

C++ Templates

UNIVERSIDAD  
COMPLUTENSE  
DE MADRID

TPV 2

Lecture Notes  
Samir Genaim  
TPV2 2021/22  
Facultad de Informática  
Universidad Complutense de Madrid

Samir Genaim

# ¿Qué son los Templates?

- ★ Templates (Plantillas) es un mecanismo que se usa para conseguir algún tipo de abstracción: escribir código sólo una vez y usar en varios **contextos**
- ★ Templates normalmente se usan para abstracción de **tipos**. Lenguajes como **C++** permiten además abstracción de valores (**enteros**) constantes.
- ★ Los lenguajes de programación proporcionan varias técnicas para conseguir este tipo de abstracción, por ejemplo, **Java Generics**
- ★ En C++ hay 2 tipos de templates: **Function** y **Class**

# Function Templates



Lecture Notes  
Samir Genaim  
TPV2 2021/22  
Facultad de Informática  
Universidad Complutense de Madrid

UNIVERSIDAD  
COMPLUTENSE  
MADRID

# Function Templates — el problema

Queremos escribir una función para imprimir el doble de un entero, lo hacemos así

```
void printwice(int data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

y podemos llamarla pasando un entero

```
printwice(120); // Escribe 240
```

Si queremos hacer lo mismo para un valor de tipo **double**, tenemos que **sobrecargar** la función (copiar/pegar y cambiar **int** por **double**)

```
void printwice(double data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

# Function Templates — el problema

¿Cuál es el problema con este estilo de programación?

```
void printtwice(int data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

```
void printtwice(double data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

- ★ Las dos funciones son iguales, sólo el tipo del parámetro cambia.
- ★ Duplicar el código no es buena idea en general, porque **incrementa el coste del mantenimiento**.
- ★ Si hay un error en una implementación tenemos que corregirlo en todas las versiones.

# Function Templates: La solución

¿Cuál sería una solución mejor? Escribir una función paramétrica en el tipo (plantilla) como la siguiente

```
void printwice(TYPE data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

y dejar al compilador la tarea de generar varias versiones (**instantiate the template** — instanciar la plantilla), depende de su uso

```
printwice(100);      // TYPE=int  
printwice(10.33);    // TYPE=double
```

# Function Templates en C++

En C++ se puede hacer exactamente esto usando la keyword **template**:

```
template<typename T>
void printtwice(T data) {
    cout << "Twice is: " << data * 2 << endl;
}
```

Ahora, el compilador genera varias versiones depende del tipo del valor que pasamos a **printtwice**. Por ejemplo, para estas llamadas genera 2 versiones

```
printtwice(100);      // TYPE=int
printtwice(10.33);    // TYPE=double
```

# Function Templates en C++

En C++ se puede usar la keyword **typename** o **class** con el parámetro de tipo

```
template<class T>  
// ...
```

```
template<typename T>
```

Son casi iguales, se puede usar una o otra, la diferencia no es importante para lo que vamos a ver de templates, siempre vamos a usar **typename**.

Si buscáis en Google encontráis muchas discusiones sobre el tema, p.ej., para resolver ambigüedad en la sintaxis ...

# Compilación de Templates en C++

La compilación se realiza en 2 pasos:

1. **Instanciación (instantiation)**: el compilador infiere los tipos de los datos correspondientes en las llamadas y genera una implementación para cada versión distinta (y cambia las llamadas para que usen esas versiones)
2. **Compilación**: compilar todo el código (que ahora no incluye templates)

**¡Sí!** claro que hay duplicación de código, pero no hay duplicación del mantenimiento del código ...

# ¿Qué Escribe este Código?

```
template<typename T>
void printdata(T data) {
    cout << "Data: " << data << endl;
}
```

```
typedef struct {
    int x;
    int y;
} point;
```

```
int main() {
    point p = {2,3};
    printdata(100);
    printdata(102.22);
    printdata("Hola!");
    printdata(p);
}
```

Pasa la fase de instanciación, pero falla en la compilación, porque el operador `<<` (en `cout`) no puede tratar el tipo `point`.

Se resuelve definiendo el operador `<<` para el tipo `point`, por ejemplo:

```
ostream& operator<<(ostream& out, const point& p) {
    out << "(" << p.x << "," << p.y << ")" << endl;
    return out;
}
```

# Salida de tipo T

```
template<typename T>
T twice(T data) {
    return 2 * data;
}
```

```
int main() {
    cout << twice(10) << endl;
    cout << twice(3.14) << endl;
    cout << twice( twice(55) ) << endl;
}
```

Se puede usar el parámetro de tipo para cualquier declaración de tipo, p.ej., para la salida

El compilador genera sólo una instancia para cada tipo, si usamos la función 2 veces con el mismo tipo se genera sólo una ...

# Punteros, Referencias, Arrays, ...

