Programación 2
Estructuras Lineales

Fernando Orejas

1. Estructuras lineales
2. Pilas
3. Colas
4.Listas

Objetivos

1. Estudiar algunas estructuras de datos
2. Ver cómo podemos construir programas que usan clases predefinidas

Estructuras lineales

Estructuras lineales

- Son estructuras de datos que contienen secuencias de valores
- Los accesos típicos que podemos tener son
  - Al primer elemento
  - Al último elemento
  - Al siguiente elemento
  - Al anterior elemento
- Las modificaciones típicas son inserciones o supresiones que pueden estar limitadas a los extremos

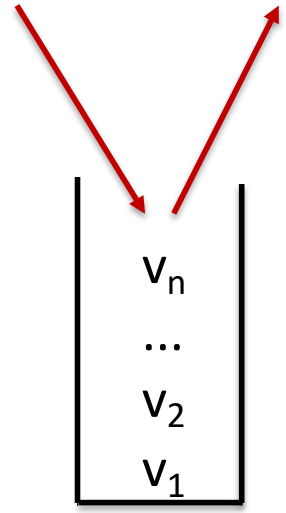
Estructuras lineales

- Ejemplos típicos son pilas, colas, deques, listas, listas con prioridades, etc.
- En la STL de C++: stack, queue, deque, list, priority list, etc.
- Son *templates* (tienen un tipo como parámetro)
- Son contenedores (*containers*)

Pilas

Pilas

- Solo se pueden *acceder* por un extremo.
- Tres operaciones básicas: ***apilar, desapilar*** y ***cumbre***.
- El último apilado es al único que se puede acceder directamente y es el primero desapilado: *Last In – First Out*
- También se les llama *pushdown stores*



Estructuras lineales

- Operaciones básicas:
  - push
  - pop
  - top
  - empty

Especificación de la clase stack

```
template <class T> class stack {
    public:
    // Constructoras

    // Pre: cert
    // Post: crea una pila vacía
     stack ();

    // Pre: cert
    // Post: crea una pila que es una copia de S
     stack (const stack& S);

    // Destructora
    ~stack();
```

```
// Modificadoras

/* Pre: La pila es [a1,...,an], n ≥ 0 */
/* Post: Se añade el elemento x como primero de la pila,
   es decir, la pila será [x,a1,...,an] */
void push(const T& x);

/* Pre: La pila es [a1,...,an] y no está vacía (n>0) */
/* Post: Se ha eliminado el primer elemento de la pila
   original, es decir, la pila será [a2,...,an] */
void pop ();
```

```
// Consultoras

/* Pre: la pila es [a1,...,an] y no está vacía (n>0) */
/* Post: Retorna a1 */
T top() const;

/* Pre: cert */
/* Post: Retorna true si y solo si la pila está vacía */
bool empty() const;

/* Pre: cert */
/* Post: Retorna el número de elementos de la pila*/
int size() const;

};
```

Suma de los elementos de una pila

```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es [a1,...,an], retorna a1+...+an
int suma (stack <int>& p);
```

Suma de los elementos de una pila

```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es [a1,...,an], retorna a1+...+an
int suma (stack <int>& p);
```

Suma de los elementos de una pila

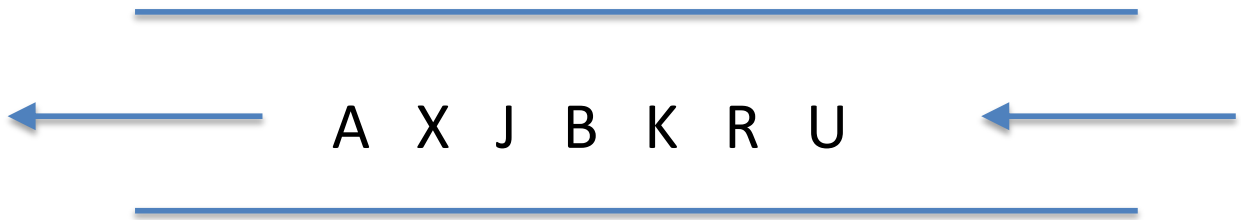
```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es [a1,...,an], retorna a1+...+an
int suma (stack <int>& p) {
   int s = 0;
      while (not p.empty()) {
         s = s + p.top();
         p.pop();
      }
   return s;
}
```

Colas

La clase queue

Tres operaciones básicas:

