FACULTAD DE INGENIERÍA
UNIVERSIDAD NACIONAL DE ENTRE RÍOS
ARGENTINA

Computación II - Programación Avanzada

Biblioteca de Plantillas Estándar de C++ (STL)

Versión 04
2012-Mayo-07

Introducción

La STL (del inglés *Standard Template Library*) es una biblioteca de clases y funciones *templates* creada para estandarizar y optimizar la utilización de algoritmos y estructuras de datos en el desarrollo de software en C++. La adopción de esta biblioteca posee grandes ventajas: al ser estándar está disponible en todos los compiladores y plataformas; está libre de errores, por lo tanto se ahorrará tiempo en depurar el código; proporciona su propia gestión de memoria.

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.

La biblioteca presenta tres componentes básicos: *contenedores*, *iteradores* y *algoritmos*. Los contenedores son los objetos capaces de almacenar otros objetos, cada uno de una forma particular. Representan a las estructuras de datos usuales, como los arreglos lineales o las listas enlazadas. Además éstos presentan otras características adicionales que los hacen más potentes. Por ejemplo: pueden aumentar el número de elementos que almacenan en forma dinámica; al ser templates, pueden alojar cualquier tipo de dato o clase; casi todos los contenedores proveen iteradores, lo cual, como se verá luego, permite que los algoritmos que se aplican a ellos sean más eficientes. Los iteradores son objetos a través de los cuales se puede acceder a los elementos del contenedor. El concepto de iterador es similar al de un puntero, sólo que al ser una clase provee mayores utilidades que éste. Pero la gran utilidad de los iteradores ocurre cuando son utilizados por los algoritmos. En la biblioteca existen más de setenta algoritmos para aplicar sobre los contenedores a través de los iteradores. Hay algoritmos de búsqueda, de ordenamiento, de transformación, matemáticos, etc.

Para lograr comprender con mayor facilidad el funcionamiento de los componentes de la librería es necesario que el lector posea sólidos conocimientos en otros temas del lenguaje, a saber: funciones, punteros, clases, objetos, herencia, templates, sobrecarga de operadores y manipulación de archivos.

En este apunte se presenta una descripción de los componentes más importantes de la STL junto con una gama de ejemplos que permitirán visualizar su funcionamiento y la conexión que existe entre ellos.

Contenedores

Son una colección de las estructuras de datos más populares utilizadas habitualmente. Un contenedor es justamente eso: un lugar en donde contener o agrupar objetos del mismo tipo. La diferencia entre un contenedor y otro está en la forma en que los objetos son alojados, en cómo se crea la secuencia de elementos y la manera en que los podrá acceder a cada uno de ellos. Éstos pueden estar almacenados en forma contigua en la memoria o enlazados a través de punteros. Esto hace que las estructuras difieran también en la forma en que se accede a los elementos, la velocidad con la cual se insertan o se eliminan estos y en la eficiencia de los algoritmos que se apliquen a ellas.

Como se verá luego, cada una las diferentes estructuras de datos que implementan los contenedores tiene un propósito particular. Estos diferentes diseños de almacenamiento de datos tienen ventajas y desventajas, lo cual produce que algunos contenedores sean más adecuados que otros para para la resolución de un problema en particular.

La Figura 1 proporciona una clasificación de los contenedores de la STL. Éstos se dividen en contenedores de secuencia o lineales y contenedores asociativos. Los de secuencia son **vector**, **list** y **deque**. Los asociativos son **set**, **multiset**, **map** y **multimap**. En esta sección se describirán las operaciones comunes de los contenedores y las estructuras de secuencia solamente. El tratamiento de contenedores asociativos se realizará luego de haber visto iteradores y algoritmos.

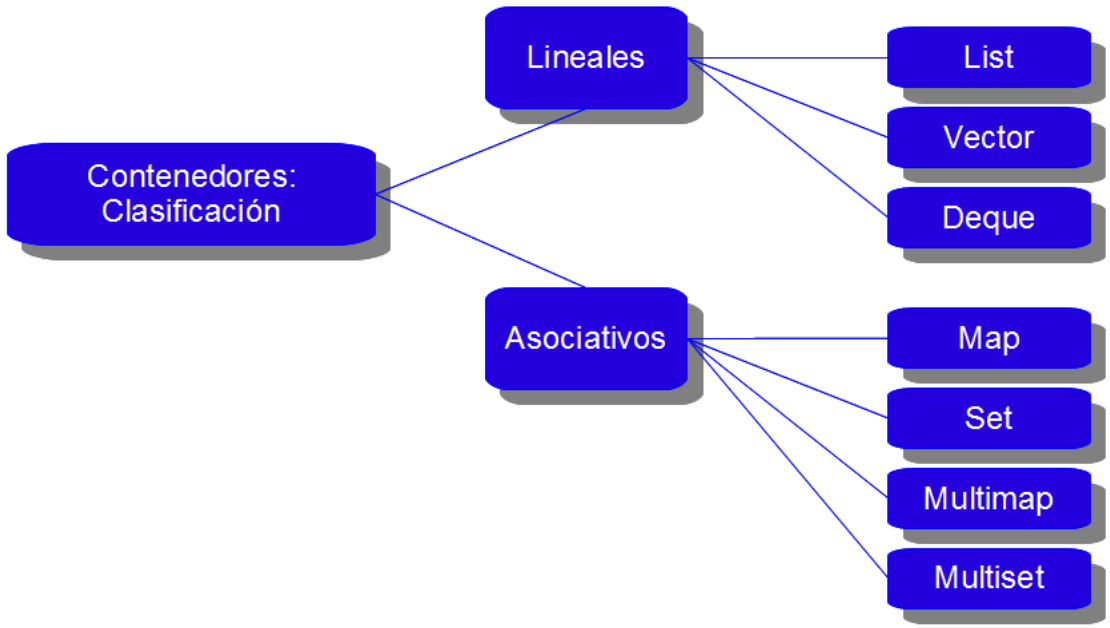

Figura 1: clasificación de los contenedores

Operaciones comunes

Antes de realizar cualquier operación con un contenedor hay que crearlo. La sintaxis utilizada para ello es la siguiente:

```
X<T>  instancia;
```

Donde X representa el tipo de contenedor que se quiere utilizar y T el tipo de dato de los elementos que almacenará la estructura. Así, por ejemplo, para crear un vector de enteros llamado “valores” se escribe:

```
vector<int>  valores;
```

Obsérvese que tanto vector como los demás contenedores son *clases template*. Por lo tanto, al hablar de clases, hablamos también de *constructores*. En la notación previa invocamos al constructor "vacío", es decir, sin parámetros. Esto hace que se cree un contenedor en memoria pero que no contiene aún ningún elemento. Esta es la sintaxis que se utilizará más a menudo, sin embargo, existirán ocasiones en que se necesite crear estructuras auxiliares que sean copias de otras preexistentes:

```
vector<int> aux(valores);
```

Aquí, "aux" es un vector de enteros y, además, es una copia exacta de "valores". En esta ocasión se utiliza el constructor de "copia". También es posible obtener el mismo resultado empleando el operador de asignación "=", como en el ejemplo que sigue:

```
vector<int> aux;
aux = valores;
```

Además de los constructores, los contenedores tienen una serie de operaciones que son comunes a todos ellos, como los de la Tabla 1. Por otra parte existen funciones que se aplican sólo a contenedores lineales, ejemplificados en la Tabla 2.

