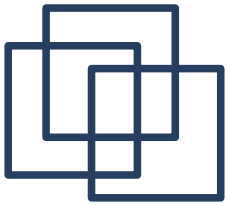
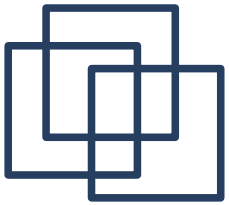
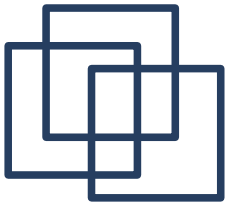


TDAs Contenedores Básicos



Objetivos

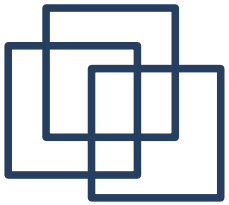
- Conocer el concepto de contenedor
- Conocer los principales tipos de contenedores básicos
- Resolver problemas usando contenedores
- Entender el concepto de iterador y ser capaz de usarlos



Contenedores

Contenedor: “Embalaje metálico grande y recuperable, de tipos y dimensiones normalizados internacionalmente y con dispositivos para facilitar su manejo.” (Diccionario R. A. E.)

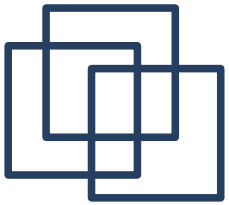
Contenedor: *TDA que puede contener un número arbitrario de otros TDAs.*



Contenedores

Un contenedor es una colección de objetos dotado de un conjunto de métodos para gestionarlos (acceder a ellos, eliminarlos, añadir nuevos objetos, buscar, etc.).

- Ofrecen también “herramientas” para recorrerlos (iterar) y revisar los objetos almacenados en ellos.
- Los contenedores se pueden **clasificar** según distintos criterios, pero el fundamental es la **forma de organización**:
  - *Lineal, jerárquica, interconexión total*

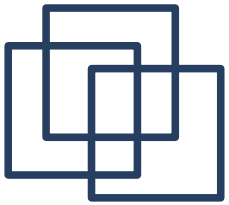


Contenedores

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

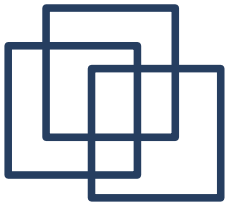
- secuencial,
- directa,
- por clave (también conocidos como asociativos)

Existe un conjunto de operaciones comunes a todos los contenedores, y operaciones adicionales atendiendo a peculiaridades particulares.



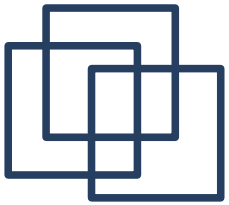
Biblioteca STL

- C++ lleva asociada la biblioteca Standard Template Library (*STL*).
- Es una biblioteca con implementaciones genéricas de los principales tipos contenedores.
- Incluye elementos para Programación Genérica.
- Tipos iteradores.



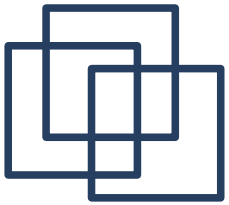
Biblioteca STL

- Algunos contenedores:
  - Listas `list`, Pilas `stack`, Colas `queue`
  - Vectores `vector`
  - Cola con prioridad `priority_queue`
  - Conjuntos `set`, Bolsas `multiset`, Diccionarios `map`



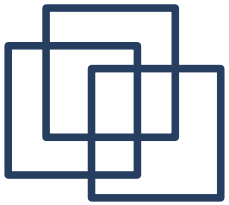
Tipos de datos en los contenedores en la STL

Miembro	Descripción
`value_type`	El tipo de los objetos almacenados en el contenedor
`reference`	Una referencia al tipo almacenado en el contenedor
`const_reference`	Referencia no modificable al tipo almacenado en el cont
`iterator`	Iterador para el contenedor
`const_iterator`	Iterador no modificable para el contenedor
`difference_type`	Tipo integral con signo que representa la diferencia entr contenedores



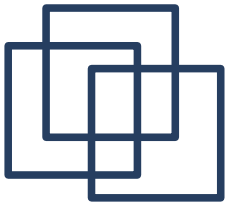
Operaciones de los contenedores en la STL

Miembro	Descripción
`C x`	Crea un contenedor vacío `x`
`C()`	Crea un contenedor vacío
`C(y)`	Crea un contenedor como copia de `y`
`C x(y)`	Crea un contenedor `x` como copia de `y`
`C x = y`	Crea un contenedor `x` y le asigna el valor de `y`
`~C()`	Destructor del contenedor. Ejecuta el destructor de cada miembro del contenedor



Operaciones de los contenedores en la STL

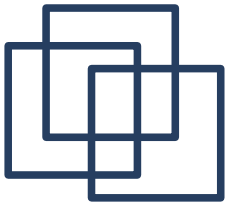
Miembro	Descripción
`x.begin()`	Devuelve un iterador que referencia al pr elemento del contenedor
`x.end()`	Devuelve un iterador que referencia a un posición detrás del último elemento del contenedor
`x.size()`	Devuelve el número actual de elementos contenedor
`x.max_size()`	Devuelve el número máximo de element contenedor puede almacenar
`x.empty()`	Devuelve `true` si el contenedor no cont elementos
`X == y, x != y, x < y, x > y, x <= y, x >= y`	Devuelve `true` en función de que se cu relaciones de orden lexicográfico (o de i dadas en los correspondientes contenedo
`x.swap(y)`	Intercambia los contenidos de los conten



Adaptadores de los contenedores de la STL

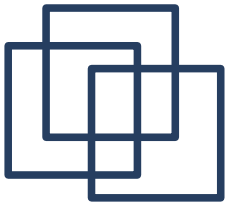
***Contenedor**: TDA que puede contener un número arbitrario de otros TDAs.*

Adaptador: un adaptador es un contenedor que *contiene* otro contenedor, permitiendo cambiar la interfaz del segundo (es decir, con una funcionalidad distinta)



Pila: descripción

- Una **pila** es un *contenedor secuencial que restringe el acceso, tanto para recuperar elementos existentes, como para insertar nuevos, a un extremo de la secuencia*.
- Al extremo donde se llevan a cabo las inserciones como las recuperaciones se denomina “tope de la pila”.
- Los elementos que se insertan se van “apilando” unos encima de otros (el último insertado se dispone encima del penúltimo, y así sucesivamente).



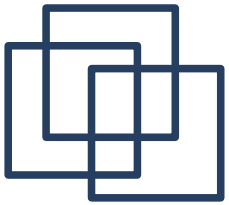
Pila: descripción y creación

- El último elemento en ser introducido en ella será el primero en ser recuperado.
- Secuencia LIFO.

