CURSO C++ 123_138

Vídeo 123: ¿Qué es la programación orientada a objetos?

El término “**objeto”** puede tener distintos significados dependiendo del contexto. En POO un objeto es un espacio en memoria que almacena datos y acciones encapsulándolas en un paquete autónomo y reutilizable

Los objetos en POO siempre tienen dos componentes principales:

1. La lista de propiedades relevantes (similares a los objetos tradicionales)
2. Comportamientos que el objeto POO puede llevar a cabo

En POO sus propiedades (**variables miembro**) y acciones (**funciones miembro**) son inseparables. Encapsuladas en un paquete autónomo y reutilizable.

Toda nueva clase que creemos en C++ crea un nuevo tipo asignable y exportable a otros programas.

En C++ moderno **Clases** y **structs** son casi exactamente lo mismo y se usan indistintamente. La única diferencia es que de modo predeterminado las clases definen a sus miembros como **privados** mientras que en los struts de forma predeterminada sus miembros son **public**. Esto se puede modificar con los “**especificadores de acceso”** que se verán más adelante.

Cuando declaras la clase o el struct no le asignamos ningún espacio en memoria. Para entenderlo, una clase es como un plano que diseña un nuevo tipo con el que después podemos crear objetos de ese tipo que sí ocupan espacio en memoria. Hasta que no instancias el objeto no ocupa memoria.

La definición de las clases y structs deben terminar con un punto y coma “`;`” después de la llave de cierre o tendremos un error de compilación.

Para **instanciar una clase**, creamos variables del tipo de nuestra clase y los inicializamos con sus miembros, es entonces cuando ocupan memoria.

A las funciones definidas dentro de la clase se les llama **funciones miembro**, o también **métodos**. Estas funciones se pueden definir tanto dentro como fuera de la clase, pero se verá más adelante para no complicarnos.
Para acceder a variables o funciones miembro de una clase, se utiliza el **operador de selección de miembro** punto “`.`”

Las mejores prácticas aconsejan iniciar el nombre de las clases y los structs con una letra mayúscula.

Vídeo 124: Especificadores de acceso a Clases. public: y private:

Funciones miembro no necesitan declaración anticipada.

Además de variables y funciones, las clases también pueden tener "**tipos miembro**", conocidos como **tipos anidados**. Suelen usarse en plantillas de clases como alias de tipos. Para acceder al alias de tipo desde fuera de la clase, debemos usar la clase (no un objeto) como namespace del alias de tipo.

```
Nombre_clase::tipo_alias
```

También suelen utilizarse para simplificar tipos muy largos y confusos. Recordar que para los alias había que utilizar `using nombre_alias = tipo;`

Aunque se pueden anidar clases dentro de clases, no es muy aconsejable y no suele usarse, salvo excepciones como los alias de tipos. Incluso estos alias se aconseja solo utilizarlos dentro de la clase.

Los miembros **públicos** de una Clase o Struct son accesibles desde cualquier parte del programa. Los miembros **privados** solo desde dentro de la propia Clase o Struct.

Los especificadores de acceso a clase en C++ son:

- **public:**
- **private:**
- **protected:**

Podemos especificar distintos especificadores de acceso dentro de una misma clase. La norma general es que las variables miembro sean privadas y las funciones miembro públicas, pero hay muchas excepciones y depende de las necesidades en cada caso.

Al grupo de miembros públicos de una clase se le conoce como "**interfaz pública**" de la clase.

En cuanto a la colocación de si antes los miembros privados o públicos va por gustos. Hay quien prefiere poner lo privado antes y otros al final, ya que no afecta la colocación de estos dentro de la clase.

Los especificadores de acceso controlan el acceso a los miembros de una clase, no a los objetos de esa clase. Por lo tanto, una función miembro que pueda acceder a los miembros privados de su Clase, puede acceder a los miembros privados de todos los objetos que se creen de esa clase.

Vídeo 125: Funciones de acceso y encapsulación

La **encapsulación** u **ocultación de información** es el proceso de mantener ocultos los detalles sobre cómo se implementa un objeto para los usuarios del objeto. Los miembros se organizan en una interfaz (pública) y una implementación (privada).

El modo de aplicar la encapsulación a las clases es a través de los especificadores de acceso, vistos en el vídeo anterior.

