*SEMINARIO 2*

*C++*

***ALEJANDRO CAMPOS***

***DARIAN DOMINGUEZ***

***DANIEL DOMINGUEZ***

***ABEL CRUZ***

***YADIEL FELIPE***

***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.***

***a.***

Con C++11 se introducen los punteros inteligentes que se usan para administrar automáticamente el alcance de memoria asignada dinámicamente. Los punteros inteligentes se prefieren a los punteros "sin procesar" en la mayoría de los casos. Una de sus características es que hacen explícita la semántica de propiedad de la memoria asignada dinámicamente, comunicando en sus nombres si se pretende que un objeto sea compartido o de propiedad única.

***std :: shared\_ptr :***

* La plantilla de clase std::shared\_ptr define un puntero compartido que puede compartir la propiedad de un objeto con otros punteros compartidos.
* El comportamiento de compartir se implementa a través de una técnica conocida como conteo de referencias, donde el número de punteros compartidos que apuntan al objeto se almacena a su lado. Cuando este conteo llega a cero, ya sea a través de la destrucción o la reasignación de la última instancia de std::shared\_ptr , el objeto se destruye automáticamente.
* Siempre que sea posible se debe usar la función “make\_share” cuando se cree el recurso por primera vez en la memoria. Ya que se utiliza el mismo llamado para asignar memoria al bloque de control y al recurso. Reduciendo la sobrecarga de la construcción.
* De forma predeterminada, shared\_ptr incrementa el recuento de referencia y no transfiere la propiedad. Sin embargo, se puede hacer para transferir la propiedad usando **std::move**

***std :: weak\_ptr :***

* Las instancias de std::weak\_ptr pueden apuntar a objetos que son propiedad de instancias de std::shared\_ptr y solo se convierten en propietarios temporales.
* Los punteros débiles no alteran el recuento de referencias del objeto y, por lo tanto, no impiden la eliminación de un objeto si todos los punteros compartidos del objeto se reasignan o destruyen
* Dado que std::weak\_ptr no mantiene vivo su objeto referenciado, no es posible el acceso directo de datos a través de std::weak\_ptr . En su lugar, proporciona una función miembro **lock()** que intenta recuperar un std::shared\_ptr para el objeto al que se hace referencia.
* Es muy eficiente a usar cuando existen referencias cíclicas(no son convenientes) entre shared\_ptr. Ya que el contador de refernecias no puede hacerse cero, produciendo una fuga de memoria. Para ello uno de los punteros en el ciclo puede ser weak\_ptr.

***std :: unique\_ptr :***

* Es std::unique\_ptr una plantilla de clase que administra la vida útil de un objeto almacenado dinámicamente. A diferencia de std::shared\_ptr , el objeto dinámico es propiedad de solo instanciade std::unique\_ptr en cualquier momento.
* Cuando un puntero std::unique\_ptr que posee un objeto queda fuera del alcance, el objeto propio se elimina.
* Se Puede transferir la propiedad del contenido de un puntero inteligente std::unique\_ptr a otro puntero utilizando **std::move**, lo que hará que el puntero inteligente original apunte a nullptr.
* std::unique\_ptr no comparte su puntero. No se puede copiar a otro unique\_ptr, pasar por valor a una función o usar en algoritmos por copia. A este tipo de puntero solo se puede mover.

***c.***

C ++ no es un lenguaje gestionado por la memoria. La memoria asignada dinámicamente (es decir, los objetos creados con new ) se "filtrará" si no se desasigna explícitamente (con delete ). Es responsabilidad del programador asegurarse de que la memoria asignada dinámicamente se libere antes de descartar el último puntero a ese objeto.

Es con C++11 que a partir del uso de punteros inteligentes, al trabajar con ellos, nos podemos abstraer de liberar la memoria utilizada por los recursos que apuntan. Pero sigue siendo nuestro trabajo liberar la memoria de clases que no manejen punteros.

Sin lugar a dudas cuando hablamos de manejo de memoria en C++ no podemos dejar de mencionar los destructores de clases. Encargados de liberar de forma automática los recursos computacionales que el objeto de dicha clase haya adquirido durante su creación (memoria dinámica o liberando recursos usados, como ficheros, dispositivos, etc.). No es más que una función que se garantiza que será invocada de forma automática una vez que el objeto es destruido. Los destructores son llamados a realizar su trabajo una vez que el objeto salga de su scope.

