lkptr.h: Programación por Referencia Java para C++

Adolfo Di Mare
Universidad de Costa Rica, Escuela de Ciencias de la Computación e Informática
San José, Costa Rica
adolfo.dimare@ecci.ucr.ac.cr

Resumen. La recolección automática de memoria dinámica hace la programación Java más fácil que la programación C++. Al usar la biblioteca "`lkptr<X>`" aquí descrita se pueden obtener muchas de las ventajas de Java sin necesidad de incorporar un recolector de basura para C++.

Palabras clave: Cuenta de referencia, memoria dinámica, ingeniería de sistemas, implementación.

1 Programación con punteros inteligentes

La biblioteca STL de C++ incluye la clase "`auto_ptr<X>`" en el archivo de encabezado `<memory>`. La clase "`auto_ptr<X>`" le permite al programador cliente usar punteros inteligentes que destruyen el objeto que referencian cuando su ámbito de existencia termina; estos punteros son "dueños" del objeto que referencian. Por ejemplo, en el siguiente bloque de código el programador no tiene que preocuparse por retornar la memoria dinámica de la que el puntero "`p`" es dueña:

```
{   auto_ptr<BigMatrix> p , q( new BigMatrix( 50*1000 , 100*1000 ) ) ;
    // ...
    q->foo( 15*1000 ); // "q" funciona como un puntero
    p = q; // Ahora "q" es NULL y "p" es el nuevo dueño
    // ...
}   // Aquí el destructor de "p" devuelve la memoria
```

La clase "`auto_ptr<X>`" funciona como si la implementación de la asignación de punteros fuera similar a la siguiente [6]:

```
template <class T> // OJO: "q" no es un argumento "const"
auto_ptr<T>& auto_ptr<T>::operator= ( auto_ptr<T>& q ) {
    if ( this != &q ) {
        delete m_ptr; // "m_ptr" es el campo que referencia el objeto
        m_ptr = q.m_ptr; // nuevo dueño
        q.m_ptr = NULL; // deja a "q" en NULL
    }
    return *this;
}
```

La clase "`auto_ptr<X>`" tiene algunas desventajas bien conocidas entre la que destaca el que estos punteros inteligentes no se deben usar para almacenar valores en los contenedores de la biblioteca STL porque cada puntero "`auto_ptr<X>`" siempre es el único dueño del objeto que referencia. La operación de inserción en cualquier contenedor STL siempre almacena una copia profunda del objeto pues se supone que la copia almacenada puede ser modificada sin cambiar el valor original [7]. En el

CIESC 2008 JUI, pp. 163-172, 2008.

ejemplo de arriba, la asignación "`p = q;`" deja al puntero "`q`" con un valor nulo ("`NULL`") de manera que el objeto referenciado no sea destruido 2 veces (la primera una vez por el destructor de "`p`" y la segunda por el de "`q`").

La forma de remediar las carencias de la clase "`auto_ptr<X>`" es usar cuentas de referencia de manera que varios punteros inteligentes puedan compartir la misma instancia. Existen varias implementaciones de las cuentas de referencia, pero la implementación de la clase "`lkptr<X>`" escogida aquí es la llamada "enlaces de referencia" (*reference linking*), que tiene las siguientes ventajas:

- No usa memoria dinámica adicional.
- No es "intrusiva" porque no requiere de un contador almacenado en el objeto referenciado.
- Es viable usar contenedores STL para almacenar referencias inteligentes "`lkptr<X>`".
- Es muy simple de usar pues la implementación completa está en el archivo de encabezado "`lkptr.h`".
- Es adecuada para aplicaciones en que se usa programación concurrente mediante múltiples hilos o procesos.

2 Implementación de la clase "`lkptr<X>`"

Los punteros inteligentes "`lkptr<X>`" tienen la cualidad de que comparten un mismo objeto. Una forma de lograr que varios punteros queden marcados porque referencian el mismo objeto es incluir en el objeto referenciado un contador que indica cuántos punteros le referencian. La cuenta se incrementa si a un puntero inteligente se le asigna el valor de otro y el destructor del puntero inteligente es el responsable de decrementar la cuenta; en este caso, si la cuenta llegara a ser cero, también el destructor del puntero debe destruir el objeto referenciado porque ya ningún otro puntero inteligente la está referenciando.

```
Lkptr<Chunche> foo() {
     lkptr<Chunche> p1, p2, p3;
     p3.reset( new Chunche( "Costa Rica Pura Vida" ) );
     p1 = p2 = p3; // los 3 comparte la instancia
     {    lkptr<Chunche> p4 (new Chunche( "Viva Italia" ) );
          // ...
     } // aquí "p4" es destruído

     return p2; // p1, p2, p3 son destruidos
}    //         ... el objeto referenciado persiste
```

