vector<T>

Descripción

`Un vector` es un contenedor que permite el acceso aleatorio (mediante valores enteoros) a los elementos, inserción y borrado en tiempo constante si se realiza al final del vector. Sin embargo, la inserción y el borrado son de orden lineal si se realizan en posiciones intermedias del vector. El número de elementos en el vector puede variar dinámicamente; la gestión de la memoria es automática. `El vector` es el contenedor más simple de la STL, y en muchos casos el más eficiente.

Ejemplo

```
#include <vector>

int main(){
    int a;

     vector<int> X(10);
     cout << "capacidad " << X.capacity() << endl;
     cout << "size " << X.size() << endl;
     X.push_back(1);
     X.push_back(2);
     X.push_back(3);
     X.insert(X.begin(),7);

     vector<int>::iterator it;
     for (it = X.begin(); it!= X.end(); ++it)
       cout <<*it << " ";
       cout << endl;
     cout << "capacidad " << X.capacity() << endl;
     cout << "size " << X.size() << endl;
     vector<char> Y(100, 'a');
     cout << "capacidad " << Y.capacity()<<endl;
     cout << "size " << Y.size() << endl;

     cin >> a;
     }
```

Definición

Definido en el fichero cabecera <vector> y en <vector.h>

Parámetros de la plantilla

Parámetro	Descripción	Defecto
`T`	El tipo del valor del vector: el tipo de objeto que se almacena en el vector.	
`Alloc`	El allocator del `vector`, usado para toda la gerencia interna de la memoria.	`alloc`

Tipos y Miembros

Tipo		Descripción
`size_type`		Entero sin signo
`iterator`		Iterator sobre `vector`.
`const_iterator`		El iterator constante de un `vector`.
`reverse_iterator`		Iterator que recorre en orden inverso un `vector`.
`const_reverse_iterator`		Iterator constante que recorre en orden inverso un `vector`.
Miembros	**Eficiencia**	**Descripción**
`iterator begin()`	O(1)	Devuelve un `iterator` que señala al principio del `vector`.
`iterator end()`	O(1)	Devuelve un `iterator` que señala al final del `vector`.
`const_iterator begin() const`	O(1)	Devuelve un `const_iterator` que señala al principio del `vector`.
`const_iterator end() const`	O(1)	Devuelve un `const_iterator` que señala al final del `vector`.
`reverse_iterator rbegin()`	O(1)	Devuelve un `reverse_iterator` que señala al principio del vector en orden inverso.
`reverse_iterat`	O(1)	Devuelve un `reverse_iterator` que señala al final del vector en orden

`or rend()`		inverso.
`const_reverse_iterator rbegin() const`	O(1)	Devuelve un `const_reverse_iterator` que señala al principio del vector en orden inverso.
`const_reverse_iterator rend() const`	O(1)	Devuelve un `const_reverse_iterator` que señala al final del vector en orden inverso.
`size_type size() const`	X	Devuelve el número de elementos del `vector`.
`size_type max_size() const`	O(1)	Devuelve el tamaño más grande posible de la `vector`.
`size_type capacity() const`	O(1)	Número de elementos para los que se ha asignado memoria. La `capacidad()` es siempre mayor o igual `tamaño ()`
`bool empty() const`	O(1)	`verdad` si el tamaño del `vector` es `0`.
`T & operator[](size_type n)`	O(1)	Devuelve una referencia al elemento `n'th del vector.` `Precondiciones: n < capacity()`
`const T & operator[](size_type n) const`	O(1)	Devuelve una referencia constante al elemento `n'th del vector` `Precondiciones: n < capacity()`