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

* Definir los constructores clásicos de C++ (C++0x, el constructor move y las sobrecargas  
  del operador =.
* Son unos métodos especiales que se ejecutan automáticamente al crear un objeto de la clase.
* En su declaración no se especifica el tipo de dato que devuelven, y poseen el mismo nombre que la clase a la que pertenecen.
* Al igual que otros métodos, puede haber varios constructores sobrecargados, aunque no pueden existir constructores virtuales.
* Si no existe ningún constructor en la clase el compilador proporciona el constructor por defecto

***Constructor por defecto***

* Es el constructor que no recibe ningún parámetro en la función. Si no se definiera ningún constructor, el sistema proporcionaría uno predeterminado.
* La inicialización de las variables se hace con valores por defecto.

***Constructor de copia***

* Es un constructor que tiene como parámetro de entrada una referencia a otro objeto de la misma clase.
* Las variables del objeto que se está creando se inicializan con los valores del objeto a copiar.
* Al igual que el predeterminado, si no se define, el sistema proporciona uno, el cual realiza una copia bit a bit entre los objetos.
* Es utilizado cuando una función recibe el objeto como parámetro por valor.

***Constructor por movimiento***

* Este constructor vió a luz con el estándar c++11, al igual que el operador de asignación move.
* Tiene como objetivo evitar la copia innecesaria de grandes cantidades de información. Esto lo consigue, básicamente, con el movimiento de los punteros, en vez de mover la información en los mismos.
* Un efecto secundario es que el objeto original queda inservible después de la operación de mover. O sea se debe dejar el objeto del que estamos moviendo en un estado valido pero no especificado, ya que eventualmente será destruido.
* La idea es manejar los lvalues separadamente de los rvalues. Los constructores move toman los rvalues.
* Se utiliza en el retorno de funciones.

***¿Qué es un lvalue y un rvalue?***

* Un lvalue es un valor que reside en la memoria(heap o pila) y se puede tomar su dirección. Se puede ver como algo que apunta a una localización especifica de la memoria. Los lvalues vienen siendo aquellos objetos que tienen nombre. Toda variable incluyendo las no modificables son lvalues.
* Un rvalue reside en la parte derecha de una expresión de asignación, por ejemplo los literales(excepto las cadenas que son arrays), el resultado de cualquier operador aritmético o lógico y el retorno de una función que no devuelva una referencia. Se pudiera decir que no apuntan a nada y por lo general son temporales y de corta vida.
* Los lvalues pueden realizar asignaciones mediante ”=”, se puede acceder a su dirección de memoria con ”&” y usarlos para inicializar una referencia, cosa que con un rvalue no sucede.
* Podemos ver a los lvalues como contenedores y los rvalues como objetos contenidos en contenedores.

***Referencias lvalue***

* Es una referencia que puede ser inicializada con un lvalue, no con un rvalue.
* Para que la referencia este asociada a un objeto, esta debe ser inicializada.

***Referencias rvalue***

* Es introducida en c++11 y se identifica por T&&. Esta refiere a temporales que son modificables luego de su inicialización.
* Las referencias R-Value también conocidas como R-Referencias son idénticas a las referencias normales, pero se declaran con && en lugar de &.
* 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.

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

* En c++11 se permite la utilización de constructores por movimiento en contrapartida a los constructores por copia y además de los operadores de asignación por copia se permiten por movimiento también.
* El constructor por movimiento es usado en vez del constructor por copia si el objeto es una referencia rvlaue(&&).
* Std::move() es un casteo que produce una referencia rvalue un objeto.

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

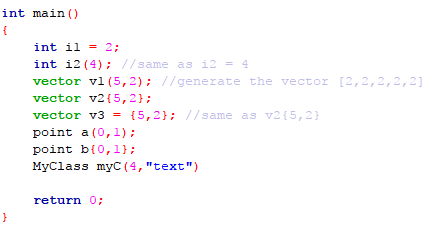
***posible.***

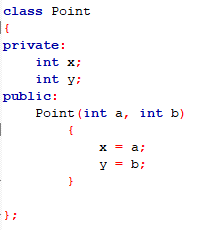
1. ***Compare la utilización del {} v.s ().***

***Inicializador:***

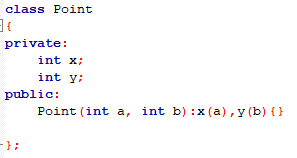
Es una parte opcional de un declarador, consiste del carácter “=” seguido de una expresión o una lista de expresiones separadas por comas y colocadas entre llaves. Esta lista se le denomina “lista de inicialización”(inicialization list).

