Seminario #14: C++ y Metaprogramación

Integrantes: - Carlos Aguila - Eduardo García - Ricardo Piloto

1 y 2. Metaprogramación y Templates

La *Metaprogramación* es la programación que manipula las entidades de un programa, como clases y funciones.

Los *Templates* brindan soporte directo para la programación genérica en forma de programación utilizando tipos como parámetros. El mecanismo de templates de **C++** permite que un tipo o un valor sea un parámetro en la definición de una clase, una función o un alias de un tipo. Los templates proporcionan una forma sencilla de representar una amplia gama de conceptos generales y formas sencillas de combinarlos. Las clases y funciones resultantes pueden coincidir con código escrito a mano, menos general, en tiempo de ejecución y eficiencia de espacio.

Esto lleva a la noción de *Template Metaprogramming (TMP)* como un ejercicio de escritura de programas que computan en tiempo de compilación y generan programas.

TMP surgió de forma accidental; durante el proceso de estandarización del C++ se descubrió que su sistema de templates es **Turing-completo**, o sea, capaz de computar cualquier función **Turing-computable**.

Ejemplos

Calculando el 20-ésimo número de la secuencia de Fibonacci

```
#include <iostream>
using namespace std;

template<int n>
struct Fib{
    static const int res = Fib<n-1>::res + Fib<n-2>::res;
};
template<>
struct Fib<1>{
    static const int res = 1;
};
template<>
struct Fib<2>{
    static const int res = 2;
};

int main(){
    cout << Fib<20>::res << '\n';
```

```
    return 0;
}
```

Calculando la sumatoria de 1 a 10 (*Unroll* de una expresión *for*) Código para calcularla en tiempo de ejecución:

```
#include <iostream>
using namespace std;

int main(){
    int sum = 0;
    for (int i = 1; i <= 10;){
        sum += i++;
    }
    cout << sum << '\n';

    return 0;
}
```

Código para calcularla en tiempo de compilación:

```
#include <iostream>
using namespace std;

template<int n>
struct Sumatory{
    static const int sum = Sumatory<n-1>::sum + n;
};
template<>
struct Sumatory<1>{
    static const int sum = 1;
};

int main(){
    cout << Sumatory<10>::sum << '\n';
    return 0;
}
```

Imprimiendo la tabla de multiplicación del 2 Código para calcularla en tiempo de ejecución:

```
#include <iostream>
using namespace std;

void mult_print(int index, int value){
    cout << value << " x " << index << " = " << value*index << '\n';
}
```

```cpp
int main(){
    for (int i = 1; i <= 10; i++){
        mult_print(i, 2);
    }

    return 0;
}
```

Código para calcularla en tiempo de compilación:

```cpp
#include <iostream>
using namespace std;

template<int index, int end, int value, typename Function>
class FunctionExecutorLoop{
public:
    static void exec(Function f){
        f(index, value);
        FunctionExecutorLoop<index+1, end, value, Function>::exec(f);
    }
};
template<int end, int value, typename Function>
class FunctionExecutorLoop<end, end, value, Function>{
public:
    static void exec(Function f){
        f(end, value);
    }
};

void mult_print(int index, int value){
    cout << value << " x " << index << " = " << value*index << '\n';
}

int main(){
    FunctionExecutorLoop<1, 10, 2, decltype(mult_print)>::exec(mult_print);
    return 0;
}
```

3. SFINAE (Substitution Failure Is Not An Error)

Substitution Failure Is Not An Error (SFINAE) se refiere a una situación en C++ en la que una sustitución no válida de los parámetros del template no es en sí misma un error. Específicamente, al crear un conjunto de candidatos para la resolución de sobrecarga, algunos (o todos) los candidatos de ese conjunto pueden ser el resultado de templates instanciados con argumentos sustituidos por los parámetros del template correspondientes. Si se produce un error durante

la sustitución de un conjunto de argumentos para un template determinado, el compilador elimina la sobrecarga potencial del conjunto candidato en lugar de detenerse con un error de compilación. Si quedan uno o más candidatos y la resolución de la sobrecarga tiene éxito, la invocación está bien formada.

