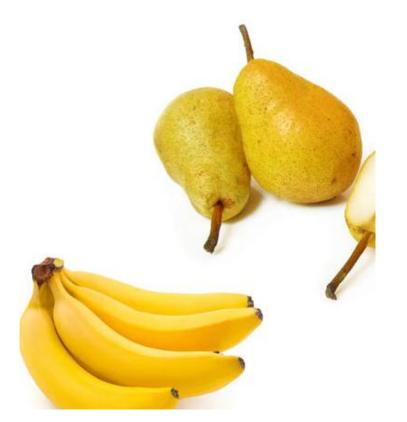
# Templates en C++

Algo2 - C1 - 2019

### Motivación

 Nos gustaría implementar una clase que nos permita asociar dos valores no necesariamente del mismo tipo



¡Si! Peras con bananas.

¡Pero en prog1 nos dijeron que no se podía!

A menos que...

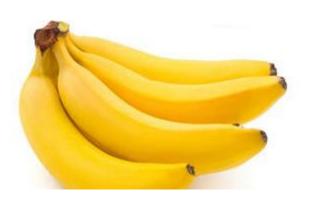
### Motivación

No hay problema

```
class Ensalada {
  private:
    Pera _pera;
    Banana _banana;
};
```

### Motivación





```
¿y si queremos
Manzanas
con
Bananas?
```

### Dominio de variables

- Vemos que esta forma de diseñar no escala
- Necesitamos una clase que generalice esta idea
- Antes, recordemos el dominio de nuestro tipo de variable Ensalada; asumiremos que hay dos instancias posibles de peras y bananas.

#### Dominio de variables

- Necesitaremos una variable especial para poder generalizar cualquier "ensalada". Por ej T.
- Estas variables se denominan variables de tipo y su dominio es:

Notar que null no esta en lista por no ser un tipo de C++ ni definible por el usuario. Sin embargo una ensada puede ser null.

### **Templates**

- C++ implementa Variables de Tipo a través de Templates
- Generalizaremos todas las ensadalas posibles de dos elementos con una clase "Template".

### Ensalada

```
Ensalada.hpp
  template<Class T1, clase T2>
  class Ensalada {
                                            template<class T1, class T2>
           public:
                                            Ensalada <T1, T2>:: Ensalada(T1 t1, T2 t2)
            Ensalada(T1 t1, T2 t2);
                                                     : t1(t1), t2(t2) {}
           T1 t1() const;
           T2 t2() const;
                                            template<class T1, class T2>
           private:
                                            T1 Ensalada <T1, T2>::t1() const{
           T1 t1;
                                                     return t1;
           T2 t2;
  };
T1 ,T2 toman un valor
                                            template<class T1, clase T2>
concreto en tiempo de
                                            T2 Ensalada <T1, T2>::t2() const{
compilación.
                                                     return t2;
C++ genera una versión para
```

cada valor de T1,T2

### Abstracción

 Como en algo2 también apuntamos a una mayor abstracción, utilizaremos:

```
Tupla.hpp
                                                   template<class T1, class T2>
                                                   Tupla<T1, T2>:: Tupla(T1 t1, T2 t2)
template<class T1, class T2>
                                                             : t1(t1), t2(t2) {}
class Tupla {
          public:
          Tupla(T1 t1, T2 t2);
                                                   template<class T1, class T2>
          T1 t1() const;
                                                   T1 Tupla<T1, T2>::t1() const{
          T2 t2() const;
                                                             return t1;
          private:
          T1 t1;
          T2 t2;
                                                   template<class T1, class T2>
                                                   T2 Tupla<T1, T2>::t2() const{
};
                                                             return t2;
```

¡Ciudado! A pesar que defina == en Pera, Manzana y Banana no podre comparar peras con bananas.

¿Podre comparar ensaladas?

```
bool operator==(Tupla t) const;
 template<class T1, class T2>
 bool Tupla<T1, T2>::operator==(Tupla t) const{
        return t1==t.t1() \&\& t2==t.t2()
 Luego, si T1 y T2 tienen el operador == definido
 podremos preguntarnos si dos instancias de tupla:
                         t1 == t2
Recordar que ==
1) es una relación de equivalencia:
  reflexiva, simétrica, transitiva
```

2) tiene que modelar la igualdad observacional

¿Qué pasa si T1 no tiene definido ==?

Error en tiempo de compilación

# Limitaciones / Compilación

El santo grial de los Templates tiene limitaciones



El compilador no tiene *binding dinámico*, entonces no puede inferir T en tiempo de ejecución.

### Además

```
#ifndel TUPLA HPP
                                                       template<class T1, class T2>
#define TUPLA HPP
                                                       Tupla<T1, T2>:: Tupla(T1 t1, T2 t2)
                                                                  : t1(t1), t2(t2) {}
template<class T1, class T2>
Class Tupla {
                                                       template<class T1, class T2>
                                                       T1 Tupla<T1, T2>::t1() const{
           public:
           Tupla(T1 t1, T2 t2);
                                                                  return t1;
           T1 t1() const;
           T2 t2() const;
                                                       template<class T1, class T2>
           bool operator==(Tupla t) const;
                                                       T2 Tupla<T1, T2>::t2() const{
                                                                  return t2;
           private:
           T1 t1;
           T2 _t2;
                                                       int main() {
                                                                  Banana b;
                                                                  Pera p;
template<class T1, class T2>
                                                                  Tupla<Banana, Pera>
bool Tupla<T1, T2>::operator==(Tupla t) const{
                                                       ensalada1(b,p);
           return t1==t.t1() \&\& t2==t.t2()
                                                       #endif
```

### Finalmente

```
#include Tupla.hpp

int main() {
         Banana b;
         Pera p;
         Manzana m;
         Tupla<Banana, Pera> ensalada1(b,p);
         Tupla<Banana, Manzana> ensalada2(b,m);
}
```

Esto SI compila, porque se genera una versión particular para cada Tupla:
Tupla<Banana, Pera>
Tupla<Banana, Manzana>

### Convención

.h: headers

.cpp: implementaciones

.hpp: templates(todo)

-Notar que no tenemos headers, nuevamente

por una limitación del lenguaje)

### Buenas practicas

Para los TAD definidos por el usuario es recomendable sobrecargar << (siempre que tenga sentido hacerlo).

Por ejemplo, es recomendable dejar una version "imprimible" de nuestra Tupla y por transitividad de nuestra ensalada.

```
Tupla.h

#include <iostream>
...

ostream& operator<<(ostream&, const Tupla& t);

Tupla.cpp

ostream& operator<< (ostream& os, const Tupla& t) {
    os << t.t1() << t.t2() << '\n';
    return os;
}
```