Tema 4: Contenedores, Iteradores y Algoritmos en C++

Biblioteca estándar de plantillas (STL: Standard Template Library)

C++, en su biblioteca estándar STL, nos provee de un conjunco de clases y utilidades, que resuelve muchos problemas. Entre ellos se encuentra los **Contenedor**, **Iteradores** y **Algoritmos**.

- Los contenedores son objetos capaces de almacenar otros objetos, cada uno de una forma particular.
- Casi todos los contenedores proveen iteradores (herramientas para poder acceder a sus elementos).
- Los iteradores son objetos a través de los cuales se puede acceder a los elementos del contenedor.
- Un iterador es similar a un puntero, sólo que al ser una clase provee mayores utilidades que éste.
- En la STL hay 70 algoritmos que se aplican sobre contenedores, haciendo uso de los iteradores. Al tener los contenedores iteradores comunes, un mismo algoritmo puede aplicarse a distintos contenedores. Hay algoritmos de búsqueda, de ordenamiento, de transformación, matemáticos, etc.

Contenedores: Definición

Un contenedor es un objeto capaz de almacenar otros objetos (del mismo tipo).
Contienen clases que implementan las estructuras de datos más utilizadas: vectores, listas, colas.
Acciones: Se pueden añadir, consultar y eliminar objetos de un contenedor sin tener que preocuparse de **reservar espacio**, **redimensionar** o **liberar memoria** en el contenedor.
Al ser templates (plantillas), pueden alojar cualquier tipo de dato o clase.

Contenedores: Tipos de Contenedores

- Dependiendo del tipo de problema a resolver conviene utilizar un tipo de contenedor u otro.
- La diferencia entre un contenedor y otro está en la forma en que los objetos son alojados, en cómo se crea la secuencia de elementos y la manera en que se puede acceder a cada uno de ellos.
- Éstos pueden estar almacenados en forma contigua en la memoria o enlazados a través de punteros.
- Esto hace que las estructuras difieran también en la forma en que se accede a los elementos, la velocidad con la cual se insertan o se eliminan y la eficiencia de los algoritmos que se apliquen a ellas.
- Para poder utilizar un contenedor: #include <vector>, #include <list>, #include <map>, etc.

Tipos de contenedores:

- **Contenedores de secuencia:** Almacenan los elementos en secuencia
  - **vector:** array unidimensional
  - **list:** lista doblemente enlazada
  - **deque:** cola (array unidimensional) con doble extremo

- **Contenedores asociativos:** Permite el acceso a los elementos mediante claves.
  - **set:** conjunto (sin duplicados)
  - **multiset:** multiconjunto o bolsa (con duplicados).
  - **map:** array asociativo
  - **multimap:** array asociativo (con duplicados).

- **Adaptadores de contenedores:** Se definen a partir de los contenedores de primera clase (contenedores de secuencia y asociativos), utilizando sólo algunas de sus características.
  - **stack:** pila
  - **queue:** cola
  - **priority_queue:** cola con prioridad

- **Casi contenedores:** No poseen todas las capacidades de los contenedores.
  - **Array del tipo del lenguaje C**
  - **string:** cadena de caracteres
  - **bitset:** conjunto de mapas de bits
  - **valarray:** contenedor tipo vector especial que permite operaciones matematicas más rápidas.

Iteradores: Definición

Un iterador es un objeto a través del cual se puede acceder a los elementos de un contenedor.
Un iterador es similar a un puntero, sólo que al ser una clase provee mayores utilidades que éste.
Un iterador permite seleccionar cada elemento de un contenedor sin necesidad de conocer cómo es la estructura interna de ese contenedor. Esto permite crear algoritmos genéricos que funcionen con cualquier contenedor, utilizando operaciones comunes como ++, -- o *.

Iteradores: Tipos de Iteradores

- **iterator**: iterador que apunta al tipo de elemento almacenado en un contenedor.
  con ++ avanza. Permite lectura y escritura
- **const_iterator**: iterador constante que apunta al tipo de elemento almacenado en un contenedor.
  con ++ avanza. Permite solo **lectura** (solo se puede usar para leer, no para modificar)
- **reverse_iterator**: iterador inverso que apunta al tipo de elemento almacenado en un contenedor; permite iterar en sentido inverso.
  con ++ retrocede. Permite lectura y escritura
- **const_reverse_iterator**: iterador inverso y constante que apunta al tipo de elemento almacenado en un contenedor; permite iterar en sentido inverso.
  con ++ retrocede. Permite solo **lectura**

Iteradores: Categoría de Iteradores

Categoría	**Capacidad**	**Movimiento**	**Pasadas**
Entrada (E)	Lectura	Adelante	Una (*)
Salida (S)	Escritura	Adelante	Una (*)
Avance (A)	Lectura y Escritura	Adelante	Una (*)
Bidireccional (B)	Lectura y Escritura	Adelante y atrás	Varias
Aleatorio (R)	Lectura y Escritura	Adelante, atrás y directo	Varias

(*) no se puede utilizar el mismo iterador para recorrer por 2ª vez la misma secuencia.

Iteradores: Operaciones de los Iteradores

Operacion	**Descripción**	**Categoría**
++p , p++	avance previo, posterior	todos
--p, p--	retroceso previo, posterior	B, R
*p	accede al contenido del iterador (a donde apunta el iterador)	todos
p1 = p2	asigna iterador p2 al iterador p1	todos
p + i p – i p += i p -= i p[i]	Avanza i veces (directo) Retrocede i veces (directo) equivale a p=p+i equivale a p=p-i equivale a *(p+i)	R R R R R
p1 < p2 p1 <= p2 p1 > p2 p1 >= p2 p1 == p2 p1 ! = p2	true: p1 se encuentra antes que p2 true: p1 antes o misma pos que p2 true: p1 está despues que p2 true p1 despues o misma pos que p2 true: p1 misma posicion que p2 true: p1 posicion diferente que p2	R R R R todos – S todos – S
advance(it,n) advance(it,-n)	for(int i=0;i<n;i++) it++ (it=it+n) it=it-n	todos (B,R) B, R

Iteradores: soportado por los contenedores

Contenedor	**Tipo de iterador**
vector **deque**	acceso aleatorio (todas las operaciones)
list **set** **multiset** **map** **multimap**	bidireccional (++, --, *, =, ==, !=)
stack **queue** **priority queue**	no soporta iterador

Iteradores de inserción y almacenamiento : Definición

Un iterador de inserción es un objeto a través del cual se puede insertar elementos en un contenedor secuencial.

Iteradores de inserción: Tipos

- **back_insert_iterator**: iterador de salida que inserta elementos al final de un contenedor.
  Ej: vector<int> v;
  back_insert_iterator< vector<int> > it(**v**); //iterador que inserta al final del vector v
- **front_insert_iterator**: iterador de salida que inserta elementos al inicio de un contenedor.
  Ej: list<fracc> l; //lista de objetos tipo fracc
  front_insert_iterator< list<fracc> > it(**l**); //iterador que inserta al inicio de la lista l
- **insert_iterator**: iterador de salida que inserta elementos en cualquier parte de un contenedor.
  Ej: list<fracc*> l; //lista de punteros a objetos tipo fracc
  list<fracc*>::iterator it; //iterador para la lista
  ...
  it=l.begin(); it++; it++; //it apunta al 3er elemento
  insert_iterator< list<fracc*> > it(**l, it**); //iterador que inserta a partir de it en la lista l

Ejemplo de uso de iteradores

```
#include <iostream>
#include <iterator>   //para usar iteradores
#include <vector>     //para usar el contenedor vector
using namespace std;

int main(int argc, char *argv[ ]) {
  int t[ ]={1,2,3,4}, n;
  n=sizeof(t)/sizeof(int)-1; //nº elementos-1
  vector<int> v(t,t+n) , c(v), d,e;//crea vector d,e vacio y v,c  con [1 2 3]
  vector<int>::iterator  it; //iterador para recorrer vector
  vector<int>::reverse_iterator  rit; //iterador para recorrer vector
  it=v.begin(); //it apunta al 1er elto del vector
  while (it!=v.end() ) {        //v.end() apunta a despues del ultimo elto
    cout << *it;    it++;
    if (it!=v.end()) cout << ",";
  }
  cout << endl;
  for(it=v.end()-1; it!=v.begin()-1; it--)    cout << *it << " ";
  cout << endl;
  for(rit=v.rbegin(); rit!=v.rend(); rit++)    //rbegin() apunta al ultimo
     cout << *rit << " ";                      //rend() antes del 1º
  cout << endl;
  it=v.begin();                  //it apunta al 1er elto del vector
  *(it+1)=8;                     //[1 8 3]
  it[2]=7;    e=v;               //[1 8 7]
  *it=0;                         //[0 8 7] it sigue apuntando al 1er elto...
  for(it=v.begin(); it!=v.end(); it++)    cout << *it << " ";
  cout << endl;
  it=v.begin()+2;                //it apunta al 3er elto del vector
  insert_iterator< vector<int> > iit(v, it);
  *iit=20;              //[0 8 20 7] inserta 20 en 3ª pos y avanza
  *iit=40;              //[0 8 20 40 7] inserta 40 en 4ª pos y avanza
  back_insert_iterator<  vector<int>  > bit(v);
  *bit=33;              //[0 8 20 40 7 33] inserta 33 al final
  *bit=55;              //[0 8 20 40 7 33 55] inserta 55 al final
  for(it=v.begin(); it!=v.end(); it++)     cout << *it << " ";
  for(cout << endl, it=c.begin(); it!=c.end(); it++)   cout << *it << " ";
  for(cout << endl, it=d.begin(); it!=d.end(); it++)  cout << *it << " ";
  for(cout << endl, it=e.begin(); it!=e.end(); it++)  cout << *it << " ";
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Todos los contenedores secuenciales y asociativos tienen métodos **begin()** y **end()** que devuelven respectivamente un iterator o const_iterator que hace referencia al primer elemento y a la siguiente posición después del final del contenedor, y métodos **rbegin()** y **rend()** que devuelve un reverse_iterator o const_reverse_iterator al último elemento y a la posición anterior al 1er elemento.

Los contenedores secuenciales y asociativos tienen constructores que permiten crear un contenedor vacio,constructores que permiten crear un contenedor con u-p elementos inicializados a los valores que hay en el rango determinado por los iteradores o punteros[p,u) y constructor de copia.

Todos los contenedores tienen destructores que se encargan de eliminar la memoria de los objetos que contienen y tienen sobrecargado el operator=

A la hora de recorrer todos los elementos de un contenedor es preferible usar iteradores (o reverse_iterator si el recorrido es inverso) ya que el código es el mismo independientemente del tipo de contenedor usado.

Si usamos métodos concretos de ese contenedor o iteradores bidireccionalesB (it- -) o aleatorios R (it+n,it-n) el código no sirve para cualquier contenedor ya que cada uno tiene métodos propios y no todos soportan iteradores B o R.

Pantalla:

