Práctica 2. Casos prácticos de monitores en C++11.

1. Múltiples Productores-Consumidores con monitores y buffer FIFO.

```
// -----------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// Archivo: prodcons1_mu_FIFO.cpp
//
// Ejemplo de un monitor en C++11 con semántica SU, para el problema
// del productor/consumidor, con productor y consumidor únicos.
//
//
// Se realiza el problema de los múltiples productores consumidores con monitores.
// Se hace con FIFO(primero en entrar es el primero en salir).
// -----------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <random>
#include <thread>
#include "scd.h"

using namespace std ;
using namespace scd ;

constexpr int
num_items = 15 ,   // número de items a producir/consumir.
np        = 5  ,   // número de hebras productoras.
nc        = 5  ;   // número de hebras consumidoras.
int
siguiente_dato = 0 , // siguiente valor a devolver en 'producir_dato'
p = num_items/np,    // número de items por cada hebra productora.
c = num_items/nc,    // número de items por cada hebra consumidora.
items_producidos[np] = {0};     // contador de items producidos por cada hebra

constexpr int
min_ms    = 5,     // tiempo minimo de espera en sleep_for
```

```
max_ms    = 20 ;   // tiempo máximo de espera en sleep_for


mutex
mtx ;                // mutex de escritura en pantalla
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: producidos
cont_cons[num_items] = {0}; // contadores de verificación: consumidos

//----------------------------------------------------------------------
// funciones comunes a las dos soluciones (fifo y lifo)
//----------------------------------------------------------------------

int producir_dato(int ih)
{

assert (ih < np);   // Verifico que el índice de la hebra es menor que el núm. de hebras
productoras.
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
const int inicio = ih*p;
const int valor_producido = items_producidos[ih] + inicio;
items_producidos[ih]++;
mtx.lock();
cout << "hebra productora nº" << ih << " produce " << valor_producido << endl << flush ;
mtx.unlock();
cont_prod[valor_producido]++ ;
return valor_producido ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned valor_consumir, int ih)
{
assert(ih < nc);
if ( num_items <= valor_consumir )
{
cout << " valor a consumir === " << valor_consumir << ", num_items == " << num_items <<
endl ;
assert( valor_consumir < num_items );
}
cont_cons[valor_consumir] ++ ;
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
mtx.lock();
```

```
cout << "                hebra consumidora nº"' << ih << "' consume: " << valor_consumir << endl
;
mtx.unlock();
}
//----------------------------------------------------------------------

void test_contadores()
{
bool ok = true ;
cout << "comprobando contadores ...." << endl ;

for( unsigned i = 0 ; i < num_items ; i++ )
{
if ( cont_prod[i] != 1 )
{
cout << "error: valor " << i << " producido " << cont_prod[i] << " veces." << endl ;
ok = false ;
}
if ( cont_cons[i] != 1 )
{
cout << "error: valor " << i << " consumido " << cont_cons[i] << " veces" << endl ;
ok = false ;
}
}
if (ok)
cout << endl << flush << "solución (aparentemente) correcta." << endl << flush ;
}

// *****************************************************************************
// clase para monitor buffer, version FIFO, semántica SC, multiples prod/cons

class ProdConsMu : public HoareMonitor
{
private:
static const int           // constantes ('static' ya que no dependen de la instancia)
num_celdas_total = 10;   //   núm. de entradas del buffer
int                        // variables permanentes
buffer[num_celdas_total],//   buffer de tamaño fijo, con los datos
primera_libre ,          //   indice de celda de la próxima inserción
primera_ocupada,         //   indice de celda de la próxima extracción.
n;                       //   ( == número de celdas ocupadas)
```

```
CondVar                  // colas condicion:
ocupadas,               //  cola donde espera el consumidor (n>0)
libres ;                //  cola donde espera el productor  (n<num_celdas_total)

public:                  // constructor y métodos públicos
ProdConsMu() ;           // constructor
int  leer();             // extraer un valor (sentencia L) (consumidor)
void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

ProdConsMu::ProdConsMu(  )
{
primera_libre = 0 ;
ocupadas      = newCondVar();
libres        = newCondVar();
}
// -----------------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdConsMu::leer(  )
{
// esperar bloqueado hasta que 0 < primera_libre
if ( n == 0 )
ocupadas.wait();

cout << "leer: ocup == " << n << ", total == " << num_celdas_total << endl ;
assert( 0 < n  );

// hacer la operación de lectura, actualizando estado del monitor
const int valor = buffer[primera_ocupada] ;
n--;
primera_ocupada = (primera_ocupada+1) % num_celdas_total;

// señalar al productor que hay un hueco libre, por si está esperando
libres.signal();

// devolver valor
return valor ;
}
// -----------------------------------------------------------------------------
```

```
void ProdConsMu::escribir( int valor )
{
// esperar bloqueado hasta que primera_libre < num_celdas_total
if ( primera_libre == num_celdas_total )
libres.wait();

cout << "escribir: ocup == " << n << ", total == " << num_celdas_total << endl ;
assert( primera_libre < num_celdas_total );

// hacer la operación de inserción, actualizando estado del monitor
buffer[primera_libre] = valor ;
primera_libre = (primera_libre+1) % num_celdas_total ;
n++;

// señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
ocupadas.signal();
}
// *****************************************************************************
// funciones de hebras

void funcion_hebra_productora( MRef<ProdConsMu> monitor, int ih )
{
for( unsigned i = ih*p; i < ((ih*p)+p) ; i++ )
{
int valor = producir_dato(ih) ;
monitor->escribir( valor );
}
}
// ----------------------------------------------------------------------------

void funcion_hebra_consumidora( MRef<ProdConsMu>  monitor, int ih )
{
for( unsigned i = ih*c; i < ((ih*c)+c) ; i++ )
{
int valor = monitor->leer();
consumir_dato( valor, ih ) ;
}
}
// ----------------------------------------------------------------------------

int main()
{
```

```
cout << "------------------------------------------------------------------" << endl
<< "Problema del productor-consumidor únicos (Monitor SU, buffer FIFO). " << endl
<< "------------------------------------------------------------------" << endl
<< flush ;

// crear monitor  ('monitor' es una referencia al mismo, de tipo MRef<...>)
MRef<ProdConsMu> monitor = Create<ProdConsMu>() ;



// crear y lanzar las hebras
thread hebras_prod[np],
hebras_cons[nc];
for(int i = 0; i < np; i++)
hebras_prod[i] = thread( funcion_hebra_productora, monitor, i);

for(int i = 0; i < nc; i++)
hebras_cons[i] = thread( funcion_hebra_consumidora, monitor, i);

// esperar a que terminen las hebras
for(int i = 0; i < np; i++)
hebras_prod[i].join();

for(int i = 0; i < nc; i++)
hebras_cons[i].join();

test_contadores() ;
}
```

