5. STL: Contenedores

5.1 Introducción

La Standard Template Library, STL, implementa los contenedores lineales vistos en el capítulo 4. De esta forma, los nombres de las clases asociadas a estos contenedores lineales y como incluir la librería donde están definidos se puede ver en la siguiente tabla:

T.D.A.	**Clase**	**Como incluirla**
Vector Dinámico	vector	#include<vector>
Lista	list	#include<list>
Cola/Cola con prioridad	queque / priority_queue	#include<queue>
Pila	stack	#include<stack>
Doble Cola	deque	include <deque>
Vector Estático	array	include <array>
Listas enlace simple	forward_list	include <forward_list>

En este tema nuestro primer objetivo será revisar estos contenedores lineales que se implementan en la STL. De esta forma en este tema se describe una versión ajustada a los contenedores estudiados. Una versión con más detalle de la STL se puede consultar en `http://www.cplusplus.com/`. En una segunda parte veremos contenedores no lineales: set, multiset, map y multimap. Y otro contenedores no lineales que permiten realizar búsquedas en tiempos constantes como son: unordered$_set$, $unordered_map$.

5.1.1 Matriz

Como ejemplo de uso de la clase vector de la STL vamos a implementar la clase Matriz. Para ello Matriz se representa como un vector de vectores.

```
#ifndef _MATRIZ_H
#define _MATRIZ_H
#include <iostream>
#include <vector>
#include <cassert>
#include <algorithm>
using namespace std;
template <class T>
class Matriz{
        private:
                    vector<vector<T> > m; //vector de vectores
                    int nr,//numero de filas
                        nc;//numero de columnas

                    /**
                      @brief Copia una matriz en otra
                      @param M: matriz origen
                     */
                    void Copiar(const Matriz<T> &M);
                    /**
                      @brief Borra la memoria asignada a matriz
                     */
                    void Borrar()
        public:
                /**
                  @brief Constructor por defecto
                  */
                    Matriz():nr(0),nc(0){}

                /**
                  @brief Constructor por parametros
                  @param nr: numero de filas
                  @param nc: numero de columnas
                  @param value: valor al que se inicia cada elemento de la matriz
                 */
                    Matriz(int nr,int nc, const T&value = T());

           /**
                  @brief Constructor de copia
                  @param M: matriz que se copia
                 */
                    Matriz(const Matriz<T>&M){
                          Copiar(M);
```

```
44                      }
45
46              /**
47                @brief Destructor
48                */
49                      ~Matriz(){
50                         Borrar(); //se podria dejar en blanco.
51                        }
52              /**
53                @brief Operador de asignacion
54                @param M: matriz a asignar
55                @return una referencia al objeto
56                */
57                      Matriz<T> & operator=(const Matriz<T> & M){
58                        if (this!=&M){
59                                  Borrar();
60                                  Copiar(M);
61                        }
62                        return *this;
63              }
64
65              /**
66                @brief Devuelve el numero de filas
67                */
68                      int filas()const{ return nr;}
69
70              /**
71                @brief Devuelve el numero de columnas
72                */
73                      int cols()const{ return nc;}
74              /**
75                @brief Anhade una columna. La inserta  en una posicion dada
76                @param pos: posicion donde inserta la nueva columna
77                @param v: vector con los elementos de la columna
78                @pre v debe tener filas() elementos
79                */
80                      Matriz<T>& addCol(int pos,const  vector<T> &v);
81
82
83              /**
84                @brief Anhade una fila. La inserta en una posicion
85                @param pos: posicion donde inserta la nueva fila
86                @param v: vector con los elementos de la fila
```

```
87                    @pre v debe tener cols() elementos
88                    */
89                          Matriz<T>& addRow(int pos,const  vector<T> &v);
90
91                  /**
92                    @brief Borra una columna de la matriz
93                    @param pos: posicion de la columna a borrar
94                    @pre pos debe estar comprendida entre 0 y cols()-1
95                    */
96                          Matriz<T>& delCol(int pos);
97
98                  /**
99                    @brief Borra una fila de la matriz
100                   @param pos: posicion de la fila a borrar
101                   @pre pos debe estar comprendida entre 0 y filas()-1
102                   */
103                         Matriz<T>& delRow(int pos);
104
105                 /**
106                   @brief Obtiene el elemento de una fila,columna
107                   @param i: fila del elemento
108                   @param j: col del elemento
109                   */
110                         const T & operator()(int i,int j)const;
111
112                 /**
113                   @brief Obtiene el elemento de una fila,columna
114                   @param i: fila del elemento
115                   @param j: col del elemento
116                   */
117                         T & operator()(int i,int j);
118
119                 /**
120                   @brief Intercambio de dos matrices
121                   @param M: matriz con la que se realiza el intercambio
122                    */
123
124                         void swap(Matriz<T> &M);
125
126                 /**
127                   @brief Escritura de una matriz por el flujo estandar
128                   @param os: flujo de salida
129                   @param M: matriz que se escribe
```

```
130                  */
131                         template <class U>
132                         friend ostream &operator<<(ostream & os, const Matriz<U> &M);
133
134
135                 /**
136                   @brief Lectura de una matriz desde el flujo estandar
137                   @param is: flujo de entrada
138                   @param M: matriz en la que se lee. ES MODIFICADA
139                  */
140                         template <class U>
141                         friend istream & operator>>(istream & is,  Matriz<U> &M);
142 };
143 #include "Matriz.h"
144 #endif
```

En este ejemplo se ha representado la matriz haciendo uso de vector de la STL. Esta clase no gestiona la memoria dinámica por lo tanto no necesita implementar el operador de asignación ni el constructor de copias. No obstante para ver el uso de las funcionalidades de la clase vector se han implementado. Veamos a continuación como se realiza la implementación de los métodos.

```
1  template <class T>
2  void Matriz<T>::Copiar(const Matriz<T>&M){
3          this->nr= M.nr;
4          this->nc = M.nc;
5          m = vector< vector<T> >(nr);
6          typename vector<vector<T> >::iterator it_r;
7          typename vector<vector<T> >::const_iterator it_r_m;
8          for (it_r_m=M.m.begin(),it_r=m.begin();it_r_m!=M.m.end(); ++it_r_m,++it_r){
9                  //reservamos la memoria para la fila
10                 (*it_r)=vector<T>(nc);
11                 //asignamos a la fila todos los elementos de la misma fila en M
12                 (*it_r).assign((*it_r_m).begin(),(*it_r_m).end());
13         }
14         // La asignacion mas facil
15         // for (int i=0;i<nr;i++){
16         //    m[i]= vector<T>(nc);
17         //    for (int j=j<nc;j++)
18         //                  m[i][j]= M.m[i][j];
19         // }
20 }
21 template <class T>
22 void Matriz<T>::Borrar(){
23         //OPCION 1: m.clear();
```

```
24          //OPCION 2: m.erase(m.begin(),m.end());
25          //OPCION 3: borrar cada fila y luego el vector que direcciona a las filas
26          typename vector<vector<T> >::iterator it_r;
27          for (it_r = m.begin(); it_r!=m.end();++it_r)
28           (*it_r).erase((*it_r).begin(),(*it_r).end());
29
30          m.erase(m.begin(),m.end());
31  }
32  template <class T>
33  Matriz<T>::Matriz(int nr,int nc, const T&value = T()){
34          this->nr=nr;
35          this->nc=nc;
36          m= vector< vector<T> >(nr);
37          typename vector <vector<T> >::iterator it;
38          for (it=m.begin(); it!=m.end();++it)
39            *it=vector<T>(nc,value);
40  }
41
42  template <class T>
43  Matriz<T>& Matriz<T>::addCol(int pos,const  vector<T> &v){
44          //nos aseguramos que v tiene tantos elementos como filas
45           assert(v.size()==nr);
46
47           typename vector <vector<T> >::iterator it;
48           typename vector<T>::const_iterator itv=v.begin();
49           for (it=m.begin();it!=m.end(); ++it,++itv){
50             //tenemos que hacer un resize de cada fila
51             (*it).resize(nc+1);
52
53             for (int i=nc;i>pos;i--)//abrimos hueco para insertar el elemento
54              (*it)[i]=(*it)[i-1]; //OPCION2: *it.at(i)=*it.at(i-1)
55
56             (*it)[pos]=*itv;//ponemos el elemento
57           }
58           nc++;//incrementamos el numero de columnas
59           return *this;
60  }
61  template <class T>
62  Matriz<T>& Matriz<T>::addRow(int pos,const  vector<T> &v){
63           //v debe tener tantos elementos como columnas
64           assert(v.size()==nc);
65
66           m.resize(nr+1);//tenemos que hacer un resize de cada  columna
```

```
67          for (int i=nr;i>pos;--i)//Copiamos del final hacia pos la anterior
68                    m[i].assign(m[i-1].begin(),m[i-1].end());
69          m[pos].assign(v.begin(),v.end());//copiamos v como la nueva fila
70          nr++;//el numero de filas se incrementa
71          return *this;
72  }
73  template <class T>
74  Matriz<T>& Matriz<T>::delCol(int pos){
75          typename vector <vector<T> >::iterator it;
76          for (it=m.begin();it!=m.end();++it){ //de cada fila
77            for (int i=pos;i<nc-1;i++)//movemos una posicion los elems. desde pos a nc-1
78                    (*it).at(i)=(*it).at(i+1);
79            (*it).pop_back();//borramos el ultimo
80          }
81          nc--;//decrementamos el numero de columnas
82          return *this;
83  }
84  template <class T>
85  Matriz<T>& Matriz<T>::delRow(int pos){
86          for (int i=pos;i<nr-1;i++)
87             m[i]=m[i+1];//invocamos al operador = de vector
88          m.pop_back();//se borra la ultima fila
89          nr--;//decrementamos el numero de filas
90          return *this;
91  }
92  template <class T>
93  void Matriz<T>::swap(Matriz<T> & M){
94          m.swap(M.m);//se invoca a swap de vector
95          std::swap(nr,M.nr);//es necesario anteponer std para eliminar ambiguedad
96          std::swap(nc,M.nc);
97  }
98  template <class U>
99  ostream & operator<<(ostream & os, const Matriz<U> &M){
100         for (int i=0;i<M.nr;i++){
101                   os<<"\n";
102                   for (int j=0;j<M.nc;j++)
103                            os<<M.m[i][j]<<" ";
104         }
105         return os;
106 }
107 template <class U>
108 istream & operator>>(istream & is,  Matriz<U> &M){
109         int nr,nc;
```

```
110         is>>nr>>nc;
111         Matriz<U>Maux(nr,nc);
112         for (int i=0;i<nr;i++){
113          for (int j=0;j<nc;j++)
114           is>>Maux.m[i][j];
115         }
116         M=Maux;
117         return is;
118 }
```

