

Universidad de Valparaíso  
Facultad de Ingeniería

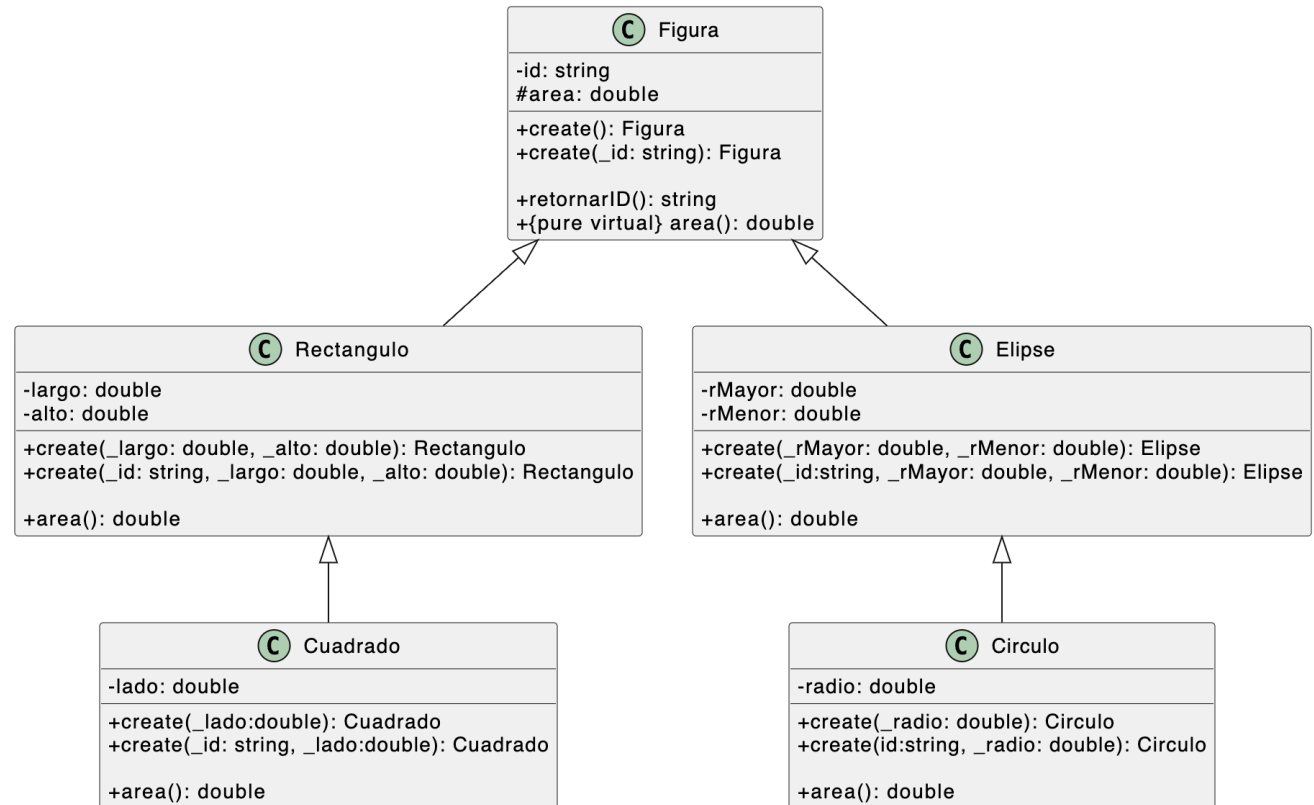


Escuela de  
Ingeniería Informática

# Programación orientada a objetos

# Implementación c++

# Implementación de jerarquía



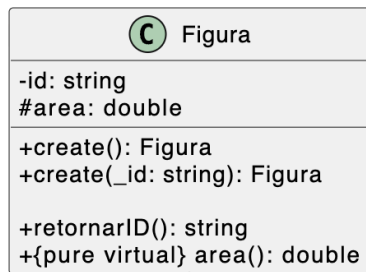
## Sintaxis

```
class ClassName : public BaseClassName { ... };
```

Nombre de la clase superior  
(superclase)

# Implementación de jerarquía

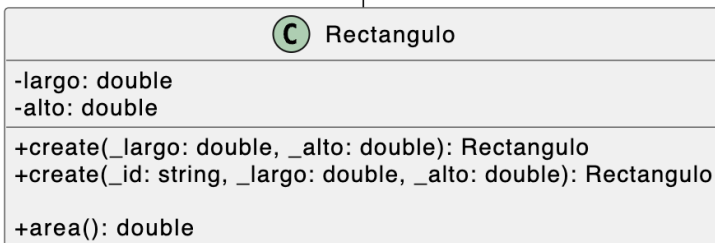
## Implementación de la jerarquía



```
class Figura{
    ...
    Figura(){ ... }
    Figura(std::string _id){ ... }
    ...
}
```

La clase base asigna el identificador de la figura.