```
template<typename T>
double ave(T tArray[], int nElements) {
    T tSum = T(); ←
    for (int nIndex = 0; nIndex < nElements; ++nIndex) {
        tSum += tArray[nIndex];
    }
}
```

constructora por defecto de T

T tiene que definir el operador +=

```
return double(tSum) / nElements;
}
```

T tiene que ser convertible a double

```
int main() {
    int IntArray[5] = { ... };
    float FloatArray[3] = { ... };
}
```

El compilador infiere que T puede ser int/float.

```
}
```

```
cout << ave(IntArray, 5) << endl;
cout << ave(FloatArray, 3) << endl;
```

# Punteros, Referencias, Arrays, ...

Se puede usar & (y const) cuando sea necesario ...

```
template<typename T>
```

```
void
```

```
template<typename T>
```

```
+1
```

```
T& getMax(T& t1, T& t2) {
```

```
    if (t1 > t2) return t1;
```

```
    else return t2;
```

```
}
```

```
Twice
```

```
...
```

```
int x
```

```
int y
```

```
GetM
```

```
template<typename T>
```

```
ave(const T +Array[] int nElements){
```

```
...
```

```
}
```

```
template<typename T>
```

```
void printwice(const T& data) {
```

```
    cout<<"Twice: " << data * 2 << endl;
```

```
}
```

# Variables locales de tipo T

```
template<typename T>
T add(T n1, T n2) {
    T result;
    result = n1 + n2;
    return result;
}
```

```
int main() {
    cout << add(10,11) << endl;
    cout << add(4.3,3.14) << endl;
}
```

Se puede usar el parámetro de tipo también para definir variables locales.

Qué pasa si añadimos este código al main?

```
cout << add(1,3.14) << endl;
```

# Inferencia de tipos

```
template<typename T>  
void f(T o) { ... }
```

```
template<typename T>  
void g(T &o) { ... }
```

```
int main(int ac, char **av) {
```

```
    A x;
```

```
    const A y;
```

```
    f(x);
```

```
    f(y);
```

```
    g(x);
```

```
    g(y);
```

```
}
```

C++ usa varios algoritmos de inferencia de tipos para inferir los parámetros de tipo, etc. También se usan para inferir los tipos de `auto`

T=A, el tipo del parámetro de f es A

T=A, el tipo del parámetro de f es A

T=&A, el tipo del parámetro de f es &A

T=const A, el tipo del parámetro de f es const &A

# Inferencia de tipos

```
template<typename T>
void f(TypeParam a) {
    ...
}
```

En general, los algoritmos **de inferencia de tipos** infieren los valores de **T** y **TypeParam** depende del uso y de la forma de **TypeParam**, etc. No vamos a entrar en muchos detalles, pero veremos algunos casos cuando sea necesario...

Es muy recomendable leer el capítulo 1 (Deducing Types) del libro Effective Modern C++ de Scott Meyers para entender la inferencia de tipos

<https://www.oreilly.com/library/view/effective-modern-c/9781491908419/ch01.html>

# Varios parámetros de tipo

Se puede usar varios parámetros de tipo:

`template<typename T1, typename T2, ...>`

```
template<typename T1, typename T2>
void printNumbers(const T1& t1Data, const T2& t2Data) {
    cout << "First value:" << t1Data << endl;
    cout << "Second value:" << t2Data << endl;
}

int main() {
    int x = 1;
    double y = 2.3;
    printNumbers(x, y);
    printNumbers(y, x);
}
```

# Argumentos de tipo explícitos

Se puede decir al compilador de manera explícita qué tipo tiene que usar para cada parámetro (o para parte de ellos). En ese caso se intenta a convertir entre los tipos si no son iguales.

```
template<typename T1, typename T2>
void printNumbers(const T1& t1Data, const T2& t2Data) {
    cout << "First value:" << t1Data << endl;
    cout << "Second value:" << t2Data << endl;
}

int main() {
    int x = 1;
    double y = 2.3;
    printNumbers<double,double>(x, y);
    printNumbers<double,double>(y, x);
}
```

Se genera sólo una versión

# Argumentos de tipo explícitos

Se puede usar tipos explícitos para resolver problemas de compatibilidad de tipos ...

