Librería Estándar de Plantillas (Standard Template Library STL)

(Para una información detallada de la librería, podeis consultar los siguientes links http://www.sgi.com/tech/stl/ y http://www.cppreference.com/)

La STL es una potente herramienta que implementa clases y funciones de uso general. Por un lado, se definen contenedores genéricos (vector, lista, pila, etc) y por otro lado se definen funciones genéricas para operar con ellos.

1. Contenedores de Secuencia e Iteradores

Los contenedores son clases que contienen o almacenan objetos. En esta lección nos centraremos en explicar los dos contenedores más comunes: *vector, lista y string*. Estos contenedores se denominan también secuencia. Esto es porque los elementos tienen un orden dentro del contenedor. Por tanto, podrá hacerse un recorrido ordenado de los mismos.

1.1 Clase Vector

La clase vector esta definida en el fichero <vector> y bajo el espacio de nombres std. Esta clase se parece mucho a la clase que creamos en la Práctica 4. Internamente, la clase se encarga de asignar memoria o eliminarla de forma dinámica en función de las necesidades en tiempo de ejecución. Pasemos directamente a ver un ejemplo:

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


int main()
{
    vector<int> vi;//declares an empty vector of integers
    vector<float> vf(10);//declares a float vector with 10 elements in
it
    vector<double> vd(5,1);//declares a double vector with 5 elements
all with value 1

    //iterate through the vector vd
    for(unsigned int i=0;i<vd.size();i++)
        cout<<vd[i]<< " ";
    cout<<endl;
```

```
    //Add elements to a vector
    for(unsigned int i=0;i<10;i++)
        vi.push_back( i );
    //See the vector elements
    for(unsigned int i=0;i<vi.size();i++)
        cout<<vi[i]<< " ";
    cout<<endl;

    //remove elements from the back of the vector
    vi.pop_back();
    vi.pop_back();
    //Let's see the vector size now
    cout<<" VSize="<<vi.size()<<endl;
    //Let's iterate again
    for(unsigned int i=0;i<vi.size();i++)
        cout<<vi[i]<< " ";
    cout<<endl;

   //Resize the vector to 5 elements
   vi.resize(5); //the last elements are removed. Only the first 5 are
kept

   //Now, we are going to overwrite the vector element values
   for(unsigned int i=0;i<vi.size();i++)
    vi[i]=i*i;
   //See the new values
   for(unsigned int i=0;i<vi.size();i++)
    cout<<vi[i]<< " ";
    cout<<endl;

 //Finally, let's see how easy it is to copy a vector
  vector<int> vcopy;
  vcopy=vi;
 }
```

Bueno, el ejemplo es bastante explicativo. Muestra como es posible manejar vectores que se redimensionan de forma dinámica para ajustarse a las necesidades en tiempo de programación. Para una información detallada de las funciones miembro de la clase vector véase http://www.cppreference.com/cppvector/index.html En el ejemplo anterior hemos visto :

- Tres tipos diferentes de constructores
- La función *size()* que indica el número de elementos
- La función *push_back()* que añade un elemento al final
- La función *pop_back()* que elimina el último elemento del vector

- La función *resize()* redimensiona el vector
- El operador [] nos permite acceder al elemento para ver o modificarlo
- El operador de asignación = que nos permite copiar vectores

En el ejemplo anterior, solo usamos tipos genéricos, pero como ya sabemos de prácticas anteriores, es posible instanciar vectores de clases. Eso si, hay una serie de requisitos que una clase debe cumplir para poder ser utilizada. La clase debe implementar:

- El constructor vacío
- El constructor de copia
- El operador de asignación (=) si se desea hacer asignaciones del tipo v[i]=Object.

Veamos un ejemplo:

```
#include <vector>
#include <iostream>
using namespace std;
class MyClass
{
    public:
        MyClass(){_ival=10;_fval=10;}
        MyClass(int i,float f){_ival=i;_fval=f;}
        MyClass(const MyClass&C){ _ival=C._ival; _fval=C._fval; }
        ~MyClass(){}
        MyClass & operator=(const MyClass&C){ _ival=C._ival; _fval=
C._fval; }

        friend ostream &operator<<(ostream & str,const MyClass &C)
        {
            str<<C._ival<<" "<<C._fval<<" ";
            return str;
        }
        int getIVal()const {return  _ival;}
 private:
        int _ival;
        float _fval;
};

int main()
{
    //Creates a vector with 10 elements.
    //Empty constructor is employed
    vector<MyClass> vClass(10);
    for(unsigned int i=0;i<vClass.size();i++)
```

