Lenguaje de programación C++

Templates

¿Qué son los templates?

Cuando Bjarne Stroustrup diseña C++ expresa (en 1986) que sería deseable diseñar dos características en el futuro del lenguaje: Los Templates y las Excepciones.

Por templates buscaba expresar la parametrización de una clase contenedora, y la excepción el manejo de errores en tiempo de ejecución.

Cuando realmente se implementó, luego de la estandarización, el Template fue más allá que una herramienta de parametrización de contenedor, finalizó siendo una de las características más importantes de C++ al permitir una programación genérica

¿Cuando debemos implementar o usar Templates?

Son varios los motivos o momentos en que se deben usar, uno de ellos es cuando tenemos funciones importantes y debemos duplicarlas o triplicarlas solo para que realicen lo mismo para diferentes tipos. Ejemplo:

```
int maximo(int x, int y){
    return (x >= y) ? x : y;
}

char maximo(char a, char b){
    return (a >= y) ? a : b;
}

double maximo (double x, double y){
    return (x >= y) ? x : y;
}
```

Estas tres funciones podrían ser reemplazadas por una:

```
template<class T>
    const T& maximo (const T& a, const T& b){
    return (a >= b) ? a : b;
}

int main(){
    cout << "int mayor: " << maximo(4, 6) <<endl;

}
```

Uso del template

C++ va a inferir el tipo de dato. Hay que tener en cuenta, que T será reemplazado por el mismo tipo dentro de la función. Y si se usara un tipo de dato abstracto creado por uno, los operadores tendrán que estar sobrecargados para que pueda saber como usarlos.

El código que se genera al compilar si va a poseer tantas funciones maximo como tipo se utilicen dentro del programa. Realmente funciona como una plantilla..

Particularizando al template

Hasta ahora vimos el uso de tipos primitivos con templates, veamos este ejemplo:

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

template<class T> const T& maximo(const T& a, const T& b){
    return (a >= b) ? a : b;
}
int main (){
 cout << "con int ( 5 - 10):" << maximo (5, 10) << endl;
 cout << "con string: " << maximo ("hola", "chauchau") << endl;
 return 0;
}
```

El mensaje de compilación será el siguiente:

Particularización

```
$ g++ -Wall -std=c++11 -otemplateString plantillaString.cpp
plantillaString.cpp: En la función ‘int main()’:
plantillaString.cpp:11:54: error: no hay una función coincidente para la
llamada a ‘maximo(const char [5], const char [9])’
   11 |  cout << "con string: " << maximo ("hola", "chauchau" ) << endl;
      |                                                      ^
plantillaString.cpp:5:28: nota: candidate: ‘template<class T> const T&
maximo(const T&, const T&)’
    5 | template<class T> const T&  maximo(const T& a, const T& b){
      |                             ^~~~~~
plantillaString.cpp:5:28: nota:   falló la deducción/sustitución del argumento
de plantilla:
plantillaString.cpp:11:54:  nota:   se deducen tipos en conflicto para el
parámetro ‘const T’ (‘char [5]’ y ‘char [9]’)
   11 |  cout << "con string: " << maximo ("hola", "chauchau" ) << endl;
      |
```

Particularización

Vemos que C++ no logró reemplazar un patrón, interpreta correctamente que la cadena pasada es un arreglo de caracteres y por lo tanto una dirección de memoria, para corregir esto particularizaremos al template:

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

template<class T> const T& maximo(const T& a, const T& b){
    return (a >= b) ? a : b;
}
const char* maximo(const char* a, const char* b){
   if (strcmp(a, b) > 0)
     return a;
   else
     return b;
}
```

Particularización

```
int main (){

 cout << "con int ( 5 - 10):" << maximo (5, 10) << endl;
 cout << "con string: " << maximo("Hola", "zzChau") << endl;
 return 0;
}
```

Podemos compilar y ejecutar el programa:

```
$ g++ -Wall -std=c++11 -otemplateCadena plantillaCadena.cpp
$ ./templateCadena
con int ( 5 - 10):10
con string: zzChau
$
```

Particularización

Con el caso de los strings podemos evitar el problema si usamos los objetos que nos da C++

