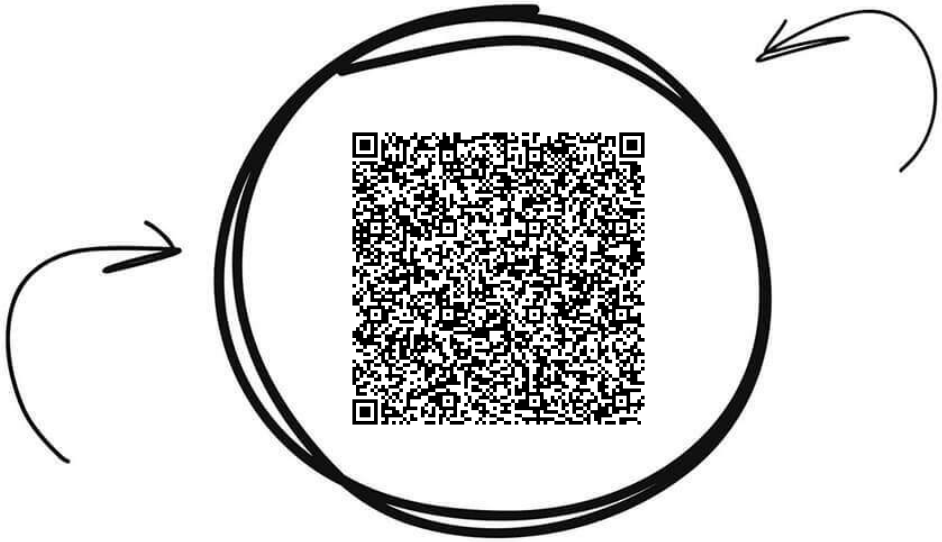
```
    private:
        int n;
};

int main()
{
    const C c;

    c.mostrar();

    return 0;
}
```

PEKIS®
for Foodies

¿RELLENANDO APUNTES? RELLENATE UN TACO.

RELLENOS PARA TACOS PEKIS® BRUTALES.

Programación Orientada a Obj...



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

TAREA 2.2

Ejercicio 1. Enumere las diferencias existentes entre inicializar un atributo en la lista de inicialización y asignarle un valor en el cuerpo del constructor.

Al inicializar un valor, lo creas directamente con un valor especifico, determinado, mientras que si lo asignas, el atributo en un principio se inicializará con basura y no será hasta el momento de la asignación que el atributo tendrá el valor deseado. Esto puede dar igual mientras que el atributo sea no constante, pero si este está marcado como `const`, no se podrá modificar el valor inicial, es decir, el atributo permanecerá con valor basura.

Ejercicio 2. La lista de inicialización y la lista inicializadora, ¿cumplen la misma función en la construcción de objetos? ¿Es posible utilizar estas listas en otros métodos de una clase?

Ambas listas se usan para inicializar, ahora, ¿qué inicializa cada una?

- La lista inicializadora inicializa tipos contenedores, es decir, secuencias de elementos de un mismo tipo, como pueden ser vectores, colas...
    - Esta sí se puede usar en otros métodos de la clase. Ejemplo:

```
#include <iostream>
#include <initializer_list>


class C
{
    public:
        C(int n = 1): numero{n} {} // Lista de inicialización

        void print_values(std::initializer_list<int> values)
        {
            for(int i:values)
                std::cout << i << std::endl;
        }

    private:
        int numero;
};

int main()
{
    C c();

    c.print_values({1,5,2,4,9,7,5})

    return 0;
}
```

- La lista de inicialización inicializa un objeto atributo a atributo. Al contrario de la lista inicializadora, la lista de inicialización no se puede usar en métodos que no sean constructores.

Ejercicio 3. Diga qué función de la clase punto se llama en cada una de lassiguientes líneas. Si alguna depende de una línea anterior que sea incorrecta, corríjala previamente.

```
class punto
{
    double x, y; // Private

    public:

    punto(double a = 0., double b = 0.) : x{a}, y{b} {}
    punto(const punto& p) : x{p.x}, y{p.y} {}

    punto& operator =(const punto& p)
    {
        x = p.x; y = p.y;
        return *this;
    }
};

int main()
{
    punto p;         // Constructor predeterminado.
    punto q();       // Error
    punto r(2.,);    // Error.
    punto s{3.4};    // Constructor con un parámetro.
    punto t{};       // Constructor predeterminado.
    punto u(q);      // Constructor por copia.
    punto v = r;     // Constructor por copia.
    t = s;           // Operador de asignación.

    return 0;
}
```

Error 1 `punto q()` : el compilador se cree que está llamando a una función llamada q que devuelve un punto.
Error 2 `punto r(2.,)` : el compilador espera otro parámetro tras la coma.

Ejercicio 4. Sean las clases Libro1 y Libro2:

```
class Libro1
{
    string titulo_;
    int pags_;

    public:
        Libro1(string t = "", int p = 0);
// ...
};

class Libro2
{
    string titulo_;
    int pags_;

    public:
        Libro2(string t, int p = 0);
        Libro2(const char* c);
 // ...

 };
```

Decida si X se puede sustituir por 1, 2 o ambos en los siguientes items:

1. Se puede definir: LibroX lib1
   - `Libro1 lib1` : SI.
   - `Libro2 lib1` : NO, porque no tenemos un constructor predeterminado.

2. Se tiene un constructor de conversión de std::string a LibroX
   - `Libro1` : SI.
   - `Libro2` : SI.

3. Se puede definir: LibroX lib2[5]
   - `Libro1 lib1[5]` : SI.
   - `Libro2 lib1[5]` : NO, porque no tenemos un constructor predeterminado.

