Cuarto Proyecto de LP – C++ y la Metaprogramación

Equipo 2:

Alejandro Escobar Giraudy C312

Airelys Collazo Perez C312

Henri Daniel Peña Dequero C311

Orientación:

1 - Implemente utilizando **TMP** (**Template Metaprogramming**) de **C++**:

1.1 - Un algoritmo que calcule en tiempo de **COMPILACIÓN** el n-ésimo valor de la secuencia de **Fibonnacci**.

1.2 - Implemente un **template** que permite desenrollar (**unroll**) una expresión **for** de manera que se genere código secuencial que simule la ejecución de esta expresión.

2. ¿Cómo surge la **meta-programación** utilizando **templates**?

3. Explique el principio **SFINAE** de **C++**.

4. Implemente utilizando **constexpr** de **C++**:

4.1 - Un algoritmo que calcule en tiempo de compilación el n-ésimo valor de la secuencia de **Fibonnacci**.

4.2 - Una clase **Point** que se pueda instanciar en tiempo de compilación.

4.3 - Un algoritmo que calcule el punto intermedio entre dos puntos en tiempo de compilación.

5. Explique hasta dónde se pueden utilizar las **constexpr** de **C++14**.

6. Sobre las características propuestas para **C++17**.

6.1 - Explique en que consiste **template argument deduction for class templates** y en qué puede ser útil.

6.2 - Explique la propuesta **if (init; condition)** y **switch (init; condition)**. (Puede ser útil en **C#**?)

6.3 - Exponga brevemente otras 2 propuestas al lenguaje que le parezcan atractivas.

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

Introducción:

La idea de este proyecto es explicar una de las herramientas más poderosas que se ha ido desarrollando en **C++** en los últimos tiempos, la **Metaprogramación**.

La **Metaprogramación** en general consiste en diseñar programas que hacen su trabajo en tiempo de compilación, y no en tiempo de ejecución. Es una solución muy útil para optimizar el tiempo de ejecución de programas que realizan gran cantidad de operaciones que no necesariamente deben esperar a una respuesta externa para ser ejecutados, ya que trabajarán siempre de la misma forma y dando la misma respuesta.

En concreto en **C++**, tenemos el **Template Metaprogramming** (**TMP**), el cual se refiere a lograr la **Metaprogramación** a partir del uso de plantillas.

Es poco frecuente que un usuario de **C++** lo use, no por ser menos útiles, sino porque es un paradigma de programación aunque poderoso, muy complejo, más bien utilizado por desarrolladores en la creación de bibliotecas (en general es poco frecuente el uso de plantillas por parte de los usuarios, se limitan a la programación genérica).

Eso sí, la **Metaprogramación** es más limitada que la programación genérica, por tanto requiere del programador un alto nivel de habilidades, refactorización de código (algunas veces no se permite usar todas las líneas deseadas, sobre todo en versiones más antiguas de **C++**), así como constantes de enumeración, herencia, especialización parcial por parte de las plantillas, etc.

1-)

El **Template Metaprogramming** (**TMP**) es una técnica de **metaprogramación** en la que un compilador utiliza las plantillas o **template** para generar código fuente temporal, que el compilador fusiona con el resto del código fuente y luego lo compila. Los **template** se usan para generar código en tiempo de compilación. La salida de estas plantillas puede ser constantes, estructuras de datos y funciones completas. Los **TMP** son utilizados por varios lenguajes, el más conocido es C ++, pero también lo tienen Curl, D, Nim y XL.

1.1-)

Usando **template** calcularemos el n-ésimo valor de la sucesión de **Fibonacci** en tiempo de compilación.

