Seminario C++11 y C++14

Equipo #2:

Wendy Díaz Ramírez

Eliane Puerta Cabrera

Liset Alfaro González

Leandro González Montesino

Joel David Hernández Cruz

Uno de los errores principales en la programación de estilo C es la pérdida de memoria. Las fugas a menudo se deben a un error al llamar a `delete` para la memoria asignada con `new`. C++ Moderno nos brinda en una manera fácil de resolver este problema a través de los punteros inteligentes, lo cual abordaremos con mayor nivel de especificación más adelante.

C++ Moderno destaca la estrategia para el manejo de recursos RAII (*Resource Acquisition Is Initialization*). La cual se encarga de liberar y reservar recursos en C++ y es utilizada en varios lenguajes orientados a objetos. En esta, los recursos se adquieren en la inicialización y son liberados en la destrucción del objeto, y garantiza que el único código que va a ser ejecutado después de ser lanzada una excepción son los destructores de los objetos que quedaron en la pila. Con esta técnica se garantiza un mecanismo de limpieza determinista, debido a que, es el usuario el encargado de eliminar los recursos que ya no se utilice. Esta eliminación determinista se contrapone a la seguida por otros lenguajes como C#, el que sigue una estrategia no determinista al utilizar una estrategia de *GarbageCollection*.

Para admitir la adopción sencilla de los principios de C++ RAII, la biblioteca estándar proporciona tres tipos de punteros inteligentes: *std:: unique_ptr*, *STD:: shared_ptr* y *STD:: weak_ptr.*

Un puntero inteligente controla la asignación y eliminación de la memoria que posee. Por ejemplo, si una instancia creada con el `new` tiene tipos por valor y tipos por referencia entre sus miembros, y se le hace `delete` solamente se destruirá la memoria de los tipos por valor y la variable que guarda la dirección de memoria de los tipos por referencia, pero no a lo que apuntan estos últimos. Sin embargo, si utilizáramos un puntero inteligente y decidiésemos que queremos liberar su memoria, bastaría con llamar a su método destructor, el cual se encargará de encapsular los llamados a los `delete` de cada uno de los recursos del tipo en cuestión.

A continuación, se dará una breve explicación de los tres punteros inteligentes introducidos en C++ 11:

std::unique_ptr:

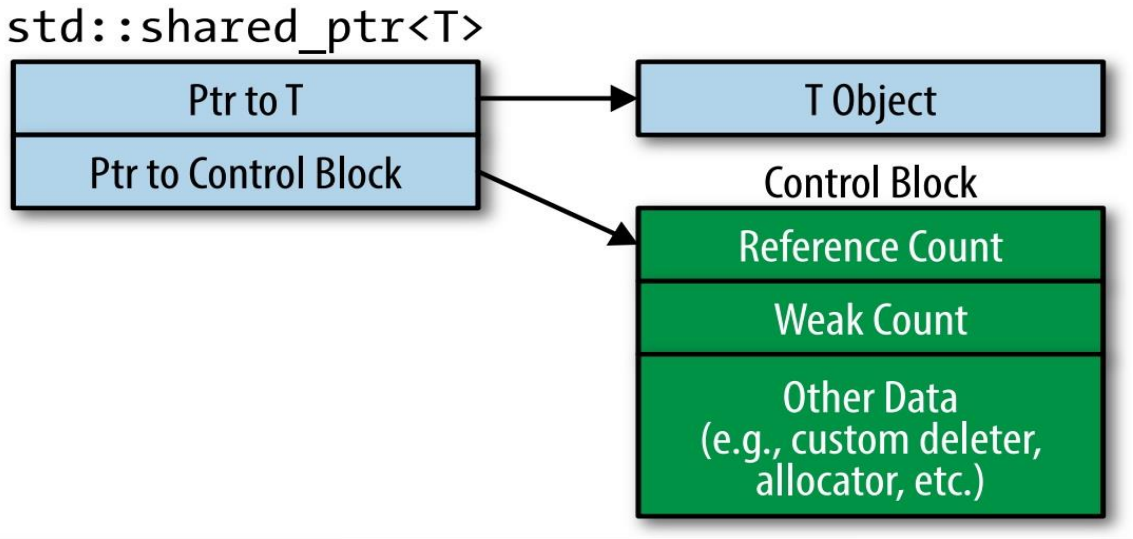
La clase *unique_ptr* reemplaza a *auto_ptr* que existe desde C++98. Incorporan semántica de pertenencia exclusiva, o sea siempre les pertenecen lo que apuntan. Son de tipo solo movimiento (*move-only type*), las copias entre *unique_ptr* no están permitidas. Su destrucción por defecto se realiza usando `delete` a los *raw_pointer* dentro del *unique_ptr*. Al ser creado, se puede especificar su forma de destrucción.

