

CONTENEDORES BÁSICOS EN LA STL

Objetivos

- Concepto de contenedor
- Conocer los principales tipos de contenedores básicos
- Resolución de problemas usando contenedores
- Iteradores
- Algoritmos

Contenedores

Un *contenedor* es una colección de objetos dotado de un conjunto de métodos para gestionarlos (acceso, eliminarlos, añadir, etc.).

Los contenedores se pueden clasificar según distintos criterios, pero la fundamental es la *forma de organización*:

- lineal
- jerárquica
- interconexión total

También se diferencia por las *formas de acceder a los componentes*:

- secuencial
- directa
- por clave (también conocidos como *asociativos*)

Contenedores (II)

- Contenedores *secuenciales*:

Organizan los datos en orden lineal: primer elemento, segundo, . . . Además, quedan identificados por la posición que ocupan en el contenedor.

Vector dinámico (vector)

Lista (list)

Pila (stack)

Cola (queue)

Cola doble (deque)

- Contenedores *directos*: garantizan acceso a cualquier componente en tiempo constante:
Vector dinámico

- Contenedores *asociativos*:

Gestionan los datos mediante claves que los identifican (por ejemplo, DNI) y permiten su recuperación eficiente a partir de ellas.

Conjunto (set)

Bolsa (multiset)

Diccionario (map)

Multidiccionario (multimap)

Set/Map no ordenados

`unordered_set`

`unordered_map` → Tablas Hash

Contenedores (III)

- **Pila (stack):**

Estructura de datos, representada como una sucesión de objetos, que limita la forma en que éstos elementos se añaden a ella, así como la manera de abandonarla. Concretamente, una pila sólo permite introducir y borrar elementos por un extremo de la secuencia, denominado *tope (top)* (Secuencias LIFO).

- **Cola (queue):** Contenedor, representado como una sucesión de objetos que limita la forma en que éstos se añaden a ella, al igual la manera de abandonarla. Así, una cola sólo permite introducir objetos por un extremo de la sucesión *final*, y la recuperación por el opuesto, *frente*. Los elementos se ubican en estricto orden de llegada. (Secuencia FIFO).

- Cola con prioridad (priority_queue):

Tipo especial de cola en la que se siguen cumpliendo las restricciones en cuanto a los lugares por donde se realiza la inserción y acceso de los elementos, pero en este caso los elementos no se disponen según su orden de llegada, sino de acuerdo a algún tipo de prioridad.

- Conjunto (set):

Colección de valores que no se repiten y que permite conocer de manera eficiente si un valor está contenido o no en el conjunto.

- Bolsa (multiset):

Análogo a un conjunto, pero permitiendo valores repetidos.

- Diccionario (map):

/unordered_map

Estructura de almacenamiento que implementa una relación “clave-valor”, por lo que permite almacenar valores identificados por claves únicas, y recuperarlos previamente mediante éstas.

- **Vector dinámico (vector):**

Generalización de un vector o array que almacena una colección de elementos del mismo tipo. Los elementos se acceden de manera directa por medio de un índice, en el rango de 0 a $n - 1$, siendo n el tamaño del vector. Dicho tamaño puede aumentarse o decrementarse según las necesidades de manera dinámica.

- **Lista (list):**

Contenedor que almacena sus elementos en forma de sucesión, permitiendo, por tanto, el acceso secuencial a ellos. Cada elemento establece quién es el siguiente de la sucesión.

Iteradores

Abstracción por iteración

- Son generalizaciones de punteros: objetos que “señalan” a otros objetos.
- Se utilizan normalmente para iterar (moverse) sobre un rango de objetos de manera independiente del tipo de contenedor donde estén incluidos.
- Permitirán leer elementos del contenedor, modificar, borrar o insertar nuevos.

En la STL existen cuatro clases de iteradores según puedan hacer el recorrido hacia delante o atrás, o leer objetos del contenedor (L), insertar otros nuevos o modificar ya existentes (E):

iterator	hacia adelante	L, E
const_iterator	hacia adelante	L
reverse_iterator	hacia atrás	L, E
const_reverse_iterator	hacia atrás	L

Iteradores - II

Con respecto a la STL, se especificarán seguidamente las siguientes operaciones comunes a los contenedores set, multiset, map, vector y list.

Para especificar los TDAs iterator y const_iterator, utilizaremos genéricamente el TDA contenedor.

Declaración de iteradores: **tipo de iterador**

```
contenedor<T>::iterator miIterador;
```

```
contenedor<T>::const_iterator miIteradorConst;
```

sobre qué contenedor (y tipo de dato) itera

Ejemplo:

```
list<double>::iterator iterListaDouble;
```

```
map<String,int>::const_iterator iterDiccConst;
```

Esp. TDA contenedor::iterator de T

```
/**  
TDA iterator::iterator, *, ++, --, ==, !=, ~iterator
```

El TDA iterator está asociado a un contenedor y señala los elementos de T contenidos en él. Permite recorrer el contenedor con posible modificación del mismo, por lo que sólo se puede usar con contenedores no constantes.
Es mutable.

```
TDA const_iterator::const_iterator, *, ++, --, ==, !=,  
~const_iterator
```

El TDA const_iterator está asociado a un contenedor y señala los elementos de T contenidos en él. Permite recorrer el contenedor pero no modificarlo por lo que se puede usar tanto con contenedores constantes como no constantes.

Es mutable.

```
*/
```

Esp. TDA contenedor::iterator de T

```
/**  
 * @brief Constructor primitivo.  
 * @doc Crea un iterador nulo.  
 */  
iterator();  
  
/**  
 * @brief Constructor de copia.  
 * @param i: iterador que se copia.  
 * @doc Crea un iterador copia de i.  
 */  
iterator(const iterator & i);  
  
/**  
 * @brief Obtener el elemento al que apunta el iterador.  
 * @pre El iterador NO está al final del recorrido.  
 * Es distinto de end().  
 * @return elemento del contenedor en la posición apuntado  
 * por el iterador.  
 */  
T & operator*() const;
```

Esp. TDA contenedor::iterator de T - II

```
/**  
 * @brief Operador de incremento.  
 * @pre El iterador NO está al final del recorrido.  
 *       Es distinto de end().  
 * @return Referencia al iterador receptor.  
 * @doc Hace que el iterador apunte a la posición siguiente  
 *      en el contenedor y lo devuelve.  
 */  
iterator & operator++();  
  
/**  
 * @brief Operador de decremento.  
 * @pre El iterador NO está al principio del recorrido.  
 *       Es distinto de begin().  
 * @return Referencia al iterador receptor.  
 * @doc Hace que el iterador apunte a la posición anterior  
 *      en el contenedor y lo devuelve.  
 */  
iterator & operator--();
```

Esp. TDA contenedor::iterator de T - III

```
/**  
 * @brief Comparación de igualdad.  
 * @param i: segundo iterador en la comparación.  
 * @return true, si el receptor e 'i' tienen el  
 * mismo valor.  
 * false, en otro caso.  
 */  
bool operator==(const iterator & i);  
  
/**  
 * @brief Comparación de desigualdad.  
 * @param i: segundo iterador en la comparación.  
 * @return true, si el receptor e 'i' tienen un valor  
 * distinto.  
 * false, en otro caso.  
 */  
bool operator!=(const iterator & i);
```

Ejemplo de uso del TDA Iterador

```
template <class T1, class T2>
void ImprimirContenedor (const T1<T2>& contenedor)
{
    contenedor::const_iterator iter;
    for (iter = contenedor.begin();
         iter != contenedor.end();
         iter++)
        cout << *iter << endl;
}
```

TDA Par (pair)

Este TDA es una utilidad que permite tratar un par de valores como una unidad.

