Programación Genérica

(plantillas-templates)

Según va aumentando la complejidad de nuestros programas y sobre todo, de los problemas a los que nos enfrentamos, descubrimos que tenemos que repetir una y otra vez las mismas estructuras de datos.

La programación genérica también es conocida como **tipos parametrizados, plantillas o templates**, debido a que permite que se especifique uno o más parámetros de tipo, lo cual vuelve mucho mas flexible la programación.

Por tanto Las plantillas (templates) nos permiten parametrizar clases para adaptarlas a cualquier tipo de dato.

C++ permite crear plantillas de métodos y plantillas de clases.

¿ Qué es programación genérica ?

- Es un mecanismo el cual permite que un tipo pueda ser utilizado como parámetro en la definición de una clase o una función.

¿Para qué sirven?

- Sirve para utilizar una clase la cual manipule X tipo de datos.
- El objetivo principal de las plantillas es la reutilización.

Nota: Cuando se trabaja con plantillas la implementación de los métodos debe hacerse forzosamente en el .h

Ejemplo 1:
En este ejemplo se aplica una plantilla a un método, específicamente al método ***test*** de la clase **prueba.**

```
//--------------------------estudiante.h---------------------------------------
#pragma once
#include<iostream>
#include<string>
using namespace std;

class estudiante
{
	string nombre;
public:
	estudiante(string n) { nombre = n; }
	string getNombre() { return nombre; }
};
ostream& operator<<(ostream& salida, estudiante& a) {
	salida << a.getNombre();
	return salida;
}

//--------------------prueba.h-------------------------
class prueba {
public:
	template < typename T> void test(T);
};

template < typename T>
void prueba::test(T x) { cout << "Valor recibido: " << x << endl; }
//----------------------main.cpp-------------------------

void main()
{
	prueba p;
	p.test<char>('a');
	p.test<int>(1);
	p.test<double>(2.4);
	p.test<bool>(true);
	p.test<estudiante>(estudiante("Juan"));
	p.test<estudiante>(estudiante("Maria"));

	cout << endl;
	// otra version:
	p.test('a');
	p.test(1);
	p.test(2.4);
	p.test(true);
	p.test(estudiante("lapiz"));
	p.test(estudiante("portafolio"));

	cin.get();
}
```

Ejemplo 2:
En este ejemplo también se aplica una plantilla a un método.

```
//--------------------------------experimento.h-------------------------------------------
#include<iostream>
#include<string>
using namespace std;

class experimento {
public:
	template <typename T> T probar(T n);
	template< typename T> void imprimirVector(T* v, int n);
};

template <typename T> T experimento::probar(T n) {
	cout << "Se recibio el valor:" << n << endl;
	return (n);  //retorna un dato de tipo T
}

template < typename T>  void experimento::imprimirVector(T* v, int n) {//funcion generica
	for (int i = 0; i < n; i++){
		cout << v[i] << " - ";
	}
	cout  << endl;
}
//-----------------------------------main.cpp----------------------------------------


void main()
{
	experimento* p = new experimento();
	int x = 6;
	double y = 4.37;
	string z = "Hola";
	char w = 'a';
	int x2;
	double y2;
	string z2;
	char w2;

	x2 = p->probar<int>(x); // se especifica el tipo
	y2 = p->probar<double>(y); // se especifica el tipo
	z2 = p->probar(z);// En este caso, el compilador deduce el tipo
	w2 = p->probar(w);// En este caso, el compilador deduce el tipo

	cout << endl << endl << "Llamado al metodo imprimir vector: " << endl << endl;
	int a[] = { 1,2,3,4,5 };
	p->imprimirVector<int>(a, 5);

	double b[] = { 1.1,2.2,3.3,4.4,5.5 };
	p->imprimirVector<double>(b, 5);

	char c[] = "ADIOS";
	p->imprimirVector<char>(c, 5);

	string d[] = { "VIVA", "LA", "VIDA" };
	p->imprimirVector(d, 3);// no se especifico el tipo

	cin.get();
```

```
}
```

