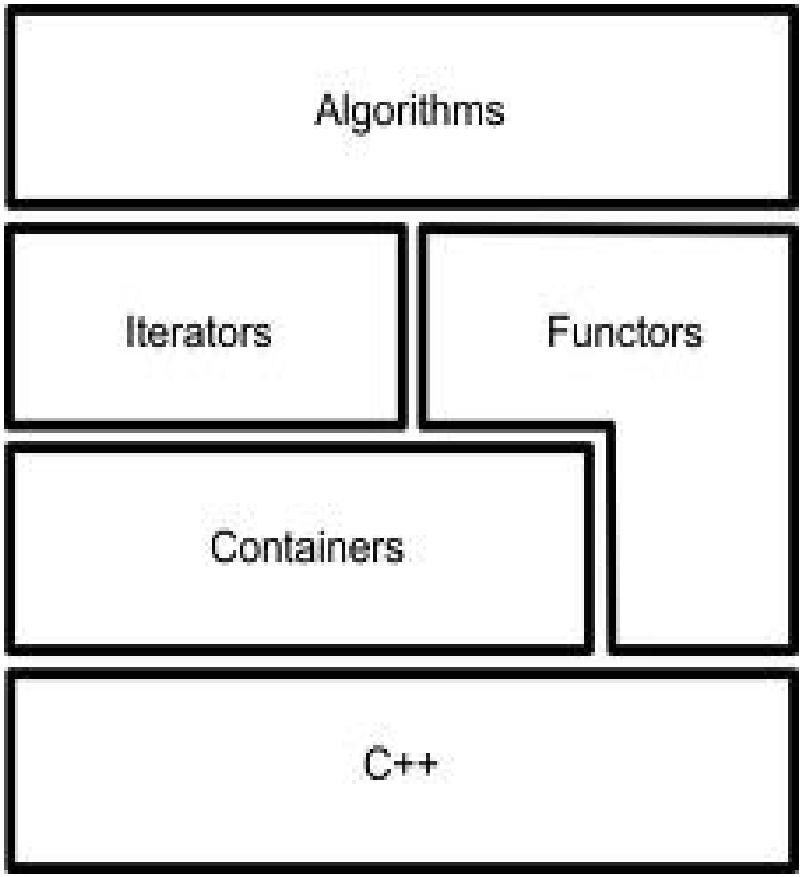
Componentes de la STL

Componentes de la STL

- Containers clases que permiten contener objetos
- Iterators hacen referencia a posiciones dentro de los contenedores
- Generic algorithms funciones que pueden trabajar en distintos contenedores
- Adaptors clases que “adaptan” otras clases
- Allocators objetos para reservar memoria



Contenedores

- Contenedores de acceso secuencial

El contenedor agrupa sus elementos como una sucesión lineal, y en consecuencia son adecuadas para accesos directos y secuenciales.

  - Vector<T>, list<T>, deque<T>, stack<T>, queue<T>
- Contenedores asociativos

Los elementos se acceden por valor, No por posición.

- Pueden utilizar un orden sobre los tipos de datos a la hora de almacenar los elementos.
  - set<T>, map<K,D>,
- No ordenados,
  - Unordered_et<T>, unordered_map<T>

Contenedores

Adaptadores

- Un adaptador de contenedor es una variación de un contenedor de secuencia o asociativo que restringe la interfaz para una mayor simplicidad y claridad.
- Los adaptadores de contenedor no admiten iteradores.
  - queue<T>, stack<T>, priority_queue<T>