```
1,2,3
3 2 1
3 2 1
0 8 7
0 8 20 40 7 33 55
1 2 3
//d no muestra nada porque esta vacio…
1 8 7       Presione una tecla para continuar . . .
```

Contenedores Secuenciales:

Los elementos del contenedor son almacenados en secuencia (cada elemento es precedido por un elemento específico y seguido por otro, excepto el 1º y el último).

Tipos de contenedores de secuencia

- **vector**: array unidimensional
- **list**: lista doblemente enlazada
- **deque**: cola (array unidimensional) con doble extremo

Características:

vector: array unidimensional con posiciones contiguas de memoria

Ej: vector<int> v;

vector<double> temperatura(10);

vector<Alumno> alumno(persona.begin(),personas.end());

- Se recomienda su uso si los datos deben ordenarse y ser fácilmente accesibles.
- Se puede recorrer usando iteradores y los operadores de subíndice [] o at
- Si la memoria asignada a un vector se agota entonces:
  1. Se le asigna un area contigua mas extensa
  2. Se copian los elementos originales.
  3. Se suprime la asignacion a la memoria anterior.

- acceso aleatorio rápido (a través del operator[] o el método at)
- Son lentos para insertar o eliminar elementos de posiciones intermedias
- Son rápidos insertando o eliminando elementos del final

deque: es un array unidimensional de doble entrada

Ej: deque<Persona> datos;

- Permite la inserción y eliminación al principio y al final del contenedor.
- Se implementa como un array unidimensional con posiciones no necesarimente contiguas
- Se puede recorrer usando iteradores y los operadores de subíndice [] o at

- acceso aleatorio rápido (a través del operator[]o el método at)
- Son lentos para insertar o eliminar elementos de posiciones intermedias
- Son rápidos insertando o eliminando elementos del principio y del final

list: Se implementa como una lista doblemente enlazada

Ej: list<double> notas;

list<Alumno> nombres alumno(persona.begin(),personas.end());

- Se puede recorrer usando iteradores

- acceso aleatorio lento
- Son rápidos para insertar o eliminar elementos en cualquier posición
- Son rápidos para acceder al inicio y al final

Contenedores Secuenciales: Mapa de métodos de los diferentes contenedores

Headers (fichero cabecera)		<array>	<vector>	<deque>	<forward_list>	<list>
Members		array	vector	deque	forward_list	list
	constructor	*implicit*	vector	deque	forward_list	list
	destructor	*implicit*	~vector	~deque	~forward_list	~list
	operator=	*implicit*	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin before_begin	begin
	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin		rbegin
	rend	rend	rend	rend		rend
const iterators	begin	cbegin	cbegin	cbegin	cbegin cbefore_begin	cbegin
	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin		crbegin
	crend	crend	crend	crend		crend
capacity	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty
	resize		resize	resize	resize	resize
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
	capacity		capacity			
	reserve		reserve			
element access	front	front	front	front	front	front
	back	back	back	back		back
	operator[]	operator[]	operator[]	operator[]		
	at	at	at	at		
modifiers	assign		assign	assign	assign	assign
	emplace		emplace	emplace	emplace_after	emplace
	insert		insert	insert	insert_after	insert
	erase		erase	erase	erase_after	erase
	emplace_back		emplace_back	emplace_back		emplace_back
	push_back		push_back	push_back		push_back
	pop_back		pop_back	pop_back		pop_back
	emplace_front			emplace_front	emplace_front	emplace_front
	push_front			push_front	push_front	push_front
	pop_front			pop_front	pop_front	pop_front
	clear		clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap
list operations	splice				splice_after	splice
	remove				remove	remove
	remove_if				remove_if	remove_if
	unique				unique	unique
	merge				merge	merge
	sort				sort	sort
	reverse				reverse	reverse
observers	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator

Legend (Leyenda):

C++98	Disponible en C++98
C++11	Nuevo en C++11

Contenedores asociativos: Mapa de métodos de los diferentes contenedores (Parte 1)

Headers		<set>		<map>	
Members		**set**	**multiset**	**map**	**multimap**
	constructor	set	multiset	map	multimap
	destructor	~set	~multiset	~map	~multimap
	assignment	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin
	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin	rbegin
	rend	rend	rend	rend	rend
const iterators	cbegin	cbegin	cbegin	cbegin	cbegin
	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin	crbegin
	crend	crend	crend	crend	crend
capacity	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty
	reserve				
element access	at			at	
	operator[]			operator[]	
modifiers	emplace	emplace	emplace	emplace	emplace
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
	insert	insert	insert	insert	insert
	erase	erase	erase	erase	erase
	clear	clear	clear	clear	clear
	swap	swap	swap	swap	swap
operations	count	count	count	count	count
	find	find	find	find	find
	equal_range	equal_range	equal_range	equal_range	equal_range
	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound
	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound
observers	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator
	key_comp	key_comp	key_comp	key_comp	key_comp
	value_comp	value_comp	value_comp	value_comp	value_comp
	key_eq				
	hash_function				
buckets	bucket				
	bucket_count				
	bucket_size				
	max_bucket_count				
hash policy	rehash				
	load_factor				
	max_load_factor				

Legend (Leyenda):

C++98	Disponible en C++98
C++11	Nuevo en C++11

Associative containers: Mapa de métodos de los diferentes contenedores (Parte 2)

Headers		<unordered_set>		<unordered_map>	
Members		**unordered_set**	**unordered_multiset**	**unordered_map**	**unordered_multimap**
	constructor	unordered_set	unordered_multiset	unordered_map	unordered_multimap
	destructor	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap
	assignment	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin
	end	end	end	end	end
	rbegin				
	rend				
const iterators	cbegin	cbegin	cbegin	cbegin	cbegin
	cend	cend	cend	cend	cend
	crbegin				
	crend				
capacity	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty
	reserve	reserve	reserve	reserve	reserve
element access	at			at	
	operator[]			operator[]	
modifiers	emplace	emplace	emplace	emplace	emplace
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
	insert	insert	insert	insert	insert
	erase	erase	erase	erase	erase
	clear	clear	clear	clear	clear
	swap	swap	swap	swap	swap
operations	count	count	count	count	count
	find	find	find	find	find
	equal_range	equal_range	equal_range	equal_range	equal_range
	lower_bound				
	upper_bound				
observers	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator
	key_comp				
	value_comp				
	key_eq	key_eq	key_eq	key_eq	key_eq
	hash_function	hash_function	hash_function	hash_function	hash_function
buckets	bucket	bucket	bucket	bucket	bucket
	bucket_count	bucket_count	bucket_count	bucket_count	bucket_count
	bucket_size	bucket_size	bucket_size	bucket_size	bucket_size
	max_bucket_count	max_bucket_count	max_bucket_count	max_bucket_count	max_bucket_count
hash policy	rehash	rehash	rehash	rehash	rehash
	load_factor	load_factor	load_factor	load_factor	load_factor
	max_load_factor	max_load_factor	max_load_factor	max_load_factor	max_load_factor

C++98	Disponible en C++98
C++11	Nuevo en C++11

Contenedores: Funciones miembro comunes a todos los contenedores secuenciales y asociativos

Constructores, destructores y operator=	
contenedor()	crea un contenedor vacio
contenedor(n)	crea un contenedor **secuencial** con n elementos inicializados a un valor por defecto
contenedor(n,x)	crea un contenedor **secuencial** con n elementos inicializados al valor x
contenedor(p,u)	crea un contenedor con u-p elementos inicializados a los valores que hay en el rango determinado por los iteradores[p,u)
contenedor(c)	constructor de copia
Destructor	Destruye el contenedor y todos sus elementos
c.operator=	asigna un contenedor a otro
Consulta de capacidad	
c.empty()	devuelve true si no hay elementos; en caso contrario, false
c.max size()	devuelve el número máximo de elementos que el contenedor puede guardar.
c.size()	devuelve el número de elementos actuales del contenedor
c.resize()	modifica el tamaño de un contenedor de secuencia. (sólo para **vector**, **list** y **deque**)
Operadores (no miembros) relacionales entre contenedores (contenedor priority no lo tiene)	
operator<(a,b)	operador " menor que" entre contenedores.
operator<=(a,b)	operador "menor o igual que" entre contenedores.
operator>(a,b)	operador "mayor que" entre contenedores.
operator>=(a,b)	operador "mayor o igual que" entre contenedores.
operator==(a,b)	operador "igual que" entre contenedores.
operator! =(a,b)	operador "distinto" entre contenedores.
Iteradores de los contenedores secuenciales y asociativos	
c.begin()	devuelve un iterator o const_iterator que hace referencia al primer elemento.
c.end()	devuelve un iterator o const_iterator que hace referencia a la siguiente posición después del final del contenedor.
c.rbegin()	devuelve un reverse_iterator o const_reverse_iterator al último elemento.
c.rend()	devuelve un reverse_iterator o const_reverse_iterator que hace referencia a la posición anterior al primer elemento.
Modificadores	
c1.swap(c2)	intercambia los elementos de los contenedores c1 y c2.
c.erase(p)	elimina el elemento apuntado por el iterador p.
c.erase(pri, ult)	elimina los elementos del intervalo de los iteradores[primero,ultimo)
c.clear()	elimina todos los elementos del contenedor

Contenedores: Funciones miembro comunes a los contenedores secuenciales (vector. list y deque)

c.push_back(x)	añade el elemento x al final del contenedor
c.pop_back()	elimina el ultimo elemento del contenedor.
c.push_front(x)	añade el elemento x al principio del contenedor (solo **deque** y **list**)
c.pop_front()	elimina el primer elemento (solo **deque** y **list**)
c.front()	Devuelve una referencia al primer elemento
c.back()	Devuelve una referencia al ultimo elemento
c[i]	Devuelve una referencia al elemento i-esimo (acceso sin verificación) (solo **vector** y **deque**)
c.at(i)	Devuelve una referencia al elemento i-esimo (acceso con verificación) (solo **vector** y **deque**)
c.assign(pri,ult)	asigna a c los elementos de otro contenedor indicados por los iteradores [pri,ult).
c.insert(pos, x) **c.insert(pos, n, x)** **c.insert(pos,p, u)**	Inserta el valor x en la posición anterior a la indicada por el iterador pos. Inserta n veces el valor x en la posición indicada por el iterador pos. Inserta en la posicion pos, los elementos del intervalo de los iteradores[p,u)

Contenedores: Funciones miembro específicas de listas (list)

c.splice(pos, l) **c.splice(pos, l, i)** **c.splice(pos, l, i, f)**	Mueve(elimina) los elementos de la lista l y los inserta en la lista c en la posición apuntada por el iterador pos. Tras ejecutarse la lista l se queda vacia. Mueve (elimina) el elemento de la lista l apuntado por el iterador i y lo inserta en la lista c en la posición apuntado por el iterador pos. Mueve los elementos [i,f) de la lista l y los inserta en la lista c en la posición pos.
c.remove(x)	elimina todos los elementos de la lista iguales a x. **Si x es un objeto, la clase de x debe tener sobrecargado operator= = para saber cuando dos objetos son iguales**
c.remove_if(pred)	elimina todos los elementos de la lista para los cuales el predicado pred es true.
c.merge(c2) **c.merge(c2, comp)**	elimina los elementos de la lista ordernada c2 y los añade a la lista ordenada c Mezcla ambas listas, las cuales previamente deben estar ordenadas. **Si la lista son objetos, su clase debe tener sobrecargado operator<** Idem pero usando la función comp (que hay que definir)
c.sort() **c.sort(comp)**	Ordena los elementos de la lista usando el operador < **Si la lista son objetos, su clase debe tener sobrecargado operator<** Ordena los elementos de la lista usando la función comp (que hay que definir)
c.unique() **c.unique(pred)**	Elimina cada elemento igual a su elemento precedente (se eliminan todos los duplicados excepto el primero de cada grupo) **Si la lista son objetos, su clase debe tener sobrecargado operator= =** Idem pero se usa el predicado pred para determinar cuando se elimina y cuando no
c.reverse()	Invierte una lista

Ejemplos:

```
#include <iostream>
#include <iterator>
#include <vector>
#include <list>
#include <deque>
using namespace std;
//si cambiamos vector por deque el resultado es el mismo
void ver(const vector<int> &v) {
  vector<int>::const_iterator it;
  cout << "[ ";
  for (it=v.begin(); it != v.end(); it++)     cout << *it << ' ';
  cout << "]\n";
}

void ver2(const vector<int> &v) {
  for (int i=0; i<v.size(); i++)     cout << v[i] << ' ';
}

int main () {
  int t[] = {6,8,7,9};
  vector<int>::iterator it;
  vector<int> a; //constructor por defecto //a:[ ] vector vacio
  vector<int> b(3,2); //constructor          //b:[2 2 2]
  a.push_back(1); a.push_back(2);       //a:[1 2]
  a[0]=3; ver(a);                        //a:[3 2]
  a.insert(a.begin()+1,4);               //a:[3 4 2]
  a.insert(a.begin()+1,2,0);             //a:[3 0 0 4 2]
  a.insert(a.end()-1, t, t+3);           //a:[3 0 0 4 6 8 7 2]
  a.erase(a.begin()+4); ver(a);          //a:[3 0 0 4 8 7 2]
  a.erase(a.begin()+1, a.begin()+3);     //a:[3 4 8 7 2]
  a.pop_back();                          //a:[3 4 8 7]
  vector<int> c(a.begin()+1,a.end()-1);  //c:[4 8]
  vector<int> d(c);  //constructor copia  //d:[4 8]
  c=a; ver2(c); cout << endl;            //c:3 4 8 7
  cout << c.front() <<"," << c.back();   //3,7
  c.assign(a.begin(), a.end()-2);        //c:[3 4]
  cout << endl; ver(c);
  d.swap(b);                             //d:[2 2 2]  b:[4 8]
  d.clear();                             //d:[ ]

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Al tener los contenedores funciones miembros comunes podemos cambiar el tipo de contenedor utilizado sin tener que modificar prácticamente nada del código.

En el ej izq podemos cambiar **vector** por **deque** sin tener que modificar nada del cógido y el resultado es el mismo.
Lo que cambia es cómo internamente está implementada la clase y los métodos, pero eso es transparente para el usuario.

Al pasar un objeto **const** el único iterador que se puede usar es un **const_iterator** o **const_reverse_iterator.**

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;
bool  mayor(int a, int b) {  return a>b; }
bool  igual(int a, int b) {  if (a==0) return false;
                             return a==b; }
bool un_digito (const int& v) { return (v<10); }
void ver(const list<int> &v) {
  for(list<int>::const_iterator it=v.begin(); it!=v.end(); it++)
     cout << *it << " ";
}

int main(int argc, char *argv[ ]) {
  list<int> v1, v2;
  list<int>::iterator it;

  for (int i=1; i<=4; ++i)  v1.push_back(i);       // v1: 1 2 3 4
  for (int i=1; i<=3; ++i)  v2.push_back(i*10);//v2: 10 20 30
  it = v1.begin();  ++it;  // apunta a 2
  v1.splice (it, v2);          // v1: 1 10 20 30 2 3 4, v2 (vacio)
                               // "it" aun apunta a 2 (el 5º elto)
  v2.splice (v2.begin(),v1, it);  // v1: 1 10 20 30 3 4, v2: 2
                                  // "it" es ahora invalido.
  it = v1.begin();
  advance(it,3);          // "it" apunta ahora a 30
  v1.splice ( v1.begin(), v1, it, v1.end()); // v1: 30 3 4 1 10 20
  for (int i=0; i<5; i++)  v2.push_front(i/2); //v2: 2 1 1 0 0 2
  v1.sort(mayor); v2.sort(mayor);         // v1: 30 20 10 4 3 1
  cout << "v2:"; ver(v2); cout << '\n';     //v2: 2 2 1 1 0 0
  v1.merge(v2, mayor);  // v1: 30 20 10 4 3 2 2 1 1 1 0 0, v2:
  cout << "v1:"; ver(v1); cout << '\n';
  cout << "v2:"; ver(v2); cout << '\n';
  v1.unique(igual);              // v1: 30 20 10 4 3 2 1 0 0
  v1.reverse();                  // v1: 0 0 1 2 3 4 10 20 30
  cout << "v1:"; ver(v1); cout << '\n';
  v1.remove(0);                  // v1: 1 2 3 4 10 20 30
  v1.remove_if(un_digito);  // v1: 10 20 30
  cout << "v1:"; ver(v1); cout << '\n';
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

unique() elimina todos los **duplicados** excepto el primero de cada grupo. Podemos indicarle una función o método que determine qué es un **duplicado**

remove_if(pred) elimina todos los elementos de la lista para los cuales el predicado pred es true. Pred puede ser una función o un método

sort() ordena los elementos de la lista Podemos indicarle una función comp que indique el criterio para ordenar

merge() mezcla listas ordenadas Podemos indicarle una función o método que indique el criterio para ordenar.

Ejemplo con clase:

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

class fracc {
 int n,d;
public:
  fracc(int a, int b) { n=a; d=b; }
  void ver() const { cout << n << "/" << d << " "; }
  bool operator<(fracc q) { return (n*q.d<d*q.n); }  //para merge(c) y sort()
  bool operator==(fracc f) { return (n*f.d==d*f.n); } //para remove(x) y unique()
  int getn() const { return n; }
  int getd() const { return d; }
  friend bool igual(fracc a, fracc b)  {   return (a.n==b.n && a.d==b.d);  }
  static bool mayor(fracc a, fracc b) {   return (a.n*b.d>a.d*b.n);  }
};

bool mayor2 (const fracc& v) { return ((float)v.getn()/v.getd() > 2); }
void ver(const list<fracc> &v) {
  for(list<fracc>::const_iterator it=v.begin(); it!=v.end(); it++)
     it->ver();
}

int main(int argc, char *argv[ ]) {
  list<fracc> v1, v2;
  list<fracc>::iterator it;

  for (int i=1; i<=4; ++i)  v1.push_back(fracc(i,1));       // v1: 1/1  2/1  3/1  4/1
  for (int i=1; i<=3; ++i)  v2.push_back(fracc(i*10,5));//v2: 10/5 20/5  30/5
  it = v1.begin();  ++it;  // apunta a 2/1
  v1.splice (it, v2);          // v1: 1/1 10/5  20/5  30/5  2/1  3/1  4/1    v2 (vacio)
                               // "it" aun apunta a 2/1 (el 5º elto)
  v2.splice (v2.begin(),v1, it);  // v1: 1/1  10/5  20/5  30/5  3/1  4/1  v2: 2/1
                                  // "it" es ahora invalido.
  it = v1.begin();
  advance(it,3);          // "it" apunta ahora a 30/5
  v1.splice ( v1.begin(), v1, it, v1.end());          // v1: 30/5  3/1  4/1  1/1  10/5  20/5
  for (int i=0; i<5; i++)  v2.push_front(fracc(i/2,1)); //v2: 2/1  1/1  1/1  0/1  0/1  2/1

  list<fracc> c1(v1), c2;                          // c1: 30/5  3/1  4/1  1/1  10/5  20/5   c2:
  c2=v2;                                           //c2: 2/1  1/1  1/1  0/1  0/1  2/1

  v1.sort();//fracc debe tener operator<              // v1: 1/1  10/5  3/1  4/1   20/5  30/5
  v2.sort();                                         //v2: 0/1  0/1  1/1  1/1  2/1  2/1
  v1.merge(v2); // v1: 0/1  0/1  1/1  1/1  1/1  10/5  2/1  2/1  3/1  4/1  20/5  30/5  v2:
  v1.remove(fracc(1,1));  // v1: 0/1  0/1  10/5  2/1  2/1  3/1  4/1  20/5  30/5
  v1.unique();                                       // v1: 0/1  10/5  3/1  4/1  30/5
  cout << "v1:"; ver(v1); cout << '\n';
  v1.reverse();                                      // v1: 30/5  4/1  3/1  10/5  0/1
  v1.remove_if(mayor2);                              // v1: 10/5  0/1

  c1.sort(fracc::mayor);                             //c1:30/5  4/1  20/5  3/1  10/5  1/1
  c2.sort(fracc::mayor);                             //c2:2/1  2/1  1/1  1/1  0/1  0/1
  c1.merge(c2, fracc::mayor);  //c1:30/5 4/1 20/5  3/1 10/5 2/1 2/1 1/1 1/1 1/1 0/1 0/1
  c1.unique(igual);                  //c1:30/5  4/1  20/5  3/1  10/5  2/1  1/1  0/1
  cout << "c1:"; ver(c1); cout << '\n';
  c1.remove(fracc(1,1));             //c1:30/5  4/1  20/5  3/1  10/5  2/1  0/1
  c1.reverse();                      //c1:0/1  2/1  10/5  3/1  20/5  4/1  30/5
  c1.remove_if (mayor2);            //c1:0/1  2/1  10/5
  system("PAUSE");   return EXIT_SUCCESS;
}
```

sort() ordena los elementos de la lista usando operator<
Si la lista contiene objetos, su clase debe tener sobrecargado operator<
Podemos indicarle una función comp que indique el criterio para ordenar

merge() mezcla listas ordenadas usando operator<
Si la lista contiene objetos, su clase debe tener sobrecargado operator<
Podemos indicarle una función o método que indique el criterio para ordenar.

unique() elimina todos los **duplicados** excepto el primero de cada grupo usando operator= =
Si la lista contiene objetos, su clase debe tener sobrecargado operator= =
Podemos indicarle una función o método que determine qué es un **duplicado**

remove(x) elimina todos los elementos de la lista iguales a x.
Si la lista contiene objetos, su clase debe tener sobrecargado operator= =

remove_if(pred) elimina todos los elementos de la lista para los cuales el predicado pred es true. Pred puede ser una función o un método

Pruebe a eliminar operator< y operator= = en la clase fracc y vea lo que ocurre…

En la función amiga igual considero que 2 fracc son iguales si tienen el mismo numerador y denominador

La función estatica mayor la utilizo para ordenar ascendentemente

Si list contiene punteros a objetos en lugar de objetos, la clase a la que pertenece los objetos no tiene que sobrecargar operator= = y operator< ya que lo que compara los métodos remove, unique, sort ym erge sería los valores de los punteros…

Si queremos ordenar los objetos a los que apunta los punteros hay que pasarle una función o método que determine como comparar…

Ejemplo con punteros a clase:

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

class fracc {
 int n,d;
public:
  fracc(int a, int b) { n=a; d=b; }
  void ver() const { cout << n << "/" << d << " "; }
  //bool operator<(fracc q) { return (n*q.d<d*q.n); }  //para merge(c) y sort()
  //bool operator==(fracc f) { return (n*f.d==d*f.n); } //para remove(x) y unique()
  int getn() const { return n; }
  int getd() const { return d; }
  friend bool igual(fracc *a, fracc *b)  {   return (a->n==b->n && a->d==b->d);  }
  static bool mayor(fracc *a, fracc *b) {   return (a->n*b->d  >  a->d * b->n);  }
};

bool mayor2 (fracc* v) { return ((float)v->getn()/v->getd() > 2); }
void ver(const list<fracc*> &v) {
  for(list<fracc*>::const_iterator it=v.begin(); it!=v.end(); it++)
     (*it)->ver();
}

void remove(list<fracc*> &v, fracc *f) {
  list<fracc*>::iterator it;
  for (it=v.begin(); it!=v.end();          )
    if (igual(*it, f))
      it=v.erase(it);   //elimino una a uno. No hago it++ porque he eliminado
    else
      it++;
}

int main(int argc, char *argv[ ]) {
  list<fracc*> v1, v2;
  list<fracc*>::iterator it;

  for (int i=1; i<=4; ++i)  v1.push_back(new fracc(i,1));       // v1: 1/1  2/1  3/1  4/1
  for (int i=1; i<=3; ++i)  v2.push_back(new fracc(i*10,5));//v2: 10/5 20/5  30/5
  it = v1.begin();  ++it;  // apunta a 2/1
  v1.splice (it, v2);          // v1: 1/1 10/5  20/5  30/5  2/1  3/1  4/1    v2 (vacio)
                               // "it" aun apunta a 2/1 (el 5º elto)
  v2.splice (v2.begin(),v1, it);  // v1: 1/1  10/5  20/5  30/5  3/1  4/1  v2: 2/1
                                  // "it" es ahora invalido.
  it = v1.begin();
  advance(it,3);          // "it" apunta ahora a 30/5
  v1.splice ( v1.begin(), v1, it, v1.end());          // v1: 30/5  3/1  4/1  1/1  10/5  20/5
  for (int i=0; i<5; i++)
     v2.push_front(new fracc(i/2,1)); //v2: 2/1  1/1  1/1  0/1  0/1  2/1

  v1.sort(fracc::mayor);                              //v1:30/5  4/1  20/5  3/1  10/5  1/1
  v2.sort(fracc::mayor);                              //v2:2/1  2/1  1/1  1/1  0/1  0/1
  v1.merge(v2, fracc::mayor); //v1:30/5 4/1 20/5  3/1 10/5 2/1 2/1 1/1 1/1 1/1 0/1 0/1
  v1.remove(new fracc(1,1));  //v1:30/5 4/1 20/5  3/1 10/5 2/1 2/1 1/1 1/1 1/1 0/1 0/1
                              //no elimina nada porque compara punteros...
  remove(v1, new fracc(1,1));       //v1:30/5  4/1  20/5  3/1  10/5  2/1  2/1  0/1  0/1
  v1.unique(igual);                  //v1:30/5 4/1  20/5  3/1  10/5  2/1  0/1
  cout << "v1:"; ver(v1); cout << '\n';
  v1.reverse();                      //v1:0/1  2/1  10/5  3/1  20/5  4/1  30/5
  v1.remove_if (mayor2);           //v1:0/1  2/1  10/5
  system("PAUSE");   return EXIT_SUCCESS;
}
```

