Informe de Lenguajes de Programación: Seminario # 4

Laura Brito Guerrero
Sheyla Cruz Castro
Ariel Antonio Huerta Martín
Julio Daniel Gambe Alcorta
Pablo Antonio de Armas
Grupo 2

Universidad de La Habana
Facultad de Matemática y Computación

1. Metaprogramación

La metaprogramación es la escritura de programas de computación que a su vez escriben o manipulan otros programas (o a ellos mismos), tanto datos como código. Esta manipulación puede suceder tanto en tiempo de compilación como en tiempo de ejecución. En algunos casos esta técnica posibilita reducir el número de líneas de código requeridas para expresar una solución. También permite escribir programas con una gran flexibilidad para manejar eficientemente nuevas situaciones sin la necesidad de recompilar.

2. Templates

Los templates son el mecanismo de C++ para implantar el paradigma de la programación genérica. Permiten que una clase o función trabaje con tipos de datos genéricos, especificándose más adelante cuales son los que se quieren usar. Por ejemplo, una definición de una clase *template*:

```
template<typename T>
class container
{
  T* items;
  size_t size_items;
  public:
  ...
  size_t size(){return size_items;}
  ...
};
```

Una vez que el *template* es definido (como lo indicamos arriba) éste puede instanciarse para ser utilizado. El *template* definido no es una clase, sino una clase templetizada. Las clases templetizadas pueden convertirse en verdaderas clases mediante la instanciación de *templates*. Al instanciar un *template* indicamos el tipo concreto que queremos usar para la instanciación particular. La clase instanciada es como si se reemplazara sintácticamente el parámetro templetizado por el tipo concreto en el *template*. En el siguiente ejemplo se instancian dos clases, con los tipos *int* y *double*:

```
container<int> integer_container;
container<double> double_container;
```

Lo interesante de el proceso de instanciación de *templates* es que es Turing completo. Es decir, permite hacer cómputos en tiempo de compilación. Para ello nos valemos de una herramienta que provee $C++$ llamada especialización de *templates*. $C++$ permite definir y especializar clases *templates* para tipos más específicos. Por ejemplo, en el siguiente código se define el *template* llamado container y luego lo especializamos para enteros.

```
template<typename T>
class container
{
  T* items;
  size_t size_items;
```

```
  public:
  ...
  size_t size(){return size_items;}
  ...
};

template<> // Especializacion para int
class container<int>
{
  ...
};
```

De esta manera cuando se instancia container< *double* >, se usa la definición genérica, pero si se instancia container< *int* > el compilador usará la definición más especializada.

```
template<typename T>
class container
{...};
template<> // Especializacion para int
class container<int>
{...};
template<typename R> // Especializacion para vectores de cualquier tipo
class container<vector<R>>
{...};
template<typename R> // Especializacion para vectores de vectores
class container<vector <vector <R>>>
{...};
```

No sólo las clases pueden templetizarse, también las funciones y los miembros de clases pueden templetizarse y especializarse.

```
template<typename T>
T max(T const& x , T const& y )
{
  return x>y?x:y;
}
int x = max(3,5);
```

3. Template Metaprogramming

Template Metaprogramming (TMP) es una técnica de metaprogramación en la que un compilador utiliza los *templates* para generar código fuente temporal, que el compilador combina con el resto de código fuente y luego es compilado. El resultado de estos *templates* incluye constantes calculadas en tiempo de compilación, síntesis de nuevos tipos de datos y funciones. El uso de *templates* puede considerarse como una ejecución en tiempo de compilación. La técnica es utilizada por varios lenguajes, el más conocido es $C++$, pero también $Curl, D$ y XL.

El uso de *templates* como técnica de metaprogramación requiere dos operaciones distintas: se debe definir un *template* y se debe instanciar un *template* definido. La definición de *template* describe la forma genérica del código fuente generado, y la creación de instancias hace que se genere un conjunto específico de código fuente a partir de la forma génerica en el *template*.

TMP es Turing-completo, lo que significa que cualquier cálculo expresable por un programa de computadora puede calcularse, de alguna forma, mediante un *template metaprogram*.

TMP está mucho más cerca de la programación funcional que $C++$ idiomático común. Esto se debe a que las "variables"son todas inmutables y, por lo tanto, es necesario utilizar la recursión en lugar de la iteración para procesar elementos de un conjunto.