Figura 1

En la Figura 1 los punteros "p1", "p2" y "p3" comparten el mismo objeto "Costa Rica Pura Vida" mientras que el único dueño de "Viva Italia" es "p4". Si el puntero "p4" dejara de existir, lo que en este caso ocurre cuando el control del programa deja el bloque en que está definido "p4", su valor referenciado también es destruido porque su cuenta de referencia llega a cero. Esto no es lo que ocurre cuando los otros 3 punteros son destruidos porque para retornar el valor de "p2" el compilador debe invocar al constructor de copia de la clase "lkptr<X>" para retornar una copia de "p2", y eso aumenta la cuenta de referencia a 4. A pesar de que el destructor de los 3 punteros decrementa el contador de referencias 3 veces, a fin de cuentas tiene valor 1 == 4-3 y, en consecuencia, el valor compartido no es destruido y sí es correctamente retornado al programa invocador.

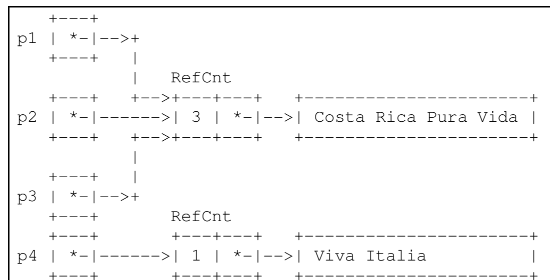

Figura 2

El problema que tiene la implementación de punteros con cuentas de referencia es que hay que incluir en el objeto referenciado un campo para almacenar ahí la cantidad de punteros que referencian el objeto. Otra forma de lograr lo mismo es poner esa cuenta en otro objeto externo como se muestra en la Figura 2. Este esquema no siempre es deseable porque requiere crear un objeto intermedio en memoria dinámica lo que en algunas aplicaciones puede ser inconveniente, como en aplicaciones de sistemas de tiempo real en donde crear objetos en memoria dinámica puede ser muy oneroso.

La otra manera de realizar la implementación es poner en una lista doblemente enlazada a todos los punteros que comparten un valor; se sabe que un puntero es la última referencia si está solo en la lista, como ocurre con "p4" en la Figura 3 (siguiente página). La desventaja de esta solución es que requiere tres veces más almacenamiento por puntero, pues además de la referencia al valor compartido es necesario incluir los 2 punteros necesarios para mantener la lista doblemente enlazada. Sin embargo, en la mayor parte de las aplicaciones este incremento en la cantidad de memoria es poco relevante porque serán relativamente pocos los punteros

que existan simultáneamente en un programa. La implementación "lkptr<X>" que se discute en este trabajo usa la lista enlazada en lugar de las cuentas de referencia porque es la que tiene mayores ventajas y menos inconvenientes. No importa cuál es el primero o el último de la lista, pues lo que importa es que la asignación de un puntero a otro resulta también en el enlace en la lista de referencias.

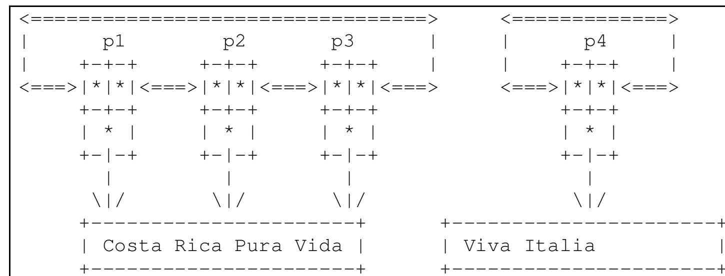

Figura 3

3 Construcción de una clase con punteros inteligentes

Una de las desventajas que tiene C++ cuando se le compara con Java es que es relativamente más caro retornar objetos grandes en C++. Por ejemplo, si se usa la clase "Matrix" descrita en [3], la forma natural de implementar los operadores aritméticos es la siguiente:

```
/// Calcula y retorna \c (A * B).
Matrix operator* ( const Matrix& A, const Matrix& B ) {
    Matrix Resultado...
    // ... algoritmo de multiplicación de matrices
    return Resultado;
}
```