Tabla 1: operaciones comunes de contenedores

`X::size()`	Devuelve la cantidad de elementos que tiene el contenedor como un entero sin signo
`X::max_size()`	Devuelve el tamaño máximo que puede alcanzar el contenedor antes de requerir más memoria
`X::empty()`	Retorna verdadero si el contenedor no tiene elementos
`X::swap(T & x)`	Intercambia el contenido del contenedor con el que se recibe como parámetro
`X::clear()`	Elimina todos los elementos del contenedor

`v == w` `v != w`
`v < w` `v > w`
`v <= w` `v >= w`
	Supóngase que existen dos contenedores del mismo tipo: *v* y *w*. Todas las comparaciones se hacen lexicográficamente y retornan un valor booleano.

Tabla 2: operaciones comunes de contenedores lineales

`S::push_back(T & x)`	Inserta un elemento al final de la estructura
`S::pop_back()`	Elimina un elemento del final de la estructura
`S::front()`	Devuelve una referencia al primer elemento de la lista
`S::back()`	Devuelve una referencia al último elemento de la lista

Como se mencionó antes, todos los contenedore proveen su propia gestión de memoria. Esto quiere decir que si el contenedor necesita "crecer" o "decrecer" en tamaño (por ejemplo, porque se invocó a la función push_back o pop_back) gestionará automáticamente las modificaciones al espacio en memoria. Cada vez que un contenedor aloca (adquiere) más memoria lo hace con la previsión de que el contenedor podría volver a requerir más memoria pronto, y aloca más memoria de la que necesita en el momento. Por esta razón, la función max_size podría devolver un valor superior al que retorna la función size.

Vector

Representa al arreglo clásico de elementos, en donde todos los elementos contenidos están contiguos en la memoria. Esta característica permite mayor velocidad de acceso a los elementos debido a que para acceder a cualquiera de ellos, sólo se debe calcular la posición relativa al primer elemento.

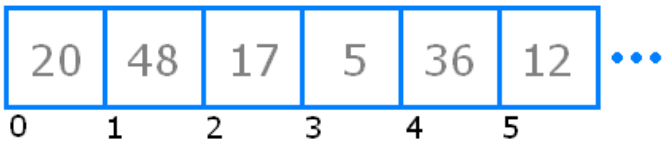

Figura 2: vector de seis elementos

La Figura 2 muestra una representación de un vector en la memoria y los índices de los elementos. En el siguiente ejemplo se muestran algunas operaciones básicas como agregar elementos, ver el contenido e insertar nuevos elementos. Se debe tener en cuenta que para poder utilizar vector es necesario incluir la librería <vector> en el código.

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

int main()
{
 vector<int> datos;
 datos.push_back(20); // se insertan elementos en el final
 datos.push_back(48);
 datos.push_back(17);
 datos.push_back(5);
 datos.push_back(36);
 datos.push_back(12);

 for(unsigned i=0; i<datos.size(); ++i)
  cout << datos[i] << endl; // mostrar por pantalla

 return 0;
}
```

Como se observa en el ejemplo, se puede acceder a los elementos de vector a través de índices: datos[i]. “i” representa la posición en el vector del elemento que se quiere acceder. Así, la salida en pantalla es el listado de los números contenidos en el orden en que fueron ingresados. Por medio de los subíndices también es posible modificar el contenido de un elemento de un vector. Por ejemplo: datos[0] = 4, reemplaza el primer elemento (que tenía el valor 20) por 4. Tener en cuenta que, en este ejemplo, el arreglo es de sólo seis elementos, por lo que el índice máximo es 5. Si se trata de acceder a un elemento inexistente: datos[6], se generará un error de violación de acceso a memoria en tiempo de ejecución.

En términos de eficiencia, los vectors son rápidos insertando o eliminando elementos al final de la estructura. También se pueden insertar elementos en una posición intermedia a través de iteradores pero, debido a que se tiene que mantener la contigüidad de los elementos en la memoria, deberán producirse desplazamientos e inserciones para poder generar el espacio para los elementos a insertar. El tiempo de inserción o eliminación de elementos ubicados en posiciones intermedias es proporcional al tamaño del contenedor.

En el siguiente ejemplo se trabaja sobre un vector de diez letras sobre el cual se aplica un algoritmo de ordenamiento. Además, se observa el uso del constructor que inicializa el vector en una cantidad de elementos (en este caso 10) dada por el parámetro.

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

int main()
{
 vector<char> valores(10); // iniciar con diez elementos

 // llenar con letras mayúsculas al azar
 for(unsigned i=0; i<valores.size(); ++i)
  valores[i] = 'A' + (rand() % 26);

 vector<char> aux(valores); // aux es copia de valores

 // ordenar aux utilizando el método de burbujeo
 for(unsigned i=0; i<aux.size(); ++i)
  for(unsigned j=1; j<aux.size(); ++j)
   if(aux[j] < aux[j-1])
   {
    char c = aux[j];
    aux[j] = aux[j-1];
    aux[j-1] = c;
```

```
   }

 // mostrar por pantalla
 for(unsigned i=0; i<aux.size(); ++i)
  cout << aux[i] << endl;

 return 0;
}
```

Deque

Esta es una estructura de datos que representa a una *cola* con *doble final*. Este contenedor es similar a vector ya que sus elementos también están contiguos en memoria. La diferencia principal radica en que al tener doble final se pueden insertar elementos por ambos extremos del contenedor.

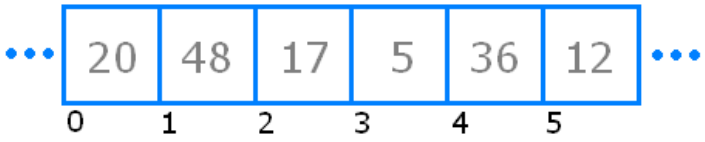

Figura 3: deque de seis elementos

Las deques tienen las mismas funcionalidades que vector, incluso se puede acceder a los elementos a través de subíndices (acceso aleatorio). Además, posee dos funciones más para insertar y eliminar elementos en la parte frontal del contenedor: *push_front(T & x)* y *pop_front()*, respectivamente.