En el constructor por omisión, el valor del identificador debe ser una decisión de diseño

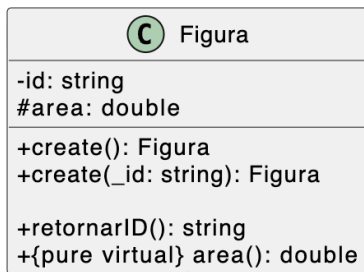


```
class Rectangulo: public Figura{
    ...

    Rectangulo(double _largo, double _alto){ ... }
    Rectangulo(std::string _id, double _largo, double _alto): Figura(_id){ ... }

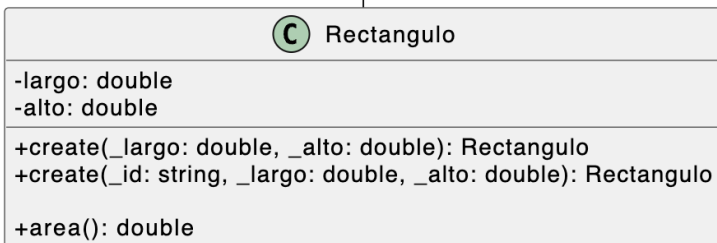
}
```

# Implementación de jerarquía



```

class Figura{
    ...
    Figura(){ ... }
    Figura(std::string _id){ ... }
    ...
}
  
```



```

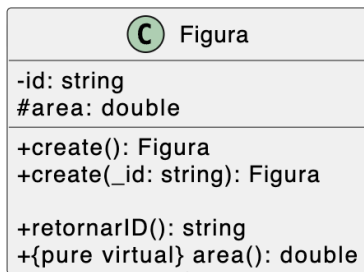
class Rectangulo: public Figura{
    ...

    Rectangulo(double _largo, double _alto){ ... }
    Rectangulo(std::string _id, double _largo, double _alto): Figura(_id){ ... }
}
  
```

Si no se especifica, la clase hija llama al constructor por omisión de la clase superior.

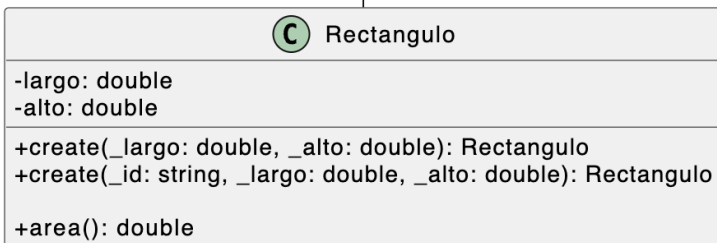
Configuran los atributos internos de la clase

# Implementación de jerarquía



```

class Figura{
    ...
    Figura(){ ... }
    Figura(std::string _id){ ... }
    ...
}
  
```



```

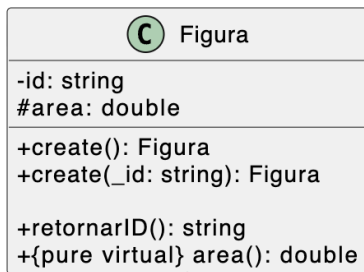
class Rectangulo: public Figura{
    ...

    Rectangulo(double _largo, double _alto){ ... }
    Rectangulo(std::string _id, double _largo, double _alto): Figura(_id){ ... }
}
  
```

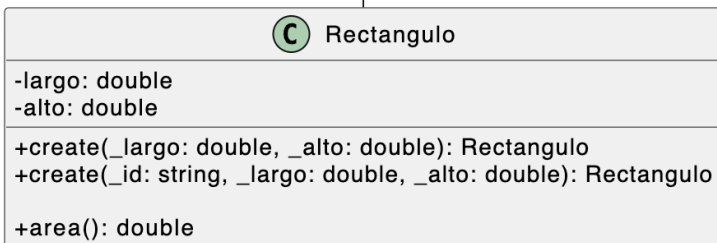
Cuando se crea un objeto a través de este constructor, la clase hija invoca al constructor de la clase base en forma explícita.

Configuran los atributos internos de la clase

# Implementación de jerarquía



```
class Figura{
    ...
    Figura(){ ... }
    Figura(std::string _id){ ... }
    ...
}
```



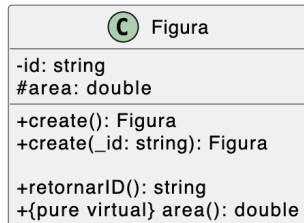
```
class Rectangulo: public Figura{
    ...

    Rectangulo(double _largo, double _alto){ ... }
    Rectangulo(std::string _id, double _largo, double _alto): Figura(_id){ ... }
}
```

Cuando se crea un objeto a través de este constructor, la clase hija invoca al constructor de la clase base en forma explícita.

