Primer Proyecto de LP – C++11, C++14

Equipo 2:

Alejandro Escobar Giraudy C312

Airelys Collazo Perez C312

Henri Daniel Peña Dequero C311

Orientación:

Implemente una clase **linked\_list** doblemente enlazada en **C++** (haciendo uso extensivo de los

elementos novedosos en el lenguaje desde **C++11**) que cumpla los siguientes requerimientos:

1. Definir las clases genéricas **linked\_list** y **node**.

a. No es necesario una exposición en genericidad, ya el Equipo 1 tocó este punto.

2. Definir miembros de datos necesarios de ambas clases.

a. ¿Cuáles son los nuevos elementos introducidos a partir de **C++11** que permiten un manejo

más “inteligente” de la memoria?

b. ¿Cómo deben inicializarse?

c. ¿Cuál es la filosofía en el uso de la memoria defendida por **C++**?

d. Usar alias para simplificar nombres de tipos.

3. Definir los constructores clásicos de **C++ (C++0x)**, el constructor move y las sobrecargas del

operador=.

a. ¿Qué hace cada uno de ellos? ¿Cuándo se llaman?

b. ¿Qué es un **lvalue** y un **rvalue**?

c. Explique **std::move** .

4. Definir un constructor que permita hacer **list-initialization** lo más parecido a **C#** posible.

a. Compare la utilización del **{}** v.s **()**.

5. Definir un constructor que reciba un vector<T>.

a. Usar for\_each con expresiones lambda.

6. Definir el destructor de la clase.

a. ¿Hace falta?

b. ¿Para qué casos haría falta un **puntero crudo** (**raw pointer**)?

7. Definir funciones **length**, **Add\_Last**, **Remove\_Last**, **At**, **Remove\_At**

a. Explique **Noexcept**.

b. Inferencia de tipo en **C++** (**auto**, **decltype**, **decltype(auto)**). Explicar todos, pero no

obligatoriamente usarlos.

8. Crear un puntero a función **Function<R, T…>** que devuelve un valor de tipo **R** y recibe un número variable de parámetros de tipo **T**.

a. Definir una función genérica **Map** a **linked\_list** en **T** y **R**, que recibe un puntero a función

que transforma un elemento **T** en uno **R**; de manera que **Map** devuelve una instancia de

**linked\_list<R>** resultado de aplicar a todos los elementos **T** de la lista original la función de

transformación.

b. Crear punteros a funciones usando alias.

c. Crear un puntero a función **Function** que permita cualquier cantidad de parámetros de

cualquier tipo.

-------------------------------------------------------------------------------------------------------------------------------

Introducción

El objetivo de este proyecto en efecto es la realización de una lista doblemente enlazada mediante el uso de conceptos y recursos específicos de **C++**, sobre todo **C++11** en adelante (siendo **C++14** y **C++17** las versiones posteriores).

Para ellos definimos las clases **linked\_list** y **node**, que explicaremos a continuación.

1-)

La clase node posee los atributos:

**T value** -> El valor que encapsula el nodo

**shared\_ptr<node<T>> previous** -> Un puntero al nodo anterior

**shared\_ptr<node<T>> next** -> Un puntero al nodo siguiente

Su constructor:

**node<T>(T in\_value) ->** Inicializa la clase a partir del comando **new**

Y la función:

**T Get\_Value()** -> Devuelve el valor en el nodo

Por otro lado.

La clase **linked\_list** posee los atributos:

**shared\_ptr<node<T>> first** -> Un puntero al primer nodo de la lista

**shared\_ptr<node<T>> last** -> Un puntero al último nodo de la lista

**int count** -> Contador que representa la cantidad de elementos en la lista

Constructores:

**linked\_list<T>()** -> Inicializa la clase a partir del comando **new**

**void operator =(linked\_list<T> list)** -> Con esto redefinimos el operador **=** para la creación de la clase.

Y sus funciones:

**shared\_ptr<node<T>> At\_Node(int index)** -> Dado un índice, devuelve el nodo perteneciente a dicha posición. Esta es la única función privada de la clase **linked\_list**, puesto que no es de interés que el usuario use los nodos de la lista, sino los valores que contienen los nodos.

**T At(int index)** -> Dado un índice, devuelve el valor almacenado en el nodo perteneciente a dicha posición

**void Add\_Last(T value)** -> Añade un elemento al final de la lista

**T Remove\_Last()** -> Elimina el último elemento de la lista y lo devuelve

**T Remove\_At(int index)** -> Dado un índice, elimina el elemento correspondiente a la posición seleccionada, y luego lo devuelve

**void Print\_Starting\_First()** -> Imprime en orden en pantalla todos los elementos de la lista, empezando por el primero y terminando por el último

**void Print\_Starting\_Last()** -> Imprime en orden en pantalla todos los elementos de la lista, empezando por el último y terminando por el primero

a)

La genericidad en **C++** se logra utilizando la plantilla **template <class T>** antes de una clase, y permite el uso de variables genéricas en la misma. **Node** y **linked\_list** lo utilizan para poder tener elementos de tipo **T**. No profundizaremos en el tema de la genericidad en **C++** puesto que no es objetivo del proyecto.

