Tema 9.- Standard Template Library (STL) de C++

Introducción

La Standard Template Library (STL) es una colección de estructuras de datos genéricas y algoritmos escritos en C++. STL no es la primera de tales librerías, así la mayor parte de los compiladores de C++ disponen (o disponían) de librerías similares y, también, están disponibles varias librerías comerciales. Uno de los problemas de estas librerías es que son mutuamente incompatibles, lo que obliga a los programadores a aprender nuevas librerías y a migrar de un proyecto a otro y de uno a otro compilador. Sin embargo, STL ha sido adoptado por el comité ANSI de estandarización del C++, lo que significa que está soportado como una extensión más del lenguaje por todos los compiladores.

El diseño de la Standard Template Library es el resultado de varios años de investigación dirigidos por Alexander Stepanov y Meng Lee de Hewlett-Packard, y David Musser del Rensselaer Polytechnic Institute. Su desarrollo se inspiró en otras librerías orientadas a objetos y en la experiencia de sus creadores en lenguajes de programación imperativos y funcionales, tales como Ada y Scheme.

Componentes de STL

La librería STL dispone de cinco componentes diferentes:

Algoritmos	la STL proporciona una amplia variedad de algoritmos comunes entre los que se incluyen los de ordenación, búsqueda y algoritmos numéricos. Los algoritmos se pueden utilizar con estructuras de datos de la librería, vectores, o estructuras de datos definidas por el usuario y provistas de iteradores
Iteradores	una generalización del puntero que permite al programador visitar cada elemento de una estructura de datos contenedora.
contenedores	estructuras de datos que contienen, y permiten manipular, otras estructuras de datos.
funciones objeto	varios de los algoritmos proporcionados por la STL, permiten pasar una función al algoritmo para adecuar su funcionalidad. Las funciones objeto son una generalización del puntero a la función del C
adaptadores	toman un contenedor y le proporciona una *interface* diferente. Esto permite transformar una estructura de dato en otra que tiene las mismas facilidades pero cuya *interface* proporciona un conjunto de operaciones diferente.

Tipos de contenedores

La STL proporciona una colección de estructuras de datos contenedoras y algoritmos genéricos que se pueden utilizar con éstas. Una estructura de datos se dice que es contenedora si puede contener instancias de otras estructuras de datos. En concreto, la STL dispone de las estructuras indicadas en la tabla siguiente:

Contenedores lineales	Contenedores asociativos	Contenedores adaptados
Vector *vector<T>*	**Conjunto** *set<T, Compare>*	**Pila** *stack<Container>*
Lista *list<T>*	**Multiconjunto** *multiset<T, Compare>*	**Cola** *queue<Container>*
Doble cola *deque<T>*	***Mapa*** *map<Key, T, Compare>*	**Cola de prioridad** *priority_queue<Container, Compare>*
	Multimapa *multimap<Key, T, Compare>*	

Estos contenedores se dice que son genéricos porque pueden contener instancias de cualquier otro tipo de dato, para lo que utilizan de forma extensiva los *templates* (plantillas) de C++.

Los contenedores se dividen en tres categorías:

- **Contenedores lineales**. Almacenan los objetos de forma secuencial permitiendo el acceso a los mismos de forma secuencial y/o aleatoria en función de la naturaleza del contenedor.
- **Contenedores asociativos**. En este caso cada objeto almacenado en el contenedor tiene asociada una clave. Mediante la clave los objetos se pueden almacenar en el contenedor, o recuperar del mismo, de forma rápida.
- **Contenedores adaptados**. Permiten cambiar un contenedor en un nuevo contenedor modificando la *interface* (métodos públicos y datos miembro) del primero. En la mayor parte de los casos, el nuevo contenedor únicamente requerirá un subconjunto de las capacidades que proporciona el contenedor original.

La adopción de STL ofrece varias ventajas:

- Al ser estándar, está disponible por todos los compiladores y plataformas. Esto permitirá utilizar la misma librería en todos los proyectos y disminuirá el tiempo de aprendizaje necesario para que los programadores cambien de proyecto.
- El uso de una librería de componentes reutilizables incrementa la productividad ya que los programadores no tienen que escribir sus propios algoritmos. Además, utilizar una librería libre de errores no sólo reduce el

tiempo de desarrollo, sino que también incrementa la robustez de la aplicación en la que se utiliza.