Este parámetro debe ser tratado por la clase base

# Implementación de jerarquía



```

class Figura{
    ...
    Figura(){ ... }
    Figura(std::string _id){ ... }
    ...
}
  
```

```

class Rectangulo: public Figura{
    ...
    Rectangulo(double _largo, double _alto): Figura(_id){ ... }
    Rectangulo(std::string _id, double _largo, double _alto): Figura(_id){ ... }
    ...
}
  
```

```

class Cuadrado: public Rectangulo{
    ...
    Cuadrado(double _lado): Rectangulo(_id, _lado, _lado){ ... }
    Cuadrado(std::string _id, double _lado): Rectangulo(_id, _lado, _lado){ ... }
    ...
}
  
```

```

Cuadrado q1 = Cuadrado("Quad01", 10);
  
```

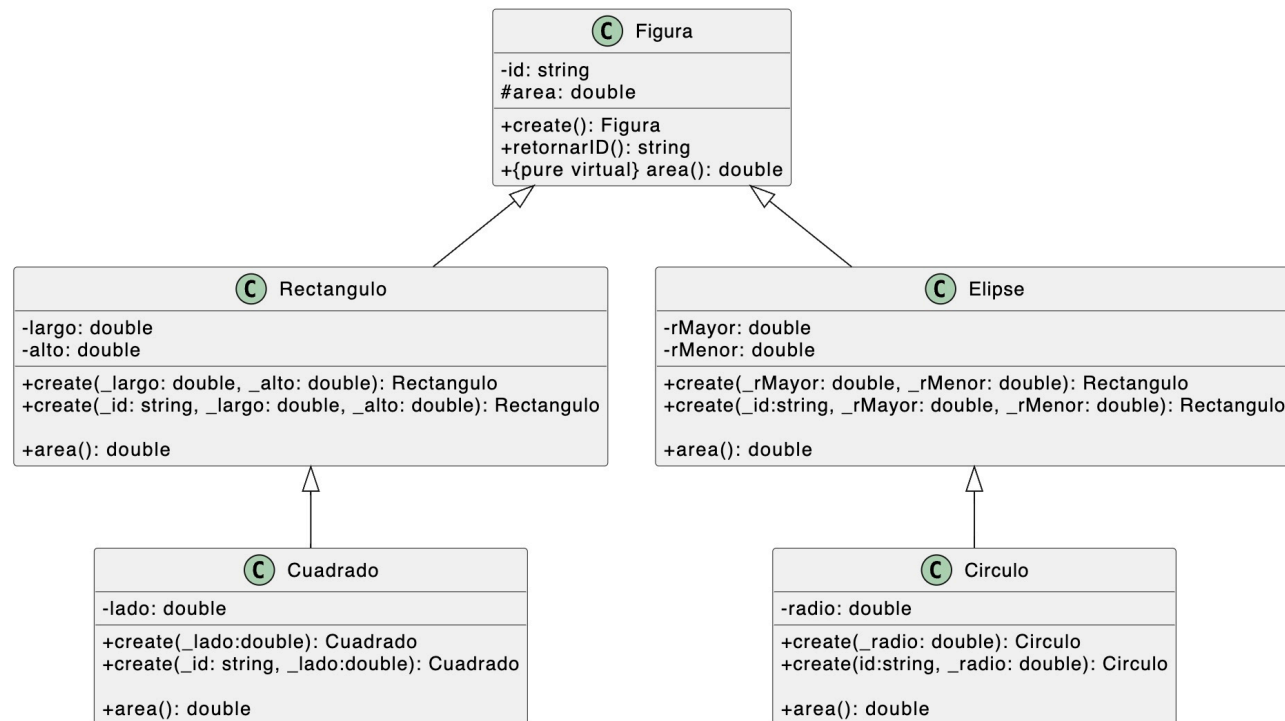


# Ejemplo de funcionamiento polimórfica

## Pseudo-Código

```

mostrarArea(f: Figura) -> void:
  print("Area=", f.area())
  
```



Recordar que para que el polimorfismo en tiempo de ejecución (sobreescribir métodos) funcione, el tipo de dato debe ser una **referencia a una clase**.

# Ejemplo de funcionamiento polimórfica

## Pseudo-Código

```
mostrarArea(f: Figura) -> void:  
    print("Area=", f.area())
```

## Formas de implementar

### Referencias con punteros de C

```
void mostrarArea(Figura* f){  
    std::cout << "El area de "<< f->getid() <<" es: " << f->area() << "\n";  
}
```

### Referencias con referencias C++

```
void mostrarArea(Figura& f){  
    std::cout << "El area de "<< f.getid() <<" es: " << f.area() << "\n";  
}
```

# Ejemplo de funcionamiento polimórfica

Cuando se utiliza referencias con **\***, los objetos de deben crear con **new**.

```
Cuadrado* q1 = new Cuadrado("micuadradoA", 10);  
mostrarArea(q1);
```

Cuando se utiliza referencias con **&**, los objetos se crean llamando directamente al constructor.

```
Cuadrado qq1 = Cuadrado("micuadradoA", 10);  
mostrarArea(qq1);
```