***Formas de declarar una variable en C++***



Existen diferencias entre cada tipo de declaración, cada una llama a un constructor diferente. Veamos a continuación dos formas diferentes de construir una clase en C++

Esta es la forma genérica, la que conocemos, en ella al constructor se le pasan dos enteros como argumento y el asigna dichos enteros a las variables internas de la clase



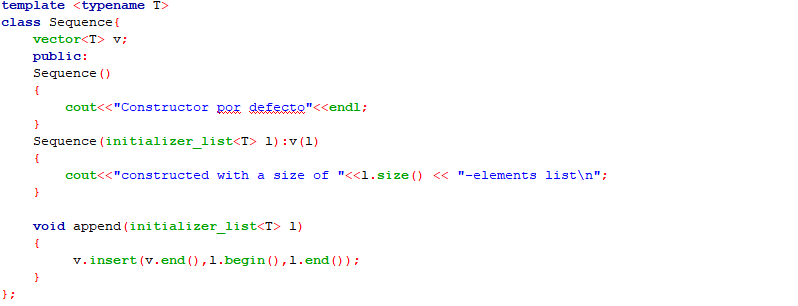
Usa initialization list en su definición. En este caso las variables X y Y se inicializan con los valores que se le pasan como parámetros a la función

Un constructor de clase puede tener un initializer list en su definición, previo a la definición del body. Cuando usamos initialization list, los valores no se asignan a la variable, sino que se inicializa la variable con dicho valor

En el primer caso, X y Y se inicializan con los valores de a y b

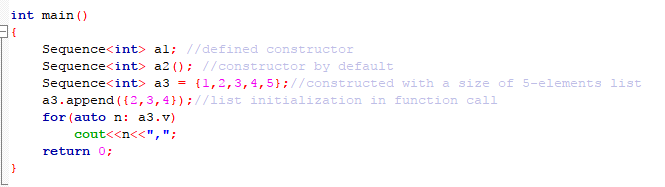
Pero C++11 provee un concepto más poderoso de inicialización de lista :

el std::initializer\_list



Aquí podemos ver un constructor normal de la clase, uno que recibe una initialization list como argumento y una función que recibe una initialization list y devuelve void

(si, las initialization list son un tipo)

Veamos ahora como llamar a cada constructor de la clase 

La 1ra instancia llama al constructor definido por nosotros e imprime defined constructor

La 2da instancia llama a un constructor que define C++ por defecto

La 3ra instancia llama al constructor que recibe la initialization list

a3.append({2,3,4}) llama a la función que recibe la initialization list como parámetro

Output

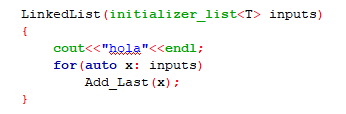
defined constructor

constructed with a size of 5-elements list

1,2,3,4,5,2,3,4

Se puede iterar sobre una initialization list, el vector se utilizó para guardar sus elementos

Constructor de Linked\_List<T> con initializer\_list



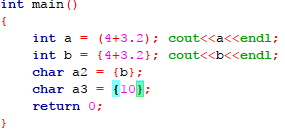


Ambos imprimen hola ☺

Entre los operadores () y {} también existe otra diferencia: clarar o no la conversión explicita de variables(uniformidad)

con llaves es uniforme, con paréntesis hace una conversión silenciosa

por ejemplo





El compilador nos advierte de que se harán conversiones explicitas y pueden cambiar los valores(con paréntesis estas operaciones son silenciosas y no avisan)

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

***a. Usar for\_each con expresiones lambda.***

LAMBDA

**Introducción**

Los especificadores de clase de almacenamiento son palabras clave que se pueden usar en declaraciones. No afectan el tipo de la declaración, pero generalmente modifican la forma en que se almacena la entidad.

**Mutable es** un especificador que se puede aplicar a la declaración de un miembro de datos no estático y node referencia de una clase. Un miembro mutable de una clase no es const incluso cuando el objeto es const.

class C {

int x;

mutable int times\_accessed;

public:

C(): x(0), times\_accessed(0) {

}

int get\_x() const {

++times\_accessed; // ok: const member function can modify mutable data member

return x;

}

void set\_x(int x) {

++times\_accessed;

this->x = x;

}

};

C ++ 11

Un segundo significado para mutable se añadió en C ++ 11. Cuando sigue la lista de parámetros de un lambda, suprime la const implícita en el operador de llamada de función del lambda. Por lo tanto, un lambda mutable puede modificar los valores de las entidades capturadas por copia.

std::vector<int> my\_iota(int start, int count) {

std::vector<int> result(count);

std::generate(result.begin(), result.end(),

[start]() mutable { return start++; });

return result;

}

**¿Qué es una expresión lambda?**

Una **expresión lambda** proporciona una forma concisa de crear objetos de funciones simples.

**Lambda mutable**

Los objetos capturados por valor en la lambda son, por defecto, inmutables. Esto se debe a que el operator() del objeto de cierre generado es const de forma predeterminada.

auto func = [c = 0](){++c; std::cout << c;}; // fails to compile because ++c

// tries to mutate the state of

// the lambda.

