uc3m | Universidad Carlos III de Madrid

# Informática Industrial I

Grado en Ingeniería en Electrónica Industrial y Automática

Álvaro Castro González

Nicolas Burus

Mohamed Abderrahim

José Carlos Castillo Montoya

## Práctica 6

## Plantillas

A lo largo de los últimos años, han colaborado en impartir las prácticas presenciales, y por tanto proporcionar comentarios para mejorar algunos aspectos de este documento los profesores siguientes:

Ioannis Douratsos
Fares Abu-Dakka
Juan Carlos González Victores
Raúl Perula
Avinash Ranganath

# Práctica 6 – Plantillas.

Las plantillas es una de las características más poderosas y sofisticadas de C++. Las plantillas permiten crear **funciones y clases genéricas** que se pueden aplicar a distintos tipos de datos sin tener que modificar el código para cada tipo.

## 1. Funciones genéricas

*Una función genérica es un conjunto de operaciones que se aplicarán a distintos tipos de datos. El tipo de dato sobre el que la función operará se pasa como parámetro. En esencia, cuando se crea una función genérica se está creando una función que se sobrecarga así misma automáticamente.*

Para definir una plantilla se utiliza la palabra reservada `template` y la forma general de definir una plantilla para una función es:

```
template <class Type,...> return_type name_function(parameter list) {
      //body of function
}
```

`Type` es una referencia al tipo de dato que se usará dentro de la función tanto para los parámetros de entrada como el de salida. El compilador lo sustituirá por el tipo de dato real que corresponda en cada llamada.

A continuación se muestra un ejemplo en el que se crea una función genérica que intercambia los valores de dos variables. Puesto que el proceso para intercambiar dos valores es independiente del tipo de datos de las variables, es un ejemplo que se ajusta muy bien para el uso de plantillas.

```
#include <iostream>

using namespace std;

template <class Type>
void swaping(Type & a, Type & b){
      Type aux = a;
      a = b;
      b = aux;
};

int main(int argc, char *argv[]) {
      int i = 10, j = 20;
      double x = 10.2, y = 43.11;
      char a = 'v', b = 's';

      cout << "Original i, j: " << i << ' ' << j << '\n';
      cout << "Original x, y: " << x << ' ' << y << '\n';
      cout << "Original a, b: " << a << ' ' << b << '\n';

      swaping<int>(i,j);
      swaping<double>(x,y);
      swaping<char>(a,b);
```

```
	cout << "Swaped i, j: " << i << ' ' << j << '\n';
	cout << "Swaped x, y: " << x << ' ' << y << '\n';
	cout << "Swaped a, b: " << a << ' ' << b << '\n';

	return 0;
}
```

En este ejemplo, Type es el tipo genérico que representa el tipo de dato de los valores que serán intercambiados. El compilador lo sustituirá por int, double y char según cree las instancias necesarias de la plantilla de la función.

**¿Cuál es la salida?**

En una plantilla se pueden usar más de un tipo genérico de datos (de ahí los puntos suspensivos en la definición genérica de una plantilla de función). Implementa la función genérica que muestre los valores de dos variables de distintos tipos que se le pasan como parámetro. Prueba que funciona. La cabecera de la función genérica sería:

```
...
template<class tipo1, class tipo2> void print(tipo1 a, tipo2 b);
...
```

Recordando la práctica anterior, al igual que ocurre con las clases, las plantillas también se pueden sobrecargar. Además, también se pueden mezclar tipos de datos estándar con tipos genéricos de datos.

Continuando con el último ejemplo implementa la función genérica que se corresponde con:

```
...
template<class tipo1, class tipo2> void print(tipo1 a, tipo2 b, double c);
...
```

¿Qué ocurre si ejecutas `print("cadena de texto", 2.5, 'a');`? ¿Por qué?

Resumiendo, las funciones genéricas deben realizar la misma acción general para todas las versiones, sólo diferenciándose en el tipo de dato. En contraste, la sobrecarga de funciones "normales" permite que cada implementación se comporte de una forma completamente diferente.

## 2. Clases genéricas

Una clase genérica define todos los algoritmos que una clase va a usar y el tipo de dato que va a manipular se especificará como un parámetro cuando se instancie la clase. Una clase genérica es útil cuando usa una lógica que se puede generalizar.

La forma general de definir una clase genérica es la siguiente:

```
template <class Type,...> class class_name {
      ...
}
```

Lo mencionado sobre `Type` para funciones genéricas también aplica para clases genéricas.

