CURSO C++ 99_127

Vídeo 99: Structs y Miembros

Struct es una abreviatura de "estructura". Es un tipo de dato definido por el usuario que agrupa múltiples variables en un único tipo.

Como todos los tipos definidos por el usuario, es necesario definir un **struct** antes de poder utilizarlo. La sintaxis básica es la siguiente:

```
struct Nombre {
    // Variables y funciones miembro
};
```

Las variables que forman parte de una struct se llaman **variables miembro**, y las funciones se conocen como **funciones miembro** o, en ocasiones, como **métodos.**

Para acceder a los miembros de un objeto struct una vez instanciado, se utiliza el operador de selección de miembros "**.**", también conocido como el operador punto. Por ejemplo, para acceder a una variable miembro se usaría `` `Nombre.miembro` ``, y para llamar a una función miembro se utilizaría `` `Nombre.funcion()` ``.

Las buenas prácticas aconsejan empezar los nombres por mayúscula usando PascalCase. Esto se aplica no solo a structs, sino también a clases, que se explican en detalle en el vídeo 123.

Vídeo 100: Inicialización de structs

Las variables miembro no se inicializan de modo predeterminado.

Los **agregados** son los tipos que pueden contener varios datos miembros. Algunos agregados requieren que todos los miembros sean del mismo tipo, como los arrays, o diferentes tipos como los structs o las clases. Se profundizará más en los agregados a lo largo del curso.

La inicialización de una struct se conoce como **inicialización agregada**. Existen tres maneras de inicialización agregada:

```
struct Alumno{

    int id{};
    int edad{};
    int tutorID{};
```

```
};

Alumno juan = {1, 17, 3}; // Por copia-lista con igual y
llaves
Alumno marta (2, 17, 2); // Directa usando lista entre
paréntesis (C++ 20)
Alumno pepe {3, 18, 1}; // Lista o uniforme entre llaves
(preferida)
```

Cada miembro se inicializa en el mismo orden en el que están declarados, en este caso el primero sería **id**, el segundo **edad**, etc. La lista no tiene que estar completa.

Las variables a las que asignemos un tipo **struct** pueden ser constantes, pero hay que inicializarlas al crearlas.

C++ 20 ha introducido **inicializadores designados**, que permiten definir explícitamente con que valores inicializar cada miembro, pero deben inicializarse en el mismo orden o dará error.

```
Alumno maria { .id{ 1 }, .tutorID{ 4 } }; // En este caso edad
se inicializa en 0

Alumno maria { .edad{ 1 }, .id{ 4 } }; // error por el orden
```

Las mejores prácticas aconsejan no utilizarlos porque pueden llegar a complicar más el código, y de ser necesario añadir nuevos miembros a un agregado, siempre añadirlo a continuación del último miembro y no en medio de los que ya se encontraban en el agregado.

Vídeo 101: Inicialización predeterminada de miembros

Al definir una struct, ya podemos proporcionarle valores de inicialización. Esta inicialización se denomina **inicialización no estática de miembros.** `int id{1};` y al valor de inicialización se denomina **inicializador de miembro predeterminado**. Si al crear el objeto no le indicas explícitamente que quieres utilizar otros valores en sus miembros, se utilizarán estos valores predeterminados.

Debemos asegurarnos que al definir los miembros en la struct estén inicializados aunque sea con llaves vacías int id{}; así no tienes que preocuparte de si crear los

objetos del modo `Alumno pepe;` o `Alumno pepe{};` La segunda siempre es la opción más segura ya que inicializa miembros que no están inicializados en la definición de la struct, y sigue dejando los predeterminados si los hubiera.

Vídeo 102: Pasar y devolver structs a funciones

Podemos pasar una struct completa a una función. Se suelen pasar por referencia const. `void imprimirAlumno(const Alumno& alumno);` para evitar tener que hacer copias.

Se pueden utilizar tipos definidos por el usuario como miembros de una struct, como un struct que tiene de miembro otro tipo struct. Para ello se puede tanto definir cada uno por su lado, como anidar una struct dentro de otra y luego utilizarla como miembro.

Lo visto para las structs es aplicable a las clases.

Vídeo 103: Selección de miembros con punteros y referencias

El uso del operador de selección de miembro punto "." no funciona con los punteros, porque el puntero solo contiene una dirección, por lo que tendremos que indireccionarlo antes de poder acceder a los miembros del objeto.

```
Alumno* ptr{ &juan };
std::cout << (*ptr).id; //Funciona pero es confuso ya que te obliga a poner el operador de indirección entre paréntesis para que tenga prioridad sobre el de selección.
```

El **operador flecha** "->" sirve para seleccionar miembros desde un puntero.

```
std::cout << ptr->id; //Opción preferida
```

Los operadores de selección de miembro punto y flecha pueden mezclarse.

```
std::cout << ptr->pata.garras // ptr contiene la dirección a un struct Animal que tiene como miembro otro struct llamado Pata. Con la flecha accedes desde la dirección y con el punto accedes normalmente al miembro del otro struct.
```

Vídeo 104: Aritmética de punteros e indexación de arrays

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.