Exámenes y Ejercicios Parcial 2
Programación Orientada a Objetos

Índice general

1. Seminarios Resueltos

1.1. Seminario 3.1: Relaciones I

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.

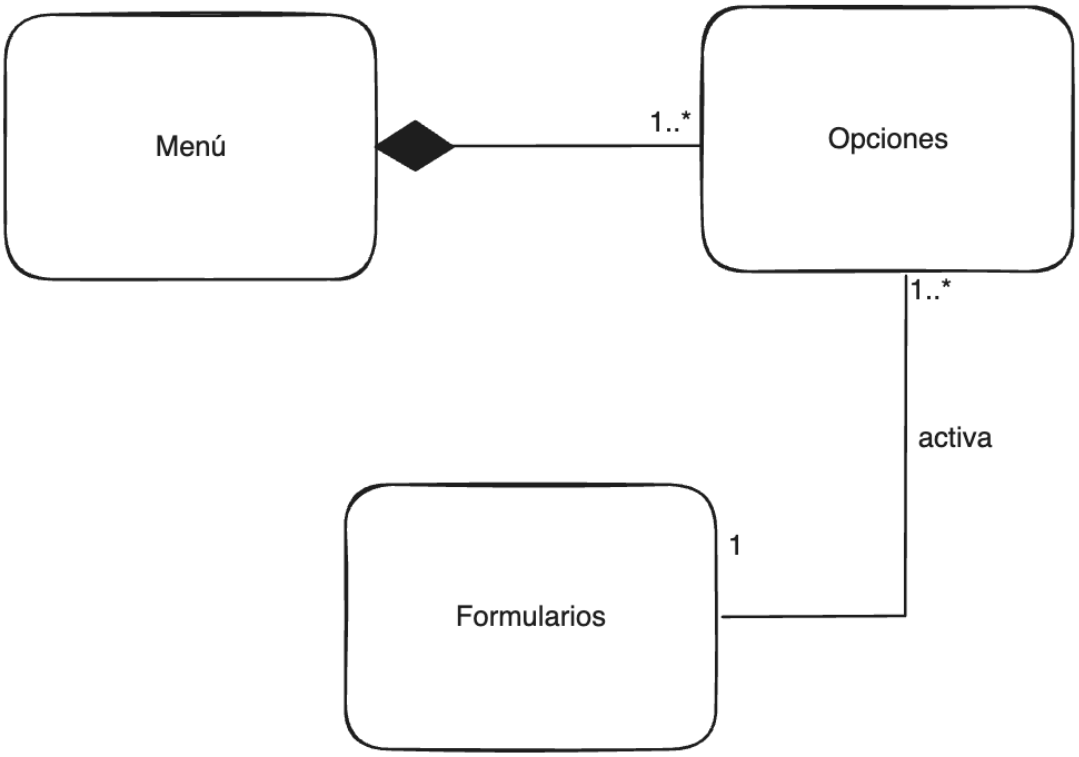


Figura 1.1: Solución diagrama de clases

Implementación de las clases del diagrama de clases:

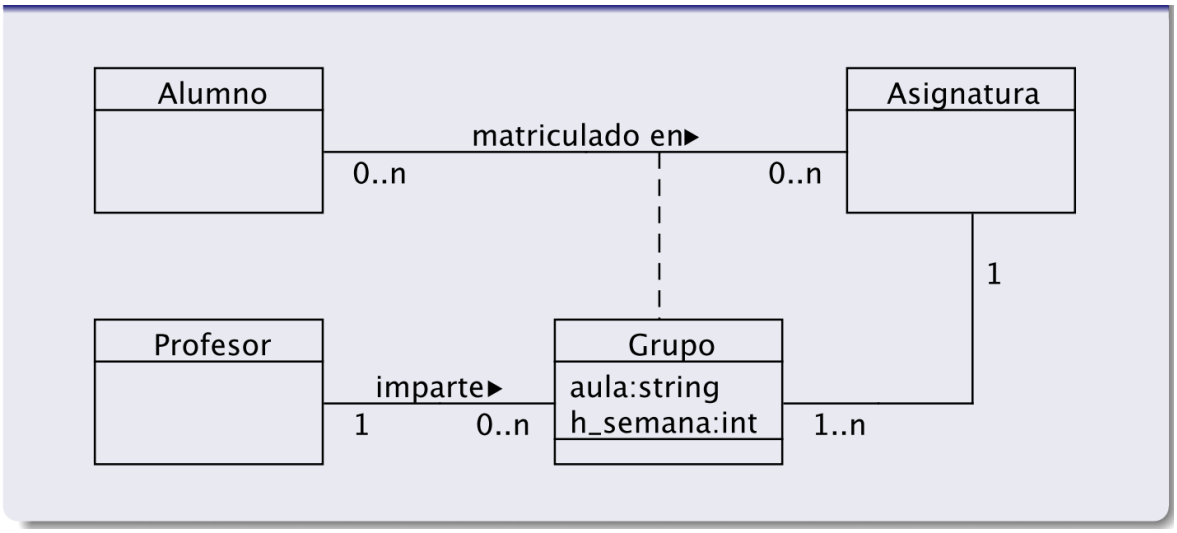
```
class Menu{
  public:
    typedef std::set<Opciones>ConjuntoOpciones;
    Menu(Opciones& opcion){}
    void setOpciones(Opciones& o)noexcept;
  private:
    ConjuntoOpciones opciones_;
};
```

```
class Opciones{
  public:
    Opciones(Formularios& f):formulario_(&f){}
    const Formularios& getFomulario()const noexcept{return *formulario_;}

    //Necesario para la composición
    friend bool operator < (const Opciones& op1, const Opciones& op2)noexcept{
      return op1.id_ < op2.id_;
    }
  private:
    Formularios* formulario_;
    int id_; //atributo que ordena opciones
};

class Fomularios{
  public:
    typedef std::set<Opciones*>COpciones;
    Fomularios(Opciones& opcion){
      setOpciones(opcion);
    }
    void setOpciones(Opciones& o){
      opciones_.insert(&o);
    }
    const COpciones& getOpciones()const noexcept{return opciones_;}
  private:
    COpciones opciones_;
};
```

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



```
class Alumno{
  public:
    //Alias del diccionario Asignatura - Grupo
    typedef std::map<Asignatura*,Grupo*>Asignaturas;
    void matriculados_en(Asignatura&, Grupo& )noexcept;
    const Asignaturas& getAsignaturas()const noexcept;
  private:
    Asignaturas asignaturas_;
};
class Asignatura{
  public:
    //Alias del diccionario Alumno - Grupo
    typedef std::map<Alumno*,Grupo*>Alumnos;
    void matricula(Alumno& , Grupo&)noexcept;
    const Alumnos& getAlumnos()const noexcept;
    //Alias del conjunto de Grupos de cada Asignatura
    typedef std::set<Grupo*>Grupos;
    void setGrupo(Grupo& )noexcept;
    const Grupos& getGrupo()const noexcept;
  private:
    Alumnos alumnos_;
    Grupos grupos_;
};
class Profesor; //Declaración adelantada
class Grupo{
  public:
    Grupo(std::string a, int h, Asignatura& asig,Profesor& p):
    ↪  aula(a),h_semana(h),asignatura_(&asig),profesor_(&p){}
  private:
    std::string aula;
    int h_semana;
    Asignatura* asignatura_;
    Profesor* profesor_;
};
class Profesor{
  public:
    //Alias del conjunto de Grupos de un profesor
    typedef std::set<Grupo*>Grupos;
    void imparte(Grupo&)noexcept;
    const Grupos& getGrupos()const noexcept;
  private:
    Grupos grupos;
};
```

Ejercicio 3: Defina los dos métodos siguientes:

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

```
void Alumno::matriculados_en(Asignatura& asig, Grupo& g)noexcept{
  asignaturas_.insert(std::make_pair(&asig,&g));
```

```
}
```

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

```
void Profesor::imparte(Grupo& g)noexcept{
  grupos_.insert(&g);
}
```

Ejercicio 4: Declare una clase de asociación Alumno_Asignatura para la relación matriculado_en. Para ello declare los atributos que considere necesarios, 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.

```
class Alumno_Asignatura{
  public:
    //Alias de los diccionarios de Alumno y Asignatura con Grupos
    typedef std::map<Alumno*,Grupo*> Alumnos;
    typedef std::map<Asignatura*,Grupo*>Asignaturas;
    //Alias de los diccionarios de la relacion
    typedef std::map<Alumno*,Asignaturas> Alum_Asig;
    typedef std::map<Asignatura*,Alumnos>Asig_Alum;
    //Relaciona las clases
    void matriculados_en(Alumno&, Asignatura&, Grupo&)noexcept;
    void matriculados_en(Asignatura&, Alumno&, Grupo&)noexcept;
    //observadores de la clase
    Alumnos matriculados(Asignatura& )const noexcept;
    Asignaturas matriculados(Alumno& )const noexcept;
  private:
    Alum_Asig directa_;
    Asig_Alum 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.

```
class Profesor_Grupo{
  public:
    //Alias de los diccionarios de la relacion
    typedef std::set<Grupo*>Grupos;
    typedef std::map<Profesor*,Grupos>Profe_Grupo;
    typedef std::map<Grupo*,Profesor*>Grupo_Profe;
    //relaciona las clases
    void imparte(Profesor&, Grupo&)noexcept;
    void imparte(Grupo&,Profesor&)noexcept;
    //Observadores de la clase
    Grupos impartidos(Profesor& )const noexcept;
```

```
    const Profesor* impartidos(Grupo& )const noexcept;
  private:
    Profe_Grupo directa_;
    Grupo_Profe inversa_;
};
```

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::matriculados_en(Alumno &alumno, Asignatura& asig,
↪  Grupo& grupo)noexcept{
  directa_[&alumno].insert(std::make_pair(&asig,&grupo));
  inversa_[&asig].insert(std::make_pair(&alumno,&grupo));
}
void Alumno_Asignatura::matriculados_en(Asignatura& asig, Alumno& alumno,
↪  Grupo& grupo)noexcept{
matriculados_en(alumno,asig,grupo); //delegamos en el método anterior.
}
```

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& prof, Grupo& grupo)noexcept{
  directa_[&prof].insert(&grupo);
  inversa_[&grupo]=&prof;
}
void Profesor_Grupo::imparte(Grupo& grupo, Profesor& prof)noexcept{
  imparte(prof,grupo);
}
```

Ejercicio 8: Defina **Profesor_Grupo::impartidos()**.

```
Profesor_Grupo::Grupos Profesor_Grupo::impartidos(Profesor& p)const noexcept{
  //Buscamos si el profesor está relacionado
  auto i = directa_.find(&p);
  if(i!=directa_.end()) return i->second;
  else{
    Profesor_Grupo::Grupos vacio;
    return vacio;
    //return std::set<Grupo*>();
  }
}

const Profesor* Profesor_Grupo::impartidos(Grupo& g)const noexcept{
  auto i = inversa_.find(&g);
  if(i!=inversa_.end()) return i->second;
  else{
    return nullptr;
  }
}
```

1.2. Seminario 3.2: Relaciones II

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.

Como queremos que solamente accedan a las asignaturas ya sean Alumnos o Profesores si lo implementamos mediante la segunda manera cualquier Persona que no sea ni Alumno ni Profesor podrá acceder a las asignatura, como eso es algo que no queremos lo implementaremos mediante la primera manera.

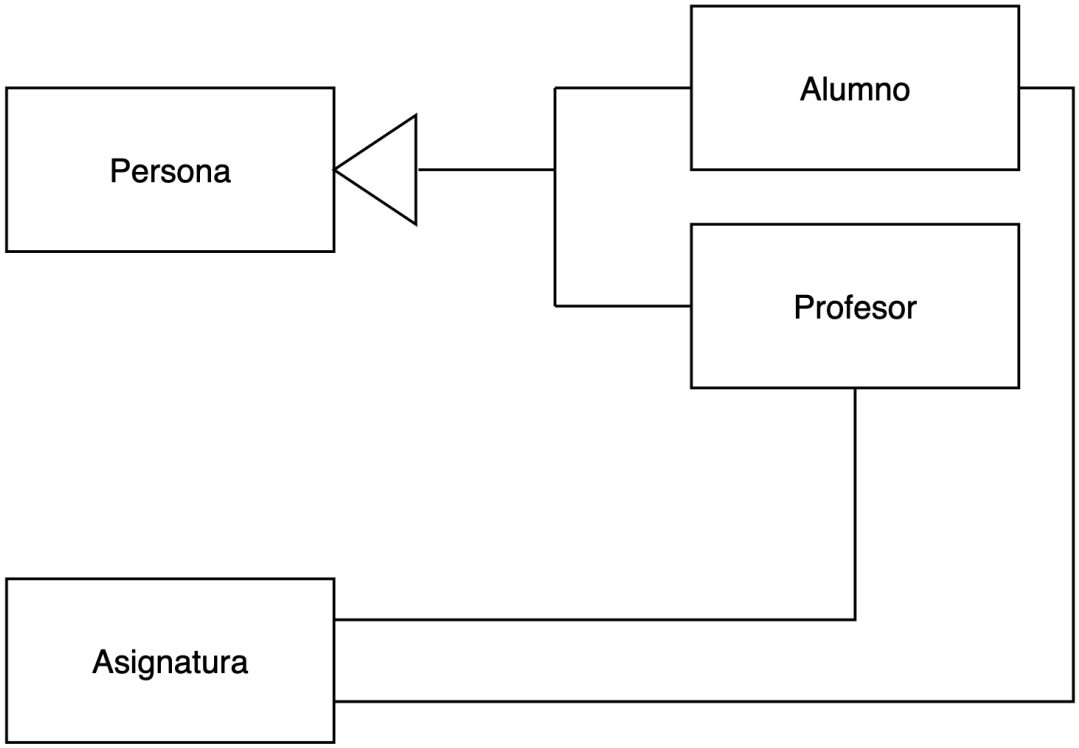


Figura 1.2: Resultado de la implementación del ejercicio

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.

Una VentanaBarra estará compuesta por una Ventana y una Barra, por tanto, podriamos implementarla como dos composiciones 1-1:

```
class Ventana{};
class Barra{};
class VentanaBarra : public Ventana, private Barra{
  public:
    VentanaBarra(Ventana& v, Barra& b):Ventana(v),Barra(b){}
};
```

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

```
class A{};
class B: public A {};
int main() {
  A objA, *pA;
  B objB, *pB;
  pA = &objA;
  pB = &objB;
  objA = objB;
  objA = (A)objB;
  objB = objA;
  objB = (B)objA;
  pA=pB;
  pB=pA;
  pB = (B*)pA;
}
```

Asignaciones:

pA = &objA; → Correcta, puntero pa de tipo A apunta a un objeto de A.

pB = &objB; → Correcta, puntero pb de tipo b apunta a un objeto de B.

objA = objB; → Correcta, conversión de un objeto B a uno A (conversión implícita hacia arriba.)

objA = (A)objB; → Correcta, conversión de un objeto B a uno A (conversión explícita).

objB = objA; → Error, no se pueden convertir objeto de A a B de forma implícita.

objB = (B)objA; →Error, no se pueden convertir objetos de A a B de forma explícita.

pA=pB; →Correcto, conversión de puntero de B a A implícitamente.

pB=pA; → Error, no se puede convertir un puntero de A a B implicitamente.

pB = (B*)pA; → Correcta, conversión explicita de puntero de A a uno de B (asignación de punteros del mismo tipo).

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();

bp=&d;
bp->f();

dp=&d;
dp->f();
```

Método f() que se llamaría:
`b.f();`→ Vemos que b es un objeto de B (base), se llama a f() de B.
`pb ->f();`→ Vemos que previamente una referencia de un objeto D (derivada) es apuntada por un puntero B (base), como la clase Base no es polimorfica nos fijamos en el tipo del puntero, por tanto , se llama a f() de B.
`dp -> f();`→ Vemos que dp es un puntero que apunta a la referencia de un objeto de D, como no cambia de tipo, se llama a f() de D.

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;
```

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

El objeto v tiene 5 miembros, y se acceden de la manera: `d.v`, `c.v`, `b.v`,`v.B::a`, `v.C::a`.

Vemos que tiene dos miembros de la clase A repetidos, esto crea una ambigüedad, por tanto, para solucionar esto hacemos que tanto la clase B como C herenden virtualmente de A, haciendo que tenga solamente 4 miembros. Los dos últimos desaparcen y al miembro de la clase A se accedería de la manera `v.a`.

1.3. Seminario 4.1: Polimorfismo I

Ejercicio 1: Dado el siguiente código:

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

¿Qué mostrarán cada una de las siguientes llamadas a mostrar?

```
A a;
a.mostrar(4);
a.mostrar(4.1);
```

La clase A tiene dos métodos mostrar (uno para int y otro para float), por tanto:
`a.mostrar(4);`→ 4[A-entero]
`a.mostrar(4.1);`→ 4.1[A-real]

```
B b;
b.mostrar(4);
b.mostrar(4.1);
```

La clase B tiene un método mostrar (float), por tanto:
`b.mostrar(4);`→ 4[B-real]
`b.mostrar(4.1);`→4.1[B-real]

Vemos que en el caso de llamar `b.mostrar(4);`, este va a devolver 4[B-real], debido a que no tiene definido un mostrar en dicha clase y hace uso del mostrar con float donde realiza una conversión implícita a dicho tipo de dato. Si definimos un mostrar con entero en B, devolvería ese y no el de float.

Ejercicio 2: 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.

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

B::f() no virutal, B no es polimorfica:
`bp->f();`→ Conversión hacia arriba previa, se llama a B::f().
`dp->f();`→Error previo a la hora de realizar una conversión hacia abajo. Si se hace bien la conversión se llama a D::f().
`dp->f();`→ Se llama a D::f().

B::f() virutal, B es polimorfica:
`bp->f();`→ Se llama a D::f().
`dp->f();`→ Si se hace la conversión correctamente, se llama a B::f().
`dp->f();`→ Se llama a D::f().

Ejercicio 3: Indique qué enviará exactamente a la salida estándar el siguiente programa al ejecutarse:

```
#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;
}
```

¿Que destructores son virtuales? ¿Cambiaría algo si quitamos la palabra virtual del destructor B?

El único destructor que es virtual es el de la clase B. Si quitamos la palabra virtual no se llamará al destructor de la clase Derivada, haciendo que no se elimine correctamente el objeto de la misma clase.

Ejercicio 4: ¿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, no cambiaría nada porque al quitar dicho miembto hará uso del método area () de la clase rectangulo (base), es el mismo comportamiento si no quitamos el método ya que en cuadrado se redefine delegando en el método area() de rectangulo.

1.4. Seminario 4.2: Polimorfismo II

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

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

Como dice que no devuelve nada sabemos que es de tipo void, y como es virtual pura tiene que terminar con "=0', entonces la definición queda como:

```
virtual void f() = 0;
```

b) ¿Qué nombre reciben las clases que son como la clase B?

Las clases que tiene al menos un método virtual puro se denominan clases abstractas, si es el único método que tienen y además es virtual puro se denominan interfaces.

c) ¿Qué hace el fragmento de código siguiente? ¿Es correcto?

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

Primero de todo, vemos que la clase B al ser una clase abstracta no puede instanciarse (crear objetos), por tanto, encontramos un error en `B b`. Luego se realiza una conversión hacia arriba implícitamente y se llama al método `f()` de D, ya que B es abstracta y por ende polimórfica.

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() { /* ... */ }
};
```

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

Vemos que la clase V es una clase abstracta por lo que si vemos que hay una relación de realización cuando las clases X e Y heredan de ella (V).

Supongamos una función `f()` para procesar objetos de esa jerarquía:

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

b) ¿Cuál es la salida del siguiente código?

```
X x;
V* pv = new Y;
f(x);
f(*pv);
```

Se realiza una conversión hacia arriba de forma implícita, se llama al método f(x) y devuelve "Procesando objeto X..." y luego se llama al método f(*pv), es decir, con el contenido al que apunta el puntero pv y devuelve "Procesando objeto Y...'

c) ¿Es la mejor forma de implementar el comportamiento polimórfico de `f()`? En caso negativo, modifique el código anterior para mejorarlo y para que produzca la misma salida.

No, no es la mejor manera ya que mediante polimorfismo podemos redefinir métodos que son virtuales, pero lo que estamos haciendo es comprobar el tipo de dato que se recibe por parámetro y mediante una selectiva devolver una cosa u otra.
Modificación para hacer uso de polimorfismo:

```
struct V {
  virtual void fv() = 0;
  virtual ~V() {}
};
struct X : V {
  virtual void fv() override {std::cout << "Procesando objeto X...\n";}
};
struct Y : V {
  virtual void fv() override {std::cout << "Procesando objeto Y...\n"; }
};
```

Así cada vez que accedamos desde un puntero V al método `fv()` se llamará a aquel método que corresponda con el tipo del objeto al que apunta, ya que V es asbtacta y por ende polimorfica como comentamos antes.

Ejercicio 3:

a) 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).
- El tamaño de dicha zona (por omisión, 256 elementos).

b) Defina dentro de la clase el atributo principal, de nombre buf, que será un vector paramétrico (de la STL).

c) Añada el constructor predeterminado. Este ha de invocar el constructor de la clase vector para que inicialice el atributo, creando tantas instancias por defecto del tipo base como indique el tamaño de la zona de memoria. Para esto, utilizaremos el constructor que toma un size_type.

```
template <typename T = byte, size_t n = 256> //Apartado 1
class Buffer{
    std::vector<T>buf; //Apartado 2
  public:
    Buffer():buf(n){} //Apartado 3
};
```

Ejercicio 4: Suponga que a la clase Buffer le añadimos los siguientes métodos:

```
void Buffer<T,n>::almacenarDato(char dato, size_t indice){
  if(indice < buf.size())
    buf[indice] = dato;
  else
    throw std::out_of_range("Almacenar: fuera de rango");
}
char Buffer<T,n>::recuperarDato(size_t indice) const {
  if(indice < buf.size())
    return buf.at(indice);
  else
    throw std::out_of_range("Recuperar: fuera de rango");
}
```

a) ¿Qué problemas observas? ¿Falta algo en este código? Haga los cambios oportunos en el código proporcionado.

Vemos que en el método `almacenarDato` está intentando guardar un char cuando el tipo de dato es T, es decir tendría que poner T dato ya que estamos haciendo uso de una plantilla.

Lo mismo sucede en el método `recuperarDato` donde devuelve un char y debería de devolver un T, ya que el método tiene que valer para cualquier tipo de dato.

```
template<T,size_t n>
void Buffer<T,n>::almacenarDato(T dato, size_t indice){
  if(indice < buf.size())
    buf[indice] = dato;
  else
    throw std::out_of_range("Almacenar: fuera de rango");
}
template<T,size_t n>
T Buffer<T,n>::recuperarDato(size_t indice) const {
  if(indice < buf.size())
    return buf.at(indice);
  else
    throw std::out_of_range("Recuperar: fuera de rango");
}
```

b) Cree una clase Almacenamiento, la cual siempre estará formada por una memoria de 128 elementos de tipo double. Por lo tanto, el siguiente programa imprimiría "Almacenar: fuera de rango".

```
int main() {
  Almacenamiento almacen;
  try{
    almacen.almacenarDato(4.0, 129);
    std::cout << almacen.recuperarDato(1);
  }catch(std::out_of_range& oor){
    std::cerr << oor.what() << std::endl;
  }
}
```

```
class Almacenamiento : public Buffer<double,128>{
};
```

c) Defina e implemente la clase Almacenamiento para que, haciendo uso de la clase Buffer, se ajsute a lo anterior.

```
class Almacenamiento : public Buffer<double,128>{
  public:
    Almacenamiento():Buffer<double,128>(){}
};
```

2. Examenes Resueltos

En este apartado vas a encontrar solamente los ejercicios (tendrán el número del ejercicio en dicho examen) de exámenes que se corresponden a los temas de relaciones entre clase de objetos y polimorfismo.

2.1. Examen Junio 2004

Ejercicio 3: Las Matemáticas enseñan que una circunferencia es una especie de elipse (degenerada), donde sus dos radios, r_x y r_y, son iguales. Consideremos este principio para desarrollar, mediante orientación a objetos, una hipotética aplicación en C++. Hay que encontrar una forma conveniente de representar estas figuras geométricas en un modelo de objetos.

Tras un primer análisis se identifican dos clases: Elipse y Circunferencia. La primera dispondrá de dos métodos observadores, `radio_x()` y `radio_y()`, mientras que la segunda dispondrá de un método observador, `radio()`. En ambas clases existirá un método `escala()` que recibirá un factor de escala y modificará los radios en dicho factor. Por último, la clase 'Elipse' poseerá una sobrecarga de `escala()` que recibirá dos factores de escala independientes: uno para r_x y otro para r_y.

a) ¿Qué relación podemos establecer entre elipses y circunferencias para poder implementar las segundas a partir de las primeras siguiendo el análisis anterior? Escriba todo lo necesario para implementar dicha relación.

b) ¿Tras un análisis más profundo, consideramos que las figuras planas son una abstracción útil de las elipses, circunferencias y otras figuras geométricas. Toda figura plana constaría de una operación de escalado para incrementar o disminuir su tamaño sin alterar su aspecto. ¿Qué relación podemos establecer entre figuras planas, elipses y circunferencias? Escriba todo lo necesario para implementar dicha relación.

Tras completar las figuras planas con una operación de dibujo, nos damos cuenta de que una serie de figuras planas pueden agruparse en una escena que poseería sus propias operaciones de dibujo y escalado, consistentes en aplicar la operación correspondiente a cada una de las figuras que la integran. ¿Qué relación podemos establecer entre escenas y figuras planas? Escriba todo lo necesario para implementar dicha relación.

Al ser una circunferencia una elipse degenerada donde el radio_x y el radio_y donde ambas son iguales, podemos relacionar las clases mediante una relación de composición o (herencia privada) ya que podemos delegar las operaciones escala y radio que se realiza en Elipse.

```
class Elipse{
  public:
    Elipse(double x, double y):x_(x),y_(y){}
    inline double radio_x()const noexcept{return x_;}
    inline double radio_y()const noexcept{return y_;}
    //Como lo hará uso Circunferencia, lo hacemos virtual, redefinimos
    inline virtual void escala(size_t factorescala){
          x_ *= factorescala;
          y_*=factorescala;
    }
```

```
    inline void escala(size_t factorescalax , size_t factorescalay){
          x_ *= factorescalax;
          y_ *= factorescalay;
    }
    virtual ~Elipse() = default; //al ser polimorifica, virtual
private:
    double x_,y_;
};

class Circunferencia: private Elipse{
  public:
    Circunferencia(double radio):Elipse(radio,radio){}
    inline double radio()const noexcept{return Elipse::radio_x();}
    //delegamos en el método escala de Elipse, debido a la herencia
    inline void escala(size_t factorescala)override{
      return Elipse::escala(factorescala);}
};
```

Como nos dice que es una abstracción de muchas figuras geométricas, podemos implementar la clase FiguraPlana como una interfaz, donde las clases derivadas haran uso de esos métodos polimórficos.

```
class FiguraPlana{
  public:
    virtual void escala(size_t) = 0;
    virtual void dibujo()const = 0;
    virtual ~FiguraPlana() = default; //polimorfica, virtual
};
```

Una escena va a contener un conjunto de figuras planas, como minimo 1, como máximo muchas, por tanto, una escena no va a existir si no hay figuras planas existentes (Elipse, Circunferencia,..).