- Las aplicaciones pueden escribirse rápidamente ya que se construyen a partir de algoritmos eficientes y los programadores pueden seleccionar el algoritmo más rápido para una situación dada.
- Se incrementa la legibilidad del código, lo que hará que éste sea mucho más fácil de mantener.
- Proporciona su propia gestión de memoria. El almacenamiento de memoria es automático y portátil, así el programador ignorará problemas tales como las limitaciones del modelo de memoria del PC.

Uso de los contenedores en programas

A la hora de usar los diferentes tipos de contenedores para su declaración, usar su constructor y aplicar la multitud de funciones miembro que poseen, hay que declarar una serie de ficheros de cabecera, para los que su nomenclatura puede variar de unos compiladores a otros, en el Borland C++, que es el compilador con el que trabajamos, estos ficheros de cabecera son los siguientes:
(Fijarse que en los compiladores C++ no es obligatorio poner.h)

Tipo de contenedor	Archivo include
Vector	<vector>
List	<list>
Deque (cola de doble entrada)	<deque>
Set y multiset	<set>
Map y multimap	<map>
Stack (pilas)	<stack>+contenedor base
Queue	<queue>+contenedor base
Priority-queue(colas con proridad)	<queue>+contenedor base
Algortimos genéricos buscar ordenar, rellenar etc.	<algorith>

Iteradores

Antes de empezar a ver la declaración y uso de los contenedores, vamos a estudiar un objeto útil y necesario para “movernos” por ellos, este objeto es el iterador.
El iterador debemos entenderlo como un puntero, específico para cada clase de contenedor, que hay que declararlo asociado al tipo de contenedor, en cuestión, esto es un contenedor vector tendrá un iterador que no vale para un iterador de tipo mapa ó multimapa.

Los contenedores son estructuralmente muy diferentes, lo que determina la forma en que se accede a los elementos o nos desplazamos de uno a otro. Existen distintos tipos de iteradores, como veremos a continuación. Todos definen como mínimo lo siguiente:

- Un constructor copia.
- Un operador de asignación.

- Un operador de igualdad (==) y desigualdad (!=) para comprobar si dos iteradores apuntan al mismo elemento.

Tipos de iteradores

Iteradores de lectura *(InputIterator)*

Este tipo de permite avanzar hacia delante o leer el elemento situado en la posición actual. Las operaciones definidas son:

- Operador de dereferencia (*), para obtener el dato apuntado.
- Operador de incremento (++).

Iteradores de escritura *(OutputIterator)*

Este tipo de iteradores permite el avance hacia delante y la modificación de los elementos apuntados.

- Operador de dereferencia (*), que permita la asignación. Si x es un iterador, debe admitir la operación *x = t.
- Operador de incremento (++).

Iteradores con avance *(ForwardIterator)*

Un iterador con avance es un iterador con la funcionalidad tanto de escritura como lectura.

Iteradores bidireccionales *(BidirectionalIterator)*

Un iterador bidireccional es un iterador de escritura/lectura que puede moverse en los dos sentidos. Define por tanto:

- Operador de decremento (--).

Iteradores de acceso aleatorio *(RandomIterator)*

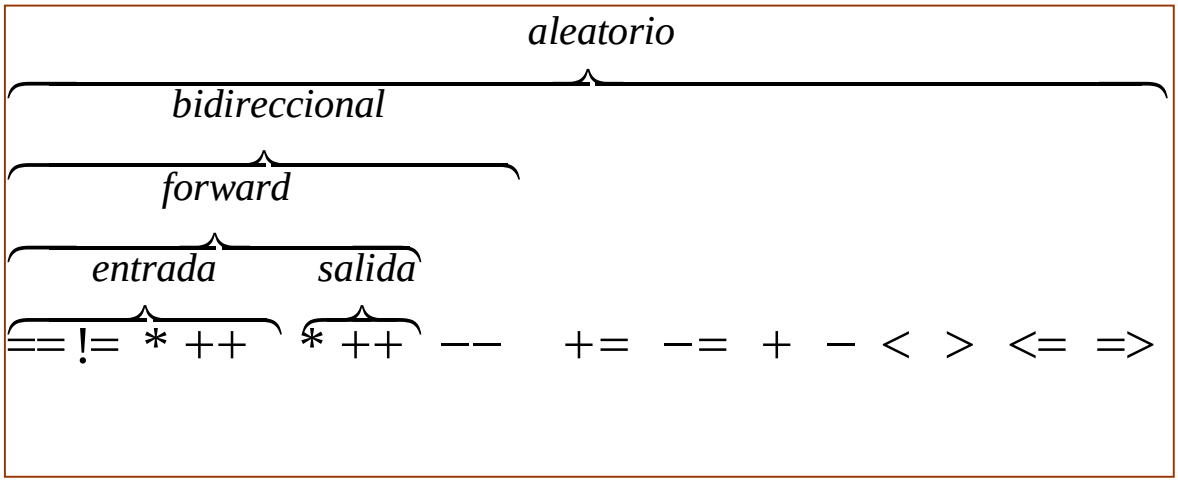
Es el iterador más potente. Permite la lectura y escritura directa de manera directa de cualquier posición. Las operaciones definidas, además de las existentes en *BidirectionalIterator* son:

-Operador suma (+). Dado un entero sin signo n, devuelve un iterador que apunta n posiciones por delante del actual.
-Operador suma con asignación (+=). Hace saltar el iterador el número de posiciones indicadas.
-Operadores resta (-) y resta con asignación (-=) de forma similar.
-Operador corchete ([]). Devuelve el elemento situado en la posición indicada, a partir de la actual. Debe permitir también la asignación.

-Operador de comparación de desigualdad (<). Comprueba si la posición apuntada por un iterador es anterior a la de otro.
Cada contenedor define un iterador adecuado para el acceso a los elementos que almacena, que normalmente será un *BidirectionalIterator* o un *RandomIterator*. Para declarar un iterador asociado a un contenedor, haremos:

contenedor<parametros>::iterator nombreiterador

Ejemplo: list<int>::iterator i; // Iterador para una lista enlazada de enteros



Contenedores

Los contenedores, son pues objetos que pueden almacenar los tipos de datos que se desee, y en función del contenedor elegido, tendremos a nuestra disposición un extenso conjunto de funciones miembro (operaciones) para realizar sobre ellos y algunos algoritmos útiles buscar, ordenar, separar etc.

NOTA GENERAL: Para todos los contenedores existe el destructor Nombre_contenedor.~tipo_contenedor() que borra todos los elementos y libera memoria y que debe usarse cuanto usemos un contenedor temporal del cual no vayamos a hacer mas uso de él, ejemplo m. ~map() ó v. ~vector(), borra y libera memoria del mapa m y del vector v respectivamente.

Contenedor vector

Un vector está implementando como un bloque de memoria contiguo de forma similar a un *array*. Las características de un vector incluyen:

- Inserciones y borrados en tiempo constante al final.
- Inserciones y borrados de coste lineal en posiciones intermedias.
- Gestión automática de memoria.

- Declarar el uso mediante #include<vector>

Cuando se crea el vector se asigna espacio para los elementos y si es necesario se redimensiona, automáticamente, el tamaño del vector.

Operaciones para el tipo de dato vector (T es el tipo de datos que contiene el vector)

Constructores		
vector<T> v;	Constructor por defecto	*O(1)*
vector<T> (int, T)	Constructor con tamaño y valor inicial dados	*O(n)*
vector<T> v (aVector);	Constructor de copia	*O(n)*
Acceso a elementos		
v[i]	Acceso por índice, también puede asignarse	*O(1)*
v.front()	Primer valor de la colección	*O(1)*
v.back()	Último valor de la colección	*O(1)*
Inserción		
v.push_back (T)	Añade un elemento al final del vector	*O(1)*
v.insert (iterator, T)	Inserta un nuevo elementos antes del iterador	*O(n)*
v.swap (vector<T>)	Intercambia valores con otro vector	*O(n)*
Borrado		
v.pop_back ()	Borra el último elemento del vector	*O(1)*
v.erase (iterator)	Borra el elemento indicado por el iterador	*O(n)*
v.erase (iterator, iterator)	Borra un rango de valores	*O(n)*
Tamaño		
v.capacity ()	Número máximo de elementos del *buffer*	*O(1)*
v.size ()	Número de elementos en el vector	*O(1)*
v.resize (unsigned, T)	Cambia el tamaño, rellenando con un valor	*O(n)*
v.reserve (unsigned)	Pone el tamaño del *buffer*	*O(n)*
v.empty ()	Cierto si el vector está vacío	*O(1)*
Iteradores		
vector<T>::iterator itr	Declara un nuevo iterador	*O(1)*
v.begin ()	Iterador que referencia al primer elemento	*O(1)*
v.end ()	Iterador que referencia al siguiente al último	*O(1)*
vector<T>::reverse_iterator ritr	Declara un nuevo reverse_iterator	
v.rbegin ()	Reverse_iterator que referencia al último elemento	*O(1)*
v.rend ()	Reverse_iterator que referencia al anterior al primero	*O(1)*