Como la variable "Resultado" es una variable local, el compilador genera código para copiar con el constructor de copia esa variable local al área de retorno en la rutina invocadora, lo que en este caso implica hacer la copia de un objeto bastante grande, pues la cantidad de memoria que usa una matriz es proporcional a la multiplicación de sus dimensiones. Lo natural sería crear la matriz en memoria dinámica, y retornar un puntero a la matriz:

```
/// Calcula y retorna un puntero a \c (A * B).
Matrix* operator* ( const Matrix& A, const Matrix& B ) {
    Matrix * Resultado = new Matrix (...)
    // ...
    return Resultado;
}
```

Esta solución es adecuada pero obliga al invocador a destruir el objeto retornado, lo que puede resultar en fugas de memoria ("*memory leaks*") y, además, es un tanto de incómodo implementar y usar. La alternativa es retornar un puntero inteligente "lkptr<X>" como se muestra en la siguiente implementación del operador de suma:

```
/// Calcula y retorna \c A+B
lkptr<Matrix> operator+ ( const lkptr<Matrix>& A, const lkptr<Matrix>& B ) {
    lkptr<Matrix> Res ( new Matrix(*A) ); Res->Add(*B); return Res;
}
```

En la práctica es incómodo modificar una clase ya existente, pues al hacer cambios es posible introducir errores. Una manera de evitar esta situación es construir una nueva clase derivada de la clase original, de manera que sea en la clase derivada adonde se hagan los cambios. Surge así la clase de plantillas "`RefMatrix`" derivada de la clase "`Matrix`". En la implementación "`RefMatrix.h`" se han agregado únicamente las operaciones aritméticas que se quieren mejorar al hacerlas retornar referencias "`lkptr<X>`" en lugar de copias completas del objeto calculado:

- Los constructores y el destructor.
- Los operadores aritméticos " `{ + - * }` ".
- La operación de "`clone()`" que retorna un puntero a una copia de la instancia.

La implementación de los constructores y el destructor de la clase "`RefMatrix`" es una redefinición directa de los de la clase "`Matrix`". Por ejemplo, el constructor de copia queda implementado en un sólo renglón:

```
RefMatrix(const RefMatrix& o) : Matrix(o) { }
```

Las operaciones aritméticas también son muy simples. Por ejemplo, la implementación de la adición es ésta:

```
template <class E>
class RefMatrix : public Matrix<E> {
    /// Constructor de copia
    RefMatrix(const RefMatrix& o) : Matrix<E>(o) { }
// ...
    /// Retorna un puntero a una copia de la instancia.
    RefMatrix<E>* clone() const { return new RefMatrix<E>(*this); }

    ///< Calcula y retorna \c A+B
    friend lkptr<RefMatrix>
    operator+ (const lkptr<RefMatrix>& A, const lkptr<RefMatrix>& B) }
        lkptr<RefMatrix> Res ( A->clone() ); Res->Add(*B); return Res;
    }
// ...
};
```

El método "`RefMatrix::clone()`" usa el constructor de copia de la clase base "`Matrix`" para obtener una copia profunda. Lo mismo puede lograrse con la siguiente función genérica:

```
/// Retorna un puntero a una nueva instancia que contiene
/// un duplicado del valor del \c "obj".
/// - La copia se hace invocando al constructor de copia.
/// \see http://www.di-mare.com/adolfo/binder/c04.htm#sc06
template <class E>
inline E* clone( const E& obj ) { return new E( obj ); }
```

La receta para lograr retornar grandes objetos usando los punteros inteligentes "`lkptr<X>`" es muy simple: basta crear una nueva clase de empaque con base en la clase original, lo que se puede lograr usando herencia, e implementar de nuevo únicamente aquellas operaciones que requieran retornar objetos grandes.