**#include <iostream>;**

**using namespace std;**

**template <int n>**

**struct Fibonacci**

**{**

**static constexpr int value = Fibonacci<n - 1>::value + Fibonacci<n - 2>::value;**

**};**

**template <>**

**struct Fibonacci<0>**

**{**

**static constexpr int value = 0;**

**};**

**template <>**

**struct Fibonacci<1>**

**{**

**static constexpr int value = 1;**

**};**

**int main()**

**{**

**cout << Fibonacci<5>::value << endl;**

**return 0;**

**};**

En el código antes visto cuando llamamos a **Fibonacci<5>** el compilador creará el código que corresponde a esta función, que en este caso es una estructura, en la cual se irá llamando recursivamente a **Fibonacci<n - 1>** y **Fibonacci<n - 2>,** que será la misma estructura lo que el **template** o plantilla con valores distintos. Cuando se llega a uno de los casos bases **n = 0** o **n = 1** el compilador entrará en el **template** que necesite según sea el caso, esto pasa debido a la regla de **SFINAE** que se explicará más adelante.

Cabe aclarar, que el compilador debe conocer el valor de sus parámetros en el momento de la compilación, lo cual tiene la condición previa de que el **Fibonacci<X>::value** solo se puede usar si se conoce **X** en el momento de la compilación. En otras palabras, **X** debe ser un literal constante o una expresión constante.

1.2-)

Ahora veamos un ejemplo de **template** que permite desenrollar (**unroll**) una expresión **for** de manera que se genere código secuencial que simule la ejecución de esta expresión.

Primero veamos un ejemplo de un ciclo **for**:

**int sum = 0;**

**for (int i = 0; i <= 5; i++)**

**{**

**sum += i;**

**}**

**cout << sum << endl;**

En el código anterior con el uso de un ciclo **for** se suma la secuencia de números del 1 al 5, ahora veamos este ejemplo resuelto con **template**.

**template < int n>**

**struct myfor**

**{**

**static const int sum = n + myfor <n - 1 >::sum;**

**};**

**template <>**

**struct myfor <1>**

**{**

**static const int sum = 1;**

**};**

**int main()**

**{**

**cout << myfor<5>::sum << endl;**

**return 0;**

**};**

Como se puede ver la idea es semejante a la de **Fibonacci**, cuando se llama a **myfor<5>** el compilador generará el código correspondiente de dicha función, una estructura, en la que se irá llamando recursivamente a **myfor <n - 1 >** y cuando se llegue al caso base **myfor <1>** el compilador entrará en dicha especificación.

2-)

La **Metaprogramación** de **C++** surgió por casualidad (de cierta forma), se diseñaban las plantillas de **C++**. Su idea inicial era que fueran una herramienta más para la programación genérica, pero sus desarrolladores se dieron cuenta de que cumplía varias cosas interesantes, principalmente que el sistema de **templates** era **Turing-Completo**, o sea, podía computar cualquier función **Turing-Computable**, y comenzaron entonces a prestarle mucha más atención. El sistema al inicio de **C++11** (que es donde fue oficialmente implementado para la **Metaprogramación**), era limitado, sin embargo, en las versiones de **C++14** y **C++17** se ha ido enriqueciendo (Ejemplos más concreto de la diferencia entre versiones las veremos más adelante), aún así, la **Metaprogramación** en general se ve limitada por las características del compilador del lenguaje al que pertenecen, siendo **C++** no una excepción.

3-)

**Substitution Failure Is Not An Error**, abreviado **SFINAE**, se refiere a cierta forma en la que son colocados los **templates** en **C++**, como bien su traducción quiere decir: “Un Fallo de Sustitución No Es Un Error”, indica que cometer un error en la sustitución de parámetros, aunque en principio parece que causará un error evidente de compilación, no será el caso (aunque quizás no funcione como se espera el programa debido a que no expresa exactamente lo que queremos hacer, pero no falla su compilación o ejecución).

Veamos un ejemplo en concreto para entender a qué se refiere el principio **SFINAE**:

**#include <iostream>**

**using namespace std;**

**struct My\_Struct**

**{**

**typedef int test\_int;**

**};**

**template <typename T>**

**string function(typename T::test\_int)**

**{**

**return "First function success";**

**}**

**template <typename T>**

**string function(T)**

**{**

**return "Second function success";**

**}**

**int main()**

**{**

**cout << function<My\_Struct>(0); //Imprime: "First function success"**

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

**cout << function<int>(0); //Imprime: "Second function success";**

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

**cout << function<char>(0); //Imprime: "Second function success";**

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

**return 0;**

**}**

Vamos a explicar qué sucedió en el código.

