Lenguajes de Programación

Seminario#1
C++11, C++14.

Equipo 2:
Marcos Manuel Tirador del Riego
Laura Victoria Riera Pérez
Leandro Rodríguez Llosa

Grupo: C-311

Índice

1. Definición de las clases genéricas linked_list y node

Listing 1: Sample Code Listing C++

```
#include <iostream>

int main()
{
  std::cout << "Hello World";
  return 0;
}
```

Una lista enlazada es una estructura de datos compuesta de nodos, donde cada nodo contiene alguna información y un puntero a otro nodo de la lista. Si un nodo tiene sólo un enlace con su sucesor en esta secuencia, la lista se denomina lista enlazada simple.

2. Definición de miembros de datos necesarios de ambas clases

2.1. Nuevos elementos introducidos a partir de C++11 que permiten un manejo más ïnteligente"de la memoria

Se introduce el principio "la adquisición de recursos es inicialización" o *RAII*, con lo cual se reserva o libera espacio en memoria.

El C++ moderno evita usar la memoria del heap tanto como sea posible declarando objetos en la pila. Cuando un recurso es demasiado grande para la pila, debe ser propiedad de un objeto. A medida que el objeto se inicializa, adquiere el recurso que posee. El objeto es entonces responsable de liberar el recurso en su destructor. El propio objeto propietario se declara en la pila.

Cuando un objeto de la pila propietario de recursos queda fuera del alcance, su destructor se invoca automáticamente. De esta manera, la recolección de basura en C++ está estrechamente relacionada con la vida útil del objeto y es determinista, a diferencia de los recolectores de basura utilizados por otros lenguajes. Un recurso siempre se libera en un punto conocido del programa, que puede controlar. Solo los destructores deterministas como los de C++ pueden manejar los recursos de memoria y los que no son de memoria por igual.

Los elementos introducidos con este fin son los punteros inteligentes, quienes apuntan a objetos y, cuando el puntero sale del alcance (scope), el objeto se destruye. Esto los hace inteligentes en el sentido de que no tenemos que preocuparnos por la liberación manual de la memoria asignada. Los punteros inteligentes hacen todo el trabajo pesado por nosotros.

2.2. Inicialización

En C++11 hay cuatro tipos de punteros inteligentes:

- `std::auto_ptr`: es un remanente en desuso de C++98. Fue un intento de estandarizar lo que más tarde se convirtió en std::unique_ptr de C++11. Hacer el trabajo bien requiere move semantics, pero C++98 no las tenía. Como solución alternativa, std::auto_ptr cooptó sus operaciones de copia para movimientos. Esto condujo a un código sorprendente (copiar un std::auto_ptr lo establece en nulo) y restricciones de uso frustrantes (por ejemplo, no es posible almacenar std::auto_ptrs en contenedores).

- `std::unique_ptr`: encarna la semántica de propiedad exclusiva.

  Un estándar no nulo `std::unique_ptr` siempre posee a lo que apunta.

  Mover un `std::unique_ptr` transfiere la propiedad del objeto desde el puntero de origen hasta el puntero de destino (el puntero de origen es establecido en nulo). No está permitido copiar un `std::unique_ptr`, porque si se pudiera, terminaría con dos `std::unique_ptr` para el mismo recurso, cada uno pensando que lo poseía (y por lo tanto debería destruirlo). `std::unique_ptr` es, por lo tanto, un tipo de solo movimiento.

  Tras la destrucción, un `std::unique_ptr` no nulo destruye su recurso. Por defecto, la destrucción de recursos es realizada aplicando `delete` al raw pointer dentro de `std::unique_ptr`, pero se puede especificar su forma de destrucción.

  Es razonable asumir que, por defecto, los `std::unique_ptr` poseen el mismo tamaño que los raw pointers, y para la mayoría de las operaciones (incluida la desreferenciación), ejecutan exactamente las mismas instrucciones. Esto significa que pueden ser usados incluso en situaciones donde la memoria y los ciclos son apretados.

- `std::shared_ptr`: Un objeto al que se accede a través de `std::shared_ptr` tiene su vida útil administrada por esos punteros a través de propiedad compartida.

  Ningún `std::shared_ptr` específico posee al objeto. En cambio, todo `std::shared_ptr` apuntando a este colabora para asegurar su destrucción en el punto donde ya no se necesite. Cuando el último `std::shared_ptr` que apunta a un objeto deja de apuntar allí (por ejemplo, porque el `std::shared_ptr` se destruye o apunta a un objeto diferente), este destruye el objeto al que apunta.