Para crear una pila:

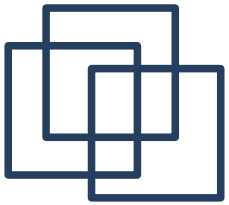
```
#include <stack>
using namespace std;
stack<T> x;
stack<T,Sequence> y;
```

donde `T` es un tipo y `Sequence` es el contenedor que almacena los datos (por defecto, `deque`)



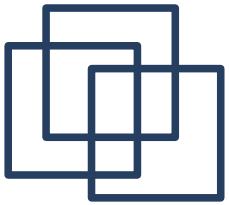
Operaciones del tipo de dato pila *(stack)* en la STL

Miembro	Descripción
`stack<T> x()`	Crea una pila vacía `x` para contener datos de tipo `T`
`stack<T>()`	Crea una pila vacía para contener datos de tipo `T`
`stack<T>(y)`	Crea una pila como copia de `y` para contener datos de tipo `T`
`stack<T> x(y)`	Crea una pila `x` como copia de `y` para contener datos de tipo `T`
`x = y`	Asigna a la pila `x` una copia del contenido de `y`



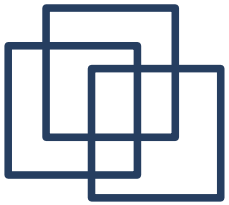
Operaciones del tipo de dato pila *(stack)* en la STL

Miembro	Descripción
`push(v)`	Empuja una copia del elemento `v` en la pila
`pop()`	Elimina el datos en el tope de la pila (el último en entrar)
`top()`	Devuelve una referencia al dato en el tope de la pila (el último en entrar)
`empty()`	Devuelve un `bool` que evalúa a `true` si la pila no contiene elementos
`size()`	Devuelve el número de datos contenidos en la pila
`operator<(stack<T> b)`	Compara dos pilas según una relación de orden dato a dato.
`operator==(stack<T> b)`	Compara dos pilas según una relación de igualdad dato a dato.



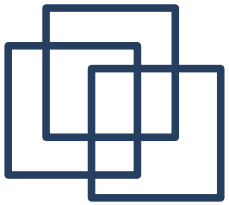
Ejemplos de uso del TDA Pila (stack)

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

stack<int> pilaEnteros;
stack<string> pilaCadenas;
stack<MiTipo> pilaMiTipo;
stack<queue<double> > pilaDeColas;
```



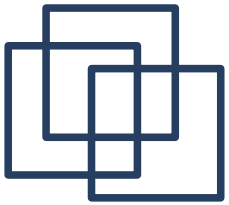
Ejemplos de uso del TDA Pila (stack)

```
#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();
```



Ejemplos de uso del TDA Pila (stack)

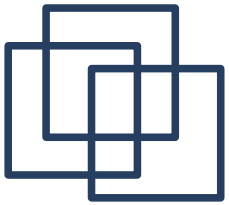
```
// Se introducen dos nuevos elementos.
    st.push(4);
    st.push(5);

    // 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();
    }
}
```



Ejemplos de uso del TDA Pila (stack)

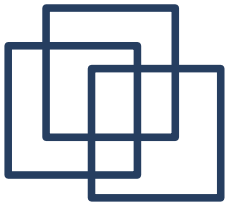
Evaluación de una expresión en postfijo.

```
/**
 @param e: expresion en postfijo (cada elemento  del vector se corresponde con un elemento de la expresión).
 @return  el resultado de evaluar la expresión.
*/

int evalua(string e)
{
```



Ejemplos de uso del TDA Pila (stack)

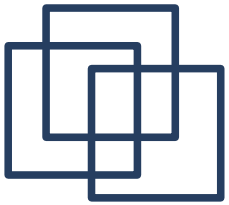
```
for (i=0; i!=e.size(); ++i) {
   // recorrido de los caracteres del string.
        if ((e[i]>='0') && (e[i]<='9'))
            v.push((int)e[i]- (int)'0');
        else  {
          dcha = v.top(); v.pop();
          izq = v.top(); v.pop();
          switch (e[i]){
           case '+': v.push(izq+dcha); break;
           case '-':  v.push(dcha-izq); break;
           case '*':  v.push(izq*dcha); break;
          }
 }
```



Ejemplos de uso del TDA Pila (stack)

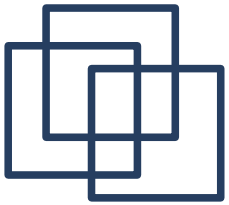
Conversión de un entero a una base.

```
/**
 @param num Número a convertir.
 @param b Base a la que convertir el número.
 @return  Cadena conteniendo el número en la base
correspondiente.
*/
string multibaseOutput(int num, int b)
{
    // digitChar[digit] es el carácter que representa
    // el dígito, 0 <= digit <= 15
    string digitChar = "0123456789ABCDEF", numStr = "";
```



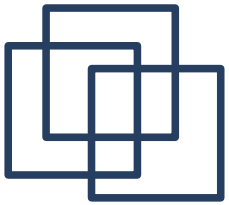
Ejemplos de uso del TDA Pila (stack)

```
// Pila que almacenará los dígitos de num en base b.
stack<char> stk;

// Extraer dígitos  base b, de derecha a izquierda,
// e introducirlos en la pila.
do     {
   // Meter el dígito más a la derecha en la pila.
   stk.push(digitChar[num % b]);
   num /= b;                  // Eliminar dicho dígito.
} while (num != 0);      // Continuar hasta que se procesen todos.