Además como FiguraPlana es una clase abastracta, esta no va a poder instanciarse por lo que la opción de composición está descartada y la imlementaremos como una agregación, donde Escena contendrá un conjunto de punteros de tipo FiguraPlana:

```
class Escena{
  public:
    typedef std::set<FiguraPlana*>Figuras;
    Escena(FiguraPlana& f){figuras_.insert(&f);}
    void dibuja()const{
        for(const auto& f : figuras_)
         f -> dibujo();
    }
    void escala(size_t factor){
        for(auto f : figuras_){
          f->escala(factor);//aplicamos la escala a cada figura.
        }
    }
  private:
    Figuras figuras_;
```

};

Ejercicio 4: Se trata de implementar en C++ una función genérica, `ordenado()`, que decida si los elementos de un rango de iteradores aparecen en un cierto orden. Para lograr la máxima generalidad, los únicos requisitos que la función impondrá es que los iteradores sean de entrada y que la relación de orden tenga las propiedades de un orden estricto débil. También hay que tratar adecuadamente los rangos vacíos, que se suponen ordenados, ya que no contienen elementos.

a) Escriba una versión que compruebe el orden respecto del operador < del tipo en cuestión.

```
template <typename Iter>
bool ordenado(Iter inicio, Iter fin)
{
  if(inicio == fin) return true; //vacio = ordenado
  Iter siguiente = inicio;
  siguiente ++;

  while(siguiente != fin)
  {
    if(*siguiente <*inicio) return false; // no ordenado
    else
    inicio++;
    siguiente++;

  }
  return true; //ordenado
}
```

b) Escriba una versión que compruebe el orden respecto de una función de comparación arbitraria.

```
template <typename Iter, typename Comparador>
bool ordenado(Iter inicio, Iter fin, Comparador comparador)
{
  if(inicio == fin)return true; //vacio=ordenado
  Iter siguiente = inicio;
  siguiente++;

  while(siguiente!=fin)
  {
      if(!comparador(*inicio,*siguiente))
          return false;
      inicio++;
      siguiente++;
  }
  return true;
}
```

c) Escriba un fragmento de código que compruebe si un vector de bajo nivel de enteros está en orden ascendente de valores absolutos. Emplee un objeto función. Lo haremos mediante un objeto a función:

```
struct ValoresABS{
  bool operator()(int a, int b){
    return abs(a) < abs(b);
  }
};

int main()
{
  //Creamos el vector de enteros
  std::vector<int>valores = {1, 4, -45, 3};
  std::vector<int>valores2 = {1,2,-45,100};
  bool ascendente = ordenado(valores.begin(),valores.end(),ValoresABS());
  bool ascendente2 = ordenado(valores2.begin(),valores2.end(),ValoresABS());

  if(ascendente2)std::cout<<"Ordenado ascendentemente";
  else std::cout<<"No ordenado ascendentemente";
  return 0;
}
```

2.2. Examen Junio 2005

Ejercicio 4: Se trata de implementar en C++ una función genérica, `ordenado()`, que decida si los elementos de un rango de iteradores aparecen en un cierto orden. Para lograr la máxima generalidad, los únicos requisitos que la función impondrá es que los iteradores sean de entrada y que la relación de orden tenga las propiedades de un orden estricto débil. También hay que tratar adecuadamente los rangos vacíos, que se suponen ordenados, ya que no contienen elementos.

a) Escriba una versión que compruebe el orden respecto de una funcion de comparación.

b) Escriba un fragmento de código que compruebe si un vector de bajo nivel de enteros está en orden ascendente de valores absolutos. Emplee un objeto función.

```
  struct Comparador{
    bool operator ()(size_t a, size_t b) const {return a<b;}
};

//ordenado (pos1, pos2, funcion());
template <typename iterador, typename Comparador>
bool ordenado(iterador inicio, iterador fin, Comparador c){
    //Comprobamos somos iguales
    if(inicio == fin) return true;
    iterador sig = inicio;
    sig++;
    if(sig != fin){
        //Me comparo con el siguiente
        if(!c(*inicio,*sig)) return false;
        else{
            sig ++;
            inicio ++;
        }
    }
    return true;
};

int main(){
  size_t v[] = {1,2,5,33,9,112};
  //Creamos el objeto a función
  Comparador compara;
  for(size_t i = 0 ; i<6; i++){
      if(ordenado(&v[i],&v[i+2],compara)){
        cout << "Esta ordenado"<<endl; //devuelve true si está ordenado o no.
      }else{
        cout<<"No ordenado"<<endl;
      }
  }
  return 0;
}
```

2.3. Examen Mayo 2008

Ejercicio 1: Dado el siguiente diagrama, defina las clases que aparecen escribiendo los miembros imprescindibles para implementar las relaciones dadas:

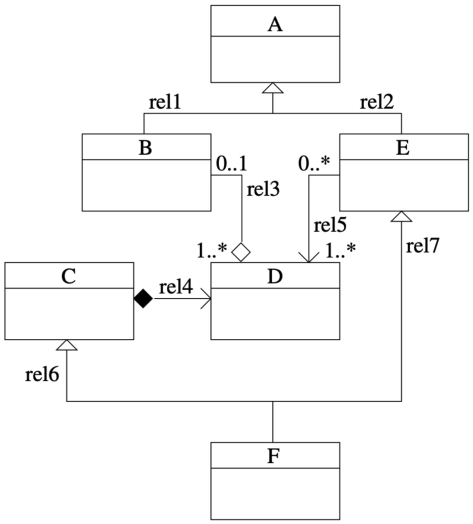


Figura 2.1: Diagrama de clases del ejercicio

```
class A{//...};
class B : public A{
  public:
    B(A& a):A(a){} //rel1
};
class C : private D{}; //rel4, composición 1-1, herencia privada
class D{
  public:
    //relacion rel3 (agregación con B), solo conoce el agregado
    void rel3(B&)noexcept;
    const B& getBs()const noexcept;
  private:
    B bs_; //rel3
};

class E : public A{ //rel1
  public:
    //Alias de la relacion con D, es unidireccional
    //multiplicidad 1-muchos, la existencia depende
    E(D& d,A& a):A(a){
      ds_.insert(&d);
    }
    typedef std::set<D*>Ds;
    void setD(D& )noexcept;
    const Ds& getDs()const noexcept;
  private:
    Ds ds_;
};
```

```
class F : public C, public E{
  public:
    F(C& c, E& e):C(c),E(e){}
};
```

Ejercicio 2: Dada las siguientes definiciones de las clases:

```
using namespace std;

class X{
  public:
    X(string s = "por omision"){
      cout<<"Constructor de X: "<<s<<endl;
      }
};

class A{
    X x;
  public:
    A():x("A"){cout<<"Constructor de A"<<endl;}
    void f(){cout<<"Metodo f() de A"<<endl;}
};
class B : virtual public A{
    X x;
  public:
    B() {cout<<"Constructor de B"<<endl;}
    void f(){cout<<"Metodo f() de B"<<endl;}
};
class C : virtual public A{
    X x;
  public:
    C(){cout<<"Constructor de C"<<endl;}
    void f(){cout<<"Metodo f() de C"<<endl;}
};
class D : public B, public C{
    X x;
  public:
    D(): x("D"){cout<<"Constructor de D"<<endl;}
};
```

Apartado a) ¿Cuántos atributos y métodos tiene la clase D?

La clase D tiene 4 atributos y tiene 3 métodos f().

Apartado b) ¿Hay algún miembro duplicado?

Gracias a la herencia virtual solucionamos el problema de la ambigüedad, haciendo que no hayan miembros repetidos.

Apartado c) ¿Cómo se accede a cada uno de los miembros? No podemos acceder a los atributos ya que estos son privados, pero podemos acceder a los métodos de la manera → `d.D::f()`, `d.B::f()` y `d.A::f()`.

A continuación, considere el siguiente programa que incluye las definiciones:

Apartado d) Diga si hay ambigüedades en la función main(). En tal caso, resuélvalas.

```
int main(){
  A *pa;
  B *pb;
  D d, *pd;

  pd = &d; //OK, apuntamos a un objeto del mismo tipo
  pa = &d; //OK, conversion de puntero de A - D implicitamente
  pa -> f(); //OK, metodo f() de A (A no polimorfica, se mira tipo
     puntero)
  pb = &d; //OK, conversion de puntero de B - D implicitamente
  pb -> f(); //OK, metodo f() de B (B no polimorfica, se mira tipo
     puntero)
  // d= *pa; //ERROR, conversion implicita de objetos hacia abajo.
  pd = (D*)pb; //OK, conversion correcta explicita de B - D
  pd -> B::f(); //OK, metodo f() de B (resolucion de ambito)
  d.C::f(); //OK, metodo f() de C (resolucion de ambito)
  return 0;
}
```

Vemos que obtenemos un error en la linea `d = *pa` ya que estamos haciendo una conversión implícita hacia abajo, no podemos asignar un objeto de tipo A a uno de tipo D de forma implícita.

Apartado f) Diga lo que imprimiría el programa una vez subsanadas las ambigüedades y los demás errores.

```
Constructor de X: A
Constructor de A
Constructor de X: por omision
Constructor de B
Constructor de X: por omision
Constructor de C
Constructor de X: D
Constructor de D
Metodo f() de A
Metodo f() de B
Metodo f() de B
Metodo f() de C
```

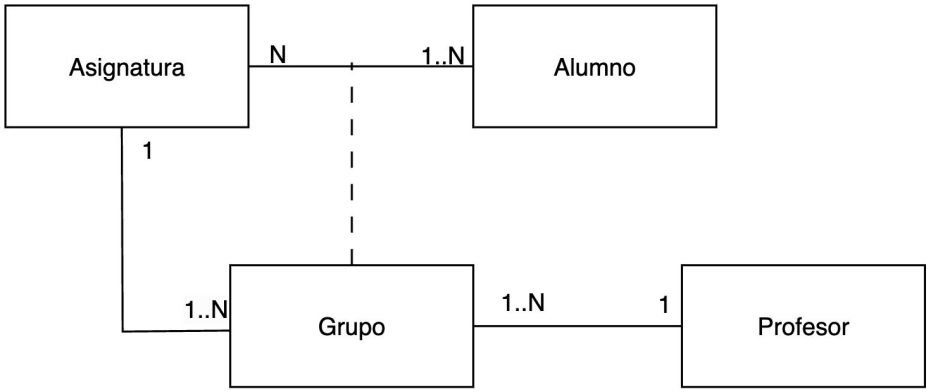
Esto obtenemos si omitimos el error comentado anteriormente.

2.4. Examen Junio 2008

Ejercicio 2: Un centro de enseñanza desea gestionar los alumnos matriculados en las diferentes asignaturas que ofrece y los profesores que las imparten. Las restricciones serán las siguientes:

- Las asignaturas se organizan en grupos de alumnos de tal forma que cada grupo pertenece a una única asignatura.
- Cada alumno pertenece a un grupo de cada asignatura en la que está matriculado.
- Los profesores podrían impartir clase en distintos grupos de la misma o diferentes asignaturas. Cada grupo sólo será impartido por un profesor.

a) Identifique las clases y las relaciones que se establecen entre ellas. Dibuje el diagrama de clases.



b) Implemente las clases que aparecen, escribiendo exclusivamente los miembros imprescindibles para implementar las relaciones.

```
class Alumno; //declaración adelantada
class Grupo; //declaración adelantada
class Asignatura{
  public:
    //Relación con la clase Alumno
    typedef std::map<Alumno*,Grupo*>Alumnos;
    //Como la minima Multiplicidad es 1, lo inicializamos en el constructor
    Asignatura(Alumno& a, Grupo& g){
      setAlumno(a,g);
      setGrupo(g);
    }
    void setAlumno(Alumno& a, Grupo& g)noexcept{
      alumnos.insert(std::make_pair(&a,&g));
    }
    const Alumnos& getAlumnos()const noexcept{
      return alumnos;
    }


    //Relación con Grupo
```

```
    typedef std::set<Grupo*>Grupos;
    void setGrupo(Grupo& g)noexcept{
      grupos.insert(&g);
    }
    const Grupos& getGrupos()const noexcept{
      return grupos;
    }
  private:
    Alumnos alumnos;
    Grupos grupos;
};

class Alumno{
  public:
    //Relación con Asignatura
    typedef std::map<Asignatura*,Grupo*>Asignaturas;
    void setAsignatura(Asignatura& a, Grupo& g)noexcept{
      asignaturas.insert(std::make_pair(&a,&g));
    }
    const Asignaturas& getAsignaturas()const noexcept{
      return asignaturas;
    }
  private:
    Asignaturas asignaturas;
};

class Profesor; //declaración adelantada
class Grupo{
  public:
    //Como la multiplicidad minima es 1, lo inicializamos en el ctor
    Grupo(Asignatura& a,Profesor& p):a_(&a),p_(&p){}
    void setAsignatura(Asignatura& a)noexcept{
      a_ = &a;
    }
    void setProfesor(Profesor& p)noexcept{
      p_= &p;
    }
    const Asignatura& getAsignatura()const noexcept{return *a_;}
    const Profesor& getProfesor()const noexcept{return *p_;}
  private:
    Asignatura* a_;
    Profesor* p_;
};

class Profesor{
  public:
    //Relación con Grupo
    typedef std::set<Grupo*>Grupos;
    //Como la multiplicidad minima es 1, lo inicializamos en el ctor
    Profesor(Grupo& g){
```

```
      setGrupo(g);
    }
    void setGrupo(Grupo&g)noexcept{grupos_.insert(&g);}
    const Grupos& getGrupos()const noexcept{return grupos_;}
  private:
    Grupos grupos_;
};
```

Ejercicio 4: Sea *Figura* una clase abstracta de la que derivan las clases *Circulo*, *Triangulo*, *Cuadrado* y otras. Supongamos que existe una función para rotar figuras definida como sigue:

```
void rotar(const Figura& fig) {
  if (typeid(fig) == typeid(Circulo)) {
      // no hacer nada
  }
  else if (typeid(fig) == typeid(Triangulo)) {
    // rotar triangulo
  }
  else if (typeid(fig) == typeid(Cuadrado)) {
      // rotar cuadrado
  }
 // y otras
}
```

a) Ponga un ejemplo de la función rotar.

```
int main(){
  //Creamos las figuras:
  Circulo circulo;
  Triangulo triangulo;
  Cuadrado cuadrado;

  //Llamamos al método rotar
  rotar(circulo);
  rotar(triangulo);
  rotar(cuadrado);

  return 0;
}
```

b) ¿Cree que esta es la mejor forma de implementar la rotación de figuras? Razone la respuesta. En caso negativo, describa cómo mejorarla y emplee el ejemplo anterior para mostrar la diferencia de uso.

