

Templates

Algoritmos y Estructuras de Datos II

Template

Inglés detectado ▾

template

Editar

'templət

Español ▾

modelo

sustantivo **la plantilla**
template, stencil, insole, jig,
sock, slipsole

Mostrar menos


max.h

```
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

double max(double a, double b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

#define user_type char
user_type max(user_type a, user_type b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

max.h'




```
char max(char a, char b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

max.h

```
user_type max(user_type a, user_type b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

todos_loos_max.h

```
#define user_type int  
#include "max.h"  
#define user_type double  
#include "max.h"  
#define user_type char  
#include "max.h"
```



todos_loos_max.h'

```
int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
double max(double a, double b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
char max(char a, char b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
template<class T>
```

- ▶ Antes de cada declaración o definición.
- ▶ Solo afecta esa declaración o definición.
- ▶ En ese contexto, T es un tipo de datos, solo que no sabemos cuál.

```
template<class T>
```

```
T max(T a, T b);
```

```
template<class T>
```

```
bool esMayor(T x, T y);
```

```
template<class K>
```

```
bool esMayor(K x, K y) {
```

```
    return x > y;
```

```
}
```

max.h

```
template<class C>
C max(C a, C b);
```

max.cpp

```
template<class C>
C max(C a, C b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

max.o

main.cpp

```
#include "max.h"
int main() {
    int x = 5;
    int y = 6;
    int j = max(x, y);
}
```

main.o

main



max.h

```
template<class C>  
C max(C a, C b);
```

max.cpp

```
template<class C>  
C max(C a, C b) {  
    if (a < b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

max.o

main.cpp

```
#include "max.h"  
int main() {  
    int x = 5;  
    int y = 6;  
    int j = max(x, y);  
}
```

main.o

main



```
graph LR; maxcpp[max.cpp] --> maxo[max.o]; maincpp[main.cpp] --> maino[main.o]; maxo --> main[main]; maino --> main;
```

main.cpp:(.text+0x21): undefined reference to 'int max<int>(int, int)'

max.h

```
template<class C>  
C max(C a, C b);
```

max.cpp

```
template<class C>  
C max(C a, C b) {  
    if (a < b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

max.o

main.cpp

```
#include "max.h"  
int main() {  
    int x = 5;  
    int y = 6;  
    int j = max(x, y);  
}
```

main.o

main



main.cpp:(.text+0x21): undefined reference to 'int max<int>(int, int)'
Al compilar max.cpp, ¿qué tipo se usa para C?

max.hpp

```
template<class C>
C max(C a, C b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

main.cpp

```
#include "max.hpp"
int main() {
    int x = 5;
    int y = 6;
    int j = max(x, y);
}
```

main.o

main

max.hpp

```
template<class C>
C max(C a, C b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

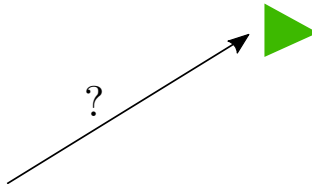
trimax.hpp

```
#include "max.hpp"
template<class C>
C trimax(C a, C b, C c) {
    return max(max(a, b), c);
}
```

main.cpp

```
#include "max.hpp"
#include "trimax.hpp"
int main() {
    int x = 5;
    int y = 6;
    int z = 10;
    int j = max(x, y);
    int h = trimax(x, y, z);
}
```

main



```
march@elaine:~/algoritmos2/clases/labo/03_templates/ejemplos_codigo/compile_2$ g++ main.cpp -o main
In file included from trimax.hpp:2:0,
                 from main.cpp:3:
max.hpp:3:3: error: redefinition of 'template<class C> C max(C, C)'
  C max(C a, C b) {
    ^
In file included from main.cpp:2:0:
max.hpp:3:3: note: 'template<class C> C max(C, C)' previously declared here
  C max(C a, C b) {
    ^
```

max.hpp

```
#ifndef MAX_HPP
#define MAX_HPP
template<class C>
C max(C a, C b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
#endif
```