  Como con la recolección de basura, los programadores no necesitan preocuparse por administrar la vida tiempo de los objetos referenciados, pero como con los destructores, el momento de la destrucción de los objetos es determinista.

  Un `std::shared_ptr` puede decir si es el último que apunta a un recurso consultando el conteo de referencia del mismo (valor asociado con el objeto que mantiene seguimiento de cuántos `std::shared_ptr` apuntan a él). Los constructores de `std::shared_ptr` incrementan este conteo, los destructores lo disminuyen y los operadores de asignación de copia hacen ambas cosas. Si un `std::shared_ptr` ve un conteo de referencia con valor cero después de realizar una disminución, no hay ningún otro `std::shared_ptr` apuntando al recurso, por lo que lo destruye.

  En comparación con `std::unique_ptr`, los objetos `std::shared_ptr` suelen el doble de grande, generan gastos generales para los bloques de control y requieren manipulaciones de conteo de referencia atómicas.

  La destrucción de recursos predeterminada se realiza mediante eliminación, pero se admiten destructores personalizados.

- `std::weak_ptr`: actúa como un `std::shared_ptr`, pero no participa en la propiedad compartida del recurso apuntado, y, por lo tanto, no afecta el conteo de referencias del mismo. En realidad hay un segundo conteo de referencia en el bloque de control, y es este el que `std::weak_ptr` manipula.

  Este tipo de puntero inteligente tiene en cuenta un problema desconocido para `std::shared_ptr`: la posibilidad de que a lo que apunta haya sido destruido. `std::weak_ptr` soluciona este problema rastreando cuando cuelga (*dangles*), es decir, cuándo el objeto al que se supone que apunta ya no existe.

A menudo lo que desea es comprobar si un `std::weak_ptr` ha caducado y, si no, acceder al objeto al que apunta.

Desde una perspectiva de eficiencia, los `std::weak_ptr` son iguales que los `std::shared_ptr`. Los objetos `std::weak_ptr` tienen el mismo tamaño que `std::shared_ptr` objetos, hacen uso de los mismos bloques de control que `std::shared_ptr` y operaciones como construcción, destrucción y la asignación implica manipulaciones de conteo de referencias atómicas.

Los posibles casos de uso de `std::weak_ptr` incluyen el almacenamiento en caché, las listas de observadores y la prevención de ciclos `std::shared_ptr`

Todos están diseñados para ayudar a manejar el tiempo de vida de objetos asignados dinámicamente, es decir, para evitar fugas de recursos al garantizar que tales objetos se destruyen de la manera apropiada en el momento apropiado (incluyendo en caso de excepciones).

2.3. Filosofía en el uso de la memoria defendida por C++

2.4. Simplificación de nombres de tipos mediante el uso de alias

Dado que typedef y la declaración de alias hacen exactamente lo mismo, es Sonable preguntarse si hay una razón técnica sólida para preferir uno sobre el otro. Pero existe una razón de peso: las plantillas. En particular, las declaraciones de alias pueden ser con plantillas (en cuyo caso se llaman plantillas de alias), mientras que typedefs no puede. Esto les da a los programadores de C++ 11 un mecanismo directo para expresar cosas que en C ++ 98 tuvo que ser pirateado junto con typedefs anidados dentro de la plantilla estructuras P

typedefs no admite la creación de plantillas, pero las declaraciones de alias sí. Las plantillas de alias evitan el sufijo "::type" y, en las plantillas, el "typename" prefijo a menudo requerido para referirse a typedefs. C++14 ofrece plantillas de alias para todas las transformaciones de rasgos de tipo de C++11.

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

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

Un constructor es una función miembro que tiene el mismo nombre que la clase. Para inicializar un objeto de una clase, usamos constructores. El propósito del constructor es inicializar un objeto de una clase Construye un objeto y puede establecer valores para los miembros de datos. Si una clase tiene un constructor, todos los objetos de esa clase serán inicializados por una llamada al constructor.