Ejemplo 3a:
En esta primera versión del ejemplo NO se usan plantillas

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

class Contenedor {
private:
	int* v; // vector de enteros
	int n;
public:
	Contenedor(int nElem);
	~Contenedor();
	int& operator[](int);
};

Contenedor::Contenedor(int nElem) :n(nElem){
	v = new int[n];
}

int& Contenedor::operator[](int indice){
	return v[indice];
}

Contenedor::~Contenedor(){
	delete[] v;
}

//-----------------------------------------------------------------


void main() {
	Contenedor myContenedor(10);

	for (int i = 0; i < 10; i++)
		myContenedor[i] = 10 - i;

	for (int i = 0; i < 10; i++)
		cout << myContenedor[i] << endl;

	cin.get();
}
```

Ejemplo 3b:
Este es el mismo ejemplo anterior pero utilizando plantillas.

Observe que las plantillas esta vez se aplican directamente a toda la clase.

Cuando se realiza la instanciación de una clase parametrizada, se debe indicar explícitamente el tipo de dato.

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

//--------------------------------coleccion.h-------------------------------------
template <class T>
class coleccion {
private:
	T* v; // vector dinámico con objetos automáticos
	int n;
public:
	coleccion(int nElem);
	~coleccion();
	T& operator[](int);
};

template <class T>
coleccion<T>::coleccion(int nElem) : n(nElem) {
	v = new T[n];
}

template <class T>
T& coleccion<T>::operator[](int indice) {
	return v[indice];
}

template <class T>
coleccion<T>::~coleccion() {
	delete[] v;
}

//--------------------------estudiante.h-----------------------------------


class estudiante
{
	string nombre;
public:
	estudiante(string n) { nombre = n; }
	estudiante() { nombre = "sin definir"; }
	string getNombre() { return nombre; }
	void setNombre(string n) { nombre = n; }
	string toString() {
		stringstream s;
		s << nombre << endl;
		return s.str();
	}
```

```
};
ostream& operator<<(ostream& salida, estudiante& a) {
	salida << a.toString();
	return salida;
}
//-------------------------------main.cpp-------------------------------------


void main() {
	coleccion<int>   vInt(10);
	coleccion<float> vFloat(10);
	coleccion<bool>  vBool(10);
	coleccion<char>  vChar(10);
	coleccion<string>  vString(10);
	coleccion<estudiante>  vEstudiante(10);

	for (int i = 0; i < 10; i++) {
		vInt[i] = 10 - i;
		vFloat[i] = (float)i / 2;
		vBool[i] = true;
		vChar[i] = 'a';
		vString[i] = "Hola";
		// los estudiante ya existen en el vector
	}


	vEstudiante[0].setNombre("Juan");
	vEstudiante[1].setNombre("Maria");
	vEstudiante[2].setNombre("Luis");
	vEstudiante[3].setNombre("Paolo");
	vEstudiante[4].setNombre("Ricardo");


	for (int i = 0; i < 10; i++)
		cout << vInt[i] << " - ";

	cout << endl;

	for (int i = 0; i < 10; i++)
		cout << " - " << vFloat[i] ;

	cout << endl;

	for (int i = 0; i < 10; i++)
		cout << " - "<< vBool[i] ;

	cout << endl;

	for (int i = 0; i < 10; i++)
		cout << " - "<< vChar[i];

	cout << endl;

	for (int i = 0; i < 10; i++)
		cout << " - " << vString[i] ;

	cout << endl;
	cout << endl;

	for (int i = 0; i < 10; i++)
		cout << " - " << vEstudiante[i] ;
```

```
	cin.get();
}
```

Ejemplo 4:

Las clases genéricas pueden ser utilizadas en los mecanismos de herencia. Es así como en el momento de instanciar el objeto es posible decidir cual será la clase antecesora.