Se utiliza en varios contenedores, en particular map y multimap, para gestionar sus elementos que son pares clave-valor.

Especificación de las operaciones:

```
/**  
 * @brief Constructor primitivo.  
 * @doc Crea un par vacío.  
 */  
pair();
```

```
/**  
 * @brief Constructor de copia.  
 * @param p: par que se copia.  
 * @doc Crea un par que es copia de p.  
 */  
pair(const pair<T1,T2> & p);
```

```
/**  
 * @brief Constructor a partir de dos valores.  
 * @param v1: primer valor.  
 * @param v2: segundo valor  
 * @doc Crea un par a partir de los dos  
 *      valores.  
 */
```

```
pair(const T1 & v1, const T2 & v2);
```

Operadores: =, ==, <

Variables Miembros: first, second.

TDA Par (pair) - II

Ejemplo de uso:

```
#include <pair>
```

```
using namespace std;
```

```
pair<int,String> miPar(3, "Hola");
```

```
cout << miPar.first << " "  
<< miPar.second << endl;
```

TDA Vector (**vector**)

Un vector es una secuencia de elementos que pueden ser accedidos mediante un índice, que indica la posición que ocupan en dicho contenedor.

Además, el vector puede crecer o decrecer dinámicamente según las necesidades. También se pueden insertar elementos en cualquier posición o borrar ya existentes.

En la STL, el TDA Vector recibe el nombre de *vector*.

Especificación del TDA Vector (vector) - I

```
/**
```

```
vector<T>
```

```
TDA vector::vector, reserve, size, capacity,  
[], push_back, begin, end, insert  
erase, clear, at, swap, ~vector.
```

Cada objeto del TDA Vector, modela un vector de elementos de la clase T.

Un vector es una secuencia dinámica de elementos identificados en ella por la posición que ocupan (índice).
El tamaño del vector puede aumentarse o disminuirse de manera dinámica.

Son objetos mutables.

Residen en memoria dinámica.

```
*/
```

Especificación del TDA Vector (vector) - II

```
/**  
 * @brief Crea un vector dinámico.  
 * @doc Crea un vector vacío.  
 */  
vector();  
  
/**  
 * @brief Crea un vector dinámico.  
 * @param n Número de elementos del vector. n>=0  
 * @doc Crea un vector de capacidad máxima n,  
 * con n valores almacenados  
 * todos nulos.  
 */  
vector(int n=0);  
  
/**  
 * @brief Constructor de copia de un rango.  
 * @param inicio: apunta al elemento inicial a copiar.  
 * @param final: apunta al elemento final a copiar.  
 * @doc Crea un nuevo conjunto que es copia del rango  
 * [inicio,final].  
 */  
vector(iterator inicio, iterator final);
```

Especificación del TDA Vector (vector) - III

```
/**  
 * @brief Modifica la capacidad máxima  
 * @param n Nueva capacidad máxima. n>=0  
 * @doc Modifica el tamaño del vector de manera que  
 *      ahora quepan en él n elementos,  
 *      conservando los elementos que ya tuviera.  
 */  
void reserve(int n);  
  
/**  
 * @brief Devuelve el número de elementos del vector  
 * @return Número de elementos  
 */  
int size() const;  
  
/**  
 * @brief Devuelve la capacidad máxima del vector  
 * @return Número máximo de elementos  
 */  
int capacity() const;
```

Especificación del TDA Vector (vector) - IV

```
/**  
 * @brief Devuelve un elemento  
 * @param i Posición del elemento a devolver. 0<=i<size()  
 * @return una referencia al elemento iésimo del vector  
 */  
T& operator[](int i);  
  
/**  
 * @brief Devuelve un elemento  
 * @param i Posición del elemento a devolver. 0<=i<size()  
 * @return una referencia constante al elemento  
 *         i-ésimo del vector  
 */  
const T& operator[](int i) const;  
  
/**  
 * @brief Añade un elemento al final del vector  
 * @param dato Elemento a añadir al final del vector  
 */  
void push_back(const T &dato);
```

Especificación del TDA Vector (vector)-VI

```
/**  
 * @brief Inserta un dato  
 * @param pos posición sobre la que se realiza la inserción  
 * @param x valor que se inserta  
 * @return Iterador indicando el dato insertado  
 */  
iterator insert(iterator pos, const T& x);  
  
/**  
 * @brief Borra un dato  
 * @param pos posición sobre la que se realiza el borrado  
 * @return Iterador indicando el dato siguiente al borrado  
 * Eficiencia: lineal.  
 */  
iterator erase(iterator pos);  
  
/**  
 * @brief Borra todos los elementos del contenedor.  
 * @doc Deja el contenedor completamente vacío.  
 */  
void clear();
```

Especificación del TDA Vector (vector)-VII

```
/**  
 * @brief Devuelve un elemento  
 * @param i Posición del elemento a devolver. 0<=i<size()  
 * @return una referencia al elemento iésimo del vector  
 * @doc Realiza comprobación de tipos si nos salimos fuera  
 *      del rango.  
 */  
 T & at (int i);  
  
/**  
 * @brief Intercambia el contenido del receptor y  
 *        del argumento.  
 * @param v: vector a intercambiar con el receptor.  
 *          ES MODIFICADO.  
 */  
 void swap (vector<T> & v);  
  
/**  
 * @brief Destructor.  
 * @post El receptor es MODIFICADO.  
 * @doc El receptor es destruido liberando todos los  
 *      recursos que usaba.  
 *      Eficiencia: lineal.  
 */  
 ~vector();
```

Otras operaciones: ==, !=, <, >, <=, >=, =.

Especificación del TDA Vector (vector) - V

```
/**  
 * @brief Devuelve el inicio del vector  
 * @return Un iterador constante con valor igual al inicio  
 */  
  
const_iterator begin() const;  
  
/**  
 * @brief Devuelve el final del vector  
 * @return Un iterador con valor igual al inicio  
 */  
const_iterator end() const;  
  
/**  
 * @brief Devuelve el inicio del vector  
 * @return Un iterador constante con valor igual al inicio  
 */  
iterator begin();  
  
/**  
 * @brief Devuelve el final del vector  
 * @return Un iterador con valor igual al inicio  
 */  
iterator end();
```

Ejemplo de uso del TDA Vector (vector)-I

```
#include <vector>
using namespace std;

vector<int> vectorEnteros;

vector<String> vectorCadenas;

vector<MiTipo> vectorMiTipo;

vector<queue<double>> vectorDeColas;
```

Ejemplo de uso del TDA vector (vector)-II

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    vector<string> sentence;

    // Se reserva espacio para 5 elementos para
    // evitar que se tenga que resituar la memoria.
    sentence.reserve(5);

    sentence.push_back("Hello,");
    sentence.push_back("how");
    sentence.push_back("are");
    sentence.push_back("you");
    sentence.push_back("?")


    cout<< "size(): " << sentence.size() << endl;
    cout<< "capacity(): " << sentence.capacity() << endl;

    // Intercambio de elementos.
```

```
swap (sentence[1], sentence[3]);  
  
// Inserción de "always" antes de "?"  
sentence.insert(find(sentence.begin(), sentence.end(),  
                     "?"),  
                     "always");  
  