Aunque la sintaxis de **TMP** suele ser muy diferente del lenguaje de programación con el que se usa, tiene usos prácticos. Algunas razones comunes para usar *templates* son implementar programación genérica (evitando secciones de código que son similares, excepto algunas variaciones menores) o realizar una optimización automática en tiempo de compilación, como hacer algo una vez en tiempo de compilación en lugar de cada vez que se ejecuta el programa, por ejemplo, al hacer que el compilador desenrolle (*unroll*) bucles para eliminar saltos y decrementos de conteo de bucles cada vez que se ejecuta el programa.

3.1. ¿Cómo surgió?

Históricamente, **TMP** es una especie de accidente; se descubrió durante el proceso de estandarización del lenguaje $C++$ que su sistema de *templates* es Turing-completo, es decir, capaz en principio de calcular cualquier cosa que sea computable. La primera demostración concreta de esto fue un programa escrito por Erwin Unruh, que calculó números primos aunque en realidad no terminó de compilar: la lista de números primos era parte de un mensaje de error (**Figura 1**) generado por el compilador al intentar compilar el código. Desde entonces, **TMP** ha avanzado considerablemente y ahora es una herramienta práctica para los creadores de bibliotecas en $C++$, aunque sus complejidades significan que generalmente no es apropiado para la mayoría de los contextos de programación de sistemas o aplicaciones.

```
// Prime number computation by Erwin Unruh
template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
  enum {prim =(p%i) && is_prime<(i > 2 ? p : 0), i -1>::prim };
};

template <int i> struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i, i-1>::prim };
  void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum {prim=1}; };
struct is_prime<0,1> { enum {prim=1}; };
struct Prime_print<2> { enum {prim = 1}; void f() { D<2> d = prim; } };
```

```
#ifndef LAST
#define LAST 10
#endif
main () {
  Prime_print<LAST> a;
}
```

```
01 | Type `enum{}' can't be converted to txpe `D<2>' ("primes.cpp",L2/C25).
02 | Type `enum{}' can't be converted to txpe `D<3>' ("primes.cpp",L2/C25).
03 | Type `enum{}' can't be converted to txpe `D<5>' ("primes.cpp",L2/C25).
04 | Type `enum{}' can't be converted to txpe `D<7>' ("primes.cpp",L2/C25).
05 | Type `enum{}' can't be converted to txpe `D<11>' ("primes.cpp",L2/C25).
06 | Type `enum{}' can't be converted to txpe `D<13>' ("primes.cpp",L2/C25).
07 | Type `enum{}' can't be converted to txpe `D<17>' ("primes.cpp",L2/C25).
08 | Type `enum{}' can't be converted to txpe `D<19>' ("primes.cpp",L2/C25).
09 | Type `enum{}' can't be converted to txpe `D<23>' ("primes.cpp",L2/C25).
10 | Type `enum{}' can't be converted to txpe `D<29>' ("primes.cpp",L2/C25).
```

Figura 1

3.2. Ejemplos

El siguiente código es la implementación en $C++$ de la conocida función factorial:

```
int factorial (int n)
{
  if (n == 0)
    return 1;
  return n * factorial (n - 1);
}
void foo ()
{
  int x = factorial (4) ; // 24
}
```

En el caso anterior, se evalúa la función factorial con una constante, por lo tanto el cálculo de factorial se podría hacer en tiempo de compilación usando **TMP**:

```
template <int N>
struct Factorial
{
  enum { value = N * Factorial<N - 1> :: value };
};
template <>
struct Factorial <0>
```

```
  {
    enum { value = 1 };
  } ;
  // Factorial <4>:: value == 24
  // Factorial <0>:: value == 1
  void foo()
  {
    int x = Factorial <4>::value; // == 24
  }
```

Se han usado *templates* y especialización de *templates* para recrear la función factorial, y que ésta se calcule en tiempo de compilación.

3.3. Beneficios y desventajas de TMP

Tiempo de compilación vs Tiempo de ejecución: Si se utiliza una gran cantidad de TMP, la compilación puede ser lenta. La definición de un *template* no implica que se creará una instancia, y la creación de una instancia de un *template* de clase no hace que sus definiciones de miembro sean instanciadas. Dependiendo del estilo de uso, los *templates* pueden compilarse más rápido o más lento que el código enrollado a mano.

Programación genérica: TMP permite al programador centrarse en la arquitectura y delegar al compilador la generación de cualquier implementación requerida por el código del cliente. Por lo tanto, **TMP** puede lograr un código verdaderamente genérico, facilitando la minimización del código y una mejor capacidad de mantenimiento.

