ISSN 1409-2441

Ingeniería

Revista de la Universidad de Costa Rica
ENERO/DICIEMBRE 2000 - VOLUMEN 10 - Nº 1 y 2

Ingeniería 10 (1,2): 81-91, 2000 San José, Costa Rica

¿CÓMO ESCONDER DATOS EN PUNTEROS?

Adolfo Di Mare H.[1]

Resumen

Con frecuencia los datos que existen en memoria dinámica están, por lo menos, alineados en frontera de palabra. En este artículo, se muestra cómo aprovechar esto para almacenar dentro de un valor *booleano* el puntero para evitar usar un campo completo en estructuras enlazadas.

Summary

Oftentimes data allocated in the heap is, at least, word aligned. I show here how to take advantage of this fact, to store within a pointer a *boolean* value that can be used to avoid using a full fledged field in linked structures.

1. INTRODUCCIÓN

Después de escribir la lista "parametrizable" del lenguaje C descrita en [DiM-1999] se trata de incluir una función equivalente al método `cl_next()`, que consiste en el avance de un puntero "p" al siguiente nodo de la misma, pero evitando como argumento del método el uso de la lista "L".

```
cl_link* cl_next (const clist *L, cl_link *p) {
    /*=========================================*\
    | Returns the position that comes after "p"  |
    | - After the last position, returns NULL.   |
    \*=========================================*/
    return ( p ==  L->last ? NULL : p-next );
} /*              ~~~                        */
    /* tomado de clist.h y clist.c en [DiM-1999] */
```

Listado No. 1: El método `cl_next()`

En el Listado No. 1 está el método **`cl_next()`**, que usa el parámetro "L" para determinar si el puntero "p" denota el último nodo de la lista, en tal caso el valor siguiente debe ser el puntero nulo **NULL**. La desventaja de usar a "L" como parámetro es que no es posible transformar esta implementación a C++, cambiando los punteros de tipo "**cl_link**" por elementos o "iteradores" similares a los usados en la biblioteca estándar del lenguaje C++, STL [*Standard Template Library*] [Mus-1994], pues para los "iteradores" STL siempre es posible calcular el siguiente nodo, que es referencia del "iterador" por medio del operador de incremento "++" (mediante la sobrecarga de "**list::iterator::operator++()**"), por consiguiente, en la implementación del operador "++" nunca se utiliza la lista como argumento.

[1] M.Sc., Prof., Esc. de Ciencias de la Computación e Informática, Fac. de Ingeniería, Univ. de Costa Rica.

```
template <class T, class A = allocator<T> >
class list {
    class node {
        node *_next, *_prev;
        T    _val; // valor almacenado
        // ...
    }; // node
    class iterator {
        node *_p;
    public:
        iterator& operator++() {  // ++p
            this->_p = _p->_next;
            return *this;
        }
        iterator  operator++(int);  // p++
        // ...
    }; // iterator
    iterator end() { return 0; }
    // ...
}; // list
```

Listado No. 2 : Iteradores para la lista STL

En el Listado No. 2 está el prototipo de la operación "++" de una lista STL, que tiene el mismo efecto de `cl_next()` en una lista "parametrizable" para el lenguaje C. La razón por la que los "iteradores" de los contenedores STL no necesitan como argumento el *container* es la siguiente: para recorrerlo, se usan dos "!iteradores", y el segundo denota un valor que está "fuera" del ***container*** ("`end()`"). Por consiguiente, los "iteradores" para las listas STL en realidad son punteros disfrazados [Nel-1995].

```
container C;
container::iterator p;
for (p=C.begin(); p != C.end(); p++) {
    cout << (*p); // p.operator*()
}
```

Listado No. 3 : Paradigma STL para recorrer un contenedor

En el Listado No. 3 se muestra la forma en que un programador cliente de la biblioteca STL recorre un ***container*** "C", para procesar todos los valores en él almacenados. Para el operador de avance "++" no se requiere usar al *container* "C" como argumento, pues el iterador "`end()`" denota a un valor que ya no está dentro del mismo.

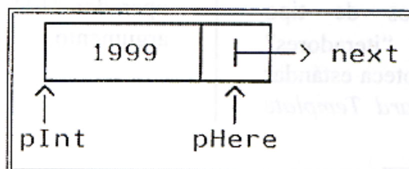

Figura No. 1: Diagrama de clist.c

En la Figura No. 1 se muestra el diagrama o modelo de una lista en la que el último nodo señala al puntero nulo `NIL==0`. En caso de que el método `operator++()` del iterador, el cual hace que se avance hacia el siguiente nodo cuando se está en el último, el valor retornado será `0`, el cual invariablemente retorna a `list::Iterator::end()`. En este caso, nunca es necesario usar el puntero que contiene la lista "`L`" para avanzar con un iterador, al siguiente nodo.

