

Tema 4. POO. Aspectos Intermedios

- ▶ Constructores y destructor
- ▶ Miembros estáticos
- ▶ Composición: Objetos como miembros de clase
- ▶ Herencia y polimorfismo
- ▶ Métodos en cascada: El uso del puntero this
- ▶ Estructura y organización de proyectos y librerías

Punteros en clases

- ▶ Como en cualquier otro tipo de variable, es posible apuntar a la dirección en donde se almacena un objeto de una clase.
- ▶ Si el ámbito está fuera de la clase, basta con crear un objeto con el operador de indirección `Clase * ptrClase; // Creando un puntero a un objeto de la clase Clase`
- ▶ En cambio, dentro de la misma clase (es decir, dentro del ámbito `Clase::`) existe la palabra reservada **this** que permite obtener la dirección al objeto que ha instanciado la clase.
- ▶ Este puntero **this** es muy útil para varios comportamientos, pero su uso primario es para evitar ambigüedades entre miembros de clase y parámetros de funciones, algunos comportamientos incluyen:
 - ▶ Retornar el objeto: `return *this` retornaría el objeto completo.
 - ▶ Pasar el objeto a otra función: si se declaran otras funciones que utilizan un objeto referenciado, es posible enviar un puntero al objeto como un parámetro `otraFuncion(this);`
 - ▶ Sobrecarga de operador: ¿Como sumar 2 objetos? Tema 5 - Aspectos avanzados.

Puntero this

- ▶ Veamos los ejemplos de uso:
- ▶ Al utilizar this, nos referimos al objeto que ha hecho uso del método, por lo que, podríamos usar un parámetro llamado lado y diferenciarlo del atributo lado al utilizar el puntero this.
- ▶ Entonces si this es un puntero a objeto, podríamos hacer la indirección y retornar una copia del puntero.
- ▶ Por otro lado, si existe una función que recibe una dirección a un objeto de esa clase, podríamos utilizar this como argumento de la función.

```
1  class Cuadrado
2  {
3  private:
4      float lado;
5
6  public:
7      void setLado(float lado)
8      {
9          this->lado = lado;
10     }
11     Cuadrado &retornarCopiaDeObjeto()
12     {
13         return *this;
14     }
15     void pasarObjeto()
16     {
17         funcionQueRecibeUnPunteroaCuadrado(this);
18     }
19 };
20
```

Constructores

- ▶ Utilizando clases, podemos asignar datos iniciales a los atributos de un objeto en el momento de su declaración, para ello hemos de agregar la función **Constructora** de la clase.
- ▶ En el header, debemos agregar una función pública con el Nombre de la clase sin parámetros.
- ▶ En el cpp, debemos escribir la implementación asegurándonos de escribir los datos iniciales con valores por defecto.

Complejo

- real: float
- imag: float

+ Complejo()
+ getters
+ setters

```
1 class Complejo {  
2     private:  
3         .....  
4     public:  
5         Complejo();  
6         .....  
7 };
```

```
1 ...  
2 Complejo::Complejo() {  
3     real = 0;  
4     imag = 0;  
5 }  
6 ...
```

Constructores

- ▶ Esta función que se llama Constructor se encargaría de asignar los datos iniciales, así como realizar cualquier tipo de operación que sea necesaria para que los datos estén correctamente inicializados.
- ▶ También es posible escribir Constructores que reciban parámetros de entrada. Por ejemplo, un número complejo podría tener un valor inicial dado por un par real-imaginario.

Complejo
- real: float - imag: float
+ Complejo() + Complejo(real: double, imag: double) + getters + setters

```
1  Complejo::Complejo() {  
2      real = 0;  
3      imag = 0;  
4  }  
5  
6  Complejo::Complejo(double real, double imag) {  
7      this->real = real;  
8      this->imag = imag;  
9  }  
10
```

Constructores

- ▶ Como vemos, al escribir Clases podemos llamar a funciones con el mismo nombre y con diferentes parámetros de entrada. `Complejo(void)` y `Complejo(double, double)`.
- ▶ Esto se cumple para cualquier número de funciones. Por ejemplo, un número complejo podría tener un valor inicial dado por a) un real, b) un imaginario, c) un par real-imaginario, d) un par magnitud-fase.
- ▶ Entonces podríamos crear diferentes constructores, los cuales reciban diferentes tipos de entrada, pero realicen la misma operación (construir el objeto).

