

Abstracción por parametrización(1/3).

Funciones Patrón	Clases Patrón
<p>Funciones idénticas excepto en el tipo de dato</p> <pre data-bbox="476 134 1253 1118"> void intercambiar (int& a, int& b) { int aux= a; a= b; b= aux; } void intercambiar (float& a, float& b) { float aux= a; a= b; b= aux; } void intercambiar (string& a, string& b) { string aux= a; a= b; b= aux; }</pre> <p>Parametrizamos el tipo de dato mediante una Plantilla</p> <pre data-bbox="476 112 1253 1118"> template <class T> void intercambiar (T& a, T& b) { T aux= a; a= b; b= aux; }</pre> <p>Uso de la plantilla desde otra función genérica</p> <pre data-bbox="476 112 1253 1118"> template <class T> void ordenar_selección (T *vector, int n) { int i,minimo; for (i=0;i<n-1;i++) { minimo= i; for (j=i+1;j<n;j++) if (vector[j]<vector[minimo]) minimo= j; intercambiar(vector[i],vector[minimo]); } }</pre>	<p>Parametrización del tipo base de una clase</p> <pre data-bbox="476 112 1253 1118"> template <class T> class Vector_Dinamico { private: T * datos; int nelementos; public: Vector_Dinamico<T>(int n); Vector_Dinamico<T>(const Vector_Dinamico<T>& original); ~Vector_Dinamico<T>(); int size() const; T& operator[] (int i); const T& operator[] (int i) const; void resize(int n); Vector_Dinamico<T>& operator= (const Vector_Dinamico<T>& original); }; Instanciación de un tipo concreto con la declaración</pre> <pre data-bbox="476 112 1253 1118"> Vector_Dinamico<int> valores; Vector_Dinamico<string> nombres; Vector_Dinamico<Polinomio> polinomios;</pre>

Abstracción por parametrización(2/3).

Definición de los métodos

```
template<class T>
Vector_Dinamico<T>::Vector_Dinamico<T>(int n)
{
    assert(n>=0);
    if (n>0)
        datos= new T[n];
    nelementos= n;
}

template<class T>
Vector_Dinamico<T>&
Vector_Dinamico<T>::operator=
(const Vector_Dinamico<T>& original)
{
    if (this!= &original) {
        if (nelementos>0) delete[] datos;
        nelementos= original.nelementos;
        datos= new T[nelementos];
        for (int i=0; i<nelementos; ++i)
            datos[i]= original.datos[i];
    }
    return *this;
}
```

Ejemplo de uso

```
template <class T>
void ordenar_selección (Vector_Dinamico<T>& vector)
{
    int i,minimo;
```

```
for (i=0;i<vector.size()-1;i++) {
    minimo= i;
    for (j=i+1;j<vector.size();j++)
        if (vector[j]<vector[minimo])
            minimo= j;
    intercambiar(vector[i],vector[minimo]);
}
```

Abstracción por parametrización(3/3).

Clase Par	Ejemplo de uso
<pre>#ifndef _utilidades_h #define _utilidades_h template <class T1, class T2> struct Par { T1 primero; T2 segundo; /* —— Operaciones —— */ Par(): primero(T1()), segundo(T2()) {} Par(const T1& p, const T2& s): primero(p),segundo(s) {} Par(const Par& p): primero(p.primer),segundo(p.segundo) {} template <class U1, class U2> Par(const Par<U1,U2>& p): primero(p.primer),segundo(p.segundo) {} ~Par() {} Par& operator= (const Par& v) {primero=v.primero;segundo=v.segundo; return *this; } bool operator==(const Par& s) const {return primero==s.primero && segundo==s.segundo;} bool operator!= (const Par& s) const {return primero!=s.primero segundo!=s.segundo;} }; #endif</pre>	<pre>#include <iostream> #include <par.h> using namespace std; template<class T> void intercambiar (T& a, T& b) { T aux=a; a=b; b=aux; } template <class T> void ordenar(Par<int,T>& a, Par<int,T>& b) { if (a.primero>b.primero) intercambiar(a,b); } int main() { Par<int,float> v1,v2; v1.primero=1; v1.segundo=2.0; v2.primero=0; v2.segundo=5.0; ordenar(v1,v2); cout << "El primero es " <<v1.primero<<","<<v1.segundo<<endl; return 0; }</pre>

Pilas (vectores).

