CURSO C++ 99_122

Índice 99_122

Vídeo 99: Structs y Miembros

"**Struct**" es una abreviatura de "estructura", un tipo de dato definido por el usuario que combina múltiples variables en un solo tipo.

Como con todos los tipos definidos por el usuario, es esencial definir una **struct** antes de poder utilizarla. 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 comenzar los nombres con mayúscula utilizando PascalCase. Esto se aplica no solo a structs, sino también a clases, que se detallan en el vídeo 123.

Vídeo 100: Inicialización de structs

Las variables miembro no se inicializan de manera predeterminada.

Los **agregados** son tipos que pueden contener varios datos miembros, algunos 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** y hay tres maneras de llevarla a cabo:

```
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 qué valores inicializar cada miembro, pero deben inicializarse en el mismo orden o dará error.

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

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

Las mejores prácticas aconsejan no utilizarlos porque pueden complicar el código, y si es necesario añadir nuevos miembros a un agregado, siempre añadirlo después 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

‘std::vector’ es un array dinámico que se encarga de autoadministrar su memoria ya que permite crear arrays cuyo tamaño se determina en tiempo de ejecución sin tener que asignar y desasignar con el uso de “**new**” y “**delete**”. Se incluye dentro del header **<vector>.**

Así se declara un **std::vector**:

```
std::vector<int> nombre;
std::vector<int> nombre{ 1, 2, 3};
```

Se puede omitir el tipo al igual que en los std::array si lo puede inferir (desde C++17). Nótese que no hay que especificar el tamaño. El tamaño se asigna dinámicamente y se puede modificar en tiempo de ejecución.

Se puede acceder a los elementos del vector al igual que se hacía en los array, con ‘`.at()`’ o `[ ]`.

Destacar que **‘std::vector’** se autolimpia cuando sale de alcance sin necesidad de que el programador lo haga manualmente.

Una de las grandes ventajas de ‘**std::vector**’ es que nos permite modificar su tamaño usando ‘`.resize(tamaño)`’, pero debemos tener en cuenta dos cosas:

1. Cuando cambiamos el tamaño de un std::vector, los valores existentes se mantienen.
2. A los nuevos elementos se les asigna el valor predeterminado.

Debemos tener en cuenta que cambiar el tamaño de un vector es computacionalmente costoso, por lo que es mejor hacerlo lo menos posible.

Si queremos declarar un ‘**std::vector’** con un número específico de elementos pero sin conocer aún los valores, podemos usar la inicialización directa usando `( )`.

```
std::vector<int> vector(5) // entre paréntesis, 5 elementos de
valor 0
std::vector<int> vector{5} // entre corchetes, 1 elemento de
valor 5
```

Por lo tanto, una regla general sería: Si queremos crear cualquier tipo de lista sin especificar valores de inicialización, debemos usar la inicialización directa (entre paréntesis).

Vídeo 113: Iteradores

Los iteradores son objetos diseñados para recorrer un contenedor, proporcionando acceso a cada elemento.

Todos los contenedores de la librería estándar ya cuentan con funciones '`.begin()`' y '`.end()`'. Además el header **<iterator>** contiene las funciones `std::begin(contenedor)` y `std::end(contenedor)` para determinar los límites de un contenedor.

Vídeo 114: Algoritmos de la librería estándar

Los algoritmos no son más que instrucciones que realizan un cálculo o resuelven un problema.

El header **<algorithm>** de la librería estándar de C++ nos ofrece algoritmos preconstruidos. Las funcionalidades provistas en esta librería se dividen en tres grandes categorías:

- **Algoritmos inspectores**: Se utilizan para ver, sin modificar.
- **Algoritmos mutadores:** Se utilizan para modificar datos de un contenedor.
- **Algoritmos facilitadores:** Se utilizan para generar resultados a partir de datos de contenedores.

Vamos a ver algunos algoritmos (funciones) de esta librería:

'**std::find()**' busca la primera coincidencia de un valor en un contenedor. Toma tres parámetros, un **iterador de inicio**, un **iterador de final** y el **elemento a buscar.** Devuelve un puntero al elemento que buscamos (si lo encuentra) o al final del contenedor si no se encuentra ningún elemento coincidente.