# Sobrecarga de constructores

Ejemplo clase Números complejos

```
class Complex{  
    private:  
        double real_;  
        double im_;  
  
    public:  
        Complex() {  
            real_ = 0;  
            im_ = 0;  
        }  
  
        Complex(double re, double im) {  
            real_ = re;  
            im_ = im;  
        }  
  
        Complex( Complex& c) {  
            real_ = c.real_;  
            im_ = c.im_;  
        }  
        ...  
}
```

Constructor de copia

# Sobrecarga de operadores

## Sintaxis

operador =

```
ClassName& operator=( const ClassName& c ){  
    if(this != &c){  
        // Código de copia  
    }  
}
```

operador +

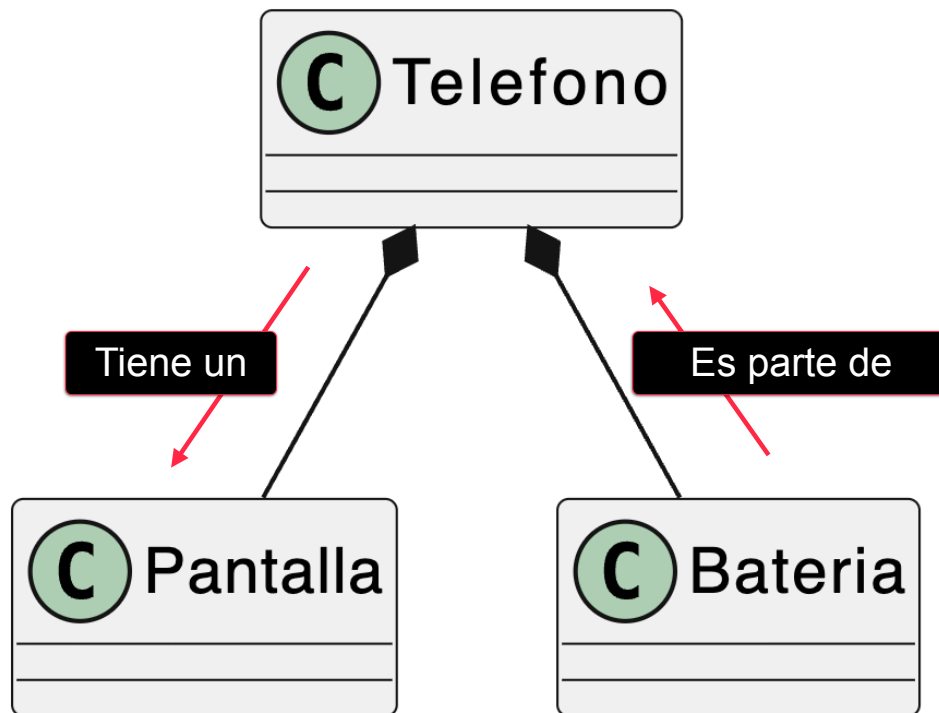
```
ClassName operator+( const ClassName& c){  
    ClassName aux = ClassName();  
  
    // Código que "suma" adecuadamente  
    // dos objetos  
  
    return(aux);  
}
```

operador ==

```
bool operator==( const ClassName& c ) const {  
    bool esIgual = false;  
  
    if(/* lógica de comparación*/){  
        esIgual = true;  
    }  
  
    return(esIgual);  
}
```

# Composición

# Composición



Principio que permite construir clases complejas a partir de clases más simples.

En lugar de heredar características de una clase base, una clase puede contener instancias de otras clases como atributos.

Reutilización

Las clases pueden ser reutilizadas en diferentes contextos sin depender de una jerarquía de herencia.

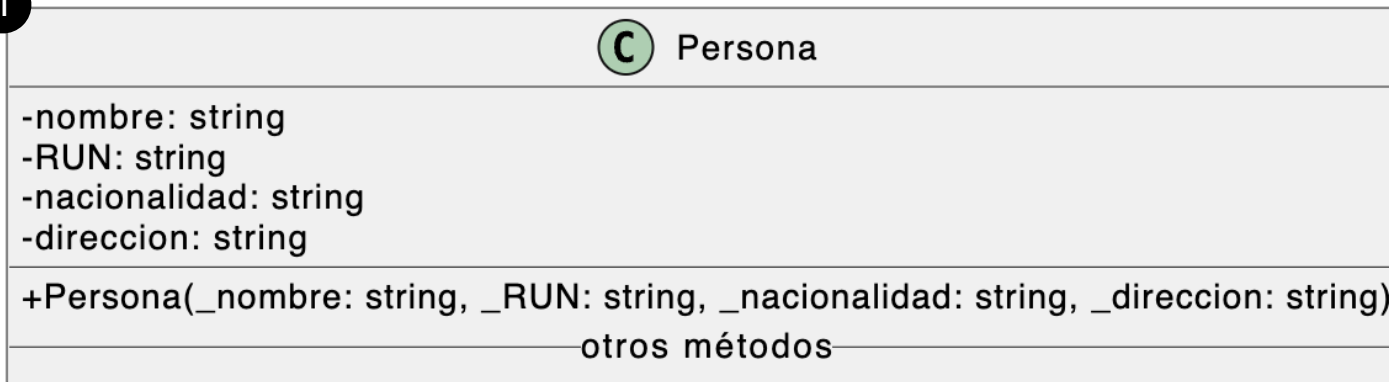
# Composición (implementación)