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

int main()
{
 deque<char> datos;

 // cargar algunos datos
 datos.push_front('A');
 datos.push_front('B');
 datos.push_front('C');
 datos.push_back(65);
 datos.push_back('Z');

 // visualizar el contenido
 for( uint i=0; i<datos.size(); ++i )
  cout << datos[i];

 datos.pop_front(); // se elimina el primer elemento
 datos.push_back('C');
 cout << endl;

 for( int i=datos.size()-1; i>=0; --i )
  cout << datos[i];

 return 0;
}
```

Estos contenedores se utilizan cuando se deben insertar o eliminar varios elementos al principio o al final de la estructura. La velocidad de acceso a los elementos es rápida, aunque no tanto como vector. Si se necesitan funcionalidades típicas de arreglos es recomendable utilizar vector en vez de deque.

List

Las listas son los contenedores adecuados cuando se requieren operaciones de inserción o eliminación en *cualquier* parte de la lista. Están implementadas como *listas doblemente enlazadas*, esto es, cada elemento (nodo) contiene las direcciones del nodo siguiente y del anterior, además del valor específico almacenado.

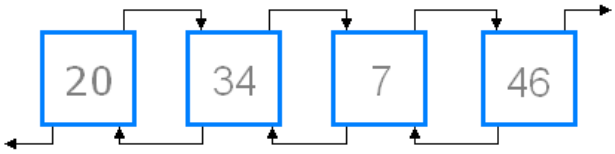

Figura 4: list de cuatro elementos

La ventaja de esta implementación es que la inserción o eliminación de un elemento se reduce a ordenar los punteros del siguiente y anterior de cada nodo. Pero la desventaja es que ya no se puede tener acceso aleatorio a los elementos, sino que se tiene un acceso secuencial en forma bidireccional. Es decir, se puede recorrer el contenedor desde el principio hasta el final o viceversa.

Para poder recorrer listas es necesario utilizar iteradores. Como ya dijimos los iteradores son objetos similares a los punteros, que indican una posición dentro de un contenedor. Todos los contenedores proporcionan dos iteradores que establecen el *rango* del recorrido: *begin* y *end*.

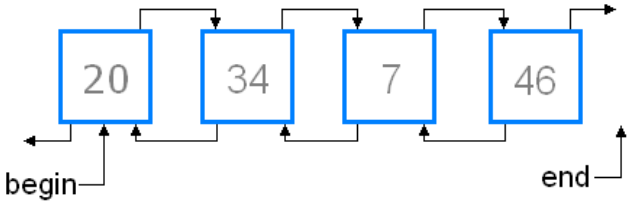

Figura 5: Iteradores de inicio y fin

El primero de ellos apunta al primer elemento de la lista y el segundo a una posición vacía (NULL) después del último elemento de la lista. El siguiente ejemplo muestra cómo se crean y utilizan estos iteradores para recorrer una lista.

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

int main()
{
 list<int> datos;

 // llenar la lista con diez elementos aleatorios
 for( uint i=0; i<10; ++i )
  datos.push_back( rand() % 10 );

 // ordenar la lista
 datos.sort();

 // crear un iterador para listas de enteros llamado p
 list<int>::iterator p;

 // hacer que p apunte al primer elemento de la lista
 p = datos.begin();

 // recorrer la lista incrementando p hasta llegar al final
 while( p != datos.end() )
 {
  // para ver el valor al que apunta p hay desrreferenciarlo
  // igual que a un puntero
  cout << *p << endl;

  // avanzar al siguiente elemento
  p++;
 }

 return 0;
}
```

Es importante notar que en este caso se accede a los elementos de la lista a través de un objeto externo al contenedor (el iterador p), el cual puede apuntar a cualquier elemento en

el contenedor. Al incrementar el iterador (p++) se logra que éste apunte al elemento siguiente en la lista. Después de pasar por el último elemento, el iterador apunta a un lugar vacío, al igual que el iterador “end”, lo que indica el fin del ciclo. Por otro lado, las listas proporcionan un método de ordenamiento *sort*, el cual ordena los elementos de menor a mayor. Esta funcionalidad no está implementada para vector y deque pero más adelante se verá que existe otra alternativa para estos casos.

Iteradores

Entender el concepto de iterador es la clave para comprender enteramente la estructura de la STL y hacer una mejor utilización de ella. Los algoritmos genéricos de esta biblioteca están escritos en términos de iteradores como parámetros y los contenedores proveen iteradores para que sean utilizados por los algoritmos. Estos componentes genéricos están diseñados para trabajar en conjunto y así producir un resultado óptimo.

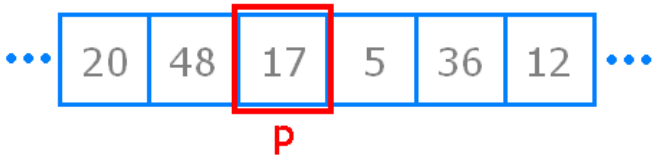

Figura 6: Esquema de un iterador "p"

Un iterador es un objeto que abstrae el proceso de moverse a través de una secuencia. El mismo permite seleccionar cada elemento de un contenedor encapsulando la estructura interna de ese contenedor. Esto permite crear algoritmos genéricos que funcionen con cualquier contenedor, utilizando operaciones comunes como ++, -- o *.

Ya hemos visto el empleo de iteradores con el contenedor list. La sintaxis general para crear un objeto iterador es la siguiente:

```
X::iterator instancia;
```

Donde "X" es el tipo de contenedor al cual estará asociado el iterador. Por ejemplo para crear un iterador a un deque de doubles llamado "inicio" sería:

```
deque<double>::iterator inicio;
```

En esta instancia se utiliza el constructor vacío, es decir que "inicio" no apunta a ningún elemento de ningún contenedor. Pero también es posible crear un iterador utilizando el constructor de copia:

```
deque <double> valores( 10,0 );
deque<double>::iterator inicio( valores.begin() );
deque<double>::iterator inicio2;
inicio2 = valores.begin();
```

Ahora tanto inicio como inicio2 apuntan al mismo lugar que valores.begin(), es decir el primer elemento del contenedor valores.