Aunque como se puede apreciar la tabla es muy prolija en funciones y miembro solo interesan las marcadas en negrita que son las de uso más habitual para los problemas de gestión que trataremos pero es bueno tenerla presente por si hay que usar alguna en especial para un problema concreto.

Aquí exponemos tres ejemplos los dos primeros para crear y recorrer un vector de enteros tanto con el iterador asociado como con el operador de índice [] y después un vector de estructuras donde los datos de la misma son accesibles mediante iterador->campo_de la_estrcutura

```
#include <vector.h>
#include<stdlib.h>

using namespace std;
main ()
{
  vector<int> v;
  vector<int>::iterator vItr;
  // constructor para un vector de enteros


  /* inicialización del vector v: */
  for (unsigned int i=0; i<50; i++)
    // función miembro push_back que inserta por el final (ver apuntes)
    v.push_back(random(100)+1);

  /* acceso a los elementos mediante el iterador vItr */
   for (vItr = v.begin(); vItr!=v.end(); vItr++)
    cout << *vItr << ' ';
  cout << endl;

}
```

Ejmplo 1 de contendor vector con acceso a los elementos mediante el iterador vitr

```
#include <vector.h>
#include<stdlib.h>

using namespace std;
main ()
{
  vector<int> v;
  vector<int>::iterator vItr;

  /* inicialización del vector v:*/
  for (unsigned int i=0; i<50; i++)
    v.push_back(random(50)+50);

  /* acceso a los elementos mediante el operador de índice []
     y controlando el final mediante la función miembro v.size()*/
    vItr = v.begin();
  for (unsigned int i=0; i<v.size(); i++)
    cout << vItr[i] << "\t";
  cout << endl;

}
```

Ejmplo 2 de contendor vector con acceso a los datos mediante el operador [] de índice y la función miembro v.size()

```
#include <vector.h>
#include<stdlib.h>
#include<string.h>

using namespace std;

typedef struct
{
        char nombre[20];
        int edad;
} persona;
main ()
{
  randomize();
  persona ficha;
  vector<persona> v;
  vector<persona>::iterator vItr;

  // constructor para un vector de estrcuturas

  /* inicialización del vector v */
  for (unsigned int i=0; i<50; i++)
        {
    strcpy(ficha.nombre,"Hello");
    ficha.edad=random(40)+10;
    v.push_back(ficha);
    }


  // los iteradores v.begin y v.end marcan los extremos del vector

  /* acceso a los elementos mediante el iterador vItr */
  //vItr = v.begin();
  for (vItr = v.begin(); vItr!=v.end(); vItr++)
    {
    // VItr es como un puntero luego si accedo a un struct
    // debo hacerlo con el operador ->
    cout << vItr->edad << ' ';
    cout << vItr->nombre << ' '<<endl;
    }
  cout << endl;

}
```

Ejemplo 3 de contenedor vector donde el dato de cada celda es un struct y el acceso es mediante el iterador y el operador -> (flecha) a cada una de sus componentes.

Contenedor List

El tipo de dato list<T> se presenta como una lista doblemente enlazada pudiéndose iterar en ambos sentidos. Entre los diferentes tipos de operaciones que proporcionan las listas destacan

- Inserciones y borrados en tiempo constante al principio y al final.
- Inserciones y borrados en tiempo constante en posiciones intermedias.