2-)

**C++11** tuvo muchas novedades con su lanzamiento, entre ellos podemos enumerar:

-Extensión del uso del inicializador de listas, pudiendo ser utilizado por todas las clases, incluso por contenedores como **std::vector**.

-Agrego de las palabras claves **auto** y **decltype**, que dan facilidad a la inicialización de variables que no se conoce de antemano de qué tipo son (por ejemplo, con la devolución de funciones).

-El for basado en rango, que se comporta de manera similar al **foreach** de **C#** o **Java**.

Entre otras cosas.

Pero entre ellos una de las novedades más importantes estuvo relacionada con la gestión de la memoria de forma “inteligente”, en este caso, fueron agregados los punteros inteligentes (o smart pointers).

Estos punteros no fueron solo un agrego a **C++**, resolvían muchos de los problemas que encontraba el usuario con los antiguos punteros y facilitaban al programador el trabajo con la memoria, al tener características diferentes y bien definidas, como, por ejemplo, no es necesario utilizar **delete** para borrar el espacio en memoria utilizado por ellos, ya que lo hacen automáticamente en cuánto todas las referencias a un espacio en memoria han sido eliminadas.

Anteriormente, uno de los punteros utilizados era **std::auto\_ptr**, este puntero solo permitía un apuntador a la vez a su espacio en memoria correspondiente, y generaba problemas a los programadores, sobre todo en el uso de contenedores, donde fácilmente terminaban perdiendo al puntero en descuidos al moverlo.

Este antiguo puntero fue reemplazado por un nuevo puntero, **std::unique\_ptr** (si bien se continuó manteniendo los **auto\_ptr** en **C++11**, estos no eran recomendados, e incluso, a día de hoy ya fueron eliminados del lenguaje).

Este nuevo puntero en esencia contaba con lo mismo, solo podía existir un puntero a un espacio en memoria determinado, y en el momento en el que ese puntero dejaba de existir, ese espacio en memoria sería borrado. La novedad yace en que el contenido del espacio en memoria de dicho puntero se puede transferir a otro, utilizando el comando **std::move**, haciendo que el primero entonces apunte a **nullptr** (incluido en **C++11** para facilitar el trabajo con punteros nulos, que originalmente tenían por defecto 0 como dirección de memoria con valor nulo).

Otro puntero inteligente añadido fue el **std::shared\_ptr**. Este puntero a diferencia del anterior, sí permite compartir la propiedad del espacio en memoria, en otras palabras, pueden existir varios apuntadores a dicho espacio. Cuando todos los apuntadores a un mismo espacio en memoria han sido eliminados, entonces lo que hay guardado en dicho espacio es automáticamente eliminado. El puntero se puede transferir mediante **std::move** igualmente al puntero anterior, a pesar de sus propiedades.

El tercer y último tipo de puntero inteligente agregado el **std::weak\_ptr**, su similitud con el anterior es que igualmente se pueden mantener varias referencias al mismo espacio en memoria, sin embargo, este tipo de puntero no participa en la regla de “una vez que todas las referencias a un espacio en memoria han sido eliminadas, se borra el contenido de dicho espacio”. En otras palabras, no participa en el número de referencias a dicho espacio, lo cual es útil en contextos donde se deba entregar un puntero, pero no depender de él para eliminar el espacio en memoria (por ejemplo, en salidas de métodos de una clase que utiliza por dentro sus propios punteros tal vez). El puntero se puede transferir mediante **std::move** igual que los anteriores.

De todos estos tipos de punteros solo fue utilizado en el proyecto el **shared\_ptr** (no se visualiza con **std::** puesto que fue declarado el comando “**using namespace std;**” anteriormente), tal y como se pudo observar anteriormente en los atributos de **node** y **linked\_list.**

b)

Su inicialización es sencilla (asumamos que se está utilizando “**using namespace std;**” para no utilizar constantemente **std::**) .Antes, para declarar un puntero, se debía colocar un **\*** junto al tipo, supongamos el caso de node para valores enteros, hubiera sido:

**node<int>\* a = new node<int>(5);**

En este caso los punteros se utilizan directamente como clases, necesitando el tipo que recibirán entre **< >**, en caso de que se inicialicen directamente.