   > Al crear un array, todos los elementos del mismo se crean mediante el constructor predeterminado.

4. Se puede definir: std::vector lib3
   - `std::vector<Libro1> lib3` : SI.
   - `std::vector<Libro2> lib3` : SI.

5. Siendo "El Quijote" una cadena literal de tipo `const char*` ; se produce una conversión implícita a string al ejecutar: `LibroX* lib4 = new LibroX("El Quijote")` .
    - `Libro1* lib4 = new LibroX("El Quijote")` : SI.
    - `Libro2* lib4 = new LibroX("El Quijote")` : SI.

6. Se puede definir: LibroX lib5 = "El Quijote"
    - `Libro1 lib5 = "El Quijote"` : NO, no hay ningún método conversor.
    - `Libro2 lib5 = "El Quijote"` : SI, gracias al constructor que recibe un `const char*`

7. Hace falta definir el destructor para LibroX
    - `Libro1` : NO, no es necesario ya que los atributos son destruidos por el destructor por defecto.
    - `Libro2` : NO, no es necesario ya que los atributos son destruidos por el destructor por defecto.

Ejercicio 5. Considere la siguiente clase Libro:

```
#include <iostream>
#include <cstring>

using namespace std;

class Libro
{
    char* titulo_;
    int paginas_;

    public:
        Libro() : titulo_(new char[1]), paginas_(0) {*titulo_= 0;}
        Libro(const char* t, int p) : paginas_(p)
        {
            titulo_ = new char[strlen(t) + 1];
            strcpy(titulo_, t);
        }

        int paginas() const { return paginas_; }
        char* titulo() const { return titulo_; }

        ~Libro() { delete[] titulo_; }


 };
```

Diga si el siguiente programa funciona correctamente. En caso afirmativo indique lo que imprime. En caso negativo haga las modificaciones necesarias para que funcione correctamente.

```
void mostrar(Libro l)
{
    cout << l.titulo() << " tiene " << l.paginas() << " páginas" << endl;
}

int main() {

    Libro l1("Fundamentos de C++", 474), l2;
    l2 = l1;
    mostrar(l2);

    return 0;
}
```

A pesar de que se imprime correctamente, el compilador usa el operador de asignación por defecto, lo cual asignará los elementos uno por uno, de manera que hará lo siguiente: `char* l2.titulo_ = l1.titulo_` y, lo único que va a conseguir con esto es cambiar el apuntado, donde apuntará l2.titulo_ y no el contenido de este. Es decir, el espacio de memoria que se había reservado para este atributo, se perderá, perdiendo también ese espacio de memoria, el cual quedará inútil.

La solución sería sobrecargar el operador de asignación por nosotros mismos:

```
class Libro
{
    char* titulo_;
    int paginas_;

    public:
        Libro() : titulo_(new char[1]), paginas_(0) {*titulo_= 0;}
        Libro(const char* t, int p) : paginas_(p)
        {
            titulo_ = new char[strlen(t) + 1];
            strcpy(titulo_, t);
        }

        Libro& operator=(const Libro& L);

        int paginas() const { return paginas_; }
        char* titulo() const { return titulo_; }

        ~Libro() { delete[] titulo_; }


 };

 Libro& Libro::operator=(const Libro& L)
 {
    if(this != &L) // Evitamos autoasignacion
    {
        if(strlen(titulo_) != strlen(L.titulo_))
```

```
        {
            delete[] titulo_; // vaciamos lo que había antes

            titulo_ = new char*[strlen(L.titulo_)+1];
            paginas_ = L.paginas_;
        }

        strcpy(titulo_, L.titulo_);
    }

    return *this;
 }
```

Ejercicio 6. a)Describa el error de compilación que provoca el código anterior. ¿Cómo modificaría las clases sin suprimir métodos para solucionarlo?.

b) Suponga que el parámetro de f() es de entrada y salida y la línea 9 es sustituida por void f(A&);. ¿Qué error de compilación se produce? ¿Y cómo se puede resolver sin suprimir métodos?

```
struct B; // Declaración adelantada

struct A
{
    A(B); // Constructor de conversión de B a A
};

struct B
{
    explicit operator A(); // Operador de conversión de B a A
};

void f(A&); // Recibe por referencia un objeto constante de A

int main()
{
    B b;
    f(b);
}
```

a) Al ejecutar este código, el compilador lanzará un error de ambigüedad debido a que existen dos métodos conversores de B -> A. Estos son el constructor de conversión de A y la sobrecarga del operador de conversión A. La solución es añadirle explicit en cualquiera de los dos:

```
struct B
{
    explicit operator A(); // Operador de conversión de B a A
};
```

b) El error se encuentra a la hora de convertir implícitamente un objeto de tipo B en uno de tipo A ya que no puede realizar la conversión de un objeto a otro tipo si este no es constante. Una solución para este problema sería:

```c
int main()
{
    B b;
    A a(b);
    f(a);
}
```

MURPHY

Tarea 2.3

Ejercicio 1. Clasifique las funciones y operadores miembro de la clase matriz en diferentes categorías (constructores, destructores, observadores y modificadores).

```
// CONSTRUCTORES
    explicit matriz(size_t m = 1, size_t n = 1, double y = 0.0);
    matriz(size_t m, size_t n, double f(size_t i, size_t j));
    matriz(const initializer_list<valarray<double>>& l);
    matriz(const matriz&) = default;
    matriz(matriz&&) = default;


// OBSERVADORES
    size_t filas() const;
    size_t columnas() const;
    double operator ()(size_t i, size_t j) const;
    valarray<double> operator [](size_t i) const;
    valarray<double> operator ()(size_t j) const;


// MODIFICADORES
    matriz& operator =(const matriz&) = default;
    matriz& operator =(matriz&&) = default;
    double& operator ()(size_t i, size_t j);
    slice_array<double> operator [](size_t i);
    slice_array<double> operator ()(size_t j);
    matriz& operator =(double y);
    matriz& operator +=(const matriz& a);
    matriz& operator -=(const matriz& a);
    matriz& operator *=(const matriz& a);
    matriz& operator *=(double y);
```