4 Uso de referencias en programas C++

Las variables "`lkptr<X>`" funcionan como punteros, por lo que al usarlas hay que usar la sintaxis C++ para punteros. En algunos casos es molesto anteponer el asterisco

"`*`" para derreferenciar el puntero, pero en la mayoría de los casos la notación de la flecha "`lk->campo`" es adecuada y elegante, como se muestra en el siguiente bloque C++ que usa 3 matrices cuyo valor es accesado con "`lkptr<unsigned>`":

```
/// Ejemplo de uso de referencia \c lkptr< RefMatrix<unsigned> >.
void use_lkptr( unsigned M, unsigned N ) {
    lkptr< RefMatrix<unsigned> > A ( new RefMatrix<unsigned>(M,N) );
    unsigned k = 0;
    for (unsigned i=0; i < A->rows(); ++i) {
        for (unsigned j=0; j < A->cols(); ++j) {
            A->at(i,j) = k++; // (*A)(i,j) = k++;
        }
    }

    lkptr< RefMatrix<unsigned> > B, C ( A ); // A y C comparten el valor
    assert( B == (void*)0 ); // B es un objeto nulo
    B = C; // A B y C comparten el valor
    B->reSize( B->cols(), B->rows() ); // todos fueron modificados
    assert( B == A );   // comparación de punteros
    assert( *B == *A ); // comparación de matrices
    B.reset( A->clone() ); // B tiene una copia de A
    B->reSize(N,M); // solo B cambia
    C = (C - C);
    assert( A->at(0,0) == 0 ); // porque C borró todo
    C = B + B - B;
    B->reSize( B->cols(), B->rows() ); // solo B cambia
    A = C * B; // ya ninguno comparte memoria
    assert( *A != *B && *B != *C && *C != *A ); // comparación de matrices
}
```

Al examinar este código se pueden sacar las siguientes conclusiones:

- En lugar de usar la notación de acceso al miembro "`A.m()`" es necesario usar "`A->m()`".
- Para obtener una copia del objeto referenciado hay que clonarlo, invocando tanto al operador "`new`" para obtener memoria dinámica como al constructor de copia de la clase base.
- Para obtener una copia del objeto referenciado no se puede usar el operador de asignación, por al hacerlo lo que se lograr en que 2 punteros "`lkptr<X>`" compartan el mismo objeto.

A diferencia de los programadores Java, los programadores C++ están entrenados para saber que los punteros objetos son diferentes a lo que referencian. Por eso, para un programador C++ no es problema distinguir entre el objeto "`X`" y una referencia "`lkptr<X>`". Sin embargo, si se usan los punteros "`lkptr<X>`" para incorporar programadores Java a un proyecto C++ es necesario mostrarles las diferencias pues de lo contrario quedarían limitados en su capacidad para escribir programas correctos. En otras palabras, si alguien puede programar en C++ con "`lkptr<X>`" de seguro podrá programar en Java.

5 El árbol binario implementado con referencias

Los punteros "`lkptr<X>`" tienen un comportamiento que permite construir programas como si se contara con un recolector de basura para el lenguaje. Un árbol binario Java contiene únicamente su puntero inteligente a la raíz "`m_root`", lo que en C++ se implementa usando un puntero inteligente "`lkptr<X>`". El nodo del árbol es la clase opaca "`Bin_Tree_Node<E>`" porque el programador cliente nunca necesita usar

nodos pues las operaciones del árbol manipulan y retornan árboles (una especificación más completa de los árboles se encuentra descrita en [4]):

```
template <class E>
class Bin_Tree {
    lkptr< Bin_Tree_Node<E> > m_root; ///< Raíz del árbol.
// ...
};
```

Al invocar "`left()`" el resultado será obtener otro árbol que contiene la referencia "`lkptr<X>`" que referencia el subárbol izquierdo. Esta es la implementación que consiste en crear una nueva referencia "`lkptr<X>`" al campo "`m_root`":

```
template <class E>
Bin_Tree<E> Bin_Tree<E>::left() const {
    if ( m_root == 0 ) { return Bin_Tree(); }
    else { return Bin_Tree( m_root->m_left ); }
}
```

Si recursivamente se invierte el orden de los hijos de un árbol se le convierte en su espejo, como se muestra en esta implementación, que se asemeja mucho a una implementación Java que haga lo mismo:

```
template <class E>
void mirror( Bin_Tree<E> & T ) {
     if ( T.isEmpty() ) {
        return; // se sale si el árbol está vacío
    }
    // intercambia los hijos
    Bin_Tree<E> Left  = T.left();  // sostiene a cada hijo
    Bin_Tree<E> Right = T.right();
    T.makeLeftChild( Right ); // Pone el hijo derecho a la izquierda
    T.makeRightChild( Left ); // Pone el hijo izquierdo a la derecha
    mirror( Left  );
    mirror( Right ); // recursivamente modifica los hijos
}
```

6 Peligro de usar referencias cíclicas

Las referencias cíclicas o circulares pueden producir problemas si se usan punteros inteligentes, pues la destrucción del objeto referenciado puede producir un llamado recursivo infinito.