```
auto encontrado{ std::find(arr.begin(), arr.end(), buscar) };
```

'**std::count()**' cuenta el número de veces que aparece un elemento en un contenedor. Los parámetros son los mismos que 'std::find()'.

'**std::sort()**' se suele utilizar para ordenar arrays de forma ascendente, pero puede hacer más que eso. Hay una versión de 'std::sort()' que toma tres parámetros y nos permite determinar cómo ordenar los elementos de una estructura.

Vídeo 115: Punteros de funciones

Los **punteros de funciones** contienen la dirección en memoria de una función. Al igual que las variables, las funciones se almacenan en una dirección en memoria al crearse. Para comprobarlo, basta con intentar imprimir el identificador de una función sin paréntesis, y obtendremos su dirección en memoria.

Las funciones tienen un tipo '**lvalue**' que incluye el tipo de los parámetros y del retorno.

Un ejemplo de un puntero de función sería el siguiente:

```
int (*ejemploPtr)()
```

Los paréntesis son imprescindibles, ya que si no se ponen, el puntero se asocia a '**int**', no a la función. Cuando creamos un puntero, podemos inicializarlo con una función o, en el caso de punteros de funciones que no sean constantes, podemos definirlos y asignarles una función posteriormente.

```
int (*ejemploPtr)(){ &ejemplo };

ejemploPtr = &otroEjemplo;
```

Tanto los parámetros como el tipo de retorno de un puntero de función y la función a la que apunta deben ser coincidentes. Cabe señalar que un puntero de función puede inicializarse o asignársele el valor '**nullptr**'.

Para llamar a una función a través del puntero, se puede hacer de forma explícita, por ejemplo:

```
int (*ejemploPtr)(int){ &ejemplo };
int resultado = (*ejemploPtr)(5);
```

O de manera implícita, que, como se puede ver, es similar a llamar a una función normalmente.

```
int (*ejemploPtr)(int){ &ejemplo };
int resultado = ejemploPtr(5);
```

Los parámetros predeterminados de funciones no funcionarán con punteros de función, ya que estos se definen en tiempo de ejecución, y los parámetros predeterminados deben resolverse en tiempo de compilación.

Podemos utilizar un condicional para comprobar que un puntero de función no sea nulo y así evitar un comportamiento indefinido.

¿Pero para qué sirven los punteros? Hay muchas situaciones en las que el compilador (y otras funciones y clases de la biblioteca estándar) nos devuelven punteros con los que debemos trabajar. Los punteros de función nos permiten pasar una función como argumento a otra función, lo que abre una cantidad enorme de posibilidades.

Vídeo 116: Pasar funciones como argumento a otras funciones

Las funciones utilizadas como argumentos en otra función a veces se llaman '**funciones de devolución de llamada**'.

Para definir y almacenar punteros de función, podemos utilizar `` `std::function` ``, que se encuentra en la librería **`<functional>`**.

```
std::function<int()> fcnPtr{ &ejemplo };
```

Aquí, '**int**' representa el tipo de retorno, y los paréntesis vacíos indican que la función no tiene parámetros. Es importante señalar que los paréntesis son obligatorios.

También podemos utilizar la palabra clave '**auto**' para deducir que se trata de un puntero de función, como se muestra a continuación:

```
auto fcnPtr { &ejemplo };
```

Es crucial que la variable esté inicializada para que la deducción sea posible, ya que esta se realiza a partir de la función a la que apunta. Aunque el uso de '**auto**' simplifica la sintaxis, tiene la desventaja de ocultar el tipo de retorno y los parámetros de la función.

El vídeo proporciona una explicación detallada sobre cómo pasar funciones como argumentos a otras funciones. Sin embargo, en este caso en particular, puede ser más efectivo comprenderlo y aplicarlo cuando surjan dudas y sea necesario ponerlo en práctica. El tema es bastante denso y puede resultar difícil de entender sin un contexto específico.

Vídeo 117: Segmentos de memoria. el heap y el stack

La memoria que usa un programa se divide en diferentes áreas llamadas **segmentos** que se encargan de distintas partes del programa.

1. **Segmento de código:** Almacena el código compilado, se le conoce también como segmento de texto y suele ser de solo lectura.
2. **Segmento de Datos no inicializados:** O segmento bbs. Variables globales o estáticas (duración igual a la del programa) inicializados a cero.
3. **Segmento de Datos inicializados:** O segmento de datos, almacena las variables globales y estáticas inicializadas.
4. **Heap o montón:** Se almacenan todas las variables del programa asignadas dinámicamente.
5. **Call stack o pila de llamada:** También stack o pila. Almacena parámetros, variables locales e información de funciones con cambios constantes.

El heap o **montón** realiza un seguimiento de la memoria utilizada para la asignación dinámica con `new`.

Ventajas y desventajas del **heap** o **montón**:

- Cuenta con un gran espacio en memoria muy superior al “stack”.
- Su funcionamiento es más lento que el del “stack”.
- La desasignación de memoria debe hacerse manualmente.
- Tenemos que acceder a la memoria asignada a través de un puntero.

La **pila** es la encargada de realizar un seguimiento de todas las funciones activas, es decir las que han sido llamadas y todavía no han finalizado su ejecución.

Cuando se inicia un programa lo primero que se coloca en la pila es la función `main()`. Las demás funciones se irán colocando encima de la pila desde que son ejecutadas hasta que su ciclo de vida finaliza.

La permanencia de una función en la pila, está muy relacionada con su ciclo de vida, y en el mismo momento que la función finaliza abandona también la pila de llamadas.

La pila es una estructura de datos de las denominadas LIFO (Last in first out). Este tipo de estructura es necesaria por el modo en el que los programas manejan los

ciclos de vida de las funciones y las relaciones entre ellas. Por eso las únicas tres cosas que se pueden hacer con las funciones que se encuentran en la pila son:

1. Acceder a la función superior de la pila
2. Retirar la función superior de la pila, lo que convierte a la que estaba debajo en la superior.
3. Colocar una nueva función en la parte superior de la pila.

Ventajas y desventajas de la **pila:**

- La asignación de memoria es más rápida.
- La memoria es asignada solo si la función se ejecuta.
- La asignación de memoria se conoce en tiempo de compilación.
- Tiene muy poco espacio. No es idónea para grandes estructuras.

Vídeo 118: El atributo capacidad de std::vector

`std::vector` es el modo más aconsejable para la asignación de arrays dinámicos:

1. Porque se encarga de gestionar la memoria
2. Porque siempre conoce su tamaño
3. Porque se puede redimensionar fácilmente

`std::vector` permite diferenciar entre tamaño y capacidad, que son dos atributos separados. En el contexto de `std::vector`

- **Tamaño:** Cuántos elementos se utilizan en el array `.size()`
- **Capacidad:** Cuántos elementos se le asignaron en memoria `.capacity()`

La capacidad nunca puede ser menor que el tamaño o los últimos elementos del vector estarían fuera de la memoria que se le ha asignado.

El rango tanto del operador de subíndices `[ ]` como de la función `.at()` .es el tamaño, no la capacidad.

En sus usos como array dinámico, el atributo “capacidad” de `std::vector` no es necesario. Pero sí cuando se usa con otros tipos de estructuras, como una estructura LIFO o de pila.

La función `.reserve()` configura la capacidad del `std::vector.` ¿Pero, por qué utilizarla? Las funciones de pila del `std::vector` como `push_back()` o `pop_back()` modifican el tamaño y la capacidad de la pila si es necesario, pero a cada cambio consume gran cantidad de recursos. Es recomendable asignarle de entrada una capacidad mayor si sabemos que es necesario insertar o eliminar elementos para evitar esa sobrecarga.

Vídeo 119: Recursión y funciones recursivas

Una función recursiva es una función que se llama a sí misma. Si se visualiza la pila explicada en el vídeo 117 se comprende perfectamente lo que son.

Para estas funciones es necesario incluir una "**terminación recursiva**" que hará que no se vuelva a llamar a sí misma y así salir del bucle.

Las entradas triviales son denominadas "casos base" y se usan como condiciones de terminación.

Las funciones recursivas suelen ser difíciles de comprender a primera vista, por lo que acompañarlas de buenos comentarios sobre su funcionamiento es recomendable.

Uno de los algoritmos recursivos más famosos es la secuencia de fibonacci (no es óptimo calcularlo con recursión)

Vídeo 120: Argumentos de línea de comandos

Podemos iniciar un programa pasándole argumentos para inicializarlo. A estos argumentos se les llama "**argumentos de línea de comandos"**, que no son más que el método main con parámetros. (hasta ahora se usaba la versión del `main()` sin parámetros).

Los "argumentos de línea de comandos" son argumentos string opcionales que una aplicación recibe de terceros (a través del sistema operativo).