Práctica 2. Casos prácticos de monitores en C++11.

2. Múltiples Productores-Consumidores con monitores y buffer LIFO.

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos
//
//
// Archivo: prodcons1_mu-LIFO.cpp
//
// Ejemplo de un monitor en C++11 con semántica SU, para el problema
// del productor/consumidor, con productor y consumidor únicos.
// Opcion LIFO
//
// Se realiza el problema de los múltiples productores consumidores con monitores.
// Se hace con LIFO(último en entrar es el primero en salir).
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <random>
#include <thread>
#include "scd.h"

using namespace std ;
using namespace scd ;

constexpr int
num_items = 15 ,   // número de items a producir/consumir.
np        = 5  ,   // número de hebras productoras.
nc        = 5  ;   // número de hebras consumidoras.
int
siguiente_dato = 0 , // siguiente valor a devolver en 'producir_dato'
p = num_items/np,    // número de items por cada hebra productora.
c = num_items/nc,    // número de items por cada hebra consumidora.
items_producidos[np] = {0};     // contador de items producidos por cada hebra

constexpr int
min_ms    = 5,     // tiempo minimo de espera en sleep_for
```

```
max_ms    = 20 ;   // tiempo máximo de espera en sleep_for


mutex
mtx ;                // mutex de escritura en pantalla
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: producidos
cont_cons[num_items] = {0}; // contadores de verificación: consumidos