```
        cout<<vClass[i]<<endl;

    //Now, creates a vector with 5 elements, using the
    //copy constructor. All elements are copy of the one passed as
last
    //parameter
    vector<MyClass> vClass2(5,MyClass(1,1));
    for(unsigned int i=0;i<vClass2.size();i++)
        cout<<vClass2[i]<<endl;

    //Now, create an empty vector and fill it with elements
    vector<MyClass> vClass3;
    for(int i=0;i<10;i++)
        vClass3.push_back(MyClass(i,-1));
    //see them
    for(unsigned int i=0;i<vClass3.size();i++)
        cout<<vClass3[i]<<endl;


    //Use the assign operator to set values (=)
    for(unsigned int i=0;i<vClass3.size();i++)
        vClass3[i]=MyClass(i*i,0);
    //see them
    for(unsigned int i=0;i<vClass3.size();i++)
        cout<<vClass3[i]<<endl;
 }
```

1.2 Iteradores

Como hemos comentado anteriormente, la clase vector (así como la clase lista y string) es una secuencia. Es decir, los elementos dentro del mismo tienen un orden. Los iteradores son un mecanismo genérico para recorrer secuencias. Podemos decir que un iterador es un elemento que itera en una secuencia (ya sea vector, lista u string)

Veamos como se usa el iterador para un vector

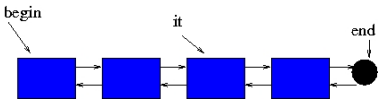
```
#include <vector>
#include <iostream>
using namespace std;
class MyClass
{
    public:
```

```
        MyClass(){_ival=10;_fval=10;}
        MyClass(int i,float f){_ival=i;_fval=f;}
        MyClass(const MyClass&C){ _ival=C._ival; _fval=C._fval; }
        ~MyClass(){}
        MyClass & operator=(const MyClass&C){ _ival=C._ival; _fval=
C._fval; }

        friend ostream &operator<<(ostream & str,const MyClass &C)
        {
            str<<C._ival<<" "<<C._fval<<" ";
            return str;
        }
        int getIVal()const {return  _ival;}
 private:
        int _ival;
        float _fval;
};

int main()
{

    vector<int> v;//Declare vector
    v.resize(10);//resize
    vector<int>::iterator it;//declare an iterator
    //Fill element values
    for( it=v.begin(); it!=v.end(); it++)
        (*it)=0;

    vector<MyClass> cv;//Declare a vector
    cv.resize(5);//resize
    vector<MyClass>::iterator it2; //declare an iterator
    //Fill the elements
    int counter=0;
    for( it2=cv.begin(); it2!=cv.end(); it2++,counter++)
        (*it2)=MyClass(counter,-1);

    //Now, call to a function
    for( it2=cv.begin(); it2!=cv.end(); it2++)
        cout<< it2-> getIVal()<<endl;


    //Call to  operator <<
    for( it2=cv.begin(); it2!=cv.end(); it2++)
        cout<< (*it2)<<endl;
}
```

Bueno, como podéis ver al principio puede parecer algo lioso, pero en realidad es muy simple. Un iterador se declara como

```
vector<Type>::iterator it;
```

A partir de ahí, podemos pensar en el iterador como un puntero a uno de los elementos del vector. Las clases secuencia definen siempre las funciones *begin()* y *end()*. La función begin() retorna un iterador (o puntero) apuntando al primer elemento de la secuencia. Mientras que *end()* retorna un iterador a un elemento justo detrás del último de la secuencia



De esta forma, podemos ir avanzado el iterador por los elementos de la secuencia simplemente haciendo *it*++, o retroceder si queremos con *it*--.

1.2.1 Eliminar elementos de una secuencia usando iteradores

Los iteradores, además de permitirnos recorrer cualquier secuencia de forma estándar, nos permiten eliminar elementos de una secuencia de forma muy simple. Vease el siguiente ejemplo

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{

    vector<int> v(10);//Declare vector
    //Fill it
    for(unsigned int i=0; i<v.size(); i++)
        v[i]=i;

    //randomly shuffle the elements of the sequence
    random_shuffle(v.begin(),v.end());

    //see them
    for(unsigned int i=0;i<v.size();i++)
        cout<<v[i]<<" ";
    cout<<endl;

    //Delete element with value 5 while iterating the sequence
    vector<int>::iterator it;
```

```
    for(it=v.begin();it!=v.end(); ){
        if ( (*it) == 5)
            it=v.erase(it);//deletes it and return a pointer to the
next element
        else it++;
    }

    //see the new vector
    for(unsigned int i=0;i<v.size();i++)
        cout<<v[i]<<" ";
    cout<<endl;
}
```