```
int main(int argc, char* argv[]){ //cuerpo del main }
int main(int argc, char** argv) { //cuerpo del main }
```

`argc` es un parámetro de tipo int, que contiene el número de argumentos de línea de comandos pasados al programa. **`argc`** siempre será al menos 1, ya que el primer parámetro es el propio nombre del programa. Cada argumento de línea de comandos que proporcione el usuario hará que el contador aumente en 1.

`argv` es un array de strings tipo C, en el que cada elemento almacena el contenido real de cada "argumento de línea de comandos" pasado.

Los "argumentos de línea de comandos" son siempre pasados como **strings,** si queremos usarlo como número debemos llevar a cabo una conversión.

Vídeo 121: Expresiones lambda (Funciones Anónimas)

Las "**expresiones lambda**" son expresiones que permiten definir funciones anónimas y anidables dentro de otras funciones.

El prototipo de una expresión lambda es el siguiente:

```
[clausulaCaptura](parametros) -> tipoRetorno {declaraciones;}
```

Tanto la cláusula de captura, como los parámetros y el tipo de retorno son opcionales, y las expresiones lambda son anónimas, lo que quiere decir que no le proporcionamos nombre.

Si no se indica tipo de retorno el programa asume que es **auto**, y aunque ya se habló que usar **auto** para tipos de retorno no era aconsejable, en las expresiones lambda sí es recomendable, principalmente porque las expresiones lambda se usan como funciones triviales, no especialmente complejas ya que si son complejas se pierden todas las virtudes de usar lambdas.

Lo siguiente es una expresión lambda válida:

```
[ ]() {}; // sin cláusula ni parámetros ni nombre ni retorno
```

Las expresiones lambda no existen para trabajar aisladas, sino anidadas dentro de otras funciones.

Se puede almacenar una expresión lambda en una variable, y luego usar esa variable en el sitio que necesitamos esa lambda. Es recomendable este método para mejorar la legibilidad del código.

Cuando asignamos una “lambda” a una variable, el compilador le aplica un tipo único, pero no es accesible ni utilizable. Existen varias maneras de almacenar una lambda (minuto 10:40 del vídeo ya que es denso de entender).

Las lambdas en realidad no deben de llamarse funciones, ya que son **functores**. Los **functores** son objetos que tienen un operator() sobrecargado, lo que les permite ser invocables como las funciones. Se verá en profundidad en el curso más adelante.

Vídeo 122: Cláusulas de captura en expresiones lambda

Las lambdas no pueden acceder a todos los identificadores de su bloque externo. Para poder hacerlo:

1. Deben ser identificadores globales
2. Deben conocerse en tiempo de compilación
3. Su duración de almacenamiento debe ser estática

Las cláusulas de captura son un modo indirecto de dar acceso desde la lambda a identificadores que, de otro modo, serían inaccesibles. Las variables capturadas de una lambda son clones de la variable original. Comparten identificador y valor, pero puede que no tipo.

Las variables que aparecen en la cláusula de captura de una lambda se convierten en variables miembro del objeto especial functor que crea la lambda.

Para variables capturadas por valor, podemos añadir a la definición de la lambda la palabra clave “**mutable**”, lo que elimina el calificador “**const**” (en este contexto) y permite a la lambda modificar el valor de la variable clonada. Pero la modificación solo repercutirá en la versión clonada de la variable definida en la lambda. Si accedemos a la variable original el valor de esta no habrá cambiado.

Pero también podemos capturar por referencia, no solo por valor. Si lo hacemos por referencia no será marcada como constante, a menos que la de origen sí esté marcado como tal, es entonces cuando la captura también será “**const**”.

En una lambda podemos capturar múltiples variables separándolas con comas. Se pueden combinar por referencia o por valor como quieras.

También es posible utilizar "**capturas predeterminadas**" usando el operador **=** como cláusula las capturaremos todas por valor, y si usamos **&** por referencia. Las variables a capturar deben estar definidas en el ámbito externo de la lambda, como cuando las capturamos explícitamente. El compilador generará implícitamente una lista de las variables siempre y cuando se mencionen o usen dentro del cuerpo de la lambda. Se pueden combinar las predeterminadas con las explícitas sin problema.