trimax.hpp

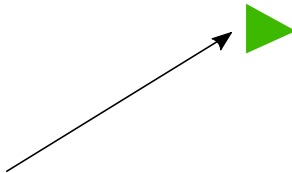
```
#ifndef TRIMAX_HPP
#define TRIMAX_HPP

#include "max.hpp"
template<class C>
C trimax(C a, C b, C c) {
    return max(max(a, b), c);
}
#endif
```

main.cpp

```
#include "max.hpp"
#include "trimax.hpp"
int main() {
    int x = 5;
    int y = 6;
    int z = 10;
    int j = max(x, y);
    int h = trimax(x, y, z);
}
```

main



(Notar los `##ifndef`).

Par.cpp

```
Par::Par(int izq, char der) : _izq(izq), _der(der) {  
}  
  
int Par::primero() const {  
    return _izq;  
}  
  
char Par::segundo() const {  
    return _der;  
}
```

Par.h

```
class Par {  
public:  
    Par(int izq, char der);  
    int primero() const;  
    char segundo() const;  
  
private:  
    int _izq;  
    char _der;  
};
```

main.cpp

```
#include "Par.cpp"  
  
int main() {  
    Par p(5, 'c');  
    Par p2(10, 'd');  
}
```

Par.h

```
class Par {
public:
    Par(int izq, char der);
    int primero() const;
    char segundo() const;

private:
    int _izq;
    char _der;
};
```

Par.cpp

```
Par::Par(int izq, char der)
    : _izq(izq), _der(der) { }

int Par::primero() const {
    return _izq;
}

char Par::segundo() const {
    return _der;
}
```

par.hpp

```
template<class Tizq, class Tder>
class Par {
public:
    Par(Tizq izq, Tder der);
    Tizq primero() const;
    Tder segundo() const;

private:
    Tizq _izq;
    Tder _der;
};

template<class Tizq, class Tder>
Par<Tizq, Tder>::Par(Tizq izq, Tder der)
    : _izq(izq), _der(der) { }

template<class Tizq, class Tder>
Tizq Par<Tizq, Tder>::primero() const {
    return _izq;
}

template<class Tizq, class Tder>
Tder Par<Tizq, Tder>::segundo() const {
    return _der;
}
```

par.hpp

```
template<class Tizq, class Tder>
class Par {
public:
    Par(Tizq izq, Tder der);
    Tizq primero() const;
    Tder segundo() const;

private:
    Tizq _izq;
    Tder _der;
};

template<class Tizq, class Tder>
Par<Tizq, Tder>::Par(Tizq izq, Tder der) : _izq(izq), _der(der) {
}

template<class Tizq, class Tder>
Tizq Par<Tizq, Tder>::primero() const {
    return _izq;
}

template<class Tizq, class Tder>
Tder Par<Tizq, Tder>::segundo() const {
    return _der;
}
```

```
#include "par.hpp"
#include <string>

using namespace std;

int main() {
    Par<int, int> p(5, 10);
    Par<char, int> q('d', 15);
    Par<string, string> r("Hola", "mundo");
    Par<Par<string, int>, char> x(Par<string, int>("Cinco", 5), '5');
}
```

¿Puedo usar cualquier tipo para un template?

```
template<class T>
T max(T a, T b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

```
int main() {
    int m_int = max(3, 5);
    Par<int, int> m_par = max(Par<int, int>(10, 2),
                             Par<int, int>(4, 10));
}
```


¿Puedo usar cualquier tipo para un template?