Complejo
- real: float - imag: float
+ Complejo() + Complejo(real: double, imag: double) + getters + setters

Constructores



- Para utilizar cualquier constructor, al momento de declarar una variable ha de utilizarse el nombre de la clase seguido de los paréntesis. Si existen argumentos de entrada específicos, se pueden definir.



```
1  Complejo c1; // Constructor por defecto
2  Complejo c2 = Complejo(); // Aqui se llama a
   1 mismo constructor pero forma explicita
3  Complejo c3 = Complejo(4.5, 1.2); // Aqui se
   llama al constructor que recibe 2 flotantes
```

Complejo

- real: float
- imag: float

- + Complejo()
- + Complejo(real: double, imag: double)
- + getters
- + setters

Constructores



- ▶ Se deben crear tantos constructores como inicializaciones diferentes se deseen.
- ▶ Agregar un constructor que solo reciba un flotante e inicialice o el componente real o el imaginario.
 - ▶ Problema: el parámetro de entrada puede ser nuevo_real o nuevo_imaginario pero ambos datos son de tipo flotante, entonces habrá un problema de ambigüedad.
 - ▶ Solución: Agregar parámetros adicionales.

```
Complejo c4 = Complejo(4.5, true);  
Complejo c5 = Complejo(4.5, false);
```

Complejo

- real: float
- imag: float

- + Complejo()
- + Complejo(real: float, imag: float)
- + Complejo(num: float, es_real: bool)
- + getters
- + setters

Destructor



- ▶ La contraparte principal del constructor es el destructor, es automáticamente invocado cuando el objeto instanciado ya no se encuentra en uso (por ejemplo, fuera de un stack)
- ▶ Tiene la misma forma de escritura que un Constructor vacío, con la adición de la virgulilla ~ (ctrl + 4) frente al nombre del objeto.
- ▶ Es especialmente útil cuando se deban realizar operaciones para liberar memoria(punteros, arreglos y string) o para saber cuándo un objeto ya no está en uso.
- ▶ Para llamar a un destructor se puede:
 - ▶ Utilizar la palabra reservada delete seguido de la dirección al objeto instanciado.
 - ▶ Terminar de utilizar un objeto: Al finalizar toda función se realiza la destrucción de todos los objetos creados en ese ámbito.
- ▶ En el archivo de implementación se pueden escribir instrucciones si se desea

```
Complejo::~~Complejo() {  
    cout << "Se ha llamado al destructor" << endl;  
}
```

Complejo

- real: float
- imag: float

- + Complejo()
- + ~Complejo()
- + getters
- + setters

Miembros Estáticos

- ▶ Normalmente, existen objetos que comparten un dato. Por ejemplo, la clase Circulo necesita de la constante π para obtener el área o el perímetro.
- ▶ Este valor es siempre el mismo 3.1415... y más importante, es un atributo de la clase que no varía de objeto a objeto.

Circulo
- radio: float <u>- pi: float</u>
+ Circulo() + Circulo(radio: float) + area(): float + perimetro(): float + getRadio(): float + setRadio(float radio): void

- ▶ Reconocemos miembros estáticos cuando todos los objetos de una Clase tienen el mismo valor para un miembro específico.
- ▶ En el diagrama representamos a un miembro estático por el subrayado.
- ▶ En C++ agregamos la palabra reservada **static** para denotar miembros estáticos

Miembros Estáticos

- ▶ La definición como variable estática viene dada dentro del header.

```
...  
    static float PI;  
...
```

- ▶ Como no es posible inicializar valores dentro de la definición de una clase, para modificar una variable estática de una clase no hace falta instanciar un objeto de la clase, basta con realizar la asignación usando el operador de resolución de ámbito (::).
- ▶ La forma más común es realizar la modificación dentro del archivo de implementación, pero se puede realizar en cualquier parte de la memoria estática.

```
...
```

```
float Circulo::PI = 3.1416;
```

Miembros Estáticos

- ▶ El keyword **static** no solo está limitado a variables sino también a funciones miembro que sean compartidas.
- ▶ Se reconocen de la misma manera: si dos objetos diferentes utilizan la misma función (con los mismos datos), estamos ante una función miembro estática.

