

# **Templates**

Antonio Espín Herranz

# Templates

- C++ nos da la posibilidad de trabajar con tipos genéricos o parametrizados.
- De echo las clases contenedoras están implementadas con esta filosofía.
- Definimos `vector<T>`, siendo T un tipo primitivo o un objeto definido por el usuario.
- Es el compilador el encargado de generar el código concreto para el tipo indicado.

# Templates

- Vamos a poder definir **funciones genéricas y clases genéricas**.
- Sintaxis:  
template<lista\_parametros> declaración
- Ejemplos:  
**template <class T> T menor(T a, T b);**
  - menor es una función que recibe dos parámetros a y b de tipo T y devuelve un objeto de tipo T.
  - Siendo T un tipo genérico.  
**template <class T> class MiClase { ... }**
  - Vamos a poder definir clases genéricas. Indicando el tipo parametrizado al crear la clase.

# Funciones genéricas

- La función menor para un tipo concreto podría ser de esta forma:  

```
int menor(int a, int b){  
    return (a < b) ? a : b;  
}
```
- Si quisiéramos aplicarla a otro tipo de objetos cambiaríamos el tipo de los parámetros y el devuelto. O la parametrizamos:

```
template<class T> T menor (T a, T b){  
    return (a < b) ? a : b;  
}
```

- Ahora la función nos sirve para tipo int, double, float, etc.
- **Siempre y cuando en el tipo que apliquemos esté definido el operador <**
- **Y además deberá estar implementado con una función friend.**

# Consideraciones

- Al igual que hacemos con las funciones podemos separar la declaración y la definición de una plantilla de función.
  - Las podemos definir y luego las utilizamos.
  - Primero se declaran, después se utilizan y finalmente se definen.
  - *(como cualquier otra función)*
- La forma en que se aplican las plantillas de función a la hora de utilizarlas es por medio de los parámetros que recibe.
  - Si son int se aplica la función para int, así sucesivamente.
- Para aplicar mediante el tipo devuelto hay que hacer una indicación.
  - Ejemplo:  
`template<class T> T *asignarMem(int tam){ ... }`

La llamada:

```
double *pd;   pd = asignarMem(10);
```

**// Si da ERROR, utilizar la forma siguiente:**

Habría que indicar el tipo:

```
double *pd;   pd = asignarMem<double>(10); // OK
```

# Especialización de plantillas de función

- Una versión de una plantilla para un parámetro concreto se denomina *especialización*.
- Se puede dar el caso que si utilizamos la plantilla de función anterior (menor) con `char *`, no funciona correctamente. Debería de trabajar con la función `strcmp(char *, char *)`.
- Habría que implementar una especialización.

```
template<> char *menor<char *>(char *a, char *b){  
    return(strcmp(a, b) < 0) ? a : b;  
}
```

# Especialización parcial

- Este caso se utiliza mas.
- Es lo que hacemos cuando trabajamos con las clases contenedoras. Estamos eligiendo el contenedor (vector, list, etc) y parametrizamos el tipo T.

```
template<class T> vector<T>menor (vector<T> a, vector <T>b){  
    return (a < b) ? a : b;  
}
```

- Si quisiéramos hacer la especialización de un tipo de vectores concretos:

```
template<> vector<double> menor(vector<double>, vector<double>)  
    return (a < b) ? a : b;  
}
```

# Sobrecarga de plantillas de función

- Las plantillas de función se pueden sobrecargar de la misma forma que sobrecargamos las funciones.
- Incluso se puede combinar la sobrecarga de estas con funciones normales.



# Ejemplo

```
template<class T> T menor (T a, T b){  
    return (a < b) ? a : b;  
}
```

```
template<class T> vector<T> menor (vector<T> a, vector <T>b){  
    return (a < b) ? a : b;  
}
```

```
double menor (double a, double b){  
    return (a < b) ? a : b;  
}
```

```
int main(){  
    vector <double> v1(10);  
    vector <double> v2(10);
```

```
    menor(10, 27);           // Se genera una función menor para int.  
    menor(v1, v2);          // La plantilla de vector para double.  
    menor(26.3, 15.22);     // La función menor de dos double.
```

# Organización del código

- Se puede colocar la parte de la declaración y la definición en el mismo fichero o en ficheros separados.
- **Opciones:**
  - Colocar todo el código las declaraciones del .H y el .CPP en el fichero .H. El CPP se borra y luego en main se incluye el .H como siempre!!
  - Tenerlo separado, pero en el CPP hay que especificar los tipos que vamos a utilizar:

# Ejemplo

- En el .H se especifica la declaración de la clase.
- En el .CPP se implementa, pero se especifican los tipos:

```
#include "Lista.h"
```

```
// PREDECLARACION DE TIPOS!
```

```
template class Lista<int>;
```

```
template class Lista<string>;
```

# ¿Para las clases?

- En cuanto a la separación de los ficheros de cabecera **h** e implementación **cpp**:
  - A aplicamos los mismos criterios que las plantillas de funciones.

# Clases genéricas

- Una clase genérica es una plantilla para definir un conjunto de clases que se diferencian en el tipo de los datos que manipulan.
- Las clases contenedoras son clases genéricas nos proporcionan una serie de operaciones (insertar, eliminar, etc.) y como parámetro se indica el tipo.
- Normalmente para crear una plantilla de clase siempre nos basamos en una clase ya creada y depurada, de esta forma eliminaremos los posibles errores generados de la propia clase.

# Clases genéricas

- Por ejemplo partiendo de una clase que representa un vector de `*double`, vamos a implementar una plantilla de cualquier tipo de puntero.
- Para que nos sirva para `*int`, `*float`, etc.
- Al igual que hacemos con otras clases, en el fichero de implementación `cpp` tenemos que indicar el ámbito.