## Situación

Se quiere almacenar datos de personas: nombre, RUN, nacionalidad, dirección

Una posible solución se muestra en el diagrama de clases ❶

❶



Si bien esta solución logra almacenar datos de una persona, es difícil implementar un sistema de búsqueda de personas, por ejemplo, por apellido, calle, ciudad, etc, debido a que estos datos pertenecen a un campo que tiene otros datos.



# Composición (implementación)

## Situación

Se quiere almacenar datos de personas: nombre, RUN, nacionalidad, dirección

Un buen análisis de los datos a almacenar puede ayudar a comprender cómo almacenarlos en forma correcta.

Dato a guardar	Ejemplo	Observaciones
nombre	Zacarías Flores del Campo	El dato es un string, pero posee distintos campos: nombre, apellido1 y apellido2
RUN	11.111.111-1	Es un string. No hay más campos.
nacionalidad	chilena	Es un string. No hay más campos.
dirección	General Cruz 222 Valparaíso	Es un string, pero posee distintos campos: calle (string), un número (entero) y una ciudad (string)

De estas observaciones, se puede decidir que tanto el dato "nombre" como "dirección" son campos que pueden ser representados por una estructura que tenga diversos componentes (por ejemplo, una clase)

# Composición (implementación)

## Diseño de la clase Nombre

Esta clase debe almacenar el nombre y los apellidos en forma separada. Esto para permitir, a futuro, por ejemplo, realizar búsquedas sobre estos campos.

Además, debe permitir retornar el nombre completo de la persona en formato de string

### Nombre

-nombre: string  
-apellido1: string  
-apellido2: string

+Nombre(\_n: string, \_ap1: string, \_ap2: string)  
+toString(): string

# Composición (implementación)

Implementación de la  
clase **Nombre**

**C** Nombre

-nombre: string  
-apellido1: string  
-apellido2: string

+Nombre(\_n: string, \_ap1: string, \_ap2: string)  
+toString(): string

```
class Nombre{
    private:
        std::string nombre;
        std::string apellido1;
        std::string apellido2;

    public:
        Nombre(std::string _n, std::string _ap1, std::string _ap2){
            nombre      = _n;
            apellido1    = _ap1;
            apellido2    = _ap2;
        }

        std::string toString(){
            std::string nombreCompleto;

            nombreCompleto = nombre + " " + apellido1 + " " + apellido2;
            return(nombreCompleto);
        }
};
```

# Composición (implementación)

## Diseño de la clase Direccion

Esta clase debe almacenar la calle, número y ciudad en forma separada. Esto para permitir, a futuro, por ejemplo, realizar búsquedas sobre estos campos.