Circulo

- radio: float
- pi: float

+ Circulo()
+ Circulo(radio: float)
+ area(): float
+ perimetro(): float
+ getRadio(): float
+ setRadio(radio: float): void
+ radToDeg(rad: float): float

```
...  
static float radToDeg(float rad);
```

En la declaración debemos incluir la palabra static

En la implementación NO debemos incluir la palabra static

```
...  
float Circulo::radToDeg(float rad) {  
    return 180 * rad / pi;  
}
```

Miembros Estáticos

>04

- Para utilizar cualquier miembro estático podemos hacer la llamada desde un objeto o desde la resolución de ámbito. De preferencia habrá que utilizar este último método.



```
1  cout << "Ejemplo 4" << endl;
2  cout << "Usando metodos estaticos con resolucion de ambito" << endl;
3  cout << "El valor de PI es: " << Circulo::PI << endl;
4  cout << "0.8 radianes equivale a: " << Circulo::radianesAGrados(0.8) << " Grados de Euler" << endl;
5
6  Circulo c1 = Circulo(4.5);
7  cout << "Usando metodos estaticos desde un objeto" << endl;
8  cout << "El valor de PI es: " << c1.PI << endl;
9  cout << "pi radianes equivale a: " << c1.radianesAGrados(3.14159265) << " Grados de Euler" << endl << endl;
```

PОО Aspectos Intermedios

- ▶ Ninguna de las funciones de construcción puede ser estática ya que cada objeto debe ser diferente de otro y el constructor es el encargado de la asignación de objetos nuevos.
- ▶ Los constructores pueden describirse más de una vez, pero debe escribirse siempre con diferentes datos de entrada, siempre que no exista ambigüedad. Eso significa que Clase(int A) y Clase(int B) no definen diferentes constructores, mientras que Clase(int A) y Clase(float B) , si lo hacen.
- ▶ De hecho, todas las funciones de una clase pueden volver a escribirse con el mismo nombre, pero con diferentes datos de entrada. Esta técnica se utiliza para agrupar comportamientos de semánticas similares.
- ▶ Esto se define en PОО como **sobrecarga de funciones** (method overloading), en donde se puede tener el mismo nombre de función para operaciones diferentes.

Sobrecarga de funciones

- ▶ Agreguemos a nuestra clase Circulo la capacidad de obtener cuerdas.
- ▶ Normalmente será útil determinar la cuerda con respecto a dos puntos cualesquiera del círculo, pero seguramente también será útil determinar la cuerda de un punto cualquiera con el extremo derecho del círculo.

Circulo

- radio: float
- pi: float

...

+ getCuerda(angulo_x: float): float
+ getCuerda(angulo1_x: float, angulo2_x: float): float

...



```
1 float Circulo::getCuerda(float angulo_rad) {  
2     return 2 * radio * sin(angulo_rad / 2);  
3 }  
4  
5 float Circulo::getCuerda(float ang1_rad, float ang2_rad) {  
6     return 2 * radio * sin(abs(ang2_rad - ang1_rad) / 2);  
7 }
```

Sobrecarga de funciones

- ¡Cuidado! La sobrecarga de funciones define NO diferentes funciones para funciones con los mismos parámetros de entrada, a pesar de tener diferentes nombres de variables.

Circulo

- radio: float

- pi: float

...

+ degToRad(float degrees): float

+ getCuerda(float angulo_x): float

+ getCuerda(float angulo_rad_x): float

...

```
float Circulo::getCuerda(float angulo_x) {  
    return 2 * radio * sin(angulo_x / 2);  
}  
  
float Circulo::getCuerda(float angulo_rad_x) {  
    return 2 * radio * sin(Circulo::degToRad(angulo_rad_x) / 2);  
}
```


Sobrecarga de funciones



- ▶ Utilizar estas funciones sobrecargadas es igual que cualquier función común y corriente.
- ▶ Ha de tenerse en cuenta que los argumentos de entrada tienen que corresponder con una de las firmas de las funciones sobrecargadas.





```
1  Circulo c2 = Circulo(10);
2  float radianes = 30 / Circulo::radianesAGrados(1);
3  float cuerda = c2.getCuerda(radianes);
4  cout << "La cuerda desde el eje de abscisas a " << radianes << " rad. es: " << cuerda << endl;
5  float rad2 = 90/Circulo::radianesAGrados(1);
6  cuerda = c2.getCuerda(radianes, rad2);
7  cout << "La cuerda entre los angulos " << radianes << " rad. y " << rad2 << " es: " << cuerda << endl;
```

Composición y Agregación

- ▶ Continuando con el paradigma de Programación Orientada a Objetos, cada clase debe ser diseñada de acuerdo con el conjunto de datos que conforman al objeto.
- ▶ Así, por ejemplo, un Coche debe tener atributos como, por ejemplo
 - ▶ velocidad: float
 - ▶ gasolina: float
 - ▶ ruedas[4]: Rueda
 - ▶ motor: Motor
- ▶ En donde podemos deducir que Rueda es a su vez una clase nueva e independiente del Coche (Una rueda existe sin un coche.)
- ▶ Asimismo, Motor es otra clase nueva porque un motor existe sin un coche.
- ▶ En este caso, se dice que una clase Base está relacionada con una Clase Completa.