El siguiente código es un ejemplo donde se da esta situación:

```
#include <iostream>
using namespace std;

struct OutterInt{
    typedef int InnerInt;
};

template<typename T>
void print(typename T::InnerInt){
    cout << "T has inner type definition" << '\n';
}
template<typename T>
void print(T){
    cout << "T has no inner type definition" << '\n';
}

int main(){
    print<int>(17); // T has no inner type definition
    print<OutterInt>(17); // T has inner type definition

    return 0;
}
```

En el ejemplo, intentar usar `T::InnerInt` resulta en un fallo de deducción para `print<int>` porque `int` no contiene un tipo interior llamado `InnerInt`, pero la invocación esta bien formada porque aún existe una función en el conjunto de funciones válidas candidatas. `print<int>(17)` ejecuta la segunda definición de `print`, sin error aunque no exista `int::InnerInt`, gracias a SFINAE.

4 y 5. Palabra clave `constexpr`

`constexpr` es un feature añadido al lenguaje en su versión 11, denota que un objeto o función puede ser evaluada en tiempo de compilación. La expresión señalada por `constexpr` puede ser utlizada por otras expresiones constantes.

El objetivo principal de esta palabra clave es mejorar el rendimiento de los programas al reducir la cantidad de cálculos en tiempo de ejecución mediante la realización de estos en tiempo de compilación.

La palabra clave ha sido mejorada con el avance de las versiones: a partir de **C++14** aumentaron las sentencias de retorno que podria tener una función

marcada con `constexpr` y a partir de **C++17** la palabra clave puede ser utilizada en bloques *if-else* donde una sola de sus secciones seria seleccionada para compilar.

Algunas restricciones de funciones marcadas con `constexpr`:

- debe referenciar solamente a constantes globales,
- solo pueden llamar a funciones marcadas con `constexpr`,
- no puede tener un retorno `void`, el tipo del retorno debe ser *LiteralType*,
- no puede ser marcada como `virtual`,
- no puede contener `goto` o `try`, ni operadores como `++` prefijo.

Ejemplos de funciones

Calculando el 20-ésimo número de la secuencia de Fibonacci

```
#include <iostream>
using namespace std;

constexpr int fib(int n){
    return (n==1 || n==2) ? n : fib(n-1) + fib(n-2);
}

int main(){
    constexpr int fib = fib(20);
    cout << fib << '\n';

    return 0;
}
```

Simple suma

```
#include <iostream>
using namespace std;

constexpr int sum(int a, int b){
    return a + b;
}

int ten(){
    return 10;
}

int main(){
    constexpr int result1 = sum(10, 7); // compila

    int t = ten();
    constexpr int result2 = sum(t, 7); // no compila
```

```
    cout << result1 << '\n';
    cout << result2 << '\n';

    return 0;
}
```

En el ejemplo anterior, para que el programa pueda compilar, o bien no se marca `result2` como `constexpr` para que el compilador no se vea obligado a cumputarla, o bien se marca la función `ten` como `constexpr` ya que podría.

`constexpr` para instanciar objetos en tiempo de compilación

`constexpr` puede ser utilizado en constructores para instanciar clases en tiempo de compilación. El cuerpo del constructor debe estar vacío y sus miembros, inicializarse con expresiones constantes.

A continuación, un ejemplo de una clase `Point` que puede ser instanciada en tiempo de compilación:

```
#include <iostream>
using namespace std;

class Point{
    float _x;
    float _y;

public:
    constexpr Point(float x = 0, float y = 0): _x(x), _y(y) {}

    constexpr float x() const {
        return _x;
    }
    constexpr float y() const {
        return _y;
    }
};

int main(){
    constexpr Point p(0, 5.46);
    cout << "p = P(" << p.x() << ", " << p.y() << ")\n";

    return 0;
}
```

y una función que puede calcular el punto medio entre dos puntos, tambien en tiempo de compilación:

```cpp
#include <iostream>
using namespace std;

constexpr Point mid_point(const Point& p1, const Point& p2){
    return Point((p1.x() + p2.x()) / 2, (p1.y() + p2.y()) / 2);
}

int main(){
    constexpr Point p1(0, 5.46);
    constexpr Point p2(4.3, 15);
    constexpr Point mp = mid_point(p1, p2);

    cout << "mp = P(" << mp.x() << ", " << mp.y() << ")\n";

    return 0;
}
```

¿Hasta dónde se puede utilizar `constexpr` en C++14?