Ejemplo de declaración de un *unique_ptr*:

```
//Igualarlo a un puntero convencional
T* ptr = new T();
std::unique_ptr<T> uptr = ptr;  //error de compilación

//Pasando el raw pointer al constructor
T* ptr = new T();
std::unique_ptr<T> uptr(ptr);   //CORRECTO

//Llamar el método reset del unique_ptr
uptr.reset(ptr);    //CORRECTO

//¿Cómo crear un delete personalizado en un unique_ptr?
T* ptr = new T();
auto custom_deleter = [](T* object)//Lambda Expression
            {
                //write code here
                delete object;
            };

std::unique_ptr<T, decltype(custom_deleter)>
uptr(ptr, custom_deleter);
```

std::shared_ptr

Los *shared_ptr* son fáciles de crear a partir de los *unique_ptr*. Administran su tiempo de vida a través de la pertenencia compartida. Destruyen al objeto cuando el último *shared_ptr* deja de apuntar a este. Usa un destructor por defecto, pero también se puede crear uno personalizado al igual que en *unique_ptr*, aunque con un diseño diferente.



Implicaciones de la existencia del contador de referencia (*reference count*)

- Los *shared_ptr* son el doble del tamaño de los *raw pointer*, ya que internamente tienen un puntero al recurso de dato y otro al contador de referencia (*reference count*).
- La memoria para el *reference count* tiene que ser reservada dinámicamente.

- Las operaciones de incremento y decremento del contador de referencia tienen que ser atómicas, ya que su lectura y escritura es comparativamente costosa.
- Realizar *Move-constructing* de un *shared_ptr* a otro *shared_ptr* es más rápido que copiarlos, ya que copiarlos requiere incrementar el contador de referencia, no así usando el constructor *move*

Ejemplo de declaración de un *shared_ptr*:

```
//Igualarlo a un puntero convencional
T* ptr = new T();
std::shared_ptr<T> shptr = ptr; //ERROR COMPILACIÓN

//Pasando el raw pointer al constructor
T* ptr = new T();
std::shared_ptr<T> shptr(ptr);  //CORRECTO

//Llamar el método reset del shared_ptr
shptr.reset(ptr);           //CORRECTO
```

Reglas usadas en la creación del bloque de control (*control block*):

- *make_shared* siempre crea un bloque de control. Un bloque de control es creado cuando un *shared_ptr* es construido desde puntero de pertenencia única (ejemplo *unique_ptr* o *auto_ptr*).
- Cuando un *shared_ptr* es creado de un puntero convencional, se crea un bloque de control. No así si es creado a partir de otro *shared_ptr* o un *weak_ptr*, especificado como argumento del constructor

Consecuencias:

- Construir más de un *shared_ptr* desde un solo *raw pointe*r da un comportamiento indefinido.

Ejemplo:

```
int main(){
    int* p = new int;
    shared_ptr<int> sptr1(p);
    shared_ptr<int> sptr2(p);
}
```

	Reference count
shared_ptr<int> sptr1(p)	1
shared_ptr<int> sptr2(p)	1
main ends -> sptr1 goes out of scope	0 -> memory block pointed by sptr1 or p is destroyed as ref count is 0.
main ends -> sptr2 goes out of scope	0 -> Crashhhhhhh!!!!!. Because this tries to release memory block associated with sptr2, which is already being destroyed.

```
//¿Cómo crear un delete personalizado en un shared_ptr?
auto my_del = [](A *pw) // DESTRUCTOR PERSONZALIZADO
{
    delete pw;
};

//TIPO DEL DESTRUCTOR ES PARTE DEL TIPO DEL PUNTERO
std::unique_ptr<A, decltype(my_del)> upw(new A, my_del);

//TIPO DEL DESTRUCTOR NO ES PARTE DEL TIPO DEL PUNTERO
std::shared_ptr<A> spw(new A, my_del);
```

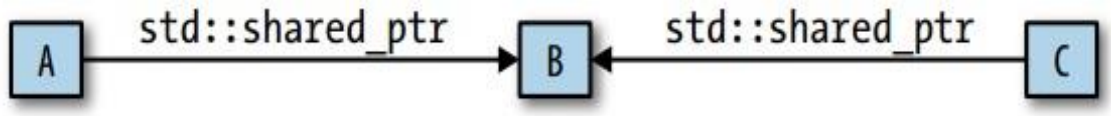
std::weak_ptr

Ejemplo de declaración de un *weak_ptr*:

```
// Igualarlo a un puntero convencional
T* ptr = new T();
std::weak_ptr<T> shptr = ptr;   //ERROR COMPILACIÓN

// Pasando el raw pointer al constructor
T* ptr = new T();
std::weak_ptr<T> shptr(ptr);    // ERROR COMPILACIÓN
// Llamar el método reset del shared_ptr
shptr.reset(ptr);           //CORRECTO

//Igualar el weak_ptr a un shared_ptr:
//COMPILA PERO NO INCREMENTA EL CONTADOR DE REFERENCIA
shared_ptr<T> shptr(new T());
weak_ptr<T> wptr = shptr;

//Pasando un shared_ptr al constructor del weak_ptr:
//COMPILA PERO NO INCREMENTA EL CONTADOR DE REFERENCIA
shared_ptr<T> shptr(new T());
weak_ptr<T> wptr(shptr);
```

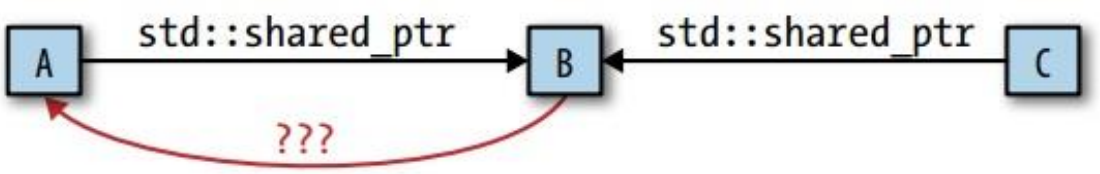
¿Para qué sirven los *std::weak_ptr*?