Tomemos como ejemplo la clase **node<T>** para explicar el cómo crear punteros inteligentes.Para los **shared\_ptr** se hace de la siguiente forma:

**shared\_ptr<node<T>> new\_node = shared\_ptr<node<T>>(new node<T>(value));**

O mediante la función **make\_shared<type>(parameter)**de la siguiente forma:

**shared\_ptr<node<T>> new\_node = make\_shared<node<T>>(new node<T>(value));**

Donde value es un valor de tipo **T**.

Para el caso de los **unique\_ptr** se hace con la función **make\_unique<type>(parameter)** de la siguiente forma:

**unique\_ptr<node<T>> new\_node = make\_unique<node<T>>(new node<T>(value));**

Otra forma de instanciar **unique\_ptr** es:

**unique\_ptr<node<T>> new\_node = new unique\_ptr<node<T>>(new node<T>(value));**

Y en el caso de los **weak\_ptr**, estos no pueden apuntar a un espacio en memoria por sí solos, depende de que ya existan un **shared\_ptr** que apunte a un espacio en memoria determinado. Esto se logra pasando por argumento el **shared\_ptr** al **weak\_ptr** de la siguiente forma:

**shared\_ptr<node<T>> new\_node = shared\_ptr<node<T>>(new node<T>(value));**

**weak\_ptr<node<T>> new\_node = new weak\_ptr<node<T>>(new\_node);**

c)

La filosofía del uso de memoria defendida por **C++** es que sea el programador quien controle la memoria que es utilizada por los programas. Esta debe ser asignada y borrada manualmente (a diferencia de lenguajes como **C#** por ejemplo, que posee el llamado **Garbage Collector**), de forma que es necesario una cuidadosa gestión en función de evitar un derroche de memoria innecesario. Si bien los nuevos punteros inteligentes automatizan y facilitan la gestión en memoria, el programador sigue teniendo que gestionar la memoria del resto del programa, así como no debe abusar de los punteros inteligentes tampoco para dicha gestión.

d)

Los alias en **C++** es un mecanismo para redefinir el nombre de una clase, dándole al programador la capacidad de utilizar nombres más cómodos para las clases tanto definidas por él mismo como las ya existentes en **C++**.

Para usar un **alias** se utiliza la palabra clave **using**, seguido del nuevo nombre y se iguala al tipo de la siguiente forma:

**using new\_name = class\_name;**

Es un recurso bastante sencillo de utilizar, sobre todo con clases con nombres extensos.

En el proyecto por ejemplo fue utilizado a modo de demostración en la clase **node** de la siguiente forma:

**using shared = shared\_ptr<node<T>>;**

3-)

a)

Los constructores de una clase son funciones que permiten crear una instancia de una clase.

En **C++** se dividen en 3 tipos, el constructor clásico, el constructor **move** y la sobrecarga del operador **=**.

El constructor clásico es aquel que permite inicializar una clase (devolviendo un puntero de la misma), es una función que no tiene parámetro de devolución, y su nombre coincide con el nombre de la clase. Se utiliza de la siguiente forma:

**Class\_name(parameters)**

**{**

//Aquí se deben inicializar los atributos de la clase

**}**

Un ejemplo más concreto lo podemos tomar del constructor de **node**, que en el proyecto se implementó de la siguiente forma:

**node<T>(T in\_value)**

**{**

**this->value = in\_value;**

**this->previous = nullptr;**

**this->next = nullptr;**

**}**

Este constructor es utilizado cuando, por ejemplo, dada la clase **node**, hacemos:

**node<int>\* a = new node<int>(5);**

Si una clase no define un constructor clásico como el anterior, se le definirá uno por defecto, el cuál no recibe parámetros.

Hay un uso específico del constructor clásico que se llama **Copy Constructor**, la idea de este constructor es utilizar la idea anterior para, en vez de copiar las referencias, copiar los valores de los atributos de una instancia de una clase a otra instancia de la misma clase. Para esto se debe recibir como parámetro una instancia de la misma clase a la que se le realiza el constructor.

La idea sería la siguiente, suponiendo el ejemplo con lo implementado en la clase **node**:

node<T>(const node<T> &other\_node)

{

this.value = other\_node.value;

this.previous = other\_node.previous;

this.next = other\_node.next;

}

Los cambios en general serían:

-El uso de class\_instance.atribute en vez de class\_instance->atribute, puesto que ahora se está copiando el valor del atributo en vez de su referencia.

-El uso de &, puesto que recibimos por parámetro una referencia, debemos acceder al valor del espacio en memoria al que apunta

-El uso de la palabra clave **const**, para evitar que la instancia que se pasa como argumento a la función sea modificada.

El segundo tipo de constructor es la sobrecarga del operador **=**. Básicamente se trata de redefinir qué ocurre cuando queremos instanciar una clase a partir de igualarla a otra instancia ya existente de la misma clase.