Se puede permitir la modificación utilizando la palabra clave mutable, que hace que el operator() del objeto más cercano no sea const :

auto func = [c = 0]() mutable {++c; std::cout << c;};

Si se usa junto con el tipo de retorno, mutable viene antes.

auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};

**Expresiones lambda: Sintaxis y uso combinado con algoritmos**

Los estándares del lenguaje **ISO C++ 2011** y posteriores proporcionan una manera concisa de generar objetos función mediante las denominadas **expresiones lambda**. Su utilidad fundamental consiste en la definición *in situ* de acciones específicas a realizar por parte de los algoritmos. Así, por ejemplo, la eliminación de los valores pares en un vector de enteros puede realizarse de dos formas alternativas:

**(1) Definición tradicional de un predicado como un tipo de objeto función (estilo C++98/03):**

   #include <algorithm>    #include <iostream>    #include <vector>    struct is\_even {       auto operator()(int n) const noexcept -> bool { return n%2 == 0; }    };    auto main() -> int    {       auto v = std::vector<int>{1, 2, 3, 4, 5, 6};       v.erase(std::remove\_if(v.begin(), v.end(), is\_even{}), v.end());       for (int i : v)          std::cout << i << ' '; // output: 1 3 5    }

**(2) Empleo de una expresión lambda (C++11 y posteriores):**

auto main() -> int {     auto v = std::vector{1, 2, 3, 4, 5, 6};     v.erase(std::remove\_if(v.begin(), v.end(), [](int n){ return n%2 == 0; }), v.end());     for (int i : v)        std::cout << i << ' '; }

Como puede observarse, la utilidad fundamental de la expresión lambda reside en la generación directa del objeto función *en el lugar mismo de su uso* en el algoritmo, a través de una sintaxis simplificada. El compilador es el encargado de generar un tipo de objeto función análogo al introducido en la opción **(1)**, si bien en este caso el nombre de la clase permanece desconocido para el usuario. De ser necesaria la reutilización del objeto función, éste puede ser inicializado antes de la ejecución del algoritmo como sigue (observemos la necesidad de utilizar inferencia automática de tipos):

   Auto v = std::vector{1, 2, 3, 4, 5, 6};    auto is\_even = [](int n){ return n%2 == 0; };    v.erase(std::remove\_if(v.begin(), v.end(), is\_even), v.end());

De forma notable, el operador llamada a función operator() de una expresión lambda es declarado constante (const) por defecto. Será, además, constexpr de cumplirse las condiciones para ello. Así pues, las expresiones lambda poseen el comportamiento que --como el tiempo ha demostrado-- debieran adoptar por defecto todas las funciones en el lenguaje, aunque esto resulte inviable por motivos obvios de retrocompatibilidad. Puedes visualizar una curiosa propuesta a este respecto en la referencia bibliográfica [3].  
  
En la sintaxis general de una expresión lambda, cabe distinguir las siguientes **secciones** principales:

|  |  |
| --- | --- |
| [*capture\_list*] (*parameters*) mutable(optional) noexcept(optional) -> *return\_type* { *body* } |  |

|  |  |  |
| --- | --- | --- |
| **Mutable** | **Nombre** | **Propiedades** |
| [] | Introductor o cláusula de captura | Con o sin lista de captura interior de variables externas (ver más adelante). |
| () | Declarador o lista de parámetros | Puede omitirse en ausencia de parámetros siempre que no se utilicen los declaradores mutable y noexcept o atributos. |
| mutable |  | Elimina la especificación const adoptada por defecto para el operador llamada a función, permitiendo la modificación de las variables capturadas en el introductor, así como la llamada a sus funciones miembro no constantes. |
| noexcept |  | Indica que la función no emite excepciones. |
| -> return\_type | Tipo de retorno | De ser omitido, el tipo es inferido a partir de las cláusulas de retorno de la función (o bien se toma como void en ausencia de las mismas). |
| {} | Cuerpo de la función lambda |  |

Como se ha indicado, una expresión lambda es capaz de capturar variables en su ámbito mediante su **introductor** o **cláusula de captura**. Entre los modos de captura fundamentales, se distinguen los siguientes casos (más información en [2]):

|  |  |
| --- | --- |
| **Modo** | **Propiedades** |
| [] | Modo por defecto. Ninguna variable externa es capturada. |
| [=]  ﻿ | La expresión lambda captura por valor todas las variables en su ámbito de las que haga uso. En los métodos de una clase, el objeto \*this es capturado implícitamente por referencia en este modo, lo que se considera obsoleto/*deprecated* a partir de C++20. |
| [&] | La expresión lambda captura por referencia todas las variables en su ámbito de las que haga uso, incluyendo el puntero this y el objeto \*this en los métodos de una clase. |
| [*capture\_list*] | Sólo la lista de nombres *capture\_list* es capturada (por referencia si el nombre la variable viene precedido del operador de dirección &). Se permite la inicialización de nuevas variables en el introductor. |
| [=,*capture\_list*] | Todas las variables no incluidas en *capture\_list* son capturadas por valor. |
| [&,*capture\_list*] | Todas las variables no incluidas en *capture\_list* son capturadas por referencia. |

Desde C++17, [=, \*this] realiza una copia independiente del objeto actual de una clase. Por su parte, C++20 permitirá la captura [=, this], actualmente equivalente a [=]. [&, this] y [&, \*this] resultan equivalentes a [&].  
  