Supongamos que tenemos una estructura de datos con los objetos A, B y C, donde A y C tienen a B como recurso compartido, por lo que ambos (A y C) son *shared_ptr*.



Supongamos que deseamos tener un puntero desde B hasta A.

¿Qué tipo de puntero usar?



Tenemos tres opciones:

1. Puntero convencional. Si A es destruido, pero C continúa apuntando a B, entonces B contendrá un puntero a A que estará colgado. B no es capaz de detectar esto, por lo que puede desreferenciar este puntero, lo que provocaría un comportamiento indefinido.
2. *share_ptr*. Esto provocaría un ciclo entre A y B, ya que ambos son del mismo tipo y cada uno apunta al otro, previendo ser eliminados en cualquier momento. Será imposible para el programa acceder a ellos por lo que sus recursos nunca podrán ser reclamados.
3. *weak_ptr*. Evita ambos problemas de arriba. Si A es destruido, entonces puntero hacia atrás de B estará colgado y B será capaz de detectarlo. Además, aunque A y B apuntan el uno al otro, el puntero de B no afecta el contador de referencia de A, por lo que B no previene a A de ser eliminado.

Claramente usar *weak_ptr* es la mejor opción.

ALIAS

Desde C++3 se integró el *typedef* para brindar una forma de declarar un nombre para usar como sinónimo de un tipo declarado previamente, C++ 11 trae consigo una nueva manera de declarar un alias resolviendo las limitaciones del *typedef*.

```
//C++ 11
using counter = long;
// C++03 equivalent:
// typedef long counter;
```

¿Cuál es la verdadera diferencia entre uno y el otro?

Una limitación del mecanismo *typedef* es que no funciona con los templates. Sin embargo, la sintaxis de alias de tipo en C++ 11 permite la creación de templates de alias:

```
template<class T>
// Declaracion de la clase
class MyClass{
    public:
        MyClass();
        // Definir constructor de la clase
        template<class T> MyClass::MyClass(){ /*...*/ }
}

using MyClassInt = MyClass<int>;
```

Constructores clásicos de C++ (C++0x), el constructor *move* y las sobrecargas del operador =.

Características generales de los constructores:

- Son llamados cuando los objetos son creados.
- Permiten darle una inicialización a cada uno de los miembros de la clase.
- Es una función miembro de la clase, con su mismo nombre y no tienen valor ni tipo de retorno.

Constructor por defecto:

- Las mismas reglas que en C++98, generado solo si la clase no contiene constructores declarados por el usuario.
- No se especifican parámetros o en contrario, los valores de los parámetros tienen valores asignados por defecto.

  Ejemplo:

```
// Default empty constructor
DoubleLinkedList() { std::cout << "Default Constructor" << std::endl; }
```

Constructor de copia:

- Es llamado cuando se crea un objeto inicializándolo a partir de otro de la misma clase.
- Se le especifica un solo argumento que es una referencia constante a un objeto de la misma clase.
- Si el programador no lo ha definido, el compilador de C++ genera uno por defecto, el cual realiza una copia bit a bit entre los objetos.
- Es utilizado cuando una función recibe el objeto como un parámetro por valor.
- También se utiliza cuando una función tiene como valor de retorno un objeto.

```
// Copy Constructor
DoubleLinkedList(const DoubleLinkedList& lst){
    cout << "Constructor por Copia: " << endl;
    auto current = lst.first;
    while (current != nullptr){
        Add_Last(current->data);
        current = current->next;
    }
}
```

Constructor de un solo argumento:

En una clase que tenga un constructor que recibe un solo argumento, en C++ es permisible la siguiente sintaxis.

```
//One Argument constructor
DoubleLinkedList(T value) {
    cout << "Constructor de un solo elemento" << endl;
    Add_Last(value);
}

int main(){
    DoubleLinkedList<int> lls = 5;
}
```

Constructor explicit:

El uso de este tipo de sintaxis podría traer como consecuencia conversiones inesperadas en tiempo de ejecución, por lo que una solución preventiva es crear un constructor *explicit*, obligando al programador a realizar un *casteo* explícito a la clase que se desea instanciar.

```
//One Argument constructor
explicit DoubleLinkedList(T value) {
    cout << "Constructor de un solo elemento" << endl;
    Add_Last(value);
}

int main(){
    DoubleLinkedList<int> lls = (DoubleLinkedList<int>)5;
}
```

El operador de asignación (=):