- Declarar el uso de #include<list>

Operaciones para el tipo de dato list

Constructores y asignación		
list<T> v	Constructor por defecto	*O(1)*
list<T> l (aList);	Constructor de copia	*O(n)*
l = aList	Asignación	*O(n)*
Acceso a elementos		
l.front()	Primer valor de la colección	*O(1)*
l.back()	Último valor de la colección	*O(1)*
Inserción y borrado		
l.push_front (T)	Añade un elemento al principio de la lista	*O(1)*
l.push_back (T)	Añade un elemento al final de la lista	*O(1)*
l.insert (iterator, T)	Inserta un nuevo elementos antes del iterador	*O(1)*
l.swap (list<T>)	Intercambia valores con otra lista	*O(1)*
l.pop_front ()	Borra el primer elemento de la lista	*O(1)*
l.pop_back ()	Borra el último elemento de la lista	*O(1)*
l.remove(T)	Eliminar todos los elementos iguales a uno dado	*O(n)*
l.remove_if(predicate)	Eliminar todos los valores que cumplan una condición	*O(n)*
l.erase (iterator)	Borra el elemento indicado por el iterador	*O(1)*
l.erase (iterator, iterator)	Borra un rango de valores	*O(1)*
Tamaño		
l.size ()	Número de elementos en la lista	*O(n)*
l.empty ()	Cierto si la lista está vacía	*O(1)*
Iteradores		
list<T>::iterator itr	Declara un nuevo iterador	*O(1)*
l.begin ()	Iterador que referencia al primer elemento	*O(1)*
l.end ()	Iterador que referencia al siguiente al último	*O(1)*
list<T>::reverse_iterator ritr	Declara un nuevo reverse_iterator	
l.rbegin ()	Reverse_iterator que referencia al último elemento	*O(1)*

l.rend ()	Reverse_iterator que referencia al anterior al primero	*O(1)*
Otros métodos		
l.reverse()	Invierte la lista	*O(n)*
l.sort() (elementos únicos)	Ordena los elementos de menor a mayor	*O(n logn*
l.merge(list<T>)	Mezcla con otra lista ordenada	*O(n)*
l.sort(comparision)	Ordena los elementos según una función	*O(n logn*

```
#include <iostream.h>
#include <list.h>
#include<stdlib.h>
using namespace std;
void ver( list<int> l);
void main (void)
{
  list<int> l;

  // meter 10 elementos por el principio
  for (int c = 1; c <= 10; c++)
  {
    l.push_front (random(50)+10);
  }

  cout <<"vemos los 10 numeros\n";
  ver(l);
  // insertamos tres 5
  for (int c = 0; c < 3; c++)
  {
  l.push_front (5);
  }
  cout <<"vemos los numeros\n";
  ver(l);
  l.remove(5);
  cout <<"borramos los 5\n";
  ver(l);
  l.remove (10); // Eliminar el elemento 10 de la lista
  cout <<"borramos el 10\n";
  ver(l);
  l.sort();
  cout <<"ordenamos menor mayor\n";
  ver(l);
}
```

```
void ver( list<int> l)
{
list<int>::iterator il;
il=l.begin();
while (il != l.end ())
  {
    cout << *il << " " ;
    il ++;
  }
  cout <<endl;
}
```

Ejemplo del contenedor list con enteros se usan varias funciones miembro descritas en la tabla precedente concretamente push_front, remove y sort (elemtos únicos). Además hemos construido la función ver donde el parámetro (por valor) es el contador list, es decir se pueden pasar contenedores a funciones, si el contenedor va a ser modificado ha de pasarse por referencia poniendo función(list<int>&l)

El contenedor list , tiene una limitación y es que sólo se puede definir el tipo de datos T, como unitario es decir, int flot char o string , no admite tipos agregados (struct), pero eso se puede salvar haciendo una lista de punteros a struct y manejando dichos punteros, el ejemplo siguiente muestra como hacerlo:

```
#include <list> // The STL list class
#include<stdio.h>
#include <string>
using namespace std;
typedef struct MYSTRUCT
{
        int valor1;
        float valor2;

}NODOS;

void inserta(list<NODOS*>& ,int);
int main ()
{
list<NODOS*> x;
 randomize();

 int t;
 //for (t=1;t<=10;t++)
 //      x.push_front(t);
 for (t=1;t<=5;t++)
        inserta(x,t);
  list<NODOS*>::iterator i;
```

```
// vemos el contenido de la lista
 for (i = x.begin(); i != x.end(); i++)
   {
     cout <<(*i)->valor1<<"--";
   cout << (*i)->valor2 <<"--";

  }

  printf("\n\n\n");

  printf("%d\n",x.size());
  printf("%d\n",x.max_size());

}


void  inserta(list<NODOS*>& x,int val)
{
// asigna el valor val y su mitad (float)
// crea el nodo (NEW)
NODOS* pMySrtuct = new NODOS;
// asigna los valores
          pMySrtuct->valor1 = val;
   pMySrtuct->valor2 = (float)val/2;
// inserta en la lista
          x.push_back(pMySrtuct);
}
```