Beneficios de las clases encapsuladas:

- **Menos complejas:** Solo necesitamos conocer las funciones miembro públicas disponibles. Qué parámetros toman y qué valor devuelven.
- **Protegen datos:** Solo se puede acceder a miembros de datos a través de funciones que pueden comprobar la validez de los cambios.
- **Cambios más sencillos:** Permiten llevar a cabo cambios en los miembros privados sin romper todos los programas que están usando la clase.
- **Mejor depuración:** Si solo se puede acceder a través de funciones públicas, facilita encontrar posibles errores.

Las **funciones de acceso** se especializan en acceder a las variables privadas tanto para obtener como para modificar su valor. Son los llamados **getters** y **setters.**

Los **getters** deben devolver por valor o por referencia const, nunca por referencia no const.

Vídeo 126: Constructors

Un **constructor** es un tipo especial de función que se llama automáticamente cuando se instancia un objeto. Se utilizan para inicializar las variables miembro privadas y otras configuraciones necesarias, aunque “inicializar” no es del todo correcto ya que técnicamente son asignaciones. Esto se ve más en detalle en el vídeo siguiente.

Una vez ejecutado el constructor(automáticamente), el objeto debe de haber inicializado todo lo que necesita para su correcto funcionamiento.
Reglas obligatorias de los constructors:

1. El constructor debe tener exactamente el mismo nombre que la clase.
2. Nunca pueden tener tipo de retorno (ni siquiera void).
3. Pueden tomar parámetros o no, si no los toma es un constructor predeterminado.

Constructor predeterminado significa que los valores están escritos directamente en la Clase, no pasados por el usuario.

Podemos crear todos los constructors que queramos, usando sobrecarga de funciones.

Las mejores prácticas aconsejan inicializar los nuevos objetos con inicialización uniforme `{ }` entre llaves. Hay casos que será obligado usar la inicialización directa con paréntesis `( )` pero se verán más adelante en el curso.

Los constructors siempre son necesarios, pero pueden ser implícitos o explícitos. Si la clase no tiene definido ningún constructor, el compilador crea un “**constructor implícito**”, sin cuerpo y sin parámetros.

Vídeo 127: Lista de inicialización de miembros en Constructors

En los constructores se les asigna valores a las variables privadas, pero estas deben haber sido inicializadas o declaradas previamente.

C++ nos proporciona un modo de inicializar las variables miembro desde la firma del constructor, no asignarlas en su cuerpo. Es lo que se denomina **lista de inicialización de miembros** o **lista de inicializadores miembro.**

```
class Nombre_clase{
    private:
        const int var1{};
        int var2{};

    public:
//parámetros(opcionales)    //lista de inicializadores miembro
        Nombre_clase(int valor1, int valor2) : var1{valor1},
var2{valor2} {
                //Cuerpo del constructor
        }
};
```

La lista de inicializadores miembro va a continuación de los parámetros (tenga o no), empiezan con dos puntos ":" y luego se inicializan las variables, separadas por comas y sin punto y coma al final.

En el código de ejemplo hay una variable constante que se inicializa en la lista del constructor sin ningún problema, cosa que no podríamos hacer si la asignáramos en el cuerpo del constructor.

Los miembros de una lista de inicialización pueden ser de cualquier tipo válido, incluidos arrays o std::vector, por ejemplo.

Importante recordar lo siguiente:

- La tarea de un constructor es crear un objeto en un estado válido y utilizable. Entonces, si su llamada a función es necesaria para que la instancia del objeto sea válida, llámela en el constructor.

Vídeo 128: Inicialización de miembros no-static

Si existe un constructor explícito, el implícito no se crea (el que no pide parámetros ni tiene cuerpo). La solución es crearlo tú explícitamente y podrías usarlo sin problemas.

Otra alternativa es utilizar la palabra clave '**default**' para que el compilador construya el constructor implícito aunque haya otros constructors explícitos. Por ejemplo si la clase se llama Rectangulo, en la parte **public:** lo definimos.

```
Rectangulo() = default;
```

Los constructors solo usan los valores predeterminados de las variables miembro si no los incluyen en su lista de inicialización de miembros. Si los incluyen tienen preferencia sobre los predeterminados.