- Destacar la diferencia entre object a = b; y a = b (para dos tipos a, b previamente inicializados). En el primer caso se llama al constructor de copia, y en el segundo es llamado el operador de asignación.
- Es generado por el compilador solo si el programador no lo ha declarado, y realiza una asignación bit a bit sobre los valores miembros de la clase.

```
//Assign by copy
DoubleLinkedList& operator=(DoubleLinkedList& A){
    cout << "Asignacion por copia: " << endl;

    if (this != &A){
        // Si no son el mismo objeto entonces
        first = nullptr;
        last = nullptr;
        length = 0;

        sPTR<Node<T>> temp = A.first;

        while( temp != nullptr) {
            Add_Last(temp->data);
            temp = temp -> next;
        }
    }

    return *this;
}

int main(){
    DoubleLinkedList<int> lls = {1, 2, 3, 4};
    DoubleLinkedList<int> lls2;
    lls2 = lls; // Aqui se llama al operador= por copia ya que en la parte derecha de la
asignacion tenemos una referencia a un lvalue (en este caso un DoubleLinkedList&)
}
```

A partir de C++11, se suman dos nuevos miembros de funciones, el constructor *move* y el operador de asignación *move.*

```
// Movement Constructor
DoubleLinkedList(DoubleLinkedList&& lst) noexcept {
    cout << "Constructor por Movimiento" << endl;
    *this = move(lst);
}

//Assign by movement
DoubleLinkedList& operator=(DoubleLinkedList&& A) noexcept {
    cout << "Asignacion por movimiento: " << endl;

    if (this != &A){

        first = nullptr;
        last = nullptr;
        length = 0;

        swap(first, A.first);
        swap(last, A.last);
        swap(length, A.length);
    }

    return *this;
}
```

Las reglas de generación y comportamiento son análogas a sus copias hermanas. Las operaciones *move* son generadas solo si son necesitadas, y si son generadas, ellas ejecutan movimientos a nivel de miembros sobre los miembros no estáticos de las clases.

Esto significa que el constructor *move* transfiere-construye cada dato de los miembros no estático de la clase desde el correspondiente miembro de su parámetro rhs, y el operador *move* transfiere-asigna cada miembro no estático desde su parámetro.

Las operaciones *move* son generadas en las clases (cuando son necesitadas) solo si los siguientes tres requisitos son verdaderos.

- No hay operaciones de copia declaradas en la clase.
- No hay operaciones de *move* declaradas en la clase.
- No hay destructor declarado en la clase.

rvalues

Los R-Valores son aquellos que solo pueden estar a la derecha de una expresión y que carecen de sentido a la izquierda.

```
int x;
1 = x;
&x = 3;
```

Es obvio que eso no se puede hacer, no podemos asignar un valor a un literal ni tampoco a un objeto temporal como es en este caso la dirección de memoria que creamos.

Donde usar los rvalues.

- Literales, excepto las cadenas, que son arrays
- El resultado de cualquier operador aritmético o lógico
- El retorno de una función que no devuelva una referencia

```
// rvalues:
int foobar();
int j = 0;
j = foobar(); //CORRECTO , foobar() ES UN rvalue
int* p2 = &foobar();// ERROR!, NO SE PUEDE PEDIR LA DIRECCIÓN DE UN rvalue
j = 42; // CORRECTO, 42 ES UN rvalue
```

lvalues:

Cualquier lvalue se puede comportar siempre como un rvalue.

```
int x, y = 4;
x = y;
```

Donde usar lvalue:

- Los nombres de variables y referencias

- El retorno de una función que devuelva una referencia

Hay tres cosas básicas que puedes hacer con un *lvalue* que no puedes hacer con un *rvalue*.

- Asignación con el operador =
- Obtener su dirección con el operador &.
- Usarlos para inicializar una referencia.

```
// lvalues:
int i = 42;
i = 43; // ok, i es un lvalue
int* p = &i; // ok, i es un lvalue
int& foo();
foo() = 42; // ok, foo() es un lvalue
int* p1 = &foo(); // ok, foo() es un lvalue
```

C++11 introduce un nuevo tipo de referencia denominada *rvalue* references, identificada por T&& (T es el tipo del objeto referenciado). Esta refiere a temporales que son modificables después de su inicialización, lo cual es la piedra angular de la semántica de movimiento.

- Son una referencia a un valor con la categoría de rvalue.
- Permiten distinguir un lvalue de un rvalue.
- Detectan perfectamente si un valor es un objeto temporal o no.

Semántica de movimiento:

Utilizando referencias rvalues podemos tener una sobrecarga de un método tomando una referencia a un *lvalue y otra tomando una referencia a un tipo rvalue (el compilador se encarga de chequear que método llamar según el tipo del valor).

```
int&& rvalue_ref = 99;
```

- El uso fundamental de una referencia rvalue es crear un constructor de movimiento y un operador de asignación de movimiento, que permitan mover un objeto en lugar de copiarlo.
- El constructor de movimiento y el operador de asignación toma una instancia de un objeto como argumento y crea una nueva instancia del argumento.
- El constructor de movimiento recibe referencias rvalue, por lo que conoce que el argumento es un objeto temporal.
- Evita relocalización de memoria en tiempo de ejecución porque no necesita copiar el valor, solo tiene que moverlo.
- El significado de construir un objeto con el constructor de movimiento es utilizar los recursos reservados del valor temporal en lugar reservar nuevos recursos y copiar los valores.
- referencias rvalue y semántica de movimiento permiten evitar copias innecesarias trabajando con objetos temporales.

