

INFORMÁTICA INDUSTRIAL I

Grado en Ingeniería en Tecnologías Industriales
3^{er} curso

6. Lenguaje C++: Plantillas

Profesores: M. Abderrahim / F. Abu-Dakka / Á. Castro González
Departamento de Ingeniería de Sistemas y Automática

AGENDA

- Plantillas
- Sintaxis
- Plantillas de Funciones
- Funciones con Plantillas como Parámetros
- Pasar una Plantilla Genérica
- Miembros Estáticos: datos y funciones
- Biblioteca de Plantillas

Plantillas

- 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.
- Por ejemplo, a menudo tendremos que implementar arrays dinámicos para diferentes tipos de objetos, o listas dinámicas, pilas, colas, árboles, etc.
- El código es similar siempre, pero estamos obligados a rescribir ciertas funciones que dependen del tipo o de la clase del objeto que se almacena.
- Las plantillas (templates) nos permiten parametrizar estas clases para adaptarlas a cualquier tipo de dato.

Sintaxis

- C++ permite crear plantillas de funciones y plantillas de clases.
- La sintaxis para declarar una plantilla de función es parecida a la de cualquier otra función, pero se añade al principio una presentación de la clase que se usará como referencia en la plantilla:

```
template <class|typename <id>[,...]>  
<tipo_retorno> <identificador>(<lista_de_parámetros>)  
{  
// Declaración de función  
};
```

```
template<typename T>  
void f(T s)  
{  
    std::cout << s << “\n”;  
}
```

Sintaxis

- La sintaxis para declarar una plantilla de clase es parecida a la de cualquier otra clase, pero se añade al principio una presentación de la clase que se usará como referencia en la plantilla:

```
template <class|typename <id>[,...]>  
class <identificador_de_plantilla>  
{  
  // Declaración de funciones  
  // y datos miembro de la plantilla  
};
```

```
template <class T1>  
class Tabla {  
  public:  
    Tabla();  
    ~Tabla();  
    ...  
};
```

Sintaxis

- Del mismo modo, cuando tengamos que definir una función miembro fuera de la declaración de la clase, tendremos que incluir la parte del **template** y como nombre de la clase **incluir la plantilla antes del operador de ámbito (::)**. Por ejemplo:

```
template <class T1>  
Tabla<T1>::Tabla()  
{  
    // Definición del constructor  
}
```

Plantillas de Funciones

- Un ejemplo de plantilla de función puede ser esta que devuelve el valor máximo:

```
template <class T>  
T max(T x, T y)  
{  
    return (x > y) ? x : y;  
};
```

- La podemos utilizar con datos de diferentes tipos:

```
int a=3, b=5, c;  
char f='a', g='k', h;  
c = max(a,b);  
h = max(f,g);
```

Plantillas para Tablas

- Ahora estamos en disposición de crear una plantilla a partir de la clase Tabla que hemos definido antes.
- Esta vez podremos usar esa plantilla para definir Tablas de cualquier tipo de objeto.

Plantillas para Tablas

```

template <class T>
class Tabla {
public:
    Tabla(int nElem);
    ~Tabla();
    T& operator[(int indice) { return pT[indice]; }
private:
    T *pT;
    int nElementos;
};
// Definición:
template <class T>
Tabla<T>::Tabla(int nElem) : nElementos(nElem) {
    pT = new T[nElementos];
}
template <class T>
Tabla<T>::~~Tabla() {
    delete[] pT;
}

```

Plantillas para Tablas

- Para declarar Tablas del tipo que queramos. La sintaxis es:

```
<identificador_de_plantilla><<tipo/clase>> <identificador/constructor>;
```

- La podemos utilizar con datos de diferentes tipos:

```
Tabla<int> TablaInt(32); // Tabla de 32 enteros  
Tabla<float> TablaFloat(12); // Tabla de 12 floats  
Tabla<bool> TablaBool(10); // Tabla de 10 bools
```