Legibilidad: Con respecto a $C++$, la sintaxis y las expresiones idiomáticas de **TMP** son esotéricas en comparación con la programación convencional de $C++$, y los *templates metaprograms* pueden ser muy difíciles de entender.

3.4. Fibonacci con TMP

```
template<int n>
struct fib
{
  static const int result = fib<n-2>::result + fib<n-1>::result;
};
template<>
struct fib<1>
{
  static const int result = 1;
};
template<>
struct fib<2>
{
static const int result = 1;
```

```
};
```

Para esto el compilador va sustituyendo los *templates* hasta que llega a los casos base y a partir de ahí regresa sumando los valores hasta llegar al primer llamado, retornando el resultado final.

Nota: Es importante tener en cuenta que los *templates* no se pueden relacionar con variables que no sean constantes, pues se necesita determinar su valor en compilación.

4. Loop Unrolling

Es una técnica de transformación de ciclos que ayuda a optimizar el tiempo de ejecución de un programa. Básicamente remueve o reduce las iteraciones. Uno de los ejemplos más ilustrativos del funcionamiento de la metaprogramación es el desenrollado de un ciclo. Por ejemplo, se quiere representar con *templates* el comportamiento del siguiente ciclo:

```
int sum = 0;
for(int i = 1; i <= n; i++)
{
  sum += i;
}
```

Quedaría así:

```
template<int i>
struct unroll
{
  static const int sum = i + unroll<i - 1>::sum;
};
template<>
struct unroll<1>
{
  static const int sum = 1;
};
```

Para esto se necesita más de un *template*, el segundo caso funciona como caso base y de ahí en adelante el compilador se encarga de la sumatoria.

5. SFINAE: Substitution Failure Is Not An Error

Esta regla se aplica durante la resolución de sobrecargas de funciones *templates*. Cuando la sustitución del tipo explícitamente especificado o deducido por el parámetro *template* falla, la especialización se descarta del conjunto de sobrecargas en lugar de causar un error de compilación. Este feature es usado en *template metaprogramming*.

Explicación: Los parámetros de las funciones *templates* son sustituidos dos veces:

- los argumentos del *template* explícitamente especificados son sustituidos antes de la *template*

argument deduction.

- los argumentos deducidos y los argumentos obtenidos son sustituidos antes de la *template argument deduction*.

La sustitución desde $C++11$ ocurre en:

- todas las expresiones usadas en la función tipo
- todas las expresiones usadas en una declaración de parámetros de *templates*

Una falla de sustitución es cualquier situación en la que el tipo o la expresión anterior estaría mal formada si es escrita usando los argumentos sustitutos.

Solo los fallos en los tipos y expresiones en el contexto inmediato de la función tipo o sus tipos de parámetros *templates* son errores **SFINAE**.

Con el uso de *templates* el proceso de compilación se puede ver comprometido por su mal uso. Ejemplo:

```
/*Funcion de dos parametros que devuelve el casteo del segundo
al tipo del primero.*/
template<typename T, typename U>
T cast(U x)
{
  return static_cast<T>(x);
}
/*Funcion de dos parametros que devuelve el casteo de un subtipo del
segundo al tipo del primero.*/
template<typename T, typename U>
T cast(typename U::other_type x)
{
  return static_cast<T>(x);
}
int main ()
{
  std::cout << cast<int,float>(12.56) << endl;
  return 0;
}
```

La razón por la que esto funciona es que el compilador realiza una búsqueda entre todos los *templates* de " cast", enfocándose en que exista uno que cumpla con los requisitos de argumentos, no en reportar que alguno no lo hace.

```
template<typename T, typename U>
U cast(T x)
{
  return static_cast<U>(x);
}
```

```
template<typename T, typename U>
T cast(U x)
{
  return static_cast<T>(x);
}
```

Aquí el compilador busca un *template* que acepte la sustitución de tipos y cumpla con los requisitos de los argumentos, pero en caso de encontrar más de uno con la misma "prioridad de sustitución"no podrá saber como resolver la ambigüedad.

Si se utilizan *templates* este problema se puede resolver en compilación sin afectar la ejecución, es mejor sacrificar tiempo una vez que todas las veces.

```
void f(int i)
{
std::cout << "int": << "\n";
}
void f(char c)
{
std::cout << "char" << "\n";
}
```

6. Constexpr

Es un feature añadido en $C++11$ y mejorado en $C++14$. La palabra clave **Constexpr** especifica que el valor de un objeto o una función puede ser evaluado en tiempo de compilación y la expresión puede ser usada en otras expresiones constantes.