//----------------------------------------------------------------------
// funciones comunes a las dos soluciones (fifo y lifo)
//----------------------------------------------------------------------

int producir_dato(int ih)
{

assert (ih < np);   // Verifico que el índice de la hebra es menor que el núm. de hebras
productoras.
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
const int inicio = ih*p;
const int valor_producido = items_producidos[ih] + inicio;
items_producidos[ih]++;
mtx.lock();
cout << "hebra productora nº" << ih << " produce " << valor_producido << endl << flush ;
mtx.unlock();
cont_prod[valor_producido]++ ;
return valor_producido ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned valor_consumir, int ih)
{
assert(ih < nc);
if ( num_items <= valor_consumir )
{
cout << " valor a consumir === " << valor_consumir << ", num_items == " << num_items <<
endl ;
assert( valor_consumir < num_items );
}
cont_cons[valor_consumir] ++ ;
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
mtx.lock();
```

```
cout << "                hebra consumidora nº"' << ih << "' consume: " << valor_consumir << endl
;
mtx.unlock();
}
//----------------------------------------------------------------------

void test_contadores()
{
bool ok = true ;
cout << "comprobando contadores ...." << endl ;

for( unsigned i = 0 ; i < num_items ; i++ )
{
if ( cont_prod[i] != 1 )
{
cout << "error: valor " << i << " producido " << cont_prod[i] << " veces." << endl ;
ok = false ;
}
if ( cont_cons[i] != 1 )
{
cout << "error: valor " << i << " consumido " << cont_cons[i] << " veces" << endl ;
ok = false ;
}
}
if (ok)
cout << endl << flush << "solución (aparentemente) correcta." << endl << flush ;
}

// *****************************************************************************
// clase para monitor buffer, version FIFO, semántica SC, multiples prod/cons

class ProdConsMu : public HoareMonitor
{
private:
static const int           // constantes ('static' ya que no dependen de la instancia)
num_celdas_total = 10;   //   núm. de entradas del buffer
int                        // variables permanentes
buffer[num_celdas_total],//   buffer de tamaño fijo, con los datos
primera_libre ;          //   indice de celda de la próxima inserción ( == número de celdas
ocupadas)

CondVar                    // colas condicion:
```

```
ocupadas,               //  cola donde espera el consumidor (n>0)
libres ;                //  cola donde espera el productor  (n<num_celdas_total)

public:                 // constructor y métodos públicos
ProdConsMu() ;          // constructor
int  leer();            // extraer un valor (sentencia L) (consumidor)
void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

ProdConsMu::ProdConsMu(  )
{
primera_libre = 0 ;
ocupadas      = newCondVar();
libres        = newCondVar();
}
// -----------------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdConsMu::leer(  )
{
// esperar bloqueado hasta que 0 < primera_libre
if ( primera_libre == 0 )
ocupadas.wait();

cout << "leer: ocup == " << primera_libre << ", total == " << num_celdas_total << endl ;
assert( 0 < primera_libre  );

// hacer la operación de lectura, actualizando estado del monitor
primera_libre-- ;
const int valor = buffer[primera_libre] ;

// señalar al productor que hay un hueco libre, por si está esperando
libres.signal();

// devolver valor
return valor ;
}
// -----------------------------------------------------------------------------

void ProdConsMu::escribir( int valor )
{
```

```
// esperar bloqueado hasta que primera_libre < num_celdas_total
if ( primera_libre == num_celdas_total )
libres.wait();

cout << "escribir: ocup == " << primera_libre << ", total == " << num_celdas_total << endl ;
assert( primera_libre < num_celdas_total );

// hacer la operación de inserción, actualizando estado del monitor
buffer[primera_libre] = valor ;
primera_libre++ ;

// señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
ocupadas.signal();
}
// ****************************************************************************
// funciones de hebras

void funcion_hebra_productora( MRef<ProdConsMu> monitor, int ih )
{
for( unsigned i = ih*p; i < ((ih*p)+p) ; i++ )
{
int valor = producir_dato(ih) ;
monitor->escribir( valor );
}
}
// ----------------------------------------------------------------------------

void funcion_hebra_consumidora( MRef<ProdConsMu>  monitor, int ih )
{
for( unsigned i = ih*c; i < ((ih*c)+c) ; i++ )
{
int valor = monitor->leer();
consumir_dato( valor, ih ) ;
}
}
// ----------------------------------------------------------------------------

int main()
{
cout << "--------------------------------------------------------------------" << endl
<< "Problema del productor-consumidor únicos (Monitor SU, buffer LIFO). " << endl
<< "--------------------------------------------------------------------" << endl
```

```
<< flush ;

// crear monitor  ('monitor' es una referencia al mismo, de tipo MRef<...>)
MRef<ProdConsMu> monitor = Create<ProdConsMu>() ;



// crear y lanzar las hebras
thread hebras_prod[np],
hebras_cons[nc];
for(int i = 0; i < np; i++)
hebras_prod[i] = thread( funcion_hebra_productora, monitor, i);

for(int i = 0; i < nc; i++)
hebras_cons[i] = thread( funcion_hebra_consumidora, monitor, i);

// esperar a que terminen las hebras
for(int i = 0; i < np; i++)
hebras_prod[i].join();

for(int i = 0; i < nc; i++)
hebras_cons[i].join();

test_contadores() ;
}
```

Práctica 2. Casos prácticos de monitores en C++11.

3. Fumadores con monitores SU.

```
// -----------------------------------------------------------------------------
// fumadores-monitores.cpp
// FECHA: 13/11/23
// Se completa el problema de las hebras fumadores y la hebra estanquero.
// -----------------------------------------------------------------------------

#include <iostream>
#include <cassert>
#include <thread>
#include <mutex>
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include "scd.h"

using namespace std ;
using namespace scd ;

// numero de fumadores

const int num_fumadores = 3 ;

//---------------------------------------------------------------------------
// Función que simula la acción de producir un ingrediente, como un retardo
// aleatorio de la hebra (devuelve número de ingrediente producido)

int producir_ingrediente()
{
// calcular milisegundos aleatorios de duración de la acción de fumar)
chrono::milliseconds duracion_produ( aleatorio<10,100>() );

// informa de que comienza a producir
cout << "Estanquero : empieza a producir ingrediente (" << duracion_produ.count() << "
milisegundos)" << endl;

// espera bloqueada un tiempo igual a "duracion_produ' milisegundos
this_thread::sleep_for( duracion_produ );

const int num_ingrediente = aleatorio<0,num_fumadores-1>() ;
```

```
// informa de que ha terminado de producir
cout << "Estanquero : termina de producir ingrediente " << num_ingrediente << endl;

return num_ingrediente ;
}

//----------------------------------------------------------------------
// Función que simula la acción de fumar, como un retardo aleatoria de la hebra

void fumar( int num_fumador )
{

// calcular milisegundos aleatorios de duración de la acción de fumar)
chrono::milliseconds duracion_fumar( aleatorio<20,200>() );

// informa de que comienza a fumar

cout << "Fumador " << num_fumador << "  :"
<< " empieza a fumar (" << duracion_fumar.count() << " milisegundos)" << endl;

// espera bloqueada un tiempo igual a "duracion_fumar' milisegundos
this_thread::sleep_for( duracion_fumar );

// informa de que ha terminado de fumar

cout << "Fumador " << num_fumador << "  : termina de fumar, comienza espera de
ingrediente." << endl;