Clasificación

Los algoritmos genéricos se construyen empleando iteradores que realizan distintas operaciones con ellos. Sin embargo, no todos los iteradores pueden soportar todas las operaciones posibles. Entonces, la clasificación que surge aquí es por la forma en que un iterador puede moverse a través de un contenedor:

Tabla 3: clasificación de iteradores

Forward iterators	Iteradores que pueden avanzar al elemento siguiente
Bidirectional iterators	Pueden avanzar al elemento siguiente o retroceder al anterior
Random acces iterators	Pueden avanzar o retroceder más de una posición de una vez

La siguiente imagen ilustra el conjunto de operaciones que se pueden realizar con estos tipos de iteradores.

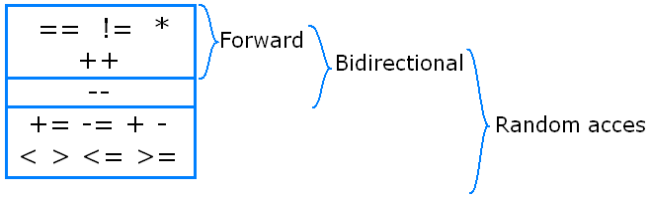

Figura 7: operación con iteradores

También es importante conocer cuáles de estos iteradores proveen los contenedores antes vistos. Estas diferencias ocurren según la estructura interna de cada secuencia. En el caso de las listas doblemente enlazadas sólo se pueden realizar movimientos de avance o retroceso sobre la secuencia, por lo tanto, éstas proveen iteradores bidireccionales. Tanto vector como deque tienen sus elementos contiguos en memoria y permiten "saltar" a las diferentes posiciones sin mayor complicación. Estos contenedores proporcionan iteradores de acceso aleatorio.

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

// función template para mostrar los elementos
// de un contenedor por pantalla utilizando iteradores
template <class Iter>
void MostrarEnPantalla( Iter inicio, Iter final )
{
 while(inicio != final)
  cout << *inicio++ << " ";
}

int main()
{
 vector<char> letras(20); // arreglo de 10 letras

 for(unsigned i=0; i<20; ++i)
  letras[i] = 'A' + (rand() % 26);

 // visualizar el contenido
 MostrarEnPantalla(letras.begin(), letras.end());
 cout << endl;

 // visualizar el contenido en orden inverso
 MostrarEnPantalla(letras.rbegin(), letras.rend());
 cout << endl;

 // visualizar sólo los 10 elementos del medio
 MostrarEnPantalla(letras.begin() + 5, letras.begin() + 15);
 cout << endl;

 return 0;
}
```

Ahora se tiene un ejemplo que emplea iteradores de acceso aleatorio de un vector. La función MostrarEnPantalla recibe como parámetros los iteradores de inicio y fin del rango que se quiere mostrar en la pantalla. Lo nuevo que aparece aquí es la utilización de la función con los parámetros rbegin() y rend(). Estos nuevos iteradores son llamados iteradores inversos (*reverse iterators*), donde el primero de ellos apunta al último elemento del contenedor y el segundo a una posición antes del primero, y al incrementarlos (inicio++) se retrocede en una posición en la estructura. De esta forma se recorre el vector de atrás hacia delante. Por otro lado, al ser de acceso aleatorio se puede sumar posiciones a un iterador y de esa forma saltar a un elemento distante (6° elemento: *begin() + 5*, 16° elemento: *begin() + 15*).

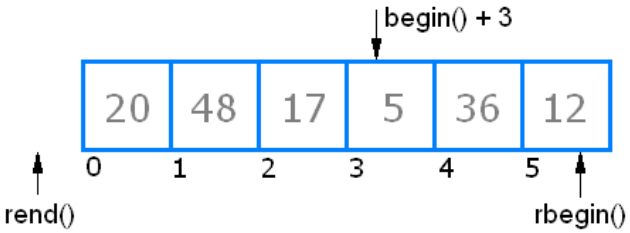

Figura 8: Iteradores inversos y aleatorios

Iteradores de entrada/salida

Existen además otros tipos de iteradores que permiten manipular objetos del tipo "streams" de entrada y salida como si fueran contenedores. Los streams más comunes son los archivos de texto y la consola. En este caso, los iteradores pueden avanzar a través de un archivo extrayendo la información que éste contiene (entrada) o escribiendo en él (salida).

Los streams_iterators son los objetos con los cuales se puede manipular estos archivos, son del tipo forward iterator y sólo pueden avanzar de a un elemento por vez desde el inicio del archivo. El siguiente ejemplo muestra cómo se utilizan para leer datos de un archivo y escribir los datos modificados en otro archivo.

```
#include <fstream> // para archivos
#include <iterator> // para streams_iterators
#include <vector>
using namespace std;

int main()
{
 // abrir el archivo para lectura
 ifstream archi("datos.txt");

 // crear un iterador de lectura para leer valores flotantes
 // en el constructor se indica a dónde apunta
 istream_iterator<float> p(archi);

 // crear un iterador que indique el fin del archivo
 istream_iterator<float> fin;

 // crear un contenedor para almacenar lo que se lee
 vector<float> arreglo;

 // recorrer el archivo y guardar en memoria
 while(p != fin)
 {
  arreglo.push_back(*p);
  p++;
 }

 archi.close();

 // calcular el valor medio de los elementos del contenedor
 float v_medio = 0;
 for(unsigned i=0; i<arreglo.size(); ++i)
  v_medio = v_medio + arreglo[i];
 v_medio = v_medio/arreglo.size();

 // restar el valor medio a cada elemento
 for(unsigned i=0; i<arreglo.size(); ++i)
  arreglo[i] -= v_medio;

 // crear un archivo para escritura
 ofstream archi2("datos_modif.txt");

 // crear un iterador de escritura para guardar los datos nuevos
 ostream_iterator<float> q(archi2, "\n");

 // grabar los datos modificados en el archivo
 for(unsigned i=0; i<arreglo.size(); ++i, q++)
  *q = arreglo[i];