Como ejemplo de aplicación de la tabla anterior, consideremos el siguiente código:

#include <functional>

#include <memory>

auto main()-> int    { auto a = 0, b = 1; auto p = std::make\_unique<int>(2); std::invoke([&a, x = b, q = std::move(p)]() mutable{ ++x; a += x + \*q; }); // (A)std::cout << std::boolalpha << !p << '\n'

// output: true (B)

<< a << '\n'// output: 4    (C)    << b; // output: 1    (D)    }

En **(A)**, el introductor de la expresión lambda toma una referencia del entero a, realiza una copia del entero b de nombre x y mueve el contenido del puntero inteligente std::unique\_ptr<int> p a un nuevo puntero q del mismo tipo. La expresión lambda utiliza el especificador mutable para permitir la modificación de la variable x dentro del cuerpo de la función. Tras la ejecución de la expresión lambda, el puntero p queda vacío, tal y como se comprueba en **(B)**. Asimismo, el valor original de a, variable capturada por referencia, se ve modificado por la función **(C)**. Por contra, el entero b permanece invariable al haber sido capturado por valor **(D)**.

El lenguaje permite también la introducción de **expresiones lambda genéricas**, parametrizadas con la palabra clave auto, de forma que sea el compilador el encargado de inferir los tipos de forma automática. Ello facilita enormemente la creación de algoritmos genéricos y, en muchos casos, simplifica sustancialmente la sintaxis de nuestro código, tal y como demuestra el siguiente ejemplo de ordenación de un vector de punteros inteligentes std::vector<std::unique\_ptr<S>>, siendo S una clase de objetos comparables:  
  
**(a) Versión con expresión lambda específica:**

   std::sort(v.begin(), v.end(), [](std::unique\_ptr<S> const& u1,                  std::unique\_ptr<S> const& u2){ return \*u1 < \*u2; });

**(b) Versión con expresión lambda genérica:**

   std::sort(v.begin(), v.end(), [](auto const& u1, auto const& u2){ return \*u1 < \*u2; });

Es más, desde **C++20**podremos adoptar una sintaxis análoga a la de las plantillas de función tradicionales cuando sea conveniente, típicamente en combinación con el empleo de **conceptos**. El siguiente ejemplo define una expresión lambda que mide el tiempo de ejecución de un objeto invocable con un conjunto de parámetros arbitrario:

#include <chrono> #include <concepts> #include <iostream> #include <string\_view> #include <thread> #include <utility> auto main() -> int {     using namespace std::chrono;     using namespace std::chrono\_literals;     auto time\_op = []<std::invocable Op, typename... Args>(Op op, Args&&... args){        auto const start = high\_resolution\_clock::now();        op(std::forward<Args>(args)...);        auto const end = high\_resolution\_clock::now();        auto const elapsed\_time = duration\_cast<milliseconds>(end - start).count();        std::cout << "time: " << elapsed\_time << " ms";     };     time\_op([]{ std::this\_thread::sleep\_for(7ms); }); // output: time: 7 ms }

A partir de C++20, se permitirá el uso de expresiones lambda en contextos no-evaluados (*unevaluated contexts*)  como decltype:

   using File\_ptr = std::unique\_ptr<FILE, decltype([](FILE\* f){ if (f) fclose(f); })>;    auto const fptr = File\_ptr{fopen("data.txt", "w")};    // gestión automática de cierre del fichero...

C++20 proporcionará también un modo de captura para *parameter packs*:

  template<typename... Args>   auto f(Args... args)   {      return [...args = std::move(args)] {         // empleamos los argumentos...      };   }

***6. Definir el destructor de la clase.***

***a. ¿Hace falta?***

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

***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.

***Los destructores tienen algunas características especiales:***

• Tienen el mismo nombre de la clase a la que pertenecen, pero tienen el símbolo ~ delante.

• No tienen tipo de retorno

• No tienen parámetros

• No pueden ser heredados

• Deben ser públicos

• No pueden ser sobrecargados

Como tal, no hace falta definir el destructor ya que el compilador de C++ proporciona un destructor de oficio que es casi siempre adecuado (excepto para liberar memoria de vectores y matrices).