Iteradores de Matriz

Una vez vistos los métodos de Matriz vamos a añadir un iterador para que recorra los elementos de la matriz. La forma de recorrerlos depende de la semática del iterador. Por ejemplo supongamos que queremos un iterador para que recorra primero todos los elementos de la primera fila a continuación los elementos de la segunda fila y así.

```
1  template <class T>
2  class Matriz{
3          private:
4           vector<vector<T> > m; //vector de vectores
5           int nr,//numero de filas
6           int nc;//numero de columnas
7          ...
8          public:
9          class const_iterator;
10         class iterator{
11           private:
12             typename vector<vector<T> >::iterator it_row;//recorre las filas
13             typename vector<vector<T> >::iterator it_row_end;//fin de las filas
14             typename vector<vector<T> >::iterator it_row_begin;//comienzo de las filas
15             typename vector<T>::iterator it_col;//en la  fila recorre columna
16           public:
17             iterator(){}
18             iterator & operator++(){
19                 ++it_col;//avanzamos a la siguiente columna en la misma fila
20                 if ((*it_row).end()==it_col){//hemos llegado al final de la fila
21                   ++it_row;//avanzamos en fila
22                   if (it_row!=it_row_end) //no he recorrido todas las filas
23                    it_col = (*it_row).begin();//se inicia a la primera columna
24                   else
25                    it_col = (*it_row_begin).end();//lo pongo a la ultima de la primera
26                 }
27                 return *this;
28             }
```

```
29              iterator & operator--(){
30                 if (it_col==(*it_row).begin()){//es el primero
31                  if (it_row!=it_row_begin){
32                      --it_row;
33                      it_col= (*it_row).end();
34                      --it_col;
35                  }
36                  else {
37                    it_col=*(it_row).end();
38                    it_row=it_row_end;
39                  }
40                }
41                else --it_col
42                return *this;
43              }
44
45              T & operator *()const{
46                 return *it_col;
47              }
48              bool operator==(const iterator &i){
49               return i.it_row==it_row && i.it_col=it_col;
50              }
51              bool operator!=(const iterator &i){
52               return !(i.it_row==it_row && i.it_col=it_col);
53              }
54              friend class Matriz<T>;
55              friend class const_iterator;
56            };
57
58
59            class const_iterator{
60             private:
61              typename vector<vector<T> >::const_iterator it_row;//recorre las filas
62              typename vector<vector<T> >::const_iterator it_row_end;//fin de las filas
63              typename vector<vector<T> >::const_iterator it_row_begin;
64              typename vector<T>::const_iterator it_col;
65             public:
66              const_iterator(){}
67              const_iterator(const iterator &i){
68                it_row =i.it_row;
69                it_row_end=i.it_row_end;
70                it_row_begin= i.it_row_begin;
71                it_col = i.it_col;
```

```
72
73              }
74              const_iterator & operator++(){
75                ++it_col;//avanzamos a la siguiente columna en la misma fila
76                if ((*it_row).end()==it_col){//hemos llegado al final de la fila
77                    ++it_row;//avanzamos en fila
78                    if (it_row!=it_row_end) //no he recorrido todas las filas
79                     it_col = (*it_row).begin();//se inicia a la primera columna
80                    else
81                     it_col = (*it_row_begin).end();//lo pongo a la ultima de la primera
82                }
83                return *this;
84              }
85              const_iterator & operator--(){
86                if (it_col==(*it_row).begin()){//es el primero
87                  if (it_row!=it_row_begin){
88                      --it_row;
89                      it_col= (*it_row).end();
90                      --it_col;
91                  }
92                  else {
93                    it_col=*(it_row).end();
94                    it_row=it_row_end;
95                  }
96                }
97                else --it_col
98                return *this;
99              }
100
101             const T & operator *()const{
102                return *it_col;
103             }
104             bool operator==(const const_iterator &i){
105              return i.it_row==it_row && i.it_col=it_col;
106             }
107             bool operator!=(const const_iterator &i){
108              return !(i.it_row==it_row && i.it_col=it_col);
109             }
110             friend class Matriz<T>;
111
112          };
113
114         ....
```

```
115          //Funciones begin y end
116          iterator begin(){
117           iterator i;
118           i.it_row = m.begin();//fila inicial
119           i.it_col = (*(m.begin())).begin();//columna primera de la primera fila
120           i.it_row_end=m.end();//siguiente a la ultima fila
121           i.it_row_begin= m.begin();//fila primera
122           return i;
123          }
124          iterator end(){
125           iterator i;
126           i.it_row = m.end();//siguiente a la ultima fila
127           i.it_col = (*(m.begin())).end();//ultima de la primera fila
128           i.it_row_end=m.end(); //siguiente a la ultima fila
129           i.it_row_begin= m.begin();//primera fila
130           return i;
131          }
132
133          const_iterator begin()const{
134           const_iterator i;
135           i.it_row = m.begin();
136           i.it_col = (*(m.begin())).begin();
137           i.it_row_end=m.end();
138           i.it_row_begin= m.begin();
139           return i;
140          }
141          const_iterator end()const {
142           const_iterator i;
143           i.it_row = m.end();
144           i.it_col = (*(m.begin())).end();
145           i.it_row_end=m.end();
146           i.it_row_begin= m.begin();
147           return i;
148          }
149
150 };
151
152 #endif
```

Un ejemplo de uso de la clase iterador de Matriz se puede ver en el siguiente código:

```
1 template <class T>
2 void Imprimir(const Matriz<T> &M){
3       typename Matriz<T>::const_iterator it;
```

```
4         for for (it=M.begin(); it!=M.end();++it){
5           cout<<*it<<endl;
6         }
7   }
8   template <class T>
9   Matriz<T> operator+(const Matriz<T> &M1, const Matriz<T>&M2){
10    Matriz<T>Ms(M1.filas(),M1.cols());
11    typename Matriz<T>::iterator it_s;
12    typename Matriz<T>::const_iterator it1,it2;
13    for (it1=M1.begin(),it2=M2.begin,it_s=Ms.begin; it1=M1.end();++it1,++it2,++it_s)
14       *it_s=(*it1)+(*it2);
15    return Ms;
16  }
17
18  int main(){
19
20    Matriz<int> M1,M2,M3;
21    //leemos las matrices
22    cin>>M1;
23    cin>>M2;
24    M3=M1+M2;
25    Imprimir(M3);
26  }
```

Ejercicio 5.1
Dar la implementación y representación de un iterador de Matriz que la recorra de la siguiente forma: En primer lugar todos los elementos de la primera columna, a continuación los elementos de la segunda columna, asi hasta llegar a recorrer la última columna. Implementar los métodos begin y end de Matriz para este iterador

□

5.1.2 Functor: Funciones Objeto

Antes de seguir viendo el resto de cotenedores, vamos a hacer un paréntesis para describir que son los functores. Los functores son objetos que pueden ser tratados como funciones o punteros a funciones. En este sentido podríamos tener el siguiente código:

```
1  miclasefunctor functor;
2  functor(1,2,3);
```

C++ permite sobrecargar el operador (), siendo esta la función a la que realmente se llama (*operator()(int,int,int)* de la clase miclasefunctor). El operador () puede tomar cualquier numero de parámetros y de cualquier tipo y devolver lo que desees. El objeto functor puedes construirlo de diferentes formas

para que tenga diferentes estados y por lo tanto su forma de actuar sea dependiente del estado en el que se encuentre. Por ejemplo:

```
class miclasefunctor{
        private:
         int x;
        public:
         miclasefunctor(int xx):x(xx){}
         int operator()(int y){
                 return x+y;
         }
};
int main(){
        miclasefunctor a1(5); //se construye con a1.x=5
        std::cout<<a1(6);//se escribe 11
}
```