// Se asigna "!" al último elemento.  
sentence.back() = "!";  
}
```

TDA Pila (stack)

Una pila es una secuencia de elementos accesibles sólo por un extremo, denominado tope, tanto para introducir nuevos elementos como para recuperarlos. Sólo se puede acceder al elemento en el tope. El resto no están accesibles directamente.

Ejemplos:

- Una “pila” de libros, en donde éstos se “apilan” unos sobre otros y sólo se puede coger el que está en la parte superior de ella y poner un nuevo libro encima del colocado en dicha parte superior.
- Una “pila” de bandejas en los comedores universitarios. Los comensales cojen la que está en la parte superior y el personal, una vez limpias, las va colocando encima de la última colocada.

En la STL, este TDA recibe el nombre de *stack*.

Especificación del TDA Pila (stack)

```
/**  
stack<T>
```

```
TDA stack::stack, empty, clear, swap, top, push  
pop, size, ~stack
```

Cada objeto del TDA pila, modela una pila de elementos de la clase T.

Una pila es una secuencia de elementos donde las inserciones y borrados tienen lugar en un extremo denominado "tope". Son contenedores del tipo LIFO (Last In, First Out).

Son objetos mutables.

Residen en memoria dinámica.

```
*/
```

Especificación del TDA Pila (stack) - II

```
/**  
 * @brief Constructor primitivo.  
 * @doc Crea una pila vacía.  
 */  
stack();  
  
/**  
 * @brief Constructor de copia.  
 * @param p: pila que se copia.  
 * @doc Crea una pila que es copia de p.  
 */  
stack(const stack<T> & p);  
  
/**  
 * @brief Informa si la pila está vacía.  
 * @return true, si la pila está vacía.  
 *          false, en otro caso.  
 */  
bool empty() const;  
  
/**  
 * @brief Borra todos los elementos del contenedor.  
 * @doc Deja el contenedor completamente vacío.  
 */  
void clear();
```

Especificación del TDA Pila (stack) - III

```
/**  
 *brief Intercambia el contenido del receptor y  
 *      del argumento.  
 *param c: pila a intercambiar con el receptor.  
 *         ES MODIFICADO.  
 *doc Este método asigna el contenido del  
 *     receptor al del parámetro y el del  
 *     parámetro al del receptor.  
 */  
void swap (stack<T> & p);  
  
/**  
 *brief Acceso al elemento en lo alto de la pila.  
 *pre El receptor no puede estar vacío: !empty().  
 *return Referencia al elemento en el tope de  
 *       la pila.  
 */  
T& top ();  
  
/**  
 *brief Acceso al elemento en el tope de la pila.  
 *pre El receptor no puede estar vacío: !empty().  
 *return Referencia constante al elemento en el  
 *       tope de la pila.  
 */  
const T& top () const;
```

Especificación del TDA Pila (stack) - III

```
/***
 *brief Deposita un elemento en la pila.
 *param elem: Elemento que se inserta.
 *doc Inserta un nuevo elemento en la pila.
 *      Dicho elemento se sitúa en el tope de
 *      la pila.
 */
void push(const T & elem);

/***
 *brief Quita un elemento de la pila.
 *pre El receptor no puede estar vacío: !empty().
 *doc Elimina el elemento en el tope de la pila.
 */
void pop();

/***
 *brief Obtiene el número de elementos.
 *return Número de elementos de la pila.
 */
size_type size() const;
```

Especificación del TDA Pila (stack) - IV

```
/**  
 *brief Destructor.  
 *post El receptor es MODIFICADO.  
 *doc El receptor es destruido liberando todos  
 *     los recursos que usaba.  
 */  
~stack();
```

Otras operaciones: ==, !=, <, >, <=, >=, =.

Ejemplos de uso del TDA Pila (stack) - I

Declaración de objetos:

```
#include <stack>
using namespace std;
```

```
stack<int> pilaEnteros;
```

```
stack<String> pilaCadenas;
```

```
stack<MiTipo> pilaMiTipo;
```

```
stack<queue<double>> pilaDeColas;
```

Ejemplos de uso del TDA Pila (stack) - II

```
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    stack<int> st;

    // Se introducen tres elementos en la pila.
    st.push(1);
    st.push(2);
    st.push(3);

    // Se sacan e imprimen dos elementos.
    cout << st.top() << ' ';
    st.pop();
    cout << st.top() << ' ';
    st.pop();

    // Se modifica el del tope.
    st.top() = 77;

    // Se introducen dos nuevos elementos.
    st.push(4);
    st.push(5);
```

```
// Se elimina el del tope sin procesarlo.  
st.pop();  
  
// Se sacan de la pila el resto de elementos.  
while (!st.empty()) {  
    cout << st.top() << ' ';  
    st.pop();  
}  
cout << endl;  
}
```

TDA Cola (queue)

Una cola es una secuencia de elementos que permite el acceso a los mismos sólo por los dos extremos: por uno se insertan nuevos objetos (final, end) y por el otro se accede a ellos (frente, front). El resto de elementos no está accesible de manera directa.

Ejemplos:

- Una cola de clientes en una caja de un supermercado esperando a ser cobrados. Los clientes acceden por el final de la cola y son atendidos por el frente. El cajero sólo atiende a aquel cliente que esté en el frente.
- Una cola de impresión que gestiona la impresión de trabajos en una impresora.
- La cola que se forma en el banco para realizar alguna operación.

En la STL, el TDA Cola recibe el nombre de *queue*.

Especificación del TDA Cola (queue)

```
/**  
 * queue<T>  
  
TDA queue::queue, empty, clear, front, push  
pop, size, swap, ~queue  
  
Cada objeto del TDA cola, modela una cola  
de elementos de la clase T.  
  
Una cola es un tipo particular de secuencia  
en la que los elementos se insertan por un  
extremo (final) y se consultan y suprimen  
por el otro (frente). Son secuencias del tipo  
FIFO (First In, First Out).  
  
Son objetos mutables.  
Residen en memoria dinámica.  
*/
```

Especificación del TDA Cola (queue) - II

```
/**  
 * @brief Constructor primitivo.  
 * @doc Crea una cola vacía  
 */  
queue();  
  
/**  
 * @brief Constructor de copia.  
 * @param c: cola que se copia.  
 * @doc Crea una cola que es copia de c.  
 */  
queue(const queue<T> & c);  
  
/**  
 * @brief Informa si la cola está vacía.  
 * @return true, si la cola está vacía.  
 *         false, en otro caso.  
 */  
bool empty() const;  
  
/**  
 * @brief Borra todos los elementos del contenedor.  
 * @doc Deja el contenedor completamente vacío.  
 */  
void clear();
```

Especificación del TDA Cola (queue) - III

```
/**  
 *brief Acceso al elemento al principio de la cola.  
 *pre El receptor no puede estar vacío.  
 *return Referencia al elemento en el frente de  
 la cola.
```

```
*/
```

```
T & front ();
```

```
/**
```

```
*brief Acceso al elemento al principio de la cola.  
*pre El receptor no puede estar vacío.  
*return Referencia constante al elemento en  
el frente de la cola.
```

```
*/
```

```
const T & front () const;
```

```
/**
```

```
*brief Añade un elemento en la cola.  
*param elem: Elemento que se inserta.  
*doc Inserta un nuevo elemento al end de la cola.
```

```
*/
```

```
void push (const T & elem);
```

Especificación del TDA Cola (queue) - IV

```
/**  
 * @brief Quita un elemento de la cola.  
 *  @pre El receptor no puede estar vacío.  
 *  @doc Elimina el elemento en el frente de la cola.  
 */  
void pop();  
  
/**  
 * @brief Obtiene el número de elementos en la cola.  
 *  @return número de elementos incluidos en la cola.  
 */  
size_type size() const;  
  
/**  
 * @brief Intercambia el contenido del receptor y  
 *        del argumento.  
 *  @param c: cola a intercambiar con el receptor.  
 *            ES MODIFICADO.  
 *  @doc Este método asigna el contenido del  
 *       receptor al del parámetro y el del  
 *       parámetro al del receptor.  
 */  
void swap (queue<T> & c);
```

Especificación del TDA Cola (queue) - V

```
/**  
 *brief Destructor.  
 *post El receptor es MODIFICADO.  
  