Vídeo 129: Superposiciones y constructors delegados

Las clases pueden tener varios constructors, lo que significa que pueden duplicar funcionalidades (código repetido en cada constructor). El código redundante siempre es negativo, y debemos evitarlo si es posible.

Llamar a un constructor desde otro constructor (en el cuerpo del constructor) puede provocar comportamientos indefinidos, por lo que es mejor evitarlo o directamente descartarlo por completo.

Siempre que formen parte de la misma clase, podemos instanciar un constructor desde la lista de inicialización de miembros de otro constructor. Este proceso se denomina '**delegación de constructors**' o '**encadenamiento de constructors**'.

Para instanciar un constructor dentro de otro constructor, debemos inicializarlo en su lista de inicialización de miembros simplemente llamándolo, pero usando llaves en vez de los paréntesis, es como si lo instanciásemos, no como si llamáramos a una función.

```
     Cualquiera(){ // código para hacer lo que sea }

Cualquiera(int valor) : Cualquiera{} { // lo que sea }
```

Consideraciones a tener en cuenta:

1. Si un constructor delega en otro constructor en su lista de inicialización, el constructor que delega no puede inicializar variables. O delega o inicializa, no ambas cosas.
2. Si un constructor delega en otro constructor que a su vez delega en el primer constructor, se crea un bucle infinito.

Entonces, si con un constructor necesitamos tanto inicializar como delegar, ¿qué hacemos?. La mejor solución es inicializar con el constructor, y, si hay código que estaría duplicado en varios constructor, crear una función solo para ese código y llamar a esa función en los constructors que lo necesiten.

Los constructors no pueden llamar a otros constructors en su cuerpo, pero las funciones 'normales' tampoco. Los constructors no pueden llamarse ya que no están creados para eso, sino para que se ejecuten automáticamente cuando instanciamos un nuevo objeto.

'**this**', además de ser un puntero, se utiliza para "nombrar genéricamente" a un objeto desde la clase, ya que no se puede conocer cada nombre. '

En este ejemplo se crea un nuevo objeto Cualquiera y se usa la asignación para sobreescribir nuestro objeto implícito.

```
*this = Cualquiera();
```

Vídeo 130: Destructors

Al igual que los 'constructors', los '**destructors'** son otro tipo especial de función miembro de clase que se ejecuta automáticamente cuando se destruye un objeto.

Cuando un objeto normal sale de alcance o un objeto asignado dinámicamente se elimina con la palabra '**delete**' el compilador llamará automáticamente al destructor de esa clase, si es que existe, para realizar la limpieza necesaria antes de que el objeto se elimine de la memoria.

El '**destructor**' es el lugar adecuado para destruir los recursos que un objeto ha necesitado para ejecutarse. Para las clases simples que solo inicializan los valores de las variables miembro no será necesario, pero sí para los objetos que contienen algún otro tipo de recurso, como puede ser memoria asignada dinámicamente, un archivo que ha cargado el constructor, etc.

Reglas de los '**destructors**':

1. El nombre del destructor debe ser el mismo que el de la clase precedido por una tilde ~
2. No pueden tomar argumentos.
3. No pueden tener tipo de retorno.
4. Cada clase solo puede tener un destructor.

Los '**destructors**' pueden llamar a las funciones miembro de la clase de modo seguro.

RAII, que significa "Resource Acquisition Is Initialization", es un patrón de diseño de software comúnmente utilizado en C++. Este patrón vincula la vida útil de un objeto con el control de un recurso que debe ser limpiado al finalizar.

El patrón RAII se utiliza para manejar recursos como archivos, memoria, conexiones de red, etc., que tienen un tiempo de vida limitado. La idea es que cuando un objeto es creado, adquiere el recurso y cuando el objeto es destruido, libera el recurso.

Aquí hay un ejemplo simple de cómo se puede usar RAII para manejar la memoria:

```
class MemoryBlock{
     public:
          // Adquiere el recurso.
          MemoryBlock(size_t length): _length(length),
                      _data(new int[length])
          {
          }

          // Libera el recurso.
          ~MemoryBlock()
          {
               delete[] _data;
          }

          // ... Otros métodos ...

     private:
          size_t _length;
          int* _data;
};
```

En este ejemplo, el constructor de `MemoryBlock` adquiere un bloque de memoria, y el destructor libera ese bloque de memoria. Esto asegura que la memoria se libere automáticamente cuando el objeto `MemoryBlock` salga de su ámbito, evitando así las fugas de memoria.

