



UNIVERSIDAD TÉCNICA  
FEDERICO SANTA MARÍA



# Clase Lista C++ Estándar

---

Agustín J. González  
EIO329



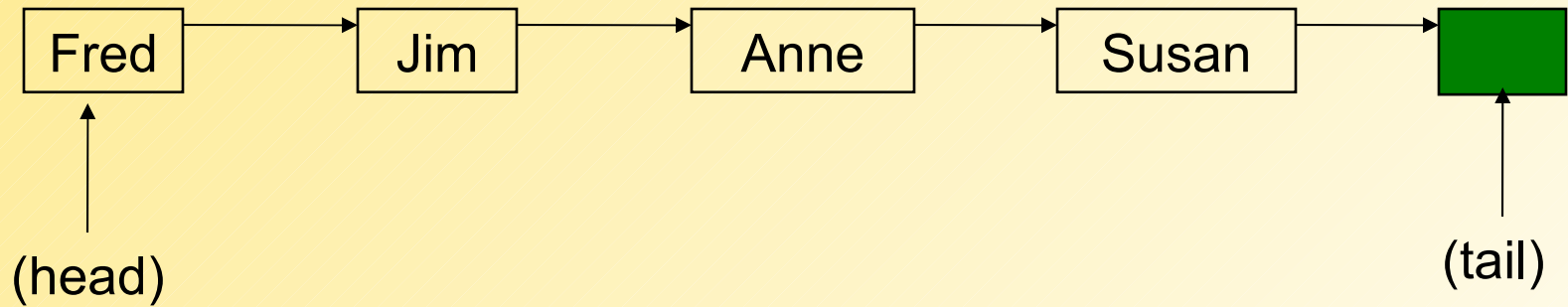
# Estructura de Datos Lista (List)

---

- La estructura de datos Lista es una secuencia conectada de nodes, cada uno de los cuales contiene algún dato.
- Hay un nodo al comienzo llamado la cabeza o frente (head o front).
- Hay un nodo de término llamado la cola o atrás (tail o back).
- Una Lista sólo puede ser recorrida en secuencia, usualmente hacia atrás o adelante.
- Hay varias formas de implementar una lista, como se muestra a continuación...

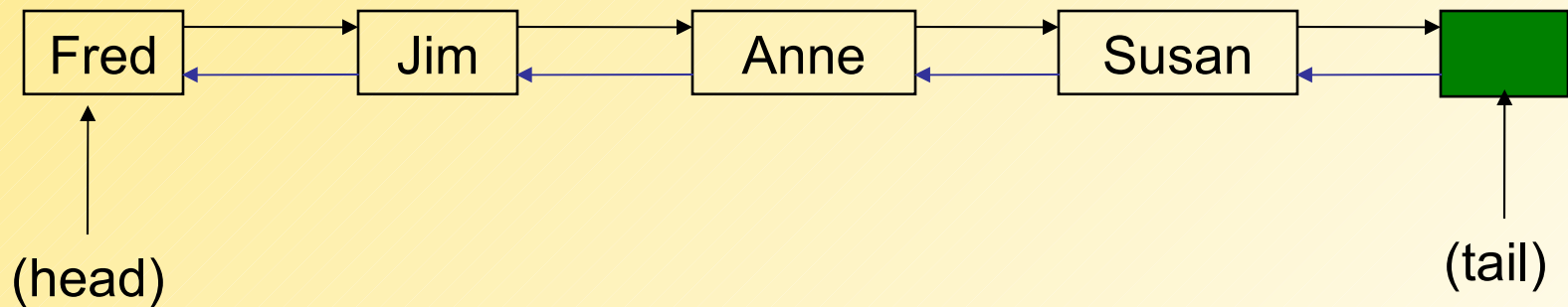
# Lista Simplemente Enlazada

- Una lista simplemente enlazada tiene punteros conectando los nodos sólo en dirección hacia la cola. Cada uno de los nodos contiene un string en este ejemplo:

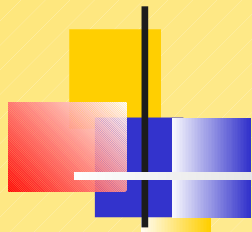


# Lista Doblemente Enlazada

- Una lista doblemente enlazada tiene punteros conectando los nodos en ambas direcciones. Esto permite recorrer la lista en ambas direcciones:



Las clase List en la biblioteca estándar usa esta implementación.