 El receptor es destruido liberando todos los  
 recursos que usaba.  
 */  
~queue();
```

Otras operaciones: ==, !=, <, >, <=, >=, =.

Ejemplos de uso del TDA Cola (queue) - I

Declaración de objetos:

```
#include <queue>
using namespace std;

queue<int> colaEnteros;
queue<String> colaCadenas;
queue<MiTipo> colaMiTipo;
queue<list<double>> colaDeListas;
```

Ejemplos de uso del TDA Cola (queue) - II

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
    queue<string> q;

    // Inserción de tres elementos en la cola.
    q.push("These ");
    q.push("are ");
    q.push("more than ");

    // Lectura e impresión de dos elementos.
    cout << q.front();
    q.pop();
    cout << q.front();
    q.pop();

    // Inserción de dos nuevos elementos.
    q.push("four ");
    q.push("words!");

    // Pasamos del elemento del frente.
    q.pop();
```

```
// Impresión de dos elementos.  
cout << q.front();  
q.pop();  
cout << q.front() << endl;  
q.pop();  
  
// impresión del número de elementos en la cola.  
cout << "Número de elementos: " << q.size()  
    << endl;  
}
```

```
#include <iostream>
#include <iomanip>
#include <stdio>
#include <cctype>
#include <queue>
#include <stack>

using namespace std;

int main()
{
    int c;
    stack<char> pila;
    queue<char> cola;
    cout << "Detector de palindromos\n"
        "Introduzca los caracteres. ^D para finalizar\n";
    while ((c = getchar()) != EOF)
    {
        if (!isspace(c))
        {
            pila.push(tolower(c));
            cola.push(tolower(c));
        }
    }
    while (!cola.empty())
    {
        if (cola.front() != pila.top())
        {
            cout << "La cadena NO es un palindromo\n";
            return 1;
        }
        cola.pop();
        pila.pop();
    }
    cout << "La cadena SI es un palindromo\n";
    return 0;
}
```

TDA Cola con prioridad (priority_queue)

Una cola con prioridad es una secuencia de elementos que permite el acceso a los mismos sólo por los dos extremos: por uno se insertan nuevos objetos (final, end) y por el otro se accede a ellos (frente, front). El resto de elementos no está accesible de manera directa. La principal característica es que al insertar los elementos estos se sitúan en la cola según una cierta prioridad.

Ejemplos:

- La cola de espera de unas emergencias sanitarias. Según la gravedad del enfermo que llega a ella, se atiende antes o después de otros enfermos.
- Una cola de impresión que gestiona la impresión de trabajos en una impresora. Los trabajos de ciertos usuarios se imprimirán antes que los de otros. O, aquellos que sean más cortos antes que los que sean más grandes.

En la STL, el TDA Cola con prioridad recibe el nombre de *priority_queue*.

Esp. TDA Cola con prioridad (priority_queue) - I

```
/** priority_queue<T, Contenedor, Comp>
    priority_queue::priority_queue,
    empty, clear, top, push, pop, size,
    swap, ~priority_queue.
```

Cada objeto del TDA cola con prioridad, modela una cola con prioridad de elementos de la clase T.

Una cola con prioridad es un tipo particular de secuencia en la que los elementos se insertan por un extremo (final) y se consultan y suprimen por el otro (frente). Los elementos se disponen en ella ordenados según su prioridad.

Son objetos mutables.
Residen en memoria dinámica.

*/

Esp. TDA Cola con prioridad (priority_queue) - II

```
/**  
 * @brief Constructor primitivo.  
 * @doc Crea una cola con prioridad vacía.  
 */  
priority_queue();  
  
/**  
 * @brief Constructor de copia.  
 * @param c: cola con prioridad que se copia.  
 * @doc Crea una cola con prioridad que es copia de c.  
 */  
priority_queue(const priority_queue<T> & c);  
  
/**  
 * @brief Informa si la cola con prioridad está vacía.  
 * @return true, si la cola con prioridad está vacía.  
 *         false, en otro caso.  
 */  
bool empty() const;  
  
/**  
 * @brief Borra todos los elementos del contenedor.  
 * @doc Deja el contenedor completamente vacío.  
 *      Eficiencia: lineal.  
 */  
void clear();
```

Esp. TDA Cola con prioridad (priority_queue) - III

```
/**
```

@brief Acceso al elemento al principio de la cola con prioridad.

@pre El receptor no puede estar vacío.

@return Referencia al elemento en el frente de la cola con prioridad.

```
*/
```

```
T & top();
```

```
/**
```

@brief Acceso al elemento al principio de la cola con prioridad

@pre El receptor no puede estar vacío.

@return Referencia constante al elemento en el frente de la cola con prioridad.

```
*/
```

```
const T & top() const;
```

```
/**
```

@brief Añade un elemento en la cola con prioridad.

@param elem: Elemento que se inserta.

@pre Se requiere que exista una operación ‘‘<’’.

@doc Inserta un nuevo elemento en la cola según su prioridad, quedando ordenados de mayor a menor. Eficiencia: logarítmica.

```
*/
```

```
void push (const T & elem);
```

Esp. TDA Cola con prioridad (priority_queue) - IV

```
/**  
 * @brief Quita un elemento de la cola con prioridad.  
 * @pre El receptor no puede estar vacío.  
 * @doc Elimina el elemento en el frente de la cola  
 *      con prioridad.  
 * Eficiencia: logarítmica.  
 */  
void pop();  
  
/**  
 * @brief Obtiene el número de elementos en la cola  
 *        con prioridad.  
 * @return número de elementos incluidos en la cola  
 *        con prioridad.  
 */  
size_type size() const;  
  
/**  
 * @brief Intercambia el contenido del receptor y  
 *        del argumento.  
 * @param c: cola con prioridad a intercambiar.  
 *           ES MODIFICADO.  
 * @doc Este método asigna el contenido del  
 *      receptor al del parámetro y el del  
 *      parámetro al del receptor.  
 */  
void swap (priority_queue<T> & p);
```

```
/**  
 * @brief Destructor.  
 * @post El receptor es MODIFICADO.  
  
 * El receptor es destruido liberando todos los  
 * recursos que usaba.  
 */  
priority_queue();
```

Otras operaciones: ==, !=, <, >, <=, >=, =.