No. Si nos fijamos en el método rotar estamos comparando tipos de datos, es decir, al comparar el tipo de dato de la figura entrante con el tipo de dato triángulo, por ejemplo, vemos que "devuelve" (rotar triángulo), es decir, aun teniendo una clase abstracta como es Figura no estamos aplicando polimorfismo, por tanto, hemos encontrado una manera mejor y más eficiente que la comparación de tipo de datos (que es algo muy costoso).

Vamos a ver como quedaría:

```
class Figura{
  public:
    //tenemos que convertirla en interfaz
    virtual void rotar()=0;
    virtual ~Figura() = default;
};
class Circulo:public Figura{
  public:
    virtual void rotar()noexcept override{
      //no hace nada
    }
};
class Triangulo:public Figura{
  public:
    virtual void rotar()noexcept override {
      cout<<"Rotar Triangulo"<<endl;
    }
};
class Cuadrado:public Figura{
  public:
    virtual void rotar()noexcept override {
        cout<<"Rotar Cuadrado"<<endl;
      }
};

int main(){
  //Creamos las figuras
  Circulo circulo;
  Cuadrado cuadrado;
  Triangulo triangulo;

  circulo.rotar();
  cuadrado.rotar();
  triangulo.rotar();
}
```

2.5. Examen Septiembre 2010

Ejercicio 2: Dado el siguiente programa:

```
struct B {
  void f ( ) { std::cout << "f ( ) de B" << std::endl; }
  };

struct D:B {
  void f ( ) { std::cout<<"f ( ) de D" << std::endl;}
  };

void f(B b) {
  std::cout << "f ( ) externa"<< std::endl;
  b.f( );
  }

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

a) Diga si tiene error de compilación o ejecución. Modifique el código para solucionarlo y después escribe lo que imprime. Si no, sólo lo que imprime.

   No hay ningún error en el código por tanto, ejecuta correctamente:

   ```
   f() externa
   f() de B
   f() externa
   f() de B
   ```

   Imprime esto debido a que aunque el método void f(B b) reciba un objeto de tipo D, se llama al método B::f() ya que B no es una clase polimórfica y se tiene en cuenta el tipo del objeto que se recibe (de tipo B).

b) Repite el anterior pero suponiendo **B::f()** como virtual.

   Da como resultado lo mismo.

c) Repite el anterior pero teniendo en cuenta que el parámetro función externa `f()` se recibe por referencia.

   Si mantenemos como virtual B::f() y modificamos el método void f() haciendo que reciba una referencia, vemos que ahora imprime:

   ```
   f() externa
   f() de B
   f() externa
   f() de D
   ```

Ahora vemos que llama al método D::f(), esto es debido a que ya no le pasamos un objeto, si no una referencia a dicho objeto, y esto sumado a que B es una clase polimorfica hace que el compilador sepa que tiene que llamar al método D::f().

d) Repita el 2º apartado suponiendo que la definición de f()(externa) se ha cambiado a: `void f(B *b) std::count<< "f() externa«<std::endl; b.f();`

Ahora el método void f() recibe una dirección de memoria , por tanto, primero de todo necesitamos hacer cambios en el main, ya que le tenemos que pasar una referencia al método:

```
struct B {
  virtual void f () { std::cout << "f ( ) de B" << std::endl; }
};
struct D:B {
  void f () { std::cout<<"f ( ) de D" << std::endl;}
};

void f (B* b) {
  std::cout << "f ( ) externa"<< std::endl;
  b->f(); //cambiamos b.f() por b->f();
}

int main(){
  B b;
  D d;
  f(&b); //ahora le pasamos direcciones de memoria
  f(&d);
}
```

Tanto una referencia como un puntero están involucrados en la manipulación de direcciones de memoria en C++. El resultado después de aplicar los cambios es el mismo que en el apartado anterior.

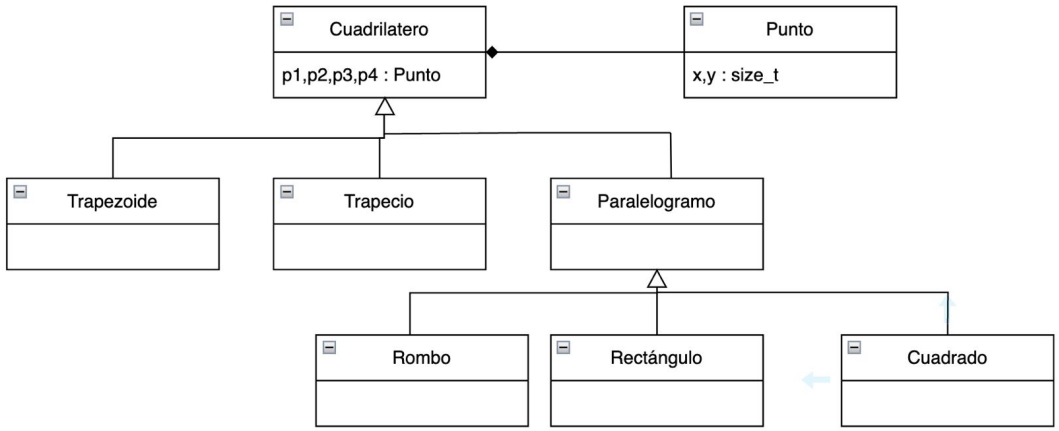
En la explicación previa, utilizamos una referencia a un objeto pasado por valor mediante void f(B& b). Si ese objeto es de tipo D, se llamará a D::f(). Ahora, al utilizar f(B* b), estamos manipulando un objeto apuntado por un puntero. Si este puntero apunta a un objeto de tipo D, se llamará a D::f(), siempre y cuando la clase B sea polimórfica, es decir, tenga al menos una función virtual.

Ejercicio 3: Para una aplicación de geometría se dispone de la clase Punto y es necesario desarrollar las clases Cuadrilátero, Trapezoide, Paralelogramo, Rombo, Rectángulo Y Cuadrado. Los atributos de estos polígonos serán los cuatro puntos que los forman en orden consecutivo.

a) Dibuje el diagrama de clases atendiendo a las siguientes definiciones:

- Cuadrilátero: polígono de cuatro lados.
- Trapezoide: cuadrilátero con ningún lado paralelo a otro.
- Trapecio: cuadrilátero con sólo dos lados paralelos.
- Paralelogramo: cuadrilátero con sus lados paralelos dos a dos.
- Rombo: paralelogramo con los cuatro lados iguales.
- Rectángulo: paralelogramos cuyos lados forman ángulos rectos entre sí.
- Cuadrado: paralelogramo cuyos lados son iguales y forman ángulos rectos entre sí.

Como cualquier figura contiene 4 puntos y todas estas figuras se pueden obtener de una en común (a partir de un cuadrilátero, podemos obtener Trapezoide, Trapecio y Paralelogramo) y a través de un Paralelogramo obtenemos (Rombo, Rectángulo y Cuadrado), por tanto, podemos implementarlo de la siguiente manera:



b) Declare (solo la interfaz) todas las clases considerando que se requieren, entre otros, métodos para calcular el perímetro y para imprimirlos (el procedimiento de impresión será diferente para todos y cada uno de los polígonos).

```
class Punto{
  public:
    Punto(size_t a, size_t b):x_(a),y_(b){}
    inline size_t x()const {return x_;}
    inline size_t y()const {return y_;}
  private:
    size_t x_,y_;
};

class Cuadrilatero{
  public:
    Cuadrilatero(Punto p1,Punto p2, Punto p3, Punto p4):
      p1_(p1),p2_(p2),p3_(p3),p4_(p4){}
```

```
      double perimetro()const{
        return
        ↪  longitud(p1_,p2_)+longitud(p2_,p3_)+longitud(p3_,p4_)+longitud(p4_,p1_);
      }
    virtual void impresion () const = 0;
  protected: //Lo hacemos protected, para que puedan tener acceso las clases
  ↪  derivadas
    Punto p1_,p2_,p3_,p4_;
    double longitud(const Punto& p1, const Punto& p2) const {
    return sqrt(pow(p1_.x() - p2_.x(), 2) + pow(p1_.y() - p2_.y(), 2));
    }
};

class Trapezoide : public Cuadrilatero{
  public:
    Trapezoide(Punto& p1,Punto& p2,Punto& p3,Punto&
    ↪  p4):Cuadrilatero(p1,p2,p3,p4){}
    //Método específico llamado impresion
    virtual void impresion()const override{
      std::cout<<"El perimetro del Trapezoide es:
      ↪  "<<Cuadrilatero::perimetro()<<std::endl;
    }
};
class Trapecio : public Cuadrilatero{
  public:
    Trapecio(Punto& p1,Punto& p2,Punto& p3,Punto&
    ↪  p4):Cuadrilatero(p1,p2,p3,p4){}
    //Método específico llamado impresion
    virtual void impresion()const override{
      std::cout<<"El perimetro del Trapecio es:
      ↪  "<<Cuadrilatero::perimetro()<<std::endl;
    }
};
class Paralelogramo: public Cuadrilatero{
  public:
    Paralelogramo(Punto& p1,Punto& p2,Punto& p3,Punto&
    ↪  p4):Cuadrilatero(p1,p2,p3,p4){}
    //Método específico llamado impresion
    virtual void impresion()const override{
      std::cout<<"El perimetro del Paralelogramo es:
      ↪  "<<Cuadrilatero::perimetro()<<std::endl;
    }
};
class Rombo : public Paralelogramo{
  public:
    Rombo(Punto& p1,Punto& p2,Punto& p3,Punto&
    ↪  p4):Paralelogramo(p1,p2,p3,p4){}
   virtual void impresion()const override{
      std::cout<<"El perimetro del Rombo es:
      ↪  "<<Cuadrilatero::perimetro()<<std::endl;
```

```
    }
};
class Rectangulo : public Paralelogramo{
  public:
    Rectangulo(Punto& p1,Punto& p2,Punto& p3,Punto&
    ↪  p4):Paralelogramo(p1,p2,p3,p4){}
   virtual void impresion()const override{
      std::cout<<"El perimetro del Rectangulo es:
      ↪  "<<Cuadrilatero::perimetro()<<std::endl;
    }
};
class Cuadrado : public Paralelogramo{
  public:
    Cuadrado(Punto& p1,Punto& p2,Punto& p3,Punto&
    ↪  p4):Paralelogramo(p1,p2,p3,p4){}
   virtual void impresion()const override{
      std::cout<<"El perimetro del Cuadrado es:
      ↪  "<<Cuadrilatero::perimetro()<<std::endl;
    }
};
```

2.6. Examen Junio 2013

Ejercicio 1: Considere el siguiente esquema:

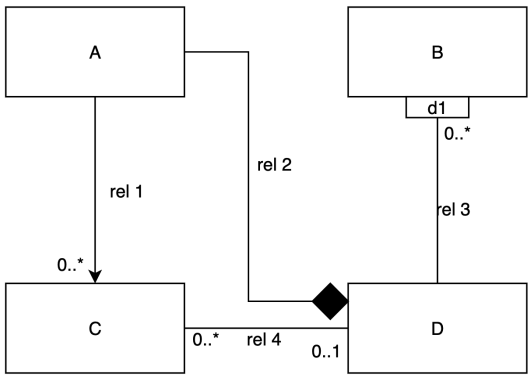


Figura 2.2: Diagrama de clase del ejercicio

a) Escriba para cada clase los atributos extrictamente necesarios para la implementación de las relaciones en las que participan.

```
class A{
  public:
    //Alias de la relacioncon C
    typedef std::set<C*>Cs;
    void setC(C&)noexcept;
    const Cs& getC()const noexcept;
  private:
    Cs cs_; //rel1
};

class C{
  public:
  //como la multiplicidad es 0-1, no es necesario que para crear C exista D
    void setD(D&)noexcept;
    const D& getD()const noexcept;
  private:
    D* d_;//rel4
};

class B{
  public:
   //Alias de la relacion con D, suponemos que el tipo de dato de d1 es int
    typedef std::map<int,D*>Ds;
    void setD(D& d)noexcept;
    const Ds& getD()const noexcept;
  private:
    Ds ds_;//rel3
};

class D : private A{//rel2 como es es 1-1, herencia privada
```

```
  public:
    D(A& a):A(a){} //rel 2
    //Alias de la relacion con B
    typedef std::set<B*>Bs;
    void setB(B&)noexcept;
    const Bs& getB()const noexcept;

    //Alias de la relacion con C
    typedef std::set<C*>Cs;
    void setC(C&)noexcept;
    const Cs& getC()const noexcept;

    int clave() const noexcept{return clave_;}
  private:
    Bs bs_; //rel3
    Cs cs_; //rel4
    int clave_; //calificador
};
```

b) Defina los constructores que sean oportunos para la clase C.

C solo tendrá un constructor debido a puede que un objeto de C no esté instanciado con ningun objeto de D, por tanto, tendrá un constructor y será por omisión.

c) Suponga que se añade un atributo de enlace x en rel1, ¿Como cambiaría los miembros de la clase A?

Al añadirse un atributo de enlace x cualquiera, como la relación tiene una multiplicidad 1 - N, ese atributo de enlace se alamcena en la clase cuya multiplicidad es N (en este caso C) y la clase con la multiplicidad 1 (A) sigue recibiendo un conjutno de elementos de C. Por tanto, para cada elemento de tipo C* en dicho conjunto tendrá un atributo x (el de enlace) diferente.

```
template <typename T>
class A{
  public:
    //nueva relacion con C, donde T es el tipo del atributo de enlace
    typedef std::set<C*>Cs;
    void setC(C&);
    const Cs& getC()const noexcept;
  private:
    Cs cs_;//nueva rel1
}
```

No se añade nada a C de la relación, debido a que es una relación unidireccional.

