

# *Programación Orientada a Objetos*

## *Plantillas (Templates)*

*Universidad de Los Andes - Facultad de Ingeniería  
Escuela de Sistemas  
Programación Digital II*

*Profesor: Gilberto Diaz  
gilberto@ula.ve*

Los programadores implementan algoritmos para resolver una diversidad de problemas, por ejemplo:

- Ordenar números
- Ordenar nombres
- Realizar operaciones aritméticas
- Operaciones matriciales
- Etc, etc, etc

Supongamos que queremos sumar vectores. Y la primera implementación se realiza con un **tipo de dato entero**.

Luego, por alguna razón, se necesita sumar vectores con números punto flotantes.

En lo primero que uno piensa es en copiar el código original y arreglarlo para que ahora utilice **tipos de datos punto flotante**

Supongamos que queremos sumar vectores. Y la primera implementación se realiza con un **tipo de dato entero**.

Luego, por alguna razón, se necesita sumar vectores con números punto flotantes.

En lo primero que uno piensa es en copiar el código original y arreglarlo para que ahora utilice **tipos de datos punto flotante**

Entonces, repetimos el mismo código para cada tipo de dato que necesitemos y estaremos reinventando la rueda cada vez.

C++ proporciona un mecanismo para evitar este tipo de problema.

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

```
class VectorEnteros {
```

```
    int tamano;  
    int *valores;
```

```
public:
```

```
    //Constructores
```

```
    VectorEnteros();
```

```
    VectorEnteros(const VectorEnteros&);
```

```
    VectorEnteros(const int, const int *);
```

```
    //Destructor
```

```
    ~VectorEnteros();
```

```
//Métodos de acceso
```

```
int obtenerElemento(const int);
```

```
int obtenerTamano();
```

```
void mostrarVector();
```

```
//Métodos de modificación
```

```
void asignarElemento(const int, const int);
```

```
void inicializarVector(const int);
```

```
//Sobrecarga de operadores
```

```
VectorEnteros operator+(VectorEnteros);
```

```
bool operator==(VectorEnteros);
```

```
};
```

```
#include <iostream>
#include "VectorEnteros.h"

//Constructor por omisión
VectorEnteros::VectorEnteros(){
    tamaño = 0;
}

//Constructor por copia
VectorEnteros::VectorEnteros(const VectorEnteros& vec){
    int i,j;
    tamaño = vec.tamaño;
    valores = new int[tamaño];
    for(i=0; i<tamaño; i++){
        valores[i] = vec.valores[i];
    }
}
```

```
//Constructor paramétrico
VectorEnteros::VectorEnteros(const int t, const int *vals){
    int i;
    tamaño = t;
    valores = new int[t];
    for(i=0; i<t; i++){
        valores[i] = vals[i];
    }
}
```

//Destructor

```
VectorEnteros::~~VectorEnteros(){  
    delete(valores);  
}
```

//Métodos de acceso

```
int VectorEnteros::obtenerElemento(const int i){  
    return this->valores[i];  
}
```

.....

//Métodos de modificación

```
void VectorEnteros::asignarElemento(const int i, const int val){  
    valores[i] = val;  
}
```

```
.....  
VectorEnteros VectorEnteros::operator+(VectorEnteros v){  
.....
```

```
//Sobrecarga de operadores  
bool VectorEnteros::operator==(VectorEnteros v){  
    int i;  
    if(tamano != v.tamano){  
        return false;  
    }  
    else{  
        for(i=0; i<tamano; i++){  
            if(this->valores[i] != v.valores[i])  
                return false;  
        }  
        return true;  
    }  
}
```

```
#include "VectorEnteros.h"
int main(){
    int vector[5], i;
    for(i=0; i<5; i++)
        vector[i] = i;
    VectorEnteros v1(5, vector);
    VectorEnteros v2(5, vector);
    if(v1 == v2)
        cout<<"Iguales"<<endl;
    else
        cout<<"Distintos"<<endl;
    VectorEnteros v3(5, vector);
    //v3.mostrarVector();
    v3 = v1 + v2;
    v3.mostrarVector();
    return 0;
}
```

Para adaptar ese código y funcione con cualquier tipo de dato numérico, podemos utilizar una plantilla.

A continuación mostramos como.

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

```
template <class V>
class Vector {
```

```
    int tamano;
    V *valores;
```

```
public:
```

```
    //Constructores
```

```
    Vector();
```

```
    Vector(const Vector&);
```

```
    Vector(const int, const V *);
```

```
    //Destructor
```

```
    ~Vector();
```

//Métodos de acceso

**V** obtenerElemento(const int);

int obtenerTamano();

void mostrarVector();

//Métodos de modificación

void asignarElemento(const int, const **V**);

void inicializarVector(const int);

//Sobrecarga de operadores

Vector operator+(Vector);

bool operator==(Vector);

**V** &operator[](const int& i){  
    return valores[i];

}

};

//Constructor por omisión

```
template <class V>
```

```
Vector<V>::Vector(){
```

```
    tamaño = 0;
```

```
}
```

//Constructor por copia

```
template <class V>
```

```
Vector<V>::Vector(const Vector& vec){
```

```
    int i,j;
```

```
    tamaño = vec.tamaño;
```

```
    valores = new V[tamaño];
```

```
    for(i=0; i<tamaño; i++){
```

```
        if(vec.tamaño > i){
```

```
            valores[i] = vec.valores[i];
```

```
        }
```

```
    }
```

```
}
```

//Constructor paramétrico

**template <class V>**

Vector<V>::Vector(const int t, const V \*vals){

int i;

tamano = t;

valores = new V[t];

for(i=0; i<t; i++){

valores[i] = vals[i];

}

}

//Destructor

**template <class V>**

Vector<V>::~~Vector(){

delete valores;

}

//Métodos de acceso

**template <class V>**

```
V Vector<V>::obtenerElemento(const int i){  
    return this->valores[i];  
}
```

**template <class V>**

```
int Vector<V>::obtenerTamano(){  
    return tamano;  
}
```

//Métodos de modificación

**template <class V>**

```
void Vector<V>::asignarElemento(const int i, const V val){  
    valores[i] = val;  
}
```

//Sobre Carga de operadores

```
template <class V> bool Vector<V>::operator==(Vector v){  
    int i;  
    if(tamano != v.tamano){  
        return false;  
    }  
    else{  
        for(i=0; i<tamano; i++){  
            if(this->valores[i] != v.valores[i])  
                return false;  
        }  
        return true;  
    }  
}
```

```
#include "Vector.h"
int main(){
    int vec[5], i;
    float vecf[5];
    for(i=0; i<5; i++){
        vec[i] = i;
        vecf[i] = 3.8;
    }
    Vector <int> v1(5, vec);
    Vector <int> v2(5, vec);
    Vector <float> v3(5, vecf);
    .....
}
```