```
template<class T>
T max(T a, T b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

```
int main() {
    int m_int = max(3, 5);
    Par<int, int> m_par = max(Par<int, int>(10, 2),
                             Par<int, int>(4, 10));
}
```

```
130 march@elaine:~/algoritmos2/clases/labo/03_templates/ejemplos_codigo) g++ tipos.cpp -o tipos
tipos.cpp: In instantiation of 'T max(T, T) [with T = Par<int, int>]':
tipos.cpp:15:51:   required from here
tipos.cpp:5:11: error: no match for 'operator<' (operand types are 'Par<int, int>' and 'Par<int, int>')
    if (a < b) {
        ^
```

Constructores y Lista de Inicialización (repaso)

Algoritmos y Estructuras de Datos II

Conceptos

- ▶ Constructor por defecto: `T::T()`
- ▶ Constructor por copia: `T::T(const T&)`
- ▶ Destructor: `T::~~T()`
- ▶ Operador asignación: `T::operator=(const T&)`

Constructor por copia

¿Qué pasaría si Lista no tuviera constructor por copia?

```
class Lista {  
public:  
    Lista();  
    Lista(const Lista& otra);  
  
    void agregarAtras(T&);  
    int longitud() const;  
    ...  
  
private:  
    struct Nodo {  
        T valor;  
        Nodo* siguiente;  
    }  
  
    Nodo* _primero;  
}
```

```
int main() {  
    Lista<int> l1;  
    l1.agregarAtras(1);  
    Lista<int> l2(l1);  
    l2.agregarAtras(2);  
    l1.longitud(); // ??  
}
```

Constructor por copia

¿Qué pasaría si Lista no tuviera constructor por copia?

```
class Lista {  
public:  
    Lista();  
    Lista(const Lista& otra);  
  
    void agregarAtras(T&);  
    int longitud() const;  
    ...  
  
private:  
    struct Nodo {  
        T valor;  
        Nodo* siguiente;  
    }  
  
    Nodo* _primero;  
}
```

```
int main() {  
    Lista<int> l1;  
    l1.agregarAtras(1);  
    Lista<int> l2(l1);  
    l2.agregarAtras(2);  
    l1.longitud(); // ??  
}
```

¡Pizarrón!

¿Dónde, dónde está el constructor por copia?

¿Dónde se llama al constructor por copia en este ejemplo?

```
int maximo(Lista<int> l) {  
    int max = l[0];  
    for (int i = 1; i < l.longitud(); i++) {  
        if (l[i] > max) {  
            max = l[i];  
        }  
    }  
    return max;  
}
```

```
Lista<int> l1;  
l1.agregarAtras(1);  
l1.agregarAtras(3);  
l1.agregarAtras(2);  
int m = maximo(l);
```

¿Y en este?

```
Lista<int> l1;  
l1.agregarAtras(1);  
Lista<int> l2 = l1;
```


¿Y en esteeee?

```
Lista<int> rango(int desde, int hasta) {  
    Lista<int> ret;  
    for (int i = desde; i < hasta; i++) {  
        ret.agregarAtras(i);  
    }  
    return ret;  
}
```

```
Lista r = rango(5, 25);
```

¿Y en este otro?

```
Lista<int> l1;  
l1.agregarAtras(1);  
l1.agregarAtras(10);  
Lista<int> l2;  
l2 = l1;
```

¿Qué pasaría si Lista no tuviera operador de asignación?

```
class Lista {  
public:  
    Lista();  
    Lista(const Lista& otra);  
    Lista& operator=(const Lista& otra);  
  
    void agregarAtras(T&);  
    int longitud() const;  
    ...  
  
private:  
    struct Nodo {  
        T valor;  
        Nodo* siguiente;  
    }  
  
    Nodo* _primero;  
}
```

```
int main() {  
    Lista<int> l1;  
    l1.agregarAtras(1);  
    Lista<int> l2;  
    l2.agregarAtras(10);  
    l2 = l1;  
    l2.agregarAtras(2);  
    l1.longitud(); // ??  
}
```

¿Qué pasaría si Lista no tuviera operador de asignación?

```
class Lista {
public:
    Lista();
    Lista(const Lista& otra);
    Lista& operator=(const Lista& otra);
    void agregarAtras(T&);
    int longitud() const;
    ...

private:
    struct Nodo {
        T valor;
        Nodo* siguiente;
    }

    Nodo* _primero;
}
```

```
int main() {
    Lista<int> l1;
    l1.agregarAtras(1);
    Lista<int> l2;
    l2.agregarAtras(10);
    l2 = l1;
    l2.agregarAtras(2);
    l1.longitud(); // ??
}
```

¡Pizarrón!