Ejercicio 2. Si existen los siguientes constructores, escriba una instrucción en cada caso en la que se invoque al mismo y diga si se utiliza en el programa de prueba (en caso afirmativo indique dónde): constructor predeterminado, constructor de copia, constructor de movimiento y constructor de conversión.

```
#include <iostream>
#include "matriz.h"

using namespace std;
// Inserción de una matriz en un flujo de salida .
ostream& operator <<(ostream& fs, const matriz& a)
{
    for (size_t i = 0; i < a.filas(); ++i)
    {
        for (size_t j = 0; j < a.columnas(); ++j)
            fs << a(i,j) << '␣';
```

```
        fs << endl;
    }

    return fs;
}
// Función delta de Kronecker.
inline double delta(size_t i, size_t j)
{
    return i == j;
}


// Prueba.
int main()
{
26  matriz A(3, 3); // Matriz nula de 3 x 3
27  matriz B(3, 3, 2.0); // Matriz de 3 x 3 con todos sus elementos a 2
28  matriz C(3, 3, delta); // Matriz identidad de 3 x 3
29  matriz D = { {1, 2, 3},
                 {4, 5, 6},
                 {7, 8, 9} };

    A = C;
    cout << "A = \n" << A << endl;

    B += -A;
    cout << "B = \n" << B << endl;

    C *= C += C;
    cout << "C = \n" << C << endl;

39  cout << "2A + B * C = \n" << 2 * A + B * C << endl;

    cout << "D=\n" << D << endl;

    matriz tD(3, 3);

    for (size_t i = 0; i < D.filas(); ++i)
        tD[i] = D(i);

    cout << "traspuesta de D = \n" << tD << endl;

    for (size_t i = 0; i < D.filas(); ++i)
        for (size_t j = 0; j < D.columnas(); ++j)
            --D(i,j);
    cout << "D=\n" << D << endl;

    D = -1.;
    cout << "D=\n" << D << endl;
}
```

Constructor predeterminado: `Matriz M;` No se usa.

Constructor por copia: `Matriz N{M};` Se usa en la linea 39 del *main*.

Constructor por movimiento: `Matriz H{std::move(N)}`. No se usa en el programa de prueba.

Constructor de conversión: `Matriz J(3,1,6.5)`. Se usa en las lineas 26, 27, 28, 29.

Ejercicio 3. Describa los errores que hay en el siguiente código:

```
matriz A = 3; // Error
matriz B = matriz(5);
matriz C(3);
B = 2;  // Error
A = matriz(5);
```

La instrucción `matriz A = 3;` produce un error ya que el constructor de conversión está marcado como `explicit` y no permitirá que se le pase ningún otro tipo que no sea un `size_t` .

La otra instrucción que produce un error es `B = 2;` , esto es debido a que no hay ningún operador de conversión `int` -> `matriz` .

¿Por qué se declara `explicit` el primer constructor de la clase matriz?

Para evitar que el compilador haga conversiones implícitas no deseadas.

¿Podría causar algún problema si no se hiciera así?

Si quitamos el `explicit` , en la instrucción `B = 2;` el compilador no sabría si transformar el 2 a una `matriz` o transformarlo a un `double` debido a la sobrecarga del operador de asignación que recibe un `double` .

¿se podría evitar ese problema definiendo un operador de conversión de int a matriz `operator matriz(int)` ?

No, seguiría sin arreglarse ya que el compilador seguiría sin saber que hacer, si transformar el entero en una matriz y llamar al operador de asignación que recibe una matriz o convertir ese entero en un double y llamar al operador de conversión el cual recibe dicho tipo de elemento, seguiría habiendo la misma ambigüedad (como mínimo).

Ejercicio 4. ¿Por qué devuelven tipos distintos los operadores de signo '+' y '-' ?

> Interpreto por operadores de signo los dos operadores unarios que hay, ya que los aritméticos (binarios) devuelven el mismo tipo.

```
friend matriz operator -(const matriz& a)  // Externo con acceso a la parte
privada
{
    matriz c(a);
    c.x = -c.x;
    return c;
}
const matriz& operator +(const matriz& a) // Externo
{
    return a;
}
```

Partimos de la base de que el '-' lo que hace es devolver la matriz opuesta a la que se le pasa, no convertir la que se le pasa en su matriz opuesta, por lo que hace necesaria la creación de una copia que, al encontrarse dentro del ámbito de la función, se destruirá automáticamente al acabar esta, por lo cual no tendría sentido pasar una dirección de una variable que una vez acaba dicha función no existe. Por eso mismo se devuelve ese objeto por valor, es decir, por copia.

En cuanto al operador suma '+', en este caso devuelve una referencia, simple y llanamente porque es un método observador debido a que no modifica ningún elemento ni objeto.

En resumen: devuelven diferentes tipos porque hacen cosas totalmente diferentes, el operador '-' devuelve un nuevo objeto con la matriz opuesta a `a` y el operador '+' devuelve el mismo objeto que se le pasa, si hacerle ningún cambio.

