```
template<class T>
class Vector {

 public:

  class iterador;
  class const_iterador;

  Vector();
  Vector(int n);
  Vector(int n, T val);
  Vector(const Vector<T> & v2);

  Vector<T> & operator=(const Vector<T> & x);

  void reserve(int n);
  void resize( int n);

  int size() const;
  int capacity() const;
  bool empty() const;

  T & operator[](int n);
  const T & operator[](int n) const;
  T & at(int n);
  const T & at(int n) const;

  iterador insert(iterador it, const T & val);
  iterador erase(iterador it);

  iterador begin() { return v; }
  const_iterador begin() const { return v; }
  iterador end() { return v+tama; }
  const_iterador end() const { return v+tama; }
```

```
class iterador {
    friend class Vector;

    public:

      iterador();
      iterador(T * v);
      bool operator==( iterador x);
      bool operator!=( iterador x);
      iterador<T> &  operator++();
      iterador<T> operator++(int);
      iterador<T> &  operator--();
      T & operator*();

    private:
       T * inicio; // apunta al principio del vector
       T * n;
   };  //class iterador

class const_iterador {
    friend class Vector;

    public:
      const_iterador();
      const_iterador(T * v);
      bool operator==( const_iterador x);
      bool operator!=( const_iterador x);
      const_iterador<T> &  operator++();
      const_iterador<T> &  operator--();
      const T & operator*();

    private:
       T * inicio; // apunta al principio del vector
       T * n;
 }; //class const_iterador
```

```
 private:
  int capa; // Numero de elementos que podria almacenar el vector
  int tama; // Numero de elementos que estan alojados actualmente
  T *v; // Memoria ocupada por el vector

}; //End de clase Vector
====================================
void escribeVector(const Vector<int> & b)
{
  Vector<int>::const_iterador i;

  i=b.begin();
  while (i!=b.end())
    {
      cout << *i << ' ';
     i++;
    }
  cout << endl;
}
Vector<int> Concatenar(const Vector<int> & a, const Vector<int> & b)
{
  Vector<int> c;
  int tama, i, j;

  tama = a.size()+b.size();
  c.resize(tama);
  for (i=0;i<a.size();i++)
    c[i] = a[i];  // NO incremente el tamao del vector (no chequea)
  for (j=0;j<b.size()-4;j++,i++)
    c.at(i) = b.at(j);
  return c;
}
```

```
/*
 ************************************************************
 * Implementacin
 ************************************************************

  Funcin de Abstraccin:
  ---------------------
  Dado el objeto del tipo rep r, r = {v, tama, capa}, el objeto
  abstracto al que representa es:
    l = < v[0], v[1], ..., v[tama] >

  Invariante de Representacin:
  ----------------------------
   -  0 <= r.tama <= r.capa


 */
/********** IMPLEMENTACION DE VECTOR ****************/

template <class T> Vector<T>::Vector()
 {
      tama = 0;
      capa = 100; // por defecto reserve espacio de memoria para almacenar 10
                  // objetos de tipo T
      v  = new T[100];
      assert (v!=0);
 }

template <class T> Vector<T>::Vector(int n)
{
      tama = 0;
      capa = n;
      v  = new T[n];
      assert (v!=0);
}
```

```
template <class T> Vector<T>::Vector(int n, T val)
{
      tama = n;
      capa = n;
      v  = new T[n];
      assert (v!=0);
      for (int i=0; i<n;i++) v[i]=val;

 }
template <class T> Vector<T>::Vector(const Vector<T> & v2)
{
  int i;

  tama = v2.tama;
  capa = v2.capa;
  v = new T[v2.tama];

  for (i=0;i<tama ;i++)
    v[i]=v2.v[i];

}
template <class T> void Vector<T>::reserve(int n)
 {
    if (capacity() < n) {
      delete [] v ;
      v = new T[n];
      assert (v!=0);
      tama = 0;
      capa = n;
    }
  }
```

```
template <class T> void Vector<T>::resize( int n)
{
      v = (T *) realloc(v, (sizeof(T) * n));
      assert (v!=0);
      capa = n;
      if (n<tama) tama = n;
 }