while (!stk.empty()) {        // Sacar los caracteres de la pila
   numStr += stk.top();// Añadir el dígito del tipo a  numStr
   stk.pop();                 // Eliminar dicho dígito del tope.
}
return numStr;
}
```



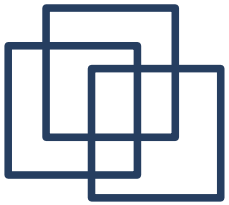
Especificación del tipo de dato pila *(stack)* en la STL

https://decsai.ugr.es/mgsilvente/stl/stack.html



Cola: descripción

- Una **cola** es un *contenedor secuencial que permite el acceso a los elementos que almacena por los dos extremos*:
  - Por el frente (*front*), para recuperar elementos.
  - Por la cola (*back*), para insertarlos.
- Tiene una interfaz muy parecida a la del tipo p*ila*.
- Es una secuencia del tipo FIFO.



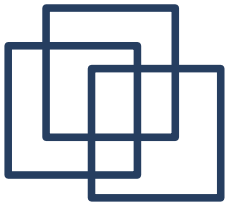
Cola: descripción y declaración

- Se utilizan en aplicaciones donde sea necesario recuperar objetos en su orden de ocurrencia.

Para crear una cola:

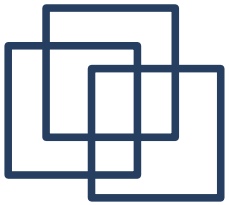
```
#include <queue>
using namespace std;
queue<T> x;
queue<T,Sequence> y;
```

donde `T` es un tipo y `Sequence` es el contenedor que almacena los datos (por defecto, `deque`)



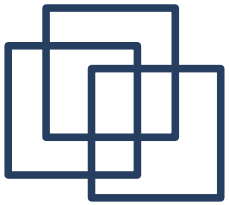
Operaciones del tipo de dato cola *(queue)* en la STL

Miembro	Descripción
`queue<T> x()`	Crea una cola vacía `x` para contener datos d `T`
`queue<T>()`	Crea una cola vacía para contener datos de ti
`queue<T>(y)`	Crea una cola como copia de `y` para conte datos de tipo `T`
`queue<T> x(y)`	Crea una cola `x` como copia de `y` datos de tipo `T`
`x = y`	Asigna a la cola `x` una copia del contenido d



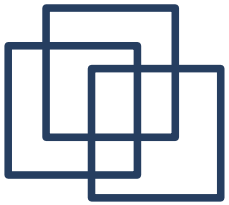
Operaciones del tipo de dato cola *(queue)* en la STL

Miembro	Descripción
`push(v)`	Empuja una copia del elemento v al final de la cola
`pop()`	Elimina el datos del frente de la cola (el primero en entrar)
`front()`	Devuelve una referencia al dato en el frente de la cola (el primero en entrar)
`back()`	Devuelve una referencia al dato en el final de la cola (el último en entrar)



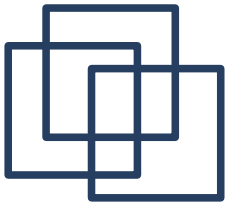
Operaciones del tipo de dato cola *(queue)* en la STL

Miembro	Descripción
`empty()`	Devuelve un `bool` que evalúa a `true` si la pila no contiene elementos
`size()`	Devuelve el número de datos contenidos en la pila
`operator<(const stack<T> &b)`	Compara dos pilas según una relación de orden dato a dato.
`operator==(const stack<T> &b)`	Compara dos pilas según una relación de igualdad dato a dato.



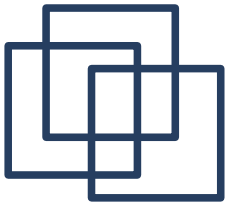
Ejemplos de uso del TDA Cola

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

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



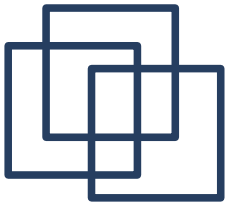
Ejemplos de uso del TDA Cola

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

int main()
{
   queue<string> q;

   // Inserción de tres elmentos en la cola.
   q.push("Tenemos ");   q.push("más");  q.push("de");
```



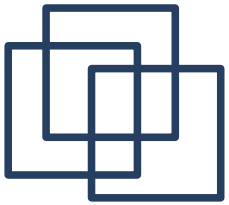
Ejemplos de uso del TDA Cola

```
    // Impresión de dos elementos.
    cout << q.front();     q.pop();
    cout << q.front();     q.pop();

    // Inserción de dos nuevos elementos.
    q.push("cuatro");
    q.push("elementos");

    // 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;
}
```



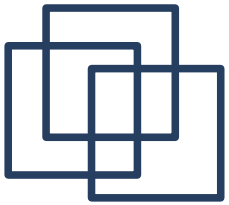
Especificación del tipo de dato cola *(queue)* en la STL

https://decsai.ugr.es/mgsilvente/stl/queue.html



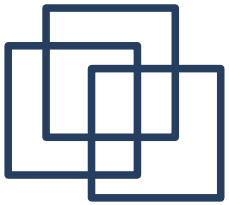
Cola con prioridad: descripción

- Una **cola con prioridad** es un contenedor en el que el elemento disponible es **el mayor** de todos los que se encuentran almacenados.
- Para poder definir una cola con prioridad sobre un tipo es necesario que éste tenga definida una operación de comparación `comp` sobre los elementos del tipo.



Cola con prioridad: otras consideraciones

- La clase con que se instancia la cola con prioridad debe de tener definido el operador <.



Cola con prioridad: declaración

Para crear una cola con prioridad:

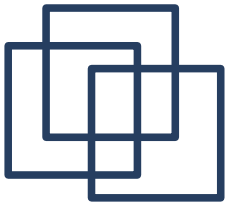
```
#include <queue>
priority_queue<T> x;
priority_queue<T,Sequence,Compare> y;
```

donde `T` es un tipo, `Sequence` es el contenedor que almacena los datos (por defecto, `vector`) y `Compare` es el functor de comparación (por defecto `less`).

```
priority_queue<int> ce_m;
```

es equivalente a:

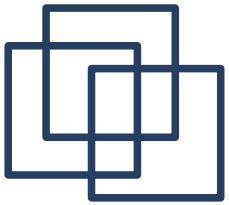
```
priority_queue<int,vector<int>,less<int> > ce_m;
```



Cola con prioridad: Ejemplo de uso

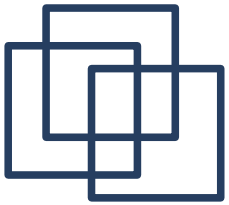
```
int main() {
    priority_queue<int> ce;


    for (int i=0;i<10;i++)
      ce.push(i);
    while (!ce.empty())
    {
        cout << " " << ce.top(); ce.pop();
    }
    cout << endl;
}
```

Salida: 9 8 7 6 5 4 3 2 1 0



Cola con prioridad: otras consideraciones

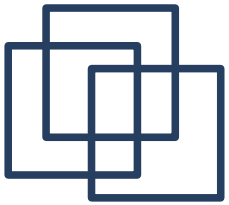
- Podemos modificar el comportamiento de la cola con prioridad para permitir que los elementos se ordenen con otro criterio
- Para ello, es necesario indicarle:
  - Contenedor sobre los que se almacenarán los elementos (vector o deque)
  - Criterio de comparación



Cola con prioridad: Usar la función de orden >

```
priority_queue<int, vector<int>, greater<int> >
    ce_m;
// compara con operador mayor
for (i=0;i<10;i++)
    ce_m.push(i);
while (!ce_m.empty())
{
    cout << " " << ce_m.top(); ce_m.pop();
}
cout << endl;
```

Salida: 0 1 2 3 4 5 6 7 8 9



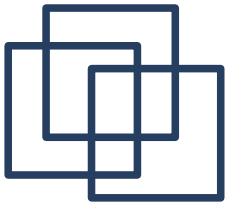
Colas con prioridad con tipos definidos

Podemos definir los operadores menor(<) y mayor(>)

```
class MiTDA{
public:
    bool operator<(const MiTDA & a) const
        { return x < a.x;}
    bool operator>(const MiTDA & a) const
        {return x > a.x;}
private: int x;
};
```

e instanciar la cola con prioridad como

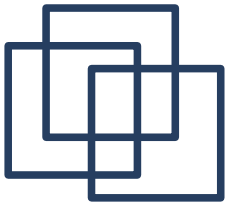
```
priority_queue<MiTDA, vector<MiTDA>,less<MiTDA> >
cp1;
priority_queue<MiTDA,
vector<MiTDA>,greater<MiTDA> > cp2;
```