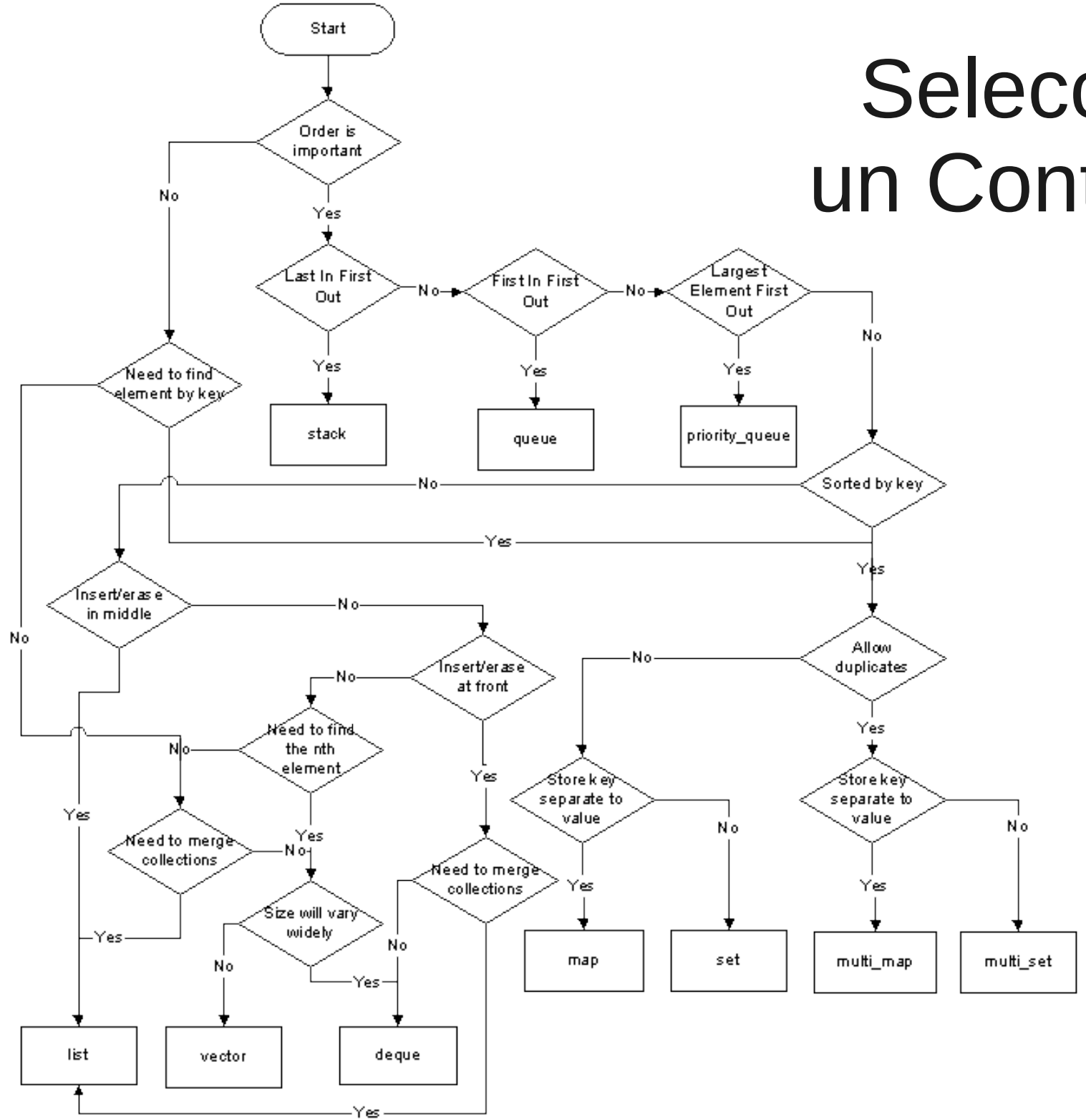
Resumen Contenedores

Contenedor	Características
Vector<T>	Acceso aleatorio, inserción eficiente en el final
list<T>	Inserción y borrado eficientes en todo el contenedor Acceso en orden lineal,
deque<T>	Acceso aleatorio. Inserción eficiente al principio y al final
set<T>	Los elementos se almacenan ordenados. Son eficientes O(log n) en la inserción, borrado y búsqueda
multiset<T>	Set que permite elementos repetidos
map<K,D>	Accede a los elementos (D) a través de su clve (K). Son eficientes O(log n) en la inserción, borrado y búsqueda
multimap<K,D>	Map que permite elementos con clave (K) repetida
string	Contenedor para cadenas de caracteres
unordered_set<T>	Los elementos NO se almacenan ordenados, Son muy eficientes (O(1) en inserción, borrado y búsqueda
unordered_map <K,D>	Accede a los elementos (D) a través de su clave (K). Son muy eficientes O(1) en la inserción, borrado y búsqueda

Resumen Adaptadores

Contenedor	Características
stack<T>	Inserciones y borrados en el final
queue<T>	Inserción en el final y borrado en el frene
priority_queue<T>	Acceso al elemento de máxima prioridad. Inserción y borrado eficientes O(log n) de elementos

Selección de un Contenedor



Iteradores

- Es un concepto fundamental para trabajar con contenedores
- Hacen referencia a posiciones del contenedor
  - Imponen un orden entre los elementos del contenedor.
    - Vector/ Map (orden creciente del valor del índice/clave)
    - Set (orden creciente del valor almacenado)
    - List (orden secuencial sobre los elementos se insertan)
    - unodered_set/unordered_map (orden “aleatorio”)
  - Cuando iteramos, las posiciones son más lógicas que físicas. Un elemento sigue a otro considerando el orden impuesto por las propiedades del contenedor.
- Cada contenedor de la STL debe proporcionar métodos begin() y end() que de forma eficiente determinan la posición del primer elemento y la posicion final (siguiente al último) del contenedor.
  - Se suele utilizar un par de iteradores para definir un rango de elementos: (ini,fin[

todos los elementos: C.begin(), C.end()

Iteradores

- En la declaración de un iterador se le asocia a contenedor concreto sobre el que itera. Su implementación depende del contenedor y permanece oculta al usuario.
  - vector<int>::iterator it1;
  - list<int>::iterator it2;
- Ofrecen métodos para recorrer, de-referenciar, y detectar los límites del contenedor al que se asocian.
- Son utilizados por muchos de los algoritmos de la STL
- Existen distintos tipos de iteracion

Tipos de Iteradores

Iterartor	Descripción
Input iterator	Sólo de lectura, movimiento hacia delante • istream_iterator
Output Iterator	Sólo de escritura, movimiento hacia delante • ostream_iterator
Forward Iterator	Lectura/Escritura, movimiento hacia delante •forward_list
Bidirectional Iterator	Lectura/Escritura, movimiento delante/atrás •List •Set/multiset •Map/multimal •unordered_set/map
Random access Iterator	Lectura/Escritura, acceso directo •Vector •deque

Pueden ser

- Constantes: Sólo permiten el acceso a los elementos, no puede ser utilizado para modificar los elementos del contenedor
- No constantes:

Propiedades

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b;` `}`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([ ])	`a[n]`

Ejemplos InputIterator: find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last,  const T& value)
{
  while (first != last && *first != value)
    ++first;
  return first;
}
```

Ejemplo Uuso:

- list<int>::iterator where = find(aList.begin(), aList.end(),7);
- vector<int>::iterator w = find(aVect.begin(), aVect.end(),7);

Ejemplo OutputIterator: copy