<i>Pila.h</i>	<i>Pila.h</i>
<pre> template <class T> class Pila{ private: Vector<T> v; int num_elem; public: Pila(): v(1), num_elem(0) {} Pila(const Pila<T> & p): v(p.v), num_elem(p.num_elem) ~Pila() {} Pila& operator= (const Pila<T>& p) { v=p.v; num_elem= num_elem; } bool Vacia() const {return num_elem==0;} T& Tope () { assert(num_elem!=0); return v[num_elem-1]; } };</pre>	<pre> const T & Tope () const { assert(num_elem!=0); return v[num_elem-1]; } void Poner(const T & elem) { if (num_elem==v.size()) v.Resize(2*num_elem); v[num_elem]= elem; num_elem++; } void Quitar() { assert(num_elem!=0); num_elem--; if (num_elem<v.size()/4) v.Resize(v.size()/2); } int Num_elementos() const { return num_elem; }</pre>

Pilas (celdas enlazadas).

Pila.h

```
template <class T>
class Pila{
private:
    struct Celda {
        T elemento;
        Celda *siguiente;
    };
    Celda() : siguiente(0) {}
    Celda(const T & elem, Celda * sig)
        : elemento(elem), siguiente(sig)
    {};
};

Celda * primera;
int num_elem;

public:
    Pila(): primera(0), num_elem(0) {}
    Pila(const Pila<T> & p);
    ~Pila();
    Pilas& operator=(const Pilas<T>& p);
    bool Vacia() const {return primera==0;}
    T & Tope()
    {
        assert(primer!=0); return primera->elemento;
    }
    const T & Tope() const
    {
        assert(primer!=0); return primera->elemento;
    }
    void Poner(const T & elem);
    void Quitar();
    int Num_elementos() const { return num_elem; }
};
```

Pila.cpp

```
template <class T>
Pila<T>::~Pila()
{
    Celda *aux;
    while (primera!=0) {
        aux= primera;
        primera= primera->siguiente;
        delete aux;
    }
}

template <class T>
void Pila<T>::Poner(const T & elem)
{
    primera= new Celda(elem, primera);
    num_elem++;
}

template <class T>
void Pila<T>::Quitar()
{
    assert(primer!=0);
    Celda *aux= primera;
    primera= primera->siguiente;
    delete aux;
    num_elem--;
}
```

Colas (celdas enlazadas).

Cola.h	Cola.cpp
<pre> template <class T> class Cola{ private: struct Celda { T elemento; Celda * siguiente; }; Celda() : siguiente(0) {} Celda(const T & elem, Celda * sig) : elemento(elem), siguiente(sig) {} }; Celda * primera; Celda * ultima; int num_elem; public: Cola(): primera(0),ultima(0),num_elem(0) {} Cola(const Cola<T> & p); ~Cola(); Cola& operator=(const Cola<T>& p); bool Vacia() const {return num_elem==0;} T & Frente () { assert(primer!=0); return primera->elemento; } const T & Frente () const { assert(primer!=0); return primera->elemento; } void Poner(const T & elem); void Quitar(); int Num_elementos() const { return num_elem; } } </pre>	<pre> template <class T> Cola<T>::~Cola() { Celda *aux; while (primera!=0) { aux= primera; primera= primera->siguiente; delete aux; } } template <class T> void Cola<T>::Poner(const T & elem) { Celda *aux=new Celda(elem,0); if (primera==0) primera=ultima= aux; else { ultima->siguiente=aux; ultima= aux; } num_elem++; } template <class T> void Cola<T>::Quitar() { assert(primer!=0); Celda *aux=primer; primera= primera->siguiente; delete aux; if (primer==0) ultima=0; num_elem--; } </pre>

Colas (vectores).

<i>Cola.h</i>	<i>Cola.h</i>
<pre> template <class T> class Cola{ private: Vector<T> v; int num_elem; int anterior,posterior; void Expandir(int nelem) { assert(nelem>v.size()); Vector<T> aux(nelem); for (int i=0;i<num_elem;i++) aux[i]=v[(anterior+i)%v.size()]; anterior=0; posterior=(anterior+num_elem); v=aux; } void Contraer(int nelem) { assert(nelem<v.size()); Vector<T> aux(nelem); for (int i=0;i<num_elem;i++) aux[i]=v[(anterior+i)%v.size()]; anterior=0; posterior=(anterior+num_elem); v=aux; } public: Cola(): v(1),num_elem(0),anterior(0),posterior(0) {} Cola(const Cola<T> & p):v(p.v),num_elem(p.num_elem), anterior(p.anterior),posterior(p.posterior) {} ~Cola() {} }</pre>	<pre> Cola& operator= (const Cola<T>& p) { v=p.v; num_elem=p.num_elem; anterior=p.anterior; posterior=p.posterior; } bool Vacia() const { return num_elem==0; } T& Frente () { assert(num_elem!=0); return v[anterior]; } const T & Frente () const { assert(num_elem!=0); return v[anterior];} void Poner(const T & elem) { if (num_elem==v.size()) Expandir(2*v.size()); v[posterior]=elem; posterior=(posterior+1)%v.size(); num_elem++; } void Quitar() { assert(num_elem!=0); anterior=(anterior+1)%v.size(); num_elem--; if (num_elem<v.size()/4) Contraer(v.size()/2); } int Num_elementos() const { return num_elem; } }</pre>