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

class BaseA {
public:
	BaseA() { cout << "BaseA"; }
	virtual void imprimir() { cout << "Soy BaseA" << endl; }
};
//-----------------------------------------------
class BaseB {
public:
	BaseB() { cout << "BaseB"; }
	void imprimir() { cout << "Soy BaseB" << endl; }
};
//-----------------------------------------------

template <class T>
class Derivada : public T {
public:
	Derivada() { cout << "::Derivada" << endl; }
	 void imprimir() { cout << "Soy Derivada, mi padre es "<< typeid(T).name() << endl; }
};
//-----------------------------------------------

void main() {
	Derivada<BaseA> ob1;
	Derivada<BaseB> ob2;
	cout << endl << endl;
	ob1.imprimir();
	ob2.imprimir();
	cin.get();
}
```

Ejemplo 5:

En el siguiente ejemplo también se hace uso de herencia.

```
#include <iostream>
using namespace std;
//-------------------BaseA------------------------

template <class T>
class baseA {
public:
		baseA();
		virtual void p();
		virtual void q();
		virtual T z(T);
};

template <class T>
baseA<T>::baseA() {}

template <class T>
void baseA<T>::p() {}

template <class T>
void   baseA<T>::q() { }

template <class T>
T  baseA<T>::z(T x) { return x; }

//-------------------derivadaA1------------------------
//Esta clase no tiene plantillas y hereda de una con plantillas
class derivadaA1 : public baseA<int> {};
//-------------------derivadaA2------------------------
//Clase con plantillas que hereda de una con plantilla
template <class T>
class derivadaA2 : public baseA<T> {};
//-------------------baseB------------------------
class baseB {};

//-------------------derivadaB1------------------------
//Clase con plantillas que hereda de una sin plantillas
template <class T>
class derivadaB1 : public baseB {};

//-------------------main------------------------
void main() {
		// Ver practica mas abajo
		system("pause");
}
```

Cree las siguientes instancias en el main:

5.1.) Un puntero de tipo `baseA`, que apunte a una instancia de `baseA`, esta plantilla trabajara en base a cualquier tipo primitivo.
5.2.) Un puntero de tipo `baseA`, que apunte a una instancia de `baseA`, esta plantilla trabajara en base a punteros a `baseB`.
5.3.) Un puntero de tipo `derivadaA1`, que apunte a una instancia de tipo `derivadaA1`.
5.4.) Un puntero de tipo `derivadaA2`, que apunte a una instancia de tipo `derivadaA2` la cual trabaje con cualquier tipo.
5.5.) Un puntero de tipo `baseB` que apunte a una instancia de `baseB`.
5.6.) Un puntero de tipo `baseB` que apunte a una instancia de `derivadaB1` y la cual trabaje con cualquier tipo primitivo.
5.7.) Un puntero de tipo `derivadaB1` que apunte a una instancia de `derivadaB1` y la cual trabaje con cualquier tipo primitivo.

Solución

```
void main() {
	baseA<int>* obj1 = new baseA<int>();  // # 5.1.
	baseA<baseB*> * obj2 = new baseA<baseB*>(); // # 5.2.
	derivadaA1* obj3 = new derivadaA1();  // # 5.3.
	derivadaA2<int>* obj4 = new derivadaA2<int>();// # 5.4
	baseB* obj5 = new baseB();//# 5.5
	baseB* obj6= new derivadaB1<int>();// # 5.6.
	derivadaB1<int>* obj7 = new derivadaB1<int>();// # 5.7.
	system("pause");
}
```

Ejemplo 6:

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

template <typename T> T f(T);

int main(int argc, char** argv) {
        int x = 4;
        cout << "f(" << x << ") = " << f<int>(x) << endl;
        double y = 4.7;
        cout << "f(" << y << ") = " << f<double>(y) << endl;

        // En algunos casos, el compilador puede inferir el tipo de la función a partir de los
    //  valores utilizados como argumento. En este caso, el argumento es de tipo double.
        double w = 2.37;
        cout << "f(" << w << ") = " << f(w) << endl;

        // La funcion de plantilla no puede utilizar tipos para los cuales no estan definidas:
        string s = "Hola!";
        // cout << "f(" << s << ") = " << f<string>(s) << endl;


        cin.get();
        cin.get();
        return 0;
}


template <typename T> T f(T x) {
        return (2 * x) / 3;
}
```