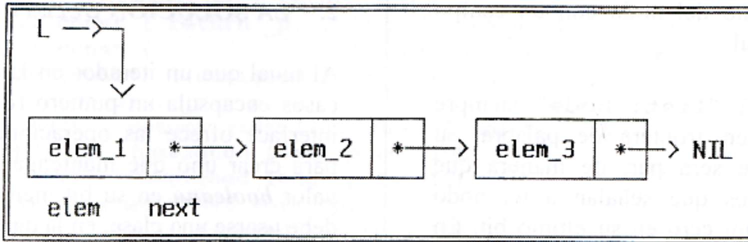

Figura No. 2: Diagrama de clist.c

En la Figura No. 2 se muestra cómo está aplicada la lista circular descrita en [DiM-1999]: el último nodo apunta al primero, y la lista contiene un puntero al último nodo. De esta manera, es posible insertar valores, tanto al principio de la lista como al final, en una cantidad constante de tiempo O(1). Sin embargo, si un puntero "`p`" denota algún nodo de la lista, no hay forma de saber cuándo "p" denota el último nodo, lo que hace imposible usar esta lista como base para poner en práctica el paradigma STL, de acuerdo con la especificación del Listado No. 2, pues no es posible implementar el operador de avance "++".

Para ejecutar el operador de avance del iterador, es necesario incluir un campo adicional en cada nodo de la lista, para que este pueda determinar si ha llegado o no, al final de la lista.

```
template <class T, class A = allocator<T> >
class list {
    class node {
        node *_next, *_prev;
        T     _val;
        bool  _marked;  // ¿puntero final?
        // ...
    }; // node
    class iterator {
        node *_p;
    public:
        iterator& operator++() {    // ++p
            _p = (_p->_marked ? 0 : _p->_next);
            return *this;
        }  // ...
    }; // iterator
    iterator end() { return 0; } // ...
}; // list
```

Listado No. 4: Nodo con marca

En el Listado No. 4 al nodo se le ha agregado el campo "`_marked`", que indica si él está marcado como el nodo final. Para aplicar la operación de avance del iterador cabe verificar si el nodo apuntado está marcado, en tal caso lo que procede es retornar el puntero nulo "0". Esta solución es efectiva, pero requiere engrosar el tamaño del nodo con un campo ***booleano*** adicional.

Como los nodos "`list::node`" siempre están alineados en frontera de palabra, su dirección siempre será par, de manera que todos los punteros que señalan a un nodo siempre tendrán un cero en su último bit. En ese bit se puede almacenar un valor ***booleano*** si se tiene el cuidado de restablecerlo a cero antes de referenciar el puntero. Por ello, otra solución (en términos de espacio), es almacenar dentro del puntero "`_p`" que contiene el valor ***booleano*** "`_marked`". Para saber si el iterador apunta al último nodo, bastará invocar la operación "`_p.marked()`".

2. LA SOLUCIÓN `ptrbit`

Al igual que un iterador en la mayoría de los casos encapsula un puntero (debido a que su interface ofrece las operaciones del mismo), para crear uno que mantenga almacenado un valor ***booleano*** en su bit menos significativo, debe usarse una clase, en la que por sobrecarga estén disponibles las operaciones que caracterizan a los punteros.

```
template <class T> class pointer {
    T *_p;  // el puntero
public:
    pointer()                        {};
    pointer(T* p)          : _p(p)   {};
    pointer(pointer &o)    : _p(o._p) {};
    pointer& operator++()        { ++_p;  return *this; } // ++p
    pointer& operator--()        { --_p;  return *this; } // --p
    pointer  operator++(int)     { pointer<T> o = *this;  // p++
             this->operator++();   return o; }
    pointer  operator--(int)     { pointer<T> o = *this;  // p--
             --(*this);            return o; }
    pointer& operator+=(int i) { _p+=i; return *this; }
    pointer& operator-=(int i) { _p-=i; return *this; }
    pointer& operator=(pointer p) { _p = p._p;
                                   return *this; }
    friend int operator==(pointer, pointer);
    friend int operator!=(pointer, pointer);
    friend unsigned long operator-(pointer, pointer);
    T& operator*() { return *_p; }  // derreferencia
    operator int() { return _p!=0; }
}; // pointer
```

Listado 5: Operaciones para punteros `pointer.h`

En el Listado No. 5 se muestran las operaciones que caracterizan a un puntero, estas son: inicialización, incremento, asignación, comparación, diferencia, referencia y conversión.

