Metaprogramación
C++11, C++14 y C++17

Ciencias de la Computación UH

Gabriel Martín Fernández C-311
Miguel A. Asin Barthelemy C-311
Samuel D. Suárez Rodríguez C-312
Enmanuel Verdesia Suárez C-311

¿Qué es la metaprogramación?

"...La metaprogramación consiste en escribir programas que describen o manipulan otros programas (o a si mismos) como datos, o que hacen en tiempo de compilación parte del trabajo que, de otra manera, se realizaría en tiempo de ejecución. Esto permite al programador ahorrar tiempo en la producción de código..."[1]

Una definición más concisa de la idea de metaprogramación sería:

"Es la habilidad de usar código para generar código"

Así como todo programa construye un modelo para describir su dominio. El dominio de un metaprograma es otro programa denominado programa objeto o base y tendrá un objeto que describe a ese programa, al que llamamos metamodelo.

En el siguente ejemplo, nuestro dominio contiene diferenes tipos de animales, entre ellos perros y humanos.

El programa describe las características de los elementos del dominio utilizando (por ejemplo) clases, métodos y atributos. Entonces, el modelo contiene una clase Perro, que modela a los perros en el dominio, mientras que el programa manipula instancias de esta clase Perro.

Un metaprograma tendrá a su vez un (meta)modelo que describe a su dominio, el programa base. Así como en el dominio hay animales concretos, los habitantes del "metadominio" (=programa base) serán los elementos del programa: por ejemplo, clases, métodos atributos. Entonces el metamodelo deberá tener clases que permitan describir esos conceptos, por ejemploen el metamodelo de Java encontraremos clases Class, Method, Field. Este metamodelo describe la estructura posible de un programa Java. En otro lenguaje, este metamodelo tendría diferentes elementos.

Metaprogramación en C++

En C++, la metaprogramación se refiere al uso de macros o plantillas para generar código en tiempo de compilación. En general, las macros están mal vistas en este rol y se prefieren las plantillas, aunque no sean tan genéricas.

La metaprogramación de plantillas a menudo hace uso de cálculos en tiempo de compilación, ya sea a través de *templates* o funciones *constexpr*, para lograr sus objetivos de generación de

código. Sin embargo, los cálculos en tiempo de compilación no son metaprogramación en sí, como vimos anteriormente.

template metaprograming

La metaprogramación en C++ usando templates fue descubierta de forma accidental durante el proceso de estandarización del lenguaje. Mientras se escribían las reglas de instanciación de templates, resultó que el sistema de templates era Turing-completo (en principio capaz de de computar cualquier función Turing computable). Sin embargo, la técnica de metaprogramación con templates ha avanzado mucho desde descubrimiento y ahora es usada por creadores de librerías en C++. Su complejidad en sintaxis y semántica explica por qué no es más utlizada por la mayoría de los programadores de este lenguaje.

El siguiente ejemplo muestra la suma de dos enteros en tiempo de compilación utilizando *templates*.

```
using namespace std;

template<int a, int b> struct Add{
    enum { result = a + b };
};
int main()
{
    cout << Add<2, 3>::result << endl; //imprime 5
    return 0;
}
```

Consideremos que se desea calcular el $n-$ésimo valor de la suseción de fibbonacci en tiempo de compilación usando *templates*. Basándonos en la idea del código anterior (plantilla de estructura) dicho algoritmo sería el siguiente:

```
using namespace std;

template<int n> struct Fibbonacci{
    static const int value = Fibbonacci<n - 1>::value + Fibbonacci<n - 2>::value;
};
template<> struct Fibbonacci<0>{
    static const int value = 1;
};
template<> struct Fibbonacci<1>{
    static const int value = 1;
};
int main()
{
    cout << Fibbonacci<7>::value << endl; //imprime 21
    return 0;
}
```

En el código anterior, al hacer el llamado de la estructura $Fibbonacci < 7 >$, el compilador genéra el código asociado a dicha función (crea una estructura $Fibbonacci < 7 >$). Dentro de esta estructura se vuelve a llamar recursivamente a $Fibbonacci < n - 1 >$ y $Fibbonacci < n - 2 >$,

misma estructura pero valores de plantilla distintos, por lo que se genera su código correspondiente. Esta operación se realiza hasta que $n = 0$ o $n = 1$, en el cual (aplicando la regla SFINAE) el compilador entrará en su respectiva especialización: $Fibbonacci < 0 >$ o $Fibbonacci < 1 >$. Note que independiente de la cantidad de llamados que se realice a $Fibbonacci < n - k >$ $(1 < k < n - 1)$, solo se generará un código asociado a $Fibbonacci < n - k >$.