***Sintaxis de los destructores:***

***Casos en los que es necesario usar raw pointer:***

• Iteradores

• Punteros a funciones (debería decir: funciones)

• Cuando no se está seguro del tipo de pertenencia que tiene el objeto (no me queda claro)

• Cuando se quiere compartir pertenencia de objetos y se desea ahorrar memoria

**Punteros a funciones:** es una variable que almacena la dirección de una función. Esta función puede ser llamada a través el puntero. Este tipo de construcción es útil pues encapsula comportamiento que también puede ser llamado a través de un puntero.

**Sintaxis:**

<tipo> (\*<identificador>) (<lista\_de\_parámetros>)

De esta forma se declara un puntero a una función que devuelve un valor de tipo <tipo> y acepta la lista de parámetros especificada. Es muy importante usar los paréntesis para agrupar el asterisco con el identificador, ya que de otro modo estaríamos declarando una función que devuelve un puntero al tipo especificado y que admite la lista de parámetros indicada.

***Utilidad:***

La utilidad de los punteros a funciones se manifiesta sobre todo cuando se personalizan ciertas funciones de biblioteca. Podemos, por ejemplo, diseñar una función de biblioteca que admita como parámetro una función, que debe crear el usuario (en este caso otro programador), para que la función de biblioteca complete su funcionamiento.

***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.***

***7.a***

noexcept es un nuevo recurso de C++11, que no es más que un operador unario que determina si la evaluación de su operando puede propagar una excepción. Tenga en cuenta que los cuerpos de las funciones llamadas no se examinan, por lo que noexcept puede producir falsos negativos. El operando no es evaluado. Al declarar una función, especifica si la función puede o no propagar una excepción. Solo, declara que la función no puede propagar una excepción. Con un argumento entre paréntesis, declara que la función puede o no puede propagar una excepción dependiendo del valor de verdad del argumento.

***b.***

AUTO:

**auto, const, y referencias**

La palabra clave auto por sí misma representa un tipo de valor, similar a int o char. Se puede modificar con la palabra clave const y el símbolo & para representar un tipo const o un tipo de referencia, respectivamente. Estos modificadores se pueden combinar. En este ejemplo, s es un tipo de valor (su tipo se deducirá como std::string ), por lo que cada iteración del bucle for *copia* una cadena del vector en s.

std::vector<std::string> strings = { "stuff", "things", "misc" };

for(auto s : strings) {

std::cout << s << std::endl;

}

Si el cuerpo del bucle modifica s (por ejemplo, al llamar a s.append(" and stuff") ), solo se modificará esta copia, no el miembro original de las strings .Por otro lado, si s se declara con auto& será un tipo de referencia (se inferirá que es std::string& ), por lo que en cada iteración del bucle se le asignará una *referencia* a una cadena en el vector:

for(auto& s : strings) {

std::cout << s << std::endl;

}

En el cuerpo de este bucle, las modificaciones a s afectarán directamente el elemento de las strings que hace referencia. Finalmente, si s se declara const auto&, será un tipo de referencia const, lo que significa que en cada iteración del bucle se le asignará una *referencia const* a una cadena en el vector:

for(const auto& s : strings) {

std::cout << s << std::endl;

}

Dentro del cuerpo de este bucle, s no se puede modificar (es decir, no hay métodos no const pueden ser llamados en él).

Cuando se utiliza el modo auto con el rango for bucles, generalmente es una buena práctica usar

const auto& si el cuerpo del bucle no modifica la estructura que se está repitiendo, ya que esto

evita copias innecesarias.

La inferencia automática de tipos mediante la palabra clave **auto** --técnica introducida por vez primera en el estándar **ISO C++ 2011**y refinada en estándares posteriores-- permite la declaración de variables de forma tal que sus tipos sean deducidos automáticamente a partir de los tipos de sus inicializadores, siguiendo las reglas clásicas de deducción de argumentos para plantillas de función (*function templates*). La inferencia automática puede ser también empleada en la signatura de una función para requerir que el tipo retornado sea deducido a partir de sus instruciones de retorno.  
  
Esta potente herramienta de programación fue diseñada por Bjarne Stroustrup a principios de los años 1980, pero tuvo que aguardar hasta su inclusión definitiva en el lenguaje por motivos de compatibilidad con el lenguaje C.  
  