Bueno, aquí aparecen varias cosas nuevas. En primer lugar, vemos que hemos usado una nueva función (*random_shuffle*()). Esta función, recibe como argumentos un iterador inicial y otro final y reordena los elementos entre ellos dos de forma aleatoria. Esta función es parte de las funciones genéricas que trae la STL y que veremos más adelante (Para más información véase http://www.cppreference.com/cppalgorithm/index.html

Por otro lado, vemos que podemos eliminar un elemento de una secuencia usando la función *erase()*. Esta función recibe como argumento un iterator y retorna un iterador al siguiente elemento de la secuencia. Se suele asignar it al valor retornado

```
it=v.erase(it);
```
porque si no, it será invalido ya que apuntaría a un elemento borrado.

1.2.2 Insertar elementos en una secuencia

Al igual que la eliminación, los iteradores nos sirven para insertar elementos. Véase el siguiente ejemplo:

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

int main()
{
    vector<int> iv(10,1);//creates a list with 10 elements
    vector<int>::iterator it=iv.begin();
    //advance the iterator
    advance(it,3); //it is equivalent to it+=3;
    //insert a new element
```

```
    iv.insert(it,9999);
    //print the vector
    for(unsigned int i=0;i<iv.size();i++)
        cout<<iv[i]<<endl;
}
```

La función *advance()* mueve el iterador el número de veces indicado. Después, la función *insert()* inserta el elemento deseado en la posición donde se encuentra el iterador.

1.3 Clase Lista

La clase lista es la adecuada cuando se desean realizar inserciones o eliminaciones en lugares arbitrarios de la secuencia ya que el tiempo de inserción y eliminación en una lista es menor que en un vector (sobre todo cuando no se inserta o elimina del final). La clase comparte un interfaz muy similar al de vector, pero en esta ocasión no se puede realizar el acceso aleatorio (*operator[]*). Veamos un ejemplo:

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

int main()
{
    list<int> il(10);//creates a list with 10 elements
    //set a value for each element
    list<int>::iterator it=il.begin();
    int counter=0;
    for(; it!=il.end();it++)
        (*it)=counter++;
    //Now add elements to the end
    il.push_back(999);
    //adds a element at the beginning
    il.push_front(-1);
    //delete the last elment
    il.pop_back();
    //delete the first element
    il.pop_front();

    //adds an element in the middle of the sequence
    it=il.begin();
    //advance the iterator 3 positions
    advance(it,3);
```

```
    //insert the element after the 3th elment
    il.insert(it,1234);

    //see the result
    for(it=il.begin(); it!=il.end();it++)
        cout<<*it<<endl;
}
```

Como podeis observar, la clase lista añade *pop_front()* y *push_front()* pero elimina el operador *[]*. Para más información sobre la lista véase http://www.cppreference.com/cpplist/index.html

1.4 String

La clase string ya se ha visto anteriormente en las prácticas. Esta clase está implementada como un tipo especial de vector al que se agregan métodos específicos para el manejo de cadenas. Para más información véase http://www.cppreference.com/cppstring/index.html

2 Funciones genéricas

Finalmente, la STL proporciona una serie de funciones genéricas que operan normalmente sobre secuencias. Ya vimos como ejemplo la función random_shuffle(). Pero hay muchas más http://www.cppreference.com/cppalgorithm/index.html

Entre ellas, cabe destacar:

- *sort()*: ordena los elementos de una secuencia. La secuencia debe permitir acceso aleatorio, por tanto, deberá ser un vector. Los objetos de la secuencia deben de implementar el operador <.
- *max(),min()*: busca maximo o minimo en una secuencia. Los objetos de la secuencia deben de implementar el operador <.
- *count()*: cuenta el número de elementos de una secuencia que son iguales que uno dado. Los objetos de la secuencia deben implementar el operador ==.
- Y otros muchos algoritmos de uso general.