Este operador se sobrecarga de la siguiente forma:

**void operator =(class\_name class\_instance)**

**{**

//Aquí se pasan los parámetros de **class\_instance** a nuestra

//instancia de la clase

**}**

Un ejemplo del uso de esto lo podemos dar a partir de lo que hicimos en **node**:

**void operator =(node<T> new\_node)**

**{**

**this->value = new\_node->Get\_Value;**

**this->previous = new\_node->previous;**

**this->next = new\_node->next;**

**}**

Que se ve reflejado cuando hacemos, por ejemplo:

**node<int>\* a = new node<int>(5);**

**node<int>\* b = a;**

La idea de redefinir este operador viene de que el programador pueda controlar qué ocurre cuando copiamos un elemento. Un ejemplo de cuándo es útil es el siguiente:

**class\_1\* a = new class\_1(parameters);**

**class\_1\* b = a;**

**a.Funtion(); //Función que modifica los parámetros dentro de a**

Si bien, hemos modificado solamente a **a**, si revisamos los atributos de b, veremos que también fueron modificados, esto pasa porque el operador **=** por defecto hace que b apunte ahora al mismo espacio en memoria que **a**.

¿Qué pasaría si el programador no quisiera esto?, si quisiera que **a** y **b** fueran independientes incluso cuando usa el operador **=**, pues podría redefinir el operador **=** para lograr su objetivo (accediendo a los valores de la clase, siguiendo la misma idea que el **Copy** Constructor).

El tercer tipo de constructor es el **move**.

La idea de este último constructor es poder pasar los valores (no las referencias) de los atributos dentro de una primera instancia de una clase a una segunda instancia de la misma clase, dejando la primera instancia totalmente vacía.

Una forma de verlo es como si estuviéramos pasando los valores de un espacio en memoria a otro espacio en memoria, borrando lo que se encontraba en el primero.

Para lograrlo se debe colocar una función con nombre igual al de la clase, recibiendo como parámetro una instancia de esa misma clase (siendo esta la instancia de la que se deben tomar los atributos para copiar, y que debe quedar vacía al final), además en el parámetro de entrada se debe colocar, delante del nombre del parámetro, un **&&**.

Sería de la forma siguiente:

**Class\_name(class\_name&& class\_instance)**

**{**

**//**Aquí copiamos los valores de los atributos de **class\_instance** a los atributos de

**//** nuestra clase

**//**Luego borramos los valores en los atributos de **class\_instance**

**}**

Para ser más específicos, mostraremos un ejemplo a partir de la clase **node**:

**node<T>(node<T>&& other\_node)**

**{**

**this.value = other\_node.value;**

**this.previous = other\_node.previous;**

**this.next = other\_node.next;**

**other\_node.value = NULL;**

**other\_node.previous = nullptr;**

**other\_node.next = nullptr;**

**}**

Podemos notar que es un procedimiento sintácticamente parecido al del **Copy Constructor**, solo que esta vez no utilizamos el const puesto que sí modificaremos la instancia que se encuentra en el parámetro de entrada.

Para hacer uso de este constructor, se utiliza el comando **std::move**, de la siguiente forma:

**class\_1\* a = new class\_1(parameters);**

**class\_1\* b = std::move(a);**

b)

En los lenguajes de programación se conoce como una expresión a una serie secuencial de operadores y operandos que especifican una acción que debe ser realizada por la computadora.

A partir de **C++11** las expresiones fueron divididas en distintas categorías, para ser exactos, estas categorías son **glvalue** (generalized left value), **xvalue** (expiring value), **lvalue** (left value), **prvalue** (pure right value) y **rvalue** (right value).

En este proyecto es de interés analizar dos de ellos, los **rvalue** y los **lvalue**.

Los **lvalue** pueden ser considerados aquellas entidades que tienen un identificador, como pueden ser las variables que representan un espacio en memoria. De forma coloquial podemos decir que, en una expresión, es aquello que se encuentra en la parte izquierda (left). Es posible obtener su dirección en memoria a partir del uso de **&**.

Los **rvalue** pueden ser considerados aquellos resultados de operadores, funciones, etc., que representen un valor, no una referencia. De forma coloquial podemos decir que, en una expresión, es en muchos casos (no todos) aquello que se encuentra en el lado derecho (right) del **=**. No es posible obtener su dirección en memoria a partir del uso de **&** (por ejemplo, **&23** es inválido en **C++**).

Aquí ocurre algo curioso que tiene relación con el Constructor **Move** que vimos anteriormente, y que trataremos más a fondo más adelante.