Composición y Agregación

- ▶ En ambos ejemplos, la Clase Completa (Coche) no puede existir sin las Clases Base (Ruedas y Motor).
- ▶ La **Composición** es el mecanismo por el cuál una clase se construye a partir de otras, pudiendo compartir funcionalidades y operar unas sobre otras en donde la Clase completa no puede existir sin la Clase Base.
- ▶ La **Agregación** tiene la misma definición con la diferencia de que la Clase Completa puede existir sin la Clase Base.
- ▶ Como las clases Coche y Motor son diferentes, se deben diagramar en recuadros diferentes y para definir las relaciones se utiliza:
la flecha A  B para definir que B es una clase compuesta por A.
la flecha A  B para definir que B es una clase que agrega a A.

Composición y Agregación

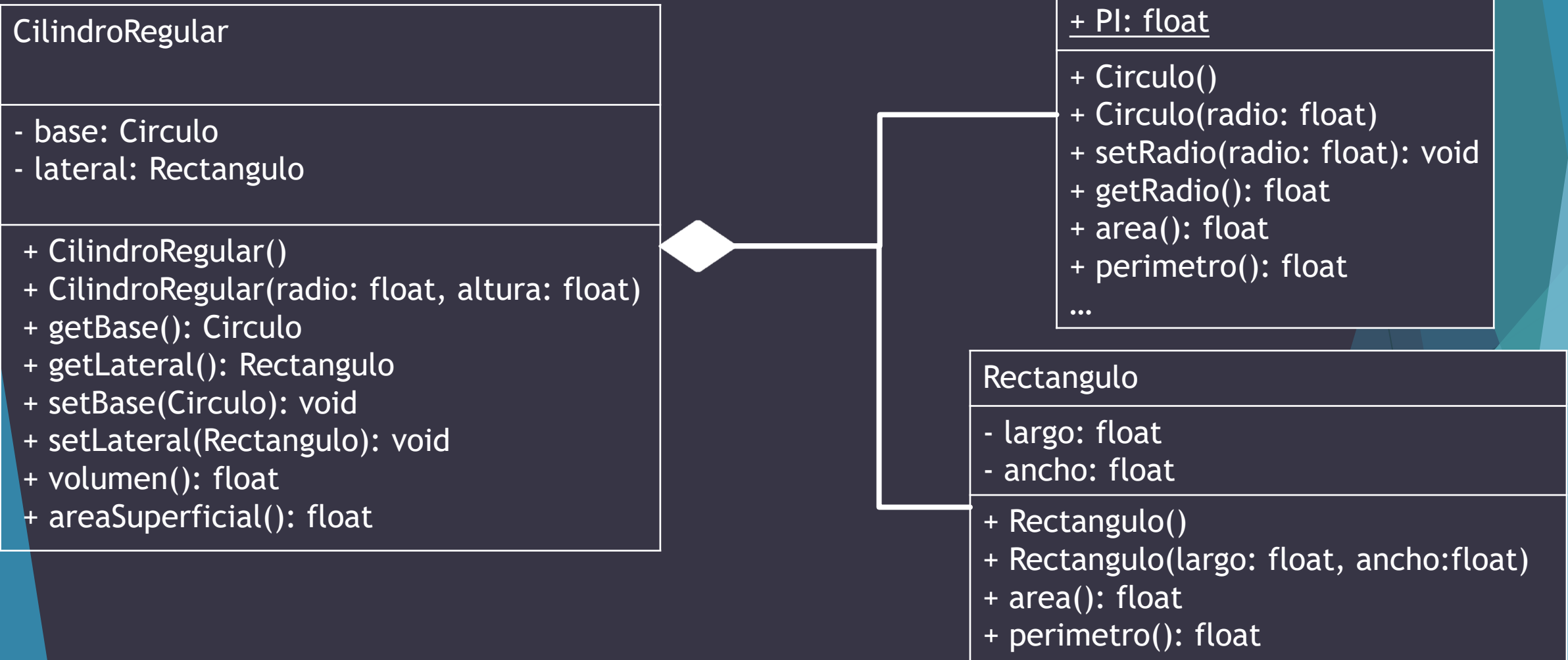
Clase Base	Clase Completa	Composición/Agregación
Motor	Coche	Composición
Ratón	Ordenador	Agregación
Folio	Libro	
Salón	Piso	
Pantalla	Móvil	
Billetes	Cartera	
Alumno	Curso	

- La definición de Composición o Agregación se hará según el grado de utilidad. Un Curso real puede existir sin alumnos, pero no es útil, por tanto, es una Composición.