template <class T> int Vector<T>::size() const
 { return tama; }


template <class T> int Vector<T>::capacity() const
 { return capa; }
```

```
template <class T> bool Vector<T>::empty() const
 { return begin() == end(); }


template <class T> T & Vector<T>::operator[](int n)
  {
    cout << "En operador []" << endl;
 return (v[n]); }


template <class T> const T & Vector<T>::operator[](int n) const
{ cout << "En operador const []" << endl;
  return *(v + n); }



template <class T> T & Vector<T>::at(int n)
 {
  if ( (n<0  ) || (n >= tama )) {
    cout << "Error en la direccion del vector" << endl;
    return v[0]; //elemento por defecto
   }
  else return (v[n]);
  }

template <class T> const T & Vector<T>::at(int n) const
 { if ( (n<0  ) || (n >= tama )) {
    cout << "Error en la direccion del vector" << endl;
    return v[0]; //elemento por defecto
   }
  else return *(v + n);
  }
```

```
template <class T> Vector<T> &  Vector<T>::operator=(const Vector<T> & x)
{
   int i;
  tama = x.tama;
  capa = x.capa;
  delete []v;
  v = new T[x.tama];

  for (i=0;i<tama ;i++)
    v[i]=x.v[i];

  return *this;
}


template <class T>
Vector<T>::iterador Vector<T>::insert(Vector<T>::iterador it, const T& val)
{
  iterador i_aux;
  int i;


  if ( (it.n >= v+tama  ) || (it.n < v )) {
    cout << "Error en la dimension del vector" << endl;
    return i_aux; // el iterador nulo
  }
  if (capa==tama)
     {
       T *aux;
       aux = new T[2*tama]; // Duplica el tamao
```

```
       for (i=0;i<tama;i++)
           aux[i] = v[i];
       delete [] v;
       v = aux;
       capa *=2;
     }
 for (i_aux=end(), i = tama ; i_aux!=it ; i--, --i_aux)
     v[i] = v[i-1];

   *it = val;
   tama++;

   return it;
 }


template <class T>
Vector<T>::iterador  Vector<T>::erase( Vector<T>::iterador it)
{
   int i;
   iterador i_aux, ss;
  if ( (it.n > v+tama  ) || (it.n < v )) {
    cout << "Error en la dimension del vector" << endl;
    return i_aux; // el iterador nulo
   }
   for ( ss = i_aux = it; i_aux!=end(); i_aux ++){
     ss ++;
     *i_aux = *ss;
   }
   tama--;
   return it;
 }
```

```
template <class T>
Vector<T>::iterador Vector<T>:: begin()
{ return Vector<T>::iterador( *this ); }


template <class T>
Vector<T>::const_iterador Vector<T>:: begin() const
{
return (*this);
 }


template <class T>
Vector<T>::iterador Vector<T>:: end()
{
  Vector<T>::iterador it (*this) ;
  it.n =  v+tama;
  return it;
}

template <class T>
Vector<T>::const_iterador Vector<T>:: end() const
{
  Vector<T>::const_iterador it (*this) ;
  it.n =  v+tama;
  return it;
}


template <class T> Vector<T>::~Vector()
{
  delete [] v;
}
```

```
/*****************************************************/
/********** IMPLEMENTACION DE ITERADOR ***************/
/*****************************************************/

template <class T> Vector<T>::iterador::iterador()
{
  inicio = NULL;
  n = NULL;
}

template <class T> Vector<T>::iterador::iterador(const Vector<T> & x)
{
  inicio = x.v;
  n = x.v;
}

template <class T> Vector<T>::iterador::iterador(const iterador & i)
{ inicio = i.inicio;
  n=i.n;
}

template <class T>
Vector<T>::iterador &  Vector<T>::iterador::operator++()
{
  n++;
  return *this;
}
```

```
template<class T>
Vector<T>::iterador  Vector<T>::iterador::operator++(int)
{
  iterador copia(*this); // Hace una copia del objeto receptor
  operator++();
  return copia;
}

template <class T>
Vector<T>::iterador &  Vector<T>::iterador::operator--()
{
  n--;
  return *this;
}

template <class T>  T & Vector<T>::iterador::operator*()
{
  return  *n;
}

template <class T>
 bool Vector<T>::iterador::operator==(Vector<T>::iterador x)
{
  return ((inicio == x.inicio) && ( n == x.n));
}

template <class T>
bool Vector<T>::iterador::operator!=(Vector<T>::iterador x)
{
  return !(*this==x);
}
```