Ejercicio 5. ¿El operador '-' de cambio de signo es miembro de la clase matriz? ¿Se podría definir de la otra forma? En caso afirmativo, escriba la declaración. ¿Qué ventajas e inconvenientes tendría?

No, el operador '-' no pertenece a la clase matriz, sin embargo es amiga, es decir, que tiene acceso a la parte privada aun no siendo miembro de la misma.

Sí, se podría definir como un método de la clase de la siguiente manera:

```
class matriz
{
    public:

        //...
        matriz operator -();

    private:

        //...
}
```

Ventajas:

- Ningún método externo tiene acceso a la parte privada.
- El número de parámetros formales disminuye a 0 (se le pasa de forma implícita mediante el objeto `this`).

Desventajas:

- Si el parámetro implícito que se le pasa necesita alguna conversión implícita, esta no se aplicará al ser implícito.

Ejercicio 6. ¿Es correcto definir el operador '*=' como sigue?

```
inline matriz& matriz::operator *=(const matriz& a)
{
    n = a.columnas();
    x *= a.x;
    return *this;
}
```

No. Según esta definición, la multiplicación de dos matrices se hace multiplicando uno por uno sus elementos, algo que es incorrecto, por lo cual no se es correcta dicha definición.

TAREA 3.1

Ejercicio 1. Suponga que hay que desarrollar la interfaz de usuario de una aplicación. Dicha interfaz estará formada por menús, opciones y formularios. Hay que tener en cuenta que:

- **Desde cada menú se puede ejecutar una serie de opciones.**

- **La selección de una opción desencadena la activación de un formulario.**

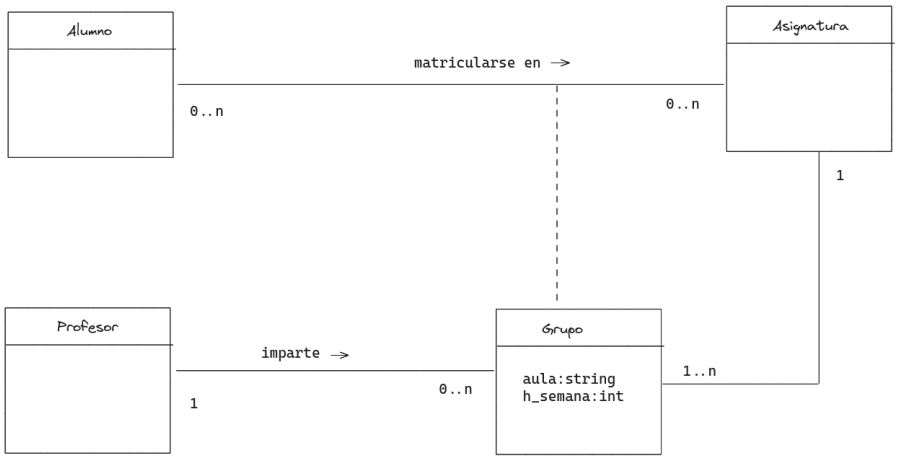
- **Una opción solo puede aparecer en un menú, pero un mismo formulario puede ser compartido por varias opciones.**

Describa e implemente las relaciones que se establecerán entre estas clases.

```
#include <iostream>
#include <set>
#include <map>

class Menu
{
    public:
        typedef std::set<Opcion> opciones; // Conjunto de opciones.

        void ejecuta(Opcion& O) {opciones_.insert(&O);} // REVISAR
        const opciones& ejecuta() const noexcept {return opciones_;}//REVISAR
        friend bool operator <(const Opcion& A, const Opcion& B) const;

    private:
        opciones opciones_;
};

bool operator <(const Opcion& A, const Opcion& B) const;


class Opcion
{
    public:

        void esta_en(menu& m);
        menu& esta_en() const noexcept {return *menu_;}

        void activa(Formulario& f);
        Formulario& activa() const noexcept;

    private:
        Menu* menu_;
        Formulario* formulario_;
};
```

```
class Formulario
{
    public:
        typedef std::set<Opcion*> opciones_formulario; // Conjunto de
formularios compartidos por opciones

        void activado_por(Opcion& O);
        const opciones_formulario& activado_por() const noexcept;

    private:
        opciones_formulario opciones_formulario_;
};
```

Ejercicio 2. Implemente las clases del diagrama, declarando exclusivamente los miembros imprescindibles para implementar las relaciones.




```
#include <iostream>
#include <set>
#include <map>

class Alumno
{
    public:
        typedef std::map<Asignatura*, Grupo*> Asignatura_Grupo;

        void matriculado_en(Asignatura& A, Grupo& G);
        Asignatura_Grupo* matriculado_en() const noexcept;

    private:
        Asignatura_Grupo asignatura_grupo_;

};

class Asignatura
{
    public:

        typedef std::map<Alumno*, Grupo*> Alumno_grupo;
```

```
        typedef std::set<Grupo*> Grupos;

        Asignatura(Grupos& G)

        void tiene(Grupo& G);
        Grupo& tiene() const noexcept;

    private:
        Alumno_grupo alumno_grupo_;
        Grupos grupos_;
};

class Grupo
{
    public:
        Grupo(Asignatura& A, Profesor& P);

        void impartido(Profesor& P);
        const Profesor& impartido() const noexcept;

        void esta(Asignatura& A);
        const Asignatura& esta() const noexcept;


    private:

        Profesor* profesor_;
        Asignatura* asignatura_;
};

class profesor
{
    public:
        typedef std::set<Grupo*> Grupos_impartidos;

        void imparte(Grupo& G);
        Grupos_impartidos* imparte() const noexcept;

    private:
        Grupos_impartidos grupos_impartidos_;
};
```