Vídeo 131: El puntero oculto “this”

Sin entender al compilador no podemos escribir buen código. Cualquier función miembro que creemos tiene asociada automáticamente un puntero ‘**this**’ que siempre apunta al objeto al que llama la función.

En C++, ‘**this’** es un puntero que guarda la dirección del objeto en el que se está ejecutando el método actual. Cuando indireccionas **this** con ***this**, obtienes una referencia al objeto actual, o lo que es lo mismo, el objeto en sí para usarlo.

Un ejemplo es esta función de la clase “Simple”, lo que nosotros escribimos y lo que realmente “ve” el compilador:

```
void setID(int id){
  m_id = id;
}
```

```
void setID(Simple* const this, int id){
    this->m_id = id;
}
```

Existen tantos punteros ‘**this**’ como funciones miembro y siempre apuntan a la memoria del objeto sobre el que operan.

El uso explícito de ‘**this->**’ puede evitar ambigüedades entre nombres de variables miembro y variables locales o parámetros de función. Se aconseja usar el prefijo `m_` en las variables miembro para diferenciar estas de las variables locales o los parámetros de funciones y así evitar usar el ‘**this->**’ innecesariamente.

Puede ser muy útil que una función miembro de clase tenga como valor de retorno el objeto sobre el que está operando. Un ejemplo de cómo se haría usando la clase “Simple”:

```
class Simple {
     private:
          int m_valor;
     public:
          Simple(int valor) : m_valor(valor) {}

          Simple& sumar(int valor) {
               m_valor += valor;
               return *this;
          }
};
```

Así funciona el encadenamiento de métodos en C++ con Simple& y *this:

Cuando llamas a un método que devuelve **Simple&**, como sumar, ese método devuelve una referencia al objeto en el que se llamó el método. Esto se logra con **return *this;**, donde **this** es un puntero al objeto actual y ***this** es una referencia a ese objeto.

Debido a que el método devuelve una referencia al objeto, puedes llamar inmediatamente a otro método en esa referencia. Esto es lo que permite el encadenamiento de métodos.

Por ejemplo, si tienes el siguiente código:

```
Simple s(5);
s.sumar(3).sumar(4);
```

Aquí es lo que sucede:

Primero, se llama a `s.sumar(3)`. Dentro de este método, se suma 3 al valor de `m_valor` en el objeto **s**, y luego se devuelve una referencia al objeto **s** con **return *this**;.

Luego, porque `s.sumar(3)` devolvió una referencia a **s**, puedes llamar inmediatamente a `sumar(4)` en esa referencia. Así que se suma 4 al valor de m_valor en el objeto s.

Finalmente, `s.sumar(4)` también devuelve una referencia a **s**, pero como no haces nada con esta referencia, se descarta.

Por lo tanto, después de ejecutar `s.sumar(3).sumar(4);`, el valor de `m_valor` en el objeto **s** se ha incrementado en 7 (es decir, 3 + 4).

El puntero oculto '**this**' se agrega a todas las funciones miembro "no-static" y es constante: puede modificar el valor del objeto al que apunta, pero no podemos hacer que apunte a otro objeto.

PD: Este vídeo es bastante largo y lleno de ejemplos, se recomienda verlo a menudo.

Vídeo 132: Diseño de clases y archivos header

C++ nos permite separar la interfaz de la Clase de su implementación, definiendo funciones miembro de la Clase fuera de la propia definición de la Clase. Para hacer esto, solo debemos definir las funciones fuera de la clase pero prefijando el nombre de la clase antes del nombre de la función, usando el operador de resolución de alcance `::`

Importante destacar que los prototipos de estas funciones sí deben mantenerse dentro de la definición de la Clase.

Pueden haber funciones que se definan dentro de la clase y otras que estén definidas fuera sin problema. Por lo general se suelen dejar los getters y setters definidas dentro ya que son funciones simples y las demás definirlas fuera. Los constructor también se pueden definir fuera sin problemas.