```
class C{
  int x;//atributo de enlace
};
```

Ejercicio 2: Implemente la relación rel4 del ejercicio anterior mediante una clase de asociación. Para ello:

- Defina la clase con los atributos que estime oportunos y declarando métodos `asocia()` para cada sentido de la relación.
- Defina una función miembro que asocie un objeto C con otro de D en los supuestos:
  - Cuando, en el caso de que el objeto origen ya esté asociado a otro, este objeto origen simplemente pasa a estar asociado al nuevo objeto.
  - Cuando, en tal circunstancia, se lanza la cadena "validación de multiplicidad".
- Defina una función miembro que asocie un objeto de D con uno de C.

```
class DC{
  public:
    //Alias de las relaciones
    typedef std::set<C*>Cs; //Conjunto de valores que tendrá la clave D
    typedef std::map<C*,D*>CDs;
    typedef std::map<D*,Cs>DCs;

    void asocia(D&,C&)noexcept;
    void asocia(C&,D&)noexcept;

    //observadores
    CDs getCs(D&)const noexcept; //puede haber valor de D o no
    D* getD(C&)const noexcept; //puede haber valor de C o no
  private:
    CDs directa_;
    DCs inversa_;
}

//Implementación de los métodos
void CD::asocia(D& d, C& c)noexcept{

  auto i = directa_.find(&c);
  if(i != directa_.end() && i->second == &d){
    throw std::runtime_error("validación de multiplicidad"<<std::endl);
  }
  directa_[&c] = &d; //permite sobreescribir enlaces
  inversa_[&d].insert(&c);
}
void CD::asocia(C& c, D& d)noexcept{ asocia(d,c); }

DC::CDs getCs(D& d)const noexcept{
  //buscamos d está asociado
  auto i = inversa_.find(&d);
  if(i!=inversa_.end())return i->second;
  else{
    //Creamos un objeto vacio para devolverlo
    DC::Cs vacio;
    return vacio;
  }
```

```
}
D* getD(C& c)const noexcept{
  //buscamos si el objeto c está asociado
  auto i = directa_.find(&c);
  if(i!=directa_.end()) return i->second;
  else return nullptr; //si no está asociado, devolvemos un puntero nulo.
}
```

Ejercicio 3: Dado el siguiente diagrama de clase:

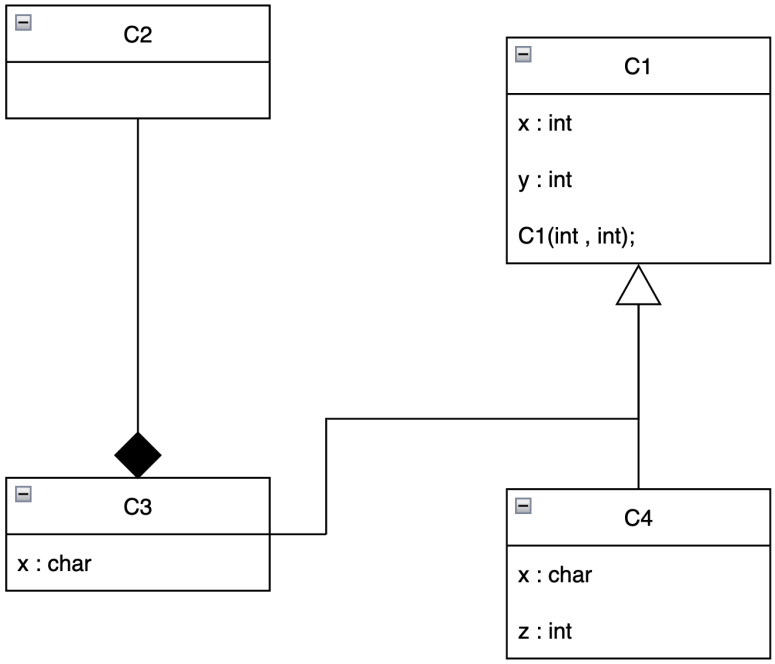


Diagrama de clases del ejercicio

1. Defina las clases únicamente con los miembros implicados en el diagrama e implemente el constructor de la clase C3.

```
class C1{
    int x,y;
  public:
    C1(int a, int b):x(a),y(b){};
};
class C2{//..};
class C3:public C1, private C2{
    char x;
  public:
    //implementación del constructor de C3
    C3(int i, int j, char a ):C1(i,j),x(a){}
};
class C4: public C1{
    char x; int z;
  public:
    C4(char a, int b):C1(b,b)x(a),z(b){}
}
```

Ejercicio 4: ¿Que devuelve por pantalla este programa?

```
using namespace std;
class B{
  public:
    void f() { cout << "f() de B" << endl; }
    virtual void g() { cout << "g() de B" << endl; }
    virtual void h() = 0;
  protected:
    int b;
};
class D1 : virtual public B{
  public:
    void f() { cout << "f() de D1" << endl; }
    virtual void g() { cout << "g() de D1" << endl; }
  protected:
    int d1;
};
class D2 : virtual public B{
  public:
    void f(int i) { cout << "f(" << i << ") de D2" << endl; }
    virtual void h() { cout << "h() de D2" << endl; }
  private:
    int d2;
};
class D3 : public D1{
  public:
    void g() { cout << "g() de D3" << endl; }
    void h() { cout << "h() de D3" << endl; }
  private:
    int d3;
};
class D4 : public D1, public  D2{
  private:
    int d4;
};
void f(B &b){
    cout << "f() externa" << endl;
    b.f();
    b.g();
    b.h();
}
```

Este programa devuelve por pantalla:

```
int main(){
  B b, *pb;
  D1 d1;
  D2 d2;
  D3 d3;
  D4 d4;
  f(b); f(d1); f(d2); f(d3); f(d4);
  d4.D1::f();
  d4.f(5);
  d4.f(3.7);
  d4.g();
  d4.h();
  pb = new D4;
  pb->f();
  pb->D4::f(3);
  pb->f();
  pb->g();
  pb->h();
  delete pb;
}
```

Primero de todo, vemos que en la linea 1 se quiere crear un objeto de B cuando esta es una clase abstracta y no se puede instanciar. También vemos que la clase D1 al heredar de la clase B no está declarando el método void h() que hace que B sea abstracta, se tendría que redefinir.

```
class D1 : virtual public B{
  public:
    void f() { cout << "f() de D1" << endl; }
    virtual void g() { cout << "g() de D1" << endl; }
    virtual void h() {cout<< "h() de D1"<<endl;}
  protected:
    int d1;
};
```

Como la clase D4 hereda publicamente de D1 y D2, al llamar al método d4.f(5) y d4.f(3.7) este serán ambigüos debido a que no sabe a cual llamar si a los de D1 o D2, lo mismo pasa con d4.h().

```
  class D4 : public D1, public  D2{
    public:
      void f(int){cout<<"f(int) de D4"<<endl;}
      void h(){cout<<"h() de D4"<<endl;}
    private:
      int d4;
  };
```

Por último vemos que convertimos correctamente un puntero de tipo B a uno de tipo D4 y luego queremos acceder al método f() de D4, pero B no se especializa directamente en una clase D4 si no en una clase D1 o D2, por tanto, tendríamos que realizar una conversión explicita mediante dynamic_cast.

```cpp
int main(){
  B *pb; //desaparece el objeto b
  D1 d1; //clase bien definida
  D2 d2;
  D3 d3;
  D4 d4;
  //f(b); desaparece
  f(d1);
  f(d2);
  f(d3);
  f(d4);
  d4.D1::f();
  //ya no da error de ambigüedad
  d4.f(5);
  //ya no da error de ambigüedad
  d4.f(3.7);
  d4.g();
  d4.h();
  pb = new D4;
  pb->f();
  //tenemos que convertir el
  //puntero pb explicitamente a un de
  ↪  d4.
  //pb->D4::f(3);
  if(D4* pd4 = dynamic_cast<D4*>(pb)){
    pd4->f(3);
    delete pd4; //eliminamos el
    ↪  puntero usado
  }
  pb->f();
  pb->g();
  pb->h();
  pb = nullptr; //hacemos que apunte a
  ↪  nada antes de eliminar, para
  ↪  evitar errores.
  delete pb; //eliminamos el puntero
}
```

```
Salida de programa
   corregido:
  f () externa
  f () de B
  g() de D1
  h() de D1
  f () externa
  f () de B
  g () de B
  h () de D2
  f () externa
  f () de B
  g () de D3
  h () de D3
  f () externa
  f () de B
  g() de D1
  h() de D4
  f() de D1
  f(int) de D4
  f(int) de D4
  g() de D1
  h() de D4
  f () de B
  f(int) de D4
  f () de B
  g() de D1
  h() de D4
```

2.7. Examen Abril 2016

Ejercicio 1: Sea este diagrama:

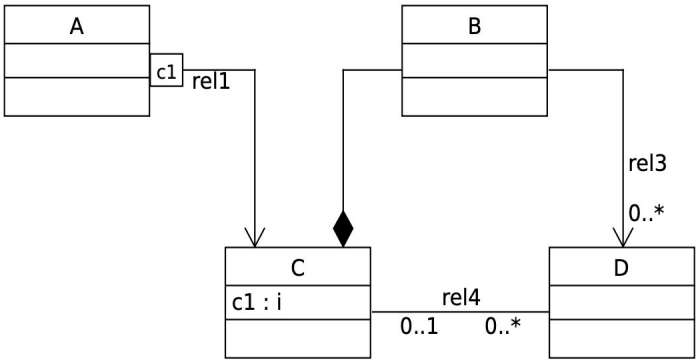


Figura 2.3: Diagrama de clases del ejercicio

a) Escriba para cada clase los atributos estrictamente indispensables para la implementación de las relaciones en las que participa.

Preguntar si hace falta implementarlas.

```
//Como es una relacion calificada, suponemos que el tipo de dato del
↪  calificador es un entero.
class A{
  public:
    typedef std::map<size_t, C*>Calificadas;
    void rel1(C&
    ↪  c1)noexcept{calificadas_.insert(std::make_pair(c1.getid(),&c1));}
    const Calificadas& getCalificada()const noexcept{return calificadas_;}
  private:
    Calificadas calificadas_;
};

class C : private B{ //rel2
  public:
    C(B& b):B(b){}
    size_t getid()const noexcept{return id_;}

    //Alias de la relacion con D rel3
    typedef std::set<D*>Ds;
    void rel4(D& d)noexcept{ds_.insert(&d);}
    const DS& getD()const noexcept{return ds_;}
  private:
    size_t id_; //atributo calificador
    Ds ds_;
};
```

```
class D{
  public:
    void rel4(C& c){c_ = &c;}
    C* getC()const noexcept{return c_;}
  private:
    C* c_;
};

class B{
  public:
    //Alias del conjunto de D
    typedef std::set<D*>DS;
    void rel3(D& d)noexcept{ds.insert(&d);}
    const DS& getD()const noexcept{return ds_;}
  private:
    DS ds_;
};
```

b) Defina los constructores que estime oportunos para la clase D.

La clase D va a tener solamente un constructor por defecto, debido a que no recibe ningún parámetro procedente de otras clases relacionadas, ya que un objeto de la clase C puede estar o no instaciado con uno de la clase D (multiplicidad 0..1) y la relación con la clase B es unidireccional, por tanto, no recibe el objeto o puntero de la clase B.

c) Suponga que se añade un atributo de enlace de tipo X a rel3 ¿cómo cambiariarían los miembros?

Sabemos que tenemos una relación de asociación unidireccional 1 - muchos entre las clases B y D (rel3), si nos ceñimos a la teoría sabemos que el atributo de enlace se guarda en la clase que tiene la multiplicidad muchos (D), por tanto no cambiaría nada ya que la clase B seguiría recibiendo un conjunto (set) de punteros de tipo D, y en D se guardaría dicho atributo de enlace en su parte privada.

Ejercicio 2: Implemente la rel4 del ejercicio anterior mediante una clase de asociación. Para ello:

a) Defina la clase con los atributos que estime oportunos declarando dos métodos `asocia()` y otros dos llamados `asociados()`, una pareja para cada sentido de la relación.

b) Defina las funciones miembro `asocia()` de tal forma que ambas permitan crear/modificar el doble enlace entre un objeto de C y otro de D. Si D ya está asociado a un C, se desvinculará del mismo y se enlazará al nuevo.

c) Defina las dos funciones miembro `asociados()`.

```
class CD{
public:
  //Alias del conjunto de punteros de tipo D
  typedef std:: set<D*> Ds;

  //Alias de las relaciones
  typedef std::map<C*,Ds> CDs;
  typedef std::map<D*,C*> DC;

  void asocia(C& c, D& d)noexcept{
    //Si D está asociado con otro C, se desvincula y se asocia el que le pasamos
    ↪  por parámetro
    auto i = inversa_.find(&d);
    if(i!=inversa_.end()){ //D está asociado con un C
      directa_[i->second].erase(&d);//eliminamos el antiguo D.
    }
    inversa_[&d]=&c;
    directa_[&c].insert(&d);
  }
  void asocia(D& d,C& c)noexcept{asocia(c,d);}

  Ds asociados(C& c)const noexcept{
    auto i = directa_.find(&c);
    if(i != directa_.end()) return i->second;
    else{
      //Devolvemos un conjunto vacio
      CD::Ds vacio;
      return vacio;
    }
  }

  C* asociados(D& d)const noexcept{
    auto i = inversa_.find(&d);
    if(i!=inversa_.end()) return i->second;
    return nullptr;
  }
private:
  CDs directa_;
  DC inversa_;
};
```

Ejercicio 3 : Dadas las siguientes definiciones de clases:

```
-------------------------------------------------------------------------------
struct X{
  X(char c) : c(c) { cout << "Ctor. de X" << endl; } char c;
};

struct A{
  A(X x);
 void f() { cout << "Método f() de A" << endl; } ~A() { cout << "Dtor. de A" << endl; }
  X x;
};
struct B{
  B(X x);
 void f() { cout << "Método f() de B" << endl; } ~B() { cout << "Dtor. de B" << endl; }
  X x;
};
-------------------------------------------------------------------------------
```

a) Escriba las definiciones de los constructores de A y B de forma que impriman el texto del constructor de A y el constructor de B respectivamente.