En la función amiga igual, considero que 2 fracc son iguales si tienen el mismo numerador y denominador

La función estatica mayor la utilizo para ordenar ascendentemente

Si list contiene punteros a objetos en lugar de objetos, la clase a la que pertenece los objetos no tiene que sobrecargar operator= = y operator< ya que lo que compara los métodos remove, unique, sort y merge sería los valores de los punteros…

Si queremos ordenar (**sort**), mezclar (**merge**), o eliminar duplicados (**unique**) los objetos a los que apunta los punteros hay que pasarle una función o método ya que si no compararía los valores de los punteros y ordenaría o eliminaría según los valores de los punteros y no de a donde apunta esos punteros …

v1.sort() ordenaria por los valores de los punteros, no por los valores a donde apunta el puntero…

v1.unique() eliminaría si los punteros son duplicados, no si los valores a los que apuntan son duplicados…

remove(x) elimina todos los elementos de la lista iguales a x.
Si la lista contiene punteros a objetos, elimina solo si ambos punteros valen lo mismo, es decir, apuntan a la misma dirección de memoria, no si los objetos apuntandos por los punteros son iguales → no elimina porque son dir de memoria diferentes

Implementamos una función que elimine según el objeto al que apunta el puntero.

Ejemplo:
Crear unaclase Libro con los atributos (string autor y char *titulo) y con un método público info( ) que devuelva en un string el autor y titulo y otro método publico getAutor( ) que devuelva el autor.
Crea una clase Librería que permita almacenar un numero **ilimitado** de Libros (los objetos Libros son almacenados en el propio objeto Libreraa) y que tenga métodos que permitan agregar un libro, borrar el libro que se encuentra en una posición, borrar todos los libros de un autor, buscar la posición del primer libro de un autor, buscar la posición del ultimo libro de un autor, listar por pantalla todos los libros y devolver en un string el libro que esta en una determinada posición.

Implementar la clase Librería:

a) usando arrays dinámicos
b) usando un contenedor vector

Paso1: crear la clase Libro

```
#include <iostream>
#include <iterator>    //para usar iteradores
#include <vector>      //para usar el contenedor vector
#include <list>      //para usar el contenedor list
#include <string>      //para usar string
#include <sstream>   //para usar stringstream

using namespace std;

class Libro {      //al tener Libro memoria dinamica hay que
  string autor;   //hacer un constructor de copia, operator=
  char *titulo;     //y un destructor
public:
  Libro(string a="", char *tit="") { //constructor por defecto
    autor=a;
    titulo=new char[strlen(tit)+1];
    strcpy(titulo, tit);
  }

  Libro(const Libro &lib) { //constructor copia
    autor=lib.autor;
    titulo=new char[strlen(lib.titulo)+1];
    strcpy(titulo, lib.titulo);
  }

  Libro& operator=(const Libro &lib) { //operator=
    if (this != &lib) {
      autor=lib.autor;
      delete [] titulo;
      titulo=new char[strlen(lib.titulo)+1];
      strcpy(titulo, lib.titulo);
    }
    return *this;
  }

  ~Libro() {   //cout << "Destruyendo " << titulo << endl;
     delete [] titulo;
  }       //destructor

  string getAutor() { return autor; }

  string info() {
    stringstream s;
    s << autor << " " << titulo;
    return s.str();
  }
};
```

Al ejecutar este main( ) debe generar la salida mostrada:

```
int main(int argc, char *argv[]) {
   {
 Libreria lib;
 lib.agregar(Libro("Luis","Algebra"));
 lib.agregar(Libro("Pepe","Mi primer libro"));
 lib.agregar(Libro("Pepe","La Colmena"));
 lib.agregar(Libro("Pepe","El Cuerpo"));
 lib.agregar(Libro("Cela","Mi tercer libro"));
 lib.agregar(Libro("Cela","El Mendigo"));
 lib.agregar(Libro("Anonimo","El Santo"));
 lib.agregar(Libro("Pepe","La Ballena"));
 lib.listar();
 cout << "Primero de Pepe -> ";
 cout << lib.listar(lib.buscarprimero("Pepe")) << endl;
 int i=lib.buscarultimo("Cela");
 if (i!=-1) {
   cout << "Ultimo de Cela -> ";
   cout << i << ": " << lib.listar(i) << endl;
   lib.borrar(i);
 }
 lib.borrar("Pepe");
 cout << "tras borrar Pepe y ultimo de Cela\n";
 lib.listar();
   }
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

```
0: Luis Algebra
1: Pepe Mi primer libro
2: Pepe La Colmena
3: Pepe El Cuerpo
4: Cela Mi tercer libro
5: Cela El Mendigo
6: Anonimo El Santo
7: Pepe La Ballena
Primero de Pepe -> Pepe Mi primer libro
Ultimo de Cela -> 5: Cela El Mendigo
tras borrar Pepe y ultimo de Cela
0: Luis Algebra
1: Cela Mi tercer libro
2: Anonimo El Santo
Presione una tecla para continuar . . .
```

class Librería; //al alojar la clase Librería objetos Libro
//la clase Libro debe tener un constructor
//por defecto…

Paso2: crear la clase Librería (izq: con **arrays dinamicos**, dcha: con contenedor **vector**)

```
class Libreria {
  int n, nmax;
  Libro *lista; //Libro lista[nmax]
 public:
  Libreria() {
   n=0; nmax=n+1;
   lista=new Libro[nmax]; //constructor por defecto en Libro
  }

 ~Libreria() {   //cout << "Destruyendo Libreria...\n";
   delete [] lista; //al ser objetos se invoca los destructores
  }

  void agregar(const Libro &lib) { //& para que no hacer copia
   if (n==nmax) {
     Libro *aux=lista;
     nmax++;
     lista=new Libro [nmax];
     for(int i=0; i<n; i++)
       lista[i]=aux[i]; //copio los libros... (libro::operator=)
     delete [] aux;
   }
   lista[n]=lib;
   n++;
  }

  void borrar(int i) {
   if (i < n) {
     for(int j=i+1; j< n; j++)
       lista[j-1]=lista[j];
     n --;
   }
  }

  void borrar(string autor) {
   for(int i= n -1; i>=0; i--) {
     if (lista[i].getAutor()==autor) {
       for(int j=i+1; j< n; j++)
         lista[j-1]=lista[j];
       n --;
     }
   }
  }

  int buscarprimero(string autor) { //-1 si no existe
   for(int i=0; i< n; i++)
     if (autor==lista[i].getAutor())
       return i;
   return -1;
  }

  int buscarultimo(string autor) { //-1 si no existe
   for(int i= n -1; i>=0; i--)
     if (autor==lista[i].getAutor())
       return i;
   return -1;
  }

  void listar() {
   for(int i=0; i< n; i++)
     cout << i << ": " << lista[i].info() << "\n";
  }

  string listar(int i) { return lista[i].info(); }
};
```

```
class Libreria {
  vector<Libro> t; //Libro t[ilimitado]
  vector<Libro>::iterator it; //it: 'puntero' a Libro (Libro *it)
public:
  Libreria() {
    //nada, vector crea un vector vacio
  }
  ~Libreria() {    //cout << "Destruyendo Libreria...\n";
    //nada, la clase vector invoca los destructores de los
  }          // objetos que contiene

void agregar(const Libro &lib) { //& para que no hacer copia
    t.push_back(lib); //añade al final
  }

  void borrar(int i) {
    if (i < t.size()) {
      it=t.begin();    t.erase(it+i); //t.begin()+i
    }
  }

  void borrar(string autor) {
    for(it=t.begin(); it!=t.end();     ) {
      if (it->getAutor()==autor)
        it=t.erase(it); //erase no admite un  reverse_iterator it
      else it++;          //erase devuelve un it al elto que
    }                     //sigue al eliminado
  }

  void borrar2(string autor) { //idem a borrar
    it=t.begin();
    for(int i=t.size()-1; i>=0; i--)
      if (t[i].getAutor()==autor)
        t.erase(it+i); //t.begin()+i
  }

  int buscarprimero(string autor) { //sin iteradores...
    for(int i=0; i<t.size(); i++)
      if (autor==t[i].getAutor())
        return i;
    return -1;
  }

  int buscarultimo(string autor) {//con iteradores...
    vector<Libro>::reverse_iterator it;
    int i=t.size()-1;
    for(it=t.rbegin(); it!=t.rend(); it++,i--)
      if (autor==it->getAutor())
        return i;
    return -1;
  }

  void listar() { //con iteradores...
    int i=0;
    for (it=t.begin(); it!=t.end(); ++it, ++i)
      cout << i << ": " << it->info() << "\n";
  }

  void listar2() { //sin iteradores...
    for(int i=0; i<t.size(); i++)
      cout << i << ": " << t[i].info() << "\n";
  }

  string listar(int i) { return t[i].info(); }
  string listar2(int i) { it=t.begin(); return (it+i)->info(); }
};
```

Ejemplo: idem, pero ahora, la clase Librería no almacena los objetos **sino punteros a los objetos**

Implementar la clase Librería:

a) usando arrays dinámicos
b) usando listas enlazadas (contenedor **list**)

Paso1: crear la clase Libro (igual que antes…)

```
#include <iostream>
#include <iterator>    //para usar iteradores
#include <vector>      //para usar el contenedor vector
#include <list>        //para usar el contenedor list
#include <string>      //para usar string
#include <sstream>     //para usar stringstream

using namespace std;

class Libro {      //al tener Libro memoria dinamica hay que
  string autor;    //hacer un constructor de copia, operator=
  char *titulo;    //y un destructor
public:
  Libro(string a, char *tit) { //NO constructor por defecto
    autor=a;
    titulo=new char[strlen(tit)+1];
    strcpy(titulo, tit);
  }

  Libro(const Libro &lib) { //constructor copia
    autor=lib.autor;
    titulo=new char[strlen(lib.titulo)+1];
    strcpy(titulo, lib.titulo);
  }

  Libro& operator=(const Libro &lib) { //operator=
    if (this != &lib) {
      autor=lib.autor;
      delete [] titulo;
      titulo=new char[strlen(lib.titulo)+1];
      strcpy(titulo, lib.titulo);
    }
    return *this;
  }

  ~Libro() {   //cout << "Destruyendo " << titulo << endl;
     delete [] titulo;
  }       //destructor

  string getAutor() { return autor; }

  string info() {
    stringstream s;
    s << autor << " " << titulo;
    return s.str();
  }
};
```

Al ejecutar este main( ) debe generar la salida mostrada:

```
int main(int argc, char *argv[]) {
   {
 Libreria lib;
 lib.agregar(new  Libro("Luis","Algebra"));
 lib.agregar(new  Libro("Pepe","Mi primer libro"));
 lib.agregar(new  Libro("Pepe","La Colmena"));
 lib.agregar(new  Libro("Pepe","El Cuerpo"));
 lib.agregar(new  Libro("Cela","Mi tercer libro"));
 lib.agregar(new  Libro("Cela","El Mendigo"));
 lib.agregar(new  Libro("Anonimo","El Santo"));
 lib.agregar(new  Libro("Pepe","La Ballena"));
 lib.listar();
 cout << "Primero de Pepe -> ";
 cout << lib.listar(lib.buscarprimero("Pepe")) << endl;
 int i=lib.buscarultimo("Cela");
 if (i!=-1) {
   cout << "Ultimo de Cela -> ";
   cout << i << ": " << lib.listar(i) << endl;
   lib.borrar(i);
 }
 lib.borrar("Pepe");
 cout << "tras borrar Pepe y ultimo de Cela\n";
 lib.listar();
   }
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

```
0: Luis Algebra
1: Pepe Mi primer libro
2: Pepe La Colmena
3: Pepe El Cuerpo
4: Cela Mi tercer libro
5: Cela El Mendigo
6: Anonimo El Santo
7: Pepe La Ballena
Primero de Pepe -> Pepe Mi primer libro
Ultimo de Cela -> 5: Cela El Mendigo
tras borrar Pepe y ultimo de Cela
0: Luis Algebra
1: Cela Mi tercer libro
2: Anonimo El Santo
Presione una tecla para continuar . . .
```

class Librería; //al alojar la clase Librería punteros a
// objetos Libro la clase Libro no
// necesita un constructor por defecto…

Ejecute el código mostrando los mensajes del destructor de Libro y Libreria…
A continuación, comente los 3 **delete lista[i]** que encuentre en el código y vuélvalo a ejecutar…

…Comprenderá la importancia de los **delete lista[i]** y **delete (*it)** que hay en el código.

