```
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: