TEMA 4

Introducción a la POO. Clases e instancias. Atributos, métodos y mensajes. Encapsulamiento. El modelo de Objetos en C++. Constructores y destructores. Sobrecarga de métodos. Problemas de programación.

Programación Orientada a Objetos

Es un *paradigma* de programación que resuelve problemas mediante conjuntos de objetos que interactúan entre sí.
Objetivos: El software se vuelve fácilmente modificable, escalable y reusable y además se dispone de un modelo natural para representar un dominio.
Ventajas: el software se reutiliza, es más flexible y más cercano a la forma de pensar de las personas.

Introducción a la POO

Objetos

Es una entidad que encapsula datos (***atributos***) que describen el estado del objeto y comportamientos (***métodos***) que son las acciones que el objeto puede realizar dentro de una unidad independiente. Es también una *instancia de una clase*, lo que significa que hereda las características y comportamientos definidos en esta.
Los objetos interactúan entre sí mediante ***mensajes***, que es la solicitud para que un objeto ejecute uno de sus ***métodos***. Los mensajes pueden incluir ***parámetros*** que se utilizan por el método para realizar la acción especificada en el mensaje. Los métodos pueden devolver información al objeto emisor del mensaje para que realice otras acciones. Ejemplo: el objeto *Persona* envía el mensaje *comer(pizza)* al objeto *Comida*.
Todo objeto tendrá además un ***estado***, que viene determinado por los valores que toman sus datos en un momento determinado y una ***identidad***, por la que será siempre diferenciable del resto.

Clases

Es un bloque que contiene los atributos y métodos necesarios para crear un conjunto de objetos según un modelo predefinido. El término atributo se utiliza en análisis y diseño orientado a objetos y el término variable instancia se suele utilizar en programas orientados a objetos.
En C++ los métodos se denominan ***funciones miembro*** y al momento de su declaración se relacionan con la clase con "::". Ejemplo: `void clase::método (...){...}`

Constructores

Son métodos identificadores de los objetos que llevan el mismo nombre de la clase y se utilizan para inicializar objetos con los mínimos datos necesarios al momento de su creación y así controlar su estado inicial. No poseen valor de retorno y no pueden ser invocados directamente como otro método.

Constructores de copia

Se invoca al crear un objeto a partir de otro, al pasar un objeto por valor a una función o cuando se devuelve un objeto por valor desde una función.

```
Clase(const Clase &copia); <- Forma de declarar un constructor copia
Clase::Clase (const Clase &c){} <- Otra forma de declararlo
```

Destructor

Método particular para destruir un objeto de una clase determinada liberando su memoria y los recursos que hubiera utilizado (archivos, conexiones a DB, etc.). No reciben parámetros, ni devuelven valores, no pueden ser heredados y deben ser públicos. Es recomendable definirlo en caso de utilizar memoria dinámica o punteros o si se quiere trabajar con logs.

```
Clase::~Clase(){
delete[] this->nombre;
}
```

Siempre que la clase sea padre debe definirse su destructor como *virtual* para evitar que al eliminar una clase hija a través de un puntero de la clase padre no se llame al destructor de la hija y se generen memory leaks o los recursos no se liberen correctamente.

Contención o composición

Es una relación donde una clase *tiene* otras clases como miembros y las clases contenidas se crean y destruyen junto a la principal. Es de propiedad fuerte ya que implica dependencia de existencia: *La clase contenida no tiene sentido por sí sola sin la existencia de la clase contenedora*. Por ejemplo: un motor no tiene sentido sin un auto.
Esta relación implica que la clase contenedora puede utilizar los métodos de la clase contenida, pero si se quiere que la clase contenida acceda a métodos o atributos del contenedor, este debe serle pasado como referencia o declarado como ***friend*** en la clase contenida.

Pilares de la POO

Abstracción

Es el poder diferenciar entre las propiedades externas y la composición interna de una entidad, ignorando los detalles propios del objeto para poder usarlo como una única unidad comprensible. Mediante la abstracción se diseñan y fabrican estos sistemas complejos en primer lugar y, posteriormente, los componentes más pequeños de los cuales están compuestos.
En el modelado orientado a objetos significa centrarse en *qué es* y *qué hace* en lugar de *cómo* debe implementarse. Al imaginar los componentes en función de la abstracción (***herramientas abstractas***) es posible centrarse en cómo cada uno interactúa con los otros y no perderse en detalles.

Encapsulamiento

Es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación, asegurando que la información de un módulo no sea accesible ni visible desde el exterior.

Es el principio fundamental que une ***atributos*** y ***métodos***. Los ***atributos*** son generalmente ***privados***, lo que significa que sólo los ***métodos*** del objeto pueden acceder a ellos, y los ***métodos*** son ***públicos***, lo que significa que pueden llamarse desde fuera del objeto.
Esto permite controlar el acceso a los datos del objeto y protegerlos de cambios no autorizados.

Es un buen método para reutilizar código, como si fuera herencia, pero la composición debe utilizarse cuando necesitan reutilizarse métodos y no corresponde el uso de herencia. Ej.: la clase *Libre* y la clase *Regular* usan el método calculadora (de la clase Alumno) para saber si el alumno aprueba o no.

Friend

Permite que una **función o clase externa** tenga acceso a los **miembros privados y protegidos** de otra clase, pero **sin ser parte de ella**.

Herencia

Es un mecanismo que permite crear una clase (***hija***) a partir de una clase existente (***padre***), **heredando sus atributos y métodos**. Al utilizar herencia, puede reutilizarse el código y aplicar *polimorfismo*.