En los ejemplos anteriores se usaron distintos tipos de variables para obtener dichos valores (*enum* en el primer caso y *static const int* en el segundo), ambos constituyen valores constantes que se pueden declarar dentro de clases. La diferencia entre ambos está en que los *static const* poseen dirección en memoria, a dierencia de los *enum*. El compilador debe pasar la dirección del parámetro, instanciar y asignar memoria para la definición del miembro estático.

Los *enum* no poseen dirección en memoria asociada, por tanto si se pasan como parámetros no se usa ninguna memoria estática.

Consideremos que se desea implementar un *template* que permita desenrollar (*unroll*) una expresión *for* de manera que se genere código secuencial que simule la ejecución de esta expresión. Usando la idea anterior (ver Fibbonacci), dicha implementación será algo parecida a lo siguiente:

```
using namespace std;

template<int start, int end, int inc> struct _for{
    static void run(){
        //code here
        _for<start + inc, end, inc>::run;
    }
};
template<int end, int inc> struct _for<end, end, inc>{
    static void run(){
        return ;
    }
};
int main()
{
    _for<0, 10, 1>::run();
    return 0;
}
```

Substitution Failure Is Not An Error

En los ejemplos anteriores hicimos referencia a la regla SFINAE (Substitution Failure Is Not An Error por sus siglas en inglés). Este término es usado en C++ para referirse a una situación donde una sustitución indebida de los parámetros de un *template* no es un error en si. Como tal el compilador sigue esta regla para lidiar con las sobrecargas que pueda tener un *template* determinado puesto que estas sobrecargas se instancian con parámetros distintos y el compilador puede sustituir erróneamente los parámetros. Esto genera un error, pero el uso de esta regla le permite al compilador avanzar hacia la siguiente sobrecarga sin detenerse y descartar la anterior, al ser inválida para los parámetros que se tienen.

El siguiente código muestra un ejemplo de la aplicación de esta regla:

```
using namespace std;
```

```cpp
struct PInt{
    typedef int PseudoInt;
};
template<typename T> int f(typename T::PseudoInt){
    return 1;
}
template<typename T> int f(T){
    return 2;
}
int main(){
    cout << f<PInt>(5) << endl; //imprime 1
    cout << f<int>(5) << endl;  //imprime 2
    return 0;
}
```

Al realizar el primer llamado a f con el tipo de dato $PInt$, el compilador comprobará que este posee el tipo de dato $PseudoInt$ o int, por lo que realizará el retorno sin ningún problema, pero en el segundo caso, el tipo de dato int no posee ningún atributo de tipo $PseudoInt$, sin embargo en vez de retornar un error de compilación, comprueba la especialización de f la cual recibe este tipo de datos sin ningún problema y ertorna el valor correspondiente.

Introduciendo *constexpr*

constexp es una palabra clave agregada en C++11 que se puede usar para marcar el valor de una variable como una expresión constante, una función potencialmente utilizable en expresiones constantes, o (desde C++17) una declaración if que tiene solo una de sus ramas seleccionadas para compilar. En esencia sirve para especificar que el valor de un objeto o función puede ser evaluado en tiempo de compilación y la expresión puede ser usada por otras expresiones constantes.

Algunas restricciones que posee el uso de funciones *constexpr* son:

- En C++11 una función *constexpr* debe contener solo una sentencia de retorno. A partir C++14 se pueden tener varias.
- Una función *constexpr* solo debe referenciar a variables que sean constantes globales.
- Dentro de la función solo pueden llamarse otras funciones *constexpr*.
- La función no puede retomar el tipo void, el retorno debe ser un $LiterarType$.
- No puede ser virtual.
- El cuerpo de la función no puede contener sentencias goto ni try.

Es común el hecho de encontrar el parecido entre *constexpr* e *inline*, pues ambas funciones se utilizan para mejorar el rendimiento del código. Una función o constructor *constexpr* es implícitamente *inline*. Recordemos que las funciones *inline* son evaluadas siempre en tiempo de ejecución, sin embargo sigue las mismas reglas para ambos tipos de funciones. Nótese que *constexpr* no implica *inline* para variables no estáticas (comparando con las variables *inline* de C++17).