Un ejemplo mas elaborado para usar functores. Suponer que tenemos la clase punto compuesto de valor x y del valor y. Además tenemos un vector de puntos en el que algunas veces queremos ordenarlo por la abscisa x y otras veces por la ordenada y.

```
class Punto{
 private:
         int x,y;
 public:
 ...
   int Get(bool want_x){
     if (want_x) return x;
     else return y;
   }
 ...
};
class PuntoSorted{
  private:
          bool ordenx;
  public:
          PuntoSorted(bool ox): ordenx(ox){}
          bool operator()(const Punto &p1,const Punto &p2) {
            return p1.Get(ordenx)<p2.Get(ordenx);
          }
  ...
};
```

La clase *PuntoSorted* se puede crear con dos estados: un valor de *ordenx* a true; o un valor de *ordenx* a false. De esta forma cuando se invoca el operator() la comparación del punto se hará según el valor x si *ordenx* es true o según el valor de y si *ordenx* es false.

```
int main(){
        std::vector<Punto> mispuntos;
        //leemos en mispuntos una secuencia de puntos
        //Creamos dos functores
        PuntoSorted orden_x(true);
        PuntoSorted orden_y(false);

        //ordenamos por x
        sort(mispuntos.begin(),mispuntos.end(),orden_x);

        //ordenamos por y
        sort(mispuntos.begin(),mispuntos.end(),orden_y);

}
```

En el main supuesto que tenemos un vector de objetos de tipo Punto se puede ordenar por x y por y usando dos functores. Para ello se crear un functor *orden_x* con estado true. Asi cuando se invoca al operador () se realizará la comparación por x. Esto es lo que orcurre cuando se hace la llamada a sort para que ordener desde el principio hasta el fin del vector teniendo como tercer parámetro el functor *orden_x*. Ahora para ordenar por el valor y simplemente tenemos que volver a llamar sort con el functor *orden_y*.

5.1.3 Listas

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

int main()
{
    list<int> milista;
    for (int i=1; i<=5; i++)
        milista.push_back(i); //insertamos elementos al final de la lista

    list<int>::reverse_iterator rit; //para recorrer la lista al reves

    for (rit=milista.rbegin();rit!=milista.rend();++rit)
        cout << *rit << " ";

    //Las funciones rbegin y rend son especificas para reverse iterator,
    //devuelven rbegin el final de la lista y rend el principio.

    //Tras el for tendriamos la siguiente salida: 5 4 3 2 1

    list<int> iterator r;
```

```
22
23      for (ri=milista.begin(); ri!=milista.end(); ++ri)
24          cout << *ri << " ";
25
26      //Tras el for tendriamos la siguiente salida: 1 2 3 4 5
27
28      cout << "El tamanio de mi lista es " << milista.size();
29
30      //La funcion size devuelve el tamanio de una lista
31
32      int sum=0;
33
34      while (!milista.empty()) {
35          sum+=milista.front();   //devuelve el valor al frente de la lista
36          milista.pop_front();    //elimina el valor al final de la lista
37      }
38
39      //Alternativamente, podriamos haber usado los metodos back para
40      //obtener el final de la lista y pop_back para borrarlo.
41  }
```

Las funciones vistas hasta ahora son:

1. **push_back(i)**: añade el elemento i al final de la lista
2. **reverse_iterator**: un objeto de la clase reverse_iterator recorre la lista de final a principio.
3. **rbegin()**: método que devuelve un objeto reverse_iterator que apunta al último elemento válido de la lista
4. **rend()**: método que devuelve un objeto de reverse_iterator que apunta antes del elemento inicial de la lista.
5. **size()**: devuelve el número de nodos de la lista
6. **empty()**: nos dice si la lista está vacía
7. **front()**: nos devuelve el primer nodo de la lista
8. **back()**: nos devuelve el último nodo de la lista
9. **pop_front()**: borra el primer elemento de la lista
10. **pop_back()**: borra el último elemento de la lista.

Ahora vamos a ver una función nueva: **erase**.

Erase

Tiene dos versiones:

1. *iterator erase (iterator posicion)*: borra un elemento con su posición
2. *iterator erase (iterator first, iterator end)*: elimina todos los elementos desde first hasta end, end no incluido

Un ejemplo de **mal** uso de esta función sería:

```
1  void ErasePares (list<int> &l) {
2      list<int>::iterator it;
```

```
3
4       for (it=l.begin();it!=l.end();++it)
5           if ((*it)%2==0)
6               l.erase(it); //CUIDADO:Hay que recuperar it
7   }
```

Al hacer un erase del primer elemento, perderíamos it y por tanto, podríamos estar borrando cualquier cosa menos los elementos pares de la lista. Lo correcto sería:

```
1   void EraserPares (list<int> &l) {
2       list<int>::iterator it;
3       it=l.begin();
4       while (it!=l.end()) {
5           if ((*it)%2==0)
6               it = l.erase(it);
7
8           else
9               ++it;
10      }
11  }
```

Otro ejemplo de uso de la clase Lista sería:

```
1   #include <iostream>
2   #include <list>
3   using namespace std;
4
5   int main() {
6       int mienteros[] = {13,12,10,20};
7       lista<int> milista;
8       milista.assign(mienteros, mienteros+4);
9
10      //ahora mi lista tiene los valores 13 12 10 20
11      //otra forma de inicializarla seria milista.assign(7,100);
12      //y tendria 7 elementos iguales (100).
13
14      list<int> otralista;
15
16      otralista.assign(milista.begin(), milista.end());
17      //otralista tiene los mismos elementos que milista
18
19      list<int> l3;
20
21      list<int>::iterator it;
```

```
22
23      for (it=otralista.begin();it!=otralista.end();++it)
24          l3.push_back(*it); //l3 tiene los mismos elementos que otralista y que milista
25
26      it=l3.begin();
27      ++it;       //it apunta al segundo elemento de l3, 12
28
29      l3.insert(it,2,30); //ahora l3 tiene 13 30 30 12 10 20
30                          //it sigue apuntando a 12
31
32      l3.insert(it,4); //ahora l3 tiene 13 30 30 4 12 10 20
33                       // it apunta al 4 ahora
34
35      vector<int> v;
36
37      v.assign(l3.begin(),l3.end()); //v = 13 30 30 4 12 10 20
38
39      it = milista.begin();
40
41      milista.insert(it,v.begin(),v.end()); //milista = 13 30 30 4 12 20 13 12 10 20
42
43      milista.swap(l3); //intercambiamos el contenido de ambas
44
45      l3.clear(); //limpiamos l3.l3 se queda sin elementos.
46  }
```

Las funciones vistas en este ejemplo son:

1. **assign**: según los parámetros, asigna a la lista unos valores u otros. Las posibilidades de llamada de assign son
   a) *assign(iterator inicio,iterator fin)*: el inicio y el final de una serie de elementos dados por dos iteradores. Asigna a la lista los valores de ese rango.
   b) *assign(int n,const T &v)*: un entero y un objeto de un tipo concreto. Asigna a la lista el segundo parámetro tantas veces como diga el primer parámetro.
2. **insert**: según los parámetros funciona de una manera u otra:
   a) *insert(iterator it,int n,const T v)* tres parámetros: un iterador, un entero y un objeto de un tipo. Inserta en la posicion apuntada por el iterador el objeto tantas veces como diga el entero. El iterador sigue apuntando al valor que apuntaba antes de la inserción
   b) *insert(iterator it,const T v)* dos parámetros: una posicion y un objeto. Inserta en la posición dada por el iterador el objeto y el iterador pasa a apuntar a ese objeto
   c) *insert(iterator it_dest,iterator it_source_inicio,iterator it_source_fin)* tres parámetros: inserta en la primera posición, de la lista destino, todos los elementos desde la segunda hasta la tercera posicion del mismo u otro contenedor.
3. **swap**: intercambia dos listas. Necesita un único parámetro que es la lista con la que se intercambia.

4. **clear**: deja la lista que llama a este método vacía

Splice

Mueve los elementos de una lista a otra. Según sus parámetros puede ser:

1. *splice(iterator it,lista l)* dos parámetros: un iterador y una lista. Coge todos los elementos de la lista l y los mueve a la lista con la que se llama a splice. Se inserta a partir de la posicion dada por it.La lista l queda vacía.
2. *splice(iterator itdes, lista l, iterator itsource)* Tres parámetros. Mueve el elemento apuntado por itsource de l a la lista con la que se llama al método y lo pone donde apunta itdes.
3. *splice(iterator itdes, lista l, iterator itsource_ini,iterator itsource_fin)* Cuatro parámetros: mueve todos los elementos apuntados desde itsource_ini hasta itsource_fin de l a la lista con la que se llama al método y se inserta a partir de la posición itdes.

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

int main () {
    list<int> l1, l2;
    list<int>::iterator it;

    for (int i=1;i<=4;i++)
        l1.push_back(i);     //l1 =  1 2 3 4

    for (int i=1;i<=3;i++)
        l2.push_back(i*10);      //l2 = 10 20 30

    it = l1.begin();
    ++it;

    l1.splice(it,l2);   //a partir de it, coge todos los elementos de
                        //l2 y los pone en l1:
                        //l1 = 1 10 20 30 2 3 4 (it apunta a 2)
                        //l2 = (vacia)

    l2.splice(l2.begin(), l1, it); //de los elementos de l1 coge el elemento it
                                   //l1 = 1 10 20 30 3 4
                                   //l2 = 2
                                   //el iterador se invalida

    it=l1.begin();

    advance(it,3); //avanza el iterador en tres posiciones *it=30

```

```
    l1.splice(l1.begin(),l1,it,l1.end()); //l1 = 30 3 4 1 10 20
}
```

Remove

void remove (const value &val): Elimina de una lista todos los elementos con valor val.

```
int mienteros[] = {3,4,79,15};
list<int> l (mienteros,mienteros+4); //l = {3,4,79,15}
l.remove(79); //l = 3 4 15
```

Si queremos ejecutar remove sobre objetos que hemos creado debemos sobrecargar el operator== en la clase. Por ejemplo, para hacer l.remove(P) (donde P es un objeto de tipo Palabra), debemos sobrecargar el operator== en la clase Palabra.