Diferencias entre () y {} al crear objetos

La sintaxis seleccionada para inicialización de objetos en C++11 puede ser especificada con paréntesis, un signo de igual o llaves.

```
int x(0); // INICIALIZACIÓN ENTRE PARÉNTESIS
int y = 0; // INICIALIZACIÓN SEGUIDO "="
int z { 0 }; // INICIALIZACIÓN DENTRO LLAVES
int z = { 0 }; // INICIALIZACIÓN USA "=" Y LLAVES
```

Para los tipos *built-in* como *int*, la diferencia es académica, pero para los tipos definidos por el usuario es importante diferenciar la inicialización de la asignación, ya que distintos llamados de funciones son involucrados:

```
void main(){
    // LLAMA AL CONSTRUCTOR POR DEFECTO
    my_list w1;
    // NO ES UNA ASIGNACIÓN, LLAMA AL CONSTRUCTOR DE COPIA
    my_list w2 = w1;
    // UNA ASIGNACIÓN; LLAMA AL OPERADOR DE COPIA =
    w1 = w2;
}
```

C++11 introduce inicialización uniforme que permite inicializar objetos con términos entre llaves, además se puede usar para inicializar valores para miembros no estáticos de datos, compartida con la sintaxis de inicialización =, no así con los paréntesis.

```
std::vector<int> v{ 1, 3, 5 }; // V INICIALMENTE ES 1, 3, 5
class my_list {
//…
private:
    int x{ 0 }; // CORRECTO, VALOR POR DEFECTO DE X ES 0
    int y = 0; // CORRECTO
    int z(0); // ERROR!
};
```

La inicialización entre llaves prohíbe las conversiones de estrechamiento (*narrowing conversions*)

```
void main(){
    double x, y, z;
    //…
    int sum1{ x + y + z }; // ERROR! SUMA DE DOUBLE NO PUEDE SER EXPRESADA COMO INT
}
```

La inicialización usando paréntesis y “=” no comprueba conversiones de estrechamiento.

```
void main(){
    int x = 1, y = 2, z = 3;
    // CORRECTO (VALOR DE LA EXPRESIÓN TRUNCADO A UN ENTERO
    int sum2(x + y + z);
    int sum3 = x + y + z;
}
```

Otra de las características de la inicialización con llaves es la inmunidad de las molestias del parser de C++.

```
//LLAMA AL CONSTRUCTOR DE OBJECT CON ARGUMENTO 10
object obj(10);
```

Si llamas al constructor de object con cero argumentos usando una sintaxis análoga, entonces declaras una función en vez de un objeto.

```
//MOLESTIAS DEL PARSER DECLARA UNA FUNCIÓN LLAMADA obj
//QUE RETORNA UN object
object obj();
```

Las funciones no pueden ser declaradas usando llaves para la lista de parámetros, por lo que se puede llamar al constructor por defecto de la siguiente manera:

```
//LLAMA AL CONSTRUCTOR DE object sin argumento
object obj{};
```

En las llamadas a los constructores, los paréntesis y las llaves tienen el mismo significado, siempre que el parámetro std::initializer_list no esté involucrado.

Ejemplo:

```
class A {
public:
    A(int i, bool b) {}
    A(int i, double d) {}
};
void main(){
    A w1(10, true); // LLAMA AL PRIMER CONSTRUCTOR
    A w2{ 10, true }; // LLAMA AL PRIMER CONSTRUCTOR
    A w3(10, 5.0); // LLAMA AL SEGUNDO CONSTRUCTOR
    A w4{ 10, 5.0 }; // LLAMA AL SEGUNDO CONSTRUCTOR
}
```

La determinación de los compiladores para emparejar inicializadores usando llaves con los constructores que reciben std::initializer_lists es tan fuerte, que gana igual si el mejor emparejamiento del constructor que recibe std::initializer_list no puede ser llamado.

```
class A {
public:
    A(int i, bool b);
    A(int i, double d);
    A(std::initializer_list<bool> il);
};

void main(){
// ERROR! REQUIERE CONVERSIÓN DE ESTRECHAMIENTO
    A w{ 10, 5.0 };
}
```

Dicho error existe porque la llamada al constructor necesita convertir un int (10) y un double (5.0) a bool, pero ambas conversiones serían de estrechamiento (*narrowing*) y

las conversiones de estrechamiento (*narrowing conversions*) son prohibidas dentro de inicializadores que usan llaves; por lo que el llamado es inválido.

Solo si no hay manera de convertir los tipos de argumentos en una inicialización por llaves, a el tipo de la std::initializer_list entonces el compilador regresa a la resolución normal de la sobrecarga.

```
class A {
public:
    A(int i, bool b) {…}
    A(int i, double d) {…}
// std::initializer_list ELEMENTO DE TIPO std::string
    A(std::initializer_list<std::string> il){…}
};
int main(){
    A w1(10, true); // LLAMA AL PRIMER CONSTRUCTOR
    A w2{ 10, true }; // LLAMA AL PRIMER CONSTRUCTOR
    A w3(10, 5.0); // LLAMA AL SEGUNDO CONSTRUCTOR
    A w4{ 10, 5.0 }; // LLAMA AL SEGUNDO CONSTRUCTOR
}
```

Supongamos que usamos un conjunto vacío de llaves para construir un objeto que soporta el constructor por defecto y también soporta el constructor que recibe std::initializer_list.