Definimos al inicio un **struct** que contiene dentro un atributo de tipo **int** llamado **test\_int**.

El primer **function** recibe específicamente un tipo que contenga un atributo de nombre **test\_int**, por tanto al llamarlo con **function<My\_Struct>** directamente reconoce que **My\_Struct** tiene dentro a **test\_int** y trabaja con esta función.

Por otro lado, las llamadas de **function<int>** en teoría debería conducir a un error, puesto que no se recibe el tipo apropiado, en vez de un tipo que contenga un atributo de nombre **test\_int**, recibimos un **int** (que no contiene un atributo con dicho nombre). Aquí es donde entra el **SFINAE**, ya que el compilador antes de lanzar el error, revisa el segundo **function** para ver si coincide con la entrada descrita. Al ver que la entrada **int** es totalmente válida para el tipo **T** descrito, entonces ejecuta el segundo **function.** Exactamente lo mismo ocurre al hacer **function<char>**.

Aunque es difícil de asimilar al inicio, se asemeja al principio del **Pattern Matching** de los lenguajes funcionales, aunque orientado más bien a los tipos específicamente.

4-)

El **constexpr** es una palabra reservada de **C++** utilizada para definir un valor o retorno de una expresión como constante, con el objetivo de que sea usado para resolver código en tiempo de compilación.

Esta no se debe confundir con **const**, que si bien marca un valor como constante, no lo hace en tiempo de compilación, sino en tiempo de ejecución.

Sin embargo, estas funciones como dijimos más arriba tienen varias limitaciones para poder ser usadas en tiempo de compilación, con respecto a las funciones clásicas.

Entre estas limitantes se encuentran que solamente pueden guardar su valor en variables **constexpr**, así como solo pueden recibir valores de otras funciones **constexpr** (puesto que no pueden guardar ni recibir nada de variables que se ejecutarán en tiempo de ejecución), por tanto, la función no puede tener como valor de retorno **void**, así mismo no puede ser declarada como virtual y no puede tener dentro sentencias como el **try**.

4.1-)

En **C++11** específicamente era más difícil de utilizar las funciones **constexpr**, puesto que además de todo lo mencionado tenían la limitante de que debían resolverse en una solo sentencia.

**Fibbonacci** en **C++11** hubiera tenido que ser diseñado de la siguiente forma:

**#include <iostream>**

**using namespace std;**

**constexpr int Fibbonacci(int x)**

**{**

**return x == 1 || x == 0 ? 1 : Fibbonacci(x - 2) + Fibbonacci(x - 1);**

**}**

**int main()**

**{**

**cout << Fibbonacci(10); //Imprime 89**

**return 0;**

**}**

En las versiones posteriores a **C++11**, ya se permite escribir más de una sentencia de una función **constexpr**, por ejemplo:

**constexpr int Fibbonacci(int x)**

**{**

**if (x == 1 || x == 0)**

**return 1;**

**else**

**return Fibbonacci(x - 2) + Fibbonacci(x - 1);**

**}**

4.2-)

También podemos utilizar el **constexpr** para declarar tipos, los cuales serán usados en tiempo de compilación, un ejemplo de esto es con la clase **Point**, que podemos definirla de la siguiente forma:

**#include <iostream>**

**using namespace std;**

**class Point**

**{**

**public:**

**double x;**

**double y;**

**constexpr Point(double in\_x, double in\_y) : x(in\_x), y(in\_y)**

**{**

**}**

**};**

**int main()**

**{**

**constexpr Point a = Point(0, 0); //Se crea este Point en tiempo de Compilación**

**Point b = Point(1, 1); //Se crea este Point en tiempo de Ejecución**

**return 0;**

**}**

Es muy importante notar que aunque no declaremos como **constexpr** la variable al inicializarla en el **main**, si bien no ocurrirá un error de compilación, la variable no será creada en tiempo de compilación, sino en tiempo de ejecución.