- Añadir un nuevo elemento (entrar, push)
- Eliminar el primer elemento que ha entrado (salir, pop)
- Ver quién es el primer elemento (primero, front)



Especificación de la clase queue

```
template <class T> class queue {
    public:
    // Constructoras

    // Pre: cert
    // Post: crea una cola vacía
    queue ();

    // Pre: cert
    // Post: crea una cola que es una copia de q
    queue (const queue& q);

    // Destructora
    ~queue();
```

```
// Modificadoras

/* Pre: La cola es [a1,...,an], n ≥ 0 */
/* Post: Se añade el elemento x como último de la cola,
   es decir, la cola será [a1,...,an,x] */
void push(const T& x);

/* Pre: La cola es [a1,...,an] y no está vacía (n>0) */
/* Post: Se ha eliminado el primer elemento de la cola
   original, es decir, la cola será [a2,...,an] */
void pop ();
```

```
// Consultoras

/* Pre: la cola es [a1,...,an] y no está vacía (n>0) */
/* Post: Retorna a1 */
T front() const;

/* Pre: cert */
/* Post: Retorna true si y solo si la cola está vacía */
bool empty() const;

};
```

Listas

Listas

1. Iteradores.
2. Especificación de la clase Lista.
3. Ejemplos de operaciones con listas
4. La operación ***splice***.
5. Fusión de listas ordenadas

Iteradores

Listas

Las listas son estructuras lineales que permiten:

- Recorridos secuenciales de sus elementos.
- Insertar elementos en cualquier punto.
- Eliminar cualquier elemento
- Concatenar una lista a otra.
- Lo que nos permite hacer estas cosas son los ***iteradores***

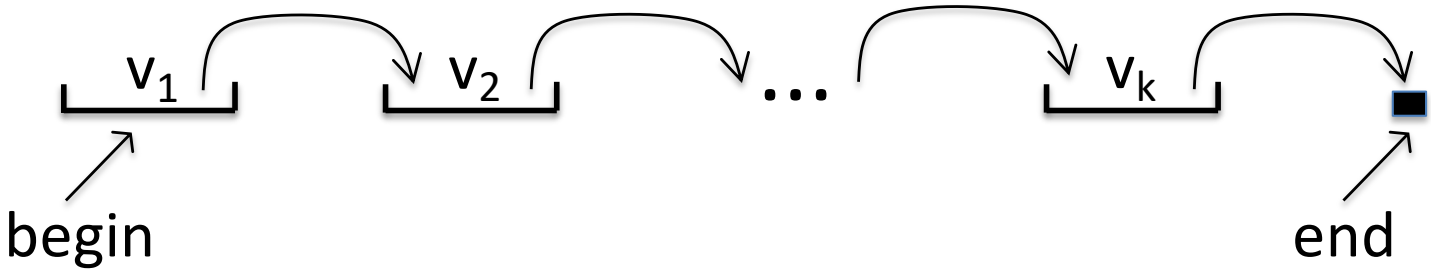
Contenedores e iteradores

- Un *iterador*, es un objeto que designa un elemento de un contenedor para desplazarnos por él.
- Un *contenedor* es una estructura de datos (un template) para almacenar objetos.
- Las listas son casos particulares de contenedores.

Iteradores: declaración (instanciación)

- Método **`begin()`**
- Método **`end()`**
- **`list<Estudiant>::iterator it = l.begin();`**
- **`list<Estudiant>::iterator it2 = l.end();`**
- Si una llista **`l`** está vacía, entonces **`l.begin() = l.end()`**

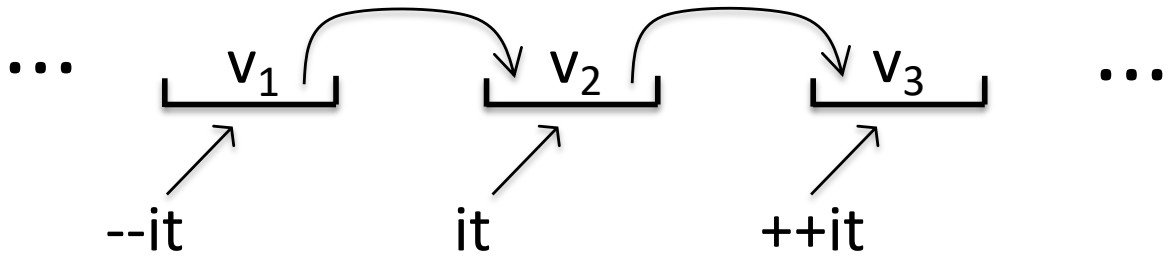
Iteradores



Operaciones con iteradores

- `it1 = it2;`
- `it1 == it2, it1 != it2`
- `*it (si it != l.end() )`
- `++it, --it (salvo si estamos en l.begin o en l.end() )`
- **NO**: `it + 1, it1 + it2, it1 < it2, ...`

Iteradores



Esquema frecuente

```
list<t> l;

...

list<t>::iterator it = l.begin();

while (it != l.end() and not
 cond(*it)) {

   ... acceder a *it ...

++it; }
```

Iteradores constantes

Los iteradores constantes prohiben modificar el objeto referenciado por el iterador. Por ejemplo:

```
list<Estudiant>::const_iterator it1 = l.begin();

list<Estudiant>::iterator it2 = l.end();
```

Estaría prohibido:

```
*it1 = ...;
```

pero no:

```
it2 = it1;

*it2 = ...;

it1 = it2;
```

Imprimir una lista de estudiantes

```
void imprimir_llista(const list<Estudiant>& L) {
for(list<Estudiant>::const_iterator it = L.begin();
     it != L.end(); ++it)
  (*it).escriure();
}
```

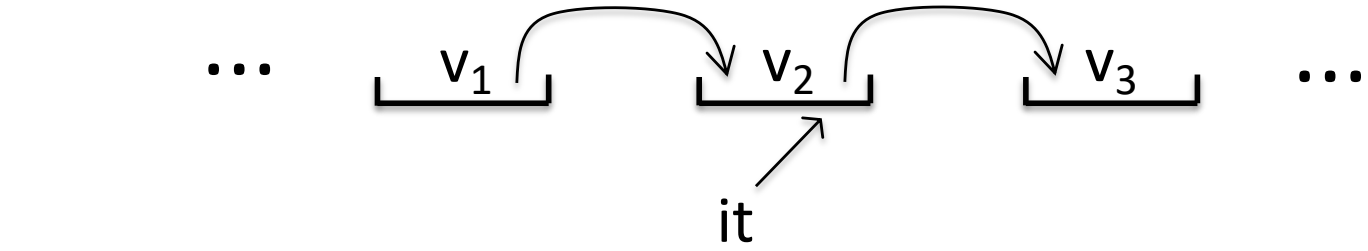
Especificación de la clase Lista

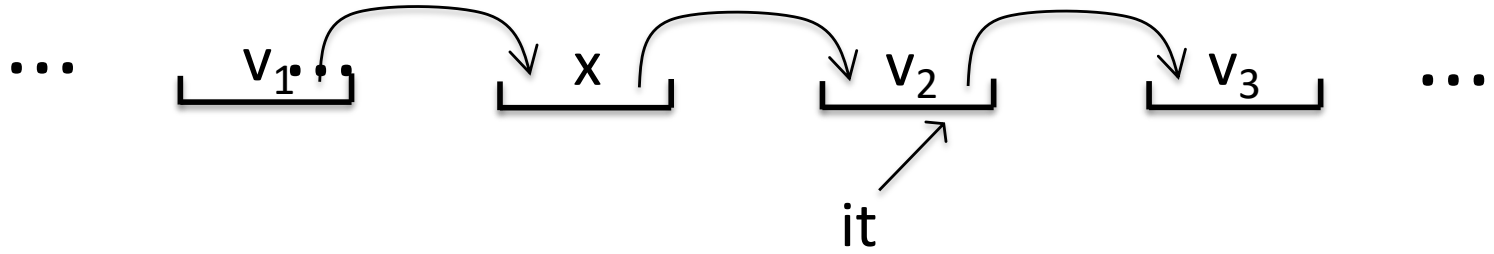
La clase list

```
template <class T> class list {

public:

// Subclases de la clase lista

class iterator { ... };

class const_iterator { ... };

// Constructoras

// Crea la lista vacía

list();

// Crea una lista que es una copia de original

list(const list & original);
```

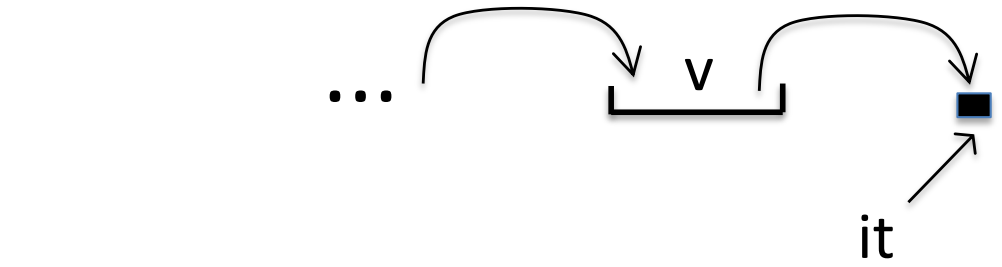
```
// Destructora:

~list();

// Modificadoras

// Pre: true
// Post: El PI pasa a ser la lista vacía

void clear();

// Pre: true
/* Post: Se inserta en el PI un nodo con el valor x delante
de la posición apuntada por it  */

void insert(iterator it, const T& x);
```

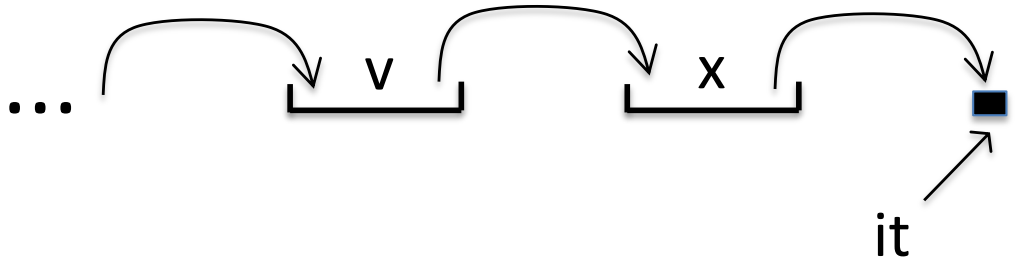
insert(it,x)



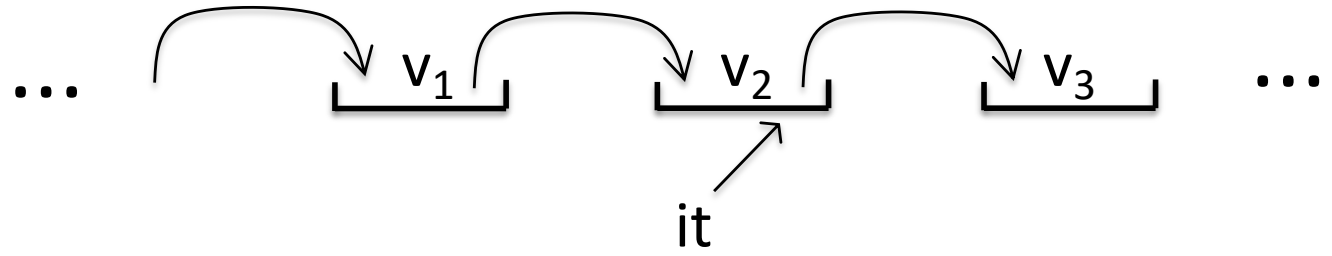


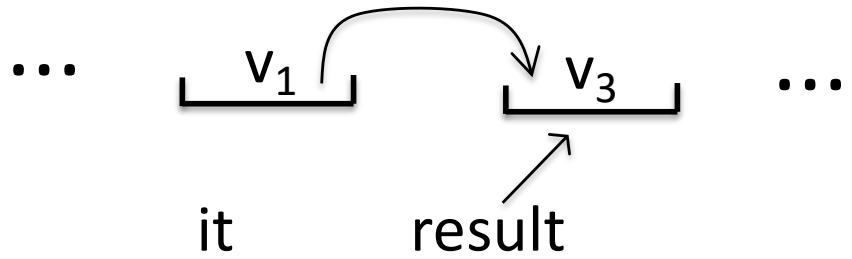
insert(it,x)





```
// Pre: it es distinto de end
/* Post: Se elimina del PI el elemento apuntado por it,
   it queda indefinido y se retorna la posición siguiente a
   la que apuntaba it */

iterator erase(iterator it);
```






```
// Es habitual

it = erase(it);
```

```
// Consultoras

// Pre: true
/* Post: Retorna si el PI es la lista vacia*/

bool empty() const;

// Pre: true
/* Post: Retorna el numero de elementos del PI*/

int size() const;
```

Ejemplos de operaciones con listas

Suma de los elementos de una lista de enteros

```
/* Pre: true */
/* Post: El resultado es la suma de los elementos de l */
int suma(const list<int>& l);
```