Constructor por copia

¿Cómo evitamos usar la asignación en este caso?

```
class MaximoRapido {  
    public:  
        MaximoRapido(const Lista<int>& l);  
        int maximo() const;  
  
    private:  
        Lista<int> _lista;  
        Lista<int>::iterator _max;  
}  
  
MaximoRapido::MaximoRapido(const Lista<int>& l) {  
    _lista = l;  
    // buscar el máximo  
}
```

```
MaximoRapido::MaximoRapido(const Lista<int>& l) : _lista(l), _max(){  
    // buscar el máximo  
}
```

```
template<class T>  
Lista::Lista(const Lista& otra) {  
    _primero = null;  
    for (int i = 0; i < otra.longitud(); i++) {  
        this->agregarAtras(otra[i]);  
    }  
}
```

Lista de inicialización

```
class Fecha {
public:
    // pre: anio > 0, mes \in [1, 12],
    // dia \in [1, diasEnMes(anio, mes)]
    Fecha(Anio anio, Mes mes, Dia dia);
    Fecha(Fecha fecha, Periodo periodo);

    Anio anio() const;
    Mes mes() const;
    Dia dia() const;

    bool operator==(Fecha o) const;
    bool operator<(Fecha o) const;

    void sumar_periodo(Periodo p);

private:
    Anio _anio;
    Mes _mes;
    Dia _dia;

    void ajustar_fecha();
    void sumar_anios(Anio anios);
    void sumar_meses(Mes meses);
    void sumar_dias(Dia dias);
};
```

```
class Intervalo {
public:
    // pre: desde < hasta
    Intervalo(Fecha desde, Fecha hasta);
    Intervalo(Fecha desde, Periodo periodo);

    Fecha desde() const;
    Fecha hasta() const;

    int enDias() const;

private:
    Fecha _desde;
    Fecha _hasta;
};
```

```
class Periodo {
public:
    Periodo(int anios, int meses, int dias);

    int anios() const;
    int meses() const;
    int dias() const;

private:
    int _anios;
    int _meses;
    int _dias;
};
```

```
Intervalo::Intervalo(Fecha desde, Fecha hasta) {
    _desde = desde;
    _hasta = hasta;
}
```

Lista de inicialización

```
class Fecha {
public:
    // pre: anio > 0, mes \in [1, 12],
    // dia \in [1, diasEnMes(anio, mes)]
    Fecha(Anio anio, Mes mes, Dia dia);
    Fecha(Fecha fecha, Periodo periodo);

    Anio anio() const;
    Mes mes() const;
    Dia dia() const;

    bool operator==(Fecha o) const;
    bool operator<(Fecha o) const;

    void sumar_periodo(Periodo p);

private:
    Anio _anio;
    Mes _mes;
    Dia _dia;

    void ajustar_fecha();
    void sumar_anios(Anio anios);
    void sumar_meses(Mes meses);
    void sumar_dias(Dia dias);
};
```

```
class Intervalo {
public:
    // pre: desde < hasta
    Intervalo(Fecha desde, Fecha hasta);
    Intervalo(Fecha desde, Periodo periodo);

    Fecha desde() const;
    Fecha hasta() const;

    int enDias() const;

private:
    Fecha _desde;
    Fecha _hasta;
};
```

```
class Periodo {
public:
    Periodo(int anios, int meses, int dias);

    int anios() const;
    int meses() const;
    int dias() const;

private:
    int _anios;
    int _meses;
    int _dias;
};
```

```
Intervalo::Intervalo(Fecha desde, Fecha hasta) {
    _desde = desde;
    _hasta = hasta;
}
```

```
const_Intervalo.cpp: In constructor 'Intervalo::Intervalo(Fecha, Fecha)':
const_Intervalo.cpp:59:46: error: no matching function for call to 'Fecha::Fecha()'
    Intervalo::Intervalo(Fecha desde, Fecha hasta) {
                                   ^
```