La idea principal es desarrollar mejoras a los programas haciendo cálculos en tiempo de compilación antes que en tiempo de ejecución. Note que una vez que un programa es compilado y finalizado por el desarrollador, se corre múltiples veces por los usuarios. La idea es gastar el mayor tiempo posible en compilación y ahorrar tiempo en ejecución (similar a **TMP**).

Por ejemplo, en el siguiente código *product*() es evaluado en tiempo de compilación:

```
constexpr int product(int x, int y)
{
  return (x * y);
}
int main()
{
  const int x = product(10, 20);
  cout <<  x;
  return 0;
}
```

Otro ejemplo válido para $C++11$ que se resuelve en tiempo de compilación es el siguiente:

```
constexpr int six() { return 6;}
int arrayten[six() + 4];
```

Una función declarada como constexpr:

- En $C++11$ una función *constexpr* contiene solo un valor de retorno. En $C++14$ permite más de una sentencia.
- Una función *constexpr* debe referenciar solo a variables constantes globales.
- Una función *constexpr* puede llamar solo a otras funciones *constexpr*, no a funciones simples.
- Una función *constexpr* no puede ser de tipo *void*.
- El cuerpo debe ser de la forma "return $<$ *expr* $>$".
- Algunos operadores como incremento prefijo $(++x)$ no son permitidos.

Ventaja de las funciones constexpr:

Estas funciones tienen la ventaja de que si el compilador puede deducir todas las dependencias en el llamado de estas entonces también será capaz de determinar el valor en tiempo de compilación. No obstante, en caso contrario tratará la función como una calculada en tiempo de ejecución. Con esto se evita la declaración doble de una misma función.

Por ejemplo, el siguiente programa calcula en tiempo de compilación *exp*1, sin embargo en *exp*2 da error en compilación:

```
#include<iostream>
using namespace std;
constexpr float exp(float x, int n)
{
  return n == 0 ? 1 : n % 2 == 0 ? exp(x * x, n/2) :exp(x*x,(n-1)/2)*x;
}
float doble(float x)
{
  return 2 * x;
}
int main()
{
  float a = doble(1);
  constexpr float exp1 = exp(2, 10);
  constexpr float exp2 = exp(a, 10);
  cout << exp1 << endl;
  cout << exp2 << endl;
  return 0;
}
```

Si no se declara *exp*2 como *constexpr* no surgirá el error anterior, pues no se ve obligada a calcular su valor en compilación, veamos el ejemplo con dicha modificación:

```
int main()
{
  float a = double(1);
  constexpr float exp1 = exp(2, 10);
  float exp2 = exp(a, 10);
  cout << exp1 << endl;
  cout << exp2 << endl;
  return 0;
}
```

Constexpr vs Inline Functions:

Por ejemplo, veamos Fibonacci usando *constexpr*:

```
#include <iostream>
using namespace std;
constexpr long int fib(int n)
{
  return (n <= 1) ? n : fib(n - 1) + fib(n - 2);
}
int main()
{
  const long int result1 = fib(30);
  cout << result1 << endl;
  long int result2 = fib(30);
  cout << result2 << endl;
  return 0;
}
```

En el programa anterior el valor de *result*1 es computado en tiempo de compilación. Cuando se corre en GCC tarda 0,003 seg mientras que *result*2 se calcula en ejecución y tarda 0,017 seg. En *inline functions* las expresiones son evaluadas en tiempo de ejecución mientras que en *constexpr functions* en tiempo de compilación.

Constexpr with constructors(C++11):

Constexpr puede ser utilizado en constructores y objetos también, para instanciar clases y structs en tiempo de compilación. El cuerpo del constructor debe estar vacío y sus miembros deben inicializarse con expresiones constantes.