Una forma de evitar estas referencias circulares es sustituirlas por referencias a un tercer objeto de intersección [1]. Por ejemplo, si los objetos "`A`" y "`B`" contienen punteros inteligentes que se referencian mutuamente `A↔B`, la idea es introducir un tercer objeto "`C`" de manera que la relación quede así: `A→C←B` (sin el ciclo).

Otra manera de manejar referencias circulares es liberar punteros usando el método `lkptr<>::weak_release()` que elimina el acople entre el puntero inteligente y su objeto apuntado, de manera que se evita la destrucción mutuamente recursiva cuando es necesario usar ciclos de punteros.

7 Almacenamiento en contenedores STL

Algunos métodos de la biblioteca STL hacen copias de los objetos suponiendo que tanto el original como la copia persisten (y que por supuesto son también iguales). Por eso, si se usa la clase "`auto_ptr<X>`" en los contenedores STL pueden presentarse problemas cuando sólo la copia del puntero "`auto_ptr<X>`" mantiene su valor. Por ejemplo, al usar un algoritmo de ordenamiento "`sort()`" en un vector STL que contiene punteros "`auto_ptr<X>`", durante el ordenamiento se hacen varias copias del mismo objeto y, en consecuencia, podría el vector quedar vacío. Este problema no existe con los punteros "`lkptr<X>`" pues tanto el original como todas sus copias mantienen su valor (de hecho lo comparten). Por eso no habrá problemas mezclando las clases y contenedores STL con valores referenciados por "`lkptr<X>`".

8 Control de concurrencia

En un ambiente en que varios procesos o hilos de ejecución pueden modificar el mismo objeto es necesario incorporar algún tipo de control de concurrencia que evite que lo que hace uno afecte al otro. La forma más simple de administrar el acceso a los campos sensitivos de la clase es prender un semáforo antes de cambiar los campos. Para eso, basta incluir las primitivas de sincronización en los métodos "`fast_release()`" y "`acquire()`" que se encargan de enlazar y desenlazar las referencias "`lkptr<X>`". Lo natura es implementar el semáforo como una variable estática de la clase lo que disminuye la contención únicamente entre los punteros "`lkptr<X>`" para valores de la clase "`X`". Lo usual es que hay pocos punteros para una misma clase "`X`", por lo que el gasto en control de concurrencia es muy bajo.

```
#include <MySemaphore.h>

template <class X>
class lkptr {
   // ...
private:
   /// Asegura que sólo 1 proceso puede modificar.
   static Semaphore m_semaphore : Semaphore() { /* ... */ }
   // ...

void fast_release() {
    m_Semaphore.wait(); // bloquea a los demás
    if ( m_prev == this ) {
        if ( m_ptr!=0 ) {
            delete m_ptr;
        }
    }
    else {
        m_prev->m_next = m_next;
        m_next->m_prev = m_prev;
    }
    m_Semaphore.signal(); // otros pueden modificar
}
```

```
void acquire(lkptr* r) throw() {
    m_Semaphore.wait(); // bloquea a los demás
    m_ptr = r->m_ptr;
    m_next = r->m_next;
    m_next->m_prev = this;
    m_prev = r;
    r->m_next = this;
    m_Semaphore.signal(); // otros pueden modificar
}
}; // class lkptr<X>
```

Figura 4

El control de concurrencia que se muestra en la Figura 4 sirve para que no haya problemas al cambiar los campos de la clase "`lkptr<X>`", pero no garantiza el control de concurrencia para el objeto referenciado. Sin embargo, una vez que se han cambiado los campos, el programador cliente puede dereferenciar el puntero inteligente "`lkptr<X>`" sin necesidad de bloquearlo porque el objeto rereferenciado no será destruido mientras exista al menos una referenciar "`lkptr<X>`" hacia él.

En cada plataforma de desarrollo C++ se usan bibliotecas diferentes para el manejo de la concurrencia. Por eso no se ha escogido ninguna en específico para implementar "`lkptr<X>`".