- Constructores clásicos:

  Un constructor sin parámetros o con parámetros predeterminados se llama predeterminado constructor. Es un constructor que se puede llamar sin argumentos:

  Otro ejemplo de un constructor predeterminado, el que tiene los argumentos predeterminados:

  Si un constructor predeterminado no está definido explícitamente en el código, el compilador generar un constructor predeterminado. Pero cuando definimos un constructor propio, el que

necesita parámetros, el constructor predeterminado se elimina y no es generado por un compilador. Los constructores se invocan cuando tiene lugar la inicialización del objeto. no pueden ser invocado directamente. Los constructores pueden tener parámetros arbitrarios; en cuyo caso podemos llamarlos usuario- constructores proporcionados:

Los constructores no tienen un tipo de devolución y su propósito es inicializar el objeto. de su clase.

- Constructor move:

  Además de copiar, también podemos mover los datos de un objeto a otro. Nosotros llamamos es una semántica de movimiento. La semántica de movimiento se logra a través de un constructor de movimiento y mover operador de asignación El objeto desde el que se movieron los datos, se deja en algún válido pero estado no especificado. La operación de movimiento es eficiente en términos de velocidad de ejecución, ya que no tienes que hacer copias.

  Move constructor acepta algo llamado referencia rvalue como argumento. Cada expresión puede encontrarse en el lado izquierdo o en el lado derecho del operador de asignación Las expresiones que se pueden usar en el lado izquierdo se llaman lvalues, como variables, llamadas a funciones, miembros de clases, etc. Las expresiones que se pueden utilizados en el lado derecho de un operador de asignación se denominan rvalues, como literales, y otras expresiones. Ahora la semántica de movimiento acepta una referencia a ese valor. La firma de un El tipo de referencia de rvalue es T&&, con símbolos de referencia doble. Para convertir algo en una referencia de valor real, usamos la función std::move. Este La función convierte el objeto en una referencia rvalue. No mueve nada.

  Si un usuario no proporciona un constructor de movimiento, el compilador proporciona implícitamente constructor de movimiento predeterminado generado.

- Sobrecargas: Los objetos de las clases se pueden utilizar en la expresión como operandos. Por ejemplo, podemos hacer el siguiendo:

  mi_objeto = otro_objeto;

  Aquí los objetos de una clase se utilizan como operandos. Para hacer eso, necesitamos sobrecargar el operadores para tipos complejos como clases. Se dice que necesitamos sobrecargarlos para proporcionar una operación significativa en los objetos de una clase. Algunos operadores pueden estar sobrecargados para clases; algunos no pueden. Podemos sobrecargar los siguientes operadores: Operadores aritméticos, operadores binarios, operadores booleanos, operadores unarios, operadores de comparación, operadores compuestos, operadores de función y subíndice:

  + ␣ - ␣ * ␣ / ␣ \% ␣ ^ ␣ \& ␣ ! = < > == ! = < = > = + = - = * = / = % = ^= & = | = < < > > > = < < = && || + + - -, - > * - > () [] |

  Cada operador lleva su firma y conjunto de reglas cuando se sobrecarga por clases.

  Algunas sobrecargas de operadores se implementan como funciones miembro, otras como funciones de ningún miembro funciones

  Un operador de asignación se implementará mediante una función miembro no estática con exactamente un parámetro. Porque un operador de asignación de copia operator= se declara implícitamente para una clase si no lo declara el usuario (12.8), una clase base El operador de asignación siempre está oculto por el operador de asignación de copia de la clase derivada. 2 Cualquier operador de asignación, incluso el operador de asignación de copia, puede ser virtual. [Nota: para una clase D derivada con una base clase B para la que se ha declarado una asignación

de copia virtual, el operador de asignación de copia en D no anula el de B operador de asignación de copia virtual.

3.2. ¿Qué es un lvalue y un rvalue ?

Cada expresión de C++ tiene un tipo y pertenece a una categoría de valor. Las categorías de valor son la base de las reglas que los compiladores deben seguir al crear, copiar y mover objetos temporales durante la evaluación de expresiones.

En C++, un lvalue es algo que apunta a una ubicación de memoria específica. Por otro lado, un rvalue es algo que no apunta a ninguna parte. En general, los valores de r son temporales y de corta duración, mientras que los valores de l tienen una vida más larga ya que existen como variables.

En pocas palabras, un lvalue es una referencia de objeto y un rvalue es un valor.

Un lvalue es una expresión que produce una referencia de objeto, como un nombre de variable, una referencia de subíndice de matriz, un puntero sin referencia o una llamada de función que devuelve una referencia. Un lvalue siempre tiene una región de almacenamiento definida, por lo que puede tomar su dirección.