Composición y Agregación

> 06

- Crear un cilindro regular compuesto por un Circulo y un Rectángulo.



Composición y Agregación “tiene un/una”



La composición y la agregación son definidas cuando un objeto se describe con el uso de otro (una relación): en diagrama se suele decir “claseA usa/utiliza/tiene una claseB”.



En la mayoría de los casos, esto significa exclusivamente que la Clase Base es siempre un atributo de la Clase Completa, ya sea como una sola variable (Motor, Coche), como un vector de variables (Paginas, Libro) o como un puntero (Persona* (Dueño del Perro), Perro).



Pero, en cualquier caso, siempre se accede a la Clase Base a través de la Clase Completa y nunca en el otro sentido (Se puede acceder a Billetes en una Cartera, pero no a una Cartera teniendo un Billeto).



Y aún más importante, en Composición y Agregación la Clase Base NO define comportamientos de la Clase Completa ni viceversa.

P00 Aspectos Intermedios

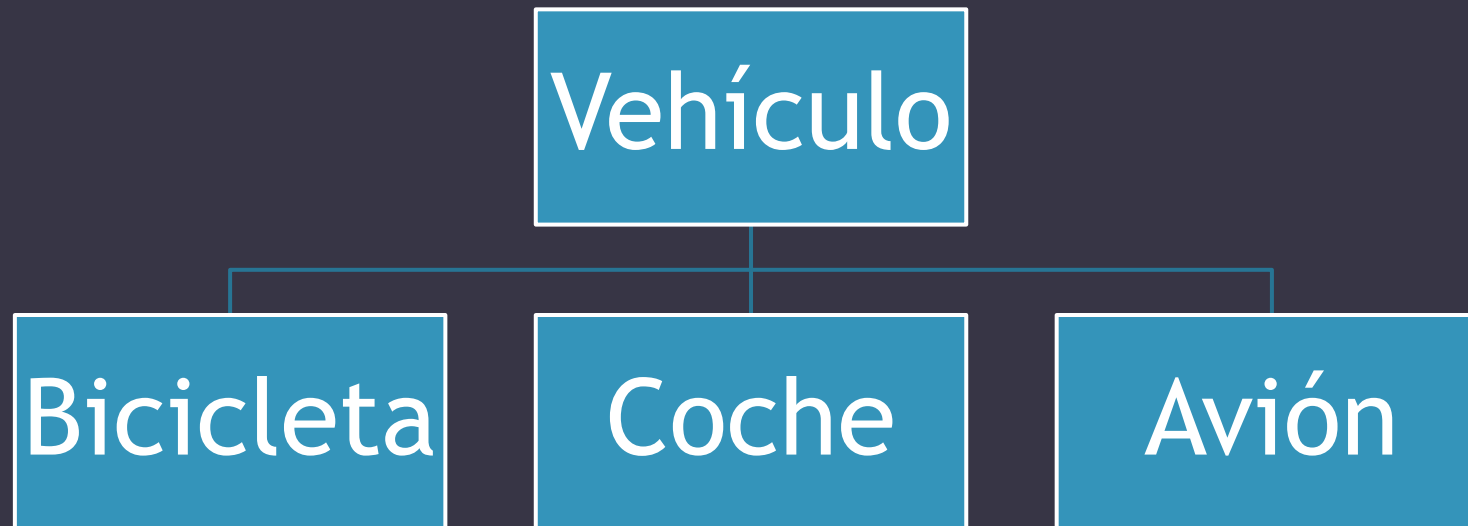
- ▶ La relación tiene-un tiene-una no siempre se puede asignar a objetos que están relacionados.
- ▶ Un Coche tiene ruedas, una bicicleta tiene ruedas.
- ▶ Pero un coche no es una bicicleta, pero, un coche y una bicicleta comparten una relación: ambos son **vehículos**.
- ▶ En este caso la relación de ambos se generaliza mediante una clase principal.
- ▶ Se diferencia a esta relación utilizando “es-un o es-una” y se extiende la definición y utilidad para reducir la cantidad de código escrito para realizar operaciones y guardar atributos semejantes.

PОО Aspectos Intermedios


- ▶ En una empresa, el Empleado no está compuesto por una Persona, el Empleado ES-una Persona.
- ▶ Aquí la generalización es la Persona, que se puede definir de cualquiera de las siguientes maneras:
 - ▶ Clase Principal
 - ▶ Clase Base
 - ▶ SuperClase
- ▶ Asimismo, la clase Empleado llevaría cualquier de los siguientes nombres:
 - ▶ Clase Secundaria
 - ▶ Clase Derivada
 - ▶ SubClase

Herencia “es un/una”

- ▶ De la relación Parent-Child se define el término Herencia.
- ▶ La herencia es un término de POO que indica que existe una Clase Principal, la cual “hereda” a una Clase Secundaria su funcionalidad.
- ▶ La superclase o clase principal tiene un conjunto de atributos y métodos que son completamente útiles en la subclase, por lo que estas dos clases pueden relacionarse mediante “es-un” y comparten todas estas funcionalidades.