Ej. de uso del TDA Cola con prioridad (priority_queue) - I

Declaración de objetos:

```
#include <queue>
```

```
using namespace std;
```

```
priority_queue<int> colaPrioridadEnteros;
```

```
priority_queue<String> colaPrioridadCadenas;
```

```
priority_queue<MiTipo> colaPrioridadMiTipo;
```

```
priority_queue<queue<double>> colaPriorDeColas;
```

Ej. de uso del TDA Cola con prioridad (priority_queue) - II

```
#include <iostream>
#include <queue>
using namespace std;

int main()
{
    priority_queue<float> q;

    // Inserción de tres elementos en la cola con
    // prioridad.

    q.push(66.6);
    q.push(22.2);
    q.push(44.4);

    // Impresión de dos elementos.
    cout << q.top() << ' ';
    q.pop();
    cout << q.top() << endl;
    q.pop();

    // Inserción de tres más.
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);
```

66.6 44.4 22.2

```
// Eliminación de uno de ellos.  
q.pop();  
  
// Sacamos e imprimimos el resto.  
while (!q.empty()) {  
    cout << q.top() << ' ';  
    q.pop();  
}  
cout << endl;  
}
```

Ej. aplicación del TDA Cola con prioridad - I

Ordenación de un vector mediante una cola con prioridad → Heapsort

```
#include <queue>

void ordenar(int *v, int n)
{
    priority_queue<int> colaPrioridad;

    int i;

    for (i=0; i< n; i++)
        colaPrioridad.push(v[i]);

    for (i=0; i< n; i++)
    {
        v[i]= colaPrioridad.top();
        colaPrioridad.pop();
    }
}
```

TDA Lista (list)

Intuitivamente: Dado un conjunto D , una lista de elementos de dicho dominio es una secuencia finita y ordenada de elementos del mismo.

Formalmente: Una lista es una aplicación de un conjunto de la forma $\{1, 2, \dots, n\}$ en un dominio D :

$$l : \{1, 2, \dots, n\} \longrightarrow D$$

Una lista se suele representar de la forma:

$$l = < a_1, a_2, \dots, a_n > \text{ con } a_i = a(i)$$

TDA Lista (list) - II

Nomenclatura de listas:

- n : longitud de la lista
- $1, 2, \dots, n$: posiciones de la lista
- a_i : es el elemento que ocupa la posición i -ésima
- a_1 : primer elemento de la lista
- a_n : último elemento de la lista
- $n + 1$: posición final (tras el último elemento)
- $\langle \rangle$: lista vacía.
- Orden inducido por la lista: a_i precede a a_{i+1} y a_i sigue a a_{i-1} .

TDA Lista (list) - III

Ejemplos:

- La lista de alumnos matriculados en un curso.
- La lista de la compra.

En la STL, el TDA Lista recibe el nombre *list*.

Especificación del TDA Lista (list) - I

```
/**
```

```
list<T>
```

```
TDA list::list, empty, clear, size, insert, erase  
begin, end, push_back, push_front,  
pop_back, pop_front, back_front, ~list
```

Cada objeto del TDA lista de T, modela listas de objetos del dominio T. Cada objeto (instancia) del TDA lista tiene vinculado un conjunto de posiciones distintas.

Son mutables.

Residen en memoria dinámica.

```
*/
```

Especificación del TDA Lista (list) - II

```
/**  
 * @brief Constructor primitivo.  
 * @doc Crea una lista vacía.  
 */  
list();  
  
/**  
 * @brief Constructor de copia.  
 * @param l: lista que se copia.  
 * @doc Crea una lista que es copia de l.  
 */  
list(const list<T> & l);  
  
/**  
 * @brief Constructor de copia de un rango.  
 * @param inicio: apunta al elemento inicial a copiar.  
 * @param final: apunta al elemento final a copiar.  
 * @doc Crea un nuevo conjunto que es copia del rango  
 * [inicio,final].  
 */  
list(iterator inicio, iterator final);
```

Especificación del TDA Lista (list) - III

```
/**  
 * @brief Informa si la lista está vacía.  
 * @return true, si la lista está vacía.  
 *          false, en otro caso.  
 */  
 bool empty() const;  
  
/**  
 * @brief Borra todos los elementos del contenedor.  
 * @doc Deja el contenedor completamente vacío.  
 *      Eficiencia: lineal.  
 */  
 void clear();  
  
/**  
 * @brief Devuelve el número de elementos de la lista.  
 * @return número de elementos de la lista.  
 */  
 int size() const;
```

Especificación del TDA Lista (list) - IV

```
/**  
 * @brief Inserta un elemento en la lista.  
 * @param p: posición delante de la que se inserta. Debe  
 *           ser una posición válida para el objeto lista  
 *           receptor.  
 * @param elemento: elemento que se inserta.  
 * @return posición del elemento insertado.  
 * @doc Inserta un elemento con el valor 'elem' en la  
 *      posición anterior a 'p'. MODIFICA al objeto  
 *      receptor.  
 */  
iterator insert(iterator p, const T & elemento);  
  
/**  
 * @brief Elimina un elemento de la lista.  
 * @param p: posición del elemento que se borra. Debe  
 *           ser una posición válida para el objeto  
 *           lista receptor.  
 * @return posición siguiente a la del elemento borrado.  
 * @doc Destruye el objeto que ocupa la posición 'p'  
 *      en el objeto lista receptor.  
 */  
iterator erase(iterator p);
```

Especificación del TDA Lista (list) - V

```
/**  
 *  @brief Obtiene la primera posición de la lista.  
 *  @return la primera posición de la lista receptora.  
 */  
iterator begin();  
  
/**  
 *  @brief Obtener la primera posición de la lista.  
 *  @return la primera posición de la lista receptora.  
 */  
const_iterator begin() const;  
  
/**  
 *  @brief Obtener la posición posterior al último  
 *  elemento de la lista.  
 *  @return la posición siguiente al último elemento de  
 *  la lista receptora.  
 */  
iterator end() const;
```

Especificación del TDA Lista (list) - VI

```
/**  
 * @brief Obtener la posición posterior al último  
 * elemento de la lista.  
 * @return la posición siguiente al último elemento de  
 * la lista receptora.  
 */  
const_iterator end() const;  
  
/**  
 * @brief Añade un elemento al final de la lista.  
 * @param dato Elemento a añadir al final de la lista.  
 */  
void push_back(const T &dato);  
  
/**  
 * @brief Añade un elemento al principio de la lista.  
 * @param dato Elemento a añadir al final de la lista.  
 */  
void push_front(const T &dato);  
  
/**  
 * @brief Elimina el elemento del final de la lista.  
 */  
void pop_back();
```

Especificación del TDA Lista (list) - VII

```
/**  
 * @brief Elimina el elemento del principio de la lista.  
 */  
void pop_front();  
  
/**  
 * @brief Devuelve el elemento al final de la lista.  
 */  
T & back();  
  
/**  
 * @brief Devuelve el elemento del principio de la lista.  
 */  
T & front();  
  
/**  
 * @brief Intercambia el contenido del receptor y  
 *        del argumento.  
 * @param l: lista a intercambiar con el receptor.  
 *          ES MODIFICADO.  
 * @doc Este método asigna el contenido del  
 *      receptor al del parámetro y el del  
 *      parámetro al del receptor.  
 */  
void swap (list<T> & l);
```

Especificación del TDA Lista (list) - VIII

```
/**  
 *brief Destructor.  
 *post El receptor es MODIFICADO.  
  
 El receptor es destruido liberando todos los  
 recursos que usaba.  
 Eficiencia: lineal.  
 */  
~list();
```

Otras operaciones: ==, !=, <, >, <=, >=, =.