En el ejemplo anterior que no usa punteros a Libro sino objetos Libro no son necesarios porque:
- al eliminar un array de objetos se ejecuta los destructores de los objetos que contiene
- si el array es de punteros a objetos, el programador debe liberar la memoria explícitamente con **delete**

Paso2: crear la clase Librería (izq: con **arrays dinamicos**, dcha: con contenedor **list**)

```
class Libreria {
  int n, nmax;
  Libro **lista; //Libro* lista[nmax]
 public:
  Libreria() {
    n=0; nmax=n+1;
    lista=new Libro*[nmax]; //NO constructor por defecto
  }                                  // en Libro
 ~Libreria() {   //cout << "Destruyendo Libreria...\n";
    for(int i=0; i<n; i++)
      delete lista[i]; //liberando memoria de los objetos
    delete [] lista; //liberando memoria de lista
  }

  void agregar(Libro *lib) { //* puntero a Libro
    if (n==nmax) {
      Libro **aux=lista;      nmax++;
      lista=new Libro* [nmax];
      for(int i=0; i<n; i++)
        lista[i]=aux[i]; //copio los punteros... no los objetos
      delete [] aux;
    }
    lista[n]=lib;    n++;
  }

  void borrar(int i) {
    if (i < n) {
      delete lista[i]; //libero memoria
      for(int j=i+1; j< n; j++)
        lista[j-1]=lista[j];
      n --;
    }
  }

  void borrar(string autor) {
    for(int i= n -1; i>=0; i--) {
      if (lista[i]->getAutor()==autor) {
        delete lista[i]; //libero memoria
        for(int j=i+1; j< n; j++)
          lista[j-1]=lista[j];
        n --;
      }
    }
  }

  int buscarprimero(string autor) { //-1 si no existe
    for(int i=0; i< n; i++)
      if (autor== lista[i]->getAutor())
        return i;
    return -1;
  }

  int buscarultimo(string autor) { //-1 si no existe
    for(int i= n -1; i>=0; i--)
      if (autor== lista[i]->getAutor())
        return i;
    return -1;
  }

  void listar() {
    for(int i=0; i< n; i++)
      cout << i << ": " << lista[i]->info() << "\n";
  }

  string listar(int i) { return lista[i]->info(); }
};
```

```
class Libreria {
  list<Libro*> t; //Libro* t[ilimitado]
  list<Libro*>::iterator it; //it: 'puntero' a Libro* (Libro **it)
public:
  Libreria() {
    //nada, vector crea un list vacio
  }
  ~Libreria() {    //cout << "Destruyendo Libreria...\n";
    for(it=t.begin(); it!=t.end(); it++)
      delete (*it);  //libera memoria objeto apuntado por *it
  }                  //*it es un Libro* (puntero a objeto Libro)
void agregar(Libro *lib) { //* puntero a Libro
    t.push_back(lib) ; //añade al final
  }

  void borrar(int i) { //list no permite acceso directo it+i
    if (i < t.size()) {   //usar advance en vez de +
      it=t.begin();    //it apunta al primer elemento
      advance(it, i); //it apunta al elemento it+i
      delete (*it);     //libero memoria
      t.erase(it);     //elimino el elemento apuntado por it
    }
  }

  void borrar(string autor) {
    for(it=t.begin(); it!=t.end();     ) {
      if ((*it)->getAutor()==autor)  {
       delete (*it); //libero memoria
        it=t.erase(it); //erase no admite un  reverse_iterator it
      }                  //erase devuelve un it al elto que
      else it++;          //sigue al eliminado
    }
  }

  int buscarprimero(string autor) { //con iteradores...
    int i=0;
    for(it=t.begin(); it!=t.end(); it++,i++)
      if (autor==(*it)->getAutor())
        return i;
    return -1;
  }
  int buscarultimo(string autor) {//con iteradores...
    list<Libro*>::reverse_iterator it;
    int i=t.size()-1;
    for(it=t.rbegin(); it!=t.rend(); it++,i--)
      if (autor==(*it)->getAutor())
        return i;
    return -1;
  }

  void listar() { //con iteradores...
    int i=0;
    for (it=t.begin(); it!=t.end(); ++it, ++i)
      cout << i << ": " << (*it)->info() << "\n";
  }

  string listar(int i) {
    it=t.begin();
    advance(it, i); //list no permite it+i
// advance(it, i); es igual a  for(int j=0; j < i; j++) it++;
    return (*it)->info();
  }

};
```

Contenedores Secuenciales:

Los elementos del contenedor son almacenados en secuencia (cada elemento es precedido por un elemento específico y seguido por otro (excepto el 1º y el último).

Tipos de contenedores de secuencia

- vector: array unidimensional
- list: lista doblemente enlazada
- deque: cola (array unidimensional) con doble extremo

Características:

vector: array unidimensional con posiciones contiguas de memoria

Ej: vector<int> v;

vector<double> temperatura(10);

vector<Alumno> alumno(persona.begin(),personas.end());

- Se recomienda su uso si los datos deben ordenarse y ser fácilmente accesibles.
- Se puede recorrer usando iteradores y los operadores de subíndice [] o at
- Si la memoria asignada a un vector se agota entonces:
  4. Se le asigna un area contigua mas extensa
  5. Se copian los elementos originales.
  6. Se suprime la asignacion a la memoria anterior.

- acceso aleatorio rápido (a través del operator[] o el método at)
- Son lentos para insertar o eliminar elementos de posiciones intermedias
- Son rápidos insertando o eliminando elementos del final

deque: es un array unidimensional de doble entrada

Ej: deque<Persona> datos;

- Permite la inserción y eliminación al principio y al final del contenedor.
- Se implementa como un array unidimensional con posiciones no necesarimente contiguas
- Se puede recorrer usando iteradores y los operadores de subíndice [] o at

- acceso aleatorio rápido (a través del operator[]o el método at)
- Son lentos para insertar o eliminar elementos de posiciones intermedias
- Son rápidos insertando o eliminando elementos del principio y del final

list: Se implementa como una lista doblemente enlazada

Ej: list<double> notas;

list<Alumno> nombres alumno(persona.begin(),personas.end());

- Se puede recorrer usando iteradores

- acceso aleatorio lento
- Son rápidos para insertar o eliminar elementos en cualquier posición
- Son rápidos para acceder al inicio y al final

Adaptadores de Contenedores:

Se definen a partir de los contenedores secuenciales utilizando sólo algunas de sus características.

- En los adaptadores de contenedores el programador puede seleccionar el contenedor secuencial en el que se basa.Por ej, podemos crear un objeto **stack** basado en un **vector** y otro basado en un **list**.
- Los adaptadores de Contenedores **no disponen de iteradores**.

Tipos de adaptadores de contenedores

- **stack:** pila (ultimo en entrar, primero en salir)
- **queue:** cola (primero en entrar, primero en salir)
- **priority_queue:** cola con prioridad (el elemento que sale es el que tiene más prioridad)

Características:

stack: representa una pila de elementos (ultimo en entrar, primero en salir).

- puede utilizar cualquier contenedor secuencial, si no se indica nada-> por defecto utiliza **deque**

Ej: stack<int> pila_enteros; //basado en **deque**
stack <double*, **vector**<double*> > t; //basado en **vector**
stack <Alumno> **list**<Alumno> >pila_alumnos; //basado en **list**

queue: representa una cola de elementos (primero en entrar, primero en salir).

- puede utilizar los contenedores **deque** y **list**, si no se indica nada-> por defecto utiliza **deque**

Ej: queue<int> cola_enteros; //basado en **deque**
queue <Alumno> **list**<Alumno> >cola_alumnos; //basado en **list**

priority_queue: representa una cola de prioridad (el elemento que sale es el de más prioridad).

- se hacen inserciones ordenadas y siempre se extrae por su parte inicial.
- puede utilizar los contenedores **vector** y **deque**, si no se indica nada-> por defecto utiliza **vector**

Ej: priority_queue <int> prioridad_enteros; //basado en **vector**
priority_queue <Alumno> **deque** <Alumno> >prioridad_alumnos; //basado en **deque**

Adaptadores de Contenedores: Funciones miembro específicas de stack, queue y priority_queue

Comunes a stack, queue y priority_queue	
adapatador.empty()	true si el adpatador es ta vacío.
adaptador.size()	devuelve el número de elementos del adaptador.
Específicas de stack	
pila.push(x)	añade el elemento x a la cima de la pila.
pila.pop()	elimina el elemento situado en la cima de la pila.
pila.top()	devuelve una referencia al elemento de la cima de la pila.
Específicas de queue	
cola.push(x)	añade el elemento x al final de la cola.
cola.pop()	elimina el primer elemento de la cola.
cola.front()	devuelve una referencia al primer elemento de la cola.
cola.back()	devuelve una referencia al ultimo elemento de la cola.
Específicas de priority_queue	
prioridad.push(x)	añade el elemento x a la cola con prioridad.
prioridad.pop()	elimina el elemento de mayor prioridad de la cola.
prioridad.top()	devuelve una referencia al elemento de mayor prioridad.

Contenedores Asociativos (ordenados):

Los elementos del contenedor no son almacenados en secuencia sino que se implementan como árboles binarios de búsqueda balanceados. Para cada elemento a almacenar se debe proporcionar también una clave que lo identifica unívocamente.
Se llaman asociativos porque asocian claves con valores.
Son contenedores asociativos ordenados porque sus eltos se guardan en orden ascendente de sus claves.
Para la comparación de 2 eltos, el usuario puede suministrar una función booleana que compare 2 objetos y devuelva true si el primero precede al segundo.
Por defecto utiliza el objeto función **less** que internamente usa operator< para comparar objetos.

