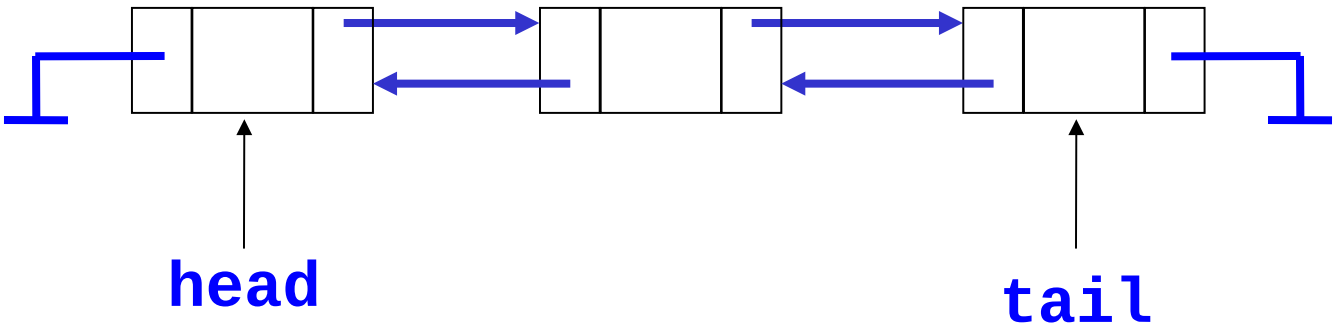
# Implementation of `List`

- Different from STL **`list`**
- Implements a doubly linked list.
- Uses sentinel nodes: (header and tail nodes).



# Implementation of List

**Four classes:**

1) `List`

2) `Node:` private nested class (data, pointers to previous and next nodes)

3) `const_iterator:` public nested class that abstracts the notion of position

4) `iterator:` same functionality as `const_iterator` except the `operator*` returns a reference to the item instead of a constant reference

# Implementation of List

```
template <typename Object>
class List
{
     private:
          struct Node
          {
           Object data;
           Node *prev;
           Node *next;

           Node( const Object & d = Object( ),
           Node * p = NULL, Node * n = NULL )
           : data( d ), prev( p ), next( n ) { }
          };
```

# Implementation of List

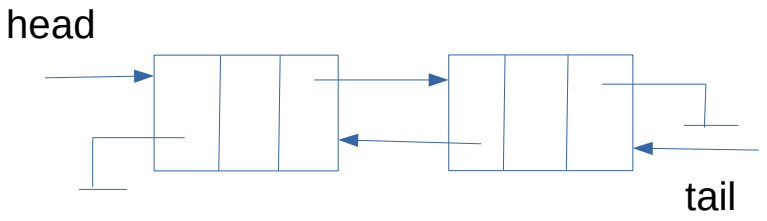
```
public:

class const_iterator
{
        public:
                const_iterator( ) : current( NULL ) { }

                const Object & operator* ( ) const
                  { return retrieve( ); }

                const_iterator & operator++ ( )  //empty parameter for prefix form (++itr)
                  {
                  current = current->next;
                  return *this;
                  }

                const_iterator operator++ ( int ) //single int parameter for postfix form (itr++)
                  {
                  const_iterator old = *this;
                  ++( *this );
                  return old;
                  }

                bool operator== ( const const_iterator & rhs ) const
                  { return current == rhs.current; }

                bool operator!= ( const const_iterator & rhs ) const
                  { return !( *this == rhs ); }

        protected:
                Node *current;

                Object & retrieve( ) const
                  { return current->data; }

                const_iterator( Node *p ) : current( p ) { }

                friend class List<Object>;
};
```

# Implementation of List

```
class iterator : public const_iterator //inherits from const_iterator
{
      public:
            iterator( ) { }

            Object & operator* ( )
             { return const_iterator::retrieve( ); }
            const Object & operator* ( ) const
             { return const_iterator::operator*( ); }

            iterator & operator++ ( )
            {
             this->current = this->current->next;
             return *this;
            }

            iterator operator++ ( int )
             { iterator old = *this; ++( *this ); return old; }

      protected:
            iterator( Node *p ) : const_iterator( p ) { }

            friend class List<Object>;
};
```

# Implementation of List

```
private:
        int theSize;
        Node *head;
        Node *tail;

void init( )
{
          theSize = 0;
          head = new Node;
          tail = new Node;
          head->next = tail;
          tail->prev = head;
}
```



# Implementation of List

```
public:

      List( )
            { init( ); }
      ~List( )
      {
            clear( );
            delete head;
            delete tail;
      }

      List( const List & rhs )
      {
            init( );
            *this = rhs;
      }

      const List & operator= ( const List & rhs )
      {
            if( this == &rhs )
             return *this;
            clear( );
            for( const_iterator itr = rhs.begin( ); itr != rhs.end( ); ++itr )
             push_back( *itr );
            return *this;
      }
```

# Implementation of List

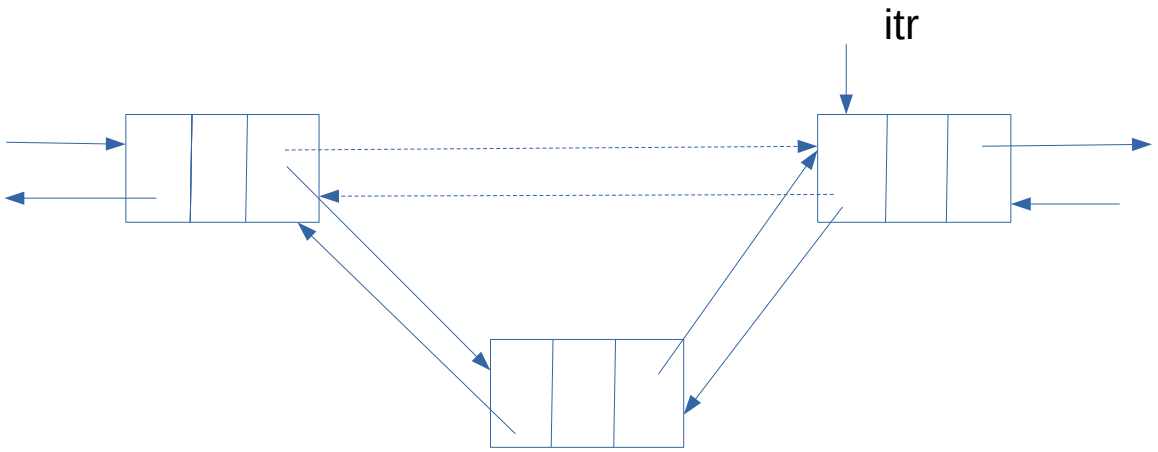
```
iterator begin( )
    { return iterator( head->next ); }
const_iterator begin( ) const
    { return const_iterator( head->next ); }
iterator end( )
    { return iterator( tail ); }
const_iterator end( ) const
    { return const_iterator( tail ); }

int size( ) const
    { return theSize; }
bool empty( ) const
    { return size( ) == 0; }

void clear( )
    {
      while( !empty( ) )
             pop_front( );
    }
```

# Implementation of List

```
Object & front( )
    { return *begin( ); }
const Object & front( ) const
    { return *begin( ); }
Object & back( )
    { return *--end( ); }
const Object & back( ) const
    { return *--end( ); }
void push_front( const Object & x )
    { insert( begin( ), x ); }
void push_back( const Object & x )
    { insert( end( ), x ); }
void pop_front( )
    { erase( begin( ) ); }
void pop_back( )
    { erase( --end( ) ); }
```

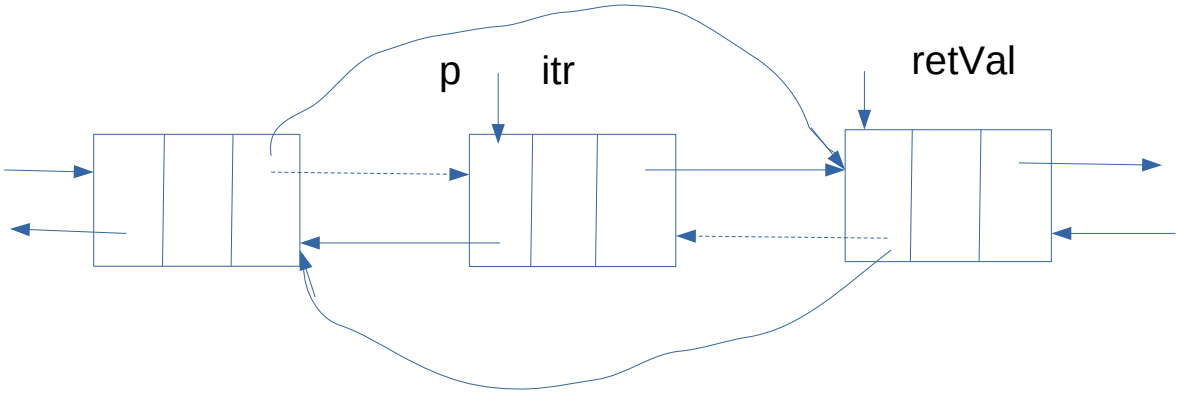
# Implementation of List

```
// Insert x before itr.
iterator insert( iterator itr, const Object & x )
     {
       Node *p = itr.current;
       theSize++;
       return iterator( p->prev = p->prev->next
              = new Node( x, p->prev, p ) );
     }
```



# Implementation of List

```
// Erase item at itr.
iterator erase( iterator itr )
{
      Node *p = itr.current;
      iterator retVal( p->next );
      p->prev->next = p->next;
      p->next->prev = p->prev;
      delete p;
      theSize--;

   return retVal;
}
```



# Implementation of List

```
// Erase item at itr.
iterator erase( iterator itr )
{
     Node *p = itr.current;
  iterator retVal( p->next );
     p->prev->next = p->next;
     p->next->prev = p->prev;
     delete p;
     theSize--;

    return retVal;
}

iterator erase( iterator start, iterator end )
{
     for( iterator itr = start; itr != end; )
       itr = erase( itr );
     return end;
}
```