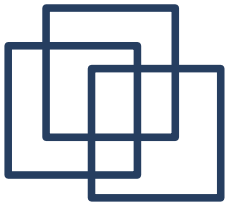
Colas con prioridad con tipos definidos

e instanciar la cola con prioridad como

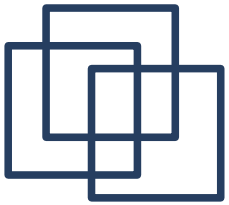
```
priority_queue<MiTDA, vector<MiTDA>,less<MiTDA> >
cp1;
priority_queue<MiTDA, vector<MiTDA>,greater<MiTDA>
> cp2;
```



```
class comp {
 public:
 bool operator() (int a, int b)
 {
  // los pares son "menores" que los impares y entre ellos por orden
  if (a%2== 0 && b%2==0) return a<b;
  else
    if (a%2==0) // b debe ser impar
       return true;
    else if (b%2==0) // a debe ser impar
        return false;

            else  return a<b;
 }
};
```

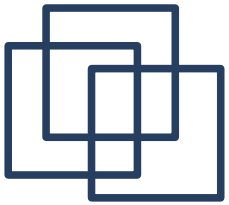


```
int main()
{
  priority_queue<int, vector<int>, comp> pq;
  for (int i= 0; i<20; i++)
    pq.push(i);
  while (!pq.empty()) {
    cout << pq.top() << " ";
    pq.pop();
  }
  cout << endl;
}
```



Cola con prioridad: especificación en la STL

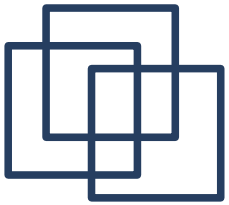
https://decsai.ugr.es/mgsilvente/stl/priority_queue.html



Contenedores asociativos en la STL

***Contenedor**: TDA que puede contener un número arbitrario de otros TDAs.*

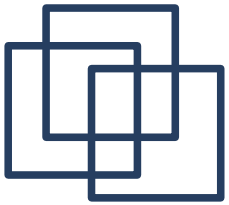
- Almacenan datos que pueden ser recuperados de forma rápida usando un identificador llamado *clave* o *llave*.
- No nos importa cómo almacena los datos sino las capacidades que tiene.
- Los valores de las claves están ordenados de forma no descendente (modificable).



El tipo *Par* (pair)

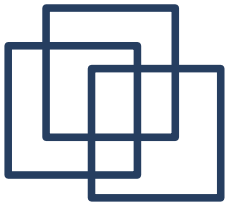
Este TDA es una utilidad que permite tratar un par de valores como una unidad.

Se utiliza en varios contenedores, en particular map y multimap, para gestionar sus elementos que son pares clave-valor, pero también podemos encontrarlos en set y multiset.



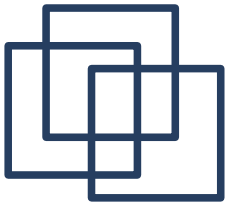
Operaciones del tipo p*air*

Miembro	Descripción
`pair<T1,T2> pair()`	Crea un par vacío
`pair(const pair<T1,T2> & p)`	Crea un par que es copia de `p`
`pair(const T1 & v1, const T2 & v2)`	Constructor primitivo
`first`	Dato de tipo `T1`
`second`	Dato de tipo `T2`
`x = y`	Asigna al par `x` una copia del contenido de `y`
`x == y`	Devuelve `true` si ambos pares son iguales en ambas coordenadas; `false`, en otro caso
`x < y`	Devuelve `true` si ambas coordenadas de `x` son menores que las de `y`; `false`, en otro caso



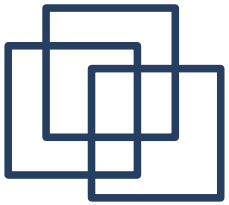
Ejemplos de uso del TDA pair

```
using namespace std;

pair<int,string> miPar(3, “Hola”);

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



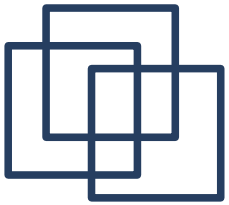
Contenedores asociativos en la STL: tipos

- **set<T> (conjunto)** sólo almacena valores de clave y no permite que haya valores repetidos.
- **multiset<T> (bolsa)** sólo almacena valores de clave y permite la existencia de valores repetidos.
- **map<key, T> (diccionario)** almacena pares (clave, dato) pero no permite valores de clave repetidos.
- **multimap<key, T>** almacena pares (clave,dato) y permite valores repetidos en la clave.



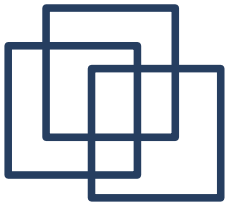
Operaciones de los contenedores asociativos

Miembro	Descripción
`value_type`	El tipo de los objetos almacenados en el conjunto
`key_type`	El tipo del valor llave del conjunto
`key_compare`	La función de comparación para un par de valores de llave cuando se ordena
`value_compare`	La función de comparación para un par de valores del conjunto cuando se ordena



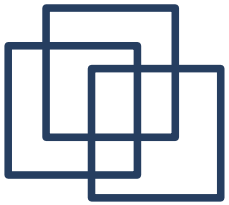
Operaciones de los contenedores asociativos

Miembro	Descripción
`C()`	Crea un contenedor vacío
`C(comp)`	Crea un contenedor vacío usando `comp` para realizar la comparación entre llaves
`C(iter1, iter2)`	Crea un contenedor con una copia de un rango
`C(iter1, iter2, comp)`	Crea un contenedor a partir de una copia, usando `comp` como comparador de llaves
`C(a_set1)`	Constructor de copia



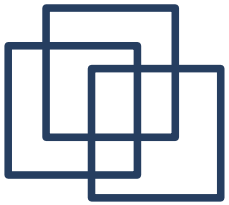
Operaciones de los contenedores asociativos

Miembro	Descripción
`key_comp()`	Devuelve el objeto para comparar dos valores de llave.
`value_comp()`	Devuelve el objeto que compara dos valores de dato.
`insert(val)`	Inserta `val` en el contenedor. Devuelve un par `pair<iter,bool>` en el que la segunda componente informa de si se ha insertado y la primera es una referencia al elemento insertado.
`insert(iter,val)`	Funciona igual que el anterior pero `iter` sugiere dónde insertar el valor `val`.
`insert(iter1,iter2)`	Inserta los valores entre `iter1` e `iter2`.



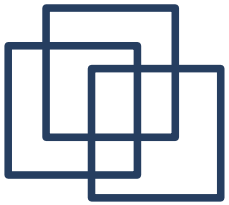
Operaciones de los contenedores asociativos

Miembro	Descripción
`erase(iter)`	Elimina el elemento indicado por `iter`
`erase(key)`	Elimina el elemento de valor de llave `key`
`erase(iter1, iter2)`	Borra todos los elementos desde `iter1` hasta `iter2`
`clear()`	Borra todos los elementos
`find(key)`	Devuelve un iterador con una referencia a una elemento de valor de clave `key` o `end()` si no existe elemento con ese valor de clave
`count(key)`	Cuenta el número de elementos con valor de clave `key`
`lower_bound(key)`	Devuelve un iterador que referencia al primer elemento cuya clave no es menor que `key`
`upper_bound(key)`	Devuelve un iterador que referencia al primer elemento cuya clave es mayor que `key`
`equal_range(key)`	Devuelve un par de iteradores que delimitan el rango de elementos cuyo valor de clave es `key`



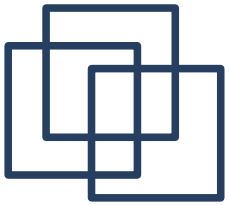
Descripción STL de los contenedores asociativos

https://decsai.ugr.es/mgsilvente/stl/AssociativeContainer.html



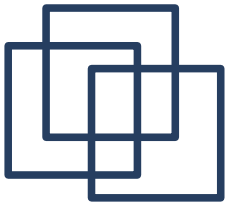
Conjunto: descripción

- Un conjunto es un contenedor asociativo de **valores únicos**, llamados claves o miembros del conjunto, que se almacenan de manera **ordenada, no descendente**.
- El tipo *Conjunto* está basado en el concepto matemático de conjunto.



Conjunto: motivación

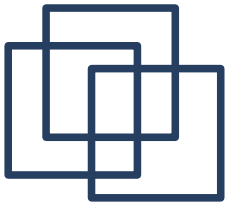
- Un corrector ortográfico de un procesador de textos puede utilizar un conjunto para almacenar todas las palabras que considera correctas ortográficamente hablando.



Conjunto: especificación y uso

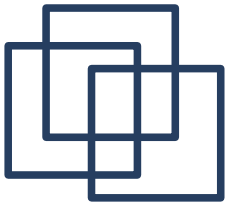
- En la STL, recibe el nombre de *set*.
- Está especialmente pensado para realizar operaciones de inserción, borrado y pertenencia de un elemento a un conjunto.

https://decsai.ugr.es/mgsilvente/stl/set.html



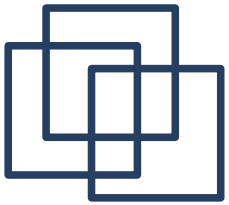
Ejemplos de uso de set

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

/* class Person
 */
class Person {
  private:
    string fn;    // first name
    string ln;    // last name
```