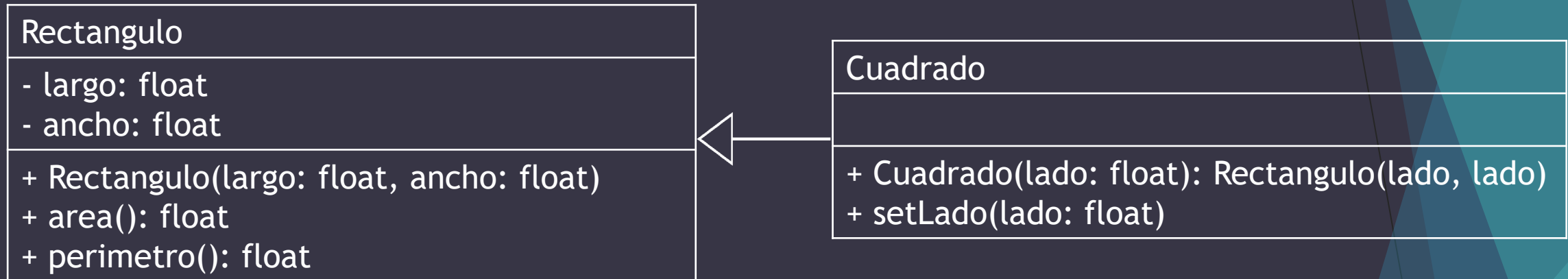


Herencia “es un/una”

- ▶ Para crear clases principales y secundarias ha de utilizarse la organización en jerarquías.
- ▶ Clases superiores en la organización definen comportamientos más generales, mientras que los comportamientos más específicos se definen en clases de menor jerarquía.
- ▶ Ejemplo:
 - ▶ Clase Base: Vehículo definirá atributos como velocidad, dirección, nombre
 - ▶ Clase Derivada: Bicicleta definirá atributos como angulo_manillar, ruedas y métodos como avanzar, frenar, etc.
- ▶ Para diagramar, se debe utilizar la flecha A  B para indicar que B es la clase Principal y A es la clase Secundaria (también se puede decir que B es más general que A o bien que A es una especialización de la clase B).

Herencia “es un/una”

- ▶ En C++, para heredar una clase ha de utilizarse el operador : durante la definición de la clase, seguido del nivel de acceso garantizado (generalmente Public) y la Clase principal.



- ▶ Entonces en según el diagrama mostrado, la definición de Cuadrado debe ser:

```
class Cuadrado: public Rectangulo {
public:
    ...
}
```

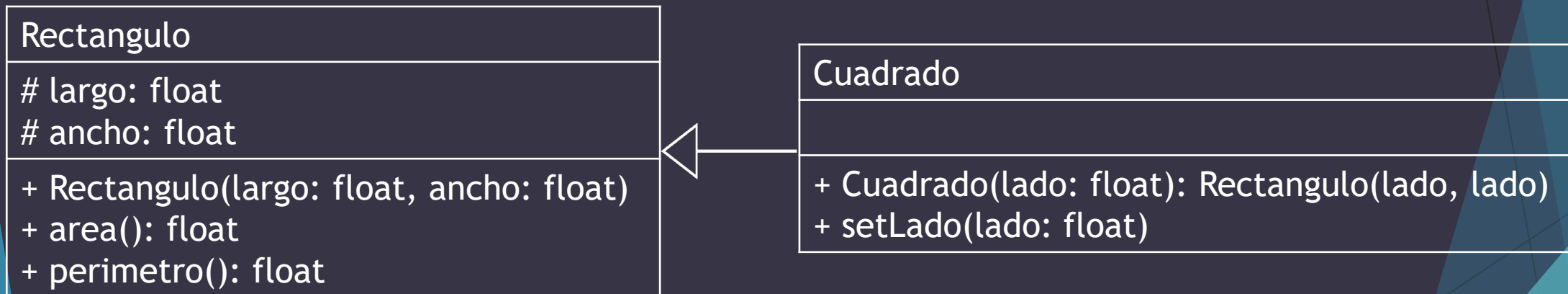
- ▶ Las buenas prácticas indican que cada constructor de la clase superior debe poder ser utilizada desde la clase inferior, esto también se realiza con el operador :

