Lección 8: Introducción a STL. Clases vector, deque y list

- Motivación
- La Standard Template Library (STL)
- Contenedores
- Secuencias
- Iteradores
- El contenedor *vector*
- El contenedor *deque*
- El contenedor *list*

Motivación

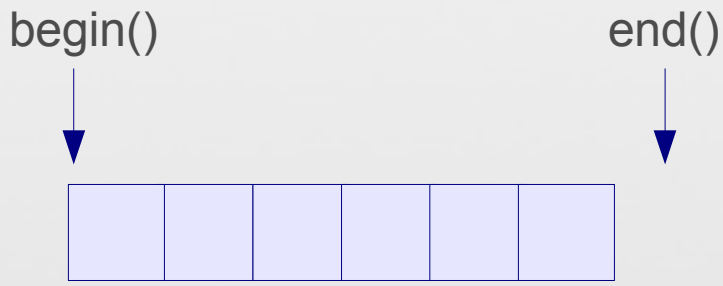
- Es importante conocer las distintas estructuras de datos y su funcionamiento pero...
- ¡No tiene sentido que cada desarrollador implemente su propia biblioteca de EEDD!
- Todos los compiladores de C++ incluyen la biblioteca STL que incluye todas las EEDD básicas
- Implementación de altísima calidad, eficiente y libre de errores
- Todos los programadores de C++ deben conocer y utilizar STL

La Standard Template Library (STL)

- Incluye patrones de clases y algoritmos genéricos con un amplio espectro de utilización
- Nos interesan los contenedores y en esta lección estudiaremos tres: **vector**, **deque** y **list**
- vector, deque y list son **contenedores secuenciales**
- STL utiliza igualmente **iteradores** para acceder cómodamente a los contenedores
- Documentación:
  - http://www.cplusplus.com/reference/stl/

Contenedores de STL

- Todos los contenedores de STL implementan obligatoriamente:
  - Constructor por defecto y constructor copia
  - Operador de asignación (=)
  - Operación *size()* para obtener el número de elementos, y *empty()* para conocer si está vacío
  - Operaciones *begin()* y *end()* para obtener un iterador apuntando al principio y al final respectivamente



Secuencias de STL

- Además de la funcionalidad de cualquier contenedor, implementan:
  - Constructor *(int n, T dato)* que inicia con n copias del dato T y constructor *(iterator i, iterator j)* que inicia copiando los datos de otro contenedor comprendidos entre i y j
  - *front()* y *back()* que devuelven el primer y último elemento respectivamente
  - Inserción: *insert(iterator pos, T dato)*, *insert(iterator pos, int copias, T dato)*, *insert(iterator pos, iterator i, iterator j)*
  - Borrado: *erase(iterator pos)* y *erase(iterator i, iterator j)*
  - Borrado completo: *clear()*

Iteradores de STL

- Todos los iteradores de STL implementan:
  - Constructor por defecto y constructor copia
  - Operador de asignación (=)
  - Operaciones de comparación == y != para comprobar si dos iteradores apuntan a la misma posición
  - Operador * para obtener el dato apuntado por el iterador y/o modificarlo
  - Operadores ++ y -- para mover el iterador al siguiente elemento
- **Muy importante:** el iterador devuelto por end() no es el último elemento del contenedor sino la marca de final. Nunca acceder a su contenido! (similar a un puntero nulo)

El contenedor vector

- Es una implementación de un vector dinámico similar a las que conocemos
- Implementa el operador [] para acceder directamente a cualquier elemento indicando la posición (entre 0 y vector.size() - 1)
- Implementa las operaciones *push_back()* y *pop_back()* para añadir y eliminar datos por el final (modifica el tamaño del vector)
- Declaración:

```
#include <vector>

std::vector<int> miVector;
```

Iteradores sobre vectores

- Declaración y ejemplo de iniciación:

```
std::vector<int>::iterator  miIterador;
miIterador = miVector.begin()
```

- Admiten saltos adelante y detras:

```
miIterador += 5
otroIterador = miIterador + 10
```

- Ojo! los iteradores sobre un vector quedan invalidados si se realizan inserciones y borrados en éste!

Ejemplo de uso

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

int main() {
    vector<int> v;

    for (int c = 2; c < 100; c++)
        v.push_back(c);