Por ejemplo, el siguiente programa muestra una clase con un constructor y una función *constexpr*, donde el objeto se inicializa en tiempo de compilación:

```
#include <bits/stdc++.h>
using namespace std;
class Rectangle
{
  int _h,_w;
  public:
    constexpr Rectangle(int h, int w) : _h(h), _w(w) {}
```

```
    constexpr int getArea() const { return _h  * _w; }
};
int main()
{
  constexpr Rectangle rectangle(10, 20);
  cout << rectangle.getArea();
  return 0;
}
```

Otro ejemplo de una clase que se puede instanciar en tiempo de compilación es el siguiente donde el constructor de la clase es una función *constexpr*:

```
class Point
{
  double _x, _y;
  public:
    constexpr Point(double x = 0, double y = 0) noexcept : _x(x), _y(y) {}
    constexpr  double getX() const noexcept { return _x; }
    constexpr double getY() const noexcept { return _y; }
};
int main()
{
  constexpr Point p(6,10);
  cout << p.getX();
  return 0;
}
```

El siguiente algoritmo calcula el punto intermedio entre dos puntos(*Point*) en tiempo de compilación con el uso de función *constexpr*.

```
constexpr Point intpoint(const Point& p1, const Point& p2)
{
  return Point((p1.getX() + p2.getX())/2, (p1.getY() + p2.getY())/2);
};
int main()
{
  constexpr Point p1(6,10);
  constexpr Point p2(12,20);
  constexpr Point p3 = intpoint(p1,p2);
  cout << "X: " << p3.getX() << endl;
  cout << "Y: " << p3.getY() << endl;
  return 0;
}
```

Constexpr vs Const:

Constexpr sigue una idea similar a *const* que tiene el objetivo de evitar el cambio de valor de una variable, pero se aplica además a funciones y constructores de clases. Aunque *constexpr* es más para optimización mientras que *const* es para hacer prácticos los objetos constantes como por ejemplo Pi.

Ambos se pueden aplicar a métodos, cuando se utiliza *const* se quieren prever cambios accidentales en estos mientras que cuando se utiliza *constexpr* la idea es computar las expresiones en tiempo de compilación y ahorrar así tiempo en ejecución. *Const* solo puede ser utilizado con $non-static$ *member functions* mientras que *constexpr* se puede utilizar con *member* y $non-member$ *functions*, y ambos pueden ser utilizados en constructores, pero con la condición de que los argumentos y el tipo de retorno sean tipos literales.

¿ Por qué no usar siempre *constexpr*?

Es fácil pensar que debido al doble comportamiento y a los múltiples contextos donde se prefieren las funciones y objetos *constexpr* estos deben utilizarse siempre que sea posible. No obstante, una vez que se declaran los objetos o funciones como *constexpr* y posteriormente no se quiere continuar con este comportamiento y se decide eliminar *constexpr*, esto puede provocar que el código no compile y acarrear consigo múltiples efectos secundarios. Por lo tanto, se deja a consideración del programador el uso o no de *constexpr* según lo que este desee.

7. C++11 vs C++14

7.1. Deducción de tipo de retorno

- $C++11$ permite automáticamente deducir el tipo de retorno de una función lambda cuyo cuerpo consiste en una sentencia simple.

  Ejemplo:

```
[=] () -> some_type { return foo() * 42; } // ok
[=]                                { return foo() * 42; }
// ok, deduce " -> some_type".
```

- En $C++14$ funciona con cuerpos de funciones más complejos que contienen más de una sentencia de retorno.

  Ejemplo:

```
[=]{              //ok , deduce " -> some_type".
while(something())
{
  if(exp)
  {
  return foo() * 42;  //con control arbitrario de flujo
  }
}
return bar.baz(84);      //multiples retornos
}            //los tipos deben ser los mismos
```

- $C++11$ permite automáticamente deducir el tipo de retorno de cualquier función cuyo cuerpo consiste en una sentencia simple.

  Ejemplo:

```
some_type f()                  { return foo() * 42; } // ok
auto f()  ->  some_type { return foo() * 42; } // ok
```

- En C++14

  Ejemplo:

```
auto f()    { return foo() * 42;}   //ok, deduce " -> some_type".
auto g()
{         //ok , deduce     -> some_type .
  while(something() )
  {
    if(exp)
    {
    return foo() * 42;  //con control arbitrario de flujo
    }
  }
  return bar.baz(84);     //m ltiples retornos
}           //los tipos deben ser los mismos
```

7.2. Decltype(auto)

Dadas estas funciones:

```
string lookup1();
string & lookup2();
```

- En $C++11$ podríamos escribir las siguientes "wrapper"funciones que recuerdan preservar la *referenceness* del tipo de retorno.

  Ejemplo:

```
string look_up_a_string_1() { return lookup1(); }
string& look_up_a_string_2() { return lookup2(); }
```

- En $C++14$ se puede automatizar esto.