* GENERALIZACION: ABSTRACCION POR ITERACION

Contenedor

Tipo de dato que está compuesto por una colección de elementos de algún otro tipo

Problema: Acceder a cada uno de los elementos que lo componen

¿Podemos hacer abstracción e intentar manejar distintos tipos de datos de la misma forma?

Iterador

El problema de recorrer los elementos de un contenedor, es decir, de iterar sobre los elementos del mismo, se resolverá mediante la definición de un nuevo tipo de dato abstracto, un iterador, que abstraerá la idea de indexar los elementos con un mecanismo similar al de los punteros

¿Por qué similar al de los punteros?

* $v[i]$ → Para acceder a un elemento se necesita el vector y el índice a partir de los cuales se calcula la posición del elemento y se accede a él

* $(v + i)$ → Solo con el puntero se accede al elemento indicado (sin usar el vector) y dado el puntero solo se necesita la referenciación, para acceder al elemento.

Vectores C++ e iteración

Uso de los punteros en C++ para iterar:

double * v; // vector de doubles

```
double * p; // el iterador
```

```
double * final; // para no recalcular p + n
```

```
final = v + n; // elementos después del último
```

```
for (p = v; p != final; ++p)
```

```
{cout << *p << endl;
```

Un iterador sobre un vector de reales es del tipo puntero a double, por lo que si queremos definir un nuevo tipo de dato encargado de recoger un contenedor podríamos definir:

```
typedef double * iterator;
```

- Para recoger el vector v se comienza por el primer elemento (puntero al primer elemento)
 - La iteración por todos los elementos termina al llegar a la posición final (elemento detrás del último)
- ¿Cómo abstraer esta idea?

```
typedef double * iterator;
```

```
iterator begin (double *v, int n)
{
    return v;
}
```

```
iterator end (double *v, int n)
{
    return v+n;
}
```

.....

```
iterator p;
```

```
for (p = begin(v, n); p != end(v, n); ++p)
    cout << *p << endl;
```

Si se puede haber problemas al acceder a elementos fijos.

```
void anular_elementos (double *v, int n)
```

```
{
    iterator p;
    for (p = begin(v, n); p != end(v, n); ++p)
        *p = 0.0;
}
```

```
void escribir_elementos (const double *v, int n)
```

```
{
    iterator p;
    for (p = begin(v, n); p != end(v, n); ++p)
        cout << *p << endl;
}
```

La ~~función~~ función daria un error al intentar convertir desde const double * a double *

Problema: el tipo que debemos usar para recorrer un contenedor no modificable no es el mismo que si se puede modificar

void función_correcta (double *f, int n)

```
    { double *p = f;  
    -- --  
    }
```

void función_incorrecta (const double *f, int n)

```
    { double *p = f;  
    -- --  
    }
```

Solución: definir 2 tipos diferentes de iteradores: uno para contenedores que se van a modificar (iterator) y otro para contenedores que no se van a modificar (const_iterator)

```
typedef double * iterator;
```

```
typedef const double * const_iterator;
```

```
iterator begin (double *v, int n) { return v; }
```

```
iterator end (double *v, int n) { return v+n; }
```

```
const_iterator begin (const double *v, int n) { return v; }
```

```
const_iterator end (const double *v, int n) { return v+n; }
```

```
void anular_elementos ( double *v, int n)
```

```
{ iterator p;  
for (p = begin(v, n); p != end(v, n); ++p)  
    *p = 0.0;  
}
```

```
void escribir_elementos ( const double *v, int n)
```

```
{ const_iterator p;  
for (p = begin(v, n); p != end(v, n); ++p)  
    cout << *p << endl;
```

```
}
```

* Un iterator se puede asignar a un const_iterator pero no al revés (un puntero a algo que se puede modificar (de lectura / escritura) puede asignarse a un puntero de solo lectura (más restrictivo), pero no al contrario.