9 Conclusión

El costo de implementación de los punteros inteligentes es "`lkptr<X>`" es de 3 punteros por referencia, y de 5 asignaciones de punteros cuando se comparten valores, más 2 asignaciones cuando el puntero se desliga de los demás. Este costo es bajo en comparación con los beneficios que retribuye su uso:

- Para usar "`lkptr<X>`" basta incluir un único archivo de encabezado "`lkptr.h`" en contraposición a utilizar una biblioteca o ambiente complete que se encargue de la recolección de basura.
- El estilo de programación que "`lkptr<X>`" soporta permite utilizar el código C++ existente sin necesidad de hacerle cambios profundos.
- No se requiere aprender un nuevo paradigma de programación para poder usar objetos grandes con comodidad si se manipulan a través de punteros inteligentes "`lkptr<X>`".
- La existencia de un objeto compartido por referencias "`lkptr<X>`" no va más allá de la última referencia activa hacia el objeto, lo que garantiza una recuperación eficiente de la memoria dinámica que ya no está en uso.
- Debido a que "`lkptr<X>`" no es un recolector de basura, no impone retardos asincrónicos con la aplicación y tampoco introduce un componente aleatorio en la duración de la ejecución.
- Los programas que usan "`lkptr<X>`" son deterministas porque bajo las mismas condiciones su tiempo de ejecución siempre es el mismo.
- Para la programación de sistemas imbuidos o de tiempo real conviene usar "`lkptr<X>`" porque la implementación no usa memoria dinámica para almacenar cada referencia.
- La implementación de "`lkptr<X>`" no levanta excepciones porque no necesita adquirir memoria dinámica. El único caso en que un excepción puede ocurrir es

si el destructor de la clase referenciada levanta una excepción, en cuyo caso el manejo usual de excepciones de C++ permite lidiar con el problema.

- En ambientes de programación concurrente por multiproceso o por el uso de múltiples hilos, es posible sincronizar efectiva y eficientemente el acceso a través de los punteros "`lkptr<X>`" utilizando un semáforo, que es un mecanismo simple y barato para el control de acceso.

Es prematuro afirmar que C++ se puede usar con la misma facilidad que Java para desarrollar aplicaciones, pero los punteros inteligentes "`lkptr<X>`" definitivamente contribuyen a que más aplicaciones Java sean implementadas en C++. Sí se puede afirmar que cada vez quedan menos espacios en donde resulta mejor una implementación Java en lugar de una implementación C++.

10 Agradecimientos

David Chavez y Alejandro Di Mare aportaron varias observaciones y sugerencias importantes para mejorar este trabajo. Francisco Arroyo sugirió las ideas principales para el manejo de concurrencia.

Referencias

1. Elcel Technology: OpenTop Reference Manual, 2007. http://www.elcel.com/docs/opentop/API/ot/ManagedObject.html 9
2. Insolvibile, Gianluca: Garbage Collection in C Programs, Linux Journal, Article 6679, 2003. http://www.linuxjournal.com/article/6679
3. Di Mare, Adolfo: Una Clase Matriz Chirrisquitica Escrita en C++, Reporte Técnico ECCI-2004-02, Escuela de Ciencias de la Computación e Informática, Universidad de Costa Rica, 2004. http://www.di-mare.com/adolfo/p/Matrix.htm
4. Di Mare, Adolfo: Una abstracción C++ completa de la clase árbol, Reporte Técnico ECCI-2005-01, Escuela de Ciencias de la Computación e Informática, Universidad de Costa Rica, 2005. http://www.di-mare.com/adolfo/p/TreeCpp.htm http://www.di-mare.com/adolfo/p/TreeCpp.doc
5. Di Mare, Adolfo: ¡No se le meta al Rep!, Reporte Técnico ECCI-2007-01, Escuela de Ciencias de la Computación e Informática, Universidad de Costa Rica, 2007. http://www.di-mare.com/adolfo/p/Rep.htm
6. Sharon, Yonat: Smart Pointers - What, Why, Which?, 1999. http://ootips.org/yonat/4dev/smart-pointers.html
7. Stroustrup, Bjarne: The C++ Programming Language, 3rd edition, ISBN 0-201-88954-4; Addison-Wesley, 1998. http://www.research.att.com/~bs/3rd.html

Código fuente

lkptr.zip: Todos los fuentes

```
http://www.di-mare.com/adolfo/p/lkptr.htm#fuentes
http://www.di-mare.com/adolfo/p/lkptr/lkptr.zip
```