Remove_if

Elimina los elementos que cumplan una determinada condición. Por ejemplo, para eliminar números pares, definimos lo que significa ser número Par y después llamamos a Remove_if:

```
bool Par (int v) {
    return (v%2==0);
}

int main () {
    list<int> l;
    for (int i=1;i<10;i++)
        l.push_back(i);

    l.remove_if(Par)
}
```

En vez de usar la función Par podríamos haber creado con el mismo objetivo un functor. Por ejemplo:

```
class mifunctorpar{
  public:
   bool operator()(int & v){
           return (v%2==0);
   }
};

int main(){
   list<int> l;
   for (int i=1;i<10;i++)
           l.push_back(i);

   mifunctorpar f;
   l.remove_if(f)
}
```

Unique

Elimina valores duplicados que se encuentran consecutivamente, dejando una única ocurrencia.

$$\{1,1,2,2,3,4,4\} \longrightarrow \{1,2,3,4\}$$

Tenemos distintas formas de usar la función:

1. **unique()**: establece la igualdad con lo que esté predefinido en el operator==
2. **unique(funcion)**: establece la igualdad según la hayamos definido en funcion.

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

bool iguales_enteros (double v1, double v2) {
      return (int)v1 == (int)v2;
}

int main() {
      double m[] = {12.15, 2.72, 73.0, 12.77, 3.14, 12.77, 73.55, 72.25, 15.3, 72.25};
      list<double> l1(m,m+10);
      l1.sort(); // l1 = 2.72 3.14 12.15 12.77 12.77 15.3 72.25 72.25 73.0 73.35
      l1.unique(); // l1 = 2.72 3.14 12.15 12.77 15.3 72.25 73.0 73.35
      l1.unique(iguales_enteros); // l1 = 2.72 3.14 12.15 15.3 72.25 73.0
}
```

Sort

Con respecto a sort, que ordena los elementos, podemos pasarle una función que teniendo dos parámetros del tipo base de la lista devuelve un booleano indicando si se cumple o no una propiedad entre los dos parámetros de entrada.

```
#include <iostream>
#include <list>
#include <string>
#include <cctype>
#include <algorithm>
// compara sin tener distincion entre mayusculas y minusculas
bool compare_nocase (const std::string& first, const std::string& second)
{
  unsigned int i=0;
  while ( (i<first.length()) && (i<second.length()) )
  {
    if (tolower(first[i])<tolower(second[i])) return true;
    else if (tolower(first[i])>tolower(second[i])) return false;
    ++i;
  }
  return ( first.length() < second.length() );
```

```
}

int main ()
{
  std::list<std::string> milista;
  std::list<std::string>::iterator it;
  milista.push_back ("one");
  milista.push_back ("two");
  milista.push_back ("Three");

  milista.sort();

  std::cout << "milista :";
  for (it=milista.begin(); it!=milista.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  milista.sort(compare_nocase);

  std::cout << "milista:";
  for (it=milista.begin(); it!=milista.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  //otra forma de ordenar usando sort de algorithm
  sort(milista.begin(),milista.end());

  //o usando compare_nocase
  sort(milista.begin(),milista.end(),compare_nocase);
  return 0;
}
```

Merge

Mezcla dos listas ordenadas. Si no le pasamos ningún argumento, usa el operator $<$ predefinido para saber quién es menor, pero podemos pasarle una función para establecer la relación de orden.

```
#include <iostream>
#include <list>
#include <string>
#include <cctype>
#include <algorithm>
// compara sin tener distincion entre mayusculas y minusculas
bool compare_nocase (const std::string& first, const std::string& second)
{
```

```
  unsigned int i=0;
  while ( (i<first.length()) && (i<second.length()) )
  {
    if (tolower(first[i])<tolower(second[i])) return true;
    else if (tolower(first[i])>tolower(second[i])) return false;
    ++i;
  }
  return ( first.length() < second.length() );
}

int main ()
{
  std::list<std::string> milista1,milista2;
  std::list<std::string>::iterator it;

  milista1.push_back ("manzana");
  milista1.push_back ("banana");

  milista2.push_back ("Pera");
  milista1.push_back ("Sandia");
  milista1.push_back ("Melon");

  milista1.sort();//banana manzana
  milista2.sort();//Melon Pera Sandias

  milista1.merge(milista2);//Melon Pera Sandia banana manzana

  milista2.push_back("apple");

  milista1.merge(milista2,compare_nocase); //apple Melon Pera Sandia banana manzana
    std::cout << "milista1 :";
  for (it=milista1.begin(); it!=milista1.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

Un segundo ejemplo de merge sería el siguiente

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

bool miComparacion (double v1, double v2)
{
```

```
6       return (int)v1 < (int)v2;
7   }
8
9   int main()
10  {
11      list<double> l1, l2;
12
13      l1.push_back(3.1);
14      l1.push_back(2.2);
15      l1.push_back(2.9);
16
17      l2.push_back(3.7);
18      l2.push_back(7.1);
19      l2.push_back(1.4);
20
21      l1.sort(); // l1 = 2.2 2.9 3.1
22      l2.sort(); // l2 = 1.4 3.7 7.1
23      l1.merge(l2); // l2 queda vacia y l1 = 1.4 2.2 2.9 3.1 3.7 7.1
24
25      l2.push_back(2.1); // l2 = 2.1
26
27      l1.merge(l2,micomparacion); // l1 = 1.4 2.2 2.9 2.1 3.1 3.7 7.1
28      // Como solo comparamos el valor entero, inserta el 2.1
29      // despues del 2.9, porque ambos son iguales
30  }
```

Reverse

Invierte una lista.

```
1   #include <iostream>
2   #include <list>
3
4   int main ()
5   {
6     std::list<char> milista;
7
8     for (char a='a'; a<'z'; ++a) milista.push_back(a);
9
10    milista.reverse();
11
12    std::cout << "milista:";
13    for (std::list<char>::iterator it=milista.begin(); it!=milista.end(); ++it)
14      std::cout << ' ' << *it;
15
```

```
16    std::cout << '\n';
17
18    return 0;
19  }
```

Ejercicio 5.2
Implementar la función:

template <class T>
void Duplicar(const list<T> entrada, list<T> salida)

tal que dada una lista genera una nueva lista para obtener otra tal que contenga los elementos de la lista de entrada intercalando el elemento en la posicion n-i-1. Así en la lista final tenemos que en la posicion 2*i se pone el elemento i de la lista original y en la posición 2*i+1 se pone el que ocupa en la posicion n-i-1. Dos ejemplos sería los siguientes:

Ejemplo 1:
Lista inicial: (a,b,c,d)
Lista final: (a,d,b,c,c,b,d,a)
Ejemplo 2:
Lista inicial: (1,2,3,4,5)
Lista final: (1,5,2,4,3,3,4,2,5,1)

□

5.1.4 Pair: par de valores

Es una estructura muy utilizada sobre todo cuando queremos representar datos que van en pares de valores diferentes. Su estructura es la siguiente:

```
1  template <class T1, class T2>
2  struct pair {
3        typedef T1 first_type; //alias de T1
4        typedef T2 second_type; //alias de T2
5
6        T1 first;
7        T2 second;
8
9        pair(): fist(T1()), second(T2()) {}
10       pair (const T1 &x, const T2 &y): first(x), second(y) {}
11       template <class u, class v>
12       pair (const pair<u,v> &p): first(p.first), second(p.second);
13 };
```

En la estructura *pair* cabe resaltar el constructor con otro pair, con posibilidad de que first y secondi tengan otros tipos:

```
      template <class u, class v>
      pair (const pair<u,v> &p): first(p.first), second(p.second);
```

Este constructor permite iniciar un objeto de tipo pair con otro tal que los tipos de los campos miembros sean compatibles. Por ejemplo podríamos tener lo siguiente:

```
 ....
 pair<int, float> pif(3,2.4);

 pair<int,int> otro(pif);//se construye con un pair con otros tipos

 pair<const char * ,const char *>cadenas("Hola","Adios");
 //se construye con de tipo const char *
 pair<string ,string>otras_cadenas(cadenas);
```

Un ejemplo de uso de pair sería:

```
#include <vector>
#include <iostream>
#include <utility> //permite hacer comparaciones de tipos pair
#include <string>

using namespace std;

int main() {
      pair <string, double> product1 ("tomates",3.25);
      pair <string, double> product2;
      pair <string, double> product3;

      product2.first="naranjas";
      product2.second=1.3;
      product3=make_pair("ciruelas", 2.2);

      vector < pair<string, double> > miv = {product1,product2,product3};

      for (int i=0; i<miv.size(); i++)
            cout << "Product " << i << ':' << miv[i].first
                  << " Precio: " << miv[i].second << endl;

      //version con iteradores
      vector < pair<string, double> >::iterator it;

      for (it=miv.begin();it!=miv.end();++it)
            cout << (*it).first << ' ' << (*it).second;
            //tambien valdria:
```

```
29              //cout << it->first << ' ' << it->second;
30  }
```

Ejemplo 5.1.1
En este ejemplo se va a dar una primera aproximación a la represetación de Diccionario. Un Diccionario es un T.D.A compuesto por una colección de pares, en el que el primer miembro es la clave y el segundo miembro es la definición. Para realizar este T.D.A vamos a usar *list* de la STL.

Un ejemplo de uso de Diccionario sería el siguiente

En este ejemplo de uso la información asociada es un string. El alumno podría pensar como modficar el fichero ejemplo de uso para que la información asociada fuese una lista de string

□

5.1.5 Pilas

Las funciones típicas de las pilas en la stl se nombran como:

1. size: numero de elementos de la pila
2. empty: true si la pila está vacía, falso en caso contrario.
3. top: devuelve el elemento que esta el la posición tope.
4. pop: elimina el elemento del tope.
5. push: inserta un nuevo elemento por el tope

Ejemplo 5.1.2
Como ejemplo vamos a implementar una cola a partir de una pila de la STL:

```
1   #include <stack> // para poder usar las pilas de la STL
2   using namespace std;
3
4   template <class T>
5   class Cola {
6   private:
7         stack<T> datos;
8
9   public:
10        T front() {
11              stack<T> aux;
12              T v;
13              while (!datos.empty()) {
14                    v=datos.top();
15                    aux.push(v);
16                    datos.pop();
17              }
18
```

```
19
20              while (!aux.empty()) {
21                    T s = aux.top();
22                    datos.push(s);
23                    aux.pop();
24              }
25
26              return v;
27        }
28
29        bool empty () const {
30              return datos.empty();
31        }
32
33        int size () const {
34              return datos.size();
35        }
36
37        void push (const T &v) {
38              datos.push(v);
39        }
40
41        void pop() { // tenemos que eliminar el que esta abajo de la pila
42              stack<T> aux;
43              while (!datos.empty()) {
44                    v=datos.top();
45                    aux.push(v);
46                    datos.pop();
47              }
48
49              aux.pop();
50              while (!aux.empty()) {
51                    T s = aux.top();
52                    datos.push(s);
53                    aux.pop();
54              }
55        }
56  }
```

□

5.1.6 Colas

Las colas como ya vimos siguen la política FIFO (*first input firts output*) y la STL la implementa como la clase queue en la biblioteca con el mismo nombre. Las operaciones típicas de las colas son:

1. size: numero de elementos de la cola
2. empty: true si la pila está vacía, falso en caso contrario.
3. front: accede al elemento en el frente
4. push: inserta un elmeneto por el final
5. back: accede al elemento por el final. (Esta no es una operación estándar de las colas).
6. pop: elimina el elemento en el frente

Ejemplo 5.1.3
Crear una función que usando la clase queue y stack ver si una frase es un palíndromo, sin tener en cuenta los espacios en blanco.

```
#include <queue> //para usar la queue
#include <stack> //para usar stack
#include <iostream>
#include <string>
using namespace std;

bool Palindromo(const string & frase){
  queue<char> q;
  stack<char> p;
  for (int i=0;i<frase.size();i++){
   if (frase[i]!=' '){
    q.push(frase[i]);
    p.push(frase[i]);
   }
  }
  while (!q.empty()){
    if (p.top()!=q.front()) return false;
    p.pop();
    q.pop();
  }
  return true;
}
```

□

5.1.7 Colas con prioridad

Permiten mantener una colección de elementos ordenados por su prioridad o preferencia. La stl implementa las colas con prioridad en la clase priority_queue en la biblioteca queue. Las funciones que caracterizan a estas colas son:

1. size: numero de elementos de la cola
2. empty: true si la cola está vacía, falso en caso contrario.
3. top: devuelve el elemento mas prioritario

4. pop: elimina el elemento mas prioritario
5. push: inserta un elemento en la posicion dictada por su prioridad.

Vamos a ver un ejemplo de uso:

```
#include <iostream>
#include <queue> //incluye tanto cola como cola de prioridad

using namespace std;

int main(){
      priority_queue<int> mypq;

      mypq.push(30);
      mypq.push(100);
      mypq.push(25);
      mypq.push(40);

      while (!mypq.empty()) {
            cout << mypq.top() << ' ';
            mypq.pop();
      }
}
```

La salida del programa será:

100 40 30 25

ya que la salida se basa en la prioridad, es decir, en qué número es mayor. Si definimos la prioridad en un objeto nuestro, debemos definir el operator> para saber qué objeto tiene más prioridad.

5.1.8 Doble cola

La doble cola (deque) contiene secuencias de elementos que cambian de tamaño de forma dinámica. De esta forma un objeto de tipo doble cola se puede expandir y contraer por los dos extremos (por el principio y final). Son similares a los vectores, pero con una mejor eficiencia en los procesos de inserción y borrado de los elementos. A diferencia de los vectores, la doble cola no garantiza almacenar sus elementos en localizaciones de memoria contiguas. Aunque se pueden acceder de forma directa a cada elemento. Si las operaciones de inserción y borrado se hacen por los extremos las doble colas se comportan mejor que los vectores. En cambio si estas operaciones se realizan en cualquier otro punto son menos eficientes que como se realizan en un vector. La doble cola se implementa en la STL en la clase deque en la biblioteca con el mismo nombre.

Ejemplo 5.1.4
Un ejemplo de uso de la doble cola es el siguiente:

```
#include <iostream>
#include <deque>
#include <vector>
int main ()
 {
    std::deque<int> mideque;
    // Inicializamos la doble cola
    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);
    // 1 10 2 3 4 5

    // "it" apunta a 10
    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());
    // 1 20 30 30 20 10 2 3 4 5

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

    std::cout << "mideque contiene:";

    for (it=mideque.begin(); it!=mideque.end(); ++it)

    std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

□

Ejemplo 5.1.5
Crear la clase PilaoCola que permita a un objeto actuar como una pila o como una cola dependiendo de como se inicialice una bandera.

```
//pilaocola.h
```

```
2  #include <deque>
3  #include <iostream>
4  using namespace std;
5  template <class T>
6  class PilaoCola{
7   private:
8      deque<T> datos;
9      bool is_cola; //bandera si es true actua como cola
10                   //si es false actua como pila
11
12   public:
13    PilaoCola(bool tipo):is_cola(tipo){}
14    T & operator()(){
15      if (is_cola){
16        return datos.front();
17      }
18      else
19        return datos.back();
20    }
21
22     const T & operator()()const{
23      if (is_cola){
24        return datos.front();
25      }
26      else
27        return datos.back();
28    }
29
30
31    int size()const{
32     return datos.size();
33    }
34
35    void Pop(){
36      if (is_cola){
37        datos.pop_front();
38      }
39      else
40       datos.pop_back();
41     }
42
43    void Push(const T & v){
44
```

```
45      datos.push_back(v);
46    }
47    bool empty()const{
48     return datos.size()==0;
49    }
50  };
```

En este código para implementar la función Frente cuando actua como cola o Tope cuando actua como Pila, hemos hecho uso del operador (). Un ejemplo de uso de la clase PilaoCola sería el siguiente:

```
1   #include "pilaocola.h"
2   int main(){
3     PilaoCola<int> pila(false);
4     PilaoCola<int> cola(true);
5
6     for (int i=10;i<100;i+=10){
7        pila.Push(i);
8        cola.Push(i);
9     }
10
11     std::cout<<"Los elementos de la pila son:";
12     while(!pila.empty()){
13        std::cout<<pila()<<' ';
14        pila.Pop();
15     }
16
17     std::cout<<std::endl;
18     std::cout<<"Los elementos de la cola son:";
19     while(!cola.empty()){
20        std::cout<<cola()<<' ';
21        cola.Pop();
22     }
23
24  }
```

□

5.1.9 Array

Los arrays son contenedores de tamaño fijo: mantienen un número especifico de elementos en una estructura lineal. Es mas eficiente que el vector en cuanto a almacenamiento, ya que no se expanden ni se contraen de forma dinámica. Esta estructura es la que conocemos como vector estático.