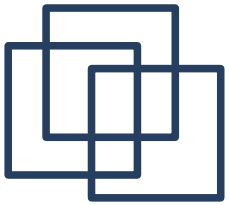
Ejemplos de uso de set

```
  public:
    Person() {};
    Person(const string& f, const string& n)
     : fn(f), ln(n) {};
    string firstname() const;
    string lastname() const;
    // ...
};
inline string Person::firstname() const {  return fn; };
inline string Person::lastname() const {  return ln; };
ostream& operator<< (ostream& s, const Person& p)
{
    s << "[" << p.firstname() << " " << p.lastname() << "]";
    return s;
}
```



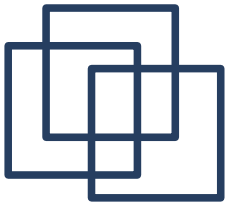
Ejemplos de uso de set

```
/* class for function predicate
 * - operator () devuelve si una persona está antes que otra
 */
class PersonSortCriterion {
  public:
    bool operator() (const Person& p1, const Person& p2)
            const {
       /* una persona antes que otra si
        * - los apellidos están antes
        * - si los apellidos son iguales pero el nombre está antes
        */
        return p1.lastname()<p2.lastname() ||
              (p1.lastname()==p2.lastname() &&
               p1.firstname()<p2.firstname());
    }
};
```



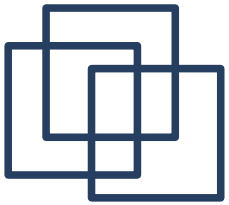
Ejemplos de uso de set

```
int main()
{
    Person p1("nicolai","josuttis");
    Person p2("ulli","josuttis");
    Person p3("anica","josuttis");
    Person p4("lucas","josuttis");
    Person p5("lucas","otto");
    Person p6("lucas","arm");
    Person p7("anica","holle");

    // declara el set con un criterio de ordenación especial
    typedef set<Person,PersonSortCriterion> PersonSet;
```



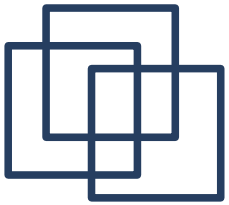
Ejemplos de uso de set

```
    // crear la colección
    PersonSet coll;
    coll.insert(p1);
    coll.insert(p2);
    coll.insert(p3);
    coll.insert(p4);
    coll.insert(p5);
    coll.insert(p6);
    coll.insert(p7);

    // sacar por pantalla los elementos
    cout << "set:" << endl;
    PersonSet::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << endl;
    }
}
```



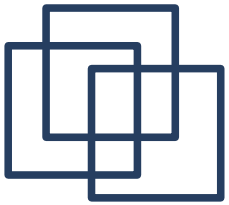
Ejemplos de uso de set

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
   typedef set<int,greater<int> > IntSet;
   IntSet coll1;       // Conjunto vacío de enteros.

   // Inserción de elementos ``aleatoriamente''.

   coll1.insert(4);
   coll1.insert(3);
   coll1.insert(5);
   coll1.insert(1);
   coll1.insert(6);
   coll1.insert(2);
   coll1.insert(5);
```



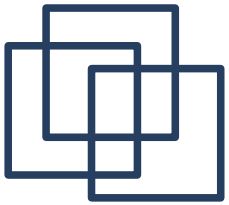
Ejemplos de uso de set

```
// Iteramos sobre todos los elementos.
IntSet::iterator pos;
for (pos = coll1.begin(); pos != coll1.end(); ++pos)
    cout << *pos << ' ';

// Insertamos el 4 otra vez y procesamos el valor
// que se devuelve.

pair<IntSet::iterator,bool> status = coll1.insert(4);

if (status.second)
   cout << “Insertado.” <<
               distance(coll1.begin(),status.first) + 1
      << endl;
else
   cout << "4 ya existe." << endl;
```



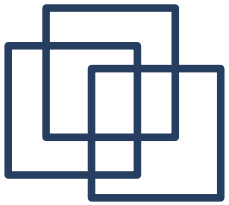
Ejemplos de uso de set

```
    // Creación de otro conjunto y asignación de los
    // elementos del primero de ellos.

    set<int> coll2(coll1.begin(),coll1.end());

    // Borrado de los elementos desde el principio hasta
    // el tres (siempre que el 3 esté en el conjunto!!)

    coll2.erase (coll2.begin(), coll2.find(3));

    // Eliminación del elemento 5.

    int num = coll2.erase (5);

    cout << num << " elemento(s) eliminados." << endl;
}
```



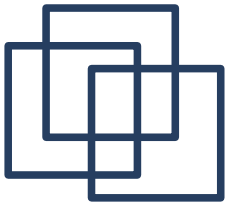
Ejemplos de uso de set

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

int main ()
{
 set<int> c;

 c.insert(1);
 c.insert(2);
 c.insert(4);
 c.insert(5);
 c.insert(6);
 cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
 cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
}
```



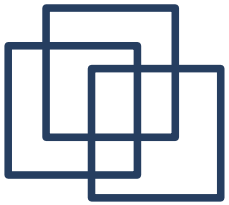
Bolsa: motivación y descripción

- Una bolsa almacena valores de clave que pueden estar repetidos (múltiples ocurrencias). También los guarda de manera **ordenada, no descendente.**
- En la STL, recibe el nombre de *multiset.*
- Está especialmente pensado para realizar operaciones de inserción, borrado y pertenencia de un elemento a una bolsa.