```
template<typename T>
T getMax(T t1, T t2) {
    if (t1 > t2)
        return t1;
    return t2;
}

int main() {
    //cout << getMax(1,2.2) << endl; // ERROR
    cout << getMax<double>(1,2.2) << endl;
}
```

incompatibilidad de tipos

Usar double para T, los valores enteros se convierten a double

# Argumentos de tipo explícitos

Se usa para **pasar tipos como parámetros**, vamos a usarlo mucho ...

```
class A { ... };  
class B { ... };  
  
template<typename T>  
T* create(int x) {  
    return new T(x);  
}  
  
int main(int ac, char** av) {  
    A* a = create<A>(1);  
    B* b = create<B>(2);  
  
    ...  
}
```

Si no decimos explícitamente que es T, el compilador no puede inferirlo ...

# Argumentos de tipo explícitos

Se usa para **pasar tipos como parámetros**, vamos a usarlo mucho ...

```
template<typename T>
void printSize() {
    cout << "Size of this type:" << sizeof(T) << endl;
}

int main() {
    printSize<float>();
    printSize<int>();
    printSize<double>();
}
```

Si no decimos explícitamente que es T, el compilador no puede inferirlo ...

# Argumentos de tipo explícitos

Se puede usar cuando el tipo de la salida no se puede inferir automáticamente ...

```
template<typename T>
T sumOfNumbers(int a, int b) {
    T t = T(); // Call default CTOR for T
    t = T(a)+b;
    return t;
}

int main() {
    double sum;
    sum = sumOfNumbers<double>(120,200);
    cout << sum << endl;
}
```

Si no decimos explícitamente que es T, el compilador no puede inferirlo

# Establecer valores por defecto ...

Se puede usar el parámetro de tipo para establecer valores por defecto para los argumentos ...

```
template<typename T>
void printNumbers(T array[], int array_size, T filter = T()) {
    for(int nIndex = 0; nIndex < array_size; ++nIndex) {
        if (array[nIndex] != filter) // Print if not filtered
            cout << array[nIndex] << endl;
    }
}

int main() {
    int a[10] = {1,2,0,3,4,2,5,6,0,7};
    printNumbers(a, 10);
    printNumbers(a, 10, 2); ←
}
```

Se genera sólo una versión

# Class Templates

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier medio durante la carrera sólo se podrán utilizar para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación o distribución posterior incluida la divulgación en redes sociales y servicios de compartición en Internet) puede vulnerar la normativa de protección de datos y/o la de propiedad intelectual y generar responsabilidad de la persona infractora. Si encuentras este material en otro sitio web que no tenga la extensión ucm.es, avísanos en denunciacontenido@ucm.es o reportcontent@ucm.es

# ¿Qué son los Class Templates?

Definir una clase `List<T>`, donde `T` es un parámetro de tipo, y usarla como `List<int>`, `List<double>`, `List<A>`, etc.

- ◆ Normalmente se usa para definir tipos abstractos de datos con comportamientos genéricos, que se puede reusar/adaptar para distintos tipos de datos
- ◆ Ya estáis usando class templates cuando usáis por ejemplo `vector<int>`

# Un Ejemplo Sencillo ...

```
class Item {  
    int data;  
public:  
    Item() : data(0) {}  
  
    void setData(int nValue) {  
        data = nValue;  
    }  
  
    int getData() const {  
        return data;  
    }  
};  
  
int main() {  
    Item item1;  
    item1.setData(120);  
    cout << item1.getData() << endl;  
}
```

Una clase para representar un elemento de tipo entero

¿y si necesitamos otra para representar un double?

Una solución: copiar y pegar. Pero ya sabemos que no es una buena idea.

Mejor solución: definir la clase sin especificar el tipo, y usarla con varios tipos

# Un Ejemplo Sencillo ...

```
template<typename T>
class Item {
    T data;
public:
    Item() : data( T() ) { }
    void setData(const T& nValue) {
        data = nValue;
    }
    T getData() const {
        return data;
    }
};

int main() {
    Item<int> item1;
    Item<double> item2;
    item1.setData(120);
    item2.setData(10.33);
}
```

Como en el caso de function templates, usamos  
template<typename T>

Usamos la clase indicando el tipo para instanciar T. El compilador genera varias versiones ...

Compilación en dos pasos ...

No hay ninguna relación entre los tipos Item<int> y Item<double>

# Ejercicio: Implementa la clase Pair

Implementa la clase Pair para que este código funcione correctamente ...