Ejemplo de uso del TDA lista (list)-I

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

list<int> listaEnteros;

list<String> listaCadenas;

list<MiTipo> listaMiTipo;

list<list<double>> listaDelistas;
```

Ejemplo de uso del TDA lista (list)-I

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

int main()
{
    list<int> list1, list2;

    for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
    }

    list<int>::const_iterator iter;

    for (iter=list1.begin(); iter!= list1.end(); iter++)
        list2.push_back(*iter);

}
```

```

template <class T>
void EliminaDuplicados(list<T> & l)
/**
 * @brief Elimina los elementos duplicados de una lista.
 * @param lista a procesar. Es MODIFICADO.
 * @doc Elimina los elementos duplicados de l.
 */
{
    typename
    for (list<T>::iterator p = l.begin();
        p != l.end();
        ++p)
    {
        typename list<T>::iterator q = p;
        ++q;
        while (q != l.end())
        {
            if (*p == *q)
                q = l.erase(q);
            else
                ++q;
        }
    }
}

```

```

/* -----
template <class T>
void comunes (const list<T> & l1, const list<T> & l2,
               list<T> & lsal )
{
    typename list<T>::const_iterator i1,i2;
    bool enc = false;

    for (i1= l1.begin(); i1!=l1.end();++i1){
        enc= false;
        for (i2 = l2.begin(); i2!= l2.end() && !enc ; ++i2)
            if (*i1 == *i2)
            {
                enc = true;
                lsal.insert( lsal.end(), *i2);
            }
    }
}

/* -----
template<class T>
class par {
public:
    T elem;
    int contador;
};

```

6.4 Lists

A list manages its elements as a doubly linked list (Figure 6.4). As usual, the C++ standard library does not specify the kind of the implementation, but it follows from the list's name, constraints, and specifications.

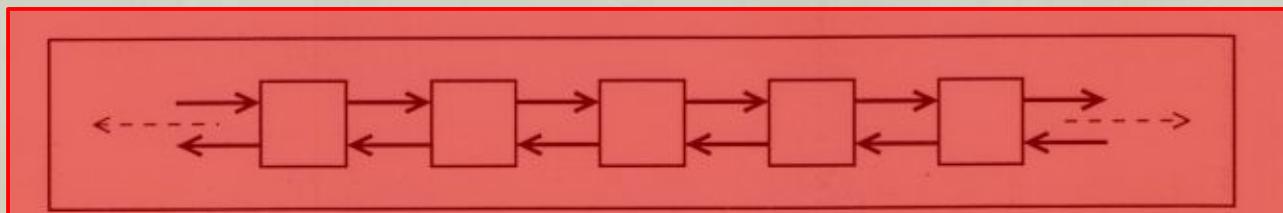


Figure 6.4. Structure of a List

To use a list you must include the header file <list>¹¹:

```
#include <list>
```

There, the type is defined as a template class inside namespace `std`:

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
    class list;
}
```

The elements of a list may have any type T that is assignable and copyable. The optional second template parameter defines the memory model (see Chapter 15). The default memory model is the model allocator, which is provided by the C++ standard library.¹²

6.4.1 Abilities of Lists

The internal structure of a list is totally different from a vector or a deque. Thus, a list differs in several major ways compared with vectors and deques:

- A list does not provide random access. For example, to access the fifth element, you must navigate the first four elements following the chain of links. Thus, accessing an arbitrary element using a list is slow.
 - Inserting and removing elements is fast at each position, and not only at one or both ends. You can always insert and delete an element in constant time because no other elements have to be moved. Internally, only some pointer values are manipulated.

¹¹ In the original STL, the header file for lists was `<list.h>`.

¹² In systems without support for default template parameters, the second argument is typically missing.

Inserting and deleting elements does not invalidate pointers, references, and iterators to other elements.

A list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can't get into an intermediate state in which only half of the operation is complete.

The member functions provided for lists reflect these differences compared with vectors and deques as follows:

Lists provide neither a subscript operator nor `at()` because no random access is provided.

Lists don't provide operations for capacity or reallocation because neither is needed. Each element has its own memory that stays valid until the element is deleted.

Lists provide many special member functions for moving elements. These member functions are faster versions of general algorithms that have the same names. They are faster because they only redirect pointers rather than copy and move the values.

4.2 List Operations

Create, Copy, and Destroy Operations

The ability to create, copy, and destroy lists is the same as it is for every sequence container. See Table 6.12 for the list operations that do this. See also Section 6.1.2, page 144, for some remarks about possible initialization sources.

Operation	Effect
<code>list<Elem> c</code>	Creates an empty list without any elements
<code>list<Elem> c1(c2)</code>	Creates a copy of another list of the same type (all elements are copied)
<code>list<Elem> c(n)</code>	Creates a list with n elements that are created by the default constructor
<code>list<Elem> c(n, elem)</code>	Creates a list initialized with n copies of element elem
<code>list<Elem> c(beg, end)</code>	Creates a list initialized with the elements of the range [beg, end)
<code>c.^list<Elem>()</code>	Destroys all elements and frees the memory

Table 6.12. Constructors and Destructor of Lists

Modifying Operations

Lists provide the usual operations for size and comparisons. See Table 6.13 for a list and Section 6.1.2, page 144, for details.

As usual, these operations do *not* check whether the container is empty. If the container is empty, calling them results in undefined behavior. Thus, the caller must ensure that the container contains at least one element. For example:

```
std::list<Elem> coll;           // empty!

std::cout << coll.front();      // RUNTIME ERROR ⇒ undefined behavior

if (!coll.empty()) {
    std::cout << coll.back();   // OK
}
```

Iterator Functions

To access all elements of a list, you must use iterators. Lists provide the usual iterator functions (Table 6.16). However, because a list has no random access, these iterators are only bidirectional. Thus, you can't call algorithms that require random access iterators. All algorithms that manipulate the order of elements a lot (especially sorting algorithms) fall under this category. However, for sorting the elements, lists provide the special member function `sort()` (see page 245).

Operation	Effect
<code>c.begin()</code>	Returns a bidirectional iterator for the first element
<code>c.end()</code>	Returns a bidirectional iterator for the position after the last element
<code>c.rbegin()</code>	Returns a reverse iterator for the first element of a reverse iteration
<code>c.rend()</code>	Returns a reverse iterator for the position after the last element of a reverse iteration

Table 6.16. Iterator Operations of Lists

Inserting and Removing Elements

Table 6.17 shows the operations provided for lists to insert and to remove elements. Lists provide all the actions of deques, supplemented by special implementations of the `remove()` and `remove_if()` algorithms.

As usual by using the STL, you must ensure that the arguments are valid. Iterators must refer to valid positions, the beginning of a range must have a position that is not behind the end, and you must not try to remove an element from an empty container.

Inserting and removing happens faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls.

For removing elements, lists provide special implementations of the `remove()` algorithms (see section 9.7.1, page 378). These member functions are faster than the `remove()` algorithms because they manipulate only internal pointers rather than the elements. So, in contrast to vectors or deques, you should call `remove()` as a member function and not as an algorithm (as mentioned on

Operation	Effect
<code>c.insert(pos, elem)</code>	Inserts at iterator position <code>pos</code> a copy of <code>elem</code> and returns the position of the new element
<code>c.insert(pos, n, elem)</code>	Inserts at iterator position <code>pos</code> <code>n</code> copies of <code>elem</code> (returns nothing)
<code>c.insert(pos, beg, end)</code>	Inserts at iterator position <code>pos</code> a copy of all elements of the range <code>[beg, end]</code> (returns nothing)
<code>c.push_back(elem)</code>	Appends a copy of <code>elem</code> at the end
<code>c.pop_back()</code>	Removes the last element (does not return it)
<code>c.push_front(elem)</code>	Inserts a copy of <code>elem</code> at the beginning
<code>c.pop_front()</code>	Removes the first element (does not return it)
<code>c.remove(val)</code>	Removes all elements with value <code>val</code>
<code>c.remove_if(op)</code>	Removes all elements for which <code>op(elem)</code> yields true
<code>c.erase(pos)</code>	Removes the element at iterator position <code>pos</code> and returns the position of the next element
<code>c.erase(beg, end)</code>	Removes all elements of the range <code>[beg, end]</code> and returns the position of the next element
<code>c.resize(num)</code>	Changes the number of elements to <code>num</code> (if <code>size()</code> grows, new elements are created by their default constructor)
<code>c.resize(num, elem)</code>	Changes the number of elements to <code>num</code> (if <code>size()</code> grows, new elements are copies of <code>elem</code>)
<code>c.clear()</code>	Removes all elements (makes the container empty)

Table 6.17. Insert and Remove Operations of Lists

page 154). To remove all elements that have a certain value, you can do the following (see Section 5.6.3, page 116, for further details):