¿Qué significan las llaves vacías?

1) Si significan “no argumento”, podemos obtener el constructor por defecto.
2) Si significan “vacía std::initializer_list”, podemos obtener un constructor de un std::initializer_list sin elementos.

```
class A {
public:
    //...
    A() {/*...*/}
    A(std::initializer_list<int> il){/*...*/}
};
void main(){
    A w1; // LLAMA AL CONSTRUCTOR POR DEFECTO
    A w2{}; // LLAMA AL CONSTRUCTOR POR DEFECTO
    A w3(); // MOLESTIAS DEL PARSER! ¡DECLARA UNA FUNCIÓN!
}
```

¡La regla es que se obtiene un constructor por defecto! Llaves vacías significan no argumento, no un vacío std::initializer_list

Si se quiere llamar un constructor std::initializer_list con un vacío std::initializer_list, se debe poner las llaves vacías como argumento de un constructor, o sea, poniendo las llaves vacías dentro de los paréntesis o las llaves.

```
//LLAMA AL CONSTRUCTOR std::initializer_list CON UNA LISTA VACÍA
A w1({});
A w2{ {} };
```

¿Es necesario entender bien el papel de () y {} en creación de objetos?

Veamos el siguiente ejemplo:

```cpp
// USA non-std::initializer_list
// CONSTRUCTOR: CREA 10-ELEMENTOS std::vector, TODOS //CON VALOR 20
std::vector<int> v1(10, 20);

// USA std::initializer_list
// CONSTRUCTOR: CREA 2-ELEMENTOS
// std::vector, LOS VALORES DE LOS ELEMENTOS SON 10 Y 20
std::vector<int> v2{ 10, 20 };
```

Expresiones Lambdas:

- También conocidas como clausuras, funciones lambdas, funciones literales o simplemente lambdas este término es comúnmente usado en los lenguajes de programación considerados como funcionales.
- C++11 trae consigo el uso de las mismas para darle más poder y flexibilidad al lenguaje.
- Todo lo que podemos hacer con una expresión lambda es posible hacer a mano.

Sintaxis básica de una expresión Lambda en C++ 11:
[*captures*] (*parameters*) -> *returnTypesDeclaration* { *lambdaStatements*; }

Paquetes de Parámetros:

En C++ 11 se incluyó una manera de pasar una cantidad n de parámetros a funciones y templates. La sintaxis que siguen es la siguiente:

type **...** *Args*(optional)	(1)
typename|class ... *Args*(optional)	(2)
template < *parameter-list* **> typename**(C++17)**|class ...** *Args*(optional)	(3)

En funciones:

Args **...** *args*(optional)	(4)

Una extensión de este ***feature***:

pattern **...**	(5)

1. Un conjunto de parámetros de ***template*** sin tipo con un nombre opcional.
2. Un conjunto de parámetros de ***template*** de algún tipo con un nombre opcional.
3. Un ***template*** de un conjunto de parámetros de ***template*** con un nombre opcional.
4. Un conjunto de parámetros para una función con un nombre opcional.

A este recurso se le conoce en el lenguaje como ***variadic templates***. Algunos ejemplos para entender mejor esto:

Un ***variadic class template*** puede ser instanciado con cualquier número de argumentos:

```
template<class ... Types> struct Tuple {};
Tuple<> t0;           // Types no contiene argumentos
Tuple<int> t1;        // Types contiene un argumento: int
Tuple<int, float> t2; // Types contiene dos argumentos: int and float
Tuple<0> error;       // error: 0 no es un tipo
```

Un ***variadic function template*** puede ser invocado con cualquier número de argumentos para la función.

```
template<class ... Types> void f(Types ... args);
f();       // OK: args no contiene argumentos
f(1);      // OK: args contiene un solo argumento: int
f(2, 1.0); // OK: args contiene dos argumentos: int and double
```

En un ***primary class template*** el paquete de parámetros debe ser definido al final de la lista de parámetros.

```
template<typename... Ts, typename U> struct Invalid; // Error: Ts.. No esta definido al
final
```

Sin embargo, en un ***variadic function template*** el paquete de parámetros puede aparecer antes en la declaración del template, siempre que el resto de los parámetros pueda ser deducido de los argumentos de la función. Por ejemplo:

```
template<typename ...Ts, typename U, typename=void>
void valid(U, Ts...);     // OK: Aqui es posible deducir U en el llamado a la funcion
// void valid(Ts..., U);  // Error: Ts... En esta posicion no se puede deducir cuando
termina Ts

valid(1.0, 1, 2, 3);      // OK: deduce U como un double, Ts como un {int,int,int}
```