Ejemplo 5.1.6

```
#include <iostream>
#include <array>
 int main ()
 {
 std::array<int,10> myarray;//array de 10 elementos
 unsigned int i;
    // asignamos algunos elementos

    for (i=0; i<10; i++) myarray[i]=i;

    std::cout << "myarray contains:";

    for (i=0; i<10; i++)
      std::cout << ' ' << myarray[i];

    std::cout << '\n';
    std::array<int,3> otr={2,16,77};
    std::cout<<otr.front()<<std::endl;
    std::cout<<otr.back()<<std::endl;
   return 0;
}
```

□

5.1.10 Forward List

Son contenedores de secuencias que permiten insertar y borrar en cualquier punto de la secuencia en tiempo constante. Se implementan como listas con celdas enlazadas simples. Asi la diferencia fundamental entre forward_list y list es que mantienen, forward_list, un enlace al siguiente elemento, mientras que la list mantienen dos enlaces por elemento, uno apunta al siguiente y otro al anterior. Asi que sobre una forward_list se puede iterar hacia adelante solamente. Al igual que la list la forward_list se muestra poco eficiente para acceder a un elemento por su posicion. Debes de iterar desde el principio para acceder por ejemplo al sexto elemento. Esta clase no contiene la funcion size para saber cuantos elementos tiene, por razones de optimización de espacio. Así que para saber cuantos elementos tiene una forward_list podemos usar el algoritmo distance con begin y end de la forward_list.

```
#include <iostream>
#include <array>
#include <forward_list>

int main ()
{
  std::array<int,3> myarray = { 11, 22, 33 };
  std::forward_list<int> mylist;
  std::forward_list<int>::iterator it;
```

```
10
11    it = mylist.insert_after ( mylist.before_begin(), 10 );
12    // 10
13    //  ^  <- it
14    it = mylist.insert_after ( it, 2, 20 );
15    // 10 20 20
16    //       ^
17    it = mylist.insert_after ( it, myarray.begin(), myarray.end() );
18    // 10 20 20 11 22 33
19    //                ^
20
21    it = mylist.begin();
22    // 10 20 20 11 22 33
23    //  ^
24    it = mylist.insert_after ( it, {1,2,3} );
25    // 10 1 2 3 20 20 11 22 33
26    //       ^
27
28    std::cout << "mylist contiene:";
29    for (int& x: mylist) std::cout << ' ' << x;
30    std::cout << '\n';
31
32    //Eliminamos el frente
33    mylist.pop_front();
34    // 1 2 3 20 20 11 22 33
35    //    ^                  el iterador se mantiene
36
37    return 0;
38  }
```

5.2 Contenedores asociativos

A continuación vamos a presentar un conjunto de estructuras de datos que se caracterizan por definirse como contenedores asociativos.

Contenedor Asociativo: Es una colección de pares, en la que cada par se conforma de una clave y valor. Puede que no aparezca valor asociado a la clave y puede que el valor clave aparezca mas de una vez para asoiciarle diferentes valores. Las operaciones mas frecuentes de estos contenedores son:

- Añadir un par a la colección
- Eliminar un par de la colección
- Modificar un par existente
- Buscar un valor asociado con un determinada clave.

Estás operaciones son las operaciones típicas de un diccionario. En la STL de C++ los contendores asociativos ordenados que vamos a estudiar son:

- *Set/Multiset*: conjunto de claves (no llevan valor asociado), con la posibilidad de repetir clave (multiset) o no
- *Map/Multimap*: conjunto de pares, (clave, valor asociado). En el map solamente se permite una entrada por clave (no se repiten), en el multimap se permite varias entradas para una misma clave (se pueden repetir).

Debajo de estos T.D.A. se ha usado para su implementación Árboles. Tanto *set/multiset* y *map/multimap* se matienen los datos ordenados por el valor de la clave. Otros contenedores asociativos que no están ordenados por clave son las tablas hash que en la STL se representan como:

- *unordered_set/unordered_multiset*
- *unordered_map/unordered_multimap*

5.2.1 Set/multiset

La clase *set* representa un conjunto de elementos que se disponen de manera ordenada y en el que no se repiten elementos. Los datos que insertamos en el set se llaman *claves*. La clase *multiset* es lo mismo pero permite incluir elementos repetidos en el conjunto. Las funciones más destacadas son:

Count

Devuelve el número de elementos que son iguales a un valor de entrada. Si lo ejecutamos con un set como mucho obtendremos un 1, pero si lo ejecutamos con un objeto multiset, podemos obtener cualquier valor mayor o igual que cero.

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
	set<int> micnj; //Para un multiset seria: multiset<int> micnj;

	for (int i=1; i<5; i++)
		micnj.insert(i*3); //micnj = 3 6 9 12

	for (int i=0;i<10;i++) {
		if (micnj.count(i)!=0)
			cout << "Esta en el conjunto";

		else
			cout << "no esta";
	}
}
```

Swap

Intercambia dos conjuntos.

```cpp
#include <iostream>
#include <set>

using namespace std;

int main()
{
	int myv[] = {12,75,10,32,20,25};

	set<int> s1(myv,myv+3); //s1 = 10 12 75
	set<int> s2(myv+3,myv+6); //s2 = 20 25 32

	s1.swap(s2); //s1 = 20 25 32 y s2 = 20 12 75
}
```

Find

Busca en el conjunto un elemento y devuelve un iterador a el elemento. Si no lo encuentra, devuelve un iterador a end().

```cpp
#include <iostream>
#include <set>

using namespace std;

int main() {
	set<int> micnj;

	for (int i=1;i<5;i++)
		micnj.insert(i*3);

	set<int>::iterator it=micnj.find(9);

	micnj.erase(it); //micnj = 3 6 12
	//tambien valdria: micnj.erase(micnj.find(9));

	//imprimimos por pantalla todos los elementos del conjunto
	for (it=micnj.begin();it!=micnj.end();++it)
		cout << *it;
}
```

Equal_range

Su cabecera es:

```cpp
	pair<iterator,iterator> equal_range (const value_type &val)const;
```

El primer iterador apunta al primer elemento del conjunto que coincida con val y el segundo, al primer elemento del conjunto distinto de val. Por ejemplo, si hacemos un equal_range del siguiente multiset:

3 6 6 9

El primer iterador apuntaría al 6 que va después del 3 y, el segundo, al 9. Si lo ejecutamos sobre un set, el primer iterador estará en una posición y el segundo, en la siguiente, ya que no puede haber elementos repetidos.

```
#include <iostream>
#include <set>

using namespace std;

int main() {
      set<int> micnj;

      for (int i=1;i<5;i++)
            micnj.insert(i*10); //micnj = 10 20 30 40 50

      pair<set<int>::const_iterator,set<int>::const_iterator> ret;

      ret = micnj.equal_range(30);

      cout << "Limite inferior " << (*ret).first << endl; // 30
      cout << "Limite superior " << (*ret).second << endl; //40
}
```

Alternativa a equal_range:

Lower_bound y upper_bound

lower_bound devuelve un iterador al primer elemento que coincide con el elemento de entrada, si no existe devuelve un iterador al valor que no va antes más proximo. En el caso que todos sean menores devuelve end(). Y *upper_bound* devuelve un iterador al primer elemento mayor al valor de entrada, si no existe devuelve un iterador al mayor más próximo.

```
#include <iostream>
#include <set>

using namespace std;

int main() {
      set<int> micnj;

      for (int i=1;i<5;i++)
```

```
10              micnj.insert(i*10);
11
12      set<int>::iterator itlow, itup;
13
14      itlow = micnj.lower_bound(30);
15      itup = micnj.upper_bound(60);
16
17      micnj.erase(itlow,itup); //si le pasamos solo itlow borra el elemento
18                               //al que apunta itlow solo.
19  }
```

Value_comp

Devuelve un objeto comparador de set. El objeto se puede usar para comparar dos elementos del conjunto de manera que al comparar, el objeto comparador devuelve true si el primer objeto es menor que el segundo. Por ejemplo, vamos a hacer una función que nos diga cuántos elementos menores a x hay en el conjunto s:

```
1   int menores (const set<int> &s, int x) {
2       set<int>::value_compare micomp = s.value_comp();
3       set<int>::const_iterator it = s.begin();
4       int cnt = 0;
5
6       while (micomp(*it,x)) {
7           cnt++;
8           ++it;
9       }
10  }
```

5.2.2 Map/multimap

Un *map* está formado por parejas de valores: al primero se lo conoce como **clave**, y al segundo como el valor **asociado** a dicha clave. No permite valores de clave repetidos y se ordena según su clave. Podemos acceder, de forma directa, al valor asociado a la clave a través de la clave, pero no al revés. Para poder acceder a la clave a través del valor asociado a ésta hay que realizar una búsqueda secuencial por valor asociado. Los *multimap* son lo mismo pero permiten valores de clave repetidos. Usan los mismos métodos que set y multiset. En un map podemos usar el operator[] para acceder a los elementos asociados a la clave o modificarlos.

Ejemplo 5.2.1
A continuación se define un map que tiene claves con tipo char y valor asociado string.