Ejemplo: Uso de la plantilla Tabla

// Tabla.h: definición de la plantilla tabla:

```
#ifndef T_TABLA
```

```
#define T_TABLA
```

```
template <class T>
```

```
class Tabla {
```

```
public:
```

```
    Tabla(int nElem);
```

```
    ~Tabla();
```

```
    T& operator[](int indice) {
```

```
        return pT[indice];
```

```
    }
```

```
    int NElementos() const { return nElementos; }
```

```
private:
```

```
    T *pT;
```

```
    int nElementos;
```

```
};
```

// Definición:

```
template <class T>
```

```
Tabla<T>::Tabla(int nElem) :
```

```
nElementos(nElem)
```

```
{
```

```
    pT = new T[nElementos];
```

```
}
```

```
template <class T>
```

```
Tabla<T>::~~Tabla()
```

```
{
```

```
    delete[] pT;
```

```
}
```

```
#endif
```

Ejemplo: Uso de la plantilla Tabla

```
// Tabla.cpp: ejemplo de Tabla
#include <iostream>
#include "Tabla.h"
using namespace std;
const int nElementos = 10;
int main() {
    Tabla<int> TablaInt(nElementos);
    Tabla<float> TablaFloat(nElementos);
    for(int i = 0; i < nElementos; i++)
        TablaInt[i] = nElementos-i;
    for(int i = 0; i < nElementos; i++)
        TablaFloat[i] = 1/(1+i);
    for(int i = 0; i < nElementos; i++) {
        cout << "TablaInt[" << i << "] = " << TablaInt[i] << endl;
        cout << "TablaFloat[" << i << "] = " << TablaFloat[i] << endl;
    }
    return 0;
}
```

```
WJRG-MacBookPro:Clase11 w
WJRG-MacBookPro:Clase11 w
TablaInt[0] = 10
TablaFloat[0] = 1
TablaInt[1] = 9
TablaFloat[1] = 0
TablaInt[2] = 8
TablaFloat[2] = 0
TablaInt[3] = 7
TablaFloat[3] = 0
TablaInt[4] = 6
TablaFloat[4] = 0
TablaInt[5] = 5
TablaFloat[5] = 0
TablaInt[6] = 4
TablaFloat[6] = 0
TablaInt[7] = 3
TablaFloat[7] = 0
TablaInt[8] = 2
TablaFloat[8] = 0
TablaInt[9] = 1
TablaFloat[9] = 0
```

Funciones con Plantillas como Parámetros

- Es posible crear funciones que admitan parámetros que sean una plantilla. Hay dos modos de pasar las plantillas: se puede pasar
 - **una instancia determinada de la plantilla o**
 - **la plantilla genérica.**
- Si declaramos y definimos una función que tome como parámetro una instancia concreta de una plantilla, esa función no estará disponible para el resto de las posibles instancias. Por ejemplo, **si creamos una función para una tabla de enteros, esa función no podrá aplicarse a una tabla de caracteres.**

Ejemplo: Funciones con Plantillas como Parámetros

// F_Tabla.cpp:

// ejemplo de función de plantilla para

// Tabla para enteros:

#include <iostream>

#include "Tabla.h"

using namespace std;

const int nElementos = 5;

void Incrementa(Tabla<int> &t); // (1)

int main() {

 Tabla<int> TablaInt(nElementos);

 Tabla<char> TablaChar(nElementos);

 for(int i = 0; i < nElementos; i++)

 {

 TablaInt[i] = 0;

 TablaChar[i] = 0;

 }

Incrementa(TablaInt); // (2)

// Incrementa(TablaChar); // <-- ilegal

```
for(int i = 0; i < nElementos; i++)
    cout << "TablaInt[" << i << "] = "
        << TablaInt[i] << endl;
return 0;
```

}

void Incrementa(Tabla<int> &t) { // (3)

 for(int i = 0; i < t.NElementos(); i++)

 t[i]++;

}

```
WJRG-MacBookPro:Clase11 wladimir$ g++ F_Tabla.cpp -o F_Tabla
WJRG-MacBookPro:Clase11 wladimir$ ./F_Tabla
TablaInt[0] = 1
TablaInt[1] = 1
TablaInt[2] = 1
TablaInt[3] = 1
TablaInt[4] = 1
```