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

template<class T> const T& maximo(const T& a, const T& b){
    return (a >= b) ? a : b;
}

int main (){
 string a("hola");
 string b("zzzchauchau");
 cout << "con int ( 5 - 10):" << maximo (5, 10) << endl;
 cout << "con string: " << maximo (a, b) << endl;
 return 0;
}
```

Particularización

Compilamos y ejecutamos

```
$ g++ -Wall -std=c++11 -otemplateString plantillaString.cpp
$ ./templateString
con int ( 5 - 10):10
con string: zzzchauchau
$
```

Otra forma aceptada por el Standard C++11 que tiene el mismo efecto que el ejemplo anterior

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

template<class T> const T& maximo(const T& a, const T& b){
    return (a >= b) ? a : b;
}
int main (){
 cout << "con int ( 5 - 10):" << maximo (5, 10) << endl;
 cout << "con string: " << maximo<std::string>("Hola", "zzChau") << endl;
 return 0;
}
```

Ejemplo

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

template<typename C> void sonIguales(const C& a, const C& b){
    (a == b) ? cout<<"Son iguales\n" : cout<<"Son distintos\n";
}
class Persona{
    string nombre;
    int edad;
public:
    Persona(string n, int e):nombre{n}{edad = e;}
    bool operator==(const Persona& P)const{
      return (edad == P.edad && nombre == P.nombre);
    }
    friend ostream& operator<<(ostream& os, const Persona& P);
};
ostream& operator<< (ostream& os, const Persona& P){
    os<< "Nombre: " << P.nombre << endl;
    os<< "Edad: " << P.edad << endl;
    return os;
}
```

Ejemplo (continuación)

```
int main(){
    Persona a("Pablo", 44), b("Martin", 35), c("Pablo", 44);
    sonIguales(a, b);
    sonIguales(a, c);
    cout<< a;
    return 0;
}
```

Compilamos y probamos

```
$ g++ -Wall -std=c++11 -otemplateEjemplo templateEjemplo.cpp
$ ./templateEjemplo
Son distintos
Son iguales
Nombre: Pablo
Edad: 44
```

Múltiples parámetros

¿Qué ocurre cuando necesitamos dos tipos de parámetros diferentes?

Simplemente declaramos dos tipos distintos

```
template<typename T1, typename T2>
void f (const T1& t1, const T2& t2){

    //código de la función
}
```

Algo que no mencionamos en el ejemplo anterior, hemos cambiado la palabra clave class por typename, C++ permite esto funcionando de la misma forma.

Templates de clases

Con el mismo sentido que antes, pero aquí resulta una opción mucho más trascendente ya que una clase no puede ser sobrecargada, por lo tanto el hacerla genérica es muy útil.

Veamos un ejemplo: Quiero tener un stock de mercaderías, de ellas solo guardo dos datos, el nombre y la unidad de medida. La representación del nombre es simple, con un string nos alcanzaría, pero la unidad de medida podría ser entera, o una unidad decimal. Para representar esto, el template nos ayudaría de la siguiente forma:

Template de Clase

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

template<typename T>
class Mercaderia{
  string nombre;
  T medida;
public:
  Mercaderia(string n, T m): nombre{n}, medida{m} {}
  string get_nombre() const {return nombre;}
  T get_medida() const {return medida;}
  template<class U>friend ostream& operator<< (ostream& os, const Mercaderia<U>& M);

};
template<class U>
ostream& operator<< (ostream& os, const Mercaderia<U>& M){
  os<< "Mercaderia: " << M.nombre << "-->cantidad: " << M.medida << endl;
  return os;
}
```

Vemos que la sobrecarga de la función friend debe estar bajo template también para la definición de Mercadería.

Ejemplo (continuación)

```
int main (){
  Mercaderia<int> tornillos("tornillos", 150);
  Mercaderia<float> kerosene("kerosene", 256.5);

  cout << tornillos << kerosene;

  return 0;
}
```

Compilamos y probamos

```
$ g++ -Wall -std=c++11 -otemplateClases templateClases.cpp
$ ./templateClases
Mercaderia: tornillos-->cantidad: 150
Mercaderia: kerosene-->cantidad: 256.5
$
```