```
std::list<Elem> coll;
...
// remove all elements with value val
coll.remove(val);
```

However, to remove only the first occurrence of a value, you must use an algorithm such as that mentioned on page 154 for vectors.

You can use `remove_if()` to define the criterion for the removal of the elements by a function or a function object.¹³ `remove_if()` removes each element for which calling the passed operation yields true. An example of the use of `remove_if()` is a statement to remove all elements that have an even value:

```
list.remove_if (not1(bind2nd(modulus<int>(),2)));
```

If you don't understand this statement, don't panic. Turn to page 306 for details. See page 378 for additional examples of `remove()` and `remove_if()`.

¹³ The `remove_if()` member function is usually not provided in systems that do not support member templates.

Splice Functions

Linked lists have the advantage that you can remove and insert elements at any position in constant time. If you move elements from one container to another, this advantage doubles in that you only need to redirect some internal pointers (Figure 6.5).

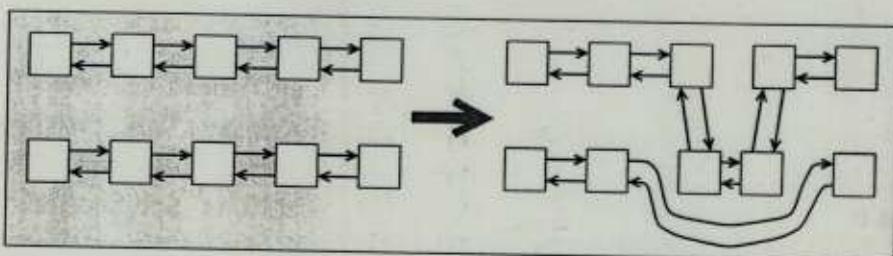


Figure 6.5. Splice Operations to Change the Order of List Elements

To support this ability, lists provide not only `remove()` but also additional modifying member functions to change the order of and relink elements and ranges. You can call these operations to move elements inside a single list or between two lists, provided the lists have the same type. Table 6.18 lists these functions. They are covered in detail in Section 6.10.8, page 244, with examples in Section 6.4.4, page 172.

Operation	Effect
<code>c.unique()</code>	Removes duplicates of consecutive elements with the same value
<code>c.unique(op)</code>	Removes duplicates of consecutive elements, for which <code>op()</code> yields true
<code>c1.splice(pos, c2)</code>	Moves all elements of <code>c2</code> to <code>c1</code> in front of the iterator position <code>pos</code>
<code>c1.splice(pos, c2, c2pos)</code>	Moves the element at <code>c2pos</code> in <code>c2</code> in front of <code>pos</code> of list <code>c1</code> (<code>c1</code> and <code>c2</code> may be identical)
<code>c1.splice(pos, c2, c2beg, c2end)</code>	Moves all elements of the range <code>[c2beg, c2end]</code> in <code>c2</code> in front of <code>pos</code> of list <code>c1</code> (<code>c1</code> and <code>c2</code> may be identical)
<code>c.sort()</code>	Sorts all elements with operator <code><</code>
<code>c.sort(op)</code>	Sorts all elements with <code>op()</code>
<code>c1.merge(c2)</code>	Assuming both containers contain the elements sorted, moves all elements of <code>c2</code> into <code>c1</code> so that all elements are merged and still sorted
<code>c1.merge(c2, op)</code>	Assuming both containers contain the elements sorted due to the sorting criterion <code>op()</code> , moves all elements of <code>c2</code> into <code>c1</code> so that all elements are merged and still sorted according to <code>op()</code>
<code>c.reverse()</code>	Reverses the order of all elements

Table 6.18. Special Modifying Operations for Lists

6.4.3 Exception Handling

Lists have the best support of exception safety of the standard containers in the STL. Almost all list operations will either succeed or have no effect. The only operations that don't give this guarantee in face of exceptions are assignment operations and the member function `sort()` (they give the usual "basic guarantee" that they will not leak resources or violate container invariants in the face of exceptions). `merge()`, `remove()`, `remove_if()`, and `unique()` give guarantees under the condition that comparing the elements (using operator == or the predicate) doesn't throw. Thus, to use a term from database programming, you could say that lists are *transaction safe*, provided you don't call assignment operations or `sort()` and ensure that comparing elements doesn't throw. Table 6.19 lists all operations that give special guarantees in face of exceptions. See Section 5.11.2, page 139, for a general discussion of exception handling in the STL.

Operation	Guarantee
<code>push_back()</code>	Either succeeds or has no effect
<code>push_front()</code>	Either succeeds or has no effect
<code>insert()</code>	Either succeeds or has no effect
<code>pop_back()</code>	Doesn't throw
<code>pop_front()</code>	Doesn't throw
<code>erase()</code>	Doesn't throw
<code>clear()</code>	Doesn't throw
<code>resize()</code>	Either succeeds or has no effect
<code>remove()</code>	Doesn't throw if comparing the elements doesn't throw
<code>remove_if()</code>	Doesn't throw if the predicate doesn't throw
<code>unique()</code>	Doesn't throw if comparing the elements doesn't throw
<code>splice()</code>	Doesn't throw
<code>merge()</code>	Either succeeds or has no effect if comparing the elements doesn't throw
<code>reverse()</code>	Doesn't throw
<code>swap()</code>	Doesn't throw

Table 6.19. List Operations with Special Guarantees in Face of Exceptions

6.4.4 Examples of Using Lists

The following example in particular shows the use of the special member functions for lists:

```
// cont/list1.cpp

#include <iostream>
#include <list>
#include <algorithm>
```

```
using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout, " "));
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout, " "));
    cout << endl << endl;
}

int main()
{
    // create two empty lists
    list<int> list1, list2;

    // fill both lists with elements
    for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
    }

    printLists(list1, list2);

    // insert all elements of list1 before the first element with value 3 of list2
    // - find() returns an iterator to the first element with value 3
    list2.splice(find(list2.begin(), list2.end(), 3),
                 list1); // destination position
    // source list

    printLists(list1, list2);

    // move first element to the end
    list2.splice(list2.end(),
                 list2,
                 list2.begin()); // destination position
    // source list
    // source position

    printLists(list1, list2);

    // sort second list, assign to list1 and remove duplicates
    list2.sort();
    list1 = list2;
```

```
list2.unique();
printLists(list1, list2);

