

Plantillas



Motivación

Muchas veces tuvimos que repetir código como:

```
int promedio(int num1, int num2, int num3)
{ return (int)((num1+num2+num3)/3); }
```

```
float promedio(float num1, float num2, float
num3)
{ return (float)((num1+num2+num3)/3); }
```

```
double promedio(double num1, double num2, double
num3)
{ return (double)((num1+num2+num3)/3); }
```

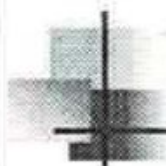


Motivación

No sería mejor que pudiéramos escribir algo más *genérico* como:

```
?? promedio(?? num1, ?? num2, ?? num3)
{ return (??) ((num1+num2+num3)/3); }
```

... y que el compilador se encargue de poner los tipos adecuados...




Programación genérica

- Paradigma de programación centrado en los algoritmos más que en los datos.

- Generalización

Significa que, en la medida de lo posible, los algoritmos deben ser parametrizados al máximo y expresados de la forma más independiente posible de detalles concretos, permitiendo así que puedan servir para la mayor variedad posible de tipos y estructuras de datos.




Comparación de paradigmas de programación

▪ Orientada al dato

Representemos un tipo de dato genérico (por ejemplo **int**) que permita representar objetos con ciertas características comunes (peras y manzanas por ejemplo). Definamos también que operaciones pueden aplicarse a este tipo (por ejemplo aritméticas) y sus reglas de uso, independientemente que el tipo represente peras o manzanas en cada caso.

▪ Funcional

Construyamos un algoritmo genérico (por ejemplo **sort**), que permita representar algoritmos con ciertas características comunes (ordenación de cadenas alfanuméricas y vectores por ejemplo). Definamos también a que tipos pueden aplicarse a este algoritmo y sus reglas de uso, independientemente que el algoritmo represente la ordenación de cadenas alfanuméricas o vectores.



¿Para qué *plantillas*?

Si se implementa el comportamiento una y otra vez se reinventa la rueda. Los errores que pueden existir en un código pueden llevarse a otros y la corrección es dificultosa.

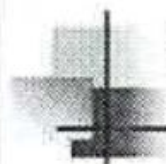
Si se escribe código para tipos genéricos se pierde la ventaja del control de tipos que hace el lenguaje. Las clases bases hacen en general más difícil de mantener el código.

Si se utilizan preprocesadores se pierde la ventaja del trabajar con formato de código propio del lenguaje.



Plantillas son la solución

- Las plantillas son funciones o clases definidas para tipos no especificados.
- Al utilizarlas se pasa el tipo como un argumento ya sea explícita o implícitamente.
- Al ser componentes del lenguaje proveen control total de tipos y de ámbito.



Plantillas o tipos parametrizados

- El mecanismo de plantillas C++ es en realidad un generador automático de código parametrizado.

Plantillas: generación de código

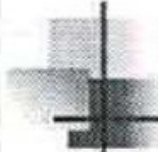
Función genérica + argumentos



*instanciación
de la plantilla*



Función concreta (o especializada)



Plantillas: generación de código


Clase genérica + argumentos



*instanciación
de la plantilla*




Clase especializada



Plantillas: meta-programación

¿Programas que escriben Programas?

(todo se resuelve en tiempo de compilación)



La palabra clave *template*


```
template <T> void miFuncion(T& ref)
{/* declaración de función genérica
*/};
```

```
template <T> class miClase
{/* declaración de clase genérica
*/};
```

Plantillas de función

- Proveen un comportamiento que puede ser invocado con distintos tipos.
- Tiene una representación similar a una función.

```
template <typename T>  
inline T const& max (T const& a, T const& b)  
{  
    // if a < b then use b else use a  
    return a < b ? b : a;  
}
```



Invocando a max

```
int i,j;
```

```
i=10, j=5;
```

```
int k = max(i,j);
```

```
double d = max(4.0 , 3.14);
```

```
UnacLase a, b;
```

```
UnacLase c = max(a,b);
```

Deducción de argumentos

- El sistema no hace la conversión automática de los tipos que uno pasa.


Por ejemplo:

`max(4,4.3)` Generará un error!!

`max(static_cast<double>(4),4.3)` //bien

- Se puede calificar la función

`max<double>(4,4.3);`



Sobrecarga de plantillas de función

- Se aplican las mismas reglas que a la sobrecarga de funciones.
- Las funciones no plantilla pueden coexistir con funciones plantilla pero ellas son preferidas.

Plantillas de clase

- Similares a las plantillas de función.
- Caso típico de clases contenedoras.
- Pueden indicarse con *typename* o *class* indistintamente

```
template <typename T>
class conjunto{
    private:
        std::vector<T> set;
    public:
        void agregar(T x);
        conjunto<T>
        unionC(conjunto<T> c);
        conjunto<T>
        interseccionC(conjunto<T> c);
        bool pertenencia(T x);
        void mostrar();
};
```

Implementación de las funciones miembro

- Se debe indicar que se trata de una plantilla de función.
- También deben indicarse explícitamente el tipo de datos que maneja.

```
template <typename T>
void conjunto<T>::agregar(T x){
    bool esta=false;
    for(int k=0;(!esta &&
        k<set.size());k++)
        if (set[k]==x) esta=true;
    if (!esta) set.push_back(x);
}
```

```
template <typename T>
conjunto<T> conjunto<T>::unionC
(conjunto<T> c){
    conjunto a=c; int k;
    for(k=0;k<set.size();k++){
        a.agregar(set[k]);
    }
    return a;
}
```