Programación Orientada a Objetos

Grado en Ingeniería Informática

Biblioteca estándar de plantillas STL

Departamento de Ingeniería Informática
Universidad de Cádiz

ver 0.2

Indice

Sección 1 | Introducción

Introducción

Biblioteca estándar de plantillas STL

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++.

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

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

Sección 2 | Contenedores

Contenedores

Descripción

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

La diferencia entre un contenedor y otro es:

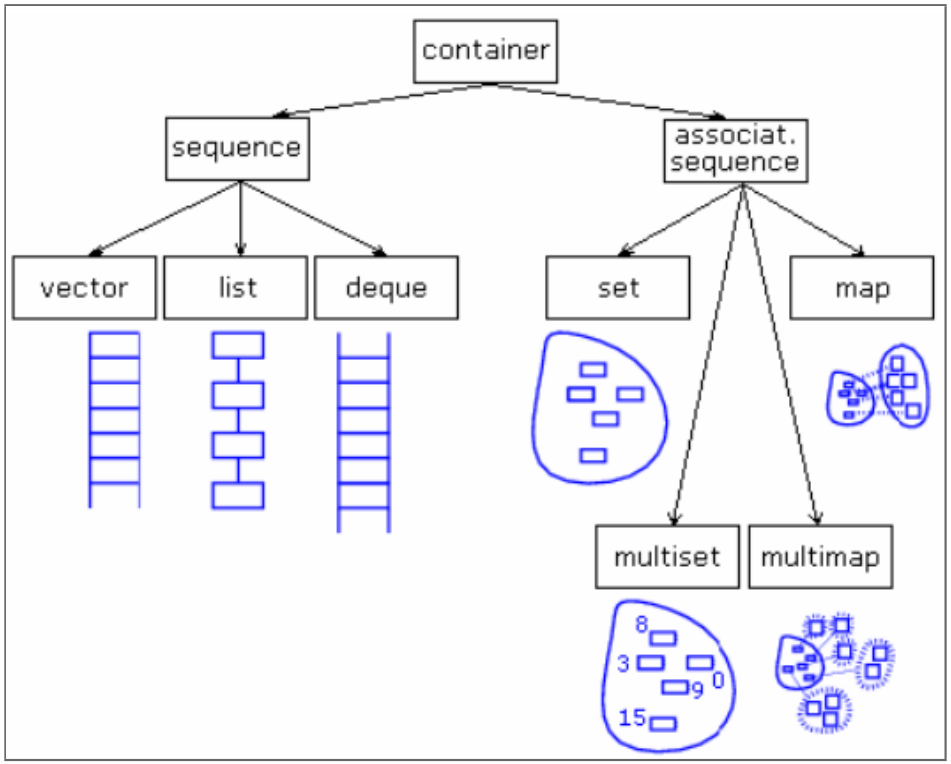
- La forma en la que los objetos **son alojados**.
- Como **se crea** la secuencia de elementos.
- La manera de **acceder** a ellos.

Contenedores

Contenedores estándar

En esta imagen podemos ver la clasificación de los contenedores de la STL. Estos se dividen en:

- contenedores de secuencia o lineales
- contenedores asociativos



Contenedores

Creación

X <T> instancia;

Donde tenemos que

- **X** representa el contenedor que vamos a utilizar
- **T** el tipo de dato que vamos a utilizar
- **instancia** el nombre que vamos a utilizar

Ejemplos

- **vector<int> valores;** Vector de enteros vacío llamado *valores*
- **vector<vector<bool> > a;** Matriz de booleanos inicialmente vacía;
- **vector<int> aux(valores);** Vector de entero copia de *valores*
- **vector<int> aux = valores;** Igual que el anterior.

Contenedores

Creación

Otra forma es construirlos por rango, suministrando un par de iteradores durante la definición. Esto permite, por ejemplo, copiar los datos de un contenedor a otro de distinta categoría:

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

En este caso tenemos que **w** es una cola doble con los elementos del vector **v**. Las funciones miembro **begin()** y **end()** permiten construir un rango que engloba a todos los elementos del contenedor y son comunes a todos los contenedores estándar.

Contenedores

Operaciones comunes

Operaciones comunes de los 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(X & 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	Supóngase que existen dos contenedores del mismo tipo: *v* y *w*. Todas las comparaciones se hacen lexicográficamente y retornan un valor booleano.
v < w v > w	
v <= w v >= w	

Contenedores

Operaciones comunes

Operaciones comunes de los 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

Contenedores

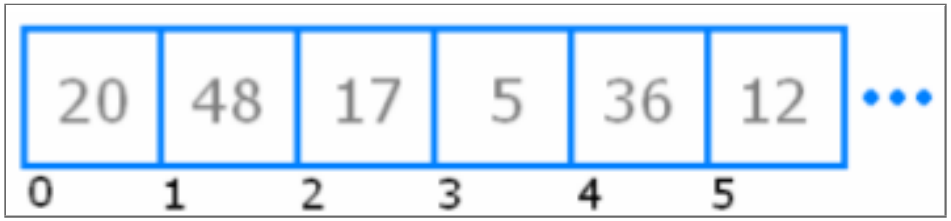
Vectores

Vectores

Contenedores

Vectores

Lo elementos contenidos **están contiguos en memoria**. Esta característica 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



Contenedores

Vectores: Ejemplo 01

```
#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;
}
```

Contenedores

Vectores: Ejemplo 02

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;
int main()
{
vector <char> valores( 10 ); // iniciar con diez elementos
// llenar con letras mayusculas al azar
for( unsigned int i=0; i<valores.size(); ++i )
valores[i] = 'A' + (rand()  % 26);
vector <char> aux( valores ); // aux es copia de valores
// ordenar aux utilizando el metodo de burbujeo
for( unsigned int i=0; i<aux.size(); ++i )
for( unsigned int 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( uint i=0; i<aux.size(); ++i )
cout << aux[i] << endl;
return 0;
}
```

Contenedores

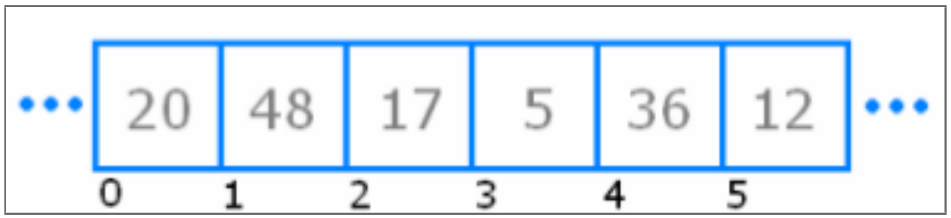
Deque

Deque

Contenedores

Deque

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



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

Contenedores

Deque: Ejemplo 01

```
#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(unsigned int 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;
}
```

Contenedores

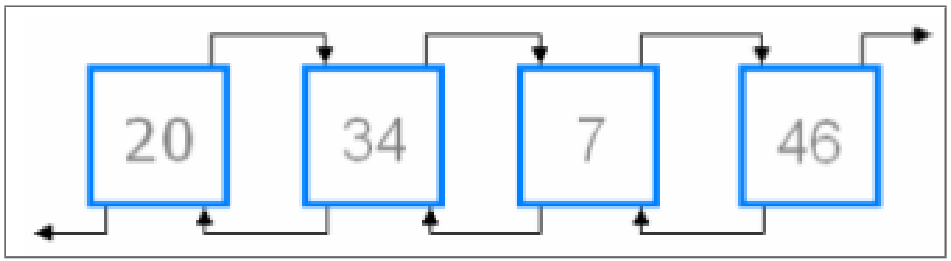
List

List

Contenedores

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.



Contenedores

List

- **Ventajas** La inserción o eliminación de un elemento se reduce a ordenar los punteros del siguiente y anterior de cada nodo.
- **Desventajas** 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**

Contenedores

Elemento para recorrer los contenedores: Iteradores

Iteradores: Recorrer contenedores

Contenedores

Elemento para recorrer los contenedores: Iteradores

Un iterador es como en 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 del recorrido: begin y end, podemos expresarlo como [begin,end)
- **begin** indica la posición del primer elemento
- **end** apunta a una posición posterior al último elemento.

Contenedores

Listas: Ejemplo 01

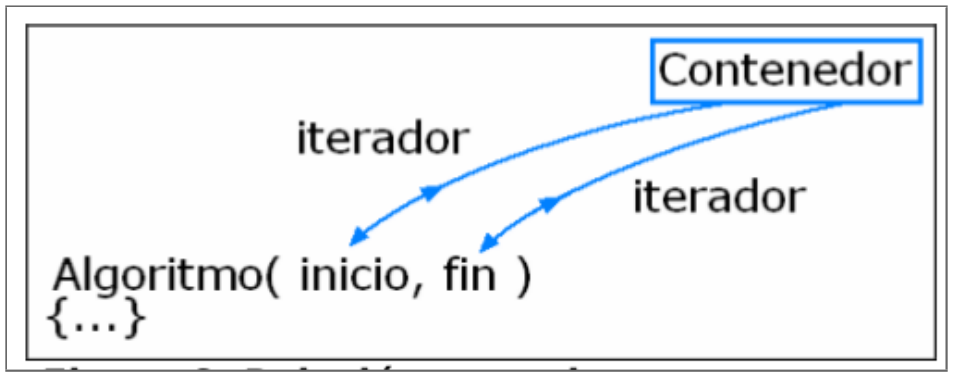
```
#include <iostream>
#include <list>
#include <cstdlib>