# Sintaxis

- **Para la declaración de la clase:**

```
template <class T> class NombreClase { ...}
```

- **Para los métodos de la clase:**

```
template <class T> tipo_devuelto  
NombreClase<T>::nombreMetodo(...){...}
```

- **Ejemplos:**

- **Constructor:**

- `template <class T> Vector<T>::Vector(int n)`

- **Constructor copia:**

- `template <class T> Vector<T>::Vector(const Vector<T> &otro)`

- **Operator=**

- `template <class T> Vector<T> & Vector<T>::operator=(const Vector<T> &otro)`

# Ejemplo

// vector.h - Plantilla de clase Vector

#ifndef VECTOR\_H

#define VECTOR\_H

template<class **T**> **class Vector** { // declaración

private:

**T** \*vector; // puntero al primer elemento de la matriz

int nElementos; // número de elementos de la matriz

protected:

**T** \*asignarMem(int);

public:

Vector(int ne = 10); // crea un Vector con ne elementos

Vector(const Vector&); // crea un Vector desde otro

~Vector() { delete [] vector; vector = 0; } // destructor

Vector& operator=(const Vector&);

**T**& operator[](int i) const { return vector[i]; }

int longitud() const { return nElementos; }

};

#endif



# Ejemplo

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

// Constructores: Crear una matriz con ne elementos

```
template<class T> Vector<T>::Vector(int ne){
    if (ne < 1) {
        cerr << "Nº de elementos no válido: " << ne << "\n";
        return;
    }
    nElementos = ne;
    vector = asignarMem(nElementos);
}
```

// Constructor copia, inicializa el atributo vector a cero. Y llama a =

```
template<class T> Vector<T>::Vector(const Vector& v) : vector(0){
    *this = v;
}
```

# Ejemplo

// Operador de asignación: Definir plantilla, el tipo devuelto, el ámbito y el operador:

```
template<class T> Vector<T>& Vector<T>::operator=(const Vector& v){
    nElementos = v.nElementos;           // número de elementos
    delete [ ] vector;                   // borrar la matriz actual
    vector = asignarMem(nElementos);     // crear una nueva matriz
    for (int i = 0; i < nElementos; i++)
        vector[i] = v.vector[i];        // copiar los valores
    return *this;                        // permitir asignaciones encadenadas
}
```

// Otros métodos

```
template<class T> T *Vector<T>::asignarMem(int nElems){
    try {
        T *p = new T[nElems];
        return p;
    }
    catch(bad_alloc e) {
        cout << "Insuficiente espacio de memoria\n";
        exit(-1);
    }
}
```

# Ejemplo

// Podemos definir funciones externas que utilicen la plantilla:

```
template<class T> void visualizar(Vector<T>&);
```

```
int main(){
```

```
    Vector<double> vector(5); // Definimos un vector de tipo double.
```

```
    for (int i = 0; i < vector.longitud(); i++)
```

```
        vector[i] = i+1;
```

```
    visualizar(vector); // Utilización de la función.
```

```
}
```

// La implementación de la función:

```
template<class T> void visualizar(Vector<T>& v){
```

```
    int ne = v.longitud();
```

```
    for (int i = 0; i < ne; i++)
```

```
        cout << setw(7) << v[i];
```

```
    cout << "\n\n";
```

```
}
```

# Clases genéricas con funciones friend

- Cuando tenemos una clase que declara **funciones externas friend** (por ejemplo, para sobrecargar un operador) y esta clase la **convertimos en template**, hay que utilizar la siguiente sintaxis:

# Ejemplo (friend con Template)

- Partimos de una clase Vector que tiene una función friend para imprimir los elementos del vector: **operator<<** y otra para sumar dos vector, **operator+**:

```
template <class T> class Vector
{
    friend std::ostream & operator<<(std::ostream &os, const
    Vector<T> &v){
        // Implementación
    }

    friend Vector<T> operator+(const Vector<T> &v1, const
    Vector<T> &v2){
        // Implementación
    }
}
```

# Template con varios

- Puede que necesitemos parametrizar una plantilla con varios tipos, se toma por convención: T, R, S, etc. T1, T2  
...
- `template <class T, class R, class S> class MiClase`

# Ejemplo en el .H

```
template <class T, class R, class S>  
class Plantilla {  
    T t;  
    R r;  
    S s;  
    // Resto de declaraciones ...  
}
```

# Ejemplo en el .CPP

- // Constructor:

```
template <class T, class R, class S>  
Plantilla<T,R,S>::Plantilla(T t, R r, S s){...}
```

- // Un método:

```
template <class T, class R, class S>  
void Plantilla<T,R,S>::print(){...}
```



# Derivación de clases genéricas

- Una plantilla puede derivarse de una clase o de otra plantilla y así construir un nuevo tipo.
- Podríamos definir una plantilla para matrices de 2D de tipo genérico apoyándonos en la plantilla de `Vector<T>`.

# Ejemplo

```
#ifndef MATRIZ2D
```

```
#define MATRIZ2D
```

```
#include "Vector.h"
```

```
template <class T> class Matriz2D : public Vector<T> {
```

```
private:
```

```
    T **matriz;
```

```
    int cols;
```

```
public:
```

```
    Matriz2D(int f = 5, int c = 5);
```

```
    // Llama a la clase Padre:
```

```
    inline int getFilas() const { return Vector<T>::longitud(); }
```

```
    inline int getCols() const { return cols; }
```

```
    T& operator()(int f, int c);
```

```
    virtual ~Matriz2D();
```

```
};
```

```
#endif
```