# Métodos de clase List

**Table 9.1 Summary of list operations**

Constructors and Assignment		
<code>list&lt;T&gt; v;</code>	Default constructor	$O(1)$
<code>list&lt;T&gt; v (aList);</code>	Copy constructor	$O(n)$
<code>l = aList</code>	Assignment	$O(n)$
<code>l.swap (aList)</code>	Swap values with another list	$O(1)$
Element Access		
<code>l.front ()</code>	First element in list	$O(1)$
<code>l.back ()</code>	Last element in list	$O(1)$
Insertion and Removal		
<code>l.push_front (value)</code>	Add value to front of list	$O(1)$
<code>l.push_back (value)</code>	Add value to end of list	$O(1)$
<code>l.insert (iterator, value)</code>	Insert value at specified location	$O(1)$
<code>l.pop_front ()</code>	Remove value from front of list	$O(1)$
<code>l.pop_back ()</code>	Remove value from end of list	$O(1)$
<code>l.erase (iterator)</code>	Remove referenced element	$O(1)$
<code>l.erase (iterator,iterator)</code>	Remove range of elements	$O(1)^a$
<code>l.remove (value)</code>	Remove all occurrences of value	$O(n)$
<code>l.remove_if (predicate)</code>	Removal all values that match condition	$O(n)$
Size		
<code>l.empty ()</code>	True if collection is empty	$O(1)$
<code>l.size ()</code>	Return number of elements in collection	$O(n)^b$
Iterators		
<code>list&lt;T&gt;::iterator itr</code>	Declare a new iterator	$O(1)$
<code>l.begin ()</code>	Starting iterator	$O(1)$
<code>l.end ()</code>	Ending iterator	$O(1)$
<code>l.rbegin ()</code>	Starting backwards moving iterator	$O(1)$
<code>l.rend ()</code>	Ending backwards moving iterator	$O(1)$
Miscellaneous		
<code>l.reverse ()</code>	Reverse order of elements	$O(n)$
<code>l.sort ()</code>	Place elements into ascending order	$O(n \log n)$
<code>l.sort (comparison)</code>	Order using comparison function	$O(n \log n)$
<code>l.merge (list)</code>	Merge with another ordered list	$O(n)$

a. Freeing the memory used by erased cells will require time proportional to the number of elements deleted.

b. Some implementations keep track of the number of elements in a list, and thus can determine the size in  $O(1)$ .





# Clase List Estándar C++

---

- La clase list es una clase template (plantilla) en la Biblioteca estándar C++
- Podemos crear listas que contengan cualquier tipo de objeto.
- Las clases list y vector comparten muchas operaciones, incluyendo: `push_back()`, `pop_back()`, `begin()`, `end()`, `size()`, y `empty()`
- El operador sub-índice ( `[ ]` ) no puede ser usado con listas.

Ver <http://www.cplusplus.com/reference/stl/>.



# Agregar y remover nodos

---

El siguiente código crea una lista, agrega cuatro nodos, y remueve un nodo:

```
#include <list>
```

```
list <string> staff;
```

```
staff.push_back("Fred");
```

```
staff.push_back("Jim");
```

```
staff.push_back("Anne");
```

```
staff.push_back("Susan");
```

```
cout << staff.size() << endl; // 4
```

```
staff.pop_back();
```

```
cout << staff.size() << endl; // 3
```



# Iteradores

- Un **iterador (iterator)** es un puntero que se puede mover a través de la lista y provee acceso a elementos individuales.
- El **operador desreferencia (\*)** es usado cuando necesitamos obtener o fijar el valor de un elemento de la lista.

```
list<string>::iterator pos;
```

```
pos = staff.begin();
```

```
cout << *pos << endl;           // "Fred"
```

```
*pos = "Barry";
```

```
cout << *pos << endl;           // "Barry"
```





# Iteradores

---

Podemos usar los operadores ++ y -- para manipular iteradores. El siguiente código recorre la lista y despliega los ítems usando un iterador:

```
void ShowList( list<string> & sList )
{
    list<string>::iterator pos;
    pos = sList.begin();

    while( pos != sList.end())
    {
        cout << *pos << endl;
        pos++;
    }
}
```



# Iterador Constante (const\_iterator)

Si pasamos una lista como constante (const list) debemos usar un **iterador constante** para recorrer la lista:

```
void ShowList( const list<string> & sList )
{
    list<string>::const_iterator pos;
    pos = sList.begin();

    while( pos != sList.end())
    {
        cout << *pos << endl;
        pos++;
    }
}
```