```
int main() {  
    Pair<int,int> x(1,2);  
    Pair<int,int> y(x);  
    Pair<double,string> z(1.2,"Hola");  
  
    if ( x == y )  
        cout << "They are equal!" << endl;  
  
    cout << x.getFirst() << " " << x.getSecond() << endl;  
    cout << y.getFirst() << " " << y.getSecond() << endl;  
    cout << z.getFirst() << " " << z.getSecond() << endl;  
}
```

Pair es un template class con 2 parámetros de tipo

Sobrecarga el operator ==

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier medio durante la carrera sólo se podrán utilizar para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación o distribución posterior (incluida la digitalización) de estos materiales o servicios de comparación en Internet, de datos y/o la de propiedad intelectual y generan responsabilidad de la persona infractora. Si encuentras esta material en otro sitio web, avísalos en denunciacontenido@ucm.es o reportcontent@ucm.es

# Non-type Template Arguments

```
template<typename T, int SIZE=100>
class MyArray {
    T elem[SIZE];
public:
    MyArray() { }

    T& operator[](int i) {
        return elem[i];
    }
};
```

```
int main() {
    MyArray<int, 10> x;
    MyArray<string, 20> y;
    MyArray<float> z;

    x[1] = 10;
    ...
}
```

Se puede usar argumentos que no son tipos (como se fueran constantes), con o sin valor por defecto. Se permiten sólo tipos enteros.

int,10

string,20

float,100

# Ejercicio: Implementa clase MyVector

Implementa una clase MyVector para que este código funcione correctamente ...

```
int main() {  
    MyVector<int> v;  
  
    v.push_back(11);  
    v.push_back(15);  
    v.push_back(17);  
    cout << v.size() << endl;  
  
    cout << v[0] << " " << v[1] << v[2] << endl;  
    v[1] = 4;  
    cout << v[0] << " " << v[1] << " " << v[2] << endl;  
}
```

# Valores por defecto ...

Definir un parámetro en términos de otro que aparece a su izquierda

```
template<typename T1=int, typename T2 = T1>
class Pair {
    T1 first;
    T2 second;
};
```

```
template<typename T, int ROWS=8, int COLUMNS=ROWS>
class Matrix {
    T TheMatrix[ROWS][COLUMNS];
};
```

# Métodos como Function Templates

```
template<typename T>
class MyVector {
```

...

```
template<typename T1>
void copy(T1 a[]) {
    for(auto i=0u; i<size_; i++) {
        a[i] = *elem_[i];
    }
}
```

T y T1 no tienen ninguna relación ...

```
int main() {
```

```
    MyVector<int> v;
```

...

```
    int a[10];
    v.copy(a);
```

```
    float b[10];
    v.copy(b);
```

}

Añade un método  
void copy(int a[])  
a MyVector<int>.

Añade un método  
void copy(float a[])  
a MyVector<int>.

Si no usamos el método  
copy no se añade nada a  
MyVector<int>

# Herencia de Templates: I

```
template<typename T>
class A {
public:
    A() {}
    virtual ~A() {}
    virtual void foo(T x) = 0;
    virtual T moo(T x) = 0;
};
```

```
int main() {
    C<int> y;
    y.moo(1);
    ...
}
```

```
template<typename T>
class C : public A<T> {
public:
    C() {}
    virtual ~C() {}
    virtual void foo(T x) {
        // ...
    }
    virtual T moo(T x) {
        // ...
    }
};
```

# Herencia de Templates: II

```
template<typename T>
class A {
public:
    A() {}
    virtual ~A() {}
    virtual void foo(T x) = 0;
    virtual T moo(T x) = 0;
};
```

```
int main() {
    B x;
    x.foo(1);
    ...
}
```

```
class B: public A<int> {
public:
    B() {}
    virtual ~B() {}
    virtual void foo(int x) {
        virtual int moo(int x) {
```

# Parameter Pack



Lecture Notes  
Samir Genaim  
TPV2 2021/22  
Facultad de Informática  
Universidad Complutense de Madrid

UNIVERSIDAD  
COMPLUTENSE  
MADRID

# Packs: Type template parameter

Se usa para definir templates con un número variable de parámetros.