Primeramente, tenemos la referencia **lvalue**, esta constituye cualquier tipo de referencia que sea declarada con un único &. Tenemos de ejemplo

**Int a = 5;** // a constituye un **lvalue**

**Int &b = a;** // b constituye una referencia **lvalue**.

Por otro lado, tenemos la referencia **rvalue**, introducida en **C++11**, y que constituye cualquier tipo de referencia declarada con doble **&&** (tal y como se hace en el Constructor **Move**, de hecho, específicamente, fueron creadas para mover un objeto de un lugar a otro en vez de copiarlo).

Es clave aclarar que una referencia **rvalue** no se puede vincular con un **lvalue**, o sea, expresiones del tipo **int &&a;** constituirían un error de compilación.

c)

Ahora con el conocimiento anterior podemos explicar mejor qué hace **std::move**.

Siendo lo contrario al **Constructor Copy**, el **Constructor Move** permite mover un objeto de un instancia a otra, convirtiendo el objeto en una referencia **rvalue** (como vimos que el parámetro de entrada del Constructor **Move** debe tener de forma obligatoria un **&&**).

Una forma parecida pero más coloquial de mostrarlo es considerar a **std::move** como una transformación, donde el objeto se convierte en una referencia **rvalue**, para luego ser trasladado a un nuevo objeto.

La principal característica es que el primer objeto no queda inutilizable, para luego poder ser reutilizado o destruido, lo cual constituye un avance en el manejo de memoria dinámico que propone **C++**.

4-)

Hacer un constructor que permita **list\_initialization** le permite a la clase poder inicializar como si fuera una lista.

Por ejemplo, hacer:

**Class\_name class\_instance = {value\_1, value\_2, …};**

Para ellos se usa un constructor de la siguiente forma:

**Class\_name(initializer\_list<T> in\_list)**

Donde se inicializa la clase con el constructor a partir de los elementos en **in\_list**, como lo defina el programador.

Un ejemplo más exacto podemos darlo a partir de lo utilizado en la clase **linked\_list**:

**linked\_list(initializer\_list<T> in\_list)**

**{**

**this->count = 0;**

**this->first = nullptr;**

**this->last = nullptr;**

**for\_each(in\_list.begin(), in\_list.end(), [this](T i)**

**{this->Add\_Last(i);});**

}

Aquí lo que hacemos es inicializar los atributos de la clase, utilizamos el **for\_each**, que recibe donde comenzamos a iterar, donde terminamos y una función que le será aplicada a los elementos dentro de la lista, para guardar los elementos (lo cual explicaremos cómo funciona con detalle más adelante).

a)

A partir de lo ya visto en cuanto a constructores, ya sabemos que dos vías para inicializar nuestra clase linked\_list pueden ser:

**linked\_list<int> linked = linked\_list<int>();**

**linked\_list<int> linked\_2 = { 1,2,3,4 };**

Donde **linked** fue inicializado con el constructor clásico que tiene definido, mientras que **linked\_2** fue inicializado a partir del constructor de **list\_inicialization**, de esta forma dos tipos de constructores pudieron ser realizados usando () y {}, y serán útiles dependiendo de cada situación.

Una ventaja que otorga el utilizar el constructor de **list\_inicialization** es poder, además, crear la **linked\_list** a partir de otras listas, lo cual resulta útil en el uso de métodos.

También, algo en lo que se diferencian los **( )** y el **{ }** (y que a nuestra opinión hace superior al **{}**), es en las conversiones que son realizadas. Las operaciones con **()** realizan la conversión de tipos cuando son utilizados sin avisarle al programador (lo cuál puede ser problemático porque no da indicios de qué está ocurriendo, pudiendo llevar a fallos o funcionamientos inesperados), por otro lado **{ }** da un aviso, el compilador nos advierte de las conversiones que son realizadas.

5-)

La clase **vector<T>** almacena dentro de sí una colección de elementos de tipo **T**.

Nuestra implementación para que a partir de un **vector** pudiéramos construir nuestra clase **linked\_list** fue idéntica a la forma de proceder con **list\_inicialization**, siendo el código de dicho constructor:

**linked\_list(vector<T> in\_vector)**

**{**

**this->count = 0;**

**this->first = nullptr;**

**this->last = nullptr;**

**for\_each(in\_vector.begin(), in\_vector.end(), [this](T i) {this->Add\_Last(i); }); }**

La razón de que esto fuera posible yace en que construir un **list\_initialization** y un **vector** se hace de manera idéntica, con **{ }**.

a)

Las llamadas expresiones **Lambda** son utilizadas en **C++** (y en otros lenguajes como **C#**) para definir funciones anónimas que puedan ser utilizadas en una pequeña sección de código. Estas no pueden ser reutilizadas ni llamadas de forma externa por alguna otra función que no sea aquella en la que se encuentran declaradas una vez han cumplido su objetivo.