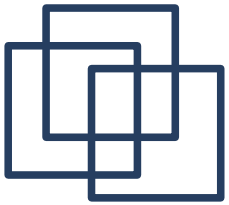
Ejemplos de uso de *multiset*

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

multiset<int> bolsaEnteros;
multiset<char, greater(char)> bolsaCaracteres;
multiset<long int , less(long int)> bolsaLong;

struct ltstr {
  bool operator()(const char* s1, const char* s2) const
  { return strcmp(s1, s2) < 0; }
};
multiset<char *, ltstr> bolsaCharAst;
```



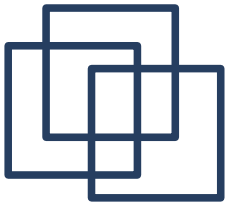
Ejemplos de uso de *multiset*

```
int main()
{
   typedef multiset<int,greater<int> > IntSet;
   IntSet coll1;

   coll1.insert(4);  coll1.insert(3);  coll1.insert(5);
   coll1.insert(1);  coll1.insert(6);  coll1.insert(2);
   coll1.insert(5);

   IntSet::iterator pos;

   for (pos = coll1.begin(); pos != coll1.end();++pos)
      cout << *pos << ' ';

   cout << endl;
```



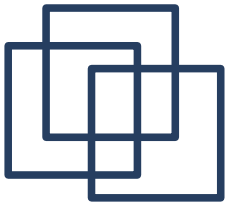
Ejemplos de uso de *multiset*

```
    // Se inserta 4 de nuevo, procesando el valor devuelto
    IntSet::iterator ipos = coll1.insert(4);
    cout << "4 se inserta como elemento "
         << distance(coll1.begin(),ipos) + 1 << endl;

    // Se crea otra bolsa por copia de rango.
    multiset<int> coll2(coll1.begin(), coll1.end());

    // Borrado de todos los elementos <= 3.
    coll2.erase (coll2.begin(), coll2.find(3));

    // Se eliminan los elementos iguales a 5.
    int num;
    num = coll2.erase (5);
    cout << num << " elemento(s) eliminados." << endl;
}
```



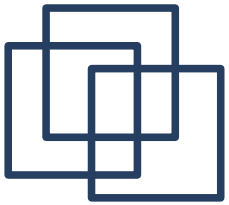
Bolsa: motivación, descripción y especificación

https://decsai.ugr.es/mgsilvente/stl/multiset.html



Descripción del Diccionario (map)

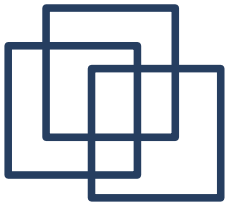
- Un diccionario es un contenedor que almacena elementos (valores) identificados mediante una clave, es decir, pares *(clave*, *valor)*, es decir, *(key_type*, *data_type)*.
- No existen pares *(clave*,*valor)* repetidos.
- Análogamente, al *multiset* para el *set*, existe el *multimap* para el *map*, permitiendo elementos con claves repetidas.



Motivación del Diccionario (map)

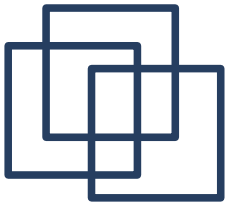
Ejemplos:

- El diccionario de la RAE contiene parejas (palabra,definición).
- El stock de un almacén: artículo, número de unidades que existe.
- Un guía telefónica: abonado, número de teléfono.



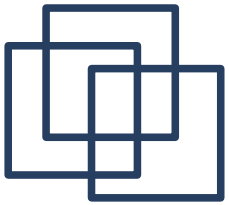
Ejemplos de uso de *map*

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

map<string,double> diccStringReal;
map<int,string> diccEntCad;
map<double, int,less<double> > diccRealEnt;

struct ltstr {
  bool operator() (const char* s1, const char* s2) const
  {
     return strcmp(s1, s2) < 0;
  }
}

map<char *, int, ltstr> diccCharEnt;
```



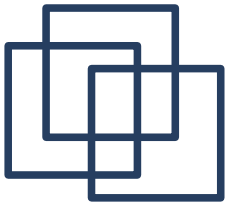
Ejemplos de uso de *map*

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

int main()
{
   typedef map<string, float> StringFloatMap;

   StringFloatMap acciones;
```



Ejemplos de uso de *map*

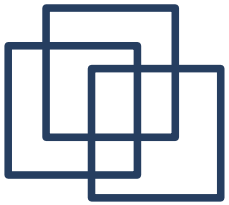
```
// Inserción de elementos.
 acciones["BASF"] = 369.50;   acciones["VW"] = 413.50;
 acciones["Daimler"] = 819.00;  acciones["BMW"]= 834.00;
 acciones["Siemens"] = 842.20;

// Impresión
   StringFloatMap::iterator pos;
   for (pos = acciones.begin(); pos != acciones.end();  ++pos)
       cout << "acción: " << (*pos).first << "\t"  << "precio: "
              << pos->second << endl;

   cout << endl;
// Se doblan los precios de la acción
   for (pos = acciones.begin(); pos != acciones.end();  ++pos)
       pos->second *= 2;
}
```



Ejemplos de uso de *map*

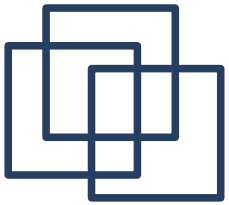
Conversión de un número en su representación en letra:
934 = nueve tres cuatro

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

typedef map<char, string, less<int> > dicc_CharString;

void construirMapa(dicc_CharString &);
void convertirCadena(string &, dicc_CharString &);
bool obtenerCadena(string &);
```



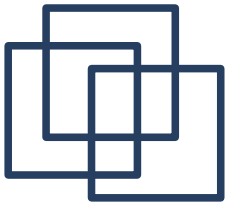
Ejemplos de uso de *map*

```
int main()
{
   dicc_CharString conversor;
   string cadena("");

   construirMapa(conversor);

   // Lectura de un número por la entrada estándar.
   while( obtenerCadena(cadena) )
      convertirCadena(cadena, conversor);

   return 0;
}
```