Podemos dividir las clases en archivos header ".h" y archivos de código ".cpp". Ya sabemos que siempre hay que utilizar los header guards en los archivos header para cumplir con la regla "ODR" (One Definition Rule). Pero, ¿qué pasa con las funciones que se definan dentro de la clase? Las funciones miembro definidas dentro de la clase no vulneran la "ODR" al considerarse como **funciones inline**: Estas funciones están exentas de esta regla. En el vídeo 44 ya se habló de las **variables inline**, y es buen momento para repasar lo que significa **inline**.

Mejores prácticas en el diseño de Clases

1. **Un único archivo .cpp:** Clases que se usan en un único archivo y no reutilizables.
2. **Archivos .h y .cpp:** Clases que se van a reutilizar.

En el archivo de cabecera ".h", generalmente se incluyen:

- Los prototipos de las funciones miembro de la clase, incluyendo los constructores y destructores.
- Las definiciones de funciones miembro simples, como los getters y setters.
- Las declaraciones de las variables miembro.
- Los parámetros predeterminados deben incluirse en el header, porque deben de conocerse ya en tiempo de compilación.

Vídeo 133: Objetos de clases y funciones miembro constantes

Los objetos instanciados de una clase también pueden declararse como '**const**' y su inicialización se realiza a través de los constructors de la clase. Una vez inicializado un objeto como '**const**', se rechaza cualquier intento de modificar las variables miembro del objeto. Esto impide tanto cambiar las variables miembro directamente, como intentar hacerlo a través de funciones de acceso.

Los objetos declarados '**const**' solo pueden llamar a funciones también declaradas como '**const**'.

Las **funciones miembro constantes** garantizan que no modificarán el objeto, ni que llamarán a otras funciones que puedan modificarlo. Para convertir una función en '**const**', hay que añadir después de los parámetros pero antes de las llaves del cuerpo la palabra '**const**'.

```
int getValor() const { // lo que sea }
```

Para funciones miembro definidas fuera de la clase, se debe añadir '**const**' tanto en el prototipo dentro de la clase como en la definición fuera de la clase.

Cualquier función miembro constante que intente modificar una variable miembro o llamar a una función miembro no constante, provocará un error del compilador. No se permiten los constructors '**const**'.

Cuando una función miembro es '**const**', su puntero oculto '**this'** también será '**const**'.

Aunque no se suele utilizar mucho, es posible sobrecargar una función para que tenga una versión '**const'** y una **no const**. Esto funciona porque el calificador '**const'** se considera parte de la firma de la función.

Las funciones '**const**' pueden trabajar tanto con objetos '**const**' como con objetos **no const,** pero siempre devuelven una referencia '**const**'.

Las mejores prácticas aconsejan declarar las funciones miembro de nuestras clases que no modifiquen los objetos como '**const**'.

Vídeo 134: Variables miembro static

Podemos convertir las variables miembro de una clase en '**static'**, como por ejemplo**:**

```
static int s_valor{};
```

Todos los objetos instanciados de una clase comparten el valor de sus variables miembro '**static**'.

La creación de las variables miembro '**static**' no depende de la instanciación de un objeto. Se crean al iniciarse el programa. Por eso, aunque podemos acceder a ellas a través de un objeto de esa clase, las mejores prácticas aconsejan acceder a las variables '**static**' usando la clase en sí, no una instancia de esa clase.

Por ejemplo: `CualquierClase::s_valor = 2;`

Las variables miembro que sean '**static**' deben estar definidas en el ámbito global, por la misma razón que si fueran **inline**, aunque con excepciones.

1. Si la variable es de tipo **integral const** o **enum const,** se puede inicializar dentro de la definición de la clase.
2. Las variables miembro de cualquier tipo declaradas como **constexpr** también se pueden definir dentro de la propia clase.
3. A partir de C++17 podemos definir miembros static también **no-const** dentro de la definición de la clase, si los declaramos como **inline**.

Si las variables miembro **static** se declaran privadas, podemos seguir definiendo las variables e inicializarlas en el ámbito global, porque no siguen las reglas de acceso. Pero no podemos acceder a ellas directamente, ni desde un objeto ni desde la clase ya que es privada.