Suma de los elementos de una lista de enteros

```
int suma(const list<int>& l) {
    int s = 0;
    for (list<int>::const_iterator it = l.begin();
              it != l.end(); ++it){
// S es la suma de los elementos visitados de la lista
        s = s + *it;
    }
    return s;
}
```

Búsqueda en una lista de enteros

```
/* Pre: true */
/* Post: El resultado indica si x está o no en l */
bool pertenece(const list<int>& l, int x);
```

Búsqueda en una lista de enteros

```
bool pertenece(const list<int>& l, int x) {
    list<int>::const_iterator it = l.begin();
// it apunta a un elemento de l
// x no está en las posiciones anteriores a it
    while ((it != l.end()) and (*it != x))
        ++it;
    return it != l.end();
}
```

Suma k a todos los elementos de una lista

```
/* Pre: l=[x1,...,xn] */
/* Post: l=[x1+k,x2+k,...,xn+k] */
void suma_k(list<int>& l, int k);
```

Suma k a todos los elementos de una lista

```
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
// Se ha sumado k a todos los elementos anteriores a it
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
```

Suma k a todos los elementos de una lista (versión 2)

```
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
// Se ha sumado k a todos los elementos anteriores a it
    while (it != l.end()) {
        int aux = (*it) + k;
        it = l.erase(it);
         l.insert(it,aux);
    }
}
```

Decir si una lista es capicua

```
/* Pre: cierto*/
/* Post: El resultado nos dice si la lista es capicua*/
void capicua(const list<int>& l);
```

Decir si una lista es capicua

```
void capicua(const list<int>& l) {
    list<int>::iterator it1 = l.begin();
    list<int>::iterator it2 = l.end();
/* it1 e it2 apuntan a posiciones simétricas de la lista,
todas las parejas de elementos simétricos anteriores a it1
y posteriores a it2 son iguales */
    for (int i = 0; i < l.size()/2; ++i) {
        --it2;
        it = l.erase(it);
     if (*it1 != *it2) return false;
         ++ it1;
    }
     return true;
}
```

Inserción en una lista ordenada

```
/* Pre: L=[x1,...,xn], está ordenada */
/* Post: L contiene a x, x1,...,xn, y está ordenada */
void inserc_ordenada(list<int>& L, int x) ;
```

Inserción en una lista ordenada

```
/* Pre: L=[x1,...,xn], está ordenada */
/* Post: L contiene a x, x1,...,xn, y está ordenada */
void inserc_ordenada(list<int>& L, int x) {
    list<int>::iterator it = L.begin();
// x es mayor que todos los elementos anteriores a it
    while (it != L.end() and (x > *it)
        ++it;
    L.insert(it,x);
}
```

La operación splice.

Inserción al por mayor

```
// Pre: true
/* Post: Se inserta en el PI toda la lista l delante del
     elemento apuntado por it y l queda vacía */

void splice(iterator it, list& l);
```

Concatenación de listas

```
// Pre: true
/* Post: Se inserta l2 al final de l1 y l2 queda vacía */

void concat(list& l1, list& l2);
```

Concatenación de listas

```
// Pre: true
/* Post: Se inserta l2 al final de l1 y l2 queda vacía */

void concat(list& l1, list& l2){
     list<int>::iterator it = l1.end();
     l1.splice(it,l2);
}
```

Fusión de listas ordenadas

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

```
/* Pre: L1=[x1,...,xn], L2=[y1,...,ym] y las dos listas
    están ordenadas */
/* Post: L1 contiene x1,...,xn,y1,...,ym y está ordenada, L2
no se modifica*/
void inserc_ordenada(list<int>& L1, const list<int>& L2);
```

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

```
void inserc_ordenada(list<int>& L1, const list<int>& L2) {
    list<int>::iterator it1 = L1.begin();
    list<int>::iterator it2 = L2.begin();
/* L1 y L2 están ordenadas. Todos los elementos de L1
anteriores a it1 son menores que el elemento apuntado por
it2. Y todos los elementos de L2 anteriores a it2 son
menores que el elemento apuntado por it1 */
    while (it1 != L1.end() and it2 != L2.end() ) {
        if (*it1 < *it2) ++it1;
        else {L1.insert(it1,*it2); ++it2;
    }
    while (it2 != L2.end() ) {
        L1.insert(it1,*it2);
        ++it2;
    }
}
```