Pasar una Plantilla Genérica

- Si declaramos y definimos una función que tome como parámetro una instancia cualquiera, tendremos que crear una función de plantilla. Para ello, como ya hemos visto, hay que añadir a la declaración de la función la parte **template<class T>**.
- Veamos un ejemplo:

Ejemplo: Plantilla Genérica

```
// F_Tabla2.cpp:
```

```
// ejemplo de función de plantilla
```

```
// Tabla genérica:
```

```
#include <iostream>
```

```
#include <cstdio>
```

```
#include "Tabla.h"
```

```
#include "CCadena.h"
```

```
using namespace std;
```

```
const int nElementos = 5;
```

```
template<class T>
```

```
void Mostrar(Tabla<T> &t); // (1)
```

```
int main() {
```

```
    Tabla<int> TablaInt(nElementos);
```

```
    Tabla<Cadena> TablaCadena(nElementos);
```

```
        // Cadena es una clase definida por el usuario
```

```
    char cad[20];
```

```
    for(int i = 0; i < nElementos; i++)
        TablaInt[i] = i;
```

```
    for(int i = 0; i < nElementos; i++)
    {
        sprintf(cad, "Cad no.: %2d", i);
        TablaCadena[i] = cad;
    }
```

```
    Mostrar(TablaInt); // (2)
```

```
    Mostrar(TablaCadena); // (3)
```

```
    return 0;
```

```
}
```

```
template<class T>
```

```
void Mostrar(Tabla<T> &t) { // (4)
```

```
    for(int i = 0; i < t.NElementos(); i++)
        cout << t[i] << endl;
```

```
}
```

Miembros Estáticos: datos y funciones

- Igual que con las clases normales, es posible declarar datos miembro o funciones estáticas dentro de una plantilla. En este caso **existirá una copia de cada uno de ellos para cada tipo de instancia que se cree**.
- Por ejemplo, si añadimos un miembro static en nuestra declaración de "Tabla", se creará una única instancia de miembro estático para todas las instancias de Tabla<int>, otro para las instancias de Tabla<Cadena>, etc.
- Hay un punto importante a tener en cuenta. Tanto si trabajamos con clases normales como con plantillas, debemos reservar un espacio físico para las variables estáticas, de otro modo el compilador no les asignará memoria, y se producirá un error si intentamos acceder a esos miembros.

Ejemplo: Miembros Estáticos

// Es posible leer/escribir miembros estáticos

// sin instanciar objetos de la clase

```
template <class T>
```

```
class MyClass
```

```
{
```

```
    // static member declaration
```

```
    static int count;
```

```
    ...
```

```
};
```

// static member definition

```
template<class T>
```

```
    int MyClass<T>::count = 0;
```

Ejemplo: Miembros Estáticos

```
// Fichero de cabecera: prueba.h
#ifndef T_PRUEBA
#define T_PRUEBA

template <class T> class Ejemplo
{
public:
    Ejemplo(T obj) {
        objeto = obj;
        estatico++;
    }
    ~Ejemplo() {estatico--;}

    static int LeeEstatico() {
        return estatico;
    }
}
```

```
private:
    static int estatico; // (1)
    T objeto;
};
#endif
```

Ejemplo: Miembros Estáticos

```
// Fichero de prueba: prueba.cpp
#include <iostream>
#include "prueba.h"
using namespace std;

// Esto es necesario para que exista una instancia
de la variable:
template <class T> int Ejemplo<T>::estatico;

void main() {

    Ejemplo<int> EjemploInt1(10);
    cout << "Ejemplo<int>: " <<
    EjemploInt1.LeeEstatico() << endl;

    Ejemplo<char> EjemploChar1('g');
    cout << "Ejemplo<char>: " <<
        EjemploChar1.LeeEstatico() << endl;
```

```
Ejemplo<int> EjemploInt2(20);
cout << "Ejemplo<int>: " <<
    EjemploInt2.LeeEstatico() << endl;

//Ejemplo<float> EjemploFloat1(32.12);
cout << "Ejemplo<float>: " <<
    Ejemplo<float>::LeeEstatico() <<
endl;

Ejemplo<int> EjemploInt3(30);
cout << "Ejemplo<int>: " <<
    EjemploInt3.LeeEstatico() << endl;
}
```

```
WJRG-MacBookPro:Clase11 wladimir$ g++ prueba.cpp -
WJRG-MacBookPro:Clase11 wladimir$ ./prueba
Ejemplo<int>: 1
Ejemplo<char>: 1
Ejemplo<int>: 2
Ejemplo<float>: 1
Ejemplo<int>: 3
```