Lista de inicialización

```
class Fecha {
public:
    // pre: anio > 0, mes \in [1, 12],
    // dia \in [1, diasEnMes(anio, mes)]
    Fecha(Anio anio, Mes mes, Dia dia);
    Fecha(Fecha fecha, Periodo periodo);

    Anio anio() const;
    Mes mes() const;
    Dia dia() const;

    bool operator==(Fecha o) const;
    bool operator<(Fecha o) const;

    void sumar_periodo(Periodo p);

private:
    Anio _anio;
    Mes _mes;
    Dia _dia;

    void ajustar_fecha();
    void sumar_anios(Anio anios);
    void sumar_meses(Mes meses);
    void sumar_dias(Dia dias);
};
```

```
class Intervalo {
public:
    // pre: desde < hasta
    Intervalo(Fecha desde, Fecha hasta);
    Intervalo(Fecha desde, Periodo periodo);

    Fecha desde() const;
    Fecha hasta() const;

    int enDias() const;

private:
    Fecha _desde;
    Fecha _hasta;
};

class Periodo {
public:
    Periodo(int anios, int meses, int dias);

    int anios() const;
    int meses() const;
    int dias() const;

private:
    int _anios;
    int _meses;
    int _dias;
};

Intervalo::Intervalo(Fecha desde, Fecha hasta)
    : _desde(desde), _hasta(hasta) {};
```

Lista de inicialización

```
class Fecha {
public:
    // pre: anio > 0, mes \in [1, 12],
    // dia \in [1, diasEnMes(anio, mes)]
    Fecha(Anio anio, Mes mes, Dia dia);
    Fecha(Fecha fecha, Periodo periodo);

    Anio anio() const;
    Mes mes() const;
    Dia dia() const;

    bool operator==(Fecha o) const;
    bool operator<(Fecha o) const;

    void sumar_periodo(Periodo p);

private:
    Anio _anio;
    Mes _mes;
    Dia _dia;

    void ajustar_fecha();
    void sumar_anios(Anio anios);
    void sumar_meses(Mes meses);
    void sumar_dias(Dia dias);
};
```

```
class Intervalo {
public:
    // pre: desde < hasta
    Intervalo(Fecha desde, Fecha hasta);
    Intervalo(Fecha desde, Periodo periodo);

    Fecha desde() const;
    Fecha hasta() const;

    int enDias() const;

private:
    Fecha _desde;
    Fecha _hasta;
};

class Periodo {
public:
    Periodo(int anios, int meses, int dias);

    int anios() const;
    int meses() const;
    int dias() const;

private:
    int _anios;
    int _meses;
    int _dias;
};

Intervalo::Intervalo(Fecha desde, Periodo periodo)
    : _desde(desde), _hasta(desde) {
    _hasta.sumar_periodo(periodo);
};
```

Lista de inicialización

```
Fecha::Fecha(Fecha fecha, Periodo periodo) :  
    _anio(fecha.anio()), _mes(fecha.mes()), _dia(fecha.dia()) {  
  
    sumar_periodo(periodo);  
}
```

```
Intervalo::Intervalo(Fecha desde, Periodo periodo)  
    : _desde(desde), _hasta(desde) {  
  
    _hasta.sumar_periodo(periodo);  
};
```

Lista de inicialización

A partir de C++11

```
Fecha::Fecha(Fecha fecha, Periodo periodo)
    : Fecha(fecha) {
    this->sumar_periodo(periodo);
}
```

```
Lista::Lista(const Lista& otra) : Lista() {
    for (int i = 0; i < otra.longitud(); i++) {
        agregarAtras(otra[i]);
    }
}
```

Iteradores y Algoritmos Genéricos en C++

Algoritmos y Estructuras de Datos II

Colecciones

Conocemos los siguientes tipos de *colecciones*:

- ▶ Arreglo.
- ▶ Secuencia.
- ▶ Conjunto.
- ▶ Multiconjunto.
- ▶ Diccionario.