Además, debe permitir retornar la dirección en formato de string

### Direccion

-calle: string  
-nro: int  
-ciudad: string

+Direccion(\_calle: string, \_nro: int, \_ciudad: ciudad)  
+toString(): string

# Composición (implementación)

## Implementación de la clase **Direccion**

**C** Direccion

-calle: string -nro: int -ciudad: string  +Direccion(_calle: string, _nro: int, _ciudad: ciudad) +toString(): string
---

```
class Direccion{
    private:
        std::string calle;
        int         nro;
        std::string ciudad;

    public:
        Direccion(std::string _calle, int _nro, std::string _ciudad){
            calle  = _calle;
            nro    = _nro;
            ciudad = _ciudad;
        }

        std::string toString(){
            std::string nombreCalle;
            nombreCalle = calle + " " + std::to_string(nro) + " " + ciudad;

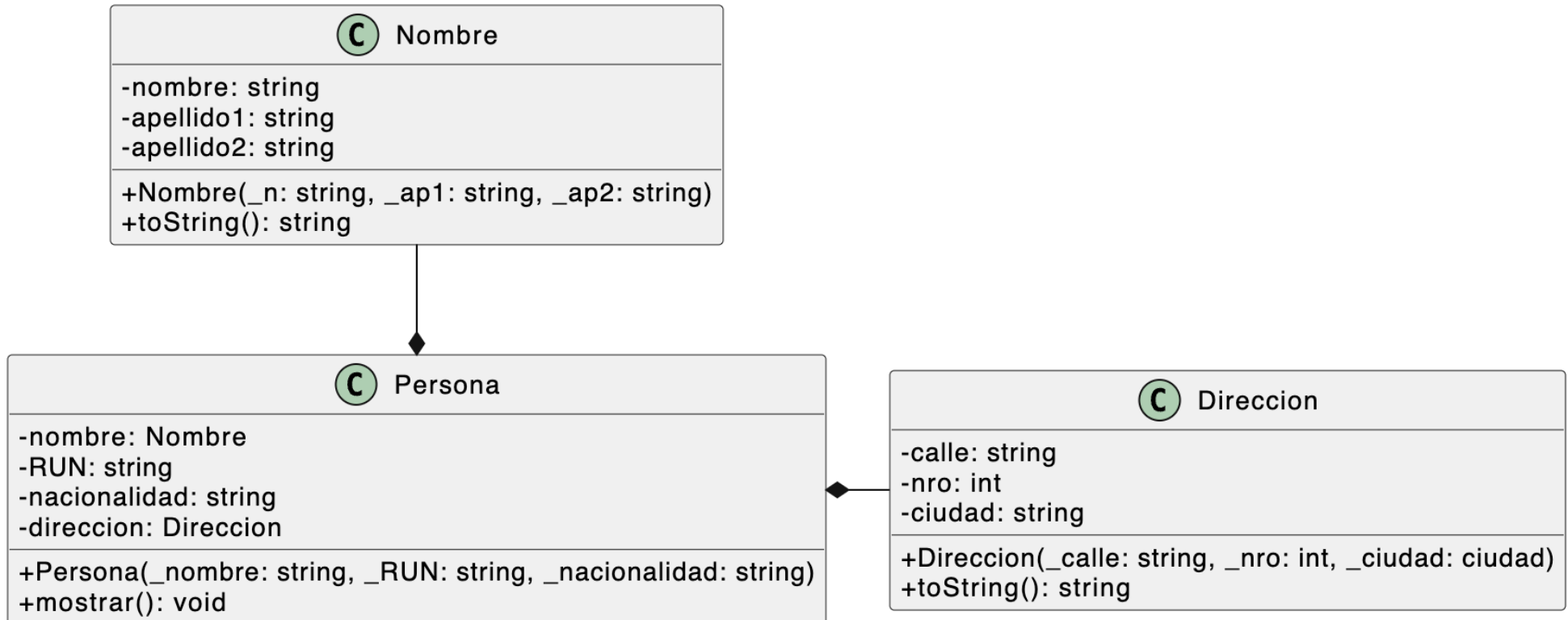
            return(nombreCalle);
        }
};
```

# Composición (implementación)

Diseño de la clase  
**Persona**

Finalmente, la clase persona  
se diseñará utilizando  
**composición de clases.**

Una persona tiene un nombre  
Una persona tiene una dirección



# Composición (implementación)

## Implementación de la clase **Persona**

```
class Persona{
    private:
        Nombre nombre;
        std::string nacionalidad;
        std::string RUN;
        Direccion direccion;

    public:
        Persona(std::string _nombre, std::string _ap1, std::string _ap2,
                std::string _nacionalidad,
                std::string _RUN,
                std::string _calle, int _nro, std::string _ciudad): nombre(_nombre, _ap1, _ap2),
                                                                    direccion(_calle, _nro, _ciudad) {

            nacionalidad = _nacionalidad;
            RUN           = _RUN;
        }

        void mostrar(){

        }

};
```

# Composición (implementación)

## Implementación de la clase **Persona**

```
class Persona{
private:
    Nombre nombre;
    std::string nacionalidad;
    std::string RUN;
    Direccion direccion;

public:
    Persona(std::string _nombre, std::string _ap1, std::string _ap2,
            std::string _nacionalidad,
            std::string _RUN,
            std::string _calle, int _nro, std::string _ciudad): nombre(_nombre, _ap1, _ap2),
                                                                direccion(_calle, _nro, _ciudad) {

        nacionalidad = _nacionalidad;
        RUN = _RUN;
    }

    void mostrar(){

    }

};
```

Crear composición (no se crea una instancia)



# Composición (implementación)

## Implementación de la clase **Persona**

```
class Persona{
    private:
        Nombre nombre;
        std::string nacionalidad;
        std::string RUN;
        Direccion direccion;

    public:
        Persona(std::string _nombre, std::string _ap1, std::string _ap2,
                std::string _nacionalidad,
                std::string _RUN,
                std::string _calle, int _nro, std::string _ciudad)
        {
            nacionalidad = _nacionalidad;
            RUN           = _RUN;
        }

        void mostrar(){

        }

};
```