Estos paquetes de parámetros pueden ser usados en otros lugares, como por ejemplo en las capturas de funciones lambdas, constructores, etc.

```
// Lambdas Expression
template<class ...Args>
void f(Args... args) {
    auto lm = [&, args...] { return g(args...); };
    lm();
}

// Initializer List Constructor
template<class... Mixins>
class X : public Mixins... {
 public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

Destructores:

Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase. El destructor realizará procesos necesarios cuando un objeto termine su ámbito temporal; por ejemplo, liberando la memoria dinámica utilizada por dicho objeto o liberando recursos usados, como ficheros, dispositivos, etc.

Al igual que los constructores, los destructores también tienen algunas características especiales:

- También tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo ~ delante.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No tienen parámetros.
- No pueden ser heredados.
- Deben ser públicos, no tendría ningún sentido declarar un destructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.
- No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.

Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido. Esto es así salvo cuando el objeto fue creado dinámicamente con el operador `new`, ya que, en ese caso, cuando es necesario eliminarlo, hay que hacerlo explícitamente usando el operador `delete`.

Si no se define un destructor, el compilador provee uno por defecto. En general esto es suficiente; solo es necesario definir un destructor personalizado cuando la clase tiene datos miembros de tipo puntero, que apuntan a recursos del sistema que necesitan ser liberados, o que tienen la propiedad del objeto al que apuntan.

Ejemplo:

```
// USA non-std::initializer_list
// CONSTRUCTOR: CREA 10-ELEMENTOS std::vector, TODOS //CON VALOR 20
std::vector<int> v1(10, 20);

// USA std::initializer_list
// CONSTRUCTOR: CREA 2-ELEMENTOS
// std::vector, LOS VALORES DE LOS ELEMENTOS SON 10 Y 20
std::vector<int> v2{ 10, 20 };
```

Se declaran siguiendo las siguientes reglas:

- No aceptan argumentos
- No retornan ningún valor
- No pueden ser declarados como **const**, **volatile**, o **static**. Sin embargo, pueden ser invocados para destruir objetos declarados como **const**, **volatile**, o **static**.
- Pueden declararse como **virtual**. Al usar destructores virtuales se pueden destruir objetos sin saber su tipo; el destructor correcto es invocado a través del mecanismo de las funciones virtuales. También pueden ser declarados como funciones puramente virtuales de clases abstractas.

Se invocan cuando ocurre alguno de los eventos siguientes:

- Un objeto local de un block scope se sale de su scope.
- Un objeto inicializado con el operador **new** se destruye explícitamente usando **delete.**
- El tiempo de vida de un objeto temporal termina.
- Un programa termina y todavía existen objetos estáticos o globales.
- Se llama explícitamente al destructor.

Los destructores pueden libremente llamar a funciones miembro de la clase y acceder a datos miembros de la clase.

Hay dos **restricciones para el uso** de los destructores:

- No se puede coger su dirección.
- Las clases derivadas no heredan el destructor de su clase base.

Cuando un objeto se sale del scope o es eliminado, la **secuencia de eventos de su destrucción** ocurre como sigue:

1. El destructor de la clase es llamado y se ejecuta el cuerpo de la función.
2. Los destructores de los miembros no estáticos se llaman en orden inverso a como aparecen declarados en la clase.
3. Los destructores de las clases base no virtuales se llaman en orden inverso a su declaración.
4. Los destructores de las clases base virtuales se llaman en orden inverso a su declaración.

Raw Pointers:

Un *raw* pointer es un puntero cuyo tiempo de vida no está siendo controlado por ningún objeto que lo encapsule, como en el caso de los *smart* pointers. Se le puede asignar la dirección de otra variable que no sea un puntero, o se le puede asignar el valor *nullptr*.

Es preferible usarlos en los siguientes casos:

- Iteradores
- Punteros a función
- Cuando no se está seguro de qué tipo de pertenencia se tendrá sobre un objeto
- Cuando se quiere compartir pertenencia sobre un objeto y se quiere ahorrar memoria.

NoExcept:

El operador *Noexcept* realiza una comprobación en tiempo de compilación que devuelve *TRUE* si una expresión declara que no lanza excepción

Sintaxis:

noexcept(expresión) devuelve un *prvalue* de tipo *bool, noexcept* no evalúa la expresión, sino que devuelve *True* si el conjunto de excepciones potenciales de la expresión está vacío, de lo contrario lanza false. El conjunto de funciones potenciales no es vacío si en

la expresión aparecen una expresión *throw*, un *dynamic_cast*, un *typeid* cuando el tipo del argumento es de una clase polimórfica, etc

```
//si foo se declara noexcept depende de si la expresión
//T() lanzará alguna excepción
template <class T>
void foo() noexcept(noexcept(T())) { }

void bar() noexcept(true){}
void baz() noexcept {throw 42;}

int main(){
    foo<int>(); // noexcept(noexcept(int)) => noexcept(true)
    bar();//bien
    baz(); //compila pero en tiempo de ejecución llama a std::terminate
}
```

Inferencia de tipos:

En C++11 encontramos una manera más amena de escribir nuestro código y evitar las extensas líneas de definiciones de tipos, pues tenemos *auto*, con función semejante al *var* de C#.

```
auto v = 4 ;          // v es de tipo int
```

Lo principal que debe recordar al usar auto es esto: use *auto* siempre que crea que mejora la legibilidad del código y evítelo siempre que oscurezca la intención del código. Al final del día, estás escribiendo el código para la siguiente persona que recoge tu módulo y no el compilador, ¿verdad?

También tenemos *decltype()*, que nos permite declarar una variable simplemente especificando que su tipo es el mismo que el de la variable que recibe como argumento.