Tipos de contenedores asociativos

- set: conjunto (sin duplicados)
- multiset: multiconjunto (permite duplicados)
- map: Array asociativo compuesto por pares de elementos.
  No permite duplicados: asocia un único valor a cada clave única
- multimap: Array asociativo compuesto por pares de elementos.
  Permite duplicados: puede asociar varios valores auna clave

Características:

set: representa un conjunto de elementos ordenados
Ej: set<Persona> grupo(persona.begin(),personas.end());
set<Persona, **less<Persona>** > grupo(persona.begin(),personas.end()); //igual, pero explicito

- Se utiliza para almacenar y recuperar claves únicas
- No posee elementos duplicados
- Se puede recorrer usando iteradores bidireccionales, pero no iteradores de acceso aleatorio

multiset: representa un multiconjunto de elementos ordenados
Ej: multiset<Fecha> fechas;
multiset<Fecha, **greater<Fecha>** > fechas

- Se utiliza para almacenar y recuperar claves que no son únicas
- Permite elementos duplicados
- Se puede recorrer usando iteradores bidireccionales, pero no de acceso aleatorio

map: representa un conjunto de pares de elementos ordenados
Ej: map<int,string> dni_personas;
map<int,string, **classcompare<int>** > dni_personas; //utiliza un objeto funcion definido propio

- Se utiliza para almacenar y recuperar claves que son únicas
- No permite elementos duplicados
- Se puede recorrer usando iteradores bidireccionales, pero no de acceso aleatorio
- El primer elemento del par representa la clave y el segundo el valor

multiset: representa un multiconjunto de pares de elementos ordenados
Ej: multimap<int,string> dni_personas ;

- Se utiliza para almacenar y recuperar claves que no son únicas
- Permite elementos duplicados
- Se puede recorrer usando iteradores bidireccionales, pero no de acceso aleatorio
- El primer elemento del par representa la clave y el segundo el valor

c.insert(x) **c.insert(ini, fin)**	Inserta el elemento x en el contenedor Inserta los elementos del intervalo de los iteradores[ini,fin)
c.erase(x) **c.erase(p)** **c.erase(ini, fin)**	Borra el elemento que tenga la clave x Borra el elemento apuntado por el iterador p Borra los elementos comprendidos en el intervalo de los iteradores [ini, fin)
c.count(x)	Devuelve la cantidad de elementos que tienen la clave x
c.find(x)	Devuelve un iterador al primer elemento que tenga la clave x, sino devuelve end()
c. lower_bound(x)	Devuelve un iterador al primer elemento con la clave <= x. Si no hay devuelve end()
c.upper_bound(x)	Devuelve un iterador al primer elemento con la clave > x. Si no, devuelve end()

Ej: map que almacena el propio objeto

```
#include <iostream>
#include <iterator>
#include <map>
#include <string>
#include <sstream>

using namespace std;

class Persona { //se debe implementar el contructor copia
  string dni;      //y operator= ya que hay memoria dinamica
  char *nombre; //no se ha hecho por razones de espacio
  int edad;
public:
//obligatorio constructor por defecto
  Persona(string dni="", char *nom="", int e=0) {
    this->dni=dni;
    nombre=new char[strlen(nom)+1];
    strcpy(nombre, nom);
    edad=e;
  }
  ~Persona () { delete [] nombre; }
  string getdni() const { return dni; }
  string info() const {
    stringstream s;
    s << dni << " " << nombre << " " << edad << endl;
    return s.str();
  }
};

typedef map<string, Persona> MAP;

class Hacienda {
  map<string, Persona> lista;
public:
  Hacienda() { } //no hace nada
  ~Hacienda() {  }//no hace nada
  void alta(const Persona &p) { lista[p.getdni()]=p;    }
  void baja(string dni) {  lista.erase(dni); } //borra el objeto

  string buscar(string dni) {
    string s="No encontrado\n";
    MAP::iterator it;
    it=lista.find(dni);
    if (it != lista.end())
      return it->second.info();
    return s;
  }
  void ver() {
    map<string, Persona >::iterator it;
    for (it=lista.begin(); it!=lista.end(); ++it) {
      string s=it->first; //clave
      cout << s << " => " << it->second.info();
    }
  }
};

int main(int argc, char *argv[]) {
  Persona a=Persona("123", "juan", 25);
  Persona b=Persona("124", "luis", 22);
  Persona c=Persona("125", "ana", 19);
  Persona d=Persona("126", "eva maria", 30);
    {
  Hacienda h;
  h.alta(a); h.alta(a);
  h.alta(b); h.alta(c); h.alta(b); h.alta(d);
  h.ver(); cout << "bajas....\n";
  h.baja("125"); h.baja("123"); h.baja("125");
  cout << h.buscar("125") << h.buscar("126");
  h.ver();
    }
//al salir de este bloque se destruye h que tiene a 124 y 126
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Pantalla:

```
123 => 123 juan 25
124 => 124 luis 22
125 => 125 ana 19
126 => 126 eva maria 30
bajas....
No encontrado
126 eva maria 30
124 => 124 luis 22
126 => 126 eva maria 30
Presione una tecla para continuar . . .
```

La clase Hacienda utiliza un map para almacenar objetos Persona utilizando como clave el dni.
Para acceder a la clave: it->**first**
Para acceder al valor: it->**second**

Al borrar se destruye el objeto que contiene

Ej: map que almacena un puntero al objeto

```
#include <iostream>
#include <iterator>
#include <map>
#include <string>
#include <sstream>

using namespace std;

class Persona { //se debe implementar el contructor copia
  string dni;      //y operator= ya que hay memoria dinamica
  char *nombre; //no se ha hecho por razones de espacio
  int edad;
public:
//no obligatorio constructor por defecto
  Persona(string dni, char *nom, int e) {
    this->dni=dni;
    nombre=new char[strlen(nom)+1];
    strcpy(nombre, nom);
    edad=e;
  }
  ~Persona () { cout << "destruyendo " << dni << endl;
                delete [] nombre; }
  string getdni() const { return dni; }
  string info() const {
    stringstream s;
    s << dni << " " << nombre << " " << edad << endl;
    return s.str();
  }
};

typedef map<string, Persona *> MAP;

class Hacienda {
  map<string, Persona *> lista;
public:
  Hacienda() { } //no hace nada
  ~ Hacienda() { //debo liberar memoria de los objetos
    MAP::iterator it;
    for (it=lista.begin(); it!=lista.end(); ++it) {
      delete it->second;
    }
  }
  void alta(Persona *p) { lista[p->getdni()]=p;    }
  void baja(string dni) {
       map<string, Persona *>::iterator it;
       it=lista.find(dni);
       Persona *p=NULL;
       if (it != lista.end()) {
         p=it->second;
         lista.erase(dni);
         delete p;
       }
  }

  string buscar(string dni) {
    string s="No encontrado\n";
    MAP::iterator it;
    it=lista.find(dni);
    if (it != lista.end())
      return it->second->info();
    return s;
  }
```

```
  void ver() {
    map<string, Persona* >::iterator it;
    for (it=lista.begin(); it!=lista.end(); ++it) {
      string s=it->first; //clave
      Persona *p=it->second; //valor
      cout << s << " => " << p->info();
    }
  }
};

int main(int argc, char *argv[]) {
  Persona *a=new Persona("123", "juan", 25);
  Persona *b=new Persona("124", "luis", 22);
  Persona *c=new Persona("125", "ana", 19);
  Persona *d=new Persona("126", "eva maria", 30);
     {
  Hacienda h;
  h.alta(a); h.alta(a);
  h.alta(b); h.alta(c); h.alta(b); h.alta(d);
  h.ver(); cout << "bajas....\n";
  h.baja("125"); h.baja("123"); h.baja("125");
  cout << h.buscar("125") << h.buscar("126");
  h.ver();
     }
//al salir de este bloque se destruye h que tiene a 124 y 126
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Pantalla:

```
123 => 123 juan 25
124 => 124 luis 22
125 => 125 ana 19
126 => 126 eva maria 30
bajas....
destruyendo 125
destruyendo 123
No encontrado
126 eva maria 30
124 => 124 luis 22
126 => 126 eva maria 30
destruyendo 124
destruyendo 126
Presione una tecla para continuar . . . .
```

La clase Hacienda utiliza un map para almacenar objetos Persona utilizando como clave el dni.
Para acceder a la clave: it->first
Para acceder al valor: it->second

En el método baja utilizo un puntero Persona para liberar la memoria, ya que al ser punteros el map borra el puntero pero no la memoria adonde apunta.

En el destructor deHacienda debo recorrer los elementos del map lista y liberar la memoria de los objetos.

Algoritmos de Contenedores (están definidos en <algorithm>):

En la STL hay 70 algoritmos que se aplican sobre contenedores, pero no trabajan directamente con los contenedores, sino con los iteradores proporcionados por dichos contenedores.
Al tener contenedores iteradores comunes, un mismo algoritmo puede aplicarse a distintos contenedores.
Los algoritmos contienen una selección de los algoritmos más comunes y actúan de forma genérica sobre los contenedores **y los arrays**. Incluye los algoritmos de búsqueda y ordenación más comunes.

Argumentos de los algoritmos: iteradores bidireccionales o de acceso aleatorio
Algunos argumentos de los algoritmos son pares de iteradores que limitan los elementos del contenedor:
- Iterador que apunta al primer elemento que se quiere procesar del contenedor.
- Iterador que apunta a la posicion situada despues del ultimo elemento que se quiere procesar

Ej: algoritmo(c.**begin()**, c.**end()**, ...); //c es un contenedor

Tipos de algoritmos: (para su uso se requiere <algorithm>)

- **Algoritmos observadores o invariantes**: realizan una consulta sobre (un rango de iteradores de) un contenedor, sin modificar sus elementos.

- **Algoritmos modicadores**: modican el contenido (de un rango de iteradores) del contenedor.

- **Algoritmos de ordenación y operaciones relacionadas**: son algoritmos basados en el orden. Se incluye operaciones sobre conjuntos (requieren que la secuencia este ordenada para ser eficiente).

  - **Operaciones de ordenación**: agrupa los algoritmos que ordenan secuencias.
  - **Operaciones de conjuntos**: Se pueden aplicar a secuencias ordenadas (contenedores ordenados) y también a **set** y **multiset**. Si se aplica a secuencias no ordenadas, la secuencia resultante puede no ser correcta.
  - **Operaciones de montículos**: agrupa los algoritmos que trtabajan con montículos (secuencia ordenada de elementos organizados como un árbol binario).
  - **Otras operaciones**: algoritmos mínimos y máximo, comparaciones lexicográficas y permutaciones.

- **Algoritmos numéricos generalizados**: algoritmos sólo aplicables a colecciones numéricas o contenedores de tipo numéricos. Para su uso se debe incluir <**numeric**>.

Algoritmos observadores o invariantes:

1. **Iterador**: for each.
2. **Busqueda en una secuencia no ordenada**: find, find_if, find_first_of, adjacent_find.
3. **Busqueda de subsecuencias en una secuencia no ordenada**: find_end, search, search_n.
4. **Contar y comparar**: count, count_if, equal, mismatch.

Algoritmos observadores o invariantes:

for_each(ini,fin,func)	Aplica la función unaria func a los eltos del contenedor delimitado por los iteradores [ini, fin)
find(i,f,x)	devuelve un iterador al 1er elemento de [i, f) que sea igual al valor x.
find_if(i,f,pred)	devuelve un it al 1er elemento de [i, f) que haga true a la función unaria pred.
find_first_of(i,f,a,b)	devuelve un iterador al 1er elemento de [i, f) que coincide con un elemento de [a, b). Los iteradores pueden referirse a un mismo contenedor o distinto.
find_first_of(i,f,a,b,pred)	Idem pero que haga true la función binaria pred
adjacent_find(i,f) **adjacent_find(i,f, pred)**	devuelve un it al 1er elto de [i, f) cuyo adyacente sea igual a él. (busca dos adyacentes que coincidan) Usa operator= = para comparar (o pred binario). **En todos ellos si no encuentra lo buscado devuelve el iterador f**
find_end(i,f,a,b)	Busca la ultima aparición de la subsecuencia [i, f) dentro de la secuencia [a, b). devuelve un iterador al 1er elto de la ultima aparición
find_end(i,f,a,b, pred)	Usa operator= = para comparar (o pred binario en versión 2).
search(i,f,i2,f2)	busca la subsecuencia [i2, f2) en la secuencia [i, f), devuelve un iterador it al 1er elto de [i,f) donde empieza la subsecuencia, si no devuelve f.
search(i,f,i2,f2,pred)	usa operator= = para comparar (o pred binario en versión 2).
search_n(i,f,n,x)	Busca una secuencia de, al menos, n eltos iguales a x en [i,f); devuelve un iterador al 1er elemento de la secuencia de n coincidencias, sino devuelve f
search_n(i,f,n,x, pred)	Usa operator= = para comparar (o pred binario en versión 2). **En todos ellos si no encuentra lo buscado devuelve el iterador f**
count(i,f,x)	devuelve el nº de veces que el valor x se encuentra en [i, f) .
count_if(i,f,pred)	devuelve el nº de eltos de [i, f) que hacen true la función unaria pred.
equal(i,f,i2) **equal(i,f,i2,pred)**	devuelve true si todos los valores [i, f) son iguales en el mismo orden a[i2,…). Usa operator= = para comparar (o pred binario en versión 2).
mismatch(i,f,i2)	busca el 1er par de eltos de [i, f) y [i2,…) que son diferentes y devuelve un par de iteradores a dichos elementos.
mismatch (i,f,i2,pred)	usa operator= = para comparar (o pred binario en versión 2).