    // Acceso por posición
    for (int c = 0; c < v.size(); c++)
        cout << v[c] << endl;
```

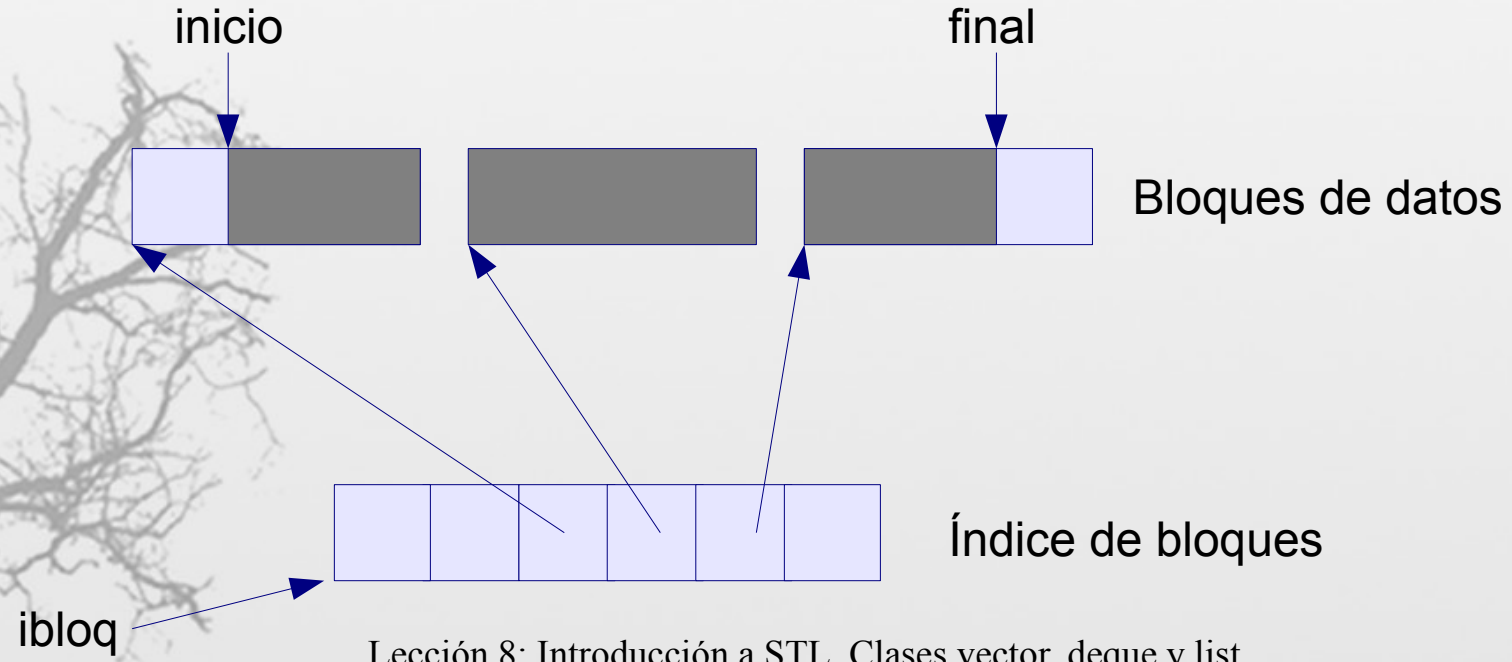
```
    // Acceso mediante iteradores
    vector<int>::iterator i;
    i = v.begin();
    while (i != v.end()) {
        cout << *i << endl;
        ++i;
    }

    // Inserciones al principio
    v.insert(v.begin(), 1);
    v.insert(v.begin(), 0);

    // Borrar el número 5
    i = v.begin() + 5;
    v.erase(i);
}
```

El contenedor deque

- Es una evolución del vector dinámico tradicional que admite inserciones y borrados por el principio en Θ(1)
- Buena alternativa a las listas simplemente enlazadas para ciertas aplicaciones



El contenedor deque (cont.)

- También soporta el operador []
- Incorpora además *push_front()* y *pop_front()* para inserciones y borrados por el principio
- Las inserciones en posiciones intermedias son muy ineficientes
- Declaraciones:

```
#include <deque>

std::deque<int> miDeque;
std::deque<int>::iterator miIterador;
```

Ejemplo de uso

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

int main() {
    deque<int> v;

    for (int c = 0; c < 100; c++) {
        v.push_back(c);
        v.push_front(c);
    }

    // Acceso por posición
    for (int c = 0; c < 200; c++)
        cout << v[c] << endl;
```

```
    // Acceso mediante iteradores
    // Bucle inverso
    deque<int>::iterator i = v.end();
    do {
        --i;
        cout << *i << endl;
    } while (i != v.begin());

    // Modificación mediante iteradores
    i = v.begin();
    while (i != v.end()) {
        *i = 0;
        ++i;
    }
}
```

El contenedor list (cont.)

- Es una lista doblemente enlazada
- Implementación muy sofisticada y eficiente:
  - Peticiones de memoria por bloques de nodos en lugar de individualmente
  - Reutilización de nodos borrados
- No soporta el operador [] (acceso sólo mediante iteradores)
- Los iteradores no admiten saltos (sólo avance/retroceso nodo a nodo)
- Incluye *push_front/back()* y *pop_front/back()*

Ejemplo:

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

int main() {
    list<int> l;

    // Insertar los 100 primeros
    // términos de la sucesión
    // de Fibonacci
    l.push_back(0);
    l.push_back(1);

    list<int>::iterator i1;
    i1 = l.begin();
    list<int>::iterator i2;
    i2 = l.begin();
    ++i2;
```

```
    for (int c = 2; c <= 100; c++) {
        l.push_back(*i1 + *i2);
        ++i1; ++i2;
    }

    // Borrar términos impares
    i1 = l.begin();
    while (i1 != l.end()) {
        if (*i1 % 2 == 1) {
            l.erase(i1++);
        }
        else
            ++i1;
    }
```