CURSO C++ 99_127

Vídeo 99: Structs y Miembros....2
Vídeo 100: Inicialización de structs....2
Vídeo 101: Inicialización predeterminada de miembros....3
Vídeo 102: Pasar y devolver structs a funciones....4
Vídeo 103: Selección de miembros con punteros y referencias....4
Vídeo 104: Aritmética de punteros e indexación de arrays....5
Vídeo 105: Asignación dinámica de memoria con new....6
Vídeo 106: Uso de delete. Punteros colgantes y fugas de memoria....6
Vídeo 107: Asignación dinámica de arrays....7
Vídeo 108: Bucles for-each....8
Vídeo 109: Punteros void....9
Vídeo 110: std::array....10
Vídeo 111: std::array con plantillas de funciones y structs....11
Vídeo 112: std::vector....11
Vídeo 123: ¿Qué es la programación orientada a objetos?....11
Vídeo 124: Especificadores de acceso a Clases. public: y private:....12
Vídeo 125: Funciones de acceso y encapsulación....13
Vídeo 126: Constructors....14
Vídeo 127: Lista de inicialización de miembros en Constructors....15

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

La aritmética de punteros en C++ nos permite llevar a cabo operaciones de suma y resta de enteros sobre punteros. Por ejemplo si `ptr` es un puntero de tipo "**int**", con `ptr + 1` apuntamos a la dirección de memoria del siguiente objeto "**int**", y con `ptr - 1` apuntamos al anterior en memoria a `ptr`.

Por lo tanto, hay que recordar que `ptr + 1` no devuelve la siguiente dirección en memoria, que siempre sería un byte, sino que devuelve la dirección del siguiente objeto del "**tipo**" al que apunta `ptr`.

El tamaño de las operaciones aritméticas sobre punteros depende del tipo de objeto apuntado. El compilador siempre multiplica el operando entero ("3" p. ejemplo) por el tamaño en bytes del objeto apuntado. Esto se denomina en C++ **escalar.**

Recordar, como ya vimos en el vídeo de los arrays, que los elementos que forman un array se localizan secuencialmente en memoria. Por ejemplo un array de "**int**", si el primer elemento se almacena en la dirección 00, el segundo estará en 04, el tercero en 08, etc, ya que al ser int cada elemento ocupa 4 bytes en memoria.

Así pues, ya que un array fijo puede decaer en un puntero que apunta a la dirección de memoria del primer elemento del array (de índice 0), array +1 debería devolvernos la dirección en memoria del segundo elemento del array (de índice 1) y así sucesivamente.

La librería **<algorithm>** nos ofrece `std::count_if` que cuenta los elementos que cumplen una condición.

Vídeo 105: Asignación dinámica de memoria con new

Hasta ahora hemos visto dos tipos de asignación de memoria:

Estos tipos de asignación de memoria nunca pueden producir pérdidas de memoria o espacios colgantes.

La **asignación dinámica de memoria** nos permite realizar solicitudes de memoria al sistema operativo cuando sea necesario. Esta memoria se asigna al "**Montón**", que es un espacio de memoria mucho más grande y administrado por el S.O.

Para asignar una única variable de forma dinámica usamos la palabra clave **"new"** seguida del tipo que queremos para nuestro objeto.

```
new int;
int* ptr{ new int };
```

Cuando asignamos dinámicamente memoria con **"new"** el compilador crea el espacio en memoria y lo que nos devuelve es su dirección (un puntero).

Vídeo 106: Uso de delete. Punteros colgantes y fugas de memoria

En el vídeo anterior se vio cómo asignar memoria dinámicamente usando "**new**". Para devolver la gestión de ese espacio de memoria al S.O y liberarla, utilizaremos la palabra clave "**delete**".

```
int* ptr1{ new int { 6 } };// Inicializar
```

```
delete ptr1;// Liberar la memoria una vez no la utilicemos
```

Pero cuando usamos “**delete”**, ¿qué eliminamos? ¿el contenido de la memoria? ¿el puntero?. La respuesta correcta es que el uso de **delete** no elimina ni el contenido en memoria ni el puntero. Lo que hace es devolver el control de un espacio en memoria al S.O. (como se había dicho al principio de este apartado, pero era necesario explicarlo).

Esto quiere decir que ese espacio estará disponible para el uso del S.O. y, si lo necesita, entonces sobreescribirá los datos que había en ese espacio.

En cuanto al puntero, es una variable, y su vida útil no la decide el programador. Seguirá existiendo dependiendo de donde se encuentre, por ejemplo si está en el **main** sobrevivirá hasta que el programa termine, de ahí los punteros colgantes. La forma más segura de neutralizar un puntero colgante, es después de usar **delete** asignarle **nullptr**. `ptr1 = nullptr;`

Mejores prácticas para evitar punteros colgantes:

1. Evitar múltiples punteros a una misma dirección dinámica
2. Asignarles a todos nullptr después de usar delete.

Las **fugas de memoria** se producen cuando nuestro programa pierde la dirección de algún espacio en memoria asignado dinámicamente. Ahora ni el programa puede acceder a ese espacio de memoria ni el S.O. puede usarla porque sigue asignada al programa.

Las fugas de memoria se pueden producir de varias maneras:

- Porque un puntero salga de alcance.
- Al cambiar la dirección a la que apunta el puntero
- Porque el puntero es usado para almacenar una nueva asignación dinámica de memoria.