4.3-)

Ahora crearemos una función **Middle\_Point** (para calcular el punto medio entre dos **Point**) que sea **constexpr** y trabaje con la clase descrita anteriormente:

**#include <iostream>**

**using namespace std;**

**class Point**

**{**

**public:**

**double x;**

**double y;**

**constexpr Point(double in\_x, double in\_y) : x(in\_x), y(in\_y)**

**{**

**}**

**};**

**constexpr Point Middle\_Point(Point a, Point b)**

**{**

**double new\_x = (a.x + b.x) / 2;**

**double new\_y = (a.y + b.y) / 2;**

**return Point(new\_x, new\_y);**

**}**

**int main()**

**{**

**constexpr Point a = Point(0, 0);**

**constexpr Point b = Point(1, 1);**

**constexpr Point c = Middle\_Point(a, b);**

**cout << c.x; //Imprime 0.5**

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

**cout << c.y; //Imprime 0.5**

**return 0;**

**}**

En caso de haber realizado esta función **Middle\_Point** en **C++11** hubiera sido necesario realizarla en una sola sentencia tal y como dijimos con **Fibbonacci** al inicio.

5-)

Como ya dijimos, las funciones **constexpr** de **C++11** solo pueden contener una única expresión que se devuelve, así como ***static\_assert*** y una pequeña cantidad de declaraciones. **C++14** suaviza esas restricciones, ahora se puede contener cualquier declaración excepto: **static** o **thread\_local** variables, declaraciones de variables sin inicializar, las condiciones ***if*** o ***switch***, ciclos incluyendo ***for****,* expresiones que cambian el valor de un objeto si la vida útil de ese objeto comenzó dentro de la función de expresión constante. Además, las sentencias ***goto*** no están permitidas.

6-)

Sobre las características propuestas para **C++17**:

**C++17** permite escribir código simple, claro y expresivo. A continuación, hablaremos de algunas de las características introducidas que más nos llamaron la atención y por supuesto las de mayor interés para este trabajo. Algunas de las características introducidas en **C++17** son:

- **Structure** **Bindings**

- ***Folding******Expressions***

- **constexpr** ***if***

- **constexpr** ***lambda***

- ***template argument deduction for class templates***

- Declaración de variable en ***if*** y***switch***

***Structure Bindings*:**

Básicamente, le permite declarar múltiples variables que se inicializan con valores de pares, tuplas genéricas o de estructuras personalizadas y estas declaraciones de variables múltiples ocurren en declaraciones únicas.

Antes  se usaba para declarar múltiples variables que se inicializaban con valores de estructuras personalizadas.

**#include <iostream>**

**#include <tuple>**

**using namespace std;**

**int main()**

**{**

**// Usando la clase tupla**

**int a, b, c;**

**tie(a, b, c) = make\_tuple(1, 2, 3);**

**}**

Nota: **tie** y **make\_tuple** si no utiliza el **namespace std** debe colocarle **std::** delante

Pero ahora con **C++ 17** se puede hacer lo siguiente:

Suponga que tiene un diccionario que tiene nombres como claves y su idioma favorito como valores y esto se implementa usando un mapa contenedor estándar y desea insertar una nueva entrada usando el método de inserción. Este método de inserción devuelve un **std::pair** que contiene dos piezas de información, el primer elemento del par es un iterador y el segundo elemento es un valor booleano.

Hay dos casos a considerar aquí si el nuevo no está presente en el diccionario o ya está presente.

Si la nueva asociación (par clave-valor) no está presente en el diccionario, se inserta, entonces en este caso el par devuelto contiene un iterador que apunta al nuevo elemento y el valor booleano se convierte en **True**.

Si la nueva clave ya está presente, el iterador apunta a la clave existente y el valor booleano se vuelve **Falso**.

Ahora mostraremos el código para inspeccionar la bandera booleana y iterador de inserción.