 archi2.close();
```

```
 return 0;
}
```

Algo particular para observar aquí es la forma en que se indica el rango en el cual deben actuar los iteradores. Para indicar que *p* apunta al inicio del archivo se pasa en el constructor el nombre lógico del archivo ya abierto en esa posición, esto hace que el iterador apunte a la misma posición en el archivo que el puntero de lectura del mismo. Cuando se quiere crear un iterador de *fin* sólo se debe crear uno que no apunte a nada (una dirección NULL). Esto es así debido a que cuando el iterador *p* llega al final del archivo (después de leer el último elemento) se encuentra con una dirección de memoria no asignada y por lo tanto apunta al mismo lugar que *fin* y el ciclo termina.
En el caso de los iteradores de escritura (*q*), se debe indicar en el constructor el nombre lógico del archivo en el que se quiere escribir y el caracter con el que se van a separar las respectivas escrituras de datos (en este caso "\n", un fin de línea). Cuando se incrementa el iterador (*q*++) se imprime el caracter delimitador en el flujo de datos.
Como se verá más adelante el uso de iteradores de flujo permite acelerar el proceso de lectura de datos desde archivos o consola. Esto se logrará mediante la utilización de los algoritmos definidos en la biblioteca STL.

Algoritmos

Como se mencionó anteriormente, existe una gran cantidad de algoritmos disponibles en la STL que pueden ser utilizados con los contenedores e iteradores que se explicaron hasta aquí. Hay algoritmos de ordenamiento, búsqueda, mezcla, matemáticos, etc. Estos algoritmos no son otra cosa que funciones template que operan sobre los contenedores a través de los iteradores de éstos. En esta sección se explicará la lógica utilizada en la creación de estos algoritmos y se expondrán ejemplos de aquellos más comúnmente utilizados.

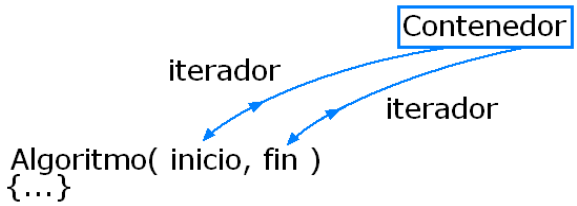

Figura 9: Relación entre los componentes

Estructura general

Para decribir la estructura general que poseen los algoritmos de la biblioteca se utilizará un ejemplo.

```
template <class T>
T Max(vector<T> & v)
{
 // crear una variable para devolver el máximo
 T maximo = v[0];

 // buscar el máximo
 for(unsigned i=1; i<v.size(); ++i)
  if(maximo < v[i])
   maximo = v[i];

 return maximo;
}
```

Aquí se tiene una función template para encontrar el elemento de máximo valor de un vector. El principal problema que tiene esta implementación es que sólo funciona con el contenedor vector y no con los demás contenedores. Otra característica es que el algoritmo busca el máximo en todo el contenedor. Sería bueno que la solución a este problema sea independiente del contenedor y además se pueda establecer un rango en el cual opere el algoritmo.

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

// crear un algoritmo genérico para encontrar el máximo
template <class Iter>
Iter Max(Iter inicio, Iter fin)
{
 // crear una variable para devolver el máximo
 Iter maximo = inicio;
 Iter aux = inicio;

 // buscar el máximo
 while(aux != fin)
 {
  if((*maximo) < (*aux))
   maximo = aux;

  ++aux;
 }

 return maximo;
}

int main()
{
```

```
 vector<int> datos;

 // llenar el vector con enteros al azar
 for(unsigned i=0; i<10; ++i)
  datos.push_back(rand() % 10);

 // mostrar por pantalla
 for(unsigned i=0; i<datos.size(); ++i)
  cout << datos[i] << " ";

 // utilizar el algoritmo para encontrar el máximo
 cout << "\nMáximo: " << *(Max(datos.begin(), datos.end()));

 return 0;
}
```

Este nuevo ejemplo presenta mayores ventajas respecto del anterior. El algoritmo funciona con cualquier contenedor que proporcione iteradores; se puede cambiar el rango de búsqueda con sólo cambiar la posición de los iteradores de inicio y fin; el iterador que se devuelve en la función encapsula tanto la información (un entero en este caso) como la posición de ella en el contenedor, por lo tanto, se tiene como resultado el mayor elemento y su posición dentro de la secuencia.

Esta forma de implementación adoptada por la STL es diferente al estilo usual de objetos, donde las funciones son miembros de las clases. Los algoritmos están separados de los contenedores, lo que facilita la escritura de algoritmos genéricos que se apliquen a muchas clases de contenedores. De esta forma resulta muy fácil agregar nuevos algoritmos sin necesidad de modificar los contenedores de la STL. Además pueden operar sobre arreglos estáticos estilo C por medio de punteros.

```
#include <iostream>
#include <list>
#include <iterator>

// para los algoritmos de la STL:
#include <algorithm>

using namespace std;

int main()
{
 // generar una lista de valores al azar utilizando "generate"
 list<int> valores(10);
 generate(valores.begin(), valores.end(), rand);

 // mostrar los elementos por pantalla utilizando "copy"
 ostream_iterator<int> salida(cout, " ");
 copy(valores.begin(), valores.end(), salida);

 // buscar la primer aparición del número 5 utilizando "find"
 list<int>::iterator p;
 p = find(valores.begin(), valores.end(), 5);

 // analizar si el iterador quedó en el rango de búsqueda
 if(p != valores.end())
  cout << "\nEl valor " << *p << " está en la lista\n";
 else
  cout << "\nNo se encontró el valor buscado\n";

 return 0;
}
```

En este ejemplo se utilizan tres algoritmos de la STL. En principio se utiliza *generate* para llenar una lista con elementos al azar. Simplemente se indica el rango de la lista y la función con la cual se generarán los elementos. Luego se emplea la función *copy* para copiar los datos almacenados en la lista a la pantalla. Este algoritmo recibe como parámetros los iteradores de inicio y fin de la *fuente* de copia, y como tercer parámetro un iterador que indique el inicio del *destino* de la copia. Cabe aclarar que el destino de la copia debe tener al menos la misma cantidad de elementos que el origen, de lo contrario se trataría de escribir en posiciones de memoria inexistentes y esto provocará un error difícil de detectar. En el caso

de los streams este problema no ocurre debido a que los archivos van generando esos espacios de memoria a medida que ingresan los datos. Finalmente se procede a buscar un número particular en la secuencia con *find*, en este caso el 5. Al comparar el iterador p con end() se analiza si la búsqueda devolvió un iterador que esté dentro del rango de búsqueda. Esto es importante, ya que si se desrreferencia end() se producirá un error de acceso a memoria.

Tipos de funciones

Antes de continuar con la descripción de los algoritmos es necesario definir los diferentes tipos de funciones que existen ya que al igual que *generate* hay otros algoritmos que reciben como parámetro alguna función que establece la forma en la que éstos actúan sobre los elementos del contenedor.

