Teoría sobre la STL

por Jose Luis Venega Sánchez

STL - Introducción

La STL 'Standard Template Library' es una biblioteca de clases y funciones template creadas para estandarizar y optimizar la utilización de algoritmos y estructuras de datos en el desarrollo de software en C++.

La biblioteca presenta tres componentes básicos: *contenedores*, *iteradores* y *algoritmos*.

STL - Contenedores

Un contenedor almacena/agrupa una colección de elementos y permite realizar ciertas operaciones con ellos.

La diferencia entre un contenedor y otro es:
- La forma en la que los son alojados.
- Como se crea la secuencia de elementos.
- La manera de acceder a ellos.

Encontramos:

- Contenedores de secuencia o lineales:
  - Vectores (vector).
  - Listas (lists).
  - Bicola (deque).

- Contenedores asociativos:
  - Set
  - Multiset.
  - Map.
  - Multimap.

Creación de contenedores

X <T> instancia;

Siendo 'X' el contenedor que vamos a utilizar, 'T' el tipo de dato que vamos a utilizar, 'instancia' el nombre que vamos a utilizar para el contenedor.

Ejemplos

vector<int> valores; → Vector de enteros vacíos con el nombre valores.
vector<vector<bool>> a; → Matriz a de booleanos vacia.
vector<int> aux(valores); → Vector de enteros copia del vector valores.
vector<int> aux = valores; → Igual que anterior.

También podemos construirlos ***por rangos***, suministrando un par de iteradores durante la definición.

Esto permite poder copiar los datos de un contenedor a otro de distinta categoría.

deque <int> w(v.begin(), v.end());

En este caso tenemos 'w' que es una cola doble con los elementos del vector v.

Por otro lado, encontramos ***begin()*** y ***end()*** que permiten construir un rango que engloba a todos los elementos del contenedor y son comunes a todos los contenedores estándar.

Operaciones comunes de los contenedores

X::size()	Devuelve la cantidad de elementos que contiene el contenedor como un entero sin signo 'unsigned'.
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 contiene ningún elemento.
X::swap(X & x)	Intercambia el contenido del contenedor con el que 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 lexográficamente y retornan en valor

Vectores

Los elementos en los vectores están contenidos en posiciones contiguas en memoria. Esto permite mayor velocidad de acceso a los elementos ya que si se quiere acceder a un elemento solo deberemos desplazarnos tantos lugares desde el principio del contenedor (vector).

Se declara de la manera → *vector <tipo_dato> nombre_vector; //se inicializa vacío
vector <tipo_dato> nombre_vector(n); //se inicializa con n elementos.

Para recorrerlos realizamos un bucle (for, while), donde tengamos un indice que nos indicará la posicion del vector. Siempre empieza en 0, hasta n (tamaño max del vector).

Solo se insertan elementos por el extremo final del contenedor.

Deque

Representa una cola con doble final, similar a un vector, pues sus datos también están contiguos en memoria. Se diferencia en que al tener doble final se pueden insertar elementos por ambos extremos del contenedor.

Tienen las mismas funcionalidades que los vectores e incluso se puede acceder a los elementos a través de subíndices (acceso aleatorio). Además encontramos dos métodos extras para poder realizar operaciones en el frente del contenedor: ***push_front(T& x)*** y ***pop_front();***
Se declara → deque <tipo_dato> nombre_deque;

List

Las listas son los contenedores adecuados cuando se requieren operación de inserción o eliminación en cualquier parte del contenedor.

Están implementadas como listas doblemente enlazadas, es decir, cada nodo contiene las direcciones del nodo siguiente y del nodo anterior, además del valor que contienen.

- Ventajas → Operaciones de inserción y eliminación de un elemento se reduce a ordenar los punteros del siguiente y anterior nodo.

- Desventajas → No tenemos acceso aleatorio a los elementos que están contenidos en la lista, si no que tenemos un acceso secuencial bidireccional (debido a esos 2 punteros). Solo podemos recorrer la lista de inicio a fin o viceversa.

Hacemos uso de los ***iteradores*** para recorrer las listas.
Se declara → list <tipo_dato> nombre_lista;

STL - Iteradores

Iteradores - Introducción

Un iterador es una abstracción del concepto de puntero que presenta los elementos de un contenedor como si fueran una secuencia que podemos recorrer.

Todos los contenedores proporcionan dos iteradores que establecen el rango recorrido: ***begin*** y ***end*** → [begin,end).

> **Begin** indica la posición del primer elemento.
> ***End*** apunta a una posición posterior al ultimo elemento.

Por eso el rango es cerrado al inicio pero abierto al final.
Un iterador es un objeto que abstrae el proceso de moverse a través de una secuencia de elementos (lista). Permite seleccionar un elemento de un contenedor sin tener la necesidad de conocer la estructura interna del contenedor.

Encontramos:

Nombre	Operaciones
Entrada	*, —> (ambos sólo lectura), ++, ==, !=
Salida	+ (sólo para escritura), ++
Monodireccionales	La de los iteradores de I/O
Bidirecionales	Las de los iteradores de I/O más --
Acceso directo	+ (con iter. y entero), -(con iter. y entero, o con dos iters.)

La sintaxis general para crear un iterador es:

X::iterator instancia; → Donde X es el tipo de contenedor

Ejemplos

```
deque<double>::iterator inicio; → No apunta a ningún elemento
deque<double> valores(10,0);
deque<double>::iterator inicio(valores.begin());
deque<double>::iterator inicio2;
inicio2 = valores.begin();
```

Es importante conocer cuales de estos iteradores proveen los contenedores antes vistos.

En el caso de las listas doblemente enlazadas sólo se pueden realizar movimientos de avance o retroceso sobre la secuencia, por los tanto, estas listas proveen iteradores bidireccionales.

Tanto vectores como deque tienen sus elementos contiguos en memoria y permiten 'saltar' a las diferentes posiciones sin mayor complicación. Estos contenedores proporcionan ***iteradores acceso aleatorio***.

Iteradores - De Entrada y Salida

Existen iteradores que permiten manipular objetos de tipo 'streams' de entrada y salida como si fueran contenedores.

Los ***streams_iterators*** son los objetos con los cuales se puede manipular estos archivos, son de un tipo similar a ***forward iterator*** y sólo pueden avanzar a un elemento por vez desde el inicio del archivo.

Iteradores - Ejemplo

```
/*Ejemplo que nos muestra los elementos de un vector por pantalla
  haciendo uso exclusivamente de iteradores*/
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;

//funcion template para mostrar los elementos
//de un contenedor por pantalla utilizando iteradores
template <class iter>
void MostrarPantalla(iter inicio, iter final){
    while(inicio!= final)
    cout<< *inicio++ <<" "; //avanzamos e imprimimos el vector
}
int main(){
    //declaramos un vector de 20 letras vacío
    vector<char>letras(20);
    //rellenamos el vector con elementos de forma aleatoria
    for(unsigned i=0; i<letras.size(); i++){
        letras[i] = 'A' + (rand()%26);
    }

    //visualizamos por pantalla el contenido del vector
    MostrarPantalla(letras.begin(), letras.end());
    cout<<endl;
    //visualizamos por pantalla el contenido del vector a la inversa
    MostrarPantalla(letras.rbegin(), letras.rend());
    cout<<endl;

    //visualizamos el contenido de los 10 elementos del medio
    MostrarPantalla(letras.begin()+5, letras.begin()+15);
    cout<<endl;

    return 0;
}
```

Iteradores Entrada y Salida - Ejemplo

Para indicar que 'p' apunta al inicio del archivo se pasan en el constructor el nombre lógico del archivo ya abierto en esa posicion (*líneas 9 y 11*), esto hace que el iterador apunte a la misma posición en el archivo que el puntero de lectura del mismo.

```
1  /*Ejemplo que nos muestra como podemos recorrer streams con iteradores*/
2  #include <fstream> //para archivos
3  #include <iterator> //para streams_iterator
4  #include <vector>
5  using namespace std;
6
7  int main(){
8      //abrimos el archivo para lectura
9      ifstream archivoEntrada("datos.txt");
10     //creamos el iterador del flujo para leer valores flotantes
11     istream_iterator <float> p(archivoEntrada);
12     istream_iterator <float> fin; //creamos otro iterador que indique el final del archivo
13     vector <float>almacen; //creamos un vector para almacenar lo que se genere
14     while(p!= fin){ //recorremos el archivo y guardamos en el vector
15         almacen.push_back(*p); //vamos metiendo por el final.
16         p++;
17     }
18     archivoEntrada.close(); //cerramos archivo
19     //procedemos a calcular el valor medio
20     float v_medio = 0;
21     for(unsigned i=0; i< almacen.size();i++){
22         v_medio += almacen[i];
23         v_medio = v_medio / almacen.size();
24     }
25     for(unsigned i = 0; i< almacen.size(); i++){ //restamos el valor medio a cada elemento
26         almacen[i] -= v_medio; //tambien vale almacen[i] += -v_medio;
27     }
28     ofstream archivoSalida("datos_modificados.txt"); //creamos un archivo de salida
29     ostream_iterator<float> q(archivoSalida, "\n");//iterador para poder almacenar los datos en el fichero de salida
30     for(unsigned i = 0 ;i<almacen.size(); i++, q++){ //almacenamos los datos en el archivo de salida
31         *q = almacen[i];
32     }
33      archivoSalida.close();
34     return 0;
35 }
```

Cuando se quiere crear un iterador de fin sólo se debe crear uno que no apunte a nada 'NULL' (*línea 12*).

Esto es asi porque cuando el iterador 'p' llega al final del archivo (después de leer el último elemento del mismo) se encuentra con una dirección de memoria no asignada y por lo tanto apunta al mismo lugar que 'fin' y por ende el bucle termina (*líneas 14 - 16*).

En el caso de los iteradores de escritura 'q', se debe de indicar en el constructor el nombre lógico del archivo en el que se quiere escribir y el carácter con el que se van a separar los elementos, en este caso vemos que es un salto de línea '\n' (*línea 29*).
Cuando se incrementa el iterador 'q' se imprime el carácter delimitador '\n' en el flujo de datos (*línea 30*).