```
template <class InputIterator, class OutputIterator>
OutputIterator copy    (InputIterator first, InputIterator last,
                                OutputIterator result)
{ //Se asume que el destino puede almacenar todos los valores.
    while (first != last)
      *result++ = *first++;
    return result;
}
```

Ejemplo Uso:

- int data[100]; vector<int> newdata(100); list<int> L(100);
- copy(data,data+100,newdata.begin());
- copy(newdata.begin(),newdata.end(),L.begin());

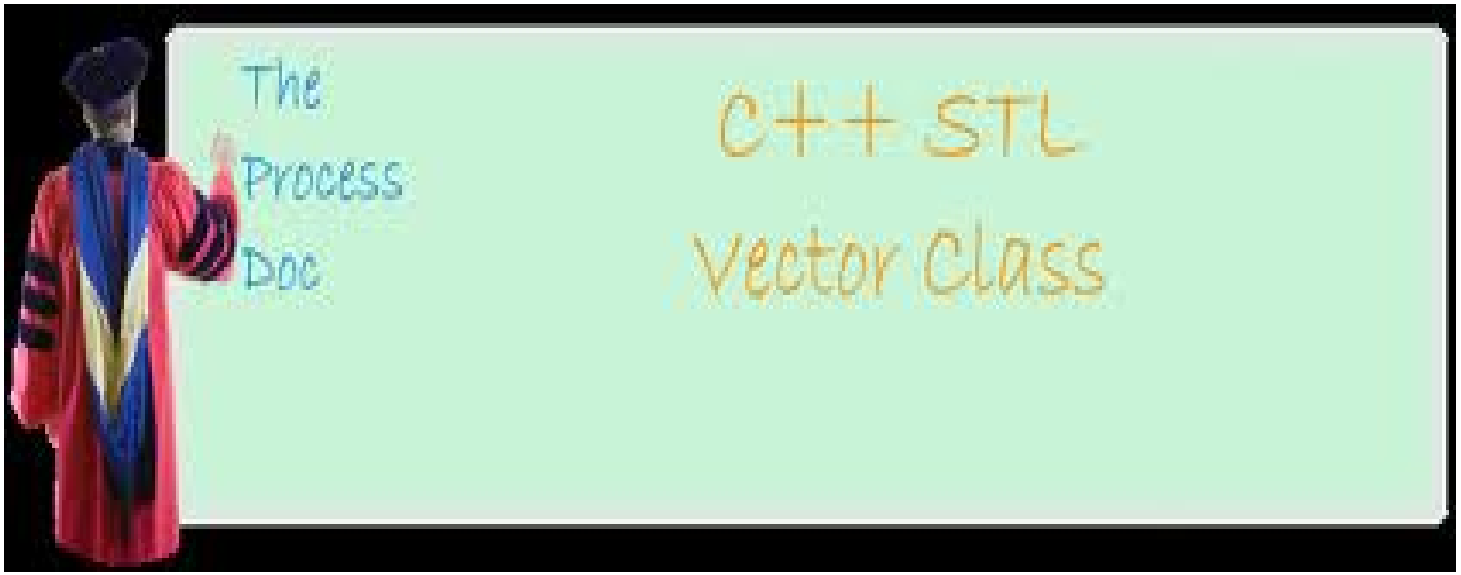
ForwardIterators: replace

```
template <class ForwardIterator, class T>
Void replace(ForwardIterator first, ForwardIterator last,
                    const T& old_value, const T& new_value)
{
   while (first != last)
   {
      if (*first == old_value)
          *first = new_value;
      ++first;
   }
}
```

Ejemplo Uso:

- vector<int> aVec;
- replace(aVec.begin(), aVec.end(), 12 , 5);

Vector STL



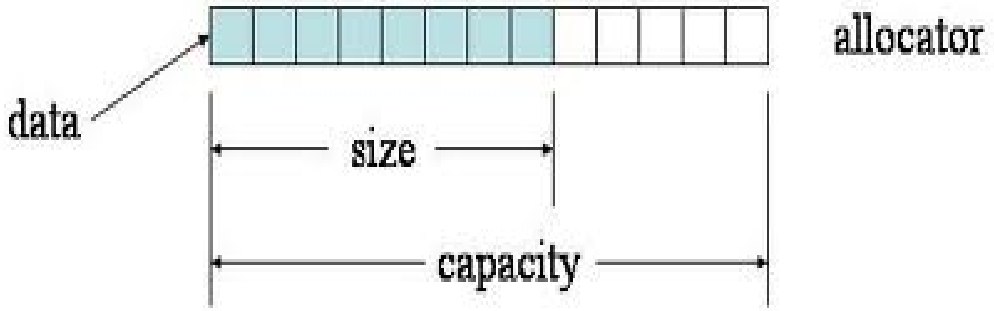
- Include File

#include <vector>

IMPORTANTE: el tipo de elementos del vector tiene que tener el constructor de copia.

STL Vector

- El vector de la STL representa un array que permite alojar elementos dinámicamente.
- Igual que el array, permite el acceso por un índice entero (0..n-1)
- Sin embargo, se diferencia en que ...
  - Da soporte para modificar el tamaño del vector en tiempo de ejecución. Los elementos se pueden insertar en el final/principio/mitad.
  - Tiene más información que un array normal, p.e. Conoce el número de elementos que tiene, cuantos puede almacenar sin realojar memoria, si está vacio, ...



STL vector: tipos

Tipo	*Descripción*
size_type	Tamaño del contenedor, unsigned int
iterator	Iterador bidireccional, acceso aleatorio
const_iterator	Iterador que no permite la modificación de los elementos almacenados
reverse_iterator	Un iterador que se mueve en sentido inverso (de atrás hacia delante)
const_reverse_iterator	Reverse iterador que no modifica los elementos
difference_type	Un entero con signo que describe distancias entre iteradores
value_type	El tipo de los elementos en el vector

STL vector: Métodos

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)

STL vector: Métodos

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

STL Vector: ejemplo

Declaración:	vector<int> X;
	vector<float> Y(100);
	vector <string> W(10,”nulo”);
	vector<string> Z(W.begin(),W.end());	

Acceso a los elementos	cout << W[1]; Y[20] = 3.5; R = Y.at(57); Cout << Y.front()<< “ “ << Y.back();	

Consultores	cout<< X.size() ;
	cout << Y.capacity() ;
	cout << X.max_size();
	If (X.empty())
		0
		100
		2305345...
		true

Ejemplo de Vectores

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

void main() {
   int Max = 100;
   vector<int> iVector(Max);
   for (int i = 0; i < Max; i++)
      iVector[i] = i;
   for (int i = 0; i < 2*Max; i++)
      iVector[i] = i;
   for (int i = 0; i < 2*Max; i++)
      iVector.at(i) = i;

}
```