Biblioteca de Plantillas

- El ANSI de C++ define ciertas bibliotecas de plantillas conocidas como STL (Standard Template Library), que contiene muchas definiciones de plantillas para crear estructuras como listas, colas, pilas, árboles, tablas HASH, mapas, etc.
- Está fuera del objetivo de este curso explicar estas bibliotecas, pero es conveniente saber que existen y que por su puesto, se suelen incluir con los compiladores de C++.
- Aunque se trata de bibliotecas estándar, lo cierto es que existen varias implementaciones, que, al menos en teoría, deberían coincidir en cuanto a sintaxis y comportamiento.
- <http://www.cplusplus.com/reference/stl/>

Biblioteca de Plantillas: Ventajas

- Disponible por todos los compiladores y plataformas.
- El uso de una librería de componentes reutilizables incrementa la productividad ya que los programadores no tienen que escribir sus propios algoritmos.
- **Libre de errores:** reduce el tiempo de desarrollo, incrementa la robustez de la aplicación.
- Las aplicaciones pueden escribirse rápidamente ya que se construyen a partir de algoritmos.
- Se incrementa la legibilidad del código, lo que hará que éste sea mucho más fácil de mantener.
- **Proporciona su propia gestión de memoria.** El almacenamiento de memoria es automático y portátil, así el programador ignorará problemas tales como las limitaciones del modelo de memoria del PC.

Biblioteca de Plantillas

- Contenedores de secuencias:
 - array: Array class (class template)
 - **vector: Vector (class template)**
 - deque: Double ended queue (class template)
 - forward_list: Forward list (class template)
 - **list: List (class template)**
- Contenedores de secuencia especiales
 - **stack: LIFO stack (class template)**
 - queue: FIFO queue (class template)
 - priority_queue: Priority queue (class template)
- Contenedores asociativos :
 - set: Set (class template)
 - multiset : Multiple-key set (class template)
 - **map: Map (class template)**
 - multimap: Multiple-key map (class template)

Ejemplo: Vector

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

int main(){
    std::vector<int> mi_vector;
    mi_vector.push_back(4);
    mi_vector.push_back(2);
    mi_vector.push_back(5);
    // para recorrer un vector podemos utilizar los iteradores o los índices
    for(std::size_t i=0;i<mi_vector.size();++i)
        std::cout << mi_vector[i] << ' ';
    std::cout << std::endl;

    std::vector<int> mi_vector(5,69); // crea el vector 69,69,69,69,69
    v[0] = 5;
    v[1] = 3;
    v[2] = 7;
    v[3] = 4;
    v[4] = 8;
    return 0;
}
```

Biblioteca de Plantillas: Iteradores

- Un iterador es un objeto que apunta a un elemento dentro de un rango (por ejemplo un array o un contenedor) y que es capaz de iterar entre los elementos del rango usando un conjunto de operadores (incremento (++) y referencia (*)).
- La forma más sencilla de iterador es un puntero: Puede apuntar a elementos de un array y desplazarse por ellos usando el operador incremento (++).
- En el caso de los contenedores, es necesario usar otro tipo de iteradores. Por ejemplo, el contenedor vector tiene un iterador específico diseñado para optimizar el acceso a sus elementos.
 - `container::iterator` provides a **read/write** iterator
 - `container::const_iterator` provides a **read-only** iterator

Ejemplo: Iterador

```
#include <iostream>
#include <vector>
int main()
{
    using namespace std;

    vector<int> vect;
    for (int nCount=0; nCount < 6; nCount++)
        vect.push_back(nCount);

    vector<int>::const_iterator it; // declare an read-only iterator
    it = vect.begin(); // assign it to the start of the vector
    while (it != vect.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the value of the element it points to
        it++; // and iterate to the next element
    }

    cout << endl;
}
```