Ejemplo 4: Uso de una lista con elementos struct, primero creamos el nodo strcut y despues le damos los valores los que insertamos en la lista no es el nodo en si sino su direccion. Para acceder a los datos mediante iteradores fijarse que ahora no es i->dato sino que usamos (*i)->dato puesto que i es una dirección de struct

Contenedor cola (deque)

EL contenedor cola muy parecido al vector, maneja los datos contenidos de forma muy similar al vector, y permite acceso aleatorio a los elementos por iterador ó por índice, con la particularidad de que esta "abierto" por ambos extremos es decir se pueden hace inserciones y borrado por ambos extremos, es pues un contenedor útil para hacer colas de espera, para usarlo hay que declarar el uso de la librería deque (**#include <deque>**):

- Provee acceso aleatorio a secuencias de longitud variable
- Las inserciones y borrados al principio de la secuencia son más rápidos
- La interfaz es muy similar a la de los vectores

Operaciones sobre el contenedor deque

Operación	Efecto	
deque<Elem> c	Crea una cola vacía sin elementos	
deque<Elem> c(n,elem)	Crea una cola y la inicializa con n copias del elemento elem	O(n)
c.˜deque<Elem>()	Borra todos los elementos y libera la memoria	

Operación	Efecto	
c.size()	Da el tamaño de elementos contenidos en la cola	
c.empty ()	Devuelve cierto si la cola esta vacía	O(1)
c.max_size()	Devuelve el num máximo de elmtos que puede ser alojados en la cola	
c[idx]	Accede al elemento cuyo índice es idx empezando a contra el primero como 0,1,….	O(1)
c.front()	Devuelve el primer elemento de la cola c (si existe)	O(1)
c.back()	Devuelve el ultimo elmto de la cola c (si existe)	O(1)
c.begin()	Devuelve un iterador al primer elemto de c	O(1)
c.end()	Devuelve un iterador al siguiente al ultimo elemto de c	O(1)

Operación	Efecto	
c.insert (pos,elem)	Inserta una copia de elem en la posición del iterador pos.	O(n)
c. insert (pos,n,elem)	Inserta n copias del elemto elem en la posicion del iterador pos.	O(n)
c.push_back (elem)	Inserta elem al final	O(1)
c.pop_back()	Borra el útimo elmto (sin devolverlo)	O(1)
c.push_front (elem)	Inserta un elem al principio	O(1)
c.pop_front()	Borra el primer elmto (sin devolverlo)	O(1)
c.erase(pos)	Borra el elmto que se encuentra en el iterador pos	O(n)
c.erase (beg,end)	Borra todos los electos desde [beg,end), beg incluido y end excluido	O(1)
c.clear()	Deja la cola totalmente vacía, pero no libera memoria.	

```
#include <iostream>
#include <deque>
#include<conio.h>
#define N 5
using namespace std;
typedef struct dato
{
int p;   int q;
} ficha;
/***************
c.front() y c.back() dan acceso al primer y útlimo miembro de la cola y a sus
elementos si estos fueran una struct de forma que pueden ser modificados es decir
si la cola se llama cola y posee como en nuestro ejemplo los elmtos p y q para
sumarle 10 al componente p del frente seria cola.front().p=cola.front().p+10;
es decir igual que con las variables de una estructura
*****************/
/* este programa crea N colas y a distribuyendo al azar elemtos*/
int main()
{
   randomize();
   int t,z,k;
   // creamos las n colas
   deque <ficha>cola[N];
   ficha elmto;
  // insertamos 20 elementos entre las colas 0 a N-1
   for (t=1;t<=20;t++)
     {
        //   min sera el num de taquilla para insertar
     int min;
          min=random(N);
```

```
        elmto.p=t*2;
        elmto.q=t;
        cola[min].push_back(elmto);
        cout <<"presentamos longitudes de colas y el elmto primero y ultimo\n";
        for(z=0;z<N;z++)
                {
            cout << "cola numero :"<<(z+1)<<"--";
        cout <<"longitud:"<<cola[z].size();
        cout<< ":primero:";
        if(!cola[z].empty())
                cout <<cola[z].front().p<<"::"<<cola[z].front().q;
                                cout<< ":ultimo:";
        if(!cola[z].empty())
                cout <<cola[z].back().p<<"::"<<cola[z].back().q;
        cout <<"\n";
        }
       getch();
    }
}// fin del main
```