Por ejemplo, en sustitución de

   std::unique\_ptr<int> p = std::make\_unique<int>(1);

podemos escribir

   auto p = std::make\_unique<int>(1);

auto puede también acompañarse de los modificadores const y &, que participarán en la deducción del tipo. Por ejemplo:

   auto x = 1.0;      // x es de tipo double

   auto b = x;        // b es una copia de x

   auto& c = x;       // c es una referencia a x

   auto const& d = x; // d es una referencia a x como double constante

Por supuesto, debe ser el programador en última instancia quien decida la conveniencia o no de declarar explícitamente el tipo de la variable en casos tan sencillos como los hasta ahora señalados. Existe sin embargo una tendencia declarativa dentro del lenguaje, conocida como **estilo 'always auto'**, que insta a inicializar las variables a través del *placeholder* auto en cualquier caso. Puedes encontrar más información en el *post* .

**Algunos ejemplos de uso**

Una de las utilidades más evidentes de la inferencia automática de tipos es la de simplificar la labor del programador y mejorar la legibilidad del código cuando el tipo de la variable a deducir no sea trivial, pero sí evidente por el contexto. Así, por ejemplo, con el fin de iterar a través de un vector, podríamos seguir un esquema propio de C++03 y escribir:

   std::vector<long double> v{0.5L, 1.2L, 2.5L, 3.9L, 4.8L};

   for (std::vector<long double>::const\_iterator p = v.cbegin(); p < v.cend(); ++p)

      std::cout << \*p << '\n';

En sustitución del bucle anterior, sin embargo, podríamos escribir:

   for (auto p = v.cbegin(); p < v.cend(); ++p)

      std::cout << \*p << '\n';

O bien, sencillamente, utilizando un bucle for basado en rango (observemos la desaparición del operador \* de desreferencia al trabajar ahora directamente con referencias a los elementos del vector):

   for (auto const& d : v)

      std::cout << d << '\n';

auto puede también evitar errores comunes de programación, en particular la conversión peligrosa de tipos:

   int const sz = v.size();  // ¡cuidado! size() retorna un entero sin signo

   auto const sz = v.size(); // ok! sz es de tipo std::vector<int>::size\_type constante

La palabra clave auto puede también emplearse en funciones y plantillas de función con el fin de deducir el tipo retornado a través de la expresión de retorno. Esta opción resulta especialmente útil en programación genérica. Consideremos, por ejemplo, la siguiente plantilla de función:

   template<typename T, typename U>

   auto sum(T const& a, U const& b)

   {

      return a + b;

   }

Observemos que, en general, los tipos T y U serán distintos, por lo que no es posible conocer *a priori* el tipo resultante de la suma de a y b. La inferencia automática resuelve esta dificultad, al ser capaz de determinar el tipo de la expresión operator+(a, b) en tiempo de compilación.

Asimismo, auto resulta imprescindible para declarar variables con nombre de tipo desconocido por el programador, como las **expresiones lambda** (véase la serie de *posts* correspondiente en este [enlace](https://dgvergel.blogspot.com/2015/07/expresiones-lambda-i.html)):

   auto same\_level = [](Enemy const& e\_1, Enemy const& e\_2){ return e\_1.level == e\_2.level; };

# C++ decltype

## Ejemplo

C ++ 11

Cede el tipo de su operando, que no se evalúa.

* Si el operando e es un nombre sin paréntesis adicionales, entonces decltype(e) es el tipo declarado de e.

int x = 42;

std::vector<decltype(x)> v(100, x); // v es un vector<int>

* Si el operando e es un acceso de miembro de clase sin paréntesis adicionales, entonces decltype(e) es el tipo declarado del miembro al que se accedió.

struct S {

int x = 42;};

const S s;

decltype(s.x) y; // y es de tipo int, aunque s.x is const

* En todos los demás casos, decltype(e) produce tanto el tipo como la [categoría](file:///\\fr\cplusplus\topic\763\value-categories) de [valor](file:///\\fr\cplusplus\topic\763\value-categories) de la expresión e , como sigue:
  + Si e es un valor de tipo T , entonces decltype(e) es T& .
  + Si e es un xvalor de tipo T , entonces decltype(e) es T&& .
  + Si e es un prvalor de tipo T , entonces decltype(e) es T

Esto incluye el caso con paréntesis extraños.

int f() { return 42; }

int& g() { static int x = 42; return x; }

int x = 42;

decltype(f()) a = f(); // a es d e tipo int

decltype(g()) b = g(); // b es de tipo int&

decltype((x)) c = x; // c es de tipo int&, aunque x es un lvalue

C ++ 14

La forma especial decltype(auto) deduce el tipo de una variable de su inicializador o el tipo de retorno de una función de las declaraciones de return en su definición, utilizando las reglas de deducción de decltype de decltype lugar de las de auto .

const int x = 123;

auto y = x; // y es de tipo int

decltype(auto) z = x; // z es de tipo const int, el tipo declarado de x

***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.***

Antes de realizar el ejercicio hablemos de 3 elementos distintos de C++ que nos ayudarán a su realización