  Ejemplo:

```
decltype(auto) look_up_a_string_1() { return lookup1(); }
// retorna string.
decltype(auto) look_up_a_string_2(){ auto str=lookup2();return(str);}
// retorna la referencia a la variable str(string&).
```

7.3. Capturas lambda

- En $C++11$ los lambdas no son fáciles de capturar, en $C++14$ tenemos las capturas lambdas generalizadas para resolver no solo ese problema sino para permitirte definir arbitrariamente nuevas variables locales en el objeto lambda.

Ejemplo:

```
auto u = make_unique<some_type> (some, parameters);
go.run ( [u = move(u) ] { do_something_with(u); } );
// mueve el puntero unique dentro del lambda.
```

En ese ejemplo mantuvimos el nombre de la variable u dentro del lambda, pero no estamos limitados a esto:

```
go.run ( [u2 = move(u) ] { do_something_with(u2); } );
//captura de u2
```

Se puede añadir además nuevos estados al objeto lambda, porque cada captura crea un nuevo tipo deducido de variable local dentro del lambda:

```
int x = 4;
int z = [ &r = x, y = x + 1]
{
  r += 2;                  // pone 6 en x.
  return y + 2;      // retorna 7 para inicializar z.
} ();
```

7.4. Lambdas genéricos

- En $C++11$ tenemos que establecer explícitamente el tipo de un parámetro lambda.

  Ejemplo:

```
For_each(begin(v),end(v),[]( decltype(*cbegin(v))x) {cout << x;});
Sort ( begin(w), end(w), [](const shared_ptr<some_type>& a,
const shared_ptr<some_type>& b) { return *a<*b; } );
auto size = [](const unordered_map<wstring,vector<string>>& m)
{return m.size();};
```

- En $C++14$ podemos deducir el tipo.

  Ejemplo:

```
for_each( begin(v), end(v), [](const auto& x) { cout << x; } );
sort(begin(w),end(w),[](const auto& a,const auto& b){return *a<*b;});
```

7.5. Templates variables

- En $C++11$ el uso de funciones *constexpr* ha reemplazado la necesidad de "*traits*" *templates*. Si se quiere computar un tipo entonces se prefiere usar un *template alias_t* $< T >$ en vez de *traits* $< T >$:: *type*, y si se quiere computar un valor entonces se prefiere usar una función *value_v*(); que es *constexpr* en vez de *traits* $< T >$:: *value*.

  Ejemplo:

```
template<typename T> constexpr T pi = T(3.14159265358979323846);
template<class T> T area_of_circle_with_radius(T r)
{return pi<T>*r*r;}
```

- En $C++14$:

  Ejemplo:

```
auto area_of_circle_with_radius = [](auto r)
{return pi<decltype(r)>*r*r;};
```

7.6. Constexpr extendido:

- En $C++11$ las estrictas limitaciones de las *constexpr* no fueron del agrado de muchos. Por ejemplo, si se quisiera extender el uso de las funciones *constexpr* para *strings*, esto sería muy bueno, pero esto requiere iterar sobre los caracteres del *string* lo cual no es permitido en las funciones *constexpr*. Además de las restricciones sobre el uso de las estructuras condicionales que se ven sujetas al operador ? : y los ciclos que se manejan a través de la recursividad.

  Ejemplo:

```
constexpr int my_charcmp( char c1, char c2 )
{
  return (c1 == c2) ? 0 : (c1 < c2) ? -1 : 1;
}
constexpr float exp(float x, int n)
{
  return n==0?1:n%2 == 0?exp(x*x, n/2):exp(x*x,(n-1)/2)*x;
}
```

  ¿ Hasta dónde es posible usar *constexpr* en $C++14$?

- En $C++14$ se pueden definir y modificar variables en el cuerpo de las funciones (con la condición de que su ciclo de vida se mantenga dentro de la función y estas no sean estáticas ni locales a hilos). Pasan además a ser permitidas las estructuras de control de flujo como if, $while$, for, $do-while$ y $switch$ dentro del cuerpo de la función.

  Ejemplo:

```
constexpr int my_strcmp( const char* str1, const char* str2 )
{
  int i = 0;
  for( ; str1[i] && str2[i] && str1[i] == str2[i]; ++i ){ }
  if( str1[i] == str2[i] ) return 0;
  if( str1[i] < str2[i] ) return -1;
  return 1;
}
```

- En $C++11$ no se pueden declarar métodos como *constexpr*. Por ejemplo, si a la clase *Point* se le desea adicionar los métodos *setX* y *setY*, no se puede pues estos tienen tipo de retorno *void*,

que no es un tipo literal en $C++11$ y además porque modifican al objeto y aquí las funciones de este tipo son implícitamente *const*.