Ejercicio 3. Defina los dos métodos siguientes:

- **Alumno::matriculado_en() para matricular a un alumno en una asignatura asignándole un grupo.**

- **Profesor::imparte() para vincular un profesor a un grupo.**

```cpp
void Alumno::matriculado_en(Asignatura& A, Grupo& G)
{
    asignatura_grupo_.insert(std::make_pair(&A, &G));
}

void Profesor::imparte(Grupo& G)
{
    grupos_impartidos_.insert(&G);
}
```

Ejercicio 4. Declare una clase de asociación `Alumno_Asignatura` para la relación matriculado_en. Para ello declare los atributos que considere necesarios y dos métodos `matriculado_en()` y `matriculados()`. El primero registra a un alumno en una asignatura asignándole el grupo y el segundo devuelve todas las asignaturas (y los correspondientes grupos) en que se encuentre matriculado un alumno. Declare sobrecargas de estos dos métodos para el otro sentido de la asociación.

```cpp
/*
Ántes de modelar esta clase, hay que quitar todo lo relacionado con la
asociación de estas clases, tanto en
Alumno como en Asignatura.
*/

class Alumno_Asignatura
{
    public:
        typedef std::map<Asignatura*, Grupo*> Asignatura_grupo;
        typedef std::map<Alumno*, Asignatura_grupo> AD // Asociación izq -> der
        typedef std::map<Alumno*, Grupo*> Alumno_grupo;
        typedef std::map<Asignatura*, Alumno_grupo> AI; // Asociación der -> izq

        void matriculado_en(Alumno& Alum, Asignatura& Asig, Grupo& G);
        const Asignatura_grupo* matriculados(Alumno& Alum) const noexcept;
        void matriculado_en(Asignatura& Asig, Alumno& alum, grupo& G);
        const Alumno_grupo* matriculados(Asignatura& Asig) const noexcept;

    private:
        AD directa;
        AI inversa;
};
```

Ejercicio 5. Declare una clase de asociación `Profesor_Grupo` para la relación imparte. Incluya en ella los atributos oportunos y dos métodos `imparte()` e `impartidos()`. El primero enlaza un profesor con un grupo y el segundo devuelve todos los grupos que imparte un profesor. Sobrecargue ambas funciones miembro para el sentido inverso de la relación.

```
/*
Ántes de modelar esta clase, hay que quitar todo lo relacionado con la
asociación de estas clases, tanto en
Grupo como en Profesor.
*/

class Profesor_Grupo
{
    public:
        typedef std::map<Profesor*, std::set<Grupos*>> profesor_grupo;
        typedef std::map<Grupo*, Profesor*> grupo_profesor;

        void imparte(Profesor& P, Grupo& G);
        Profesor& impartidos(Grupo& G) const noexcept;

        void imparte(Grupo& G, Profesor& P);
        std::set<Grupos*> impartidos(Profesor& P) const noexcept;


    private:
        profesor_grupo profesor_grupo_;
        grupo_profesor grupo_profesor_;
};
```

Ejercicio 6. Defina el método Alumno_Asignatura::matriculado_en() (y su sobrecarga) para matricular a un alumno en una asignatura asignándole un grupo. Esta función también permitirá cambiar el grupo al que pertenece un alumno ya matriculado en la asignatura.

```
void Alumno_Asignatura::matriculado_en(Alumno& Alum, Asignatura& Asig, Grupo& G)
{
    /* UNA FORMA DE HACERLO MÁS EXPLÍCITA
    auto alum_it1 = directa.find(&Alum);
    if(alum_it1 != directa.end())
    {
        auto asig_it1 = alum_it1->second.find(&Asig);
        if(asig_it1 != alum_it1->second.end())
            asig_it1->second = &G
        else
            alum_it1->second[&Asig] = &G
            directa[&Alum].insert(std::make_pair(&Asig, &G));
    }else
        directa.insert(sdt::make_pair(&Alum,
Asignatura_grupo({std::make_pair(&Asig, &G)}));
```

```
    auto asig_it2 = inversa.find(&Asig);
    if(asig_it2 != inversa.end())
    {
        auto alum_it2 = asig_it2->second.find(&Alum);
        if(alum_it2 != asig_it2->second.end())
            inversa[&Asig][&Alum] = &G;
        else
            inversa[&Asig].insert(std::make_pair(&Alum, &G));
    }else
        inversa.insert(sdt::make_pair(&Asig, Alumno_grupo({std::make_pair(&Alum,
&G)}));

    */

    // Hace lo mismo que lo de arriba
    directa[&Alum][&Asig] = &G;
    inversa[&Asig][&Alum] = &G;


}

void Alumno_Asignatura::matriculado_en(Asignatura& Asig, Alumno& Alum, Grupo& G)
{
    matriculado_en(Alum, Asig, G);
}
```

Ejercicio 7. Escriba la definición de Profesor_Grupo::imparte(). Si el grupo ya tiene un profesor asociado, se deberá desvincular del mismo y enlazarlo con el nuevo.

```
void Profesor_Grupo::imparte(Profesor& P, Grupo& G)
{
    Profesor* auxiliar;     // Profesor que queremos desvincular

    auto i = grupo_profesor_.find(&G);  // Busco el grupo
    if(i != grupo_profesor_.end())
    {
        auxiliar = i->second;   // Si existe el grupo, guardo el profesor
antiguo
        i->second = &P;         // Lo machaco con el nuevo
        profesor_grupo_[auxiliar].erase(&G); // Borro el grupo del conjunto de
grupos del antiguo profesor.
    }else   // esto es en el caso de que no exista el grupo, lo cual implica que
tmpoco tiene
    {       // un profesor asociado
        grupo_profesor_[&G] = &P;
        profesor_grupo_[&P].insert(&G);
    }
}

void Profesor_Grupo::imparte(Grupo& G, Profesor& P)
{
    imparte(P,G);
}
```