Herencia “es un/una”

- ▶ Según el ejemplo mostrado es posible crear un objeto Rectangulo c1 y un Cuadrado c2.
- ▶ Para inicializar c1 ha de utilizarse Rectangulo(lado, lado), mientras que para inicializar c2 ha de utilizarse Cuadrado (lado).
- ▶ Ambos mostrarían el mismo perímetro y la misma área con las respectivas funciones ya que el nivel de acceso de herencia es **public** haciendo que todas las funciones públicas en Rectangulo también sean públicas en Cuadrado.
- ▶ La herencia de la clase Rectángulo ha permitido escribir menos código en la clase Cuadrado, pero, por ejemplo, un método setLado(float l) no podría permitir la compilación del programa por un error de acceso.
- ▶ En este caso, necesitamos que las variables de largo y ancho puedan ser accedidas por la clase Hija, para ello, los atributos y métodos de la clase Padre dejen de ser privadas (-) y sean protegidas (#)

Herencia “es un/una”

- ▶ En el diagrama basta con cambiar de - a #, mientras que en C++ se debe modificar la palabra **private** a **protected**. En caso de que existan variables privadas que no sea accesibles, se debe dejar el apartado **private**.
- ▶ Las variables protegidas solo pueden ser accedidas por la misma clase y las subsecuentes clases hijas.



- ▶ Recordar: Las llamadas a funciones o métodos de las clases superiores también se realizan usando el operator : justo antes de definir el bloque de implementación de la función heredada.

Herencia “es un/una”



- ▶ Al utilizar herencia se mejora considerablemente el tiempo de codificación, pero las ventajas no son solo a la hora de redactar una clase, sino también a la hora de utilizar muchos objetos de la misma clase base.
- ▶ En el ejemplo anterior hablábamos de dos objetos con los mismos lados

```
Rectangulo r1 = Rectangulo(5.0, 5.0);  
Cuadrado c2 = Cuadrado(5.0);  
cout << "El area del rectangulo es: " << r1.area() << endl;  
cout << "El area del cuadrado es: " << c2.area() << endl;
```

- ▶ Pero también es posible tener un arreglo de clases Base y definir los elementos como clases Derivadas sin problemas.

```
Rectangulo rects[2];  
rects[0] = r1;  
rects[1] = c2;  
cout << "El perimetro del rectangulo es: " << rects[0].perimetro() << endl;  
cout << "El perimetro del cuadrado es: " << rects[1].perimetro() << endl;
```

Aspectos intermedios

- ▶ Normalmente toda Clase necesita una función que permita imprimir en pantalla todos sus atributos.
- ▶ Para ellos, es una práctica común definir el método toString() que devolverá toda la información que es necesaria visualizar.

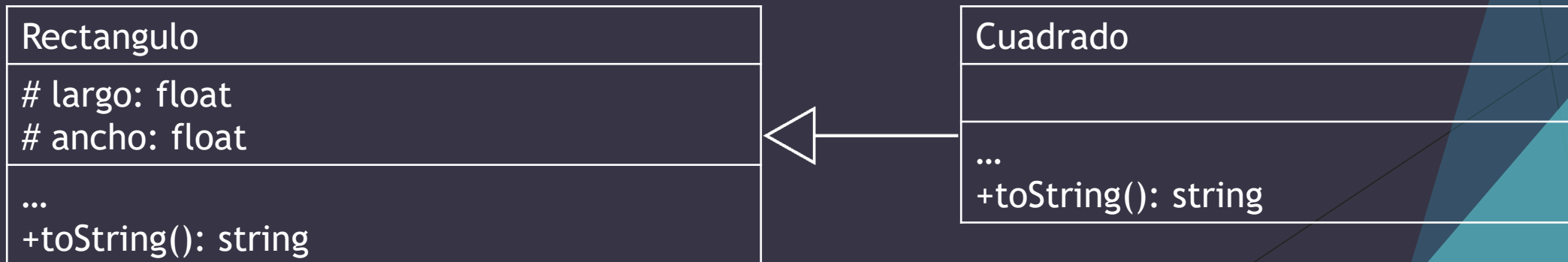
Rectangulo
largo: float # ancho: float
+ Rectangulo(l:float,a:float) + setLargo(l:float) : void + getLargo() : float + setAncho(float a) : void + getAncho() : float + area() : <u>float</u> + perimetro() : float + toString(): string

```
1. string Rectangulo::toString(){  
2.     string s = "El rectangulo tiene largo " +  
   to_string(largo) + " y ancho " + to_string(ancho);  
3.     return s;  
4. }
```

- ▶ C++ incluye el método to_string(tipo) dentro de la librería string para transformar los tipos primitivos a strings para poder visualizarlos.