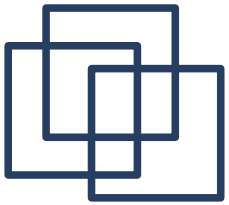
Ejemplos de uso de *map*

```
// Inicialización del conversor sin usar []
// void construirMapa(dicc_CharString & dicc) {
    dicc_CharString::iterator It;
    dicc.insert(dicc_CharString::value_type('0',"cero"));
    dicc.insert(pair<char, string>('3',"tres"));    /*alternativa*/
    dicc.insert(dicc_CharString::value_type('9',"nueve"));
    dicc insert(dicc_CharString::value_type('1',"uno"));
    dicc.insert(dicc_CharString::value_type('6',"seis"));
    dicc.insert(dicc_CharString::value_type('2',"dos"));
    dicc.insert(dicc_CharString::value_type('7',"siete"));
    dicc.insert(dicc_CharString::value_type('4',"cuatro"));
    dicc.insert(dicc_CharString::value_type('5',"cinco"));
    dicc.insert(dicc_CharString::value_type('8',"ocho"));
    dicc.insert(dicc_CharString::value_type('.',"punto"));
    dicc.insert(dicc_CharString::value_type(',',"coma"));

    for(It = dicc.begin(); It != dicc.end(); ++It)
      cout << It->first << "," << It->second << endl;
}
```



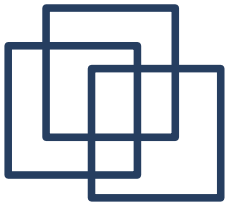
Ejemplos de uso de *map*

```
// Búsqueda de los dígitos de la cadena y conversión.
void convertirCadena(string &unaCadenaNumero, dicc_CharString &dicc)
{
  dicc_CharString::iterator unIterador;

  /* Extraer cada dígito de la cadena, encontrar su correspondiente
     entrada en el diccionario  (su palabra equivalente) y mostrarlo.      */

  for(int index = 0; index < unaCadenaNumero.length();  index++)
    {
      unIterador = dicc.find(unaCadenaNumero[index]);

      if( unIterador != dicc.end() ) // 0 - 9    ,    .
        cout << unIterador->second << " " << flush;
      else    // otro caracter.
        cout << "[err] " << flush;
    }
  cout << endl;
}
```



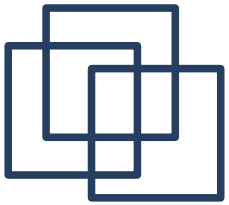
Ejemplos de uso de *map*

```
bool obtenerCadena(string &unaCadenaNumero)
{
  cout << "Introduce\"s\" para salir o introduce un'' <<
          ``número: ";

  cin >> unaCadenaNumero;

  return(unaCadenaNumero != "s");

}
```



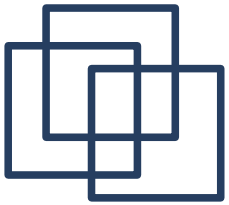
Descripción STL del tipo map

https://decsai.ugr.es/mgsilvente/stl/Map.html



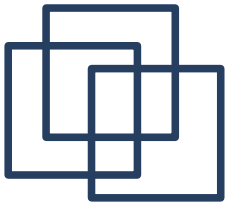
Contenedores secuencia en la STL

- ***Contenedor****: estructura de datos capaz de contener otras estructuras de datos.*
- **Contenedor de secuencia**: contenedor que contiene otras estructuras de datos de forma secuencial.



Contenedores secuencia en la STL

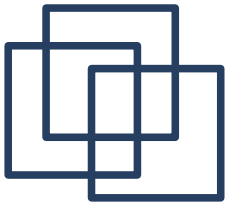
- El acceso a los elementos puede ser secuencial o aleatorio, dependiendo del tipo de contenedor.
- Almacenan y manejan los datos de forma lineal (sin índices), lo que perjudica a los accesos en ciertas aplicaciones.



Contenedores secuencia en la STL

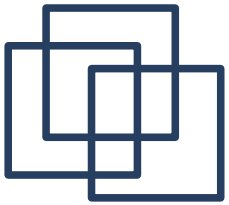
¿Qué secuencia usamos?

- **vector<T>**: cuando los elementos se tengan que acceder por el índice de la posición que ocupen y las inserciones y borrados sean al final.
- **list<T>:** cuando la mayoría de inserciones y borrados en medio de la secuencia sean comunes.
- **deque<T>** : cuando la mayoría de las inserciones y borrados tengan lugar al principio o al final de la secuencia.



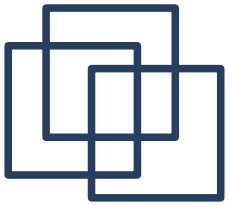
Operaciones de los contenedores de secuencia

Miembro	Descripción
`C x(n, elem)`	Crea una secuencia `x` que se inicializa con `n` copias de `elem`
`C x(iter1, iter2)`	Crea una secuencia cuyos elementos son copias de los que ya existen en otra secuencia entre los iteradores `iter1` e `iter2`
`x.insert(iter, elem)`	Inserta una copia del elemento `elem` en `x` directamente antes del elemento referenciado por `iter`
`x.insert(iter, n, elem)`	Inserta `n` copias del elemento `elem` en `x` directamente antes del elemento referenciado por `iter`
`x.insert(iter, first, last)`	Inserta una copia de los elementos desde el referenciado por `first` hasta el referenciado por `last` antes del elemento referenciado por `iter`



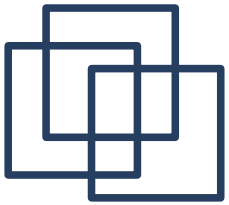
Operaciones de los contenedores de secuencia

Miembro	Descripción
`x.erase(iter)`	Elimina el elemento referenciado por `iter`
`x.erase(first, last)`	Elimina los elementos que van desde el referenciado por `first` hasta el referenciado por `last`
`x.clear ()`	Elimina todos los elementos del contenedor



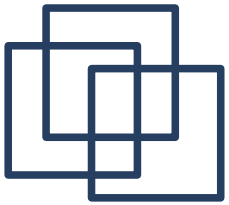
Operaciones opcionales de los contenedores secuencia

Miembro	Contenedor	Descripción
`x.front()`	`vector, list, deque`	Devuelve una referencia al primer elemento en el contenedor
`x.back()`	`vector, list, deque`	Devuelve una referencia al último elemento en el contenedor
`x.push_back (elem)`	`vector, list, deque`	Añade `elem` al final del contenedor `x`
`x.pop_front()`	`list, deque`	Elimina el primer elemento del contenedor `x`
`x.pop_back()`	`vector, list, deque`	Elimina el último elemento del contenedor `x`
`x[n], x.at(n)`	`vector`	Devuelven una referencia al `n`-ésimo elemento del contenedor `x` donde `0<=n<x.size()`



Vectores de la STL

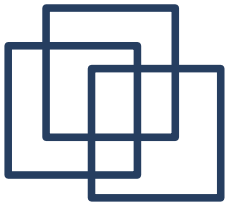
- El contenedor de secuencia más simple que existe.
- Puede concebirse como un vector de tamaño variable.
- Generalmente, almacena datos en bloques de tamaño fijo y cuando se sobrepasa ese tamaño, se reserva otro bloque.