```
template<typename ...Ts>
void f(Ts...args) {
    // some code
    g(args...)
    // some code
}
```

**Template parameter pack:**  
Una llamada a f tiene que proporcionar cero o más tipos. Ts refiere a un pack de estos tipos (T1, T2, etc)

## Parameter pack expansion:

Se convierte en la secuencia

a1, a2, etc.

El termino técnico para los tres puntos ... es Ellipsis

**Function parameter pack:**  
se convierte en la secuencia T1 a1, T2 a2, etc. El nombre args es un pack de los nombres a1, a2, ... (no pueden usar directamente)

# Ejemplo

Si  $Ts$  es  $T1, T2, T3$ . Se expande a  
 $T^*$  `create(T1 a1, T2 b2, T3 c3)`  
No se puede usar los nombres  
 $a1, a2, a3$  directamente.

Se puede instanciar con 1 ( $T$ )  
o mas ( $Ts$ ) parámetros de  
tipo.  $Ts$  refiere a un número  
desconocido de parámetros.

```
template<typename T, typename ...Ts>
T* create(Ts...args) {
    return new T(args...);
}
```

Se expande a  
`return new T(a1,a2,a3);`

$...Ts$  siempre va al final de la  
declaración del template, p.ej.  
`<typename ...Ts, typename T>`  
no compila

# Ejemplo

```
template<typename T, typename ...Ts>
T* create(Ts...args) {
    return new T(args...);
}
```

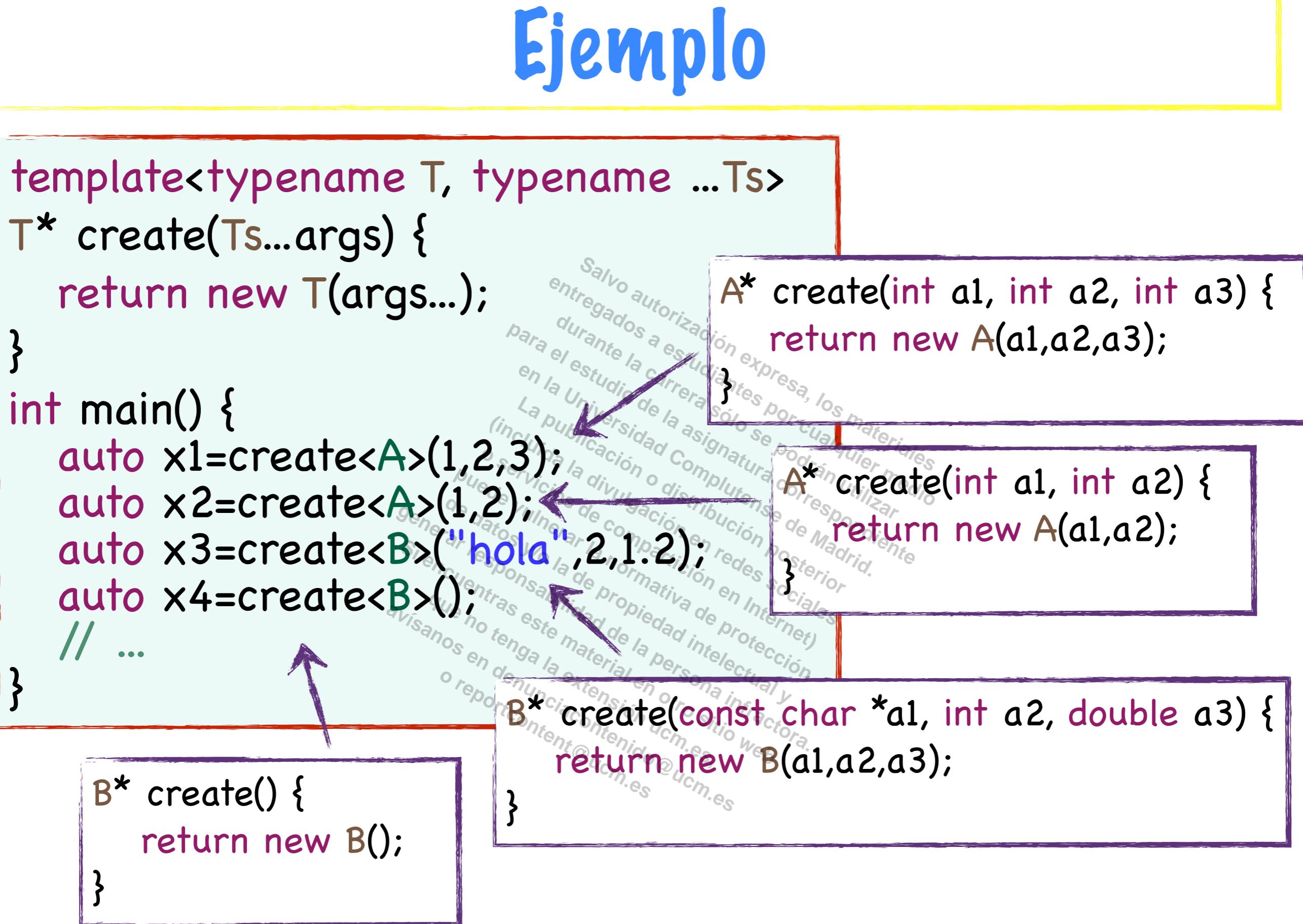
```
int main() {
    auto x1=create<A>(1,2,3);
    auto x2=create<A>(1,2);
    auto x3=create<B>("hola",2,1.2);
    auto x4=create<B>();
    // ...
}
```