Vídeo 135: Funciones miembro static

Al igual que las variables miembro static, las **funciones miembro static** no están asociadas a ningún objeto en particular, por lo que podemos llamarlas usando directamente la clase, sin instanciar ningún objeto de la misma.

Estas funciones tienen dos características que es importante conocer:

1. No tienen puntero **this**, porque no están vinculadas a un objeto.
2. No pueden acceder a miembros **no-static.**

Las funciones miembro **static** también se pueden definir fuera de la clase, en el ámbito global.

Las clases en las que todos sus miembros y funciones son **static** se llaman **Clases estáticas puras** o **monoestados**. Estas pueden ser útiles en determinados casos, pero cuentan con potenciales desventajas:

1. Al no poder instanciar objetos de la Clase, si necesitamos más de una copia, debemos clonar la clase con otro nombre.

2. Una clase estática pura significa declarar variables y funciones a las que se puede acceder globalmente a través de su namespace, pudiendo modificarlas desde cualquier lugar del programa.

C++ no admite constructors **static**.

Vídeo 136: Funciones y clases amigas

Una **función amiga** puede acceder a los miembros privados de una clase como si fuera miembro de esa clase. Puede ser tanto una función miembro de otra clase como una función que no forme parte de ninguna clase.

Para declarar una función amiga usamos la palabra clave "**friend**" delante del prototipo de la función, dentro de la clase, en su parte pública o privada. El prototipo de esa función debe tomar como parámetro un objeto de la clase de la que se hace "**friend**".

Ejemplo en una clase llamada "Acumulador"

```
friend void restablecer(Acumulador& acumulador);
```

Si la función pertenece a otra clase, se haría igual pero usando el operador de resolución de alcance `::` y el nombre de la clase a la que pertenece.

```
friend void Clase::restablecer(Acumulador& acumulador);
```

Una función puede ser amiga de más de una clase al mismo tiempo.

Podemos también hacer una clase completa amiga de otra clase, esto permitirá a todos los miembros de la clase amiga acceder a los miembros privados de la otra clase.

Por ejemplo, si queremos hacer amiga la clase "Mostrar" de la clase "Acumulador", dentro de "Acumulador" añadimos la declaración anticipada de la clase "Mostrar", anteponiendo "**friend**".

```
friend class Mostrar;
```

De igual manera se debe hacer desde un objeto de la clase a la que se es amiga, como en las funciones.

Recordar varias cosas importante sobre las clases amigas:

1. Una clase amiga no tiene acceso al puntero "**this"** de los objetos de la otra clase.
2. La amistad no es bidireccional. Para que las dos clases sean amigas entre sí deben declararse las dos como "**friend**".
3. Clases y funciones amigas vulneran el principio de encapsulación de la POO y deben usarse con cuidado.

Vídeo 137: Objetos anónimos

Los objetos anónimos cumplen un papel similar con las variables al que las "**lambdas**" cumplen con las funciones. Cuando necesitamos una variable solo de modo temporal, podemos usar un "**objeto anónimo**".

Un "**objeto anónimo**" es un valor sin nombre, que deja de existir al salir de la expresión donde se ha creado. Estos objetos se encuentran en el "**ámbito de expresión**" ya que se crea, se evalúa y se destruye dentro de una única expresión.

Un ejemplo de objeto anónimo es el siguiente:

```
int sumar(int x, int i){
    return x + y;   // esto es un objeto anónimo
}
```

También es posible construir objetos anónimos de tipos definidos por el usuario. Esto se hace creando objetos, pero omitiendo el nombre de la variable.

```
Centimos centimos { 5 }; // variable normal
Centimos{ 5 };// nace y muere aquí, no tiene nombre
```

Vídeo 138: Tipos anidados dentro de clases

Es posible definir tipos dentro de clases, como **enum**s. Si anidamos un enum dentro de una clase, éste debe ser sin ámbito ya que al pertenecer a una clase desaparecen los problemas que tenía de ámbito. Si lo declarásemos usando "**enum class**" necesitaríamos referenciar dos namespaces para acceder al enum.

Los **enum** suelen ser los tipos más comunes que se anidan dentro de clases, pero nos permite definir otros tipos dentro de una clase, como "**typedef**", o incluso anidar **clases dentro de otras clases**, pero no suele ser muy común, a excepción de las clases iteradoras.