**//Structure Bindings de C++17**

**#include <iostream>**

**#include <map>**

**#include <string>**

**using namespace std;**

**int main()**

**{**

**//Par Llave-Valor declarado dentro**

**//Estructura del mapa de datos**

**map<string, string> fav\_lang{**

**{ "John", "Java" },**

**{ "Alex", "C++" },**

**{ "Peter", "Python" }**

**};**

**//El concepto de Structure binding,**

**//a partir de posiciones y sucesos es**

**//utilizado es el primer y segundo lugar**

**auto [process, success] = fav\_lang.insert({ "Henry","Golang" });**

**// Chequeamos el valor booleano**

**if (success) {**

**cout << "Insertion done!!"**

**<< endl;**

**}**

**//Iteramos por el mapa**

**for (const auto& [name, lang] : fav\_lang) {**

**cout << name << ":"**

**<< lang << endl;**

**}**

**return 0;**

**}**

***Folding******Expressions****:*

**C ++ 11** dio la opción de plantillas variadas para trabajar con un número variable de argumentos de entrada. Las expresiones ***Fold*** son una nueva forma de descomprimir parámetros **variadic** con operadores. La sintaxis es la siguiente:

**(pack op …)**

**(… op pack)**

**(pack op … op init)**

**(init op … op pack)**

Donde paquete representa un paquete de parámetros no expandido, **op** representa un operador e **init** representa un valor.

Antes de **C++ 17** para hacer una función que toma un número variable de argumentos y devuelve la suma de argumentos.

**int sum(int num, ...)**

**{**

**va\_list valist;**

**int s = 0, i;**

**va\_start(valist, num);**

**for (i = 0; i < num; i++)**

**s += va\_arg(valist, int);**

**va\_end(valist);**

**return s;**

**}**

Ahora para implementar una función recursiva como suma, etc. a través de plantillas variadas, esto se vuelve eficiente con **C++17**, que es mejor que las implementaciones de **C++11**. A continuación se muestra la clase de plantilla de la misma:

**// Template para realizar la recursión**

**// usando variadic template**

**auto C11\_sum()**

**{**

**return 0;**

**}**

**template<typename T1, typename... T>**

**auto C11\_sum(T1 s, T... ts)**

**{**

**return s + C11\_sum(ts...);**

**}**

***constexpr******if****:*

Otra funcionalidad nueva en **C++17** son las **constexpr** ***if*** que permiten incluir código que es instanciado en dependencia de una condición establecida en tiempo de compilación. La condición debe poder ser evaluada en tiempo de compilación.

Ejemplo: Suponga que tiene alguna plantilla que funciona con algún tipo **T** genérico.

**template <typename T>**

**auto func(T const& value)**

**{**

**if**

**constexpr (T is integer)**

**{**

**// Código del if**

**}**

**else {**

**// Código del else**

**}**

**}**

***constexpr******lambda****:*

**C++17** incluye además funciones lambdas que pueden ser evaluadas en tiempo de compilación usando **constexpr**. El cuerpo de las funciones tiene que poder ser evaluado en tiempo de compilación.

Ejemplo:

**#include <iostream>**

**using namespace std;**

**auto identity = [](int n) constexpr { return n; };**

**constexpr auto sum = [](int x, int y) {**

**auto X = [=] { return x; };**

**auto Y = [=] { return y; };**

**return X() + Y();**

**};**

**int main() {**

**static\_assert (sum(2, 3) == 5, "ERROR");**

**static\_assert (identity(3) == 3, "ERROR");**

**return 0;**

**}**

*template argument deduction for class templates*:

Esta es una nueva funcionalidad para deducir los tipos de los argumentos en una instancia de un ***template*** de clases. Cuando se llama una función o se declara una variable que implementa ***template*** y no se especifican los tipos de sus argumentos, el compilador procede a deducir estos tipos siguiendo un conjunto de reglas llamadas guías de deducción.