Algoritmos modificadores:

5. **Copiar y asignar**: copy, copy_backward, fill, fill_n, generate, generate_n.
6. **Intercambiar**: swap, swap_ranges, iter_swap, partition, stable_partition
7. **Borrar**: remove, remove_if, unique.
8. **Copiar y borrar**: remove_copy, remove_copy_if, unique_copy.
9. **Sustituir**: replace, replace_if, replace_copy, replace_copy_if, transform.
10. **Invertir y rotar**: reverse, reverse_copy, rotate, rotate_copy.
11. **Permutar**: random_shuffle.

Algoritmos modificadores:

copy(i,f,i2) **copy_backward(i,f,f2)**	Copia la secuencia [i, f) a otro intervalo que comienza en i2. Copia la secuencia [i,f) a otro intervalo que termina en f2 (f en f2, f-1 en f2-1)
fill(i,f,x) **fill_n(i,n,x)**	Copia el valor x en el intervalo [i,f). [x x x x …) Copia el valor x n veces comenzando en i. [x x … n veces]
generate (i,f,gen) **generate_n(i,n,gen)**	Asigna a los eltos de [i,f) valores generados por la función gen Asigna a los n primeros eltos de [i,…) valores generados por gen.
swap(c1,c2)	Intercambia los elementos de dos contenedores c1 y c2.
swap_ranges (i,f,i2)	Intercambia los elementos [i,f) con los que empiezan en i2.
iter_swap (i1,i2)	intercambia los elementos i1 y i2.
partition (i,f,pred)	Coloca al inicio de [i,f) los eltos de [i,f) que satisfacen un predicado unario (no preserva orden original de los ele) devuelve it al 1er ele q no cumple pred
stable_ partition (i1,i2)	Idem, pero preservando el orden original de los elementos.
remove(i,f,x) **remove_if(i,f,pred)**	elimina los eltos de [i,f) con valor = = x. devuelve un it al final del rango idem, pero elimina los eltos de [i,f) que hacen true el predicado unario pred.
unique(i,f) **unique(i,f,pred)**	elimina los eltos de [i,f) duplicados consecutivos . Usa operator= = para comparar (o pred binario en versión 2).
remove_copy(i,f,i2,x) **remove_copy_if(i,f,i2,p)**	Copia todos los eltos de [i,f) a [i2,…) excepto aquellos con valor x. Idem excepto aquellos que hagan true el predicado unario p.
unique_copy(i,f,i2) **unique_copy(i,f,i2,pred)**	Realiza copia sin duplicados, copia los eltos de [i,f) a [i2,…) sin duplicados. Usa operator= = para comparar (o pred binario en versión 2).
replace(i,f,x,y) **replace_if(i,f,pred,y)**	Sustituye los elementos de [i,f) con valor x por y. Sustituye los eltos de [i,f) que hacen true el predicado unario pred por y.
replace_copy(i,f,i2,x,y) **replace_copy_if(i,f,i2,p,y)**	Copia [i,f) a [i2,…) sustituyendo los elementos con valor x por y. idem sustituyendo los eltos que hacen true el predicado unario pred por y.
transform (i,f,i2,op)	Aplica la operación unaria op secuencialmente a los eltos de [i,f) y copia el resultado en la secuencia [i2,…)
transform (i,f,i',i2,op)	Aplica la operación binaria op secuencialmente a los eltos de [i,f) y [i',…) y copia el resultado en la secuencia [i2,…) , es decir op(i,i'), op(++i,++i'),…
reverse(i,f) **reverse_copy (i,f.i2)**	Invierte el orden de los elementos del intervalo de los iteradores [i,f) Copia [i,f) de forma inversa en [i2,…), (invierte el orden en la copia)
rotate (i,m,f) **rotate_copy(i,m,f,i2)**	Rota los eltos de forma que m pasa aser el 1º: [i…m…f) → [m…f-1,i…m-1] Idem a rotate pero el resultado lo copia en [i2,…)
random_shuffle (i,f) **random_shuffle (i,f,fu)**	Permuta aleatoriamente los elementos del intervalo de los iteradores [i,f). Idem, pero la función fu modifica la distribución aleatoria rand

Algoritmos de ordenación y operaciones relacionadas: (parte 1)

12. **Ordenar**: sort, stable_sort, partial_sort, partial_sort_copy, nth_element.
13. **Busqueda en una secuencia ordenada**: binary_search, lower_bound, upper_bound, equal_range.
14. **Fusionar**: merge, inplace_merge.

Algoritmos de ordenación y operaciones relacionadas: (parte 1)

sort(i,f) **sort(i,f,pred)**	Ordena los elementos de la secuencia [i,f). Usa operator< para comparar (o pred binario en versión 2).
stable_sort(i,f) **stable_sort(i,f,pred)**	Ordena los elementos de [i,f) respetando el orden original. Usa operator< para comparar (o pred binario en versión 2).
partial_sort(i,m,f) **partial_sort(i,m,f,pred)**	Ordena los m 1º elementos de la secuencia [i,f) . Usa operator< para comparar (o pred binario en versión 2).
partial_sort_copy (i,f,i2,f2) **partial_sort_copy(i,f,i2,f2,p)**	Copia en [i2,f2) los elementos de la secuencia [i,f) ordenados. El tamaño de la 2ª secuencia determina cuantos eltos ordenados de [i,f) se copian. Usa operator< para comparar (o pred binario en versión 2).
nth_element(i,n,f) **nth_element(i,n,f,pred)**	Garantiza que el n-esimo elemento esta en el lugar que le corresponderia si la secuencia estuviera ordenada. Tras su ejecución los menores a nth están delante (sin ordenar) de él y los mayores detrás (sin ordenar) Usa operator< para comparar (o pred binario en versión 2).
binary_search (i,f,x) **binary_search (i,f,x,p)**	devuelve true si un elemento de [i, f) es equivalente (!(a<b) && !(b<a)) a x. Usa operator< para comparar (o pred p en versión 2) (!(p(a,b) && !(p(b,a)).
lower_bound (i,f,x) **lower_bound (i,f,x,comp)**	devuelve un iterador al 1er elemento de [i, f) que no sea < al valor x. idem pero usando comp en vez de <.
upper_bound (i,f,x) **upper_bound(i,f,x,comp)**	devuelve un it al 1er elto de [i, f) que sea > al valor x. (while(!(x<*i)) i++) Usa operator< para comparar (o pred comp en versión 2)
equal_range(i,f,x) **equal_range(i,f,x, pred)**	Devuelve un par de iteradores a la 1ª y ultima posicion de la secuencia ordenada [i,f) en la que x debe insertarse para que la secuencia siga ordenada. (busca el 1er y ultimo elto en una secuencia ordenada que sean iguales a x) Usa operator< para comparar (o pred binario en versión 2).
merge(i,f,i2,f2,r) **merge(i,f,i2,f2,r,pred)**	Fusiona (mezcla) las secuencias ordenadas [i,f) y [i2,f2) y lo copia en [r,…) Usa operator< para comparar (o pred binario en versión 2).
inplace_merge(i,m,f) **inplace_merge(i,m,f,p)**	Mezcla 2 secuencias ordenadas [i,m) y [m,f) en el mismo contenedor [i,f) Usa operator< para comparar (o pred binario p en versión 2).

Algoritmos de ordenación y operaciones relacionadas: (parte 2)

15. **Operaciones de conjuntos (inclusion)**: includes
16. **Operaciones de conjuntos (union e intersección)**: set_union, set_intersection, set_difference.
17. **Maximo y minimo**: max_element, min_element.
18. **Comparacion lexicográfica**: lexicographical_ compare.
19. **Permutar**: next_permutation, prev_permutation.

Algoritmos de ordenación y operaciones relacionadas: (parte 2)

includes(i,f,i2,f2) **includes(i,f,i2,f2,comp)**	true si todos los eltos del 2º contenedor ordenado [i2, f2) pertenecen al 1º [i, f). usa operator< para comparar (!(a<b) && !(b<a)) idem pero usando función binaria comp (!(comp(a,b) && !(comp(b,a)).
set_union(i,f,i2,f2,r) **set_union(i,f,i2,f2,r,p)** **set_intersection** **set_difference** **set_symmetric_difference**	Union (comunes y no comunes sin repetidos) de 2 contenedores ordenados [i, f) ,[i2, f2) en [r,...) Usa operator< para comparar (o pred p binario en versión 2). Interseccion (comunes sin repes) de 2 contenedores ordenados. Usa < o pred Diferencia (eltos del 1º que no están en el 2º) de 2 ordenados. Usa < o pred Diferencia simetrica (eltos que están en uno pero no en otro) . Usa < o pred
max_element(i,f) **max_element(i,f,comp)** **min_element(i,f)** **min_element(i,f,comp)**	devuelve un iterador al elemento máximo de [i,f). usa operator< para comparar (o función binaria comp en versión 2). devuelve un iterador al elemento mínimo de [i,f). usa operator< para comparar (o función binaria comp en versión 2).
lexicographical_ compare(i,f,i2,f2) **lexicographical_ compare(i,f,i2,f2, comp)**	compara lexicograficamente dos contenedores [i, f) y [i2, f2). devuelve true cuando la 1ª secuencia es menor lexicográficamente que la 2ª idem pero usando la función binaria comp para comparar
next_permutation(i,f) **next_permutation(i,f,p)** **prev_permutation(i,f)** **prev_permutation(i,f,p)**	Si existe la siguiente permutacion lexicografica, la genera; sino false Usa operator< para comparar (o pred p binario en versión 2). Si existe la anterior permutacion lexicografica, la genera; sino false. Usa operator< para comparar (o pred p binario en versión 2).

Algoritmos numéricos generalizados:

20. **Numéricos**: accumulate, inner_product,
21. **Secuencia Numérica**: partial_sum, adjacent_difference.

Algoritmos numéricos generalizados:

accumulate (i,f,x) **accumulate (i,f,x,func)**	devuelve la suma de los valores [i,f) + x. idem pero aplicando la función binaria func en vez de la suma
inner_product(i,f,i2,x) **inner_product (i,f,i2,x,f1,f2)**	devuelve la suma de los productos (*i)*(*i2)+…+(*f)*(*f2) +x por defecto se multiplica, obteniendose el producto escalar idem pero, la función binaria f1 indica como se calcula los totales (f1 sustituye a +) y f2 como se multiplican las 2 secuencias (f2 sustituye a *).
partial_sum(i,f,i2) **partial_sum(i,f,i2,pred)** **adjacent_difference (i,f,i2) o (i,f,i2,pred)**	Dada la secuencia [i,j,k, ...) genera en [i2,…) la secuencia [i, i+j, i+j+k, ...) Usa operator+ (o pred binario en versión 2). Dada la secuencia [a,b,c, ...) genera en [i2,…) la secuencia [a,b-a, c-b, d-c,...) Usa operator- (o pred binario en versión 2).

Objetos función (están definidos en <functional>):

Un objeto función es una plantilla que sobrecarga el operador operator( ).
En STL casi todos los algoritmos permiten emplear funciones u objetos función (predicados).
Los objetos función son una abstracción de los punteros a funciones (igual que iteradores vs punteros).

En la STL existen objetos función predefinidos, que se dividen en:

- **Predicados**: realizan comparaciones u operaciones lógicas y siempre devuelven un booleano.
  La clase que lo use debe tener sobrecargado los operadores relacionales.

- **Aritméticos**: realizan distintas operaciones aritméticas y devuelven un resultado.
  La clase que lo use debe tener sobrecargado los operadores aritméticos.

- **Adaptadores de objetos función**: permiten obtener nuevos objetos función a partir de otros.