Para crear un objeto de la clase genérica se realiza de esta forma:

```
class_name <type> object;
```

Donde `type` es el tipo de dato que manejará la clase. A continuación se puede observar un ejemplo de la implementación de la estructura de datos pila funcionando con todo tipo de datos. ¿Hay algún error en el código? Encuéntralo.

```
#include <iostream>

using namespace std;

template<class Type>
class Stack {
    public:
        Stack() : _size(10), _index(0) { // another way to initialize variables
            _data = new Type[_size];
        }
        void push(Type data);
        Type pop();

    private:
        int _size, _index; // stack size
        Type *_data;
};

template<class Type>
void Stack<Type>::push(Type data) {
      if(this->_index == _size) {
            cout << "Full stack" << endl;
      }
      else {
            this->_index++;
            this->_data[this->_index] = data;
      }
}

template<class Type>
Type Stack<Type>::pop() {
      if(this->_index == 0) {
            cout << "Empty stack" << endl;
            return 0;
      }
      this->_index--;
```

```
      return this->_data[this->_index];
}

int main() {
      Stack<char> S;
      S.pop();
      S.push('z');
      S.push("q");
      S.push(2.1);
}
```

En la especificación de una plantilla de clase también se puede utilizar lo que normalmente se piensa que es un argumento estándar. Por ejemplo, en el ejemplo anterior, si quisiéramos que el tamaño de la pila pudiera definirse en la instanciación de un objeto de la plantilla de la clase se haría como sigue:

```
...
template<class T, int size> class Stack {
      T data[_size];
}
...
```

¿Qué habría que cambiar en la implementación de las funciones miembro de la clase genérica?

**Nota**: hay que tener en cuenta que en C++ sólo se permiten utilizar "argumentos estándar" en las plantillas de los tipos entero, puntero o referencia y son tratados como constantes.

## 3. STL

La Biblioteca de Plantillas Estándar (del inglés, Standar Template Library) de C++ es una biblioteca que proporciona clases de propósito general para implementar las estructuras de datos más conocidas y usadas. Están basadas en plantillas de clases y funciones. En la STL existen tres elementos principales: los contenedores que almacenan objetos, los algoritmos que actúan sobre los contenedores, y los iteradores que permiten recorrer los contenedores de diversas formas.

Antes de desarrollar propias estructuras de datos y algoritmos (como se hizo con el ejemplo de la pila) asegurar que no se puede utilizar alguna de las STL ya existentes. Como ejemplo se va a ver como se trabaja con un vector de elementos y sus operaciones básicas.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
      // creating vector with size 10
    unsigned int i;
      vector<char> myvector(10);
      cout << "size = " << myvector.size() << endl;

      // assign elements and showing
```

```
	for(i=0;i<10;i++) {
		myvector[i] = 'z' - i;
	}
	for(i=0;i<myvector.size();i++) {
		cout << myvector[i] << ' ';
	}
	cout << endl << endl;

	// add new elements if necessary
	for(i=0;i<10;i++) {
		myvector.push_back('z'-i-10);
	}
	for(i=0;i<myvector.size();i++) {
		cout << myvector[i] << ' ';
	}
	cout << endl << endl;

	// using an iterator
	vector<char>::iterator it;
	it = myvector.begin();
	while(it != myvector.end()) {
		*it = toupper(*it);
		it++;
	}
	for(i=0;i<myvector.size();i++) {
		cout << myvector[i] << ' ';
	}
	cout << endl <<endl;

	// sorting the vector
	sort(myvector.begin(), myvector.end());
	for(i=0;i<myvector.size();i++) {
		cout << myvector[i] << ' ';
	}
	cout << endl << endl;

	cin.get(); // wait an INTRO

	return 0;
}
```

**Recordad:**

Es muy útil mientras se programa el autocompletado de código (Ctrl+Espacio), de este modo se podrán ver todas las posibilidades disponibles.

Existen multitud de fuentes donde se puede encontrar más información sobre las STL. Por ejemplo se puede consultar cualquiera de los libros recomendados en la asignatura o alguno de los siguientes enlaces:

- http://www.cplusplus.com/reference/stl/
- http://www.cppreference.com/wiki/stl/start

## Ejercicios

Puesto que las plantillas se pueden aplicar prácticamente a cualquier tipo de dato, incluidos los definidos por uno mismo, aplicar lo aprendido al simulador de ordenador de las prácticas anteriores.

Crear una lista de todas las cadenas que le llegan al procesador. Cuando le llegue la cadena `history` mostrar por pantalla el historial de todas las cadenas que haya recibido.