La ***clase padre*** es general y describe atributos y comportamientos comunes para que la ***clase hija*** pueda reutilizar todo lo que herede, ampliando sus atributos y comportamiento o redefiniéndolos. Los constructores **no** se heredan, pero pueden invocarse y los métodos y atributos **privados** se heredan conceptualmente, pero no son **accesibles directamente**. Para que haya herencia debe trabajarse con el tipo ***protected***.

Un gran problema del uso de herencia es que genera acoplamiento **fijo** (peor que en composición) lo que genera "necesidad". Por ejemplo, si la función A necesita usar a B, A está acoplada a B. Un peor caso de acoplamiento se da en la ***herencia múltiple***, donde una clase hija tiene múltiples padres.

```
class ClasePadre:public ClaseHija; //donde los : indican herencia.
```

Tipos de herencia en C++

- **Public:** Cualquier miembro público de una clase es accesible desde cualquier otra parte donde sea accesible el propio objeto.
- **Protected:** Respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases hijasse comporta como el público.
- **Private:** Sólo son accesibles por los propios miembros de la clase, pero no desde funciones externas o desde funciones de clases hijas. Es el acceso por defecto.

Sobreescritura de métodos

Se da cuando una clase hija **redefine un método** ya existente en la clase padre para cambiar su comportamiento, permitiendo utilizar el mismo nombre del método, pero con diferente comportamiento.

Al momento de sobreescribir métodos es importante resaltar el término ***virtual***: habilita el *polimorfismo dinámico*, permitiendo que un método pueda ser sobreescrito en clases hijas y que la

versión “correcta” del método se ejecute **en tiempo de ejecución** y **según el tipo real del objeto** y no del puntero/referencia.

Si no se utilizara virtual se generaría una vinculación estática, donde se resuelve a qué método llamar en **tiempo de compilación** y **según el tipo de puntero/referencia**, no del objeto real. Es por esto que para poder utilizar sobreescritura en la *clase padre* el método debe ser ***virtual***. Además, en la *clase hija* el la redefinición debe ser con la misma firma, pero si una hija sobreescribe un método, el método del padre queda inaccesible, es decir, cuando se llame al método **siempre** se tomará el de la hija.

```
virtual void Metodo()
```

Métodos y clases abstract

Una ***clase abstracta*** es una clase que **no puede instanciarse directamente** porque está diseñada para ser una **base común** para otras clases. Se utiliza para definir un **comportamiento general** pero sin **implementación concreta**. Para que una clase sea abstracta debe tener por lo menos un ***método virtual puro.***

Un ***método virtual puro*** es un método **sin implementación en la clase padre**, lo que **obliga a las clases hijas** a implementarla.

```
virtual void Metodo() = 0; //El = 0 indica que el método es virtual puro
```

En el caso de que una hija no implemente el método, ésta se vuelve abstracta y tampoco puede instanciarse.

Llamada al constructor de la clase padre en una jerarquía de herencia

Es una relación de herencia, cuando se crea un objeto de una hija, **primero se ejecuta el constructor del padre.** Y si el padre tiene un constructor con parámetros, la hija **debe llamarlo explícitamente** en la lista de inicialización. Ejemplo: Alumno es hija de Persona, por lo tanto, al crear un Alumno **se llama primero al constructor de Persona** y luego se ejecuta el constructor de alumno. Esto asegura que el nuevo objeto se construya correctamente en la jerarquía.

Casting

Proceso de convertir un **valor o referencia de un tipo de dato a otro**. En herencia se encuentran dos tipos:

Downcasting

La conversión se hace de la clase base a la clase derivada. Requiere el uso de **casting explícito** y es inseguro si el objeto no es realmente del tipo derivado, generando errores de casteo. Ejemplo: puedo pedir notas a Personas, pero si llega un Profesor se genera un error.

Upcasting

Es la conversión de un objeto derivado a su base. Se realiza de forma **implícita**, por lo que es más seguro. Ejemplo: Profesor y Alumno dan asistencia de manera diferente de forma presencial, pero si se utilizara un formulario dan asistencia de la misma forma.

dynamic_cast

Es un operador de conversión utilizado para convertir punteros o referencias dentro de una jerarquía de clases polimórficas, verificando **en tiempo de ejecución** que la conversión es segura.
Es principalmente utilizado para **downcasting seguro** y solo funciona si la clase base tiene al menos **un método virtual** (o sea, si la **jerarquía es polimórfica**).

Polimorfismo

Es la capacidad de un objeto de **comportarse de diferentes formas según el tipo real**. Ejemplo: el método `cobrar_envio();` trae el precio de moto o auto según corresponda.

El polimorfismo se implementa con ***punteros***, porque debe referenciar a la clase y no ser parte de ella, ***virtual***, debido a que habría ligadura estática (static binding) y el compilador determina **en tiempo de compilación** que versión del método usar en base al **tipo** y no del objeto real, y ***herencia***, al chequeo por tipo (tipado estático), ya que C++ no sabe si el método admite ese tipo o no.

Si bien es necesario el uso de ***virtual***, no es necesario que la clase padre sea abstracta.

Interfaz de un objeto

Es el **conjunto de métodos públicos** que el objeto expone para que otros objetos puedan interactuar con él, **sin saber su implementación interna** permitiendo ***encapsulamiento***, facilitando la ***modularidad*** y haciendo posible el uso de ***polimorfismo***.
Se representa mediante una ***clase puramente abstracta***.

Otras características

Sobrecarga de operadores

Es **definir su comportamiento para tipos definidos por el programador**, permitiendo usar operadores con objetos como si fueran tipos primitivos.
Los únicos operadores que no pueden sobrecargarse son `::` `.` `.*` y los operadores internos (`sizeof`, `alignof`, `etc`).
Para `<<` y `>>` debe utilizarse ***friend***.