Se definen todos los argumentos del constructor de la clase **Persona**

: nombre(\_nombre, \_ap1, \_ap2),  
direccion(\_calle, \_nro, \_ciudad) {

# Composición (implementación)

## Implementación de la clase **Persona**

```
class Persona{
    private:
        Nombre nombre;
        std::string nacionalidad;
        std::string RUN;
        Direccion direccion;

    public:
        Persona(std::string _nombre, std::string _ap1, std::string _ap2,
                std::string _nacionalidad,
                std::string _RUN,
                std::string _calle, int _nro, std::string _ciudad)

            : nombre(_nombre, _ap1, _ap2),
              direccion(_calle, _nro, _ciudad) {

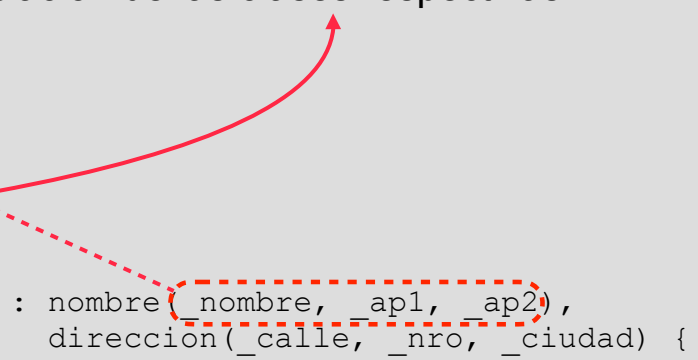
            nacionalidad = _nacionalidad;
            RUN          = _RUN;
        }

        void mostrar(){

        }

};
```

Algunos argumentos serán utilizados en la instanciación de las clases respectivas



# Composición (implementación)

## Implementación de la clase **Persona**

```
class Persona{
    private:
        Nombre nombre;
        std::string nacionalidad;
        std::string RUN;
        Direccion direccion;

    public:
        Persona(std::string _nombre, std::string _ap1, std::string _ap2,
                std::string _nacionalidad,
                std::string _RUN,
                std::string _calle, int _nro, std::string _ciudad)
        {
            : nombre(_nombre, _ap1, _ap2),
              direccion(_calle, _nro, _ciudad) {

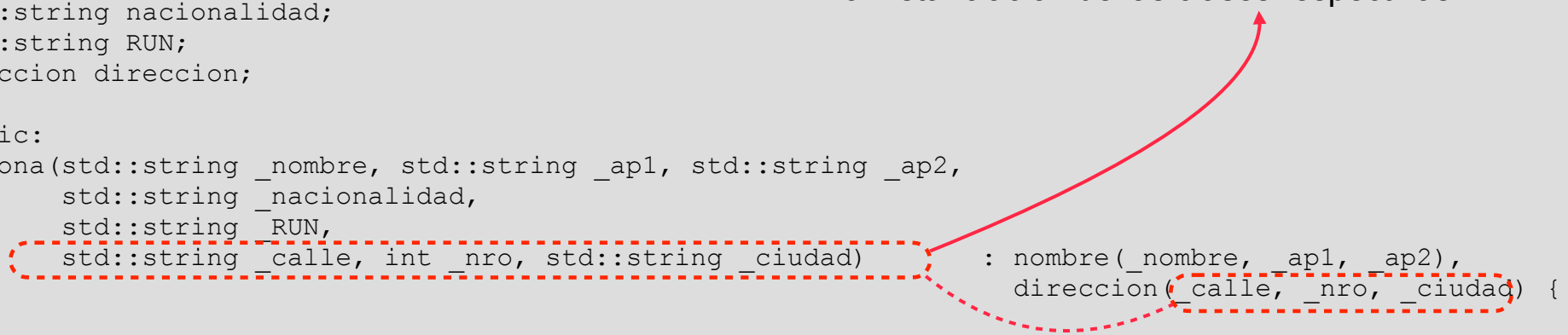
            nacionalidad = _nacionalidad;
            RUN          = _RUN;
        }

        void mostrar(){

        }

};
```

Algunos argumentos serán utilizados en la instanciación de las clases respectivas



# Composición (implementación)

## Implementación de la clase **Persona**

```
class Persona{
    private:
        Nombre nombre;
        std::string nacionalidad;
        std::string RUN;
        Direccion direccion;

    public:
        Persona(std::string _nombre, std::string _ap1, std::string _ap2,
                std::string _nacionalidad,
                std::string _RUN,
                std::string _calle, int _nro, std::string _ciudad): nombre( _nombre, _ap1, _ap2),
                                                                    direccion( _calle, _nro, _ciudad) {

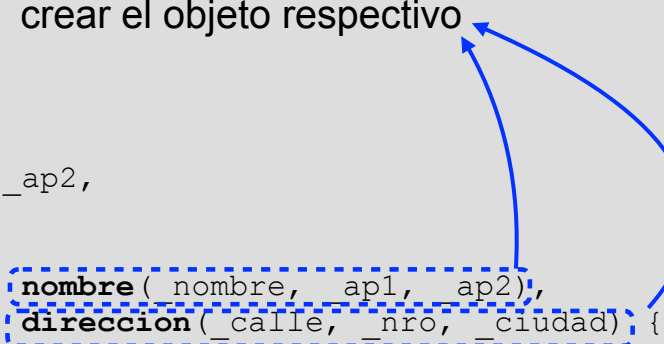
            nacionalidad = _nacionalidad;
            RUN          = _RUN;
        }

        void mostrar() {

        }

};
```