Ejemplo 5: Uso de colas en este caso se crea un vector de colas de tamaño N y se asignan valores en el numero de cola escogido al azar

Contenedor map y multimap

- Un map es una secuencia de pares (clave, valor) ordenados de menor a mayor en función de la clave. Las claves son únicas, esto es, cada clave tiene asociado un sólo valor.
- El mutimap es lo mismo paro las claves no son únicas admiten repeticiones
- El tipo de iterador sobre esta clase de contenedores es **bidireccional en orden de claves**.
- Los componentes de un map son elementos de la clase pair<T1,T2>, que tiene las siguientes características T1=clave; T2 valor asociado a T1
- Las claves pueden ser cualquier tipo de datos ordenable (los elementos se pueden comparar. Ej. String)
- Los valores pueden ser cualquier tipo
- Eficiente inserción, remoción, prueba de inclusión O(log N)
- Basado en árboles binarios balanceados
- Para declara un mapa en nuestro código se hace del siguiente modo (constructor)
- map<dni, nombre, less<dni> > nombre_del_mapa;
- multimapa<dni, nombre, less<dni> > nombre-del _multimapa;
- Para usar map y multimap declarar el uso de <map.h>

Operaciones para el tipo de dato pair

Constructores y asignación		
pair<T1,T2> p	Constructor por defecto	*O(1)*
pair<T1,T2> p (T1,T2)	Constructor con dos argumentos	*O(1)*
pair<T1,T2> p(aPair)	Constructor de copia	*O(1)*
Área de datos (públicos)		
T1 first	Nombre de la variable donde se guarda el primer elemento del par (se accede por iterador)	
T2 second	Nombre de la variable donde se guarda el segundo elemento del par (se accede por iterador)	

Operaciones sobre contenedor map y multimap

K=key clave del mapa;v=valor asociado a la clave K, contenedor asociativo

Constructores y asignación		
map<K, V,less<K>/greater<K> > m; **K y V son los tipos**	**Constructor por defecto; la declaración less ó greater especifican si el orden del mapa es de menor a mayor o de mayor a menor**	***O(1)***
multimap<K, V, less<K>/greater<K> > m	**Constructor por defecto(multimapa)**	***O(1)***