La clase "`pointer`" está definida como una plantilla para facilitar su uso. Así que el bit menos significativo de un puntero almacene un valor ***booleano*** hay que cambiar la aplicación de cada una de estas operaciones.

```
/* ptrbit.h        v1.0  (C) 2000 adolfo@di-mare.com */
#ifndef   PTRBIT_H
#define   PTRBIT_H
class ptrbit {
public:
    ptrbit() {}
    ptrbit(      void*    x) : _p(x)      {}
    ptrbit(const ptrbit& x) : _p(x._p) {}
    void* operator=(void* x) { return (_p = x); }
    void* ai() const { return _p; } // ai() ==> As Is
    void* vp() const {              // vp() ==> Void Pointer
        return (void*) ((unsigned long)_p & ~1UL); }
    void set(void* p) { _p = p; }
    int  marked() { return (int)((unsigned long)_p & 1UL); }
    void mark()     { (unsigned long)_p |=  1UL;  }
    void unmark() { (unsigned long)_p &= ~1UL; }
    static int aligned(const void *p)
        { return 0 == ((unsigned long)p & 1UL); }
private:
    void operator*()   {}               // forbid dereferencing
    operator void*()  { return _p; } // forbid conversion
private: // ptrbit member fields
    void *_p;
}; // ptrbit
inline int operator==(const ptrbit& x, const ptrbit& y) {
    return x.ai() == y.ai(); }
inline int operator!=(const ptrbit& x, const ptrbit& y) {
    return !(x == y); }

// This implementation requires long to hold a pointer
class ptrbit_check_that_long_holds_a_void_pointer__ {
    char _[sizeof(unsigned long) == sizeof(void*)];
};
#endif /* PTRBIT_H */
/* EOF:   ptrbit.h */
```

Listado 6: Clase `ptrbit`

El Listado No. 6 es la clase "`ptrbit`", la que provee las operaciones necesarias para, que en el bit menos significativo del puntero, se pueda almacenar un valor ***booleano***. Por brevedad, a la clase "`ptrbit`" se presentan los operadores necesarios para aplicar la lista circular "`clistfT<T>`". Aunque no se han usado plantillas, es sencillo derivar de "`ptrbit`" una plantilla general similar a la de la clase "`pointer<T>`" del Listado No. 5.

El método "`ai()`" ("*As Is*") sirve para obtener, desde una clase cliente, el valor almacenado en el puntero sin borrar a cero el bit menos significativo, caso contrario al método "`vp()`" que sí pone el cero en el bit, lo que se logra usando el operador **C++** de bits "`|`" con la máscara binaria "`~1`" que tiene menos números binarios en el bit menos significativo. El método "`set()`" sirve para cambiar el valor del puntero. Por otro lado, los métodos "`mark()`" "`unmark()`" y "`marked()`" se utilizan para obtener el valor ***booleano*** almacenado. Se supone que el valor "`V`" al que apunta un variable "`ptrbit`" siempre está alineado, es decir, su dirección hace verdadera la expresión **`ptrbit::aligned(&V)`**.

A diferencia de las listas tradicionales (ejemplo: la utilizada en la biblioteca STL), la lista circular **`clistfT<T>`** tiene la

particularidad de que todas las instancias de la misma siempre usan el mismo código objeto, en su aplicación (salvo para las operaciones "`inline`"). Por ello, la clase "`clistfT<T>`" está aplicada en dos archivos: el encabezado `clistf.h` que es tradicional en C++, y el archivo `clistf.cpp` que contiene los algoritmos más importantes de la lista. En esta aplicación se usan las ideas expuestas en [DiM-1999] acondicionar la lista uniendo campos de enlace, en lugar de unir sus nodos., lo cual está mostrado en la figura No. 2.

```
#include "ptrbit.h"
class clistf {      // circular list
public:
    class list_link;
    class itr : public ptrbit {
    public:
        friend class clistf; friend class list_link;
    clistf() : _last(0) {};
        itr() {}
        itr(itr       &o) : ptrbit (        o ) {}
        itr(list_link *o) : ptrbit ( (void*) o ) {}

        itr& operator++();      // ++i
        itr  operator++(int) { // i++
            itr tmp = *this;
            this->operator++(); // ++*this;
            return tmp;
        }
        itr prev();
    protected:
        list_link * ai() { return (list_link*) (ptrbit::ai()); }
        list_link * um() { return (list_link*) (ptrbit::vp()); }
        void operator=(list_link *x) { set( (void*) x); }
        void operator=(itr       &x) { set ( x.ai()  ); }
    }; // itr

    class list_link {
        itr next;
    public:
        list_link() {}
        friend class clistf; friend class itr;
    };
    unsigned count();
    int empty() { return _last == end(); }
    int valid(itr);
    void link_after( itr where, list_link &what);
    itr  unlink_after(itr where);
    void push_front(list_link &what) { link_after(0, what); }
    void push_back( list_link &what) { link_after(_last, what); }
    itr last()    { return _last; }
    itr begin();
    itr end()     { return 0; }
    itr nth(unsigned n);
    unsigned pos(itr i);
private:
    clistf(const clistf&) : _last(0) {}    // forbid copy
    void   operator=(const clistf&)  {}
private:        // member fields
    itr _last; // points to last, last points to first
}; // clistf
```

Listado No. 7: Clase `clistf`

La clase `clistf`, que se muestra en el Listado No. 7 tiene las operaciones más importantes de la lista, por ejemplo la de inserción y borrado. También para avanzar y retroceder entre los valores de la lista usando iteradores. La clase "`itr`" provee estos últimos, similares a los usados en la biblioteca STL. Por su parte, la clase privada "`list_link`" es el nodo de la lista.

```
inline clistf::itr clistf::begin() {
    return (_last.ai() == 0 