Ivector[0] <- 0
Ivector[1] <- 1
Ivector[2] <- 2

.....
Ivector[99] <- 99

Error de acceso (violación de segmento

Terminado por un
std:out_of_range
What vector::_M_range_check

Accedemos como con un array: En estos casos (utilizando el operador [ ]) el **vector NO se redimensiona**

Para redimensionar hay que utilizar insert() y push_back()

Ejemplo de Vectores

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

void main() {
   vector<int> iVector;
   for (int i = 0; i < 200; i++)
   {
      iVector.push_back(i)
      Cout << iVector.size();
      Cout << iVector.capacity();
   }
}
```

size	capacity
1	1
2	2
3	4
4	4
5	8
6	8
..	...
9	16
...	...
16	16
17	32
..	...
32	32
33	64

Accedemos como con un array: En estos casos (utilizando el operador [ ]) el **vector NO se redimensiona**

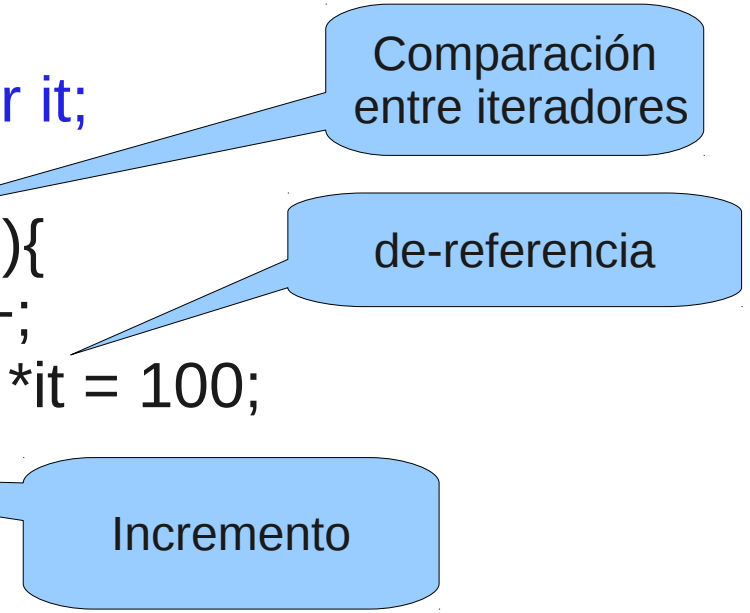
Para redimensionar hay que utilizar insert() y push_back()

vector<T>::iterator Ejemplo Uso

```
ınt main(){
{

vector<int> X(199);
....

vector<int>::iterator it;
it = X.begin();
While ( it!=X.end() ){
    if (*it==4) cont++;
    else if (*it ==10) *it = 100;
     it++;
   }
}
```



Insertar elementos en el vector

- Insertar al final del vector utlizando push_back( ) es eficiente, O(1).
- Insertar en cualquier otra posición implica desplazamientos de datos.
- Si al insertar el size()==capacity() entonces el vector dobla su capacidad. Esto implica hacer nueva reserva de memoria y realojar todos los elementos.
- Trans Insertar se INVALIDAN los iteradores .
  - Si no hay realojo de memoria, se invalidad los iteradores en posiciones posteriores al elemento insertado
  - Si hay realojo (se agranda el vector) todos los iteradores quedan invalidados

Modicando la dimensión del vector

- Es posible especificar a priori el tamaño de un vector V. Si el tamaño máximo que alcanza es conocido nos ahorraríamos continuos realojos de memoria. Con esto
  - **reserve(int n)** incrementa la capacidad del vector hasta tener una con valor mayor o igual a n. No modifica el size(), pero si capacity(). Si n es mayor que la capacidad actual del vector se realoja la memoria, si es menor no hace nada. Si hay realojo de memoria, se invalidan los iteradores: Ef. O(n)
  - **resize(int n)** el tamaño del vector. Si (n>capacity()) actual, se redimensiona el vector, incluyendo al final del vector size()-n elementos con el valor por defecto de T. Si n es menor se eliminan los n-size() ultimos elementos (en este caso no se realojan los elementos)
  - **resize(int n, T c)** Igual que resize, pero los elementos añadidos toman el valor c.

Insert() Ejemplos

- Permite insertar un elemento en una posición (dada por un iterador) del vector

```
vector<int> Y;
for (int m = 1; m < 100; m++) {
   Y.insert(Y.begin(), m);
   cout  << m  << " " << Y.capacity() ;
}
```

Y={99,98,97,96,...,1}