Anteriormente vimos como determinar la suma entre dos números y el $n-$ésimo termino de la suseción de fibbonacci en tiempo de compilación usando *templates*. A continuación se muestra una manera de obtener el mismo resultado usando *constexpr*.

```
using namespace std;

constexpr int Add(int a, int b){
    return a + b;
}
int main()
{
    cout << Add(2, 3) << endl; //imprime 5
    return 0;
}
```

Mientras que *fibbonacci* quedaría:

```
using namespace std;

constexpr int Fibbonacci(int n){
    return n == 0 || n == 1 ? 1 : Fibbonacci(n - 1) + Fibbonacci(n - 2);
}
int main()
{
    cout << Fibbonacci(7) << endl; //imprime 21
    return 0;
}
```

constexpr usado como un tipo de función tiene un conjunto de restricciones determinadas y cierta similitud con *inline*. Sin embargo visto como un tipo de variable:

- su tipo debe ser *LiteralType*,
- debe ser inicializada inmediatamente y la expresión completa de su inicialización debe ser una expresión constante y,
- su destructor debe ser constante, lo que implica que si el tipo de variable requiere un destructor específico (como por ejemplo para una clase o array) esta será *constexpr* también, de lo contrario será el destructor por defecto, el cuál es constante;

además, muy comunmente puede ser confudido con el tipo de dato *const*. Sin embargo ambos tienen propósitos diferentes ya que *constexpr* es usado principalmente para optimización de código mientras que *const* es empleada al trabajar con objetos constantes prácticos (por ejemplo el número π). Cada uno de ellos pueden ser aplicados a los miembros de un método. *const* se usa fundamentalmente para asegurarse de que no hayan cambios accidentales (ni intencionales) dentro del método, mientras que *constexpr* se emple con la idea de obtener resultados en tiempo de compilación a fin de mejorar el tiempo de ejecución final del programa.

En la programación orientada a objetos en C++, *constexpr* también tiene sus aplicaciones. Para ello crearemos una clase *Point* la cuál será posible instanciar en tiempo de compilación.

```
using namespace std;

class Point{
```

```
public:
    double X, Y;
    constexpr Point(double x, double y) : X(x), Y(y) {}
};
int main()
{
    constexpr Point p = Point(2, 3);
    return 0;
}
```

Note que al llamar al crear la variable p de tipo *Point* esta debe ser *constexpr*, pues de lo contrario, aunque el compilador no lo considere un error, la instanciación se realizaría en tiempo de ejecución. Para comprobar el correcto funcionamiento de la clase *Point*, crearemos una función *MiddlePoint* que retorne un valor tipo *Point*, el cual sea el punto medio de dos puntos que reciba como parámetro, dicho resultado se debe conocer en tiempo de compilación:

```
constexpr Point MiddlePoint(Point p1, Point p2){
    return Point((p1.X + p2.X) / 2, (p1.Y + p2.Y) / 2);
}
int main()
{
    constexpr Point p1 = Point(2, 3);
    constexpr Point p2 = Point(1, 2);
    constexpr Point m = MiddlePoint(p1, p2);
    cout << "(" << m.X << ", " << m.Y << ")" << endl; //imprime (1.5, 2.5)
    return 0;
}
```

Al ser *constexpr* y recibir como parámetro dos valores *constexpr*, es posible conocer el valor final de *m* en tiempo de compilación.

En los ejemplos de *Fibbonacci* y *MiddlePoint* solo se empleó una sola sentecia de retorno, esta limitación de C++11, se elimina a partir de C++14, en el cual ya es posible tener varias sentencias además del uso de condicionales sin necesidad de utilizar operadores ternarios.