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;
}
```

Sección 3 | Iteradores

Iteradores

Introducción

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.



Iteradores

Sobrecarga de operadores

- 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 sin la necesidad de conocer cómo es la estructura interna de ese contenedor
- Los iteradores se pueden **clasificar atendiendo a las operaciones** que se permiten sobre ellos

Nombre	Operaciones
Entrada	$*$, $->$ (ambos sólo para lectura), $++$, $==$ y $!=$
Salida	$*$ (sólo para escritura) y $++$
Monodireccionales	Las de los iteradores de entrada y de salida
Bidireccionales	Las de los iteradores monodireccionales más $--$
Acceso directo	Las de los iteradores bidireccionales más $[]$, $<$, $<=$, $>$, $>=$ $+$ (con iter. y entero) y $-$ (con iter. y entero, o con dos iter.)

Iteradores

Creación

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();
```

Iteradores

Dependencia de los contenedores

- 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 de iteradores acceso aleatorio.

Iteradores

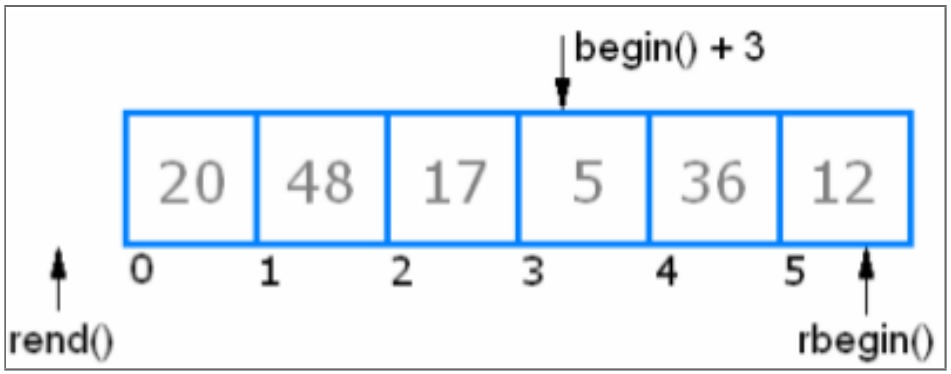
Iteradores: Ejemplo 02

```
#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 MostrarEnPantalla( Iter inicio, Iter final )
{
while( inicio != final )
cout << *inicio++ << "  ";
}
int main()
{
vector <char> letras( 20 ); // arreglo de 20 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 solo los 10 elementos del medio
MostrarEnPantalla( letras.begin() + 5, letras.begin() + 15 );
cout << endl;
return 0;
}
```

Iteradores

Explicación Ejemplo 02

- Es un ejemplo de 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
- 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 (6º elemento: begin() + 5, 16ª elemento: begin() + 15).



Iteradores

De entrada y salida

- Existen además otros tipos de iteradores que permiten manipular objetos del 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 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.

Iteradores

Iteradores: Ejemplo 03

```
#include <fstream> // para archivos
#include <iterator> // para streams_iterators
#include <vector>
using namespace std;
int main()
{
  ifstream archi( "datos.txt" ); // abrir el archivo para lectura
  istream_iterator <float> p( archi ); // crear un iterador de lectura para leer valores flotantes en el  constructor se indica a donde
      apunta
  istream_iterator <float> fin;// crear un iterador que indique el fin del archivo
  vector <float> arreglo;// crear un contenedor para almacenar lo que se lee
  while( p != fin ) // recorrer el archivo y guardar en memoria
{
arreglo.push_back( *p );
p++;
}
archi.close();
float v_medio = 0; // calcular el valor medio de los elementos del contenedor
for( unsigned i=0; i<arreglo.size(); ++i )
v_medio = v_medio + arreglo[i];
v_medio = v_medio/arreglo.size();
for( unsigned i=0; i<arreglo.size(); ++i ) // restar el valor medio a cada elemento
arreglo[i] -= v_medio;
ofstream archi2( "datos_modif.txt" ); // crear un archivo para escritura
ostream_iterator <float> q( archi2, "\n" ); // crear un iterador de escritura para guardar los datos nuevos
for( unsigned i=0; i<arreglo.size(); ++i, q++ ) // grabar los datos modificados en el archivo
*q = arreglo[i];
archi2.close();
return 0;
}
```

Iteradores

Explicación Ejemplo 03

- 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.

Sección 4 | Bibliografía

Bibliografía

Introducción

- Fundamentos de C++
- Biblioteca de Plantillas Estándar de C++ (STL) - David E. Meidina R.