

CS2013: Programación III

Teoría: Programación Genérica

José Chávez

UTEC



Contenido

- Introducción
- Funciones Plantilla
- Clases Plantilla
- Tipos de parámetros template

A stack of three books is shown on a light-colored wooden surface. The books are slightly offset, with the top book having a white cover and the bottom two having yellow covers. The background is a soft, out-of-focus light yellow. In the top right corner of the slide, there is a small orange horizontal bar.

Introducción

Programación Genérica

La librería **vector** provee una definición genérica al **vector**:

```
vector<int> v; // Vector de enteros
```

Programación Genérica

La librería **vector** provee una definición genérica al **vector**:

```
vector<float> v; // Vector de números reales
```

Recordemos que uno puede crear múltiples tipos de vectores, e.g. `vector<float>`, `vector<string>`, etc.

Programación Genérica

La librería **vector** provee una definición genérica al **vector**:

```
vector<strings> v; // Vector de 'strings'
```

Recordemos que uno puede crear múltiples tipos de vectores, e.g. `vector<float>`, `vector<string>`, etc.

¿Varios tipos = Varias Funciones?

Imaginemos que requerimos sumar dos números. En la práctica podemos definimos múltiples funciones o sobrecargar una función:

```
int sumar(int x, int y){  
    return x + y;  
}  
float sumar(float x, float y){  
    return x + y;  
}
```

¿Varios tipos = Varias Funciones?

Imaginemos que requerimos sumar dos números. En la práctica podemos definimos múltiples funciones o sobrecargar una función:

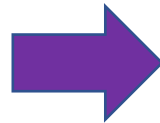
```
int sumar(int x, int y){  
    return x + y;  
}  
float sumar(float x, float y){  
    return x + y;  
}
```

Esto sería un problema si queremos sumar más tipos. O si requerimos que el tipo se defina en la interfaz.

¿Varios tipos = Varias Funciones?

Imaginemos que requerimos sumar dos números. En la práctica podemos definimos múltiples funciones o sobrecargar una función:

```
int sumar(int x, int y){  
    return x + y;  
}  
float sumar(float x, float y){  
    return x + y;  
}
```



```
template<typename T>  
T sumar(T x, T y){  
    return x + y;  
}
```

Una única función
(Función Plantilla)

A stack of books is shown on a wooden surface. The books are slightly out of focus, and the background is a warm, blurred light. An orange horizontal bar is located in the top right corner of the slide.

Funciones Plantilla

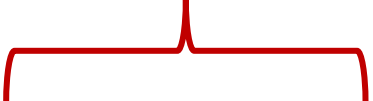
Funciones Plantilla

Este tipo de función nos permite generar distintas versiones de sí misma dependiendo del tipo de dato que necesitemos:

```
template<typename T>  
T sumar(T x, T y){  
    return x + y;  
}
```

Funciones Plantilla

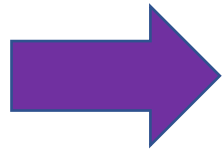
Lista de *parámetros template*



```
template<typename T>
T sumar(T x, T y){
    return x + y;
}
```

Funciones Plantilla

```
template<typename T>
T sumar(T x, T y){
    return x + y;
}
```

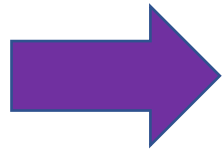


```
int main(){
    cout<<sumar(3.4,7.6)<<endl;
    cout<<sumar(3,4)<<endl;
}
```

El compilador deduce el valor de T (int para el primer caso y double para el segundo)

Funciones Plantilla

```
template<typename T>
T sumar(T x, T y){
    return x + y;
}
```

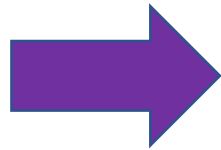


```
int main(){
    cout<<sumar(3.4,7)<<endl;
    cout<<sumar(3,4.1)<<endl;
}
```

¿Qué sucede si ingresamos dos valores de distinto tipo?

Funciones Plantilla

```
template<typename T>
T sumar(T x, T y){
    return x + y;
}
```



```
int main(){
    cout<<sumar(3.4,7)<<endl;
    cout<<sumar(3,4.1)<<endl;
}
```

¿Qué sucede si ingresamos dos valores de distinto tipo?

¿Qué opción tendríamos?

