* 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 abstrae la idea de indexar los elementos con un mecanismo similar al de los punteros

¿Por que 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 recorrer un contenedor podríamos definir:

```
typedef double * iterator;
```

- Para recorrer 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)

¿Como 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;
```

pero puede haber problemas en el acceso a elementos ctes.

```
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 [redacted] función daría 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 funcion_correcta ( double *f, int n)
    { double *p = f;
      ....
    }
void funcion_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 reves (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 interes es crear un tipo de dato (que se comporte como un puntero en los vectores) que nos permita iterar sobre los elementos de cualquier contenedor → 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 todas
3. Estudiar la forma en que pueden recorrerse elementos usando aritmética de punteros
4. Desarrollar los tipos iterator y const_iterator
5. Cualquier 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 contenedor que queremos que dispongan de un sistema común de iteración con un interfaz similar al que hemos estudiado.

Ejemplo:

Para los tipos contenedor vector dinámico, vector disperso, conjunto etc. se definen un tipo iterator 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)        // c: contenedor
   {
      typename T::const_iterator p;
      for (p=c.begin(); p!=c.end(); ++p)
            cout << *p << endl;
   }
```

Abstracción por iteración(2/3).

Clase Vector

```
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);
};
```

Clase Vector con iteradores

```
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 p;

  for (p=v.begin();p! =v.end();++p)
    cout ≪ *p ≪ endl;
}
```

Abstracción por iteración(3/3).

iterador

```
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;}
};
```

const_iterador

```
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>;
};
```

```
    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());
    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);
    cargar_indices (vec);
    cout << "Maximo de" << vec.size() << "elementos enteros:"
        << maximo (vec) << endl;

    vec.resize (10);
    cargar_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;
}
```