map<K, V> m (aMap)	Constructor de copia	*O(n)*
multimap<K, V>m (aMultimap)	Constructor de copia	*O(n)*
m = aMap	Asignación	*O(n)*
Inserción y borrado		
m[key] (sólo para map)	**Devuelve una referencia al valor asociado a key**	***O(log n)***
m.insert(pair<K,V>(valor1,valor2))	**Inserta un par con clave y valor dados, que son valor1 y valor2**	***O(log n)***
m.erase(key)	**Borra valor con clave asociada key**	***O(log n)***
m.erase(iterator)	**Borra el valor al que apunta un iterador**	***O(log n)***
Métodos de búsquedas		
m.size ()	Número de elementos de la colección	*O(n)*
m.empty ()	**Cierto si la colección está vacía**	***O(1)***
m.count(key)	**Número de elementos con clave key**	***O(log n)***
m.find(key)	**Iterador sobre elemento con una clave dada, devuelve un iterador al elemento encontrado o m.end si no se encuentra**	***O(log n)***
m.lower_bound(key)	**Iterador sobre la primera ocurrencia de un elemento con clave key**	***O(log n)***
m.upper_bound(key)	**Iterador sobre el siguiente elemento después del último con clave key**	***O(log n)***
m.equal_range(key) (sólo para multimap)	**Par de iteradores formado por lower_bound y upper_bound**	***O(log n)***
Iteradores		
map<K, V>::iterator itr	**Declara un nuevo iterador**	***O(1)***
m.begin ()	**Iterador que referencia al primer elemento**	***O(1)***
m.end ()	**Iterador que referencia al siguiente al último**	***O(1)***
map<K, V>::reverse_iterator ritr	Declara un nuevo reverse_iterator	
m.rbegin ()	Reverse_iterator que referencia al último elemento	*O(1)*
m.rend ()	Reverse_iterator que referencia al anterior al primero	*O(1)*

```
#include <iostream.h>
#include <map.h>
#include <string>
#include<stdio.h>
using namespace std;
// variable clave K
typedef unsigned long dni;
// variable datos asociada a clave K
typedef string nombre;
// definicion del mapa y su criterio
typedef map<dni, nombre, less<dni> > mapa;
// los typedef son para ahorro de nombres en las definiciones
void main (void)
{
  int t;
  dni campo1;

  // creamos mapa vacio
  mapa m;
  nombre valores[5]={"luis","eva","jose","antonio","carmen"};

  for (t=1;t<=15;t++)
       {
       // insertamos mediante la función miembro m.insert
       m.insert(pair<dni,nombre>(random(50000000)+10000000,valores[random(5)] ));
       }
  mapa::iterator i;
```

```
  // visualizamos el mapa
  // fijarse que (*i).first es el primer dato del par
  // fijarse que (*i).second es el segundo dato del par
  for (i = m.begin (); i != m.end (); i ++)
    cout << "--clave--"<<(*i).first << "--valor--" << (*i).second << endl;
  flushall();

    // Búsqueda de un valor segun su clave
  cout << " dame clave a buscar(DNI)\n" ;
  cin >> campo1;

  // búsqueda
  i = m.find (campo1);

  // resultado de la búsqueda
  if (i != m.end ())
    cout << (*i).second << endl;
  else
        cout << "no esta\n";
}// fin del main
```

Ejmplo 6: Uso de un mapa formado por DNI y NOMBRE ,ordenado ascendente, se usan las funciones miembro: m.find(), m.insert(),m.end() y los constructores pair<k,v> y map<k,v,criterio>

Para usar el hecho diferencial del los multimapas, cual es el de claves repetidas hay que hacer especial mención a las funciones miembro:

m.lower_bound(key)	**Iterador sobre la primera ocurrencia de un elemento con clave key**	***O(log n)***
m.upper_bound(key)	**Iterador sobre el siguiente elemento después del último con clave key**	***O(log n)***
m.equal_range(key)	**Par de iteradores formado por lower_bound y upper_bound**	***O(log n)***

Para buscar todas las ocurrencias de una clave dada lo mejor es primero usar la función m.count(clave) si nos da !=0 es que dicha clave existe y para recorrer todos sus valores, usamos m.lower_bound(K) y m.upper_bound(K) , que nos apuntan a la primera y la siguiente a la última ocurrencia de la clave K sabiendo que están ordenadas y por tanto contiguas dentro del mapa.

```
// iteradores pri,ult,i
   // respectivamente primero , y ultimo a recorrer e i para viajar entre ellos
   multimap<proyecto, dni, less<proyecto> >::iterator pri,ult,i;


cout <<"que valor quieres buscar (0=salida).-\n";


 cin >> buscar ;
   while (buscar!=0)
   {
          int valor=m.count(buscar);
          cout << "el numero de claves de:" <<buscar<<" es :" << valor << endl;
          if (valor==0)
          // no existen calves de valor buscar
                    cout<<"no esta\n";
          else
        {
```

```
// calculo la primera entrada de buscar y la siguiente a la ultima


      pri=m.lower_bound(buscar);
        ult=m.upper_bound(buscar);
                              for (i = pri;i!=ult;i++)
                    {
                              cout << "para la clave "<<(*i).first << "el valor de dir es: " <<
(*i).second << endl;
                    }
        }
          cout <<"que valor quieres buscar (0=salida).-\n";
          cin >> buscar ;
   }
```

Ejemplo 7:

Usamos m.lower_bound y m.upper_bound , no se ponen dentro del for puesto que cada vez que pasara por él tendría que ir al mapa a calcular los valores que serán fijos durante todo el recorrido por ello es que se calculan fuera en los iteradores pri y ult y después se usan