```
struct A{
  A(X x): x(x){}
  void f() { cout << "Método f() de A" << endl; } ~A() { cout << "Dtor. de
  ↪  A" << endl; }
  X x;
};

struct B{
  B(X x):x(x){}
  void f() { cout << "Método f() de B" << endl; } ~B() { cout << "Dtor. de
  ↪  B" << endl; }
  X x;
};
```

2.8. Examen Junio 2022

Ejercicio 1: Sean las multiplicidades de las relaciones:
Cliente - Vehiculo → N - M; Cliente - Contrato → 1 - N; Contrato - Vehiculo → 1..N - 1..M;

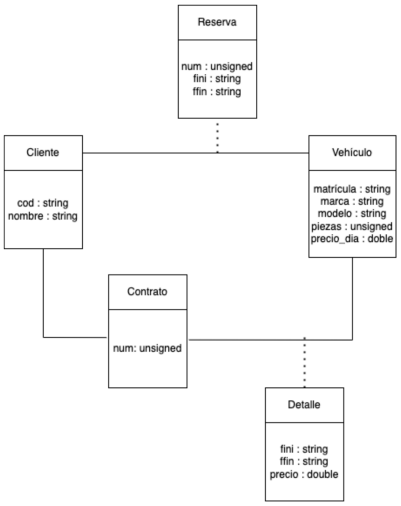


Figura 2.4: Diagrama de clase del ejercicio

a) Crea una clase de asociación entre **Cliente-Vehículo ACV**.

Implementación de la clase de asociación **Cliente-Vehículo ACV**:

Como vemos es una relación de asignación con una clase de asociación Reserva, como tenemos una multiplicidad N - M pueden haber instancias de cada clase no relacionadas, cosa que tenemos que tener en cuenta los métodos observadores.

```
class ACV{
  public:
    //Alias de los diccionarios de Cliente y Vehiculo
    typedef std::map<Cliente*,Reserva*> Clientes;
    typedef std::map<Vehiculo*,Reserva*> Vehiculos;
    //Alias de los diccionarios de las relaciones
    typedef std::map<Cliente*,Vehiculos>CVs;
    typedef std::map<Vehiculo*,Clientes>VCs;

    //Método que asocian los objetos
    void setACV(Cliente&,Vehiculo&,Reserva&)noexcept;
    void setACV(Vehiculo&,Cliente&,Reserva&)noexcept;

    //Métodos observadores
    Clientes getClientes(Vehiculo&)const noexcept;
    Vehiculos getVehiculos(Cliente&)const noexcept;
  private:
    CVs directa_;
    VCs inversa_;
};
```

Ahora vamos a implementar los métodos (no nos lo pide el enunciado, pero vamos a hacerlo para repasar).

```
void ACV::setACV(Cliente& c, Vehiculo& v, Reserva& r)noexcept{
  directa_[&c].insert(std::make_pair(&v,&r));
  inversa_[&v].insert(std::make_pair(&c,&r));
}
void ACV::setACV( Vehiculo& v, Cliente& c, Reserva& r)noexcept{
  //delegamos en la operación anterior
  setACV(c,v,r);
}

ACV::Clientes ACV::getClientes(Vehiculo& v)const noexcept{
  //vamos a comprobar si el vehículo está asociado o no
  auto i = inversa_.find(&v);
  //if(i!=inversa_.end())return i->second;
  else{
    //Creamos un conjunto vacio para devolverlo
    ACV::CVs vacio;
    return vacio;
  }
}


ACV::Vehiculos ACV::getVehiculos(Cliente& c)const noexcept{
  //comprobamos que el objeto c Cliente está relacionado
  auto i = directa_.find(&c);
  if(i!=directa_.end())return i->second;
  else{
    //Creamos un conjunto vacio
    ACV::Vehiculos vacio;
    return vacio;
  }
}
```

b) Crea la relación **Cliente-Contrato**.

Implementación de la relación **Cliente-Contrato**:

Como vemos la multiplicidad de esta asociación es 1 - N es decir, que un objeto de la clase Cliente se puede relacionar con varios de la clase Contrato, pero un objeto de la clase Contrato solamente se puede instanciar con un único Cliente. Debido a esto, vamos a implementar la relación modificando las clases Cliente y Contrato añadiendo los miembros imprescindibles para que se pueda llevar a cabo.

```
class Cliente{
  public:
    //Alias del conjunto de Contratos de un Cliente
    std::set<Contrato*>Contratos;
    inline void setContrato(Contrato& c)noexcept{
      contratos_.insert(&c);
    }
    inline Contratos getContratos()const noexcept{return contratos_;}
```

```
  private:
    std::string cod,nombre;
    Contratos contratos_;
};

class Contrato{
  public:
    Contrato(unsigned numero, Cliente& c):num(numero),cliente_(&c){}
    Cliente* getCliente()const noexcept{return *cliente_;}
  private:
    unsigned num;
    Cliente* cliente_;
};
```

c) Por último, crea la relación **Vehículo-Contrato**.

Para implementar una relación de asociación donde su multiplicidad es 1..N - 1..M, es decir, muchos - muchos donde siempre va a haber como mínimo un objeto de ambas clases relacionados, esa unión vamos a realizarla mediante un map, como ya lo hicimos mediante una clase de asociación en el apartado a, ahora lo implementaremos modificando las clases Contrato y Vehiculo, con la diferencia de que no hace falta la comprobación de si existe ni devolver un conjunto vacío gracias a la multiplicidad 1 - N, 1 - M, donde en el constructor recibirá un objeto de cada clase por parámetro.

```
class Contrato{
  public:
    Contrato(unsigned numero, Cliente& c,Vehiculo& v,Detalle&
    ↪  d):num(numero),cliente_(&c){
      //delegamos en el método que crea los enlaces.
      setVehiculos(v,d);
    }
    Cliente* getCliente()const noexcept{return *cliente_;}

    //Alias del diccionario de Vehiculo y Detalle
    typedef std::map<Vehiculo*,Detalle*>Vehiculos;

    void setVehiculos(Vehiculo& v, Detalle& d)noexcept{
      vehiculos_.insert(std::make_pair(&v,&d));
    }
    const Vehiculos& getVehiculos()const noexcept{
      return vehiculos_;
    }
  private:
    unsigned num;
    Cliente* cliente_; //relacion apartado b
    Vehiculos vehiculos_;
};

class Vehiculo{
  public:
    //Alias del diccionario de Contrato y Detalle
    std::map<Contrato*,Detalle*>Contratos;
```

```
    //Multiplicidad 1 - N, un objeto de vehiculo se inicializa con un
    ↪ Contrato y Detalle
    Vehiculo(string matri,string m, string mod,unsigned pi,double p,
    ↪ Contrato& c, Detalle &d):
    ↪ matricula(matri),marca(m),modelo(mod),piezas(pi),precio_dia(p){
      setContratos(c,d);
    }
    void setContratos(Contrato& c, Detalle& d)noexcept{
      contratos_.insert(std::make_pair(&c,&d));
    }
    const Contratos& getContratos()const noexcept{
      return contratos_;
    }
  private:
    string matricula, marca, modelo;
    unsigned piezas;
    double precio_dia;
    Contratos contratos_;
};
```

Ejercicio 2:

```
template <typename T>
class MatrizTriangularSuperior{
public:
  explicit MatrizTriangularSuperior(size_t n=1):n(n),v(n*(n+1)/2)
     {};
  ~MatrizTriangularSuperior()=default;
  T operator() (size_t i, size_t j) const{
    if(i>= n || j>=n)throw out_of_range;
    return v[i*(2+n-i+1)/2 + j - i];
  }
  size_t orden() const noexcept{return n;}
private:
  size_t n;
  std::vector<T>v;
};
```

Figura 2.5: Clase Matriz Triangular Superior del ejercicio

a) Como definiría la clase MatrizSimétrica, ¿como una Especialización o como una Composición? y por qué:

Una matriz MatrizTriangularSuperior es aquella que solo tiene valores distinto a 0 en su diagonal superior, por tanto, una matriz simétrica será aquella que los valores por encima de sus diagonales son iguales. Podemos delegar en las operaciones de la MatrizTriangularSuperior, por lo que vamos a implementarlo mediante una composición (objeto), esto hace que la implementación sea más facil y óptima.

b) Defina la clase MatrizSimetrica según lo elegido anteriormente.

```
template<typename T>
class MatrizSimetrica{
  public:
    MatrizSimetrica(size_t orden = 1) : mt(orden) {}
    size_t orden() const noexcept{
      return mt.orden();
    }
    T operator()(size_t i, size_t j)const{
      return mt.operator()(i,j);
    }
private:
  MatrizTriangularSuperior<T> mt;
};
```

c) Define una relación de realización con una nueva clase MatrizCuadrada y las anteriores.

Una matriz cuadrada es aquella que tiene el mismo número de filas que de columnas, por tanto, esta será la clase base de las que heredarán las anteriores, como es una relación de realización la clase matriz cuadrada será abstracta.

```
template <typename T>
class MatrizCuadrada {
  public:
    //Este método pasa a ser no const, para poder rellenar las matrices.
    virtual T& operator()(size_t, size_t) = 0;
    virtual size_t orden() const noexcept = 0;
    virtual ~MatrizCuadrada() = default;
};
template <typename T> class MatrizTriangularSuperior : public
↪  MatrizCuadrada<T>{
public:
  explicit MatrizTriangularSuperior(size_t n = 1) : n(n), v(n*(n+1)/2) {}
  ~MatrizTriangularSuperior() = default;

  T& operator()(size_t i, size_t j){
    if (i >= n || j >= n) throw out_of_range("Fuera de rango");
    return v[(i * n + j - (i * (i + 1) / 2))];
  }
  size_t orden() const noexcept{return n;}
private:
  size_t n;
  std::vector<T> v;
};
```

```
template<typename T> class MatrizSimetrica : public MatrizCuadrada<T>{
public:
  MatrizSimetrica(size_t orden = 1) : mt(orden) {}
  size_t orden() const noexcept{return mt.orden();}
  T& operator()(size_t i, size_t j){
  return mt.operator()(i,j);
  }
private:
  MatrizTriangularSuperior<T> mt;
};
```

d) Implementa una función genérica `rellenar<T,F>()` para matrices de cualquier tipo. Debe de funcionar con ambas matrices. Parámetro tipo F es una función, objeto función o expresión lambda que recibe coordenadas y cuyo resultado se calcula a través de un algoritmo. Usará parámetro F para objetos valor de cada elemento.

```
template <typename T, typename F>
void rellenar(MatrizCuadrada<T>& matriz, F funcion) {
  size_t orden = matriz.orden();
  for (size_t i = 0; i < orden; i++) {
    for (size_t j = 0; j < orden; j++) {
      matriz(i, j) = funcion(i, j);
    }
  }
}
int main(){
  //....
  //Vamos a hacerlo por ejemplo con una matriz simetrica
  MatrizTriangularSuperior<size_t>M(5);
  auto funcion = [](size_t i, size_t j){return i+j;};
  rellenar(M, funcion);
}
```

e) Rellenar matrices con el valor de la suma de las coordenadas. Para objetos función usar junto a la función `rellenar<T,F>()`.

```
struct SumaCoordenadas{
  size_t operator()(size_t i, size_t j)const {return i+j;};
};
```

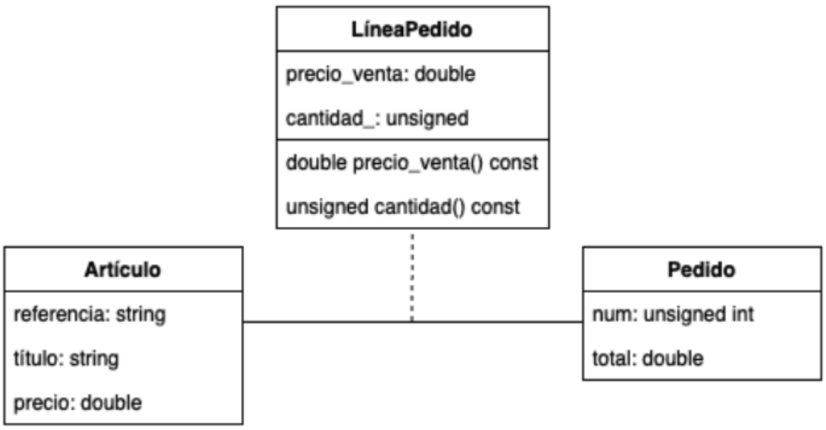
f) Escribe un fragmento de código en la que uses la función `rellenar<T,F>()` para rellenar dos matrices, una simétrica de tipo int, y otra triangular de tipo double. Para la primera deberás emplear un objeto función mientras que para la segunda deberás de emplear una función lambda.

```
int main(){
  //Creamos las matrices
  MatrizSimetrica<int>MS(5);
  MatrizTriangularSuperior<double>MTS(4);
  //Declaramos el objeto a funcion para la primera
  SumaCoordenadas coordenadas;
  rellenar(MS,coordenadas);
```

```
  //Declaramos la función lambda para la otra matriz
  auto lambda = [](size_t i, size_t j)->double{return 1.0/(i+j+1);};
  rellenar(MTS,lambda);
  return 0;
}
```

2.9. Examen Febrero 2023

Ejercicio 2: Sea el siguiente diagrama:



a) Defina la clase de asociación que permita implementar dicha relación escribiendo exclusivamente las definiciones de los miembros imprescindibles para implementarla.

Como no tenemos una multiplicidad definida suponemos que es (N - 1..M).