Ejemplo 1

Este es el formato
para crear una
función plantilla

```
template <class T>  
bool Max (T a, T b) {  
    return a>b;  
}
```

```
int main () {  
    int x=1, y=3  
    cout << Max<int>(x,y) << endl;  
  
    return 0;  
}
```


Ejemplo 1

La letra **T** (puede ser otra)
debe utilizarse en lugar del
nombre del tipo de dato

```
template <class T>
bool Max (T a, T b) {
    return a>b;
}
```

```
int main () {
    int x=1, y=3
    cout << Max<int>(x,y) << endl;

    return 0;
}
```

Ejemplo 1

```
template <class T>
bool Max (T a, T b) {
    return a>b;
}
```

```
int main () {
    int x=1, y=3
    cout << Max<int>(x,y) << endl;

    return 0;
}
```

En caso se requiera
comparar dos enteros,
utilizamos `<int>`

Ejemplo 1

```
template <class T>
bool Max (T a, T b) {
    return a>b;
}
```

```
int main () {
    float x=1.1, y=3.3;
    cout << Max<float>(x,y) << endl;

    return 0;
}
```

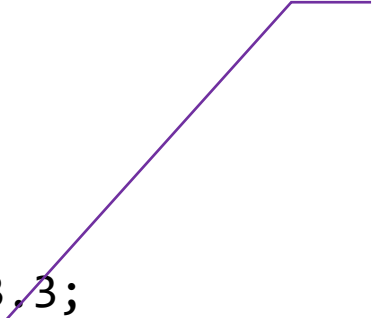
En caso se requiera
comparar dos decimales,
utilizamos `<float>`

Ejemplo 1

```
template <class T>
bool Max (T a, T b) {
    return a>b;
}

int main () {
    float x=1.1, y=3.3;
    cout << Max(x,y) << endl;

    return 0;
}
```



En el caso que el parámetro template se use como parámetro de la función, no tenemos que especificar el tipo de dato

Ejemplo 1

Podemos comparar dos
datos de distinto tipo

```
template <class T1, class T2>
bool Max (T1 a, T2 b) {
    return a>b;
}
```

```
int main () {
    float x=1.1;
    int y=3;
    cout << Max(x,y) << endl;

    return 0;
}
```

A stack of three books is shown on a light-colored wooden surface. The books are slightly offset, with the top book having a white cover and the bottom two having yellow covers. The background is a soft, out-of-focus light yellow. In the top right corner of the slide, there is a small orange horizontal bar.

Clases Plantilla

Clases Plantilla

Similar a una función plantilla, una clase plantilla permite generar distintos tipos de clase con una única máscara.

```
template<typename T>  
class MiClase{  
    ...  
};
```

Clases Plantilla

La gran diferencia con una función plantilla es que **el compilador no puede deducir** el parámetro template:

```
MiClase<int> c1;
```

```
MiClase<float> c2;
```

```
MiClase<char> c3;
```


Ejemplo 2

El parámetro template

```
template <class T>
class Pair{
    T _a, _b;
public:
    Pair(T a, T b){
        _a = a;
        _b = b;
    }
    T Max();
};
```

El parámetro template

```
template <class T>
T Pair<T>::Max () {
    return (_a>_b)? _a : _b;
}
```

Ejemplo 2

```
template <class T>
class Pair{
    T _a, _b;
public:
    Pair(T a, T b){
        _a = a;
        _b = b;
    }
    T Max();
};
```

El tipo de dato que retorna el método

```
template <class T>
T Pair<T>::Max () {
    return (_a>_b)? _a : _b;
}
```

Ejemplo 2

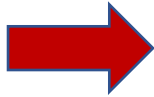
```
template <class T>
class Pair{
    T _a, _b;
public:
    Pair(T a, T b){
        _a = a;
        _b = b;
    }
    T Max();
};
```

```
template <class T>
T Pair<T>::Max () {
    return (_a>_b)? _a : _b;
}
```

El parámetro template de la clase Pair

Ejemplo 2

```
template <class T>
class Pair{
    T _a, _b;
public:
    Pair(T a, T b){
        _a = a;
        _b = b;
    }
    T Max();
};
```



```
int main () {
    Pair<float> obj(1.3, 6.2);
    cout << obj.Max() << endl;

    return 0;
}
```

```
template <class T>
T Pair<T>::Max () {
    return (_a>_b)? _a : _b;
}
```



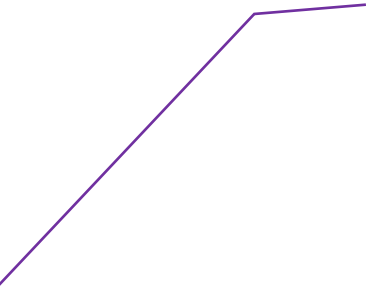
Tipos de Parámetros Template

Sintaxis

```
template <lista de parámetros>  
declaración
```

Sintaxis

template <lista de parámetros>
declaración



Cada parámetro de la lista puede ser de tres tipos:

- *type*
- *non-type*
- *template*

Parámetros Template Con Tipo

Tenemos 3:

- `type-parameter-key name(opcional)`
- `type-parameter-key name(opcional) = default`
- `type-parameter-key...name(opcional)`

Parámetros Template Con Tipo

- Un parámetro template con tipo es usado para definir luego el tipo de salida o de un parámetro.