```
decltype(v) p ;
```

Estas funciones pueden ser usadas en conjunto y pueden hacer menos extenso el código. De igual modo se debe ser cuidadoso, ya que se pudieran tener efectos no deseados.

Diferencia entre *decltype* y *auto*

1. *decltype* proporciona el tipo declarado de la expresión que se le pasa, mientras que auto hace lo mismo que la deducción de tipo de plantilla. Entonces, por ejemplo, si tiene una función que devuelve una referencia, auto seguirá siendo un valor (necesita *auto &* para obtener una referencia), pero *decltype* será exactamente el tipo del valor de retorno.

2. auto se limita a definir el tipo de una variable para la que hay un inicializador, mientras que *decltype* es una construcción más amplia que, a costa de información adicional, inferirá el tipo de una expresión.

En C++ 14 tenemos una mejora: *decltype(auto)*, este también resuelve el problema planteado que *decltype* pero utilizando una nueva sintaxis. *Auto* deduce el tipo de retorno de la función y *decltype()* lo devuelve.

Punteros a función:

Los punteros a función se usan mayormente para pasar funciones a otras funciones. En este escenario, el que llama puede personalizar el comportamiento de una función sin modificar la función en sí. En C++ moderno, las expresiones lambda proveen la misma funcionalidad con mucha mayor seguridad de tipo y otras ventajas.

La declaración de un puntero a función específica la signatura que la función a la que apunta debe tener:

```
// Declare pointer to any function that...

// ...accepts a string and returns a string
string (*g)(string a);

// has no return value and no parameters
void (*x)();

// ...returns an int and takes three parameters
// of the specified types
int (*i)(int i, string s, double d);
```

Ejemplo:
La función **combine** acepta como parámetro a cualquier función que reciba un **std::string** y devuelva un **std::string**. En dependencia de la función que se le pase a **combine** se agregará un string al inicio, o al final.

```
#include <iostream>
#include <string>

using namespace std;

string base {"hello world"};

string append(string s){
    return base.append(" ").append(s);
}

string prepend(string s){
    return s.append(" ").append(base);
}

string combine(string s, string(*g)(string a)){
    return (*g)(s);
}
int main(){
    cout << combine("from MSVC", append) << "\n";
    cout << combine("Good morning and", prepend) << "\n";
}
```

No es posible definir punteros a los **constructores** o **destructores** de clase, ya que son un tipo especial de funciones miembro de las que no puede obtenerse su dirección.

Son variables de tipo: "puntero-a-función recibiendo A argumentos y devolviendo X", donde A son los argumentos que recibe la función y X es el tipo de objeto devuelto. Cada una de las infinitas combinaciones posibles da lugar a un tipo específico de puntero-a-función.

Ejemplos:

`void (*fptr)();`	fptr es un puntero a una función, sin parámetros, que devuelve **void**.
`void (*fptr)(int);`	fptr es un puntero a función que recibe un **int** como parámetro y devuelve **void**.
`int (*fptr)(int, char);`	fptr es puntero a función, que acepta un **int** y un **char** como argumentos y devuelve un **int**.
`int* (*fptr)(int*, char*);`	fptr es puntero a función, que acepta punteros a **int** y **char** como argumentos, y devuelve un puntero a **int**.
`void (_USERENTRY * fptr)(void);`	fptr es puntero a función, sin parámetros que devuelve **void** y utiliza la convención de llamada _USERENTRY
`LONG (PASCAL * lpfnWndProc)();`	lpfnWndProc es puntero a función, sin parámetros que devuelve LONG y utiliza la convención de llamada PASCAL.

Cuando el valor devuelto por la función es a su vez un puntero (a función, o de cualquier otro tipo), la notación se complica un poco más:

`int const * (*fptr)();`	fptres un puntero a función que no recibe argumentos y devuelve un puntero a un **int** constante
`float (*(*fptr)(char))(int);`	fptr es un puntero a función que recibe un **char** como argumento y devuelve un puntero a función que recibe un **int** como argumento y devuelve un **float**.
`void * (*(*fptr)(int))[5];`	fptr es un puntero a función que recibe un **int** como argumento y devuelve un puntero a un array de 5 punteros-a-**void** (genéricos).
`char (*(*fptr)(int, float))();`	fptr es un puntero a función que recibe dos argumentos (**int** y **float**), devolviendo un puntero a función que no recibe argumentos y devuelve un **char**.
`long (*(*(*fptr)())[5])();`	fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un array de 5 punteros a función que no reciben ningún parámetro y devuelven **long**.

Los punteros a función también pueden agruparse en arrays:

```
int (* afptr[10])(int);    // matriz de 10 punteros a función
```

De la propia definición se desprende que, en estos arrays, todos los punteros señalan funciones que devuelven el mismo tipo de valor y reciben los mismos tipos de parámetros.

La gramática C++ acepta la existencia de arrays de punteros-a-función, que resultan muy útiles, pero **no** de arrays de funciones. Sin embargo, en la práctica son equivalentes; para casi todos los efectos una función puede sustituirse por su puntero. Estos arrays permiten invocar funciones (a través de sus punteros) utilizando los elementos del array mediante notación de subíndices. Por ejemplo, con el caso anterior se pudiera hacer algo como:

```
int z = afptr[n](x);
```