```
vector<int> Y;
for (int m = 1; m < 100; m++) {
   Y.insert(Y.end(), m);
   cout  << m  << " " << Y.capacity() ;
}
```

Y={1,2,3,4,...,98,99 }

size	capacity
1	1
2	2
3	4
4	4
5	8
6	8
..	...
9	16
...	...
16	16
17	32
..	...
32	32
33	64

Insert

- iterator insert(iterator position, const T& x);
  - Inserta x antes de position. El iterador que devuelve apunta al elemento insertado. El size del vector aumenta en 1. Si es necesario aumenta la capacidad. Invalida los iteradores.
- void insert(iterator position, size_type n, const T& x);
  - Inserta n copias de x antes de position.El size del vector aumenta en n. Si es necesario aumenta la capacidad. Invalida los iteradores.
- template <class InputIterator>

  void insert(iterator pos, InputIterator start, InputIterator finish);
  - Inserta copia de los elementos en el rango [start, finish) antes de pos

Erase()

- Borra elementos del vector
- Iterator erase(iterator position);
  - Borra el elemento apuntado por el iterador position. Devuelve un iterador apuntando al elemento siguiente en el vector. Decrementa el size en 1 e Invalida los iteradores
- Iterator erase(iterator start, iterator finish);
  - Borra los elementos en el rango [start, finish).Devuelve un iterador apuntando al elemento siguiente al ultimo borrado o end() si no hay elementos en el rango. Decrementa el size en el numero de elementos borrados e invalida los iteradores.

Iteradores constantes

- Se utilizan cuando el contenedor es constante **(const).** Por ejemplo, cuando se pasa como parámetro a una función

```
void ivecPrint(const vector<int> V) {

   vector<int>::const_iterator It; // DEBE ser const_iterator

   for (It = V.begin(); It != V.end(); It++)
         cout << *It;
   cout << endl;
}
```

Un const_iterator sólo permite lecturas

No se puede hacer ***it = 7**;

Mas sobre iteradores

- Los contenedores de la STL pueden tener distintos tipos de iteradores
  - Dependiendo de el sentido en el que avancen (++)
    - De izq a dcha (++ implica ->)
    - De dcha a izq (reverse) (++ implica <-)
  - dependiendo de si pueden modificar no los elementos del contenedor se clasifican en:
    - De lectura y Escritura (L: consultar ***it, E: cambiar ***it=7)
    - De sólo lectura (const) (L: consultar *it)

vector<string>::iterator: (L, E, ->).

vector<string>::const_iterator: (L, ->).

vector<string>::reverse_iterator: (L, E, <-).

vector<string>::const_reverse_iterator: (L, <-).

Mas sobre iteradores

- Los contenedores de la STL se pueden clasificar atendiendo a la forma en la que pueden recorren un contenedor en
  - Fordward: Permiten el recorrido hacia delante (it++, ++it)
  - Bidirectional: Avanzan y Retroceden (it++, ++it, it--, --it)
  - RandomAccess: (it++, ++it, it--, --it, it[ ], suma y resta)

```
vector<strring> W;
vector<string>::iterator it;
 W.insert(W.begin(), "a");  W.insert(W.begin(),"b"); W.insert(W.begin(),"c");
 W.insert(W.begin(),"d");   W.insert(W.begin(),"e");

it = W.begin();
it = it+2;
cout << *it<< endl;
cout << it[2] << endl;
```

Bidireccional

Elemento c

*(it+2): Elemento a

Comparación en contenedores

- Dos contenedores del mismo tipo son iguales si
  - Tienen el mismo tamaño
  - Los elementos entre begin() y end() son iguales uno a uno
- Es necesario que el tipo T tenga definido el operador de igualdad
    - vector<char> X(10,'a'), Y(20,'b');
    - If (X==Y) cout << “iguales”;
- También tienen implementados los operadores relacionales (<, >, . . .)
  - Es necesario que en este caso exista el operador relacional sobre el tipo T
    - If (X<Y) cout << “menor”
    - else cout “mayor”

Deque

- deque<T> es un contenedor diseñado para el almacenamiento secuencial de objetos.
  - #include <deque>
- Su nombre es una abreviatura de cola doblemente terminada (Double-Ended QUEue).
- Su funcionamiento es parecido al de vector<T>, pero con la ventaja adicional de poder añadir elementos al principio de forma eficiente.
- También es aconsejable su uso cuando el tamaño del contenedor vaya a cambiar de forma significativa.
- Posee iteradores de acceso aleatorio
  - (it++, ++it, it--, --it, it[ ], it+n, it-n)

Ejemplo deque

```
 vector<string> V_S(10,”nulo”);
deque<string> DW1,DW2;
deque<string>::iterator itd;

DW1.insert(DW1.begin(),V_S.begin(),V_S.end());
DW2.insert(DW2.begin(),DW1.begin(),DW1.end());

itd = DW.1begin();
itd= itd+2;
cout << *itd<< endl;
cout << itd[2] << endl;
```

Eficiencia

	vector<T>	**deque<T>**	**list<T>**
insertar/borrar al principio	lineal	**constante**	constante
insertar/borrar al final	constante	**constante**	constante
insertar/borrar en medio	lineal (n)	**lineal (n/2)**	constante
acceder al primer elemento	constante	**constante**	constante
acceder al último elemento	constante	**constante**	constante
acceder a un elemento intermedio	constante	**constante**	lineal

Ejeplos Vector y Deque

```
vector<int>::iterator buscar( vector<int> & V, int x)
{
bool enc=false;
vector<int>:iterator it;
for (it=V.begin();it!=V.end() && !enc; ++it)
     if (*it==x) enc = true;
return it;
}

deque<int>::iterator buscar_menor(  deque<int> & D)
{
Int m;
deque<int>::iterator it, it_m=D.begin();
if (!D.empty()) m = *it;
for (it=D.begin() ; it!=D.end(); ++it)
    if (*it_m <m) { it_m=it; m = *it;}
return it_m;
}
```

Ejemplos Vector y Deque

```
vector<int>::iterator buscar_menor( vector<int>:iterator  ini,
vector<int>::iterator  fin)
{
vector<int>::iterator   aux, it_m=ini;
if (ini!=fin) menor = *ini;
for (aux = ini; aux!=fin; aux++)
 if (*aux<menor) { it_m=aux; menor=*aux;}
return vector<int>::iterator(it_m);
}
```

Llamada