// merge both sorted lists into the first list
list1.merge(list2);
printLists(list1, list2);
}
```

The program has the following output:

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0
```

```
list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0
```

```
list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5
```

```
list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5
```

```
list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

```

#include <iostream>
#include <list>
#include <string>
using namespace std; // disponibilidad de todos los nombres del espacio
// (componentes) de la biblioteca estándar de C++
typedef list<string> LISTSTR;

void main() {
    LISTSTR uno;           // se declaran 2 listas de cadenas (vacias)
    LISTSTR dos;
    uno.push_back("Burgos"); // se añaden dos ciudades
    uno.push_back("Alicante");
    uno.push_back("Oviedo") // a uno
    dos.push_back("Oviedo"); // y a dos
    dos.push_back("Burgos");
    uno.sort(); dos.sort();
    uno.merge(dos); // se mezclan las 2 listas en la lista uno
    cout << "La lista dos está vacia: " << dos.empty() << endl;
    LISTSTR::iterator p;
    for (p=uno.begin(); p!=uno.end(); p++) // se escriben los elementos de la lista uno
        cout << *p << " "; // elementos de la lista uno
    cout << endl;
    uno.unique(); // se ordenan
    dos.unique(); // ambos listas
    uno.unique(); // se eliminan los elementos contiguos repetidos de uno
    for (p=uno.begin(); p!=uno.end(); p++)
        cout << *p << " "; // se imprime la lista uno
    cout << endl;
}

```

Salida: La lista dos está vacia: 1

~~Alcalde Burgos Burgos Oviedo~~

Burgos Oviedo
Alicante Burgos Oviedo

#ifndef _PILA_H
#define _PILA_H

Tema 5

#include "lista.h"
template <class T>
class Pila {

private:

Lista<T> pila;

public:

Pila();

Pila(const Pila<T> & p);

bool vacia() const;

T & tope();

void poner(const T & elem);

void quitar();

~Pila();

} ;

```
template <class T>
inline Pila<T>::pila()
{ };
```

```
template <class T>
inline Pila<T>::Pila (const Pila<T> & p):
    pila(p.pila)
{ };
```

```
template <class T>
inline bool Pila<T>::valia() const
{
    return pila.valia();
}
```

```
template <class T>
inline T & Pila<T>::tope()
```

```
{  
    return pila.clemento(pila.primer());
}
```

```
template <class T>
inline void pila<T>::poner (const T & elem)
```

```
{  
    pila.insertar(pila.primer(), elem);
}
```

```
template <class T>
inline void pila<T>::quitar()
```

```
{  
    pila.borrar(pila.primer());
}
```

```
#ifndef -conjunto-h
#define -conjunto-h

#include <vector.h>
#include <assert>

template<class T>
class Conjunto {
private:
    Vector<T> v;
    int nElementos;

    bool posicion_elemento (int & pos, T e) const;

public:
    Conjunto(): nElementos(0){}

    // Conjunto (const Conjunto &);

    // ~Conjunto();

    // Conjunto & operator = (const Conjunto &);

    bool insertar (T e);

    bool borrar (T e);

    bool pertenece (T e) const {
        int pos;
        return posicion_elemento (pos, e);
    }

    bool vacio() const {return nElementos == 0; }

    int size () const {return nElementos; }
}
```

```
class const_iterador {
```

```
private:
```

```
const T* puntero;
```

```
const_iterador (const T* p) : puntero(p) {}
```

```
public:
```

```
const_iterador(): puntero(0) {}
```

```
// const_iterador (const const_iterador & v),
```

```
// ~const_iterador();
```

```
// const_iterador & operator=(const const_iterador & orig);
```

```
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 Conjunto<T>;
```

```
};
```

```
typedef const_iterador iterador;
```

```
const_iterador begin() const
    { return const_iterador (&(v[0])); }

const_iterador end() const
    { return const_iterador (&(v[n_elementos - 1]) + 1); }

};
```

#include <conjunto.cpp>

#endif /* de_conjunto.h */

```
template< class T >
class Conjunto{
private:
    list<T> l;
    ...
public:
    typedef list<T>::iterator iterator;
    ...
    iterator begin() { return l.begin(); }
    iterator end() { return l.end(); }
};
```

}

Doble cola <de que>

La doble cola contiene secuencias de elementos que cambian de tamaño de forma dinámica. El tipo doble cola <deque> se puede expandir y contraer por los 2 extremos. Es similar al vector pero con una eficiencia diferente en la inserción y borrado de elementos.

~~Sólo~~ A diferencia de los vectores las dobles colas no garantizan almacenar los elementos en posiciones contiguas de memoria

<deque>

Accesos directos

operator []

at

front

back

Modificadores

push_back

push_front

insert

erase

emplace

|| pop_back

|| pop_front

|| emplace_front || emplace_back

swap

clear

Itinerarios

begin / end

rbegin / rend

Ejemplo

```
#include <iostream>
#include <deque>
#include <vector>

int main()
{
    std::deque<int> mideque;
    for (int i = 1; i < 6; ++i)
        mideque.push_back(i); // 1 2 3 4 5

    std::deque<int>::iterator it = mideque.begin(),
    ++it;
    it = mideque.insert(it, 10); // 4 10 2 3 4 5
    mideque.insert(it, 2, 20); // 1 20 20 10 2 3
    // 4 5
    it = mideque.begin() + 2,
    std::vector<int> myvector(2, 30),
    mideque.insert(it, myvector.begin(), myvector.end());
    // 4, 20 30 30 20 10 2 3 4 5

    mideque.erase(mideque.begin(), mideque.begin() + 3);
    // eliminando los 3 primeros elementos

    std::cout << "mideque contiene:";
    for (it = mideque.begin(); it != mideque.end(); ++it)
        std::cout << " " << *it;
    std::cout << '\n';
    return 0;
```

Crear con el tipo deque una clase pila o cola que permita a un objeto actuar como una pila o una cola dependiendo de cómo ~~se~~ inicialice una bandera

//pilaowla.h

#include <deque>

#include <iostream>

using namespace std;

template <class T>

class PilaOCola {

private:

deque<T> datos;

bool isCola;

//true → actua como cola

//false → actua como pila

public:

PilaOCola (bool tipo) : isCola(tipo) {}

T & operator() () {

if (is_lola)
 \dagger datos.front(),

else
 datos.back(),

}

const T & operator() () const {

if (is_lola)
 datos.front();

else datos.back();

{

int size() const

\dagger return datos.size();

{

bool empty() const {

 return datos.size() == 0;

{

```
void Pop () {  
    if (is_lolas)  
        datos.pop_front();  
    else  
        datos.pop_back();  
}
```

```
void push (const T & v) {  
    datos.push_back(v);  
}
```

Hemos implementado la función Frente cuando activa como cola o Tope cuando activa como Pila, usando el operador () .

Ejemplo de uso de la clase Pilas y Colas

```
#include "PilaCola.h"
```

```
int main()
```

```
PilaCola<int> pila(false);  
PilaCola<int> cola(true);
```

```
for (int i = 0; i < 100; i++)
```

```
{  
    pila.Push(i);  
    cola.Push(i);  
}
```

```
std::cout << "Los elementos de la pila son:"
```

```
while (!pila.empty())  
{  
    std::cout << pila() << ' ';  
    pila.Pop();  
}
```

```
std::cout << std::endl;
```

```
std::cout << "Los elementos de la cola son:";
```

```
while (!cola.empty())  
{  
    std::cout << cola() << ' ';  
    cola.Pop();  
}
```

```
}
```