}

//----------------------------------------------------------------------
// clase para monitor buffer, version FIFO, semántica SU, fumadores.
class Estanco : public HoareMonitor
{
private:
int                    // variables permanentes
ingre_mostrador;       // número de ingrediente que hay en el mostrador
CondVar                 // colas condicion:
mostr_vacio,             //  cola donde espera el estanquero.
ingr_disp[num_fumadores];   //  cola donde esperan los fumadores.
```

```
public:                 // constructor y métodos públicos
Estanco() ;             // constructor
void obtenerIngrediente(int ingre);
void ponerIngrediente(int ingre);
void esperarRecogidaIngrediente();
} ;

//------------------------------------------------------------------------
Estanco::Estanco(){
ingre_mostrador = -1;
for(int i = 0; i < num_fumadores; i++)
ingr_disp[i] = newCondVar();

mostr_vacio = newCondVar();
}

//------------------------------------------------------------------------
void Estanco::obtenerIngrediente(int ingre){
if(ingre_mostrador != ingre)
ingr_disp[ingre].wait();
ingre_mostrador = -1;
cout << "\nFumador '" << ingre << "' retira su ingrediente del mostrador\n" <<endl;
mostr_vacio.signal();
}

//------------------------------------------------------------------------
void Estanco::ponerIngrediente(int ingre){
if(ingre_mostrador != -1) // si el mostrador no está vacio...
mostr_vacio.wait();
ingre_mostrador = ingre;
cout << "\nEstanquero produce ingrediente '" << ingre << "'." << endl;
ingr_disp[ingre].signal();
}

//------------------------------------------------------------------------
void Estanco::esperarRecogidaIngrediente(){
if(ingre_mostrador != -1) // si el mostrador no está vacio...
mostr_vacio.wait();
}

//------------------------------------------------------------------------
// función que ejecuta la hebra del fumador
```

```
void  funcion_hebra_fumador( MRef<Estanco> monitor, int num_fumador )
{
while( true )
{
monitor->obtenerIngrediente(num_fumador);
cout << "\nFumador '" << num_fumador << "' retira su ingrediente del mostrador\n" <<endl;
fumar(num_fumador);
}
}

//----------------------------------------------------------------------
// función que ejecuta la hebra del estanquero

void funcion_hebra_estanquero( MRef<Estanco> monitor )
{
while( true )
{
int ingre = producir_ingrediente();
monitor->ponerIngrediente(ingre);
monitor->esperarRecogidaIngrediente();
}
}

//----------------------------------------------------------------------

int main()
{
cout << "----------------------------------------------------------------------" << endl
<< "        Problema del los fumadores (Monitor SU).                    " << endl
<< "----------------------------------------------------------------------" << endl
<< flush ;

// crear monitor  ('monitor' es una referencia al mismo, de tipo MRef<...>)
MRef<Estanco> monitor = Create<Estanco>() ;

thread hebra_estanquero,
hebras_fumadoras[num_fumadores];

hebra_estanquero = thread(funcion_hebra_estanquero, monitor);

for(int i = 0; i < num_fumadores; i++)
hebras_fumadoras[i] = thread(funcion_hebra_fumador, monitor, i);
```

```
hebra_estanquero.join();
for(int i = 0; i < num_fumadores; i++)
hebras_fumadoras[i].join();
}
```

Práctica 2. Casos prácticos de monitores en C++11.

4. Lectores-Escritores con monitores

```
// -----------------------------------------------------------------------
// Archivo lec-esc.cpp
// Se completa el problema de los lectores-escritores con monitores.
// -----------------------------------------------------------------------

#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include "scd.h"

using namespace std ;
using namespace scd ;

const unsigned
ne = 3,         // número de hebras escritoras.
nl = 2;         // número de hebras lectoras.
int valor = 0;              // para ir mostrando en pantalla el valor

mutex mtx;                  // mutex para la escritura en pantalla.
//----------------------------------------------------------------------
// clase para monitor, semántica SU, lectores_escritores.

class LecEsc : public HoareMonitor{

private:
int n_lec;      // { número de lectores leyendo }
bool escrib;    // { true si hay algún escritor escribiendo }