La expresión lambda, al crearse, se le definen los parámetros de entrada, aquellas variables que se quiere que sea usada (que hayan sido definidas previamente), y la función que será realizada con las variables anteriores.

Sería de la siguiente forma:

**[parameters\_1](parameters\_2)**

**{**

**//Cosas que realiza la función con los parámetros.**

**}**

Donde **parameters\_2** son las variables de entrada de la función, y **parameters\_1** son las variables locales que se desea que puedan ser usadas por la función.

Un ejemplo más concreto (además del que vimos en los dos constructores anteriores), podría ser:

**int main()**

**{**

**int a = 3;**

**int b = 4;**

**int c = 5;**

**auto func = [a](int number) {return number + a; };**

**cout << func(b); //Imprime 7**

**cout << "\n";**

**cout << func(c); //Imprime 8**

**return 0;**

**}**

Una cosa que es válida repetir, en este caso, **func** solo puede ser usada dentro de la función en la que está siendo declarada (a menos que sea devuelto por la función por supuesto), que en este caso es **main**.

Ahora, en el caso del **for\_each**, este permite iterar sobre una clase que represente una colección de elemento (dígase las clases **vector** o **initialization\_list** por ejemplo).

Para utilizarlo se requiere que sea incluida la biblioteca **<algorithm>**, entonces, recibe tres parámetros de entrada, el **begin()** del elemento que queremos iterar, el end() del elemento que queremos iterar, y una función que será aplicada con los elementos que se obtengan en cada iteración. Si bien existen varias formas de declarar esta función, las expresiones lambda permiten una forma cómoda de hacerlo (tal y como lo usamos en los constructores a partir de **initialization\_list** y **vector** en la página anterior).

Aunque no es objetivo del proyecto, mencionaremos que **begin()** y **end()** son funciones que debe implementar un iterador para poder ser iterado (valga la redundancia) por un **for\_each**. Estos devuelven un **const type\_of\_elements\_of\_iterator \***.

6-)

a)

En esta sección daremos respuesta al inciso a primero, puesto que lo consideramos conveniente, o sea explicaremos primero por qué es necesario utilizar un destructor para luego explicar el cómo se usa.

En **C++**, es trabajo del programador la gestión del espacio en memoria.

Cada vez que declaramos un puntero clásico (o sea, los anteriores a los punteros inteligentes), y le asignamos un espacio en memoria al que apuntar, tenemos la capacidad de gestionar lo que hay en ese espacio en memoria.

Por ejemplo, supongamos que tenemos una clase **A** cuya única función es devolvernos el valor de **pi**:

**class A**

**{**

**double pi;**

**public:**

**A()**

**{**

**this->pi = 3.14;**

**}**

**int Pi()**

**{**

**return this->pi;**

**}**

**};**

Ahora supongamos que la utilizamos dentro de una función:

**void func()**

**{**

**A\* b = new A();**

**cout << b->Pi();**

**}**

Logramos nuestro cometido de imprimir el valor de **pi** a partir de la función existente en **A**, y ha terminado la función **func**, sin embargo, ¿qué pasó con el espacio que ocupaba **b**?, pues a pesar de que **func** ya terminó, este espacio continúa ocupado en memoria innecesariamente, y nos referimos de esta forma, porque no podemos modificarlo ni acceder a él (no tenemos un puntero que nos diga donde se encuentra en este momento).

¿Cómo podemos borrar entonces la información a la que apunta **b** si no será más utilizada, y sin embargo no queremos que ocupe espacio innecesariamente?, pues esto se logra con un destructor.

El destructor es una función especial sencilla de declarar para cualquier clase.

Se declaran colocando **~** delante del nombre de la clase, sin valor de retorno,

**~Class\_name()**

**{**

//Cosas que se desea que haga el destructor

**}**

De esta forma, es posible, a partir de la palabra clave **delete**, y una instancia de una clase que posee un destructor, eliminar dicho espacio en memoria.

Una cosa curiosa que debemos mencionar es que en **C++** la mayor parte de las veces el destructor por defecto que proporciona el lenguaje es lo suficientemente bueno como para no necesitar declarar uno diferente (no es en la totalidad de los casos, dependiendo del uso que se le quiera dar a la clase, puede que sea necesario declarar un destructor que trabaje diferente).

Se puede declarar explícitamente de esta manera (de hecho, fue la que usamos en el proyecto):

**~Class\_name() = default;**

b)

Los llamados **Raw Pointer** (Puntero Crudo) no son más que los punteros clásicos que hemos estado describiendo hasta ahora, sin embargo, por muy atractivos que parezcan los punteros inteligentes, existen momentos donde es mejor usar los clásicos.