Nuestro interés es crear un tipo de dato (que se comporta como un puntero en los vectores) que nos permita iterar sobre los elementos de quiero contenido → ITERADORES Y PROGRAMACION GENERICA

```
for (P = c.begin(); p != c.end(); ++P)  
    cout << *P << endl;
```

El iterador P puede recorrer cualquier contenedor (vector, vector dinámico, conjunto etc.) que se ajuste a esa especificación y que se pueda recorrer de la misma forma.

Proceso:

1. Necesidad de un mecanismo de iteración para algunos contenedores
2. Generalizar el problema buscando una solución válida para todos
3. Estudiar la forma en que pueden recoverse elementos usando aritmética de punteros
4. Desarrollar los tipos iterator y const_iterator
5. Unquier algoritmo que se pueda diseñar e implementar utilizando esta abstracción es válido para todos los contenedores incluyendo los que en el futuro dispongan de esa misma abstracción.



ALGORITMOS GENERICOS

TDA EN C++ E ITERADORES

Problema: Diseñar todos los tipos de datos abstractos contenedores que queremos que dispongan de un sistema común de iteración con una interfaz similar al que hemos estudiado.

Ejemplo:

Para los tipos contenedor vector dinámico, vector disperso, conjunto etc. se definen un tipo iterador y otro const_iterator dotándolos de las operaciones básicas para los iteradores (incluyendo `++`, `=`, `*`, `=` etc.). Como esos 2 tipos de datos son particulares para cada uno de los tipos contenedor que tenemos, optaremos por definirlos dentro de la clase contenedor

↓ Algoritmo genérico

```
template <class T>
void escribir_elementos (const T &c) {
    typename T::const_iterator p;
    for (p = c.begin(); p != c.end(); ++p)
        cout << *p << endl;
```

y

Abstracción por iteración(2/3).

Clase Vector	Clase Vector con iteradores
<pre>template <class T> class Vector { private: T * datos; int nelementos; public: // _____ Constructores _____ Vector<T>(int n=0); Vector<T>(const Vector<T>& original); // _____ Destructor _____ ~Vector<T>(); // _____ Otras funciones _____ int size() const; T& operator[](int i); const T& operator[](int i) const; void resize(int n); Vector<T>& operator=(const Vector<T>& original); };</pre>	<pre>template <class T> class Vector { private: T * datos; int nelementos; public: ... class iterador{ ... }; class const_iterador{ ... }; iterador begin() { ... }; iterador end() { ... }; const_iterador begin() const { ... }; const_iterador end() const { ... }; }; void anula_vector(Vector<float>& v) { Vector<float>::iterador p; for (p=v.begin();p!=v.end();++p) *p= 0.0; } void escribe(const Vector<int>& v) { Vector<int>::const_iterador pi; for (pi=v.begin();pi!=v.end();++pi) cout << *pi << endl; }</pre>

Abstracción por iteración(3/3).

iterador	const_iterador
<pre> iterador begin() { iterador i; i.puntero= datos; return i;} iterador end() { iterador i; i.puntero= datos+nElementos; return i;} class iterador { private: T* puntero; public: iterador(): puntero(0) {} iterador(const iterador& v): puntero(v.puntero) {} ~iterador() {} iterador& operator=(const iterador& orig) { puntero=orig.puntero; return *this; } T& operator*() const { assert(puntero!=0);return *puntero; } iterador& operator++() { assert(puntero!=0);puntero++;return *this; } iterador& operator--() { assert(puntero!=0);puntero--;return *this; } bool operator!=(const iterador& v) const { return puntero!=v.puntero; } bool operator==(const iterador& v) const { return puntero==v.puntero; } }; </pre>	<pre> const_iterador begin() const {return const_iterador(datos);} const_iterador end() const { {return const_iterador(datos+nElementos);} } class const_iterador { private: T* puntero; const_iterador(T* p): puntero(p) {} public: const_iterador(): puntero(0) {} const_iterador(const const_iterador& v): puntero(v.puntero) {} ~const_iterador() {} const_iterador& operator=(const const_iterador& orig) { puntero=orig.puntero; return *this; } const_iterador(const iterador& v): puntero(v.puntero) {} const T& operator*() const { assert(puntero!=0);return *puntero; } const_iterador& operator++() { assert(puntero!=0);puntero++;return*this; } const_iterador& operator--() { assert(puntero!=0);puntero--;return *this; } bool operator!=(const const_iterador& v) const { return puntero!=v.puntero; } bool operator==(const const_iterador& v) const { return puntero==v.puntero; } }; friend class Vector<T>; </pre>

```
iterator begin() { iterator i; i.puntero = datos; return i; }
```

```
iterator end() { iterator i; i.puntero = datos + nElementos; return i; }
```

```
const iterator begin() const { return const_iterator(datos); }
```

```
const iterator end() const { return const_iterator(datos + nElementos); }
```

```
}
```