Tabla 4: tipos de funciones

Tipo de función	Descripción
Función	Recibe un parámetro T y retorna void
Función Unaria	Recibe un parámetro T y devuelve un valor tipo T
Función Binaria	Recibe dos parámetros T y devuelve un valor T
Función Generadora	No recibe parámetros sólo devuelve un valor T
Función Lógica	Recibe un parámetro T y retorna un valor booleano
Función Lógica Binaria	Recibe dos parámetros T y retorna un valor booleano
Función de Comparación	Recibe dos parámetros T y retorna uno de ellos dependiendo de cómo se compare

Algoritmos de búsqueda y ordenamiento

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

// Función lógica para saber si un número es par
template <class T>
bool EsPar(T val)
{
 return ((val%2) == 0);
}

int main()
{
 srand((unsigned)time(0));

 // generar una lista de valores al azar y mostrar por pantalla
 vector<int> valores(10);
 generate(valores.begin(), valores.end(), rand);
 copy(valores.begin(), valores.end(),
      ostream_iterator<int> (cout, " "));

 // Buscar todos los números pares y ordenarlos en una lista
 // utilizando "find_if"
 vector<int> pares;
 cout << "\nValores pares: ";
 vector<int>::iterator p = valores.begin();
 while
 ((p=find_if(p, valores.end(), EsPar<int>)) != valores.end())
 {
  pares.push_back(*p++);
 }

 // ordenar la lista de menor a mayor utilzando "sort"
 sort(pares.begin(), pares.end());

 // mostrarla por pantalla
 copy(pares.begin(), pares.end(),
      ostream_iterator<int> (cout, " "));

 return 0;
}
```

El algoritmo *find_if*, a diferencia del *find*, recibe como tercer parámetro una función lógica (*EsPar* en este caso) y retorna un iterador al primer elemento del rango que hace que esta función retorne un valor verdadero. De esta forma se puede realizar una búsqueda tan diversa como se quiera. Con *sort* se procede a ordenar los elementos del contenedor que están en el rango que se pasa como parámetro. La única condición para utilizarlo es que los iteradores deben ser de acceso aleatorio.

Algoritmos matemáticos

```
#include <iostream>
#include <fstream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <numeric> // para accumulate

using namespace std;

// para contar la cantidad de elementos positivos
template <class T>
bool EsPositivo(T val)
{
 return val >= 0;
}

int main()
{
 // cargar una señal de archivo
 vector<float> senial;
 ifstream archi("ecg.txt");

 copy(istream_iterator<float>(archi), istream_iterator<float>(),
      back_inserter(senial));

 archi.close();

 // calcular los valores máximo y mínimo
 cout << "Máximo: "
      << *max_element(senial.begin(), senial.end()) << endl;
 cout << "Mínimo: "
      << *min_element(senial.begin(), senial.end()) << endl;

 // calcular el valor medio
 cout << "Valor medio: "
      << accumulate(senial.begin(), senial.end(), 0.0)/senial.size()
      << endl;

 // contar la cantidad de elementos positivos
 cout << count_if(senial.begin(), senial.end(), EsPositivo<float>)
      << endl;

 return 0;
}
```

Los algoritmos matemáticos no modifican los elementos del contenedor sobre el que operan. Los dos primeros utilizados en el ejemplo, *max_element* y *min_element*, buscan el máximo y mínimo elemento de una secuencia respectivamente y retornan un iterador a él. *accumulate* suma los elementos que están en el rango más el tercer parámetro que es el valor inicial de la sumatoria. El cuarto algoritmo matemático es una variante de *count*. *count_if* cuenta la cantidad de elementos que satisfacen la función lógica que es pasada como parámetro.

Algo nuevo que aparece en este ejemplo es el uso de la clase *back_inserter*. Ésta recibe un contenedor como parámetro en el constructor y actúa como un iterador que cada vez que se incrementa agrega un elemento al final de la estructura. La ventaja de su empleo es que el contenedor no tiene que tener un tamaño inicial conocido y así poder leer los elementos de un archivo con una cantidad indeterminada de elementos.

Algoritmos de transformación

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

// función que pasa una letra a mayúsculas
void Mayusculas(char & c)
{
 if(('a' <= c) && (c <= 'z'))
  c = c - ('a' - 'A');
}

// función unaria para identificar vocales
char Vocales(char c)
{
 return
 (c=='A' || c=='E' || c=='I' || c=='O' || c=='U')? c : '-';
}

int main()
{
 // generar un vector con letras en minúsculas
 vector<char> v;
 for(int i=0; i<10; ++i)
  v.push_back('a' + i);

 // desordenarlas
 random_shuffle(v.begin(), v.end());

 copy(v.begin(), v.end(), ostream_iterator<char>(cout, endl));
 cout << endl;

 // aplicar a cada elemento la función Mayúsculas
 for_each(v.begin(), v.end(), Mayusculas);

 copy(v.begin(), v.end(), ostream_iterator<char>(cout, endl));
 cout << endl;

 // buscar las vocales
 transform(v.begin(), v.end(),
           ostream_iterator<char>(cout, endl), Vocales);

 return 0;
}
```

En este ejemplo se tiene un contenedor de letras al cual se le aplican transformaciones. Primero se desordena la secuencia por medio de *random_shuffle*. Luego *for_each* aplica una función a cada elemento del rango. En este caso, como la función *Mayusculas* modifica el parámetro, se modifican los elementos del contenedor. El tercer algoritmo empleado es el más genérico. *transform* aplica punto a punto la función unaria, pasada por parámetro, a la secuencia, y guarda el resultado en el lugar indicado por el tercer parámetro.

Tabla 5: resumen de algoritmos

return	Algoritmo	Parámetros	Descripción
int	count	FWIter primero, FWIter ultimo, T val	Devuelve la cantidad de apariciones de val
int	count_if	FWIter primero, FWIter ultimo, PredFunc pred	Devuelve la cantidad de elementos que satisfacen la función predicado
INIter	find	INIter primero, INIter ultimo, T val	Devuelve un iterador al primer elemento igual a val, o ultimo si no existe
INIter	find_if	INIter primero, INIter ultimo, PredFunc pred	Devuelve un iterador al primer elemento que satisface la función
bool	equal	INIter1 primero1, INIter1 ultimo1, INIter2 primero2	Devuelve true si ambos tienen los mismos elementos
FWIter1	search	FWIter1 primero1, FWIter1 ultimo1, FWIter2 primero2, FWIter2 ultimo2	Localiza la aparición del segundo contenedor en el primero