```
1   map<char,string> mymap;
2   mymap['a'] = "un elemento";
3   mymap['c'] = mymap['a'];
4   cout << mymap['a'];
```

□

Ejemplo 5.2.2

Usando un map vamos a implementar el T.D.A guía de teléfonos:

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

class Guiadetelefonos {
private:
      map<string,string> datos;
      //el primer string es el nombre de la persona y el segundo su tlf
      //la clave seria el nombre de la persona

public:
      string & operator[] (const string &nombre) {
            return datos[nombre];
      }

      //para saber los telefonos asociados a un nombre con un multimap
      vector<string> operator[] (const string &nombre) {
            pair<map<string,string>::const_iterator,
                  map<string,string>::const_iterator> ret;

            ret = datos.equal_range(nombre);

            vector<string> vaux;

            multimap<string,string>::const_iterator it;

            for (it=ret.first;it!=ret.second;++it)
                  vaux.push_back(it->second);

            return vaux;
      }

      string getTelefono (const string &nombre) {
            map<string,string>::iterator it=datos.find(nombre);

            if (it==datos.end())
```

```
38              //si no lo encuentra devolvemos un string vacio
39                    return string("");
40
41              else
42                    return it->second; //(*it).second;
43        }
44
45        //insertamos un numero de telefono y
46        //devolvemos true si se ha insertado con exito
47        pair <map<string,string>::iterator,bool> insert (string noombre, string tlf) {
48              pair<string,string> p(nombre, tlf);
49              pair<map<string,string>::iterator, bool> ret;
50
51              //solo tenemos que llamar a la funcion insert de map que devuelve true
52              //si el telefono apuntado por it ha sido insertado con exito en la guia
53              ret=datos.insert(p);
54
55              return ret;
56        }
57
58        //para borrar un telefono de la guia
59        void borrar (const string &nombre) {
60              map<string,string>::iterator itlow=datos.lower_bound(nombre);
61              map<string,string>::iterator itup=datos.upper_bound(nombre);
62
63              /*Para asegurarnos de que el nombre esta en la guia debemos
64              comprobar que nombre coincide con itlow.first o habernos
65              asegurado antes de llamar a la funcion, porque si no,
66              podemos borrar a otra persona*/
67              if(itlow.first == nombre)
68                    datos.erase(itlow,itup);
69
70              /*otra forma:
71              pair<map<string,string>::const_iterator,
72                    map<string,string>::const_iterator> ret;
73              ret = datos.equal_range(nombre);
74              datos.erase(ret.first,ret.second);*/
75        }
76
77        int size()const {
78              return datos.size();
79        }
80
```

```
81        //para saber cuanta gente en nuestra guia tiene el mismo nombre
82        unsigned int contabiliza (const string &nombre) {
83              return datos.count(nombre);
84        }
85
86        void clear() {
87              datos.clear();
88        }
89
90        friend class const_iterator;
91
92        class iterator {
93        private:
94              map<string,string>::iterator it;
95
96        public:
97              iterator & operator++() {
98                    ++it;
99                    return *this;
100             }
101
102             iterator & operator--() {
103                   --it;
104                   return *this;
105             }
106
107             pair<string,string> & operator*() {
108                   return *it;
109             }
110
111             bool operator== (const iterator &i)const {
112                   return it==i.it;
113             }
114
115             bool operator!= (const iterator &i)const {
116                   return it!=i.it;
117             }
118
119             friend class Guiadetelefono;
120             friend class const_iterator;
121       };
122       class const_iterator {
123       private:
```

```
124             map<string,string>::const_iterator it;
125
126       public:
127             const_iterator(const iterator &i):it(i.it){}
128             const_iterator & operator++() {
129                   ++it;
130                   return *this;
131             }
132
133             const_iterator & operator--() {
134                   --it;
135                   return *this;
136             }
137
138             const pair<string,string> & operator*() {
139                   return *it;
140             }
141
142             bool operator== (const const_iterator &i)const {
143                   return it==i.it;
144             }
145
146             bool operator!= (const const_iterator &i)const {
147                   return it!=i.it;
148             }
149
150             friend class Guiadetelefono;
151
152       };
153
154       //funciones begin y end
155       iterator begin() {
156             iterator i;
157             i.it=datos.begin();
158             return i;
159       }
160
161       iterator end() {
162             iterator i;
163             i.it=datos.end();
164             return i;
165       }
166
```

```
167
168         const_iterator begin()const {
169               const_iterator i;
170               i.it=datos.begin();
171               return i;
172         }
173
174         const_iterator end()const {
175               const_iterator i;
176               i.it=datos.end();
177               return i;
178         }
179   };
```

□

5.2.3 Contendores asociativos no ordenados

Este tipo de contenedores sirven para representar las tablas hash. La clasificación de estos contenedores se hará: 1) dependiendo si admiten valores repetidos o no; 2) y si las claves tienen valores asociados

La elementos se encuentran en un orden particular, y la recuperación de los mismo se hacen por su valor de una forma muy rápida. En un conjunto no ordenado el valor de un elemento es a su vez su llave. Este valor una vez insertado en el conjunto no ordenado no puede modificarse. Solamente podemos insertar, consultar y eliminar. Internamente los elementos no esta ordenados, pero se organizan en cubetas dependiendo del valor hash asociado. Por lo tanto este tipo de contenedor es el mas eficiente para acceder a elementos individuales pero no es eficiente cuando se quiere consultar un rango de valores. Los datos se almacenan en cubetas. Todos aquellos datos que tengan la misma función hash se almacenan en la misma cubeta. Por lo tanto ocurre colisión cuando la función hash para dos claves diferentes devuelve la misma cubeta.

Las funciones mas relevantes de estos contendores son:

- Funciones de capacidad:
  - *empty*: comprueba si el contenedor esta vacío
  - *size*: devuelve el tamaño del contenedor
  - *max_size*: devuelve el máximo tamaño del contendor
- Iteradores
  - *begin*: devuelve un iterador al principio del contenedor
  - *end*: devuelve un iterador al final
  - *cbegin*: devuelve un iterador constante al principio del contenedor
  - *cend*: devuelve un iterador constante al final
- Consulta
  - *find*: obtiene un iterador el al elemento
  - *count*: nos da el numero de elementos con un valor determinado (0 o 1).
  - *equal_range*: consigue un rango de elemento con una llave especifica.

- Modificadores
  - *insert*:inserta elementos
  - *erase*: elimina elementos
  - *clear*: limpia el contenido
  - *swap*: intercambia el contenido
- Cubetas
  - *bucket_count*: devuelve el número de cubetas
  - *max_bucket_count*: devuelve el número máximo de cubetas
  - *bucket_size*: devuelve el tamaño de la cubeta
  - *bucket*: localiza la cubeta de un elemento
- Aspectos de la función hash
  - *load_factor*: devuelve el factor de carga.
  - *max_load_factor*: maximo factor de carga.
  - *rehash*: modifica el numero de cubetas
  - *reserve*: solicita un cambio de capacidad
- Observadores
  - *hash_function*: obtiene la funcion hash
  - *key_eq*: toma dos elementos y devuelve un booleano indicadno si lo elementos son equivalentes porque tienen la misma funcion hash

Factor de Carga:*Razón entre el numero de elementos del contendor y el número de cubetas (valor obtenido con la función bucket_count).*
Hay que tener en cuenta que el factor de carga afecta a la probabilidad de colisión en la tabla hash (probabilidad de que dos elementos estén localizados en la misma cubeta). Así el contenedor usa el valor *max_load_factor* como el umbral para forzar un incremento en el número de cubetas y de esta forma procediendo a aplicar un *rehashing*.

Unordered_set/Unordered_multiset

Son contenedores que almacenan claves no repetidas (unordered_set) o si permiten claves repetidas tenemos el contenedor unordered_multiset. Para poder usar estos contenedores debemos hacer el include de la biblioteca *unordered_set*.

Ejemplo 5.2.3
En este ejemplo se muestra como inicializar un unordered_set y especialmente veremos como localizar un elemento usando la función *find*.

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_set>
4  int main ()
5  {
6  std::unordered_set<std::string> myset = { "red","green","blue" };
7  std::string input;
8  std::cout << "color: ";
```

```
 getline (std::cin,input);
//find devuelve un const_iterator ya
// que no podemos modificar el elemento
 std::unordered_set<std::string>::const_iterator got = myset.find (input);

 if ( got == myset.end() )//no lo hemos encontrador
 std::cout << "no esta el elemento "<<input;
 else
 std::cout << *got << "si esta el elemento"<<input;
 std::cout << std::endl;
 return 0;
 }
```

□

Ejemplo 5.2.4
En este otro ejemplo se hace uso de la función *count*. También hacer especial interes en la variable *auto* en el for para recorrrer los elementos de un conjunto. Para poder compilar este código deberemos hacerlo usando *-std=c++11*.

```
#include <iostream>
#include <string>
#include <unordered_set>
 int main ()
 {
  std::unordered_set<std::string> myset = { "hat", "umbrella", "suit" };

  //for con una variable auto
  for (auto& x: {"hat","sunglasses","suit","t-shirt"}) {
        if (myset.count(x)>0)
        std::cout << "myset tiene " << x << std::endl;
        else
        std::cout << "myset no tiene " << x << std::endl;

  }
  return 0;

}
```

□

Ejemplo 5.2.5
En este ejemplo se revisan las funciones: *insert, erase, clear, size, bucket_count, load_factor, rehash, hash_function*.