```
It = buscar_menor(X.begin(),X.end());
```

Ejemplos Vector y Deque

```
template <typename T>
typename vector<T>::iterator buscar_menor(
                                typename  vector<T>::iterator  ini,
                                typename  vector<T>::iterator  fin)
{
typename vector<T>::iterator   aux, it_m=ini;
if (ini==fin) return fin;
aux=ini;
aux++;
for ( ; aux!=fin; aux++)
 if (*aux<*it_m) { it_m=aux; }
return it_m;
}
```

Llamada

It = buscar_menor<int> (X.begin(),X.end());

```
template <class ForwardIterator>
ForwardIterator buscar_menor(  ForwardIterator ini,  ForwardIterator fin)
{
ForwardIterator  aux, it_m=ini;
if (ini==fin) return ini;
else {
it_m = aux = ini;
aux++;
while (aux!=fin)
 {
 if (*aux<*it_m)  it_m=aux;
 aux++;
 }
}
return it_m;
}
```

Llamada

```
It = buscar_menor(X.begin(),X.end());
```

Listas

- Contenedor secuencial (almacena secuencias de elementos del mismo tipo)
- A diferencia con los vectores, no permite el acceso aleatorio (random acces). No tiene operator[] ni at().
- Permite insertar y borrar elementos de cualquier posición dada por un iterador en tiempo constante.
- Es más conveniente cuando se realizan muchas inserciones en medio y no se accede de forma aleatoria.
- Inserciones y borrados NO invalidan los iteradores.
- Los iteradores son bidireccionales → no se pueden utilizar algunos algoritmos genéricos (sort)

Listas

- Tienen los mismos constructores que vectores
  - List(): **list<int> X;**
  - **list(size_type n): list<double> Y(100);**
  - **list(size_type, T x): list<string> W(20,"nulo");**
  - **....**
- Tienen los mismos métodos que los vectores excepto:
  capacity, reserve, operator[] y at
  - begin(), end(), rbegin(), rend()
  - size(), max_size(), empty()
  - Front(), back(), pus_front(), push_back(), pop_front(), pop_back()

Listas: Funciones miembro

- Inserción:
  - insert(iterator p, const T &x)
  - insert(iterator p, size_type n, const T& x)
- Borrado
  - erase(iterator p)
  - eraser(iterator ini, iterator fin)
  - Clear()
- Insercion/borrado al final hasta tamaño n
  - resize(size_type n, t=T())

Listas: Metodos

- Disponen de operaciones adicionales para transferir (splice) elementos en orden constante.
  - splice(iterator pos, list L)
  - splice(iterator pos, List L, iterator i)
  - splice(iterator pos, list L, iterator ini, iterator fin)

Ejemplo:

L1 = < a,b,c,d,e,f> L2 = < x,y,z >

it1 apunta al elemento c de L1

l1.splice(it1, L2)

L1 = < a,b,x,y,z,c,d,e,f>

Lilstas: Metodos

- Remove (const T & v)
  - Elimina todos las ocurrencias de v
- Unique()
  - elimina ocurrencias de elementos iguales consecutivos (menos el primero)
- merge(list & L)
  - Mezcla de listas
- Sort()
  - Ordena una lista.

Ejemplos:

- Implementar los metodos siguientes, como funciones externas a la lista.
  - Int borrar(list<T> L, T x)
    - Borra todas las ocurrencias de x en L y devuelve numero de elementos borrados
  - pair<list<T>::iterator, int> repe_consecutivos (const list<T> & L)
    - Busca la mayor secuencia de elementos consecutivos iguales y devuelve el iterador al principio y el numero de elementos
  - list<T> mezclar(list<T> & L1, list<T> & L2)
    - Mezclar dos listas en una tercera.
-

Eficiencia

	vector<T>	**deque<T>**	**list<T>**
insertar/borrar al principio	lineal	**constante**	constante
insertar/borrar al final	constante	**constante**	constante
insertar/borrar en medio	lineal (n)	**lineal (n/2)**	constante
acceder al primer elemento	constante	**constante**	constante
acceder al último elemento	constante	**constante**	constante
acceder al elemento k-ésimo	constante	**constante**	lineal

Contendores

- Contenedores de acceso secuencial

El contenedor agrupa sus elementos como una sucesión lineal, y en consecuencia son adecuadas para accesos directos y secuenciales.

  - Vector<T>, list<T>, deque<T>, stack<T>, queue<T>

- Contenedores asociativos

Utiliza un orden sobre los tipos de datos a la hora de almacenar los elementos.

  - set<T>, map<K,D>,
  - Iteradores bidireccionales

SET

- Un conjunto representa una colección de elementos de tipo K. Sobre estos elementos debe existir una relación de orden
  - Si a, b y c son elementos de tipo K entonces
    - Se puede ver si a < b
    - Si a < b && b < c Entonces a < c
    - Si ! (a<b) y !(b<a) Entonce a==b
- El set<K> almacena los elementos ordenados y por tanto las operaciones de insercion, busqueda y borrado de elementos estan optimizados (O(log n).
- NO se pueden almacenar elementos repetidos

Plantillas: set<K,Compara>

- K es un tipo para el que existe una relacion de orden total
- Compara es la funcion de comparacion. Debe generar un orden sobre los elementos de la clave.

Set<K,compara> Miembros

- Constructores:
  - Set()
    - set<int> S1;
  - set(iterator ini, iterator fin)
    - set<int> S2(S1.begin(),S1.end());
  - Set()
    - set<int, greater<int> > S2;
  - set(iterator ini, iterator fin)
    - set<int,greater<int> > S2(S1.begin(),S1.end());