# Iterador reverso (reverse\_iterator)

Un iterador reverso (reverse\_iterator) recorre la lista en dirección inversa. El siguiente lazo despliega todos los elementos en orden inverso:

```
void ShowReverse( list<string> & sList )
{
    list<string>::reverse_iterator pos;
    pos = sList.rbegin();

    while( pos != sList.rend())
    {
        cout << *pos << endl;
        pos++;
    }
}
```



# Iterador constante Reverso (const\_reverse\_iterator)

---

Un `const_reverse_iterator` nos permite trabajar con objetos lista constantes:

```
void ShowReverse( const list<string> & sList )
{
    list<string>::const_reverse_iterator pos;
    pos = sList.rbegin();

    while( pos != sList.rend())
    {
        cout << *pos << endl;
        pos++;
    }
}
```



# Inserción de Nodos

La función miembro `insert()` inserta un nuevo nodo antes de la posición del iterador. El iterador sigue siendo válido después de la operación.

```
list<string> staff;  
staff.push_back("Barry");  
staff.push_back("Charles");
```

```
list<string>::iterator pos;  
pos = staff.begin();  
staff.insert(pos, "Adele");  
// "Adele", "Barry", "Charles"
```

```
pos = staff.end();  
staff.insert(pos, "Zeke");  
// "Adele", "Barry", "Charles", "Zeke"
```



# Eliminación de Nodos

---

La función miembro `erase()` remueve el nodo de la posición del iterador. El iterador es **no válido** después de la operación.

```
list<string> staff;
staff.push_back("Barry");
staff.push_back("Charles");

list<string>::iterator pos = staff.begin();
staff.erase(pos);
cout << *pos;          // error:invalidated!

// erase all elements
staff.erase( staff.begin(), staff.end());

cout << staff.empty(); // true
```



# Mezcla de Listas

---

La función miembro `merge()` combina dos listas en según el operador de orden de los objetos que contiene. Por ejemplo en este caso el orden es alfabético.

```
list <string> staff1;  
staff1.push_back("Anne");  
staff1.push_back("Fred");  
staff1.push_back("Jim");  
staff1.push_back("Susan");
```

```
list <string> staff2;  
staff2.push_back("Barry");  
staff2.push_back("Charles");  
staff2.push_back("George");  
staff2.push_back("Ted");
```

```
staff2.merge( staff1 );
```



# Ordenamiento de una Lista

---

La función miembro `sort()` ordena la lista en orden ascendente. La función `reverse()` invierte la lista.

```
list <string> staff;
```

```
·
```

```
·
```

```
staff.sort();
```

```
staff.reverse();
```



# Nota sobre sobrecarga del operador de salida (<<)



- Siempre que sea posible, las funciones operadores (+, -, etc) deberían ser encapsuladas como funciones miembros de la clase.
- Sin embargo, hay ocasiones en que esto genera una expresión difícil de interpretar. En este caso hacemos una excepción a la regla.
- Si pudiéramos:

```
class Point {  
public:  
    ostream & operator <<(ostream & os);  
    .....  
};
```

- Podríamos tener:     Point p;  
                  p.operator <<(cout); // llamado a función  
                  p << cout // el mismo efecto

# Nota sobre sobrecarga del operador de salida (<<)



Obviamente esta codificación no es intuitiva.

- El usar una función no miembro de la clase nos permite disponer los operandos en el orden “normal”.
- La función debe ser implementada como:

```
ostream & operator << (ostream & os,  
                      const Point & p)  
{  
    os << '(' << p.GetX() << ',' << p.GetY() << ')';  
    return os;  
};
```

- Si necesitamos acceder a miembros privados de la clase, la declaramos dentro de la clase como función amiga.