```
class LineaPedido{};
class Articulo{}; //lado 1..M
class Pedido{}; //lado N

//Clase de asociación
class ArticuloPedido{
  public:
    //Alias de las relaciones Articulo - LineaPedido y Pedido-LineaPedido
    typedef std::map<Articulo*,LineaPedido*>Articulos;
    typedef std::map<Pedido*,LineaPedido*>Pedidos;
    //Alias de las relaciones Articulo - Pedido
    typedef std::map<Articulo*,Pedidos>ArticulosPedidos;
    typedef std::map<Pedido*,Articulos>PedidosArticulos;

    void setArticuloPedido(Articulo& a, Pedido& p ,LineaPedido& lp)noexcept{
      directa[&a].insert(std::make_pair(&p,&lp));
      inversa[&p].insert(std::make_pair(&a,&lp));
    }
    void setArticuloPedido(Pedido& p, Articulo& a, LineaPedido& lp)noexcept{
      //Delegamos en la anterior
      setArticuloPedido(a,p,lp);
    }

    Articulos getArticulos(Pedido& p)const noexcept{
      auto i = inversa.find(&p);
      if(i!=inversa.end())return i->second;
      return std::map<Articulo*,LineaPedido*>();
    }
    const Pedidos& getPedidos(Articulo& a)const noexcept{
      return directa.find(&a)->second;
```

```
    }

  private:
    ArticulosPedidos directa;
    PedidosArticulos inversa;
};
```

b) ¿Es obligatorio usar una clase de asociación? Si es así explica razonadamente el porqué, si no, implementa como hacerlo escribiendo las declaraciones de los miembros esenciales.

No, no es obligatorio, las relaciones de asociación podemos implementarla de dos maneras, haciendo la clase de asociación ó incluyendo los miembros imprescindibles en cada clase para que la relación se pueda llevar a cabo.

```
class Articulo{
  public:
    //Métodos propios de la clase
    //Alias del diccionario que recibe de la relacion con Pedido
    typedef std::map<Pedido*,LineaPedido*>Pedidos;
    void setPedido(Pedido& p,LineaPedido& lp)noexcept{
      pedidos_.insert(std::make_pair(&p,&lp));
    }
    const Pedidos& getPedidos()const noexcept{return pedidos_;}
  private:
    //Atributos propios de la clase
    Pedidos pedidos_; //nombre del diccionario.
};

class Pedido{
  public:
    //Método propios de la clase
    //Alias del diccionario que recibe de la relación con Articulo.
    typedef std::map<Articulo*,LineaPedido*>Articulos;
    Pedido(Articulo& a, LineaPedido& lp,/*Atributos de la clase*/) {
      setArticulo(a,lp);
    }
    void setArticulo(Articulo& a,LineaPedido& lp)noexcept{
      articulos_.insert(std::make_pair(&a,&lp));
    }
    const Articulos& getArticulo()const noexcept{return articulos_;}
  private:
    //Atributos propios de la clase
    Articulos articulos_;
};
```

Ejercicio 3: Sea la clase ListaOrdenada:

```
template<typename T>
class ListaOrdenada{
  public:
      typedef typename list<T>::const_iterator iterator;
      void insertar(const T& e);
      void eliminar(iterator p);

      iterator begin();
      iterator end();
};
```

a) Explica la relación que se puede establecer entre ListaOrdenada y la clase list. Implemente la clase ListaOrdenada.

Una ListaOrdenada podemos pensarlo como un tipo de lista donde sus elementos están ordenados mediante un criterio de ordenación a diferencia de list que será una lista con sus elementos desordenados. Por tanto, como no tienen el mismo comportamiento no podemos hacer que list se especialice en una ListaOrdenada pero si podemos delegar parte del comportamiento de list en la clase ListaOrdenada. Por ello, vamos a implementarla mediante una relación de composición 1 - 1 que se puede hacer mediante herencia privada o inclusión de un Objeto list.

```
template<typename T>
class ListaOrdenada{
  public:
      typedef typename list<T>::const_iterator iterator;
      ListaOrdenada():list<T>(){}
      void insertar(const T& e);
      void eliminar(iterator p);

      iterator begin();
      iterator end();
  private:
    list<T>listaordenada_; //objeto de tipo list
};
```

b) Añade el método `size_t contar (const T& e)` const que cuente el número de ocurrencias de un elemento dado. Para ello utilice `count_if()` de la clase STL que recibe dos iteradores y un predicado (clase objeto función que devuelve un booleano). Defina el predicado como una clase de objetos función o como una función anónima (función lambda) equivalente.

```
template<typename T>
class ListaOrdenada{
  public:
      typedef typename list<T>::const_iterator iterator;
      ListaOrdenada():list<T>(){}
      void insertar(const T& e){
        //Como vamos a insertar ordenadamante, haremos uso de lower_bound, que
        ↪  nos devuelve un iterador al primer elemento menor que el.
        auto p = std::lower_bound(listaordenada_.begin(),listaordenada_.end(),e);
        //insertamos el elemento
        listaordenada_.insert(p,e);
      }
      void eliminar(iterator p){
        listaordenada_.erase(p);

      }
      size_t contar(const T& e)const{
        //Para contar nos dice que hagamos uso de count_if
        return count_if(listaordenada_.begin(),listaordenada_.end(),[&e](const T&
        ↪  elto){return elto == e ;});
      }
      iterator begin(){return listaordenada_.cbegin();}
      iterator end(){return listaordenada_.cend();}
  private:
    list<T>listaordenada_; //composición
};
```

Ejercicio 4: Sea el siguiente código:

```
class Instrumento{
  public:
    typedef enum {instrumento, percusion, cuerda, viento}tClase;
    Instrumento(string nom):nombre_(nom){clase_ = instrumento;}

    void tocar()const{
      cout<<"Soy un "<<nombre()<<" y pertenezco a "<<clase()<<endl;
    }

    string nombre() const{return nombre_;}
    string clase()const{
      switch (clase_){
      case percusion: return "percusion";
      case cuerda: return "cuerda";
      case viento: return "viento";
      default: return "instrumento";
      }
    }
  protected:
    string nombre_;
    tClase clase_;
};

class Percusion:public Instrumento{
    Percusion(string n):Instrumento(n){clase_ = percusion;}
};

class Cuerda: public Instrumento{
    Cuerda(string n):Instrumento(n){clase_ = cuerda;}
};

class Viento: public Instrumento{
    Viento(string n):Instrumento(n){clase_=viento;}
};
```

a) ¿Se puede mejorar la implementación de esta jerarquía de clases usando métodos polimórficos? En caso afirmativo, reescribe el programa para obtener un resultado idéntico.

Si, como vemos no estamos haciendo uso de polimorfísmo en ningún momento, simplmente estamos creando objetos de tipo Instrumento y asignándoles una clase en particular. Si hacemos uso de polimorfísmo tanto ese método clase() como el enum desaparecerían ya que la clase se asignaría en cada constructor de las clases Derivadas de Instrumento, además que cada clase derivada tendría su propio método tocar(), quedando:

```
class Instrumento{
  public:
    Instrumento(string nom):nombre_(nom){}
    virtual void tocar()const{
      cout<<"Soy un "<<nombre()<<" y pertenezco a Instrumento"<<endl;
```

```
    }
    string nombre() const{return nombre_;}
    virtual ~Instrumento() =default; //Es polimorfica, destructor virtual.
  protected:
    string nombre_;
};

class Percusion:public Instrumento{
  public:
    Percusion(string n):Instrumento(n){}
    void tocar()const override {
      cout<<"Soy un "<<nombre()<<" y pertenezco a Tambor"<<endl;
    }
    string nombre()const {return nombre_;}
};

class Cuerda: public Instrumento{
  public:
    Cuerda(string n):Instrumento(n){}
     void tocar()const override{
      cout<<"Soy un "<<nombre()<<" y pertenezco a Cuerda"<<endl;
    }
    string nombre()const {return nombre_;}
};

class Viento: public Instrumento{
  public:
    Viento(string n):Instrumento(n){}
    void tocar()const override{
      cout<<"Soy un "<<nombre()<<" y pertenezco a Viento"<<endl;
    }
    string nombre()const {return nombre_;}
};
```

b) Implemente una función que clasifique un vector de punteros a instrumentos en otros 3 vectores de punteros a instrumentos, uno para cada clase: percusión, cuerda y viento, ignorando aquellos que no pertenezcan a estas clases.

```
void clasificaInstrumento(vector<Instrumento*>&instrumentos,
↪  vector<Percusion*>& percusiones, vector<Viento*>& vientos,
↪  vector<Cuerda*>& cuerdas){
  //convertimos los instrumentos en su tipo, y los guardamos en sus vectores
  for(auto i : instrumentos)
    if(Percusion* pp = dynamic_cast<Percusion*>(i)){
    ↪  percusiones.push_back(pp);
    }
    else if(Viento* pv = dynamic_cast<Viento*>(i)){ vientos.push_back(pv);
    }
    else if(Cuerda* pc= dynamic_cast<Cuerda*>(i)){ cuerdas.push_back(pc);
    }
}
```

2.10. Examen Junio 2023

Ejercicio 1: Implemente los miembros imprescindibles para que las relaciones se puedan realizar:

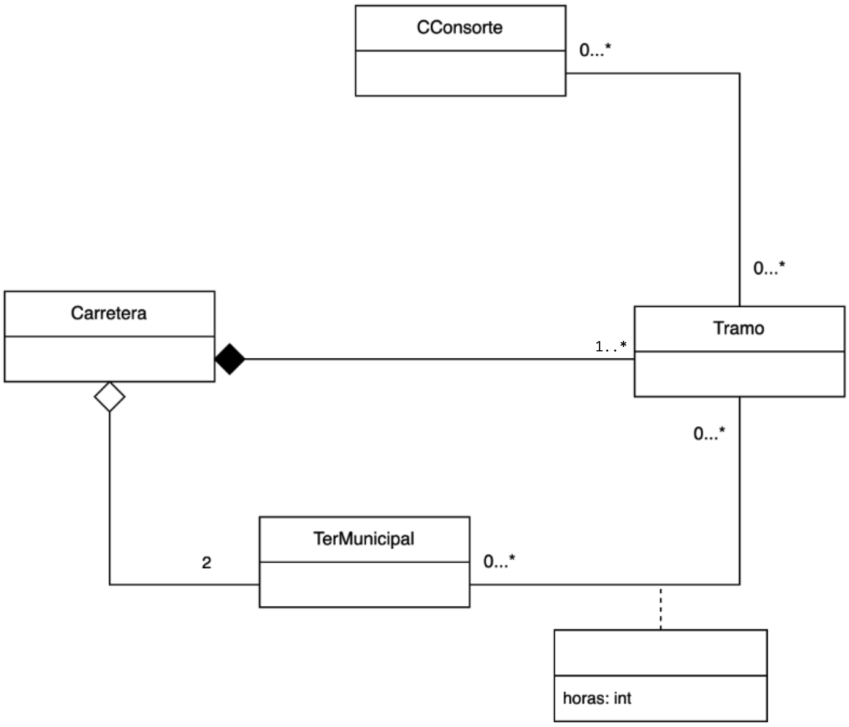


Figura 2.6: Diagrama de clase del ejercicio

Relación Cconsorte - Tramo:

Vemos que la relación entre Cconsorte y Tramos es una asociación muchos - muchos bidireccional, por tanto, la podemos implementar de dos maneras:

- Mediante una clase de asociación que almacena los miembros imprescindibles para que se lleve a cabo la relación.
- Modificando las clases y añadiendo los miembros imprescindibles de la relación en cada una de las clases asociadas.

Nosotros implementaremos esta relación modificando las clases Cconsorte y Tramo.

```
class Cconsorte{
  public:
    //Alias del conjunto de objetos Tramos
    std::set<Tramo*>Tramos;
    inline void setTramo(Tramo& t)noexcept{
      tramos_.insert(&t);
    }
    inline const Tramos& getTramos()const noexcept{
      return tramos_;
    }
```

```
  private:
    Tramos tramos_;
};

class Tramo{
  public:
    //Alias del conjunto de objeto Cconsorte
    typedef std::set<Cconsorte*>Cconsortes;
    inline void setCconsorte(Cconsorte& c)noexcept{
      cconsortes_.insert(&c)
    }
    inline const Cconsortes& getCconsortes()const noexcept{
      return cconsortes_;
    }
  private:
    Cconsortes cconsortes_;
};
```

Relación Carretera - Tramo:

La relación entre las clases Carretera y Tramo es una relación de composición 1-muchos, por tanto, no podemos implementarla mediante una herencia privada ya que la clase Carretera está formada por una serie de Tramos. Esta relación son unidireccionales donde solo se alamcena la información de la relación en la clase compuesta (la que tiene el rombo).

```
class Carretera{
  public:
    //Alias del conjunto de objetos de Tramos
    std::set<Tramo>Tramos;
    Carretera(Tramo& t){
      setTramo(t);
    }
    void setTramos(Tramo &t)noexcept{
      tramos_.insert(t);
    }
  private:
    Tramos tramos_;
};

class Tramo{
  //definimos un atributo para ordenar tramos
  size_t id_;
  public:
    //Lo anterior..
    //tenemos que sobrecargar el operador < para poder ordenar las carreteras, ya
    ↪  que al ser un set de objetos y no de punteros no tenemos una
    ↪  implementación de dicho operador por defecto.
    friend bool operator < (const Tramo& t1, const Tramo& t2)noexcept{
      return t1.id_ < t2.id_;
    }
};
```

Relación Carretera - TerMunicipal:

La relación entre las clases Carretera y TerMunicipal es de tipo agregación donde una instancia de Carretera se relaciona con dos instancias de TerMunicipal. Las agregaciones se pueden implementar como asociaciones unidireccionales donde solo se almacena la información en la clase agregada (la que tiene el rombo).