Operaciones sobre colecciones

Preguntas típicas sobre colecciones:

- ▶ Dado un elemento, ¿está en la colección?

$x \in \text{conj}$

$\text{def?}(x, \text{dicc})$

- ▶ Listar todos los elementos de una colección.
- ▶ Encontrar el elemento más chico de la colección.
- ▶ *etc.*

Operaciones sobre colecciones

¿Cómo recorreremos una colección?

- ▶ **Arreglo.** Tamaño y acceder al i -ésimo (`operator[]`).
- ▶ **Secuencia.** `prim` y `fin`.
- ▶ **Conjunto.** `dameUno` y `sinUno`.
- ▶ **Multiconjunto.** `dameUno` y `sinUno`.
- ▶ **Diccionario.** `claves` y `obtener`.

¿Hay una manera uniforme de recorrerlas?

Operaciones sobre colecciones

¿Cómo hacemos para mirar los primeros 5 elementos de una lista simplemente enlazada en C++?

- ▶ Supongamos que la operación fin es destructiva:

```
void Lista<T>::sacarPrimero() { ... }
```

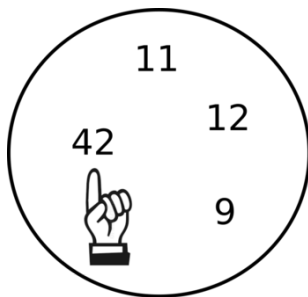
- ▶ ¿Qué pasa con la complejidad?

Iteradores

Un **iterador** es una manera abstracta de recorrer colecciones, independientemente de su estructura.

Informalmente

iterador = colección + dedo



Iteradores

Operaciones con iteradores:

- ▶ ¿Está posicionado sobre un elemento?
- ▶ Obtener el elemento actual.
- ▶ Avanzar al siguiente elemento.
- ▶ Retroceder al elemento anterior.
- ▶ Modificar el valor del elemento actual.

(Bidireccional)

(Mutable)

Iteradores en C++

Tipos

Si T es un tipo de colección:

- ▶ `T::iterator`, `T::const_iterator`
Tipo de los iteradores mutables e inmutables.
Por ejemplo `vector<int>::iterator` es un tipo.
- ▶ `T::value_type`
Tipo de los elementos que almacena la colección.
Por ejemplo `vector<int>::value_type` es `int`.

Creación de iteradores

Si `col` es una colección de tipo T:

- ▶ `col.begin()`
Iterador posicionado sobre el primer elemento de la colección.
- ▶ `col.end()`
Iterador posicionado sobre el final de la colección.
(Después del último elemento).

Iteradores en C++

Operaciones con iteradores

Si `it` es de tipo `T::iterator` o `T::const_iterator`:

- ▶ `*it` Obtiene el elemento actual.
Si `it` es un `T::iterator`, es un lvalue.
Si `it` es un `T::const_iterator`, no es un lvalue.
- ▶ `it->campo` Equivalente a `(*it).campo`.
- ▶ `++it` Avanza al siguiente elemento.
- ▶ `--it` Retrocede al elemento anterior.
- ▶ ...

Iteradores en C++

Operaciones de la colección usando iteradores

- ▶ `T::iterator T::insert(T::iterator pos, const T::value_type& elem)`
Inserta un elemento en la posición indicada.
- ▶ `T::iterator T::erase(T::iterator pos)`
Elimina el elemento en la posición indicada.
- ▶ ...