`vector()`	O(1)	Crea un vector vacío.
`vector(size_type n)`	O(n)	Crea un vector con n elementos, con valor T( ).
`vector(size_type n, const T& t)`	O(n)	Crea un vector con n copias de t.
`vector(const vector&)`	O(n)	El constructor de copia.
`~vector()`	O(1)	El destructor.
`vector& operator=(const vector&)`	O(n)	El operador de asignación
`void reserve(size_t n)`	O(n)	Si es `n` inferior o igual `capacidad()`, esta llamada no tiene ningún efecto. Si no, se realizar una petición de memoria. Si dicha petición se realiza con éxito entonces la `capacidad ()` será mayor o igual `n`; si no, la `capacidad ()` no se modifica En cualquier caso, el `tamaño ()` del vector no se modifica.
`T & front()`	O(1)	Devuelve una referencia al primer elemento.
`const T & front() const`	O(1)	Devuelve una referencia constante al primer elemento.
`T & back()`	O(1)	Devuelve una referencia al último elemento.
`const T & back() const`	O(1)	Devuelve una referencia constante al último elemento.
`void push_ba`	O(1)	Inserta un nuevo elemento al final del vector. Incrementa en 1 el size del vector. En caso de ser necesario se redimensiona la capacity del vector, siendo esta

`ck(cons t T&)`		mayor que size().
`void pop_bac k()`	O(1)	Elimina el elemento al final del vector
`void swap(ve ctor&)`	O(1)	Intercambiar el contenido de dos vectores.
`iterator insert(i terator pos, const T& x)`	O(n)	Inserta el elemento x delante de pos. Al insertar el elemento, todos los elementos se desplazan una posición adelante. Se invalidan los iteradores. Se incrementa en 1 el size()
`void insert(i terator pos, size_typ e n, const T& x)`	O(n)	Inserta n copias `de x` antes de `pos.` Al insertar el elemento, todos los elementos se desplazan una posición adelante. Se invalidan los iteradores. Se incrementa en 1 el size()
`iterato r erase(i terator pos)`	O(n)	Borra el elemento en la `posición pos` Al borrar el elemento, todos los elementos se desplazan una posición hacia atrás. Se invalidan los iteradores. Se decrementa en 1 el size()
`iterato r erase(i terator first, iterato r last)`	O(n)	Borra los elementos en el rango `[first, last)` Al borrar todos los elementos se desplazan hacia atrás tantas posiciones como sea necesario. Se invalidan los iteradores. Se decrementa e el size()
`void clear()`	O(1)	Borra todos los elementos.
`void resize( n, t = T())`		Inserta o borra elementos en el final tales que el tamaño se convierte en `N.`
`bool operator ==(const vector&, const vector&)`	O(n)	Pruebas dos vectores para la igualdad. Esto es una función global, no una función miembro.

`bool operator<(const vector&, const vector&)`	O(n)	Comparación lexicográfica. Esto es una función global, no una función miembro.

Notas

[1] Los iteradores que apuntan a un vector se invalidan cuando se reasigna la memoria. Además, insertar o suprimir un elemento en posiciones intermedias del vector invalida todos los iterators que señalen a los elementos que siguen detrás del punto de inserción o borrado.

[2] La memoria será reasignada automáticamente si más elementos que `capacidad() - size()` se insertan en el vector. La reasignación no cambia `size()`, ni cambia los valores de cualquier elemento del vector. Sin embargo, aumenta `capacidad()`, e invalida cualquier iterador que señale en el vector.

[3] Cuando es necesario aumentar `capacidad()`,el `vector` la aumenta generalmente en un factor de dos(la duplica). Es crucial que el crecimiento sea proporcional a la `capacidad()` ,actual más que una constante fija. Al ser proporcional se garantiza que la inserción de un secuencia de elementos se haga con un orden lineal, miemtras que si el crecimiento es constante la inserción se convierte en cuadrática

[4] `reserve()` implica una reasignación manual de la memoria. La razón principal para usar `reserve()` es la eficiencia: si se conoce la capacidad a la cual un `vector` debe crecer , entonce es generalmente más eficiente asignar esa memoria de una vez en lugar de confiar en el esquema automático de la reasignación. Otra razón para usar la `reserve()` es el permitir que se puedan controlar los iteradores