En los siguientes contextos, el compilador deducirá los argumentos de la plantilla del tipo de inicializador:

Cualquier declaración que especifique la inicialización de una variable y una plantilla de variable:

**std::pair p(2, 4.5); // deduces to std::pair<int, double> p(2, 4.5);**

**std::tuple t(4, 3, 2.5); // same as auto t = std::make\_tuple(4, 3, 2.5);**

**std::less l; // same as std::less<void> l;**

Una nueva expresion:

**template<class T> struct A**

**{**

**A(T, T);**

**};**

**auto y = new A{ 1, 2 }; // El tipo guardado es A<int>**

Expresiones de conversión de **function-style**:

**auto lck = std::lock\_guard(mtx); //Deduce a std::lock\_guard<std::mutex>**

**std::copy\_n(vi1, 3,**

**std::back\_insert\_iterator(vi2)); //Deduce a std::back\_insert\_iterator<T>,**

**//Donde T es del tipo del contenedor vi2**

**std::for\_each(vi.begin(), vi.end(),**

**Foo([&](int i) {...})); //Deduce a Foo<T>,**

**//Donde T es el único tipo del lambda**

el tipo de un parámetro de un **non-type template**:

**template<class T>**

**struct X**

**{**

**X(T) {}**

**auto operator<=>(const X&) const = default;**

**};**

**template<X x> struct Y {};**

**Y<0> y; // OK, Y<X<int>(0)>**

Desventajas:

- Deben proporcionarse los tipos de los argumentos de la plantilla, aunque esté bastante claro cuáles son.

- Se basa en la existencia de la plantilla **make\_pair**, por lo que si deseamos proveer de una facilidad similar a nuestras plantillas de clases debemos estar seguros que exista una función similar.

Ventajas:

- Es una buena técnica que elimina la necesidad definir una función auxiliar como **make\_pair**.

- Nos permite una mayor flexibilidad en la escritura de código.

*Propuesta* ***if (init; condition)*** *y* ***switch (init; condition)****:*

**C++17** incluye una nueva forma de tratar con las condicionales. En versiones anteriores teníamos código de la siguiente manera:

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**auto val = GetValue();**

**if (condition(val)) {**

**// caso true**

**}**

**else {**

**// caso false**

**}**

**return 0;**

**}**

Aquí nos vimos obligado a declarar la variable val antes de llegar al **if** siendo una manera de programar no muy limpia y propensa a un mal manejo de memoria.

**C++17** nos da una solución a esta problemática con la incorporación de la instrucción **if(init; condition)** con la que el código queda de la siguiente manera:

**int main() {**

**if (auto val = GetValue(); condition(val)) {**

**// caso true**

**}**

**else {**

**// caso false**

**}**

**return 0;**

**}**

En este ejemplo la variable val solo existe dentro de los **scopes** **if** y **else**

Esto también se puede aplicar para **switch** agregado también en c++ 17.

Ejemplo:

**int main() {**

**switch (auto val = GetValue(); options(val)) {**

**case option1:**

**// option1**

**break;**

**case option2:**

**// option2**

**break;**

**case default:**

**//default option**

**break;**

**}**

**return 0;**

**}**

Esta propuesta sería muy útil en otros lenguajes de programación como c# que nos permita trabajar con variables creadas y destruidas en las condicionales de una forma fácil y reduciendo líneas de código permitiendo además mejor legibilidad evitando hacer variables innecesarias que se queden “vivas” hasta que se acabe el método o se cierre el programa.

Nos parecen especialmente atractivas las propuestas ***Structure******Bindings*** y ***Folding*** ***Expressions***.

***Structure******Bindings:*** Como hemos visto anteriormente esto nos permite declarar múltiples varaibles inicializadas a partir de una supla o estructura logrando así una mayor limpieza y legibilidad en el código, se puede utilizar con arrays para obtener los elementos del array fácilmente.

***Folding*** ***Expressions:*** En general estas permiten escribir código más limpio, corto y legible dado que antes las plantillas variadas requieren muchas veces un código adicional cuando se quiere implementar funciones recursivas dado que es necesario especificar reglas para la recursividad algo que actualmente se puede hacer mucho más simple.