```
class Carretera{
  public:
  //Miembros de la relación con la clase Tramo
    std::set<Tramo>Tramos;
    Carretera(Tramo& t, TerMunicipal& ter1, TerMunicipal&
    ↪  ter2):t1(&ter1),t2(&ter2){
      setTramos(t);
    }
    void setTramos(Tramo &t)noexcept{tramos_.insert(t);}
    void setTerMunicipal(TerMunicipal& ter1, TerMunicipal&ter2){
      t1 =&ter1:
      t2 = &ter2;
    }
    const TerMunicipal& getTer1()const noexcept{return *t1;}
    const TerMunicipal& getTer2()const noexcept{return *t2;}
  private:
    Tramos tramos_;
    TerMunicipal *t1, *t2; //las dos instancias con las que se relaciona

};
```

Relación TerMunicipal - Tramo:

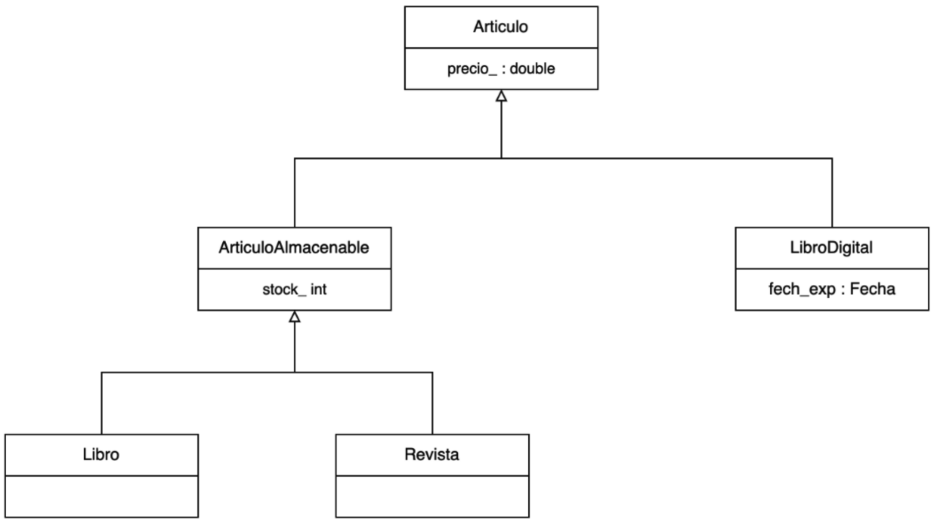
Vemos que esta relación es del tipo asociación pero contiene un atributo de enlace, al ser las dos multiplicidad muchos - muchos la manera más fácil de implementar la relación es mediante una clase de asociación donde se alamcenará los miembros imprescindibles para que se lleve a cabo la relación.

```
class TerMunicipal_Tramo{
  public:
    //Alias del diccionario TerMunicipal - horas
    typedef std::map<TerMunicipal*,int>TerMunicipales;
    //Alias del diccionario Tramos - horas
    typedef std::map<Tramo*,int>Tramos;
    //Alias de los diccionarios de la clase
    typedef std::map<Tramo*,TerMunicipales>TramosTer;
    typedef std::map<TerMunicipal*,Tramos>TerTramos;

    void setTerMunicipalTramo(TerMunicipal& ter, Tramo& t, int h)noexcept{
      directa_[&ter].insert(std::make_pair(&t,h));
      inversa_[&t].insert(std::make_pair(&ter,h));
    }
    inline void setTerMunicipalTramo(Tramo& t, TerMunicipal& ter, int h)noexcept{
      setTerMunicipalTramo(ter,t,h);
    }
    //observaores: como pueden no haber instancias, devolvemos una copia
```

```
    TerMunicipales getTerMunicipales(Tramo& t)const noexcept{
      auto i = inversa_.find(&t);
      if(i!=inversa_.end()) return i->second;
      else{
        TerMunicipal_Tramo::TerMunicipales vacio;
        return vacio;
      }
    }
    Tramos getTramos(TerMunicipal& ter)const noexcept{
      auto i = directa_.find(&ter);
      if(i!=directa_.end()) return i->second;
      else{
        TerMunicipal_Tramo::Tramos vacio;
        return vacio;
      }
    }
  private:
    TerTramos directa_;
    TramosTer inversa_;
};
```

Ejercicio 2: Sea este esquema:



a) Añade otra clase al esquema llamada DiscoDigital que contiene, al igual que Libro-Digital, la fecha de expiración del DiscoDigital. Modifique el diagrama de acuerdo a esto.

```
class Articulo{
  public:
    Articulo(double p):precio_(p){}
  protected:
    double precio_;
};

class ArticuloAlmacenable : public Articulo{
  public:
```

```
    ArticuloAlmacenable(int s, double p):Articulo(p),stock_(s){}
  protected:
    size_t stock_;
};


class LibroDigital : public Articulo{
    Fecha fech_exp;
  public:
    LibroDigital(Fecha f, double p):Articulo(p),fech_exp(f){}
};

class Libro : public ArticuloAlmacenable{};
class Revista: public ArticuloAlmacenable{};
//Clase nueva que nos piede el ejercicio
class DiscoDigital : public Articulo{
    Fecha f_expir_;
  public:
    DiscoDigital (Fecha f, double p):Articulo(p),f_expir_(f){}
}
```

b) En esta nueva clase tenemos canciones. Se requiere saber en todo momento el número de canciones que tiene cada disco con el método observador nCancion()const.

Un disco va a estar formado por una serie de canciones, como mínimo vamos a tener una canción. Por tanto, como no hay dependencia de existencia podemos implementarlo mediante una asociación a 1..muchos.

```
  class LibroDigital : public Articulo{
    Fecha f_expir_;
  public:
    LibroDigital(Fecha f, double p, Cancion& c):Articulo(p),f_expir_(f){
      setCanciones(c);
    }
    //Alias del conjunto de canciones del disco
    typedef std::set<Cancion*>Canciones;
    void setCanciones(Cancion& c )noexcept{
      canciones_.insert(c);
    }
    const Canciones& getCanciones()const noexcept{
      return canciones_;
    }
    size_t nCancion()const noexcept{return canciones_.size();}
  private:
    Canciones canciones_;
};
```

c) Implementa la clase Canción sabiendo que una canción pertenece a un solo disco, es necesario saber el número de canciones que tenemos en cada disco.

Como bien nos dice una canción solo pertenece a un disco, como un Disco Digital se compone de canciones y esta es un tipo de relación unidireccional, vamos a implementar esta relación como una asociación unidireccional a 1 donde solamente guarda

información la clase Cancion.

```
class Cancion{
  public
    Cancion(DiscoDigital& d):discodigital_(&d){}
    void setDiscoDigital(DiscoDigital& disco)noexcept{
      discodigital_ = &disco;
    }
    const DiscoDigital& getDiscoDigital() const noexcept{
      return discodigital_;
    }
  private:
    DiscoDigital* discodigital_;
};
```

Ejercicio 3:

1. Para la figura anterior, implementar una función que te permita saber el número de Artículos (Revista, Libro, LibroDigital) que es menor que un precio dado, utilizando una estructura tipo EsBarato:

   Este método va a devolver un entero que por defecto será 0, y recibe por parámetros el conjunto de artículos a comprobar junto con el precio mínimo que tienen que superar, además dentro de la clase Articulo vamos a declarar el objeto a función EsBarato.

   ```
   class Articulo{
     double precio_;
     public:
       //...
       inline const double getprecio()const noexcept{return precio_;}
       struct EsBarato{
         bool operator () (const Articulo& a, double precio){
           return a.precio_ < precio; }
       };
   };
   size_t ArticulosBaratos(const std::vector<Articulo*>& articulos, double
   ↪  precioMinimo){
     //variable que contendrá el número de articulos que son baratos
     size_t articulosbaratos = 0;
     //Definimos el objeto a funcion
     Articulo::EsBarato esbarato;
     //recorremos el conjunto de articulos comprobando si son baratos o no
     for(const auto& a : articulos){
       if(esbarato(*a,precioMinimo)) articulosbaratos++;
     }
     return articulosbaratos;
   }
   ```

2. El apartado anterior pero miramos solo los Libros y usando la forma Lambda.

   Ahora este método contará solamente los artículos que son Libros, por tanto, tendremos que convertir un objeto de tipo Articulo a un objeto de tipo Libro, lo haremos mediante dynamic_cast para poder realizar dicha conversión ya que es de arriba - abajo.

```
size_t LibrosBaratos(const std::vector<Articulo*>& articulos, double
↪  precioMinimo)
{
  //Definimos la variable
  size_t librosbaratos = 0;
  //Creamos la función Lambda
  auto EsBarato = [precioMinimo](Articuxlo* a){
    //solo contamos los libros
    if(Libro* l = dynamic_cast<Libro*>(a)){
      return l!=nullptr && l->getprecio() < precioMinimo;
    }
    else return false;
  }
  //recorremos el vector de articulos
  for(const auto& a : articulos){
    //comprobamos si es barato o no
    if(EsBarato(a))librosbaratos++;
  }
  return librosbaratos;
}
```

3. Devolver la lista de articulos que hay que reponer, dado un lista de punteros a Articulo y un stock minimo.

   Ahora vamos a crear un método que va a devolver un conjunto (vector) de los articulos que tendrán que ser repuestos, esa comprobación podemos hacerlo o bien con un objeto a función o con una función Lambda que nos comprueben el stock del articulo con el stock minimo dado por parámetro.

   Además solamente podemos obtener el stock de los ArticulosAlmacenables por tanto, al recibir un vector de tipo Articulo, tenemos que convertir dichos objetos al tipo ArticuloAlmacenable y acceder al método observador `getstock()`

```
class ArticuloAlmacenable : public Articulo{
    int stock_;
  public:
    ArticuloAlmacenable(int s, double p):Articulo(p),stock_(s){}
    inline int getstock()const noexcept{return stock_;}
};
std::vector<Articulo*> ReponeArticulos(const std::vector<Articulo*>&
↪  articulos, int stockMinimo)
{
  //creamos el vector a devolver
  std::vector<Articulo*>articulosAreponer
  //Creamos la función Lambda
  auto Reponer = [stockMinimo](Articulo* a){
    //convertimos los articulos en ArticulosAlmacenables para ver su stock
    if(ArticuloAlmacenable* AA = dynamic_cast<ArticuloAlmacenable*>(a)){
      return AA!=nullptr && AA->getstock() < stockMinimo;
    }else return false;
  }
  //Ahora recorremos el vector de articulos comprobando su stockMinimo
```

```
      for(const auto& a : articulos){
        if(Reponer(a))articulosAreponer.push_back(a);
      }
      return articulosAreponer;
    }
```

Ejercicio 4: Sea la clase Matriz:

```
class Matriz{
  public:
    Matriz(size_t m = 10, size_t n= 10, double y = 0.0):
      m(m), n(n), x(m*n,y){}

  private:
    size_t m, n;
    std::valarray<double> x;
};
```

1. Implementela con el uso de templates para cuaquier de tipo de dato, por omisión esta plantilla será tipo double. La matriz está compuesta por F(Fila), C(Columna). Con F y C por omision 10. El constructor por defecto es del tipo por omision T. Crea un objeto de tipo Matriz que sea :

   - Una matriz m1 de tipo base inicializada con los datos por omisión.
   - Una matriz m2 de tipo double de 10 x 10 con elementos 1.
   - Una matriz m3 de tipo double y 3 x 4 con elementos 0.

   Qué pasaría si hicieramos `m1 = m2; m2 = m3;`

   Escribe la declaración de un `ostream << ()` para cualquier tipo de plantilla.

Para poder asignar matrices como estas pueden tener dimensiones diferentes debemos de implementar tanto el constructor de copia como el operador de asignación por copia, para poder asignar matrices a otras sin que se pierda información.

```
template <typename T = double>
class Matriz{
  public:
    Matriz(T m = 10, T n = 10, T y = 0.0):
      m_(m),n_(n),x_(m*n , y){}

    //Constructor por copia
    Matriz(const Matriz<T>& other):m_(other.m_),n_(other.n_),x_(other.x_){}

    //operador de asignación por copia
    Matriz<T>& operator = (const Matriz<T>& other){
      if(this!=&other){//evitamos la autoasignación
        //comprobamos que tengan el mismo número de filas y columnas
        if(m_ != other.m_ || n_ !=other.n_){
          //cambiamos los tamaños
          m_ = other.m_;
          n_ = other.n_;
          //reajustamos el tamaño de la matriz
```

```
          x_.resize(m_*n_);
        }
        return *this;
      }
    }
    //operador de inserción en flujo
    friend std::ostream& operator << (std::ostream& , const Matriz<T>& )noexcept;
  private:
    T m_,n_;
    std::valarray<T> x_;
};
```

Ejercicio 5: Sean las definiciones:

```
class Base{
  public:
    Base(){std::cout<<"Ctor. de Base"<<std::endl;}
    virtual ~Base(){std::cout<<"Dtor. de Base"<<std::endl;}
};
class Derivada : public Base{
  public:
    Derivada(){std::cout<<"Ctor. de Derivada"<<std::endl;}
    ~Derivada(){std::cout<<"Dtor. de Derivada"<<std::endl;}
};
```

a) ¿Qué imprime?

```
{
  Derivada d;
}
```

Imprime:
Ctor.Base
Ctor.Derivada
Dtor.Derivada
Dtor.Base

b) ¿Qué imprime?

```
Base *pb = new Derivada;
delete pb;
```

Imprime:
Ctor.Base
Ctor.Derivada
Dtor.Derivada
Dtor.Base

c) ¿Que se imprimiría en el apartado anterior si el destructor de Base no fuera virtual?

Imprime:
Ctor.Base
Ctor.Derivada
Dtor.Base

d) Se debería declarar virtual el destructor de Derivada? ¿Por qué?

No, porque ya está declarado como virtual el Destructor de la Base esto hace que se llamen a los destructores de las clases derivadas. Además este comportamiento se hereda haciendo que si tenemos una clase Derivada2 que sea una Especialización de la clase Derivada, se acceda a su destructor, sin tener que declarar Derivada:: D() como virtual.

e) ¿Qué imprime?

```
Derivada *pd = dynamic_cast<Derivada*>(new Base);
delete pd;
```

Imprime:
Ctor. Base

Como el destructor de la Base es virtual, solamente se llama al constructor del mismo, y si no fuera virual no se podría realizar la conversión de un puntero de tipo Base a uno de tipo Derivada explicitamente.