#include <vector.cpp>

#endif

Programa de prueba : ejemplo_vector.cpp

include <iostream>

include <vector.h>

include <cassert>

include <string>

using namespace std;

void cargar_indices (vector<int> & v)

{

for (int i=0; i< v.size(); ++i)

v[i]=i;

}

template <class T>

T maximo (const Vector<T> & v)

{ assert (v.begin() != v.end());

hypote Vector<T>::const_iterator max = v.begin();

for (Vector<T>::const_iterator p = v.begin(); p != v.end(); ++p)

if (*max < *p)

max = p;

return *max;

}

int main()

{ Vector<int> vec(3);

Vector<string> cadenas(4);

Largar_indices(vec);

cout << "Maximo de " << vec.size() << " elementos enteros: "

<< maximo(vec) << endl;

vec.resize(10);

Largar_indices(vec);

cout << "Maximo de " << vec.size() << " elementos enteros: "

<< maximo(vec) << endl;

cadenas[0] = "Esto";

cadenas[1] = "es";

cadenas[2] = "una";

cadenas[3] = "prueba";

cout << "Maximo de " << cadenas.size() << " elementos cadena: "

<< maximo(cadenas) << endl;

return 0;

Listas (celdas enlazadas). 1/2

listas.h

```
template <class T>
class Lista{
private:
    struct Celda {
        T elemento;
        Celda * siguiente;
    } Celda() : siguiente(0)
    Celda(const T & elem, Celda * sig)
        : elemento(elem), siguiente(sig)
    {};
};

Celda * cab;
Celda * ultima;
int num_elem;

public:
    class iterador;
    class const_iterador;
    // Para conocer que existen
    Lista();
    Lista(const Lista & l);
    Lista& operator=(const Lista & l);
    bool Vacia() const { return num_elem==0; }
    int Num_elementos() const { return num_elem; }
    iterador Insertar(iterador p, const T & elemento);
    iterador Borrar(iterador p);
```

listas.h

```
iterador Begin()
{
    return iterador(cab,cab);
}
iterador End()
{
    return iterador(cab,ultima);
}
const_iterador Begin() const
{
    return const_iterador(cab,cab);
}
const_iterador End() const
{
    return const_iterador(cab,ultima);
}
~Lista();
```

```
class iterador {
```

```
    Celda * base;
    Celda * punt;
    ...
};
```

```
class const_iterador {
```

```
    Celda * base;
    Celda * punt;
    ...
};
```

```
#include <lista_cel.cpp>
```

Listas (celdas enlazadas). 2/2

listas.h

```

class iterador {
    Celda * base;
    Celda * punt;
    iterador(Celda *primera,Celda * p): base(primera),punt(p)
    {}
public:
    iterador(): punt(0),base(0)
    {}
    iterador(const iterador & i): punt(i.punt),base(i.base)
    {}
    T & operator*() const
    {
        return punt->siguiente->elemento;
    }
    iterador & operator++()
    {
        punt = punt->siguiente; return *this;
    }
    iterador & operator--()
    {
        Celda* aux=base;
        while(aux->sig!=punt) aux=aux->sig;
        punt= aux;
        return *this;
    }
    bool operator==(const iterador & i)
    {
        return punt == i.punt && base==i.base ;
    }
    bool operator!=(const iterador & i)
    {
        return punt != i.punt || base!=i.base ;
    }
friend class const_iterador;
friend class Lista;
};


```

listas.h

```

class const_iterador {
private:
    Celda * base;
    Celda * punt;
    const_iterador(Celda *primera,Celda * p):
        base(primera),punt(p) {}
public:
    const_iterador(): punt(0),base(0) {}
    const_iterador(const const_iterador & i):
        punt(i.punt),base(i.base) {}
    const_iterador(const iterador & i):
        punt(i.punt),base(i.base) {}
    const T & operator*() const
    {
        return punt->siguiente->elemento;
    }
    const_iterador & operator++()
    {
        punt = punt->siguiente; return *this;
    }
    const_iterador & operator--()
    {
        Celda* aux=base;
        while(aux->sig!=punt) aux=aux->sig;
        punt= aux;
        return *this;
    }
    bool operator==(const const_iterador & i)
    {
        return punt == i.punt && base==i.base ;
    }
    bool operator!=(const const_iterador & i)
    {
        return punt != i.punt || base!=i.base ;
    }
friend class Lista;
};


```