Parámetros Template Con Tipo

- Un parámetro template con tipo es usado para definir luego el tipo de salida o de un parámetro.
- También puede ser usado para declarar o castear a un tipo de valor específico.

Parámetros Template Con Tipo

- Un parámetro template con tipo es usado para definir luego el tipo de salida o de un parámetro.
- También puede ser usado para declarar o castear a un tipo de valor específico.

```
template <typename T>  
T square(T a){  
    T r = a * a;  
    return r;  
}
```

Ahora **r** tendrá el mismo tipo que el parámetro **a**

Parámetros Template Con Tipo

En caso se requiera más de un parámetro template, cada uno debe ser precedido por la palabra `typename` o `class`:

```
template <typename T1, typename T2, class T3>
T square_sum(T1 a, T2 b, T3 c){
    T r = a * a + b * b + c * c;
    return r;
}
```

Ejemplo 3

```
template<class T = int>
T raiz(int n){
    T num = n;
    return sqrt(num);
};

int main(){
    cout << raiz<float>(3) << endl; // Resultado: 1.73205
    cout << raiz<int>(3) << endl;   // Resultado: 1
    cout << raiz(5) << endl;        // Resultado: 2
}
```

Ejemplo 3

Con un valor por defecto

```
template<class T = int>
T raiz(int n){
    T num = n;
    return sqrt(num);
};
```


```
int main(){
    cout << raiz<float>(3) << endl; // Resultado: 1.73205
    cout << raiz<int>(3) << endl;   // Resultado: 1
    cout << raiz(5) << endl;        // Resultado: 2
}
```

Parámetros Template Sin Tipo

Un parámetro template *sin tipo* representa un valor en lugar de un tipo.

Parámetros Template Sin Tipo

Un parámetro template *sin tipo* representa un valor en lugar de un tipo.



Estos se definen utilizando el tipo de dato específico en lugar de las palabras `typename` o `class`

```
template<int n>
int sum_pow(int a, int b){
    return pow(a + b, n);
}
```


Parámetros Template Sin Tipo

Un parámetro template *sin tipo* representa un valor en lugar de un tipo.

Estos se definen utilizando el tipo de dato específico en lugar de las palabras `typename` o `class`

```
template<int n>
int sum_pow(int a, int b){
    return pow(a + b, n);
}
```



```
int main(){
    cout << sum_pow(a + b, n);
}
```

Parámetros Template Sin Tipo

Tenemos cuatro:

- `type name(opcional)`
- `type name(opcional) = default`
- `type...name(opcional)`
- `placeholder name`

Ejemplo 4

```
template<int n>
struct A{
    int area = n * n;
};
template<int n = 3>
struct B{
    int area = n * n;
};
template<auto k> // C++17
struct C{};

int main(){
    A<5> a1;
    B b1;
    C<'a'> c1;
}
```

Ejemplo 4

```
template<int n>
struct A{
    int area = n * n;
};
template<int n = 3>
struct B{
    int area = n * n;
};
template<auto k> // C++17
struct C{};

int main(){
    A<5> a1;
    B b1;
    C<'a'> c1;
}
```



Con un valor por defecto

Ejemplo 4

```
template<int n>
struct A{
    int area = n * n;
};
template<int n = 3>
struct B{
    int area = n * n;
};
template<auto k> // C++17
struct C{};

int main(){
    A<5> a1;
    B b1;
    C<'a'> c1;
}
```

El **placeholder** puede ser
de cualquier tipo **auto**:
auto, auto&, auto**

Parámetro Template Template

Tenemos 3:

- `template<parameter-list> type-parameter-key name(optional)`
- `template<parameter-list> type-parameter-key name(optional) = default`
- `template<parameter-list> type-parameter-key...name(optional)`

Ejemplo 5

```
template<template<class> class C, class T>
void print(C<T> cont){
    for(auto elemento: cont)
        cout << elemento << " ";
    cout << endl;
}

int main(){
    vector<int> v1{3,2,7};
    vector<float> v2{0.1,-3.2, 1.1};
    list<char> l1 = {'a','b','c'};

    print(v1); // Resultado: 3 2 7
    print(v2); // Resultado: 0.1 -3.2 1.1
    print(l1); // Resultado: a b c
}
```

Ejemplo 5

```
template<template<class> class C, class T>
void print(C<T> cont){
    for(auto elemento: cont)
        cout << elemento << " ";
    cout << endl;
}

int main(){
    vector<int> v1{3,2,7};
    vector<float> v2{0.1,-3.2, 1.1};
    list<char> l1 = {'a','b','c'};

    print(v1); // Resultado: 3 2 7
    print(v2); // Resultado: 0.1 -3.2 1.1
    print(l1); // Resultado: a b c
}
```

Esta función imprimirá los elementos de múltiples contenedores