- En $C++14$ se pueden declarar como *constexpr* los métodos *setX* y *setY* de la clase *Point*.

  Ejemplo:

```
class Point
{
  double _x, _y;
public:
  constexpr Point(double x = 0, double y = 0) noexcept:_x(x),_y(y){}
  constexpr  double getX() const noexcept { return _x; }
  constexpr double getY() const noexcept { return _y; }
  constexpr void setX(double X) noexcept {x = X;}
  constexpr void setY(double Y) noexcept {y = Y;}
};
```

8. Nuevas propuestas de C++17

- Structured bindings/Decomposition declarations.
- Init-statement for if/switch.
- Inline variables.
- Constexpr if.
- Fold expressions.
- Template Argument Deduction for class template.
- Declaring non-type template parameters with auto.

$C++17$ ofrece mejoras a las estructuras de control de flujo if y *switch*. Te permite declarar variables dentro del propio if que serán usadas en la condición del mismo. Se pueden escribir códigos de manera compacta y además dictamina que el *scope* de la variable inicializada es precisamente el cuerpo del if. Lo cual es muy importante para este tipo de lenguajes que dan tanta importancia a conceptos como tiempo de vida de las variables y *scope*.

Antes se escribía:

```
auto val = GetValue();
if (condition(val))
//on success
else
//On false
```

Ahora se puede hacer:

```
if(auto val = GetValue(); condition(val))
//on success
else
//on false
```

Nota: *val* es solo visible dentro del cuerpo del *if* y el *else*.

Veamos el caso del *switch*, ahora podemos hacer:

```
switch (initial-statement; variable)
{
  ....
  // cases
}
```

Veamos otros ejemplo:

Digamos que quieres buscar algo dentro de un string:

```
const std::string myString = "My Hello World";

const auto it = myString.find("Hello");
if (it != std::string::npos)
std::cout << it << " Hello\n"

const auto it2 = myString.find("World");
if (it2 != std::string::npos)
std::cout << it2 << " World\n"
```

Tendríamos que usar nombres diferentes y *scopes* separados. Ahora con la nueva sentencia se puede poner ese *scope* adicional en una sola línea y como la variable es visible también dentro del bloque *else*, quedaría así:

```
if (const auto it = myString.find("Hello"); it != std::string::npos)
std::cout << it << " Hello\n";
else
std::cout << it << " not found!!\n";
```

Un ejemplo bien básico del uso del *switch* es el siguiente:

```
switch (int i = rand() % 100; i)
{
  default:
  cout << "i = " << i << "\n"; break;
}
```

8.1. Class Template Argument Deduction

$C++17$ trae nuevas atractivas propuestas como lo es la adición de la clase *Template Argument Deduction* basada en *template argument deduction for function templates* disponible desde $C++$

+98.

Problemática que intenta resolver:

EL problema surge al crear un objeto determinado y tener que especificar el tipo de sus argumentos, por ejemplo:

```
std::pair<int,double> myPair(10, 0.0);
```

Deducir los tipos de los argumentos era el principal objetivo. Antes de $C++17$ se resolvió este problema con las funciones $make\_XXX$ que utiliza $template\ argument\ deduction\ for\ functions$ para determinar los tipos de los argumentos de la clase que se desea construir. Luego tendríamos algo como esto:

```
auto myPair = std::make_pair(10, 0.0);
```

Pero esta solución no fue de mucho agrado y estaba propensa a errores si no se declaraba una función para su uso. Ahora en $C++17$ no tenemos este problema:

```
std::pair myPair(10, 0.0);
```

¿Cómo funciona?:

Dada una función *template* declarada así:

```
template<class T>
int Function(T object);
```

Un usuario puede invocar esta función especificando el tipo de *template* de este modo:

```
int result = Function<float>(100.f);
```

Pero el tipo especificado es opcional, pues el compilador puede deducir el tipo de T por el tipo de argumento suministrado, así:

```
int result = Function(100.f);
```

Digamos que queremos un *template* más complicado:

```
template<class T, T* object>
int Function();
```

La función se puede llamar de este modo:

```
static float val = 100.f;
//
int result = Function<float, &val>();
```

Existe algo llamado *template deduction guides* que no son más que patrones asociados a la clase *template* que le dice al compilador como traducir un conjunto de parámetros en los argumentos del *template*. El compilador crea un conjunto imaginario de constructor $function\ templates$ llamados *deduction guides*, estas no son realmente creadas o llamadas, sino que constituyen un concepto de como el compilador elige el constructor correcto para la creación del objeto.