Vídeo 107: Asignación dinámica de arrays

En la asignación dinámica de arrays podemos considerar también que los arrays son fijos, pero el tamaño del array se asigna en tiempo de ejecución. Una vez asignado no podemos cambiarlo, por eso son fijos.

Para asignar y eliminar memoria dinámicamente en arrays usaremos la forma array de **new[ ]** y **delete[ ].**

```
int* array{ new int[tamaño]{ } }; //el tamaño no necesita ser constante, como sí ocurre en los arrays en tiempo de compilación.
```

El tipo del tamaño debe ser convertible a `std::size_t` (como un **int**).

Los programas que necesitan asignar mucha memoria lo hacen de forma dinámica por lo visto anteriormente, ya que esa memoria es del **montón** y no de la **pila** que es mucho más pequeña.

Para eliminar el array dinámico: `delete[ ] array;`

Todo lo visto en los vídeos sobre arrays anteriores es exactamente igual para los arrays asignados dinámicamente.

A partir de C++ 11 se puede usar una lista de inicialización también con los arrays asignados dinámicamente.

```
int* array{ new int[5]{ 9, 1, 5, 2 } };
```

Podemos usar **"auto"** para no tener que usar dos veces el tipo que aunque ahora parece una tontería cuando se complique la cosa es muy útil..

```
auto* array{ new int[5]{ 9, 1, 5, 2 } };
```

Vídeo 108: Bucles for-each

Usaremos el bucle **for-each** para los casos en que queremos iterar **a través de todos los elementos** de un array o en otras estructuras de tipo lista similares en orden secuencial y hacia adelante. El prototipo es el siguiente:

```
for(elemento_declaración : array ){  //expresiones;  }
```

Para obtener los mejores resultados tanto el elemento_declaración como el array que se le pase deben ser del mismo tipo. Este es un buen caso para usar auto como tipo en elemento_declaración, ya que deducirá el tipo a partir del array.

Podemos usar referencias para evitar copias.

```
for(auto& elemento : array){ }
```
Y si además de eso queremos que el bucle sea solo de lectura podemos usar const.

```
for(const auto& elemento : array){ }
```

Como mejor práctica, en las declaraciones de elementos de bucles for-each que no sean de tipos fundamentales, debemos usar referencias o referencias constantes.

El bucle for-each solo puede usarse con arrays o listas (vector, map, list, etc.) que conozcan su tamaño, así que no puede aplicarse a su versión decaída con puntero o arrays asignados dinámicamente.

Los bucles for-each no proporcionan una forma directa de obtener el índice de un elemento del array, pero desde C++ 20 se puede usar un bucle for-each con una declaración de inicio como en los bucles for.

```
for(int i{ 0 }; auto nota : notas){ }
```

Vídeo 109: Punteros void

Los punteros void se conocen también como **punteros genéricos** y son punteros especiales que pueden apuntar a cualquier tipo de objeto. `void* ptr;`

La principal limitación de estos punteros es que no pueden indireccionarse al no conocer el tipo de su dirección, así que tendremos que convertirlo explícitamente al tipo de objeto al que apunta usando static_cast<>().

Aspectos importantes de los punteros vacíos:

- Pueden inicializarse con un valor nulo nullptr
- No pueden apuntar a memoria asignada dinámicamente.
- No se puede hacer aritmética de punteros al no conocer el tamaño del objeto apuntado.
- No existen las referencias void&

Lo mejor es evitar el uso de punteros vacíos a no ser que sea estrictamente necesario.

Vídeo 110: std::array

`**std::array**` sigue siendo un array fijo, pero a diferencia de los arrays convencionales, no se degrada a un puntero cuando se pasa como argumento a una función, lo que implica que conserva información sobre su tamaño.

```
std::array<tipo, tamaño> nombre{};
```

Inicialización:

```
std::array<int, 3> nombre_array{1, 2, 3};
```

Omitiendo tipo y tamaño al inicializar (disponible desde C++17):

```
std::array nombre_array2{1.5, 4.2};
```

También es posible definir y luego asignar:

```
std::array<int, 6> mi_array;
mi_array = {1, 2, 3, 4, 5, 6};
```

`**std::array**` proporciona la función `**at()**` para acceder a sus elementos, la cual verifica los límites, a diferencia del acceso mediante corchetes (`mi_array[4]`). Aunque utilizar `at()` es más lento, ofrece una mayor seguridad en la gestión de límites.

`**std::array**` se limpia automáticamente al salir de su alcance, eliminando la necesidad de realizar limpieza manual.

Podemos emplear la función `.size()` para obtener el tamaño (número de elementos) de un `**std::array**`.

Como buena práctica, se recomienda pasar los `**std::array**` a funciones mediante referencia o referencia constante.

Para ordenar un `**std::array**` de forma ascendente, podemos utilizar `**std::sort()**` junto con `.begin()` y `.end()`, mientras que para ordenar de forma descendente, usamos `.rbegin()` y `.rend()`. Es necesario incluir `**<algorithm>**` para utilizar `**std::sort**`.

Vídeo 111: std::array con plantillas de funciones y structs

Un ejemplo del uso de plantillas con arrays es el siguiente:

```
template <typename T, std::size_t tamanio>
```

Donde typename podrá ser cualquier tipo, incluso un **struct**, pero `std::size_t` sí debe ser ese tipo. En C++ "**sizeof()**" y otras funciones que devuelven tamaños usan el tipo `std::size_t`, que se define como un tipo "integral unsigned".

Vídeo 112: std::vector

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.