CondVar         //  colas condicion:
lectura,    //  cola donde esperan los lectores cuando hay un escritor escribiendo.
escritura;  //  cola donde esperan los escritores cuando otros están escribiendo ->
//  -> escrib == true o n_lec > 0 {no hay lec. ni esc. escritura posible}
public:
LecEsc();
```

```
// { invocados por lectores }
void ini_lectura();
void fin_lectura();

//{ invocados por escritores }
void ini_escritura();
void fin_escritura();
};

//---------------------------------------------------------------------
LecEsc::LecEsc(){
n_lec = 0;
escrib = false;
lectura = newCondVar();
escritura = newCondVar();
}

//---------------------------------------------------------------------
void LecEsc::ini_lectura(){
if(escrib)  // si hay escritor.
lectura.wait();
n_lec++;    // registrar un lector más.
cout << "\n Lector comienza a leer, número de lectores: " << n_lec << endl;
lectura.signal();   // desbloqueo en cadena los posibles lectores bloqueados.
}

//---------------------------------------------------------------------
void LecEsc::fin_lectura(){
n_lec--;    // registro un lector menos.
if(n_lec == 0)          // si es el último lector.
escritura.wait();   // desbloqueo un  escritor.
}

//---------------------------------------------------------------------
void LecEsc::ini_escritura(){
if(n_lec > 0 || escrib) // si hay otros, esperar.
escritura.wait();
escrib = true;          // registrar que hay un escritor.
}

//---------------------------------------------------------------------
void LecEsc::fin_escritura(){
```

```
escrib = false;     // registrar que ya no hay escritor
if(!lectura.empty()) // si hay lectores, despertar uno
lectura.signal();
else                 // si no hay, despertar un escritor
escritura.signal();
}

//----------------------------------------------------------------------
//función que ejecutan las hebras escitoras
void funcion_hebra_escritora(MRef<LecEsc> monitor, int ih){
while( true ){
monitor->ini_escritura();   // se inicia la escritura.
valor++;
mtx.lock();
cout << "\nHebra "' << ih << "' escribe "' << valor << "'." << endl;
mtx.unlock();
this_thread::sleep_for((chrono::milliseconds) aleatorio<100,250>() ); // retraso aleatorio
simulando escritura
monitor->fin_escritura();   // se finaliza la escritura
this_thread::sleep_for((chrono::milliseconds) aleatorio<30,100>() ); // retraso aleatorio
}
}

//----------------------------------------------------------------------
//función que ejecutan las hebras lectoras
void funcion_hebra_lectora(MRef<LecEsc> monitor, int ih){
while( true ){
monitor->ini_lectura();   // se inicia la lectura.
mtx.lock();
cout << "\nHebra "' << ih << "' lee "' << valor << "'." << endl;
mtx.unlock();
this_thread::sleep_for((chrono::milliseconds) aleatorio<100,250>() );// retraso aleatorio
simulando lectura
monitor->fin_lectura();   // se finaliza la lectura
this_thread::sleep_for((chrono::milliseconds) aleatorio<30,100>() ); // retraso aleatorio
}
}

//----------------------------------------------------------------------
int main(){
cout << "----------------------------------------------------------------------" << endl
<< "        Problema del los lectores-escritores (Monitor SU).          " << endl
```

```
<< "------------------------------------------------------------------" << endl
<< flush ;

// creo el monitor
MRef<LecEsc> monitor = Create<LecEsc>();

// creo y lanzo las hebras.
thread hebra_escritora[ne], hebra_lectora[nl];

for(int i = 0; i < ne; i++)
hebra_escritora[i] = thread(funcion_hebra_escritora, monitor, i);

for(int i = 0; i < nl; i++)
hebra_lectora[i] = thread(funcion_hebra_lectora, monitor, i);

// espero a que terminen las hebras.
for(int i = 0; i < ne; i++)
hebra_escritora[i].join();

for(int i = 0; i < nl; i++)
hebra_lectora[i].join();
}
```