FWIter	min_element	FWIter primero, FWIter ultimo	Devuelve un iterador al menor elemento
FWIter	max_element	FWIter primero, FWIter ultimo	Devuelve un iterador al mayor elemento
OUTIter	copy	INIter primero1, INIter ultimo1, OUTIter primero2	Copia el contenido del primero en el segundo
void	swap	T & p, T & q	Intercambia los valores de p y q
Func	for_each	INIter primero, INIter ultimo, Func f	Aplica la función f a cada elemento del contenedor
OUTIter	transform	INIter primero, INIter ultimo, OUTIter result, UNFunc f	Aplica f a cada elemento del contenedor y guarda el resultado en result
OUTIter	transform	INIter1 primero1, INIter1 ultimo1, INIter2 primero2, OUTIter result, BINFunc f	Aplica f a ambos contenedores y guarda el resultado en result
void	replace	FWIter primero, FWIter ultimo, T & valor_viejo, T & valor_nuevo	Reemplaza todos los valores viejos por los nuevos
void	fill	FWIter primero, FWIter ultimo, T & val	Llena el contenedor con val
FWIter	remove	FWIter primero, FWIter ultimo, T & val	Elimina todas las apariciones de val
FWIter	unique	FWIter primero, FWIter ultimo	Elimina los elementos contiguos repetidos
void	reverse	BDIter primero, BDIter ultimo	Invierte el orden de la lista
void	rotate	FWIter primero, FWIter centro, FWIter ultimo	Rota la lista hacia la derecha hasta que primero == centro
void	random_shuffle	RNDIter primero, RNDIter ultimo	Reordena los elementos al azar
void	sort	RNDIter primero, RNDIter ultimo	Ordena los elementos de menor a mayor
T	accumulate	INIter primero, INIter ultimo, T val_ini	Da como resultado val_ini más la sumatoria de los elementos del contenedor
T	inner_product	INIter1 primero1, INIter1 ultimo1, INIter2 primero2, T val_ini	Producto interno entre los contenedores más val_ini

Contenedores asociativos

Hasta este momento se explicó la estructura general de la STL con los contenedores lineales, los iteradores y los algoritmos. Con esta idea general, se retoma la descripción de los contenedores faltantes, los *asociativos*. Estos contenedores son: *map*, *multimap*, *set* y *multiset*. Ellos tienen la característica de almacenar claves ordenadas que están asociadas a un valor, y en el caso de set y multiset la clave es el valor en sí mismo.

Su estructura en memoria no es secuencial como en los contenedores anteriores sino que se implementan como árboles binarios de búsqueda balanceados. Esto hace que el tiempo de búsqueda sea proporcional al logaritmo en base dos de la cantidad de elementos, en vez de ser proporcional al tamaño del contenedor, como es el caso de las listas lineales.

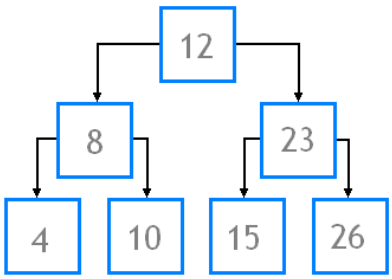

Figura 10: Representación esquemática de un árbol binario de búsqueda

La operación de comparación para el ordenamiento de las claves se establece en la instanciación, y por defecto se realiza con el operador menor (<) para un ordenamiento ascendente. Se debe tener en cuenta que los tipos de datos utilizados como clave deben "soportar" (deben tener sobrecargado) el operador que se utilice para la comparación, en caso de ser clases creadas por el programador.

Los contenedores asociativos proveen para su manipulación iteradores bidireccionales, al igual que las listas doblemente enlazadas. Por lo tanto, sólo se podrán utilizar aquellos algoritmos que requieran de estos iteradores. Sin embargo, estos contenedores proveen algunos métodos para las funciones de búsqueda y conteo, los cuales se explican a continuación.

Tabla 6: operaciones comunes en contenedores asociativos

A::insert(clave & x)	Inserta el elemento x en el contenedor
A::insert(A::iterator i, A::iterator f)	Inserta los elementos que están en el rango de los iteradores en el contenedor
A::erase(clave & x)	Borra todos los elementos que tengan la clave x
A::erase(A::iterator p)	Borra el elemento apuntado por p
A::count(clave & x)	Devuelve la cantidad de elementos que tienen la clave x
A::find(clave & x)	Devuelve un iterador al primer elemento que tenga la clave x
A::lower_bound(clave & x)	Devuelve un iterador al primer elemento que tenga la clave x
A::upper_bound(clave & x)	Devuelve un iterador al elemento siguiente al último elemento con clave x

Set y Multiset

Estos contenedores se utilizan para manipular conjuntos de elementos, especialmente en operaciones de búsqueda. Tanto en set como en multiset, la clave de ordenamiento es el valor que se quiere contener. La diferencia entre estos contenedores es que set no permite la existencia de claves repetidas, mientras que multiset sí.

```
#include <iostream>
#include <fstream>
#include <iterator>

// para set y multiset
#include <set>
```

```
using namespace std;

int main()
{
 // cargar los datos de un archivo en memoria
 set<int> valores;
 ifstream archi("datos.txt");
 istream_iterator<int> i(archi);
 istream_iterator<int> f;
 while(i != f)
  valores.insert(*i++);
 archi.close();

 // mostrar los elementos por pantalla
 copy(valores.begin(), valores.end(),
      ostream_iterator<int>(cout, " "));

 // buscar un elemento en la estructura
 set<int>::iterator p = valores.find(0);

 if(p != valores.end())
  cout << "\nEl valor 0 está en el archivo\n";
 else
  cout << "\nEl valor 0 no está en el archivo\n";

 // comparar con multiset
 multiset<int> multivalores;
 archi.open("datos.txt");
 istream_iterator<int> j(archi);
 while(j != f)
  multivalores.insert(*j++);
 archi.close();

 // mostrar los elementos por pantalla
 copy(multivalores.begin(), multivalores.end(),
      ostream_iterator<int>(cout, " "));

 // contar las apariciones de 0
 cout << "\nEl valor 0 está "
      << multivalores.count(0)
      << " veces en el archivo\n";

 // eliminar un elemento
 multivalores.erase(2);
 copy(multivalores.begin(), multivalores.end(),
      ostream_iterator<int>(cout, " "));

 return 0;
}
```

Aquí se tiene un ejemplo básico del empleo de estos contenedores asociativos para conocer si un valor particular se encuentra en un conjunto de elementos. Observar que con estas estructuras no se emplea la función push_back() como se hacía con las estructuras lineales; en cambio, se emplea la función *insert()*. Esto es lógico debido a que el elemento que se agregue al contenedor quedará ordenado en la estructura, y su posición no será en el final precisamente. Otra función se utiliza es *erase()*, que borra todas las claves que coincidan con el parámetro. El funcionamiento de *find()* y *count()* es el mismo que se vio anteriormente, solo que reciben como parámetro únicamente el valor a buscar o contar.