Evolución del lenguaje a C++17

Con el paso de C++14 a C++17 se incorporan nuevas características y funcionalidades al lenguaje:

- *template argument deduction for class templates*
- Instrucciones “if(init; condition)” y “switch(init; condition)”
- *constexpr if*
- *constexpr lambda*

template argument deduction for class templates

C++ añade 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.

```
void test(int id, string const &name){
    pair<int, string> p(id, name);
    //...
}
```

- Desventaja: deben proporcionarse los tipos de los argumentos de la plantilla aunque esté bastante claro cuáles son.

```
void test(int id, string const &name){
    auto p{make_pair(id, name)};
    //...
}
```

- Desventaja: 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 funcion similar.

```
void test(int id, string const &name){
    pair p(id, name);
    //...
}
```

- Ventaja: es una buena técnica que elimina la necesidad definir una función auxiliar como *make_pair*.

```
int main()
{
    // ...
    tuple t1{1, 2, 3}; // tuple<int, int, int>
    tuple t2{t1};      // tuple<int, int, int> not tuple<tuple<int, int, int>>

    vector v1{1, 2};   // vector<int>
    less l;            // less<void>
    return 0;
    // ...
}
```

- Ventaja: Nos perimte una mayor flexibilidad en la escritura de código.

Instrucciones "if(init; condition)" y "switch(init; condition)"

Con la llegada de C++17 se incluye una nueva forma de tratar con las condicionales. En versiones anteriores a C++17 teníamos que escribir cosas como:

```
int main()
{
    // ...
    auto val = GetValue();
    if(condition(val)){
```

```
        // true case ...
    }
    else{
        // false case ...
    }
    // ...
    return 0;
}
```

Esto obliga a la declarar la variable val en un scope externo en el cual es usado, lo cual trae consigo un mal manejo de la memoria en la ejecución del programa y por tanto no es una manera limpia de programar.

C++17 incluye la instrucción "if (init; condition)". Con lo que se puede escribir el código anterior de la siguiente manera:

```
int main()
{
    // ...
    if(auto val = GetValue(); condition(val)){
        // true case ...
    }
    else{
        // false case ...
    }
    // ...
    return 0;
}
```

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

Otro ejemplo: Digamos que queremos buscar ocurrencias de palabras en un string. Con C++14:

```
int main()
{
    // ...
    string myString = "hello world!";

    auto it = myString.find("hello");
    if(it != string.npos){
        cout << it << "hello\n";
    }

    auto it2 = myString.find("world");
    if(it2 != string.npos){
        cout << it << "world\n";
    }
    // ...
    return 0;
}
```

Tenemos que usar 2 variables (*it*, *it*2) para un mismo objetivo. Sin embargo con C++17:

```
int main()
{
    // ...
    string myString = "hello world!";
```

```
    if(auto it = myString.find("hello"); it != string.npos){
        cout << it << "hello\n";
    }

    if(auto it = myString.find("hello"); it != string.npos){
        cout << it << "world\n";
    }
    // ...
    return 0;
}
```

De manera similar funciona la instrucción "switch(init; condition)":

```
int main()
{
    // ...
    switch(auto val = GetValue(); options(val)){
        case option1:
            // option1 ...
            break;
        case option2:
            // option2 ...
            break;
        // ...
        case default:
            //default option ...
            break;
    }
    // ...
    return 0;
}
```

constexpr if

Otra de las funcionalidades incluidas en C++17 permite 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. Sintaxis:

```
constexpr bool condition(){
    // code ...
}
int main()
{
    // ...
    if constexpr (condition()){
        // code ...
    }
    // ...
    return 0;
}
```

constexpr lambda

C++17 incluye además funciones lambda que pueden ser evaluadas en tiempo de compilación. El cuerpo de las funciones tiene que poder ser evaluado en tiempo de compilación, ejemplo:

```
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;
}
```

El futuro de C++17

Desde C++14 el lenguaje permite Lambda Genéricas; pero pese a que hacen las Lambdas de C++ más útiles, las Lambda genéricas no ofrecen toda la flexibilidad que podría esperarse, hasta el punto que en algunos contextos sigue siendo más útil utilizar una función plantilla antes que una Lambda Genérica. Por lo que una propuesta para la sintaxis Lambda sería:

[captura(s)]<parametro(s) plantilla>(parametro(s) lambda) { código; }

Es decir se añaden los "<" ">" a la lambda para poder gestionar parámetros de plantilla.

Una segunda propuesta puede ser la especialización parcial de plantillas de funciones, tal que sea posible especializar funciones sin necesidad de dar valor a cada uno de los parámetros de plantilla, sino a un subconjunto de estos.

Referencias

[1] http://es.m.wikipedia.org/wiki/Metaprogramaci%C3%B3n

Bibliografía

- http://geeksforgeeks.org
- http://stackoverflow.com
- http://docs.microsoft.com
- http://cppreference.com