Por ejemplo *std* :: *vector* cuyo constructor toma un par iterable:

```
template<typename Iterator>
void func(Iterator first, Iterator last)
{
  vector v(first, last);
}
```

El compilador necesita saber el tipo de T de $vector < T >$. Nosotros sabemos que la respuesta es:

```
typename std :: iterator_traits<Iterator>::value_type
```

¿Pero cómo le decimos eso al compilador sin tener $vector < typename\ std :: iterator\_traits < Iterator >:: value\_type >$? Usando deduction guides:

```
template<typename Iterator> vector(Iterator b,Iterator e)
->vector<typename std::iterator_traits<Iterator>::value_type>;
```

Esto le dice al compilador que llame al constructor de vector y matchea ese patrón, así se deduce la especialización del vector usando el código de la parte derecha de ->. Se necesitan guías cuando la deducción del tipo de los argumentos no depende de uno solo de esos argumentos. La parte izquierda no especifica necesariamente un constructor. La forma en que funciona es, si necesita usar *template constructor deduction* en un tipo, entonces matchea los argumentos que se pasan contra todas las *deduction guides*. Si hay un match entonces la usa para determinar que *template arguments* proveerle al tipo. Por ejemplo:

Cualquier función *template* ambigua como la siguiente:

```
Template<typename T>
returnType function(paramType param);
```

puede ser llamada de este modo:

```
function(expression);
```

El compilador usa *expression* para determinar el tipo de T y el tipo de *paramType*. Esto es porque a menudo *paramType* contiene decoradores como *const*, *const*&, *const*&&, etc. Se podría pensar que el tipo de T que el compilador deduce es el mismo de *expression*, pero no es siempre el caso. La deducción de T depende tanto de *expression* como de *paramType*.

Conclusiones

- Tiene su base en $function\ argument\ deduction$.
- Nos permite deshacernos del uso de las funciones $make\_XXX$.
- Ganamos en limpieza y legibilidad.
- Las buenas prácticas con la nueva herramienta se supone son similares a las usadas con $argument\ deduction\ for\ function$: utilizarla por defecto, a no ser que se sepa que los parámetros no podrán ser deducidos o que se vaya a perder legibilidad en el código.
- Las *deduction guides* solo son utilizadas para determinar el tipo a inicializar.

8.2. Otras propuestas

- **Fold expressions:** Permite escribir código compacto con *templates* sin usar recursión explícita.

  Ejemplo:

```
template<typename... Args>
auto SumWithOne(Args... args)
{
  return(1 + ... + args);
}
```

- **Aggregate initialization of clases with base classes:** Si una clase fue derivada de otro tipo no se podía usar inicialización agregada. Pero ahora esa restricción fue removida.

  Ejemplo:

```
struct base {int a1, a2;};
struct derived : base {int b1;};

derived d1{{1,2},3};  //completa inicializacion explicita
derived d1{{},1}; //la base es un valor inicializado
```

- **Non-type template parameters with auto type:** Deduce automáticamente los parámetros del *template*.

  Ejemplo:

```
template<auto value> void f() {}
f<10>();   //deduce int
```

- **Direct-list-initialization of enumerations:** Permite inicializar *enum class* con *fixed underlying type*. Permite crear fuertes tipos fáciles de usar.

  Ejemplo:

```
enum class Handle: uint32_t {Invalid = 0};
Handle h { 42};       //ok
```

- **Constexpr lambda expressions:** *Constexpr* ahora puede ser usado en contextos lambdas.

  Ejemplo:

```
constexpr auto ID = [](int n){ return n;};
constexpr int I = ID(3);
static_assert (I == 3);

constexpr int AddEleven(int n)
{
  return [n]{ return n + 11;}();
}
```

- **Binding Declarations:**

  Ejemplo:

  Antes era:

```
int a = 0;
double b = 0.0;
long c = 0;
std::tie(a,b,c) = tuple
```

  nota: a, b y c necesitaban ser declaradas antes.

  Ahora:

```
auto [a, b, c] = tuple;
```

- **Inline variables:** Antes solo los métodos o funciones podían ser declarados *inline*, ahora también lo pueden las variables.

  Ejemplo:

```
struct MyClass
{
  Inline int const sValue = 777;
};
```