```
class ltstr {
  public:
   bool operator()(const string & s1, const string & s2) const;
};
bool ltstr::operator()(const string & s1, const string & s2) const
   {  return (s1>s2); }
int main() {
     set<string,greater<string> > W;
     set<string,ltstr> S2;
```

Set Metodos

- begin(), end(), rbegin(), rend()
- clear()
- Count(), size(), maz_size(), empty()
- erase()
- find()
- insert()
- equal_range(), lower_bound(), upper_bound()
- max_size()
- operator=()

....

Eficiencia

	vector<T>	**set<K>**	**list<T>**
insertar/borrar al principio	lineal	**log(n)**	constante
insertar/borrar al final	constante	**log(n)**	constante
insertar/borrar en medio	lineal (n)	**log(n)**	constante
acceder al primer elemento	constante	**log(n)**	constante
acceder al último elemento	constante	**log(n)**	constante
acceder al elemento k-ésimo	constante	**lineal**	lineal

Ejemplos

```
pair< set<int>::iterator,set<int>::iterator >   buscar(const set<int> & s,
                                                       int x, int y)
{
pair<set<int>::iterator,set<int>::iterator>  aux;
aux.first = aux.second = s.end();
set<int>::iterator it;
for (it = s.begin();it!=s.end();it++){
     if (*it==x) aux.first = it;
     if (*it==y) aux.second = it;
     }
return aux;
}
```

```
int main(){
set<int> X;
for (int i = 1; i<100; i++)
 X.insert(i);
 pair<set<int>::iterator,set<int>::iterator>  aux;
 aux = buscar(X,10,20);
 cout << *(aux.first) << "++"<< *(aux.second)<< endl;
```

Ejemplos set

```
void elimina_multiplos (set<int> & s, int x){
set<int>::iterator it,aux;

for (it =s.begin(); it!=s.end(); )
  if (*it%x==0) {
     aux = it;
     it++;
     s.erase(aux);
     }
  else it++;

}
```

Set: Problemas

- Int contar(const set<string> & s, const string & a, const string & b);
  - Cuenta cuantos elementos hay entre [a,b)
- Void diferencia(const set<int> & S1, const set<int> & S2)
  - Diferencia entre conjuntos
- set<int> interseccion(const set<int> & a, const set<int> & b);
  - Devuelve la intersección de dos conjuntos

SET y relación de orden

- Si la clase T con la que se particulariza el set tiene definido el operator< podemos considerar el template less<T> para construir funciones objeto (implicito). Debemos de incluir functional)
- Si se reguiere un orden, por defecto la STL considera el operador less<T>, por lo que no hay que nacer nada especial ...

set<int> X; === set<int,less<int> > X;

- Si necesitamos utilizar un orden decreciente, el tipo T debe tener definido el operator> considerando en este caso el template greater<T> para construir funciones

set<int,greater<int> > Y;

SET con clases propias: Orden

```
class alumno {

 public:
   alumno();
   void setCalif(double c);
   double getCalif( ) const;
   void setNombre( const string & n);
   string getNombre( ) const;
   void setDNI(const string & d);
   string getDNI() const;
   bool operator<( const alumno & a) const;
   bool operator>( const alumno & a) const;
 private:
   double _calif;
   string _nombre;
   string _dni;
  };
```

```
void alumno::setCalif(double c){
  _calif = c;
  }
double alumno::getCalif( ) const {
   return _calif;
   }
void alumno::setDNI(const string & d){
    _dni = d;
    }
string alumno::getDNI() const {
      return _dni;
    }
bool alumno::operator<( const alumno & a) const
    {
    return _nombre < a._nombre;
    }
bool alumno::operator>( const alumno & a) const
    {
    return _nombre > a._nombre;
    }
```

```
set<alumno> Screc;    // set<alumno,less<alumno> >
alumno a;
a.setDNI("123"); a.setNombre("Juan Lopez");
Screc.insert(a);
a.setDNI("2323"); a.setNombre("Pedro Lopez");
Screc.insert(a);
a.setDNI("1090"); a.setNombre("Ramon Sol");
Screc.insert(a);
```

Screc= { (Juan Lopez, 123), (Pedro Lopez, 2323), (Ramon Sol, 1090) }

```
set<alumno,greater<alumno> > Sdc(SA.begin(),SA.end());
```

Sdc={ (Ramon Sol, 1090), (Pedro Lopez, 2323), (Juan Lopez, 123) }

Utilizando functor

```
class comp_dni {
public:
  bool operator()(const alumno & izq, const alumno & dch);
 };

 bool comp_dni::operator()(const alumno & izq, const alumno & dch)
 {
 return izq.getDNI() < dch.getDNI();
 }
```

```
set<alumno,comp_dni > Sdni(SA.begin(),SA.end());
```

Sdni={ (Ramon Sol, 1090), (Juan Lopez, 123), (Pedro Lopez, 2323) }

Diccionarios map<key,Def>

- Un diccionario (map) es un contenedor que asocia objetos del tipo Key con los objetos de tipo Def.
- Los elementos se almacenan de forma ordenada según Key. Debe existir una relacion de orden definida sobre Key.
  - Eje: map<nombre,telefono> guia;
- Al igual que set, es un contenedor asociativo único, esto es, no hay dos elementos en el map que tengan la misma clave
- En un map al insertar y borrar elementos NO invalida los iteradores que apuntan a los objetos existentes.
- Los iteradores son bidireccionales
- Permite acceso eficiente a los elementos por Key, (O(log n)),

Map e iterator

- Un map<key,def>::iterator es un iterador que apunta a elementos dele tipo pair<key,def>
- Así