Ejemplo de vector<T>

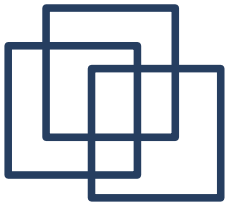
```
#include <vector>

using namespace
   std;

vector<int> v(6);

//v.size() es 6

v[0] = 12;
```

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

 vector<int> v;
 v.reserve(6) //v.size() es 0
 // no se puede usar []
 v.push_back(12);//v.size() es 1
```



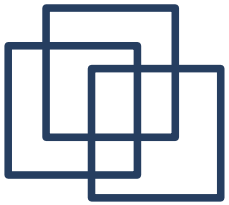
Ejemplo de vector<T>

```
vector<int> v;
int x;
cin >> x;
while (x>0) {
  v.push_back(x);
  cin >> x;
}
int suma = 0;
for (int i=0; i<v.size(); i++)
  suma+= v[i];
cout << suma;
```



Iteradores en vector<T>

- En este tipo de contenedor se puede sumar un valor a un iterador: Iteradores de acceso aleatorio.
- Se puede hacer v.begin() + 200
- v.erase(v.begin()+200);
- v.insert(v.end()-100, 12); // Debe existir esa posición!!



Matrices en la STL

- Una matriz se puede ver como un vector de vectores

vector<vector<int> > v(3)

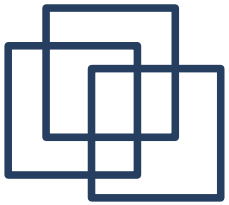
- Para inicializarlo a un valor para cada fila:

vector<vector<int> > v(3, vector<int>(8))

crea un vector con 3 filas y 8 columnas poniendo un cero en cada posición

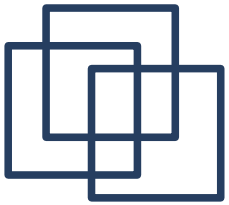
- Para acceder se usa []

cout << v[2][1];



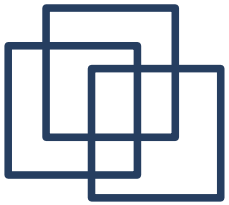
Vectores de la STL

https://decsai.ugr.es/mgsilvente/stl/Vector.html



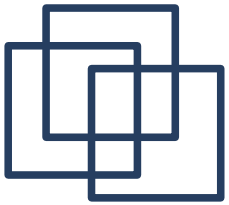
Listas en la STL

- Los vectores son ineficientes cuando se inserta o se borra en una posición que no sea la última (orden lineal)
- Los vectores hay que redimensionarlos y resulta muy ineficiente cuando se incrementa el número de elementos poco a poco.
- ***Lista****: es una secuencia de elementos en la que acceder al elemento i-ésimo requiere pasar por los i elementos anteriores e insertar un elemento en una posición tiene un orden constante.*



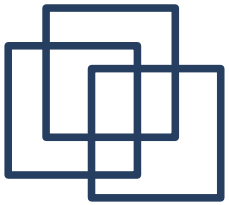
Ejemplos de uso del TDA Lista (list)

```
#include <list>

using namespace std;

list<int> listaEnteros;
list<string> listaCadenas;
list<MiTipo> listaMiTipo;
list<list<double> > listaDelistas;
```



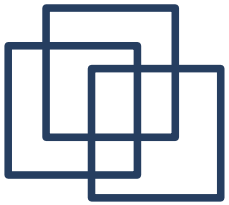
Ejemplos de uso del TDA Lista (list)

```
#include <iostream>
#include <list>
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);
}
```



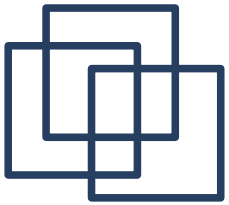
Ejemplos de uso del TDA Lista (list)

```
/**
     @param l es la lista a ordenar
     @brief Devuelve la Lista l ordenada
 */
void Ordenar(list<int> & l)
{
  list<int>::iterator itr, ant;
  itr = l.begin();
  ++itr;
  while (itr != l.end()) {
    ant = itr;
    --ant;
    if(*ant > *itr) {
      while (*ant>*itr &&
              ant!=l.begin()) --ant;
          ...
                ...
  if (ant==l.begin() &&
(*ant>*itr))
              l.insert(l.begin(),*itr);
    else {
      ++ant;
      l.insert(ant,*itr);
    }
     itr = l.erase(itr);
 } // if
 else ++itr;

 } // while
} // función
```



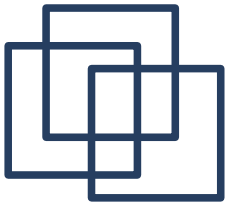
Ejemplos de uso del TDA Lista (list)

```
/**
  @brief Inserta el elemento x en l, ordenadamente
  @param  x es el elemento a insertar
  @param  l es una Lista ordenada
*/
void InsertaOrdenado(int x, list<int> &l)
{
  if (l.empty())
    l.push_back(x);
  else {
    list<int>::iterator i = l.end();
    i--;
    while (i!=l.begin() && *i>x) i--;
    if (*i<x)
      i++;
    l.insert(i, x);
  }
}
```



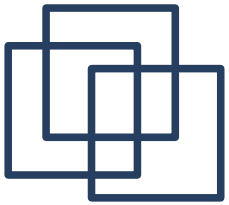
Ejemplos de uso del TDA Lista (list)

```
/**
   @brief Elimina los elementos duplicados de una lista.
   @param: lista a procesar. Es MODIFICADO.
*/
template <typename T>
void EliminaDuplicados(list<T> &  l)
{
  for (typename 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;
    }
  }
}
```



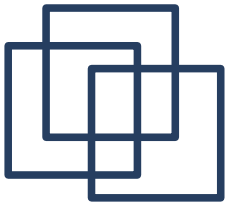
Ejemplos de uso del TDA Lista (list)

```
/**
   @brief Crea una lista con los elementos comunes a las
         dos listas argumentos.
   @param l1, l2: Listas a procesar.
   @param lsal: lista con los elementos comunes. ES MODIFICADO.
   */
template <typename 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);
          }
  } // bucle for
} // función comunes.
```



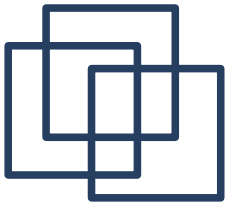
Ejemplos de uso del TDA Lista (list)

```
/**
  @brief Calcula los elementos menores para cada elemento de una lista
  @param ll lista conteniendo los elemenos

  @param lc lista donde se guarda cada valor y los menores que ese
valor

Calcula una lista lc tal que para cada elemento de  l1 contiene el número
de elementos consecutivos  en l1 que, precediendo al elemento, verifican
 que son menores que él.
    */
template <typename T>
void crecimiento(const list<T> & l1, list<pair<T, int> > &lc)
{
```



Ejemplos de uso del TDA Lista (list)

```
 typename list<T>::const_iterator it, r;
 pair<T, int> aux(*(l1.begin()), 0);

 lc.push_back(aux);
 it = l1.begin(); it++;
 while (it!= l1.end()) {
   aux.second = 0;
   aux.first = *it;
   r = it; --r;
   while (r!= l1.begin() && *r < *it) {
     --r;
     aux.second++;
   } // while
   if (r == l1.begin() && *r < *it)
     aux.second++;
   lc.push_back(aux);
   it++;
 }// for
} // función
```



Listas en la STL

https://decsai.ugr.es/mgsilvente/stl/List.html