A partir de C++14 se pueden definir y modicar variables dentro de las funciones marcadas con `constexpr`, asi como son permitidas tambien las estructuras de control de flujo (`if-else`, `for`, `while`, `do-while`, `switch-case`). A continuación un ejemplo del algoritmo para contar la cantidad de repeticiones de un caracter en un *string* como `char*`:

```cpp
#include <iostream>
using namespace std;

constexpr int count_char(const char* str, const char ch){
    int count = 0;
    for (int k = 0; str[k]; ++k){
        if (str[k] == ch){
            ++count;
        }
    }
    return count;
}

int main(){
    constexpr char* sentence = (char*)"Carlos lavaba las sábanas sucias";
    constexpr char ch = 'a';
    constexpr int count = count_char(sentence, ch);
    cout << count << '\n'; // Imprime 8

    return 0;
}
```

A partir de C++14 se pueden declarar métodos de tipo de rotorno `void` como `constexpr`:

```
#include <iostream>
using namespace std;

constexpr void square(int& n){
    n *= n;
}

constexpr int squared(int n){
    square(n);
    return n;
}

int main(){
    constexpr int n = squared(5);
    static_assert(n == 25, ""); // compila

    return 0;
}
```

Sobre Características propuestas para C++17

Template Argument Deduction for Class Templates

Propuesta basada en *Template Argument Deduction for Function Templates*, la problemática a resolver es la deducción por parte del compilador del tipo de los argumentos de la clase que sea desea construir, justo como ocurre a continuación con funciones:

```
template<class T>
int f(T obj);
```

`f` puede ser llamada especificando el tipo del template,

```
int a = f<float>(5.0);
```

o sin especificarlo, pues el compilador deduce que el tipo argumento del template gracias al tipo del argumento de la función.

```
int a = f(5.0);
```

Antes de C++17 este problema era resuelto con usando *Template Argument Deduction for Function Templates*, en lugar de especificar los tipos,

```
std::pair<int, float> p(0, 3.467);
```

se podia usar una función como

```
auto p = std::make_pair(0, 3.467);
```

A partir de C++17, ya se puede declarar sin problemas:

```
std::pair p(0, 3.467);
```

Esto se puede lograr gracias a algo llamado *template deduction guides*: patrones asociados a la clase template que le dice al compilador como traducir un conjunto de parámetros en los argumentos del template.

la introducción de *Template Argument Deduction for Class Templates* permitió evitar el uso de funciones como `std::make_pair` y lograr un código mas limpio y legible, siempre y cuando se sepa que los parámetros podrán ser deducidos por el compilador.

`if (init; condition)` y `switch (init; condition)`

Esta propuesta es para evitar el uso de variables declaradas fuera del scope de un bloque `if-else` o un bloque `switch-case` y que solo seran usadas dentro de estos scopes, asi, por ejemplo, un código que nos diga si un numero aleatorio es par o impar, pasaria de ser asi:

```
int k = random_number(); // función hipotetica
if (k % 2 == 0){
    std::cout << "Es par" << '\n';
} else {
    std::cout << "Es impar" << '\n';
}
```

a ser asi:

```
if (int k = random_number(); k % 2 == 0){
    std::cout << "Es par" << '\n';
} else {
    std::cout << "Es impar" << '\n';
}
```

en el segundo ejemplo la variable `k` solamente "vive" en el scope del bloque `if-else`.

De manera similar ocurre con la estructura `switch-case`, se puede lograr un codigo de esta forma:

```
switch(int k = random_number(); k){
case 0:
    // codigo
    break;
case 1:
    // codigo
    break;
...
case default:
```

```
    // mas codigo
    break;
}
```

Structure Bindings

Propuesta para enlazar los campos `.first` y `.second` de instancias de `std::pair` a variables.

Antes de C++17:

```
map<int, string> m{
    {1, "aaa"},
    {2, "bbb"},
    {3, "ccc"}
};

// insertar en el mapa
auto result = m.insert({4, "ddd"});
```

`result.first` es un iterator apuntado al nuevo par y `result.second` es un valor booleano que determina si el par fue insertado correctamente.

A partir de C++17:

```
map<int, string> m{
    {1, "aaa"},
    {2, "bbb"},
    {3, "ccc"}
};

// insertar en el mapa
auto [it, success] = m.insert({4, "ddd"});
if (success){
    // codigo
}
```

`it` captura el iterador apuntando al nuevo par y `success` el valor booleano que determina si el par fue insertado correctamente.

Direct List Initialization of `enums`

A partir de C++17 la inicialización de `enums` usando llaves `{}` está permitida. A continuación un ejemplo de la sintaxis:

```
enum Byte: unsigned char {};
Byte a {0};
Byte b {-1}; // error
Byte c = Byte{1};
Byte d = Byte{256}; // error
```