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.

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.