Ejercicio 8. Defina Profesor_Grupo::impartidos()

```
Profesor& Profesor_Grupo::impartidos(Grupo& G) const noexcept
{
    return *(grupo_profesor_.find(&G)->second);
}

std::set<Grupo*> impartido(Profesor& P) const noexcept
{
    if(profesor_grupo_.find(&P) != profesor_grupo_.end())
        return profesor_grupo_.find(&P)->second;

    return std::set<Grupo*>();
}
```

Tarea 3.2

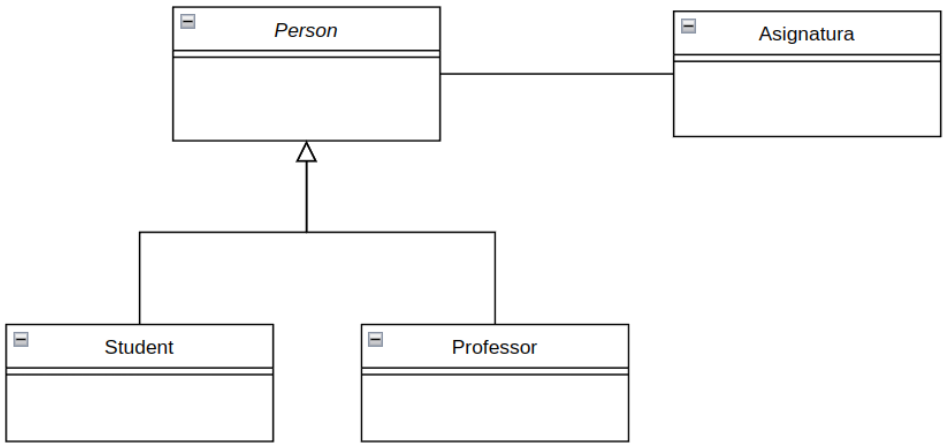
Ejercicio 1. Se dispone de una clase base Persona y dos clases especializadas Alumno y Profesor. Se quiere saber qué alumnos están matriculados en qué asignaturas y qué profesores imparten qué asignaturas, y viceversa. Para ello hay dos opciones:

- Dos asociaciones bidireccionales varios a varios, una entre Alumno y Asignatura, y otra entre Profesor y Asignatura.

- Una única asociación bidireccional varios a varios entre las clases Persona y Asignatura.

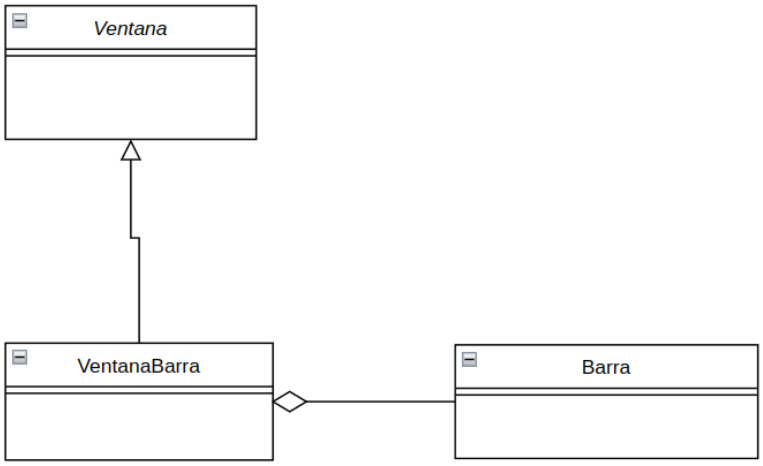
¿Cuál de los dos opciones considera más conveniente?

Considero más conveniente la segunda opción ya que, como las subclases van a heredar las relaciones de la superclase, te ahorras el tener que relacionar asignatura con Persona y Profesor. Esta decisión mejora la legibilidad, claridad y eficiencia del programa ya que te ahorras el implementar una relación por cada subclase.



Ejercicio 2. Supóngase que existen ya definidas dos clases Ventana (ventana gráfica), y Barra (barra de desplazamiento) y se quiere definir una nueva clase `VentanaBarra` (ventana con barra de desplazamiento). Indique si definiría la nueva clase utilizando alguna de las anteriores o como una nueva clase independiente. En caso de utilizar alguna de las ya definidas explique qué relaciones son las que se establecen entre ellas y cómo las codificaría. Razone la respuesta.

He decidido modelar las relaciones de la siguiente forma: `VentanaBarra` es un tipo de `Ventana`, la cual está formada por una Barra, esto es, `VentanaBarra` es una especialización de `Ventana` compuesta por `Barra`.




```
class Ventana
{
    public:
        // ...
    private:
        // ...
};

class Barra
{
    public:
        // ...
    private:
        VentanaBarra* vb_
};

class VentanaBarra: public Ventana
{
    public:
        VentanaBarra(/*atributos de ventana*/, Barra& B);
    private:
        Barra* b_;
};
```