Llama al constructor para  
crear el objeto respectivo



# Composición (implementación)

Uso de la clase  
**Persona**

```
Persona p0("Juan", "Pérez", "Quiroz", "chilena", "11.111.111-1", "12 de Febrero", 123, "La Calera");  
p0.mostrar();
```

# Composición (implementación)

## Observación

En el ejemplo de uso anterior, se optó por diseñar un constructor que tuviese los parámetros necesarios para poder instanciar las clases necesarios

```
Persona p0 ("Juan", "Pérez", "Quiroz",  
            "chilena",  
            "11.111.111-1",  
            "12 de Febrero", 123, "La Calera")
```

Argumentos para la clase **Nombre**

Argumentos para la clase **Direccion**

Otra forma, más apropiada desde el punto de vista de POO, es utilizar objetos

```
Persona p0 (Nombre ("Juan", "Pérez", "Quiroz"),  
            "chilena",  
            "11.111.111-1",  
            Direccion ("12 de Febrero", 123, "La Calera"))
```

# Composición (implementación)

```
Persona p0(Nombre("Juan", "Pérez", "Quiroz"),  
           "chilena",  
           "11.111.111-1",  
           Direccion("12 de Febrero", 123, "La Calera"))
```

Para que lo anterior funcione,  
se debe sobrecargar el  
constructor de la clase  
**Persona**.

```
class Persona{  
    private:  
        ...  
        Nombre nombre;  
        Direccion direccion;  
        ...  
  
    public:  
        ...  
        Persona(Nombre _nombre,  
                 std::string _nacionalidad,  
                 std::string _RUN,  
                 Direccion _direccion) : nombre(_nombre),  
                                       direccion(_direccion) {  
  
            nacionalidad = _nacionalidad;  
            RUN          = _RUN;  
        }  
  
        ...  
};
```

Para que se pueda  
instanciar un objeto a  
partir de otro, es  
necesario que dichas  
clases tengan un  
constructor de copia.

# Composición (implementación)

Agregar constructores de copia a las clases necesarias

```
class Nombre{
    private:
        ...

    public:
        ...

    Nombre(Nombre& n) {
        nombre      = n.nombre;
        apellido1    = n.apellido1;
        apellido2    = n.apellido2;
    }

    ...

};
```

```
class Direccion{
    private:
        ...

    public:
        ...

    Direccion(Direccion& d) {
        calle      = d.calle;
        nro        = d.nro;
        ciudad     = d.ciudad;
    }

    ...

};
```



# Excepciones

# Excepciones

## Pseudo-Código de uso

```
Inicio:
  Intentar ejecutar:
    /* llamada a un método/función
       que puede producir
       un error */
  Capturar excepción
    /*
       bloque que maneja
       el error generado
    */
Fin
```

Mecanismo para manejar errores y situaciones anómalas que pueden surgir durante la ejecución de un programa.

Es un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de las instrucciones.

Cuando se produce una un error, se "lanza" una excepción, que es un objeto que contiene información sobre el problema encontrado.

# Excepciones

## Ejemplo C++

Define un tipo de objeto que se lanzará como excepción.

`std::runtime_error`

```
class Fecha{
private:
    int day;
    [...]

public:
    [...]
    void setDay(int d){
        if(d >= 1 && d <= 31){
            day = d;
        }
        else{
            throw std::runtime_error("Error: " +
                                   std::to_string(d) +
                                   " no es un número de día válido");
        }
    }
    [...]
};
```

Implementación de un método que en caso de una asignación incorrecta, genera una excepción

# Excepciones

## Ejemplo C++

```
Fecha f0;  
int diaRandom = generarNroAleatorio(1, 50);  
  
try{  
    f0.setDay(diaRandom);  
}  
catch(std::exception& e){  
    std::cout << "Error en la asignacion del día\n";  
    std::cout << e.what() << "\n";  
    exit(EXIT_FAILURE);  
}
```

En este caso, se está simulando el uso del método `setDay()` con un argumento que no se conoce el valor exacto.

Si el método envía una excepción, ésta es capturada y se procede con el código que maneja el error

# Excepciones personalizadas

## Ejemplo C++

Se debe crear una clase derivada de la clase principal `std::exception`.

```
class FechaException : public std::exception{
private:
    std::string msg;

public:
    FechaException(std::string m){
        msg = m;
    }
    const char* what() {
        return(msg.c_str()) ;
    }
};
```

# Excepciones personalizadas

## Ejemplo C++

```
Fecha f0;  
int diaRandom = generarNroAleatorio(1, 50);  
  
try{  
    f0.setDay(diaRandom);  
}  
catch(FechaException& e){  
    std::cout << "Error en la asignacion del día\n";  
    std::cout << e.what() << "\n";  
    exit(EXIT_FAILURE);  
}
```

Si el método envía una excepción, ésta es capturada y se procede con el código que maneja el error. Ahora la excepción es personalizada.