Algunos de estas situaciones son:

-Cuando se está realizando un proceso donde el tiempo es muy clave, al punto de necesitar toda la optimización de tiempo posible. Los Punteros Inteligentes si bien administran por sí solo el cuándo borrar el espacio en memoria al que apuntan, utilizan más recursos que un Puntero Crudo, por tanto, son ligeramente más lentos. Este no es un ejemplo común, pero es necesario conocerlo.

-Encapsular funciones. Es posible encapsular funciones con los Punteros Crudos cambiando un poco la declaración de la función, esto se hace escribiendo junto al nombre de la función un \*, y encapsulándolo todo entre paréntesis (para no declarar erróneamente que estamos devolviendo un puntero del tipo que devuelve la función). Se haría de la siguiente forma:

**type (\* funtion\_name)(parameters)**.

-En el diseño de bibliotecas para otros programadores, puesto que, si no tienen el código fuente, no pueden saber qué recursos están siendo utilizados dentro, por tanto, no queda claro qué tipo de puntero te están devolviendo, ni cuándo pueden perder su referencia.

7-)

Las funciones **length**, **Add\_Last**, **Remove\_Last**, **At**, **Remove\_At** están dentro del **cpp** adjunto, y fueron mencionados al inicio del documento.

a)

Cuando declaramos una función, esta en su definición puede tener que lance excepciones en casos específicos.

Tomemos como ejemplo nuestra **linked\_list**, si se hiciera **Remove\_At(size\_of\_list + 1)**, de la forma en la que lo implementamos devolvería **NULL**.

Sin embargo, es posible declararlo para que en caso de que el **index** no esté dentro del rango de la cantidad de objetos de la lista (o sea negativo, por ejemplo), lance una excepción al programador (en **C#** por ejemplo, **index out of range**) en tiempo de ejecución.

¿Entonces cómo podemos saber cuándo una función lanzará una excepción?, pues la respuesta a esto es **noexcept**.

**noexcept** es un operador unario que devuelve true si la función no lanzará una excepción, y false si al función potencialmente puede lanzarla o si no tiene declarado inicialmente un **noexcept**.

**noexcept** se usa entonces en dos situaciones:

-Para indicar si una función puede lanzar una excepción. Para esto se debe colocar luego de la declaración de la función en cuestión (antes de los { } del cuerpo de la función y después del ( ) de los parámetros de entrada), e incluso se puede directamente colocar noexcept(true) o noexcept(false).

-Para conocer si una función puede lanzar una excepción. Para esto utilizamos noexcept como una función, o sea, noexcept(function\_name(parameters…)), devolviendo esta función true o false.

En nuestro caso, colocamos noexcept(true) en las funciones Remove\_At y Remove\_Last, así los programadores que las utilicen puedan saber que no lanzan excepciones.

b)

C++11 trajo consigo una herramienta muy poderosa al lenguaje, la inferencia de tipos.

Esto se logra mediante varias palabras claves agregadas al lenguaje, la primera es **auto**, que permite declarar variables de las que no se conoce su tipo.

Un ejemplo de su utilización puede ser:

**auto a = Func();**

Donde en este caso auto está siendo igualado al resultado de una función al que no se tiene claro qué devuelve (o puede que no devuelva siempre un mismo tipo).

Un ejemplo donde es declarado frecuentemente es en los **for\_each** y en el **for** basado en rango, así como en la declaración de funciones genéricas.

Otra palabra clave que permite el uso de genericidad es **decltype**.

Para utilizar **decltype** lo declaramos como un tipo de una variable, pero antes del nombre de la misma debemos utilizar los **( )**, donde dentro pondremos una variable, y **decltype** adoptará el tipo de dicha variable.

Un ejemplo de su uso sería:

**int f = 5;**

**decltype(f) g = 6;**

Una situación donde podría resultar útil esto sería con clases genéricas, supongamos, por ejemplo, con la clase node, podríamos hacer:

**int a = 5;**

**node<decltype(a)> new\_node;**

Donde **node** recibe un **decltype**, siendo equivalente a un **node<int>**, y es útil con respecto a funciones que devuelvan variables cuyo tipo es dudoso o desconocido para el programador.

Estas dos palabras claves pueden combinarse para crear una nueva herramienta, en este caso, sería usar **decltype** con argumento **auto**, de esta forma, podríamos obtener una variable de la que no sabemos su tipo, y colocarlo en un **decltype**.

Por ejemplo, es posible hacer cosas como:

**auto a = Func(); //Func devuelve un entero**

**decltype(auto) b = a;**

**b++; //Ahora b puede trabajar como un entero**

Donde **decltype** toma con qué tipo debe trabajar a través de **auto**.

8-)

En C++ existe una forma de declarar también parámetros genéricos para funciones específicas, tal y como hicimos con las clases en el caso de **node** y **linked\_list**.

Esto se hace utilizando antes de la función el comando:

**template<typename A, typename B, etc>**

Un ejemplo de su uso sería:

**template<typename T>**

**T Func(T a)**

**{**

**return a;**

**}**

Donde **Func** recibe un valor de tipo T.

Pero también contamos con una herramienta más poderosa, que nos permite trabajar un nuevo nivel de genericidad en funciones, poner una cantidad indefinida de argumentos.

Para utilizar una cantidad variable de argumentos de entrada en una función, solo tenemos que colocar **template<typename T…>**.Esto también es posible hacerlo con las clases.

Ejemplos de su uso en clases y funciones son:

Para las funciones:

**template<typename...T>**

**void Func(T... a)**

**{**

**//Cuerpo de la función**

**};**

Lo cual permite hacer cosas como:

**Func(1, 2, 3, 4, 5, 5, 'Living', new node<int>(5), new linked\_list<int>())**

Para las clases se haría de forma parecida:

**template<typename...T>**

**class node**

{

**//Cuerpo de la clase**

}

Lo cual permite hacer cosas como:

**node<int, int, char, linked\_list, int> a = new node<int, int, char, linked\_list, int>(parameters);**

Anteriormente habíamos mencionado, cuando explicábamos algunos de los momentos donde es preferible o necesario usar un **Raw Pointer** antes que un **Smart Pointer**, que es posible realizar punteros a funciones haciendo **(\*Funtion\_Name)** en la definición de la misma.

Tomemos el siguiente ejemplo:

**int Funtion()**

**{**

**return 13;**

**}**

**int main()**

**{**

**int (\*Funtion\_Pointer)();**

**Funtion\_Pointer = Funtion;**

**cout << Funtion\_Pointer(); //Imprime 13**

**}**

Donde hicimos que **Funtion\_Pointer** apuntara a **Funtion**, lo cual, si bien en este ejemplo no tiene mucha utilidad, puede ser explotado al pasar funciones por argumentos (funciones que pueden ser definidas por el programador), por ejemplo, el pasar por argumento a una función de **Sort** el criterio de **<**, **>**, o **=**, a conveniencia de quien lo use.

Tomando la función **Funtion** que requiere este ejercicio, se implementaría de la siguiente forma:

**template<typename R, typename... T>**

**R(\*Funtion)(T...);**

Donde esta es una función que recibe una serie de parámetros de cualquier tipo, y devuelve un valor de tipo **R**.

a)

La función **Map** es conocida en lenguajes como **Haskell** (pudiendo ser usada por los programadores novatos incluso sin tener que conocer todo lo necesario para implementarla).

La idea es recibir una serie de elementos de un tipo **A**, y un transformador de un tipo **A** a un tipo **B**, y devolver una serie de elementos de tipo **B**, resultante de aplica el transformador a cada elemento de **A**.

Un ejemplo sencillo no genérico de una función que transforma un **array** de **double** a un puntero de **int** con el concepto anterior podría ser:

**int Transformation\_Double\_To\_Int(double a)**

**{**

**return int(a);**

**}**

**int\* Transformation\_List(double in\_list[])**

**{**

**int out\_list[sizeof(in\_list)];**

**for (int i = 0; i < sizeof(in\_list); i++)**

**{**

**out\_list[i] = Transformation\_Double\_To\_Int(in\_list[i]);**

**}**

**return out\_list;**

**}**

**int main()**

**{**

**double a[] = { 1.2, 2.5, 3.9, 45674 };**

**int\* b = Transformation\_List(a); //Queda guardado el equivalente a**

**//{1,2,3,4}**

**}**

**Map** utiliza la idea general de la función anterior, pero de forma genérica, en este caso recibiendo la función de transformación **Funtion**. Quedaría entonces de la siguiente forma:

**template<typename R, typename T>**

**linked\_list<R> Map(Function<R, T> Transformation\_Funtion)**

**{**

**linked\_list<R> out\_list = linked\_list();**

**for (node<T> i = this->first; i != nullptr; i = i->next)**

**{**

**out\_list.Add\_Last(Transformation\_Funtion(i->Get\_Value()));**

**}**

**return out\_list;**

**}**

Donde recibimos la función de transformación, y devolvemos una instancia de **linked\_list<R>**.

b)

En **C++** realizar un alias al puntero de una función es sencillo, usando al palabra clave **using** (tal y como con las clases), de la siguiente forma:

**using New\_Name = Return\_Type (\*)(parameters);**

c)

Utilizando lo visto en el anterior inciso, podemos fácilmente aplicarlo a **Funtion**:

**template<typename R, typename... T>**

**using Alias\_Function = R (\*)(T...);**

Esperamos el proyecto haya sido de su agrado. Muchas Gracias.