# Aplicación: Catálogo en Línea

---

- Crear una clase para ítems que describa ítems de un catálogo de venta en línea. Un ítem contiene un número de identificación ID, descripción, y precio.
- Crear una clase Catálogo para mantener los ítems. El catálogo debe encapsular un objeto C++ Lista y proveer operaciones para agregar, buscar, y remover ítems de catálogo.
- (cubre: transparencias 19 - 32)
- Ver ejemplo en: <http://profesores.elo.utfsm.cl/~agv/elo329/Ejemplos/Catalogo/>



# Clase Ítem

```
class Item {
public:
    Item( const string & catalogID,
          const string & description = "",
          double price = 0);

    bool operator ==(const Item & I2) const;

    friend ostream & operator <<(ostream & os,
                                  const Item & I);

private:
    string m_sCatalogID;           // 5 digit catalog ID
    string m_sDescription;
    double m_nPrice;
};
```



# Clase Catálogo

---

```
class Catalog {
public:
    void Add( const Item & I );
    // Add a new item to the catalog

    list<Item>::iterator Find( const Item & anItem );
    // Find an item, return an iterator that
    // either points to the item or contains NULL
    // if the item was not found

    void Remove( list<Item>::iterator I );
    // Remove the item pointed to by I

    friend ostream & operator <<( ostream & os,
        const Catalog & C );

private:
    list<Item> m_vItems;
};
```



# Programa Cliente - 1

---

```
//Create a catalog and add some items.  
Catalog catCurrent;  
catCurrent.Add( Item("00001",  
    "Chinese TaiChi Sword",75.00));  
catCurrent.Add( Item("00002",  
    "Fantasy Dragon Sword",125.00));  
catCurrent.Add( Item("00003",  
    "Japanese Taichi Sword",85.00));  
catCurrent.Add( Item("00004",  
    "Ornate Samurai Sword",150.00));  
catCurrent.Add( Item("00005",  
    "Bamboo Practice Sword",35.00));
```



# Programa Cliente - 2

---

```
    Catalog catBackup( catCurrent );  
    // Notice how C++ creates an automatic  
    // copy constructor  
  
    catBackup = catCurrent;  
    // C++ also creates an automatic assignment  
    // operator
```



# Programa Cliente - 3

---

```
// Search for an item
list<Item>::iterator iter = NULL;

iter = catCurrent.Find( Item("00003") );
if( iter != NULL )
{
    cout << "Item found:\n" << *iter << endl;
}
```





# Programa Cliente - 4

---

```
// Remove the item we just found  
catCurrent.Remove( iter );
```

```
// Don't try to use the same iterator!  
// (runtime error)  
catCurrent.Remove( iter );
```



# Programa Cliente - 5

---

```
// Display the entire catalog  
cout << "\n--- Catalog Contents ---\n"  
      << catCurrent;
```

```
// Save it in a file  
ofstream outfile("catalog.txt");  
outfile << catCurrent;
```



# Implementación de la Clase Ítem - 1

---

```
Item::Item(const string &catalogID,  
           const string &description,  
           double price)  
// Constructor with parameters  
{  
    m_sCatalogID = catalogID;  
    m_sDescription = description;  
    m_nPrice = price;  
}
```



# Implementación de la Clase Ítem - 2

---

```
bool Item::operator ==(const Item & I2) const
// overload the equality operator
{
    return m_sCatalogID == I2.m_sCatalogID;
}
```

```
ostream & operator <<(ostream & os, const Item & I)
// Stream output operator
{
    os << I.m_sCatalogID << ", "
        << I.m_sDescription << ", "
        << I.m_nPrice;
    return os;
}
```



# Implementación de la Clase Catalogo - 1

---

```
void Catalogo::Add(const Item & I)
// Add a new item to the catalog
{
    m_vItems.push_back( I );
}
```



## Implementación de la Clase Catalogo - 2

---

```
list<Item>::iterator Catalog::Find(
    const Item & anItem )
// Find an item, return an iterator that
// either points to the item or contains NULL
// if the item was not found
{
    list<Item>::iterator I;
    for(I = m_vItems.begin();
        I != m_vItems.end(); I++)
    {
        if( *I == anItem ) // Item overloads == oper
            return I;      // found a match
    }
    return NULL;          // failed to find a match
}
```



## Implementación de la Clase Catalogo - 3

---

```
void Catalog::Remove( list<Item>::iterator I )  
// Remove a single list node. This will cause a  
// runtime error if the iterator is invalid.
```

```
{  
    if( I != NULL )  
    {  
        m_vItems.erase( I );  
    }  
}
```

Un buen manejo de excepciones previene este posible error de ejecución.



# Implementación de la Clase Catalogo - 4

---

```
ostream & operator<<( ostream & os, const Catalog & C )
// Stream output operator
{
    list<Item>::const_iterator I;
    for( I = C.m_vItems.begin(); I != C.m_vItems.end(); I++)
    {
        os << *I << endl;
    }
    return os;
}
```