Polimorfismo

- ▶ Por supuesto que, si el método toString debe estar en todas las clases, en las subclases heredadas también deseamos reimplementar este método toString a algo que sea más específico.
- ▶ Por suerte, C++ permite dicha reimplementación en las subclases con el mismo nombre.
- ▶ Con esto es siempre posible tener el método toString en todas las superclases y sus subclases.



Polimorfismo

- ▶ Al definir el método toString en la clase especifica, la función existe en este contexto, pero también en la clase general.

```
1. string Cuadrado::toString(){  
2.     string s = "El cuadrado tiene lado " + to_string(largo);  
3.     return s;  
4. }
```

- ▶ A pesar de esta dualidad, al invocar a la función, según el objeto que llama a la función se llamaría a una o a otra función.
- ▶ Este comportamiento se llama **polimorfismo**, en donde la función toma la forma según el objeto que invoca a la función.
- ▶ Con esto, el Rectangulo c1 y un Cuadrado c2 retornarán distintos datos para la misma función toString().

Metodos Virtuales



- Pero en el ejemplo mostrado, c2 utiliza el toString de la clase derivada, pero rects[1] no lo hace, a pesar de que rects[1] = c2

```
cout << r1.toString() << endl;  
cout << c2.toString() << endl;
```

```
cout << rects[0].toString() << endl;  
cout << rects[1].toString() << endl;
```

- Para un mejor control de programación, se deben incluir algunas palabras claves que facilitaran la sobrescritura de funciones:
- **virtual**: Como modificador de los métodos en las clases, base o derivadas, que deseen que dichos métodos sean sobrescritos.

Metodos Virtuales

- ▶ Ambas palabras reservadas se utilizarán exclusivamente en las declaraciones.
- ▶ `virtual`: Como modificador de los métodos en las clases superiores que deseen que dichos métodos sean sobrescritos.
- ▶ Entonces, en Rectángulo se debe agregar el método:
- ▶ `virtual string toString();`
- ▶ Por último, en Cuadrado se debe agregar el mismo método, con la misma firma.
- ▶ `string toString();`
- ▶ También es posible volver a nombrar `toString` de Cuadrado como `virtual` si existirá una tercera clase que herede de Cuadrado y no de Rectángulo (Tema 5).

Metodos Virtuales

- ▶ En la mayoría de las veces, la función hecha `virtual` puede generar datos válidos para clases derivadas.
- ▶ Por ejemplo, en caso de un Cuadrado como clase derivada de la clase Rectángulo, la función `perimetro()` retorna $2 * (\text{lado} + \text{lado})$ que es una función válida para obtener $4 * \text{lado}$.
- ▶ Esto significa que esta función `perimetro()` en la clase superior, si bien, no es específica, generaliza el comportamiento de la función deseada.
- ▶ Pero existen casos en que este método virtual no tiene forma alguna lógica de describir su funcionamiento de forma general, entonces no puede definirse una implementación generalizada. En ese caso, el método debe igualarse a 0:

```
virtual float metodoVirtual(float x) = 0;
```

- ▶ Esto define a la clase base como una **Clase Abstracta**, y con esto la clase no permite instanciar ningún objeto.

Clases Abstractas

- ▶ Las clases abstractas son útiles cuando existen varias clases derivadas que compartirán funciones con semánticas similares.
- ▶ Con esto, la persona a cargo de desarrollar la clase derivada está obligada a implementar dicha función.
- ▶ Para diagramar, basta con poner en letra cursiva el nombre de la clase (si es clase abstracta) y las funciones virtuales (si son virtuales).
- ▶ Algunos ejemplos de uso para Clases Abstractas incluyen:
 - ▶ Polígono: con función de área que varía la disposición geométrica del polígono.
 - ▶ Filtro: la clase Base define los nombres de las funciones y las derivadas las diferentes funciones matemáticas de filtrado.
 - ▶ Electrodoméstico: Lavadora y Nevera son electrodomésticos que deben implementar la función `consumirPotencia()` y `utilizar()` pero tienen diferentes comportamientos.

Clases Abstractas



- ▶ Se deben crear las clases Polinomial, Exponencial. Estas clases implementarán las funciones evaluar(), diferenciar() e integrar(). Cree el diagrama de la clase abstracta Función que junte tanto las funciones virtuales como los atributos, si los hubiere.
- ▶ Implementar la clase abstracta en C++.