Ejercicio 3. Dadas las clases A y B, indicar qué asignaciones son correctas:

```
class A { /* ... */ };
class B: public A { /* ... */ };

main()
{
    A objA, *pA;
    B objB, *pB;
    pA = &objA;      // Bien.
    pB = &objB;      // Bien.
    objA = objB;     // Bien, se copian solo los métodos y atributos que comunes.
    objA = (A)objB; // Bien, se puede convertir implícitamente de objB a objA.
    objB = objA;     // Error, hay métodos y atributos de objB que no
                     // se inicializarían.

    objB = (B)objA; // Error, no existe conversión explícita posible entre estas
                     // dos clases.

    pA = pB;         // Bien, se pueden convertir punteros de la clase de abajo
                     // en punteros de la de arriba.

    pB = pA;         // Error, no puedes convertir implícitamente un puntero de
                     // la clase de arriba en uno de la clase de abajo.

    pB = (B*)pA;     // Bien(si pB es un puntero a ClaseB), aunque peligroso,
                     // se recomienda un dynamic_cast.

}
```

E = Explícita. I = Implícita

Conversiones							
Entre objetos				Entre punteros			
Hacia Arriba		Hacia Abajo		Hacia Arriba		Hacia Abajo	
E	I	E	I	E	I	E	I
✓	✓	✗	✗	✓	✓	✓	✗

Ejercicio 4. Sea cierta clase base B y una derivada D. Ambas tienen definido un cierto método f(). Diga si el siguiente código es correcto y a qué método f() se llamaría.

```
B b, *bp;
D d, *dp;
b.f();      // Bien
bp = &d;    // Bien
bp->f();    // Bien, llamaría a B::f()
dp = &d;    // Bien
dp->f();    // Error, ambigüedad, ¿Qué f() uso, B::f() o D::f()?
```

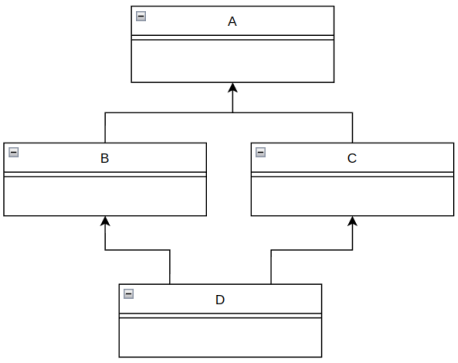
Ejercicio 5. Dadas las siguientes declaraciones :

```
struct A { int a; };
struct B : public A { int b; };
struct C : public A { int c; };
struct D : public B, public C { int d; } v;
```

¿Cuántos miembros tiene el objeto v? ¿Cómo se accede a cada uno de ellos?

Tiene en total 5 miembros.

- `v.d`
- `v.c`
- `v.b`
- `v.B::a`
- `v.C::a`



TAREA 4.1

Ejercicio 1. ¿Es correcto (compilará) el siguiente programa?

```cpp
#include <iostream>

using namespace std;

void mostrar(int i)
{
    cout << i << " [entero]" << endl;
}

void mostrar(float f)
{
    cout << f << " [real]" << endl;
}

int main()
{
    mostrar(2);   // Llama a mostrar(int)
    mostrar(2.0); // No sabe a cual llamar
    mostrar('a'); // Llama a mostrar(int)
}
```

Se producirá un error por ambigüedad en la llamada `mostrar(2.0);` ya el compilador no sabrá a que función llamar, si a `mostrar(int)` o a `mostrar(float)` ya que existen conversiones implícitas tanto de `double` a `int` como de `double` a `float`.

Ejercicio 2. ¿Qué mostrará el siguiente programa?

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

struct A
{
    void mostrar(int i) { cout << i << " [entero]" << endl; }
};
struct B: A
{
    void mostrar(float f) { cout << f << " [real]" << endl; }
};

int main()
{
    B b;
    b.mostrar(2); // Real;
    b.mostrar(2.0); // Real;
}
```

Cuando se llama a una función desde una instancia de una clase derivada, el compilador buscará primero en la clase derivada y luego en la clase base (esto se conoce como *"lookup"*), por lo que, como tanto 2 como 2.0 se pueden convertir implícitamente a un `float`, por lo que ambas llamadas imprimirán lo mismo, `2 [real]`.

Ejercicio 3. Sea cierta clase base B y una derivada D. Ambas tienen definido un cierto método f(). Diga si el siguiente código es correcto; y, si lo es, a qué método f() se llamaría, dependiendo de que B::f() sea o no virtual.

```
class B
{
    public:
        f();

    private:
        // ...
};

class D: public B
{
    public:
        f();

    private:
        // ...
};


B b, *bp;
D d, *dp;
bp = &d;
bp->f();
dp = &b;    // No se puede hacer dicha asignación.
dp->f();    // Esto daría error pues.::f().
dp = &d;
dp->f();
```

B::f()

- `bp->f();` B::f().
- `dp->f();` D::f().

virtual B::f()

- `bp->f();` D::f(), se apunta al f() del objeto al que apunta bp.
- `dp->f();` D::f().

Ejercicio 4. Indique qué enviará exactamente a la salida estándar el siguiente programa al ejecutarse:

```cpp
#include <iostream>
struct B
{
    B() { std::cout << "Constructor de B\n"; }
    virtual ~B() { std::cout << "Destructor de B\n"; }
};

struct D: B
{
    D() { std::cout << "Constructor de D\n"; }
    ~D() { std::cout << "Destructor de D\n"; }
};

int main()
{
    B *pb = new D;
    delete pb;
}
```

Imprimirá lo siguiente:

```
Constructor de B
Constructor de D
Destructor de D
Destructor de B
```

En la función `main()`, se crea un puntero de tipo `B*` que apunta a un objeto de tipo `D`, y luego se elimina el objeto a través de ese puntero, para lo cual, como es virtual el destructor de `B`, llama al destructor de `D` primero y luego al suyo

¿Qué destructores son virtuales? ¿Cambiaría algo si quitamos la palabra virtual del destructor de `B`?

Es virtual el destructor de `B`. Si quitásemos ese `virtual` del destructor, el compilador llamaría al destructor de `B`.