```
B* create() {
    return new B();
}
```

```
A* create(int a1, int a2, int a3) {
    return new A(a1,a2,a3);
}
```

```
A* create(int a1, int a2) {
    return new A(a1,a2);
}
```

```
B* create(const char *a1, int a2, double a3) {
    return new B(a1,a2,a3);
}
```



# Function parameter pack

```
void f(Ts... args)  
void f(Ts&... args)  
void f(const Ts&... args)
```

Se expande a

T<sub>1</sub> a<sub>1</sub>, T<sub>2</sub> a<sub>2</sub>, ...

dónde T<sub>i</sub> se genera a partir el i-ésimo elemento del pack Ts y el patrón que aparece a la izquierda del **ellipsis**. El nombre “args” es un pack que contiene los nombre de variables a<sub>1</sub>, a<sub>2</sub>, ...

Por ejemplo, si Ts es int, float, std::string obtenemos lo siguiente

```
void f(int a1, float a2, std::string a3)  
void f(int &a1, float &a2, std::string &a3)  
void f(const int &a1, const float &a2, const &std::string a3)
```

# Parameter pack expansion

pattern...

pattern es una expression que incluye uno o varios packs, y se expande a

p1,p2,p3,...

donde pi se obtiene a partir de pattern, usando el i-esimo valor en los packs que aparecen en pattern

f(args...) → f(a1,a2,a3)  
f(&args...) → f(&a1,&a2,&a3)  
f((int)args...) → f((int)a1,(int)a2,(int)a3)  
f(g(args)...)

→ f(g(a1),g(a2),g(a3))

# Ejemplo de uso

```
template<typename T>
class MyVector {
...
void push_back(const T& x) {
...
    elem_[size_++] = new T(x);
}
```

A x(1,2);  
v.push\_back(x);

Crea 2 objetos y requiere que la clase A tenga una constructora de copia

```
template<typename ...Ts>
void emplace_back(Ts... args) {
...
    elem_[size_++] = new T(args...);
}
...
```

v.emplace\_back(1,2);

Crea la instancia usando los parámetros

# Java Generics vs C++ Templates

```
public class MyVector<T extends Object> {  
  
    private final static int _INIT_SIZE = 2;  
    public T[] elem;  
    private int last;  
  
    public MyVector() {  
        last = -1;  
        elem = (T[]) new Object[_INIT_SIZE];  
    }  
  
    public void addElem(T e) {  
        if (last == elem.length - 1) {  
            elem = Arrays.copyOf(elem, elem.length * 2);  
        }  
        elem[++last] = e;  
    }  
  
    public T getElem(int index) {  
        if (index > last) return null;  
        else return elem[index];  
    }  
}
```

```
MyList<Integer> x = new MyList<Integer>();  
x.addElement(100);  
Integer y = x.getElement(0);
```

```
MyList<String> x = new MyList<String>();  
x.addElement("Hola!");  
String y = x.getElement(0);
```

```
public class MyList {  
  
    private final static int _INIT_SIZE = 2;  
    public Object[] elem;  
    private int last;  
  
    public MyList() {  
        last = -1;  
        elem = (Object[]) new Object[_INIT_SIZE];  
    }  
  
    public void addElem(Object e) {  
        if (last == elem.length - 1) {  
            elem = Arrays.copyOf(elem, elem.length * 2);  
        }  
        elem[++last] = e;  
    }  
  
    public Object getElement(int index) {  
        if (index > last) return null;  
        else return elem[index];  
    }  
}
```

```
MyList x = new MyList();  
x.addElement(100);  
Integer y = (Integer) x.getElement(0);
```

```
MyList x = new MyList();  
x.addElement("Hola!");  
Integer y = (String) x.getElement(0);
```