Iteradores en C++

Ejemplo (recorrer)

```
vector<int> v = {1, 2, 3, 4};
```

```
vector<int>::iterator it = v.begin();
```

```
while (it != v.end()) {
```

```
    cout << *it;
```

```
    ++it;
```

```
}
```

```
for(vector<int>::iterator it = v.begin(); it != v.end(); ++it){
```

```
    cout << *it;
```

```
}
```

Iteradores en C++

Ejemplo (eliminar)

```
vector<int> v = {1, 2, 3, 4};  
vector<int>::iterator it = v.begin();  
it += 2;  
v.erase(it);
```

// 1 2 4

Iteradores en C++

Ejemplo (insertar)

```
vector<int> v = {1, 2, 3, 4};  
vector<int>::iterator it = v.end();  
--it;  
v.insert(it, 10);
```

// 1 2 3 10 4

Iteradores en C++

Muchas veces el compilador puede inferir los tipos:

Ejemplo (auto)

```
vector<int> v = {1, 2, 3, 4};  
auto it = v.end();  
--it;  
v.insert(it, 10);
```

Iteradores en C++

Ejemplo (iteradores mutables vs. inmutables)

```
void mostrar(const vector<int>& v) {  
    for (vector<int>::iterator it = v.begin();  
         it != v.end(); ++it) {  
        cout << *it;  
    }  
}
```

```
int main() {  
    vector<int> v{1, 2, 3, 4};  
    mostrar(v);  
    return 0;  
}
```

Iteradores en C++

Ejemplo (iteradores mutables vs. inmutables)

```
void mostrar(const vector<int>& v) {  
    for (vector<int>::iterator it = v.begin();  
         it != v.end(); ++it) {  
        cout << *it;  
    }  
}
```

```
int main() {  
    vector<int> v{1, 2, 3, 4};  
    mostrar(v);  
    return 0;  
}
```

In function `void mostrar(const std::vector<int>&):`
conversion from `std::vector<int>::const_iterator [...]`
to non-scalar type `std::vector<int>::iterator [...]`

Iteradores en C++

Ejemplo (iteradores mutables vs. inmutables)

```
void mostrar(const vector<int>& v) {  
    for (vector<int>::const_iterator it = v.begin();  
         it != v.end(); ++it) {  
        cout << *it;  
    }  
}  
  
int main() {  
    vector<int> v{1, 2, 3, 4};  
    mostrar(v);  
    return 0;  
}
```

Iteradores en C++

Ejemplo (for basado en rangos)

```
void mostrar(const vector<int>& v) {  
    for (int x : v) {  
        cout << x;  
    }  
}
```

```
int main() {  
    vector<int> v{1, 2, 3, 4};  
    mostrar(v);  
    return 0;  
}
```

Iteradores en C++

Ejemplo (for basado en rangos)

```
void mostrar(const vector<int>& v) {  
    for (int x : v) {  
        cout << x;  
    }  
}
```

```
int main() {  
    vector<int> v{1, 2, 3, 4};  
    mostrar(v);  
    return 0;  
}
```

- En general se acepta la sintaxis `for (T x : col)` siempre que `col` sea una colección con la interfaz de iteradores descrita arriba.

Algoritmos genéricos

Recibiendo una colección genérica:

```
template<class Coleccion>
bool pertenece(const Coleccion& c,
               typename const Coleccion::value_type& x) {
    for (auto& y : c) {
        if (x == y) {
            return true;
        }
    }
    return false;
}

int main() {
    vector<int> v{1, 2, 3, 4};
    int dos = 2;
    cout << pertenece(v, dos);
}
```

(Ojo con el `typename`).

Algoritmos genéricos

Recibiendo un iterador genérico:

```
template<class Iterador>
bool pertenece(Iterador desde, Iterador hasta,
               typename Iterador::value_type& x) {
    for (auto it = desde; it != hasta; ++it) {
        if (x == *it) {
            return true;
        }
    }
    return false;
}

int main() {
    vector<int> v{1, 2, 3, 4};
    int dos = 2;
    cout << pertenece(v.begin(), v.end(), dos);
}
```