```
Constructor de B
Constructor de D
Destructor de B
```

Ejercicio 5. ¿Cambiaría el comportamiento de la clase cuadrado si le quitamos el miembro `area()` ?

```
class rectangulo
{
    public:
        rectangulo(double a, double l): ancho(a), largo(l) {}

        virtual double area()
        {
            return ancho * largo;
        }

        virtual ~rectangulo() = default;

    private:
        double ancho, largo;
};

class cuadrado: public rectangulo
{
    public:
        cuadrado (double l): rectangulo(l, l) {}
        double area()
        {
            return rectangulo::area();
        }
};
```

No, ya que cuadrado heredaría el método área de rectángulo.

Ejercicio 6. Dadas las siguientes clases:

```
struct A
{
    A(double valor): v(valor) {}
    void modificaV(double i) { v = v * i; }
    double v;
};

struct B: A
{
    B (double valor): A(valor), v(0.0) {}
    void modificaV(double i) { A::modificaV(i); v++; }
    double v;
};
```

¿Qué ocurriría si se realizara la siguiente modificación?

```
void B::modificaV(double i) { v++; A::modificaV(i); }
```

Pues no pasaría nada ya que cada `struct` tiene su `v` y se están incrementando.

¿Y si, además, eliminásemos el atributo `v` de la clase `B`?

En el caso en el que se le quite el atributo `B::v`, la cosa si que cambia ya que la `v` que se modifica en el método `B::modificaV()` es `A::v`, por lo que cambiaría el orden en el que se incrementa `v` y por consiguiente, el valor que esta variable tendrá en `A::modificaV(i)`.

TAREA 4.2

Ejercicio 1. Sea cierta clase base `B` y una derivada D. Ambas tienen cierto método f(), pero en `B` se define como virtual puro.

Escriba la definición de `B::f()` (no recibe parámetros ni devuelve nada).

```
void B::f() = 0;
```

La clase `B` es una clase a la que se le denomina ..., ¿cómo?

Se denomina *clase abstracta*.

¿Qué hace el fragmento de código siguiente?

```
B b, *bp;
D d;
bp = &d;
bp->f();
```

Linea por linea:

- `B b` Esta instrucción es incorrecta ya que no se puede instanciar la clase B.
- `B *bp` Se crea un puntero a un tipo B.
- `D d` Se crea un objeto d de tipo D.
- `bp = &d` Se asigna la referencia de d a bp.
- `bp->f()` Se llama a D::f() ya que B::f() es virtual pura y, por consiguiente ha de estar anulada por otra D::f().

Ejercicio 2. Consideremos la siguiente jerarquía de clases:

```
struct V
{
    virtual void fv() = 0;
    virtual ~V() {}
};

struct X : V
{
    void fv() {}
};

struct Y : V
{
    void fv() {}
};
```

```
struct Z : V
{
    void fv() {}
};

void f(V& v)
{
    if (typeid(v) == typeid(X))
    {
        std::cout << "Procesando objeto X...\n";
        // código específico para X
    }

    if (typeid(v) == typeid(Y))
    {
        std::cout << "Procesando objeto Y...\n";
        // código específico para Y
    }
    if (typeid(v) == typeid(Z))
    {
        std::cout << "Procesando objeto Z...\n";
        // código específico para Z
    }
}
```

¿Existe una relación de realización entre las clases presentadas? ¿Por qué?

Sí, debido a que la clase `V` es una interfaz, y todos sus métodos virtuales puros están sobrescritos en sus clases derivadas.

¿Cuál es la salida del siguiente código?

```
X x; // Se crea un objeto de x
V* pv = new Y; // Se crea un puntero a V el cual contiene nu objeto Y
f(x);
f(*pv);
```

Salida:

```
Procesando objeto de X...
Procesando objeto Y...
```

¿Es la mejor forma de implementar el comportamiento polimórfico de f()? Razone la respuesta. En caso negativo, describa cómo mejorar la implementación y, si es necesario, modifique el código anterior para que produzca la misma salida.

No, no es la mejor forma ya que dicha función, junto con su comportamiento, no son necesarios. Aquí la nueva implementación.

```
struct V
{
    virtual void fv() = 0;
    virtual ~V() {}
};
```

```
struct X : V
{
    void fv() { std::cout << "Procesando objeto X...\n"; }
};

struct Y : V
{
    void fv() { std::cout << "Procesando objeto Y...\n"; }
};

struct Z : V
{
    void fv() { std::cout << "Procesando objeto Z...\n"; }
};
```

Ejercicio 3. Defina una clase paramétrica llamada `Buffer` para representar una zona de memoria, cuyos parámetros sean el tipo base de cada elemento de esa zona (por omisión, el tipo cuyo tamaño es 1 byte), y el tamaño de dicha zona (por omisión, 256 elementos). Defina dentro de la clase el atributo principal, que será un vector paramétrico (de la STL), y el constructor predeterminado.

```
template <class T = byte, size_t n = 256>
class Buffer
{
    public:
        Buffer();

    private:
        std::vector<T> vector_;
};

template <class T, size_t n>
Buffer<T,n>::Buffer(){}
```

A continuación defina un objeto de tipo `Buffer` formado por 128 elementos de tipo `int`, y otro formado por 256 elementos del tipo por omisión.

```
int main()
{
    Buffer<int,128> B1;
    Buffer<> B2;

    return 0;
}
```

MURPHY