Pair

Antes de continuar con la descripción de map y multimap, se presenta una estructura que es utilizada internamente por estos contenedores. Los pairs (pares) son estructuras genéricas que contienen dos datos: *first* y *second*. Donde el primero de ellos es utilizado como clave (key) para la búsqueda de elementos y el segundo como el valor asociado a esta clave (value). Para utilizarlos se debe incluir la biblioteca *utility* de la STL.

```
template <class T1, class T2>
struct pair
{
 T1 first;
 T2 second;
```

```
 pair() {}
 pair(const T1& a, const T2& b):
  first(a), second(b) {}
};
```

Estas estructuras se utilizan también en algunos métodos como valor de retorno, cuando se quiere devolver dos datos. Esto sucede, por ejemplo, en la función insert de los contenedores asociativos set y map, donde el resultado es un pair que tiene como primer elemento el iterador que apunta al elemento insertado y como segundo elemento un valor booleano que es verdadero si el elemento no estaba en el contenedor.

Map y Multimap

Por último, y no por esto menos importante, queda describir la estructura y el funcionamiento de los contenedores asociativos map y multimap. Ellos mantienen la estructura de árbol al igual que set y multiset, pero almacenan los elementos de a pares: la *clave* y el *valor* asociado. Estas estructuras mantienen ordenados los pares por medio de sus claves, que por defecto lo hacen en orden ascendente. De esta manera, los maps son útiles cundo se necesita buscar determinada información en un volumen grande a partir de algún dato de búsqueda, que será la clave, o cuando se necesitan tablas de datos apareadas.

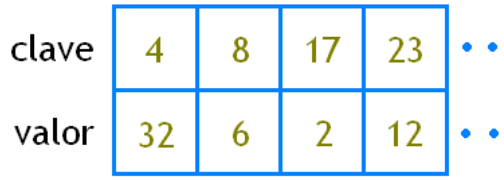

Figura 11: Representación de un map en memoria

En la figura se observa un esquema simplificado de la estructura en memoria de un map. Al igual que set y multiset, la diferencia entre map y multimap yace en que los últimos permiten que existan elementos con claves repetidas, como se puede apreciar en la siguiente figura. Obsérvese que sólo se ordenan las claves y no los valores asociados.

Figura 12: Multimap

En los siguientes ejemplos se expondrán las funcionalidades que estos contenedores poseen para la inserción, búsqueda y conteo de información. Para comenzar, se desea conocer la frecuencia de repetición de los valores de una secuencia de valores de punto flotantes almacenados en un archivo de texto. Se hará un mapeo de los valores y la cantidad de veces que aparece en el archivo.

```
#include <iostream>
#include <fstream>
#include <map> // para map y multimap
using namespace std;

int main()
{
 // crear un map donde la clave es el valor del archivo
 // y su valor asociado es su repetición
 map<float, unsigned> m;
 ifstream archi("datos.txt");
 float aux;

 while(archi >> aux)
 {
  // preguntar si ese valor no existe en la estructura
  if(m.find(aux) == m.end())
   m[aux] = 1;
```

```
  // si ya existe, incrementar en 1 su repetición
  else
   m[aux] = m[aux] + 1;
 }
 archi.close();

 // mostrar en pantalla los datos encontrados
 map<float, unsigned>::iterator i = m.begin();
 while(i != m.end())
 {
  cout << i->first << " " << i->second << endl;
  i++;
 }

 return 0;
}
```

Esta resolución del problema planteado presenta una funcionalidad extra de los maps que no poseen set, multiset ni multimap: el empleo del operador []. Con él se puede insertar un par de elementos dentro de la estructura: la clave va entre corchetes y el valor asociado es el valor que se asigna. Así, por ejemplo, para insertar el par (-2.34, 5) en un map se debería escribir:

```
m[-2.34] = 5;
```

Si no se asigna ningún valor cuando se crea un par nuevo, el valor asociado será creado con su constructor por defecto. Otro detalle a tener en cuenta es el empleo de iteradores con map. En estas estructuras los iteradores apuntan a un par de elementos, por lo tanto, al desrreferenciar un iterador se obtiene una estructura pair, donde el primer elemento (first) es la clave y el segundo (second) es el valor asociado.

El siguiente problema que se resolverá consiste en crear un objeto que busque información de una persona de un archivo a través de su identificación. El archivo tiene la información de personas que asistieron a una campaña de vacunación y para cada persona la información está codificada de la siguiente manera: ID (cadena de 20 caracteres), nombre (cadena de 20 caracteres), edad (char), sexo (char), domicilio (cadena de 50 caracteres) y teléfono (cadena de 10 caracteres). Para esto en el constructor de este objeto se cargará toda la información relevante de las personas en una estructura map.

```
#ifndef BUSCADOR_H
#define BUSCADOR_H

#include <fstream>
#include <string>
#include <map>
using namespace std;

class Buscador
{
 private:
  map<string, long int> map_pos;
  ifstream archi;
 public:
  Buscador(string nombre_archi);
  ~Buscador();
  void BuscarDatos(string ID);
};

Buscador::Buscador(string nombre_archi)
{
 archi.open(nombre_archi.c_str(), ios::in | ios::binary | ios::ate);
 long int pos_final = archi.tellg();
 archi.seekg(0, ios::beg);
 char ID[20];
 long int pos;

 while(archi.tellg() != pos_final)
 {
  archi.read(ID, sizeof(ID));
  pos = archi.tellg();
  map_pos[string(ID)] = pos;
```

```
 }
}

Buscador::~Buscador()
{
 archi.close();
}

void Buscador::BuscarDatos(string ID)
{
 if(map_pos.find(ID) != map_pos.end())
 {
  archi.seekg(map_pos[ID], ios::beg);
  char nombre[20], edad, sexo, domicilio[50], telefono[10];
  archi.read(nombre, sizeof(nombre));
  archi.read((char*)&edad, sizeof(edad));
  archi.read((char*)&sexo, sizeof(sexo));
  archi.read(domicilio, sizeof(domicilio));
  archi.read(telefono, sizeof(telefono));

  cout << "Los datos de la persona con el ID N°: " << ID
       << " son:\n"
       << "Nombre: " << nombre << endl
       << "Edad: " << edad << endl
       << "Sexo: " << sexo << endl
       << "Domicilio: " << domicilio << endl
       << "Teléfono: " << telefono << endl;
 }
 else
  cout << "No se encontró a la persona con el ID " << ID << endl;
}

#endif
```

Bibliografía

- "Standard Template Library Programmer's Guide" – SGI – 1996-1999
- "The STL made simple" – Bruce Eckel – 1999
- "STL Quick Reference" – Version 1.20 - 2000
- "STL Tutorial and Reference guide" – David Musser – 1996
- "Thinking in C++" – Bruce Eckel – 2º Edición
- "Phil Ottewell's STL Tutorial" – Phil Ottewell - 1998