-Variadic Template + Arguments Pack

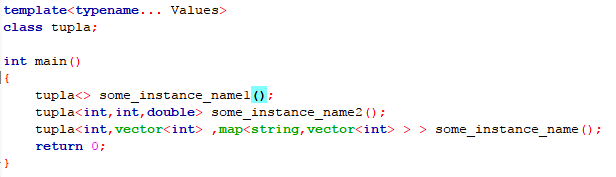
-Function Pointer

-Alias

-Variadic Template + Arguments Pack

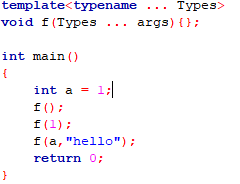
Variadic Template: Permite un número arbitrario de argumentos para el tipo template. En este caso tenemos la clase tupla, la cual admite cualquier cantidad de tipos como parámetros para su template. De esta forma podemos instanciar las tres tuplas que vemos a continuación

Paquete de argumentos



De esta forma hemos definido una clase tupla que recibe cualquier cantidad de parámetros de cualquier tipo….aunque no haga nada con ellos

Variadic templates también se pueden usar para funciones. Lo que nos permite crear funciones con una cantidad ilimitada de parámetros



El trabajo con este tipo de funciones requiere un procesamiento recursivo, pero es muy útil para definir el tipo de función con el que queremos trabajar

(las Funciones que devuelven void y reciben un int y un doble son un tipo)

Function pointer: Como su nombre indica es un puntero que apunta a una función. Su sintaxis es la siguiente

\*ReturnType\*(\*PointerName\*)(\*ParametersList\*)

Ejemplo:



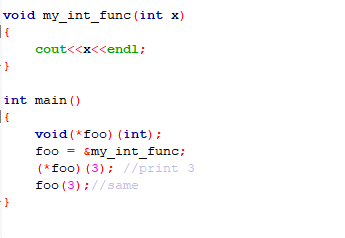
ReturnType: Tipo de retorno de la función, en este caso void

PointerName: Nombre del puntero que apunta a la función, en este caso foo

ParametersList:Tipo de los parámetros que recibe la función, en este caso un int

Esto es equivalente a declarar una función que se llama \*foo, que recibe un int y devuelve void

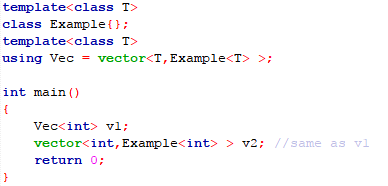
Ejemplo de uso:



Función que recibe un entero y lo imprime

Alias

1. Una declaración tipo Alias introduce un nombre que se puede usar como sinónimo a otro tipo, no cambia el significado del tipo al que apunta, es similar al uso de las macros pero se puede utilizar con los template, para sustituir los argumentos de los template

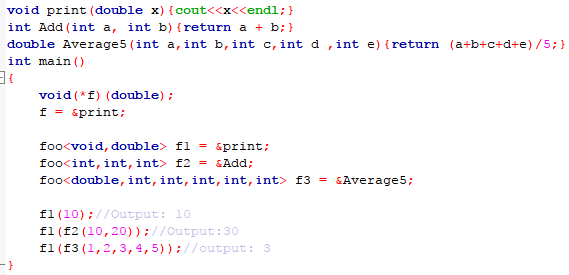
  
declaración de v1 y v2 son equivalentes

Ahora combinemos estos 3 elementos para crear un puntero a función genérico y asignémosle algunas funciones

Declaramos el alias a un puntero a función de tipo R, que recibe cualquier cantidad de parámetros de cualquier tipo



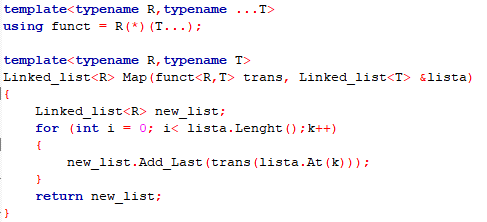
Declaramos algunas funciones y creamos un puntero para cada una de dichas funciones



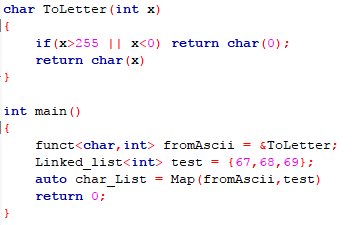
Invocamos a la función usando los punteros a dichas funciones

En el caso de la Linked\_List:

Función genérica Map que recibe un puntero a función (trans) que transforma un elemento de T a uno de R y con ella crea una Linked List de tipo R con los elementos de T de la lista original



Se usaría de la siguiente forma:



Función Map que recibe el puntero y la lista

Linked List generada con List Initialization

Puntero a dicha función

Función que mapea de int a char