3. Ejercicios

Ejercicio 3.1

Crear una aplicación para leer una lista de elementos flotantes almancenados en fichero y ordenarlos usando STL. La aplicación tomará como entrada el siguiente fichero file.dat. Deberá ir leyendo elemento a elemento hasta llegar al final del fichero. Los elementos que vaya leyendo deberá insertarlos en una lista. Finalmente, usará la función accumulate para conocer la suma de todos los elementos.

Nota de ayuda: recuerde que para lectura en modo texto de un fichero debe usar el operador << de la clase ifstream.

Ejercicio 3.2

Herede de la clase vector de forma privada para crear la clase plantilla pila. La clase pertenecerá al espacio de nombres *miscontenedores*. La clase tendrá los siguientes métodos:

- Constructor vacio
- Constructor de copia
- apila(): añade un elemento en lo alto de la pila
- desapila(): retorna el elemento en lo alto de la pila y lo quita de la pila.
- tamano(): función constante que retorna el número de elementos de la pila
- vacia(): función constante que retorna un booleano indicando si la pila está vacía.

Utilice el siguiente código para probar su clase

```
#include <iostream>
#include <cassert>
#include <pila.h>
using namespace miscontenedores;
int main()
{
    Pila<int> P;
    assert(P.tamano()==0);
    assert(P.vacia());
    cout<<"apilando"<<endl;
    for(int i=0;i<10;i++)
        P.apila(i);
    cout<<"despilando"<<endl;
    for(int i=9;i>=0;i--)
        assert( P.desapila()==i);
    assert(P.tamano()==0);
```

```
    assert(P.vacia());
    cout<<"Perfect"<<endl;
}
```

Notas y ejemplo: Heredar de una clase que es una plantilla, es algo de lo más complejo desde el punto de vista de sintaxis. Por tanto, es bastante fácil que se produzcan errores de sintaxis al compilar. Para ver un equeleto de como se debe hacer vease el esquema de plantilla y de cómo se hereda de una plantilla.

Ejercicio 3.3

Realice una función que multiplique dos vectores componente a componente.

```
/**Multiplica dos vectores y coloca el resultado en el tercer
parametro
*@param src1,src2 vectores a multiplicar
*@param dst vector resultado. Este vector será redimensionado
apropiadamente en el
* interior en caso de que la multiplicación sea posible
*@return true si la multiplicación ha tenido exito. false si los
vectores no se pueden multiplicar
*/
template<class T>
bool multiplica(vector<T> &src1,vector<T> &src2,vector<T> &dst);
```

Realice también el programa main para probar su código.

NOTA: Es posible que tenga que añadir la palabra clave typename antes de declarar un iterador en funciones o clases plantilla.

```
typename list<int>::iterator it;
```

Ejercicio 3.4

Realice el anterior ejercicio pero ahora con listas en lugar de con vectores. En este caso deberá usar iteradores forzosamente para recorrer los elementos.

Ejercicio 3.5

Realice un algoritmo de inserción ordenada en una secuencia. Su función recibirá una lista ordenada y un número a añadir. La fución deberá insertar el nuevo elmento de forma ordenada en orden creciente.

```
#include <list>
#include <vector>
#include <cassert>
#include <algorithm>
using namespace std;

/**Inserta value en orden en l. Se asume que l esta ordenada al inicio
y al fin de la llamada a esta función
*/
 void insertaOrdenado(list<int> &l,const int&value)
{
//COMPLETAD VOSOTROS
}

int main()
{
    list<int> l1;
    vector<int> va;
    srandom(time(NULL));
    for(unsigned int i=0;i<30;i++){
        int val=int( 100*float(random())/float(RAND_MAX));
        insertaOrdenado(l1,val);
        va.push_back(val);
        }

    std::sort(va.begin(),va.end());

    assert(l1.size()==va.size());

    list<int>::iterator itl=l1.begin();
    vector<int>::iterator itv=va.begin();

    for(;itl!=l1.end();itl++,itv++){
        assert(*itl==*itv);
    }
}
```