Objetos función predicados: (todos son binarios excepto el último que es unario)

equal_to<T>, not_equal_to<T>	true cuando x = = y, si no false,	true si x!=y, si no false.
greater<T>, less<T>	true cuando x > y, si no false,	true si x < y, si no false.
greater_equal<T>, less_equal<T>	true cuando x >= y, si no false,	true si x <= y, si no false.
logical_and<T>, logical_or<T>	true si x e y son true, si nofalse	true si x o y son true, si no false
logical_not<T>	devuelve !x	

La clase T debe tener sobrecargado los operadores relacionales para poder usarlo…

Objetos función aritméticos: (todos son binarios excepto el último que es unario)

multiplies<T>, divides<T>	devuelve x*y,	devuelve x/y.	(binario)
plus<T>, minus<T>	devuelve x+y,	devuelve x-y.	(binario)
modulus<T>	devuelve x%y		(binario)
negate<T>	devuelve –x.		(unario)

La clase T debe tener sobrecargado los operadores aritméticos para poder usarlo…

Objetos función adaptadores:

not1(pred1)	Toma un predicado unario y devuelve su negación.
not2(pred2)	Toma un predicado binario y devuelve su negación.
bind1st(pred,x)	devuelve un objeto función **unario** basado en pred binario, con x como 1er arg fijo.
bind2nd(pred,y)	devuelve un objeto función **unario** basado en pred binario, con 2º argumento fijo a y.
ptr_fun(f)	convierte un puntero f a función binaria o unaria en un objeto función binario o unario
mem_fun(&c::m)	convierte un método m de la clase c a un objeto función que espera un puntero a la clase c (seguido del argumento requerido para m, si lo tiene)
mem_fun_ref (&c::m)	convierte un método m de la clase c a un objeto función que espera una referencia a la clase c (seguido del argumento requerido para m, si lo tiene)

Ej:Restar 2 listas de enteros de igual longitud dejando el resultado en la primera de ellas. A continuación duplicar el valor de los elementos de ésta y por último reemplazar por 1 los elementos negativos o 0.

Solución: los algoritmos a utilizar son los siguientes:

transform (i,f,i2,op)	Aplica la operación unaria op secuencialmente a los eltos de [i,f) y copia el resultado en la secuencia [i2,…)
transform (i,f,i',i2,op)	Aplica la operación binaria op secuencialmente a los eltos de [i,f) y [i',…) y copia el resultado en la secuencia [i2,…) , es decir op(i,i'), op(++i,++i'),…
replace_if(i,f,pred,y)	Sustituye los eltos de [i,f) que hacen true el predicado unario pred por y

Al utilizar int podemos usar objetos función predicados y aritméticos ya que para int los operadores relacionales y aritméticos funcionan y no hay que sobrecargarlos.

Presentamos 3 posibles soluciones:

Con objetos función predefinidos

```
#include <iostream> , <iterator>  <list>, <algorithm>
using namespace std;
int main(int argc, char *argv[]) {
  int t[]={3,2,1,2,1};
  list<int> a(t,t+5), b;       //a: 3 2 1 2 1
  for (int i=0; i<5; ++i)   b.push_back(i%3);   //b: 0 1 2 0 1
  transform(a.begin(), a.end(),
    b.begin(), a.begin(), minus<int>());          //a: 3 1 -1 2 0
  transform(a.begin(), a.end(),
    a.begin(), bind1st(multiplies<int>(), 2));   //a: 6 2 -2 4 0
  replace_if(a.begin(), a.end(),
    bind2nd(less_equal<int>(), 0), 1);           //a: 6 2 1 4 1
  for (list<int>::iterator it=a.begin(); it!=a.end(); it++)
    cout << *it << " ";
  system("PAUSE");   return EXIT_SUCCESS;
}
```

Con punteros a función programados explicitamente

```
int restar(int a, int b) { return a-b; }          //binario
int por(int a) { return a*2; }                    //unario
bool menor_igual(int a) { return a<=0; }   //unario
int main(int argc, char *argv[]) {
  int t[]={3,2,1,2,1};
  list<int> a(t,t+5), b;       //a: 3 2 1 2 1
  for (int i=0; i<5; ++i)   b.push_back(i%3);   //b: 0 1 2 0 1
  transform(a.begin(), a.end(),  b.begin(), a.begin(), restar);
  transform(a.begin(), a.end(),
    a.begin(), por);                              //a: 6 2 -2 4 0
  replace_if(a.begin(), a.end(),
    menor_igual, 1);                              //a: 6 2 1 4 1
  for (list<int>::iterator it=a.begin(); it!=a.end(); it++)
    cout << *it << " ";
  system("PAUSE");   return EXIT_SUCCESS;
  }
```

Con punteros a funcion binarios (adaptables) que se convierten en objetos función unarios

```
#include <iostream>
#include <iterator>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;
int restar(int a, int b) { return a-b; }          //binario
int por(int a, int b) { return a*b; }             //binario
bool menor_igual(int a, int b) { return a<=b; }   //binario
int main(int argc, char *argv[]) {
  int t[]={3,2,1,2,1};
  list<int> a(t,t+5), b;       //a: 3 2 1 2 1
  for (int i=0; i<5; ++i)   b.push_back(i%3);   //b: 0 1 2 0 1
  transform(a.begin(), a.end(),
    b.begin(), a.begin(), restar);                //a: 3 1 -1 2 0
  transform(a.begin(), a.end(),
    a.begin(),bind1st(ptr_fun(por),2));          //a: 6 2 -2 4 0
  replace_if(a.begin(), a.end(),
    bind2nd(ptr_fun(menor_igual),0), 1);   //a: 6 2 1 4 1
  for (list<int>::iterator it=a.begin(); it!=a.end(); it++)
    cout << *it << " ";
  system("PAUSE");   return EXIT_SUCCESS;
}
```

En este ejemplo todas las funciones son binarias y mediante **ptr_fun** las convertimos a objetos función.

ptr_fun(por) convierte el puntero a la funcion binaria por(int a, int b) en objeto función binario.

Una vez convertido a objeto función binario podemos usar **bind1st** o **bind2nd** para fijar una de sus 2 parámetros y convertirlo en un objeto función unario.

Esta versión es más versátil que la anterior ya que las funciones binarias f( ) las podemos convertir en unarias directamente en el main( ) haciendo:

- **bind1st(ptr_fun(f),valorfijo)** o
- **bind2nd(ptr_fun(f),valorfijo)**

de modo que cambiando **valorfijo** readaptamos el código del main( ).

La mayoría de los algoritmos requieren que los objetos que están alojados en los contenedores tengan sobrecargado operator= = y operator < (en su defecto, se puede indicar una función predicado binario o unario que indique cuando son iguales y cuando menores).

Por ello lo mejor es sobrecargar dichos operadores o en su defecto definir funciones no miembros binarias o métodos estáticos que los hagan.

Si necesitamos una función unaria que haga lo mismo que la binaria podemos hacer:
- **bind1st(ptr_fun(f),valorfijo)** o **bind2nd(ptr_fun(f),valorfijo)** y fijamos el valor que nos interesa (izq o derecho)

Si tenemos un método m no estático lo podemos usar mediante **mem_fun_ref(&c::m)** o **mem_fun(&c::m)**

Ejemplo de método de clase convertido en objeto función: **mem_fun_ref(&c::m)**

```
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

class fracc {
  int n,d;
public:
  fracc(int a=0, int b=1) { n=a; d=b; }
  void ver() const { cout << n << "/" << d << " "; }
  bool operator<(fracc q) const
      { return (n*q.d<d*q.n); }
  bool operator==(fracc f) const
     { return (n*f.d==d*f.n); }
  int getn() const { return n; }
  int getd() const { return d; }
  friend bool igual(fracc a, fracc b)
     {  return (a.n==b.n && a.d==b.d);  }
  static bool mayor(fracc a, fracc b)
     {  return (a.n*b.d>a.d*b.n);  }
  friend bool pord(const fracc &a, const fracc &b)
    {  return a.d<b.d;  }
};

bool menor(fracc a, fracc b) {  return a<b; }
```

```
void ver1(const vector<fracc> &v)  {
  vector<fracc>::const_iterator it;
  for(it=v.begin(); it!=v.end(); it++)   it->ver();
}
//ver1 y ver2 hacen lo mismo…
void ver2(const vector<fracc> &v)  {
  for_each(v.begin(), v.end(),
                  mem_fun_ref(&fracc::ver));
}
```

Pantalla:

```
6/3 1/1 2/3 1/2                          6/3 1/1 2/3 1/2
1/2 2/3 1/1 6/3 2/3 2/2                  1/2 2/3 1/1 6/3 2/3 2/2
Listado ordenado:                          1/2 2/3 2/3 1/1 2/2 6/3
Listado ordenado descendente:              6/3 2/2 1/1 2/3 2/3 1/2
Listado ordenado:                          1/2 2/3 2/3 2/2 1/1 6/3
Listado ordenado por denominador:  1/1 1/2 2/2 2/3 2/3 6/3
Hay 1 > 1/1 y 3 < 1/1

Listado tras remove_if(x > 2/3):         1/2 2/3 2/2 2/3 2/3 6/3
Listado tras eliminar:                     1/2 2/3 2/3 6/3
mayor a 1/2 es 2/3
mayor a 1/2 es 2/3
mayor a 1/2 es 6/3
mayor o igual a 2/3 es 2/3
Presione una tecla para continuar . . .
```

```
int main(int argc, char *argv[]) {
  fracc t[]={fracc(6,3), fracc(2,3), fracc(1,2), fracc(1,1)};
  vector<fracc> a, b(t,t+2);       //a: 3 2 1 2 1
  vector<fracc>::iterator it;
  sort(t,t+4,fracc::mayor); //puntero a metodo estatico
  for(int i=0; i<4; i++) { t[i].ver(); } cout << endl;
  for_each(t, t+4, mem_fun_ref(&fracc::ver));
  a.resize(4); reverse_copy(t,t+4,a.begin());
  a.push_back(fracc(2,3));   a.push_back(fracc(2,2));
  cout << endl; ver1(a); cout << endl; ver2(a);
  sort(a.begin(), a.end()); //usa fracc::operator< por defecto
  cout << "\nListado ordenado: \n"; ver1(a);
  sort(a.begin(), a.end(), not2(less<fracc>())); //usa fracc::operator<
  cout << "\nListado ordenado descendente: \n"; ver2(a);
  sort(a.begin(), a.end(), mem_fun_ref(&fracc::operator<));
  cout << "\nListado ordenado: \n";   ver1(a);
  sort(a.begin(), a.end(), pord); //puntero a funcion no miembro
  cout << "\nListado ordenado por denominador: \n";   ver2(a);
  int k=count_if(a.begin(),a.end(),
          bind1st(ptr_fun(menor), fracc(1,1)));  //count_if(1/1< x)
  int j=count_if(a.begin(),a.end(),
          bind2nd(ptr_fun(menor), fracc(1,1))); //count_if(x < 1/1)
  cout << "\nHay " << k << " > 1/1 y " << j << " < 1/1\n";
  it=remove_if(a.begin(), a.end()-2,
          bind2nd(ptr_fun(fracc::mayor), fracc(2,3))); //x>2/3
  cout << "\nListado tras remove_if(x > 2/3): \n";   ver1(a);
  a.erase(it,a.end()-2);
  cout << "\nListado tras eliminar: \n";   ver1(a);
  fracc f=fracc(1,2);
  pointer_to_binary_function<fracc,fracc, bool> z=ptr_fun(menor);
  it = find_if (a.begin(), a.end(), bind1st(z,f)); //f < x    1/2< x
  while (it != a.end()) {
    cout << "\nmayor a 1/2 es " << (*it).getn() << "/" << it->getd();
    it = find_if (++it, a.end(), bind1st(z,f));
  }
  it = find_if (a.begin(), a.end(),          //!(x<2/3) → x>=2/3
                  not1(bind2nd(ptr_fun(menor),fracc(2,3))));
  cout << "\nmayor o igual a 2/3 es ";  it->ver(); cout << endl;
  system("PAUSE");   return EXIT_SUCCESS;
}
```

remove_if elimina desplazando a la izq y devuelve un it al ultimo+1
remove_if(a.begin(), a.end()-2,'3') [2 3 4 3 6 x x] → [2 4 6 3 6 x x]
Para eliminar lo gris usamos a.**erase**(**it**,a.**end**()-2);→ [2 4 6 x x]

find_if devuelve un it al elemento buscado o end() si no existe