```cpp
#include <iostream>
#include <string>
#include <array>
#include <unordered_set>
 int main ()
 {
 std::unordered_set<std::string> myset = {"yellow","green","blue"};
 std::array<std::string,2> myarray = {"black","white"};
 std::string mystring = "red";

 myset.insert (mystring);
 //insertando reddish
 myset.insert(mystring+"dish");
 //insertando elementos del array
 myset.insert(myarray.begin(), myarray.end());
 //insertando desde un conjunto constante
 myset.insert( {"purple","orange"} );

 std::cout << "myset contiene:";
 for (const std::string& x: myset) std::cout << " " << x;
 std::cout << std::endl;

 myset.clear();//vaciamos el conjunto
 myset.rehash(12); //reserva un numero como minimo de 12 cubetas aunque
                   //realmente reserva el siguiente primo 13.

 myset.insert({"USA","Canada","France","UK","Japan","Germany","Italy"});
 myset.erase ( myset.begin() );// eliminacion por iterador

 myset.erase ( "France" );// eliminacion por valor

 myset.erase ( myset.find("Japan"), myset.end() ); //eliminacion por rango

 std::cout << "myset contiene:";
 for ( const std::string& x: myset ) std::cout << " " << x;
 std::cout << std::endl;


 std::cout<<"size = " << myset.size() << std::endl;//no. elementos
 std::cout<<"bucket_count = " << myset.bucket_count() << std::endl;//no. cubetas
 std::cout<<"load_factor = " << myset.load_factor() << std::endl;//factor de carga
 //maximo factor de carga
 std::cout<< "max_load_factor = " << myset.max_load_factor() << std::endl;
```

```
44
45  //como se usa la hash_function
46  stringset::hasher fn = myset.hash_function();
47  std::cout << "purple tiene funcion hash: " << fn ("purple") << std::endl;
48  std::cout << "orange tiene funcion hash: " << fn ("orange") << std::endl;
49 return 0;
50
51  return 0;
52 }
```

□

Unordered_map/Unordered_multimap

Almacenan elementos formados por la combinacion valor clave y valor asociado. Estos contenedores son aconsejables cuando se necesita búsquedas rápidas por clave. Al igual que en los unordered_set los pares clave valor asociado, se almacenan internamente en la cubeta determinada por el valor hash asociado a la clave. Las funciones que hemos visto anteriormente para Unordered_set/Unordered_multiset son aplicables tambien para Unordered_map/Unordered_multimap.

Ejemplo 5.2.6
Un pequeño ejemplo para definir un unordered_map y ver su contenido y borrar elementos.

```
1  // unordered_map::erase
2 #include <iostream>
3 #include <string>
4 #include <unordered_map>
5 int main ()
6 {
7   std::unordered_map<std::string,std::string> mymap;
8
9  // Iniciando los valores
10   mymap["U.S."] = "Washington";
11   mymap["U.K."] = "London";
12   mymap["France"] = "Paris";
13   mymap["Russia"] = "Moscow";
14   mymap["China"] = "Beijing";
15   mymap["Germany"] = "Berlin";
16   mymap["Japan"] = "Tokyo";
17
18   mymap.erase ( mymap.begin() );// erasing by iterator
19
20   mymap.erase ("France");// erasing by key
21
22   mymap.erase ( mymap.find("China"), mymap.end() ); //borrando por rango
```

```
23
24    // contenido
25    std::unordered_map<std::string,std::string>::iterator it;
26    for (it= mymap.begin(); it!=mymap.end();++it) )
27      std::cout << it->first << ": " << it->second << std::endl;
28
29
30   // mostrar el contenido por cubetas
31    std::cout << "mymap lista por cubetas:\n";
32    for ( unsigned i = 0; i < mymap.bucket_count(); ++i) {
33
34     std::cout << "bucket #" << i << " contiene:";
35     for ( auto local_it = mymap.begin(i); local_it!= mymap.end(i); ++local_it )
36      std::cout << " " << local_it->first << ":" << local_it->second;
37     std::cout << std::endl;
38    }
39
40  return 0;
41
42  }
```

□

Ejemplo 5.2.7

El siguiente ejemplo muestra un diccionario usando para su representación un *unordered_map* de pares string,string. El primer string es para la palabra y el segundo para la definición. Las operaciones que se han definido son:

- *operador []*: para consultar una palabra
- *find*: para buscar una palabra.
- *insert*: para insertar una nueva palabra junto con su definición
- *erase*: para borrar un palabra.

```
1  #include <unordered_map>
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  class Diccionario{
6
7
8   private:
9   //string primero palabra, string segundo definicion
10    unordered_map<string,string> datos;
11
```

```
12  public:
13    /**
14    * @brief obtiene la definicion de una palabra
15    * @param pal: palabra a buscar
16    */
17    string & operator[]( const string &pal){
18       return datos[pal];
19     }
20
21   //iterator
22   class const_iterator ;//declaracion adelatanda
23   class iterator {
24    private:
25         unordered_map<string,string>::iterator it;
26   public:
27      iterator operator ++(){
28        ++it;
29        return *this;
30      }
31      pair<const string,string> & operator *(){
32          return *it;
33      }
34      bool operator==(const iterator &i){
35         return i.it==it;
36      }
37      bool operator!=(const iterator &i){
38         return i.it!=it;
39      }
40      friend class Diccionario;
41      friend class const_iterator;
42   };
43
44   class const_iterator {
45    private:
46         unordered_map<string,string>::const_iterator it;
47   public:
48      const_iterator (){}
49
50      const_iterator (const iterator &i):it(i.it){}
51
52      const_iterator operator ++(){
53        ++it;
54        return *this;
```

```
55        }
56        const pair<const string,string> & operator *(){
57            return *it;
58        }
59        bool operator==(const const_iterator &i){
60           return i.it==it;
61        }
62        bool operator!=(const const_iterator &i){
63           return i.it!=it;
64        }
65        friend class Diccionario;
66
67    };
68
69
70    /**
71     * @brief devuelve un iterador a la palabra y su definicion
72     * @param pal: palabra a buscar
73     * @note si no esta devuelve end
74     */
75
76
77    const_iterator find(const string &pal)const{
78       //usamos find del unordered_map
79       unordered_map<string,string>::const_iterator got = datos.find (pal);
80       //ahora definimos un iterador de Diccionario
81       const_iterator i;
82       i.it=got;
83       return i;
84    }
85
86    /**
87     * @brief devuelve el numero de elementos del diccionario
88     */
89    int size()const{ return datos.size();}
90
91    /**
92     @brief inserta una palabra con su definicion
93     @param pal: palabra
94     @param def: definicion de la palabra
95     */
96    pair<iterator,bool> insert( string pal, string  def){
97         pair<string,string> a(pal,def);
```

```
98          pair<unordered_map<string,string>::iterator,bool> it= datos.insert(a);
99          iterator i;
100         i.it=it.first;
101         return pair<iterator,bool>(i,it.second);
102     }
103
104     /**
105      @brief borra una palabra
106      @param pal: palabra a borrar
107     */
108     void erase(const string &pal){
109        datos.erase(pal);
110     }
111
112
113     /**
114      * @brief inicializa los iteradores al principio
115      */
116     iterator begin(){
117       iterator i;
118       i.it=datos.begin();
119       return i;
120     }
121     const_iterator begin()const{
122        const_iterator i;
123        i.it =datos.begin();
124        return i;
125     }
126
127     /**
128      * @brief inicializa los iteradores al final
129      */
130
131     iterator end(){
132       iterator i;
133       i.it=datos.end();
134       return i;
135     }
136     const_iterator end()const{
137        const_iterator i;
138        i.it =datos.end();
139        return i;
140     }
```

```
};
```

Un ejemplo de uso de Diccionario sería el siguiente

```
int main(){
   Diccionario d;
   //insertamos elementos
   d.insert("gato","mamifero felino");
   d.insert("perro","mamifero canino");
   d.insert("gorrion","ave del mediterraneo");
   d.insert("pinguino","ave polar");

   //No sale ordenado por clave
   //ya que el orden es propio
   Diccionario::iterator i;
   for (i=d.begin(); i!=d.end();++i)
      cout<<(*i).first<<" "<<(*i).second<<endl;

   //ejemplo de find
   Diccionario::const_iterator i2;
   i2= d.find("pinguino");

   cout<<"Definicion de pinguino "<<(*i2).second<<endl;

   cout<<"Definicion de perro: "<<d["perro"]<<endl;

   if (d.find("perro")!=d.end()){
      cout<<"La entrada perro esta"<<endl;;
   }
   else cout<<"La entrada perro no esta"<<endl;

   //ejemplo de borrado
   d.erase("perro");
   if (d.find("perro")!=d.end()){
      cout<<"La entrada perro esta"<<endl;;
   }
   else cout<<"La entrada perro no esta"<<endl;
}
```

□