```
    map<string,int> dias;
....
    It =dias.begin();
  Cout << (*it).first << (*it).second << endl
```

Diccionarios map<key,Def>

Constructores

- Constructor por defecto map( ):
  - map<string,int> X;
- Constructor de rango map (Initerator first, Initerator last):
  - map<string,int> Y (X.begin(),X.end());
- Constructor de copia

map<Key,Def> Metodos

- Insercion:
  - pair<iterator, bool> insert(const pair<clave,datos>& x)
  - Iterator insert(iterator pos, const pair<clave,datos>& x)
  - void insert(iterator ini, iterator fin);
- Borrado
  - void erase (iterator position);
  - Int erase (const key_type& x);
  - void erase (iterator first, iterator last);

map<Key,Def> .... operator[]

- Inserción/consulta
  - Def & operator[] (const key_type& x);

  Si la clave x esta en el contenedor, la función devuelve una referencia a la definicion asociada.
  - Cout << dias[“Junio”] ;
- Si x no esta en el contenedor, inserta un elemento con esa clave y la definicion al valor dado por el constructor por defecto Def(). Devuelve la referencia a la definicion. Incrementa el tamaño en 1.
  - dias[“enero”];
  - dias[“febrero”]=28;
- Eficiencia: Logaritmica

map<key,def> otros metodos

iterator begin(), end(), rbegin(), rend()

size_type size(), max_size(), count(key)

bool empty()

iterator find (key), lower_bound (key), upper_bound (key),

pair<iterator, iterator> equal_range (key)

map<Key,Def> Metodos

- Insercion:
  - pair<iterator, bool> insert(const pair<clave,datos>& x)
  - Iterator insert(iterator pos, const pair<clave,datos>& x)
  - void insert(iterator ini, iterator fin);
- Borrado
  - void erase (iterator position);
  - Int erase (const key_type& x);
  - void erase (iterator first, iterator last);

map<Key,Def> Metodos

```
int main ()
{
  map<char,int> mymap;
  map<char,int>::iterator it;
  mymap['b'] = 100;   mymap['a'] = 200;   mymap['c'] = 300;
  // Ver el contenido
  for ( it=mymap.begin() ; it != mymap.end(); it++ )
    cout << (*it).first << " => " << (*it).second << endl;
}
```

El operator* del iterador accede al elemento que una variable del tipo pair<const Key,Def>.

Ejemplos

- Implementar un contador de frecuencia de aparición de palabras en un texto
- Implementar una guía telefónica
- Listar todos los elementos que tengan la misma definicion

Adaptadores

Contenedor	Características
stack<T>	Inserciones y borrados en el final
queue<T>	Inserción en el final y borrado en el frene
priority_queue<T>	Acceso al elemento de máxima prioridad. Inserción y borrado eficientes O(log n) de elementos

Stack<T>

- La pilas son un tipo de contenedor especilmente diseñado para trabajar en situaciones donde la entrada/salida de datos siga un proceso LIFO (last-in first-out)
- Los elementos se insertan o extraen por el extremo
- Se pueden implementar como un adaptador de otra clase

Funciones miembro

- (constructor)
- empty dice si la pila esta vacia
- Size Devuelve el numero de elementos en la pila
- top Consulta el elemento en el tope de la pila
- push Inserta un nuevo elemento
- pop Borra el elemento.
- Otras: clear, swap.

Uso

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

int main ()
{
  stack<int> mipila;

  for (int i=0; i<5; ++i) mipila.push(i);
  mipila.top()=15;
  cout << "Saco los elementos...";
  while (!mipila.empty())
  {
     cout << " " << mipila.top();
     mipila.pop();
  }
  cout << endl;
}
```

Implementación

```
#ifndef __STACK__
#define __STACK__

#include <list>

using std::list;

template <class T>
class stack {
public:

  typedef typename list<T>::size_type size_type;
  stack();
  stack(const stack<T> & p);
  bool empty() const;
  void clear();
  void swap (stack<T> & p);
  T& top ();
  const T& top () const;
  void push(const T & ele
  void pop();
  size_type size() const;
  ~stack();

private:
  list<T> lapila;
};

#include "stack.template"

#endif
```

stack.template

```
/*

  Función de Abstracción:
  ----------------------
  Dado el objeto del tipo rep r, el objeto abstracto al que
  representa es:
    p = lapila, con el tope de la pila en la posición
        lapila.back().

  Invariante de Representación:
  ----------------------------
  - true.
 */
```

stack.template

```
template <class T>
inline
stack<T>::stack()
 : lapila()
{
}

template <class T>
inline
stack<T>::stack(const stack<T> & p)
  : lapila(p.lapila)
{
}
```

stack.template

```
template <class T>
inline T&
stack<T>::top()
{
  return lapila.back();
}

template <class T>
inline const T&
stack<T>::top() const
{
  return lapila.back();
}
```

```
template <class T>
inline void
stack<T>::push(const T & elem)
{
  lapila.push_back(elem);
}

template <class T>
inline void
stack<T>::pop()
{
  lapila.pop_back();
}

template <class T>
inline
stack<T>::~stack()
{
}
```

stack.template

```
ttemplate <class T>
inline bool
stack<T>::empty () const
{
  return lapila.empty();
}
```

```
template <class T>
inline void
stack<T>::swap(stack<T> & p)
{
  lapila.swap(p.lapila);
}
```

```
template <class T>
inline void
stack<T>::clear()
{
  lapila.clear();
}
```

```
template <class T>
inline typename stack<T>::size_type
stack<T>::size() const
{
  return lapila.size();
}
```

queue

- Métodos::queue, empty, clear, front, back. push

  pop, size, swap, ~queue

Cada objeto del TDA queue, modela una cola de elementos de la clase T.

Una cola es un tipo particular de secuencia en la que los elementos se insertan por un extremo (final) y se consultan y suprimen por el otro (frente). Son secuencias del tipo FIFO (First In, First Out).

Son objetos mutables.

Residen en memoria dinámica.

Ejemplos

```
#include <queue>
using namespace std;

int main ()
{
  queue<int> myqueue;

  myqueue.push(77);
  myqueue.push(16);

  myqueue.front() -= myqueue.back();    // 77-16=61

  cout << "myqueue.front() is now " << myqueue.front() << endl;

  return 0;
}
```