Un rvalue es una expresión que no es un lvalue. Los ejemplos de valores r incluyen literales, los resultados de la mayoría de los operadores y llamadas a funciones que devuelven no referencias. Un rvalue no tiene necesariamente ningún almacenamiento asociado.

Estrictamente hablando, una función es un lvalue, pero los únicos usos que tiene son para llamar a la función o determinar la dirección de la función. La mayoría de las veces, el término lvalue significa objeto lvalue, y este libro sigue esa convención.

C++ toma prestado el término lvalue de C, donde solo se puede usar un lvalue en el lado izquierdo de una instrucción de asignación. El término rvalue es una contrapartida lógica de una expresión que solo se puede usar en el lado derecho de una tarea.

3.3. std::move

std::move se usa para indicar que un objeto t puede ser "mover desde", es decir, permitir la transferencia eficiente de recursos de t a otro objeto.

En particular, std::move produce una expresión de valor x que identifica su argumento t. Es exactamente equivalente a un static_cast a un tipo de referencia rvalue.

Las funciones que aceptan parámetros de referencia de rvalue (incluidos los constructores de movimiento, los operadores de asignación de movimiento y las funciones de miembros regulares como std::vector::push_back) se seleccionan, mediante resolución de sobrecarga, cuando se llaman con argumentos de rvalue (ya sean prvalues como un objeto temporal o xvalues como el producido por std::move). Si el argumento identifica un objeto que posee un recurso, estas sobrecargas tienen la opción, pero no son obligatorias, de mover cualquier recurso contenido por el argumento. Por ejemplo, un constructor de movimiento de una lista vinculada podría copiar el puntero al encabezado de la lista y almacenar nullptr en el argumento en lugar de asignar y copiar nodos individuales.

Los nombres de las variables de referencia de rvalue son lvalues y deben convertirse a valores x para vincularse a las sobrecargas de funciones que aceptan parámetros de referencia de rvalue, razón por la cual los constructores de movimiento y los operadores de asignación de movimiento suelen usar std::move

Una excepción es cuando el tipo del parámetro de función es una referencia de valor r a un parámetro de plantilla de tipo (referencia de reenvío.º referencia universal"), en cuyo caso se usa std::forward en su lugar.

Mover como rvalue Devuelve una referencia de valor r a arg.

Esta es una función auxiliar para forzar la semántica de movimiento en los valores, incluso si tienen un nombre: el uso directo del valor devuelto hace que arg se considere un valor r.

En general, los valores r son valores cuya dirección no se puede obtener desreferenciandolos, ya sea porque son literales o porque son de naturaleza temporal (como valores devueltos por funciones o llamadas explícitas al constructor). Al pasar un objeto a esta función se obtiene un rvalue que hace referencia a él.

Muchos componentes de la biblioteca estándar implementan la semántica de movimiento, lo que permite transferir la propiedad de los activos y las propiedades de un objeto directamente sin tener que copiarlos cuando el argumento es un valor r.

Aunque tenga en cuenta que, en la biblioteca estándar, mover implica que el objeto desde el que se movió se deja en un estado válido pero no especificado. Lo que significa que, después de tal operación, el valor del objeto movido solo debe destruirse o asignarse un nuevo valor; si se accede a él, se obtiene un valor no especificado.

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

4.1. Compare la utilización del {} v.s ()

Dependiendo de su perspectiva, opciones de sintaxis para la inicialización de objetos en C++11 encarnar una vergüenza de riquezas o un lío confuso. Como regla general, los valores de inicialización se pueden especificar con paréntesis, un signo igual o llaves:

intx(0); // el inicializador está entre paréntesis int y = 0; // el inicializador sigue a -ïntz0; // el inicializador está entre llaves

En muchos casos, también es posible usar un signo igual y llaves juntas

int z = 0 ; // el inicializador usa -ƶ llaves

Incluso con varias sintaxis de inicialización, hubo algunas situaciones en las que C++98 no tenía manera de expresar una inicialización deseada. Por ejemplo, no fue posible indicar directamente que se debe crear un contenedor STL que contenga un conjunto particular de valores (por ejemplo, 1, 3 y 5). Para abordar la confusión de múltiples sintaxis de inicialización, así como el hecho de que no cubren todos los escenarios de inicialización, C++ 11 introduce una inicialización uniforme: una sintaxis de inicialización única que puede, al menos en concepto, usarse en cualquier lugar y expresar todo. Se basa en llaves, y por eso prefiero el término arriostrado inicialización La ïnicialización uniforme.[es] una idea. La ïnicialización reforzada.[es] una sintáctica construir.

La inicialización con refuerzos le permite expresar lo que antes era inexpresable. Uso de aparatos ortopédicos, especi-

Medir el contenido inicial de un contenedor es fácil:

estándar::vector¡int¿v 1, 3, 5 ; // el contenido inicial de v es 1, 3, 5

Las llaves también se pueden usar para especificar valores de inicialización predeterminados para datos no estáticos miembros Esta capacidad, nueva en C++ 11, se comparte con la sincronización de inicialización "=". impuesto, pero no entre paréntesis:

```
class Widget {
...
private:
```

```
int x{ 0 }; // fine, x's default value is 0
int y = 0; // also fine
int z(0); // error!
};
```

Por lo tanto, es fácil entender por qué la inicialización con llaves se llama üniforme". de C++ tres maneras de designar una expresión de inicialización, solo se pueden usar llaves cada dónde. Una característica novedosa de la inicialización con llaves es que prohíbe la conversión implícita de estrechamiento. siones entre los tipos incorporados. Si el valor de una expresión en un inicializador entre llaves no es garantizado ser expresable por el tipo de objeto que se inicializa, el código no compilar:

doble x, y, z; ... int suma1 x + y + z ; // ¡error! la suma de los dobles puede // no se puede expresar como int

La inicialización con paréntesis y "=" no verifica las conversiones restringidas, porque eso podría romper demasiado código heredado:

int suma2(x + y + z); // está bien (valor de la expresión truncado a un int)

int suma3 = x + y + z; // ídem

Otra característica digna de mención de la inicialización con refuerzos es su inmunidad a los errores de C++. análisis más irritante. Un efecto secundario de la regla de C++ de que cualquier cosa que se pueda analizar como declaración debe interpretarse como una, el análisis más desconcertante aflige con mayor frecuencia desarrolladores cuando quieren construir por defecto un objeto, pero sin darse cuenta terminan declarando una función en su lugar. La raíz del problema es que si desea llamar a una con- structor con un argumento, puedes hacerlo así,

Widget w1(10); // llamar a Widget ctor con argumento 10

pero si intenta llamar a un constructor de Widget con cero argumentos usando el análogo sintaxis, declara una función en lugar de un objeto:

Widget w2(); // análisis más irritante! declara una función // llamado w2 que devuelve un Widget!

Las funciones no se pueden declarar usando llaves para la lista de parámetros, por lo que por defecto- construir un objeto usando llaves no tiene este problema:

Widget w3; // llama a Widget ctor sin argumentos

La inicialización con corchetes es la sintaxis de inicialización más utilizada, evita reduce las conversiones y es inmune al análisis más desconcertante de C++. • Durante la resolución de sobrecarga del constructor, los inicializadores reforzados se emparejan con std::initializer_list parámetros si es posible, incluso si otra construcción Los tores ofrecen coincidencias aparentemente mejores. • Un ejemplo de donde la elección entre paréntesis y llaves puede hacer un diferencia significativa es crear un std::vector¡tipo numérico¿con dos argumentos • Elegir entre paréntesis y llaves para la creación de objetos dentro de plantillas puede ser desafiante.

5. Definición de un constructor que reciba un vector $< T >$

5.1. Usar for_each con expresiones lambda

6. Definición del destructor de la clase

Como vimos anteriormente, un constructor es una función miembro que se invoca cuando el objeto se inicializa. De manera similar, un destructor es una función miembro que se invoca cuando un el objeto es destruido. El nombre del destructor es tilde seguido de un nombre de clase

Destructor no toma parámetros, y hay un destructor por clase.

6.1. ¿Hace falta?

Los destructores se llaman cuando un objeto sale del alcance o cuando un puntero a un se elimina el objeto. No debemos llamar al destructor directamente.

Una vez que se invoca un destructor para un objeto, el objeto ya no existe; el comportamiento es indefinido si el destructor es invocado para un objeto cuya vida ha terminado (3.8). [Ejemplo: si el destructor de un objeto automático está explícitamente invocado, y el bloque se deja posteriormente de una manera que normalmente invocaría la destrucción implícita del objeto, el comportamiento es indefinido. — fin del ejemplo]

!!!!CAMBIAR

Podemos decir que hace falta un destructor, pues este realiza procesos necesarios cuando un objeto termine de ser útil, liberando la memoria dinámica utilizada por dicho objeto o liberando recursos usados, como por ejemplo ficheros o dispositivos. También son necesarios en situaciones donde se quiere compartir pertenencia sobre los objetos, y se desea ahorrar memoria. Si de lo contrario el usuario no define un destructor, el compilador provee uno por defecto y muchas veces esto es suficiente, pero sí que es necesario definir un destructor personalizado cuando la clase tiene miembros de tipo puntero, que apuntan a recursos del sistema que necesitan ser liberados, o que tienen la propiedad del objeto al que apuntan

6.2. ¿Para qué casos haría falta un raw pointer?

Un puntero es un tipo de variable. Almacena la dirección de un objeto en la memoria y se utiliza para acceder a ese objeto. Un raw pointer es un puntero cuya vida útil no está controlada por un objeto encapsulador, como un puntero inteligente. A un puntero sin procesar se le puede asignar la dirección de otra variable que no sea un puntero, o se le puede asignar un valor de nullptr. Un puntero al que no se le ha asignado un valor contiene datos aleatorios.

También se puede desreferenciar un puntero para recuperar el valor del objeto al que apunta. El operador de acceso a miembros proporciona acceso a los miembros de un objeto.

En C++ moderno, los raw pointers solo se usan en pequeños bloques de código de alcance limitado, bucles o funciones auxiliares donde el rendimiento es crítico y no hay posibilidad de confusión sobre la propiedad.

7. Definición de las funciones *length*, *Add_Last* , *Remove_Last*, *At*, *Remove_At*

7.1. Noexcept

Durante el trabajo en C++11, surgió el consenso de que la información verdaderamente significativa sobre el comportamiento de emisión de excepciones de una función era si tenía alguna. negro o blanco, una función podría emitir una excepción o garantizaría que no lo haría. Esta dicotomía de tal vez o nunca forma la base de las especificaciones de excepción de C++11, que esencialmente reemplazan a C++98. (Las especificaciones de excepción de estilo C++98 permanecen válidos, pero están en desuso). En C++ 11, el noexcept incondicional es para funciones que garantizan que no emitirán excepciones.

Si una función debe declararse así es una cuestión de diseño de interfaz. los el comportamiento de emisión de excepciones de una función es de interés clave para los clientes. Las personas que llaman pueden consultar el estado noexcept de una función, y los resultados de dicha consulta pueden afectar el excepción seguridad o eficacia del código de llamada. Como tal, si una función es noexcept es una pieza de información tan importante como si una función miembro es constante No declarar una función no, excepto cuando sabe que no emitirá un la excepción es simplemente una mala especificación de la interfaz.

Pero hay un incentivo adicional para aplicar noexcept a funciones que no duce excepciones: permite a los compiladores generar un mejor código objeto. Comprender por qué, es útil examinar la diferencia entre las formas C++98 y C++11 de decir que una función no emitirá excepciones. Considere una función f que promete a los llamadores nunca recibirán una excepción.

noexcept es parte de la interfaz de una función, y eso significa que las personas que llaman pueden depende de ello.

Las funciones noexcept son más optimizables que las funciones que no son noexcept.

noexcept es particularmente valioso para las operaciones de movimiento, intercambio, memoria funciones de desasignación y destructores.

La mayoría de las funciones son neutrales a las excepciones en lugar de no excepto.

7.2. Inferencia de tipo en C++ (auto, decltype decltype(auto)). Explicar todos, pero no obligatoriamente usarlos.

- auto: Deduce el tipo de una variable declarada a partir de su expresión de inicialización.
- decltype: produce el tipo de una expresión especificada.
- decltype(auto): se utiliza para declarar una función de plantilla cuyo tipo de valor devuelto dependa de los tipos de sus argumentos de plantilla, o bien para declarar una función de plantilla que contenga una llamada a otra función y devuelva el tipo de valor devuelto de la función contenida.

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 .

8.1. 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.

8.2. Crear punteros a funciones usando alias

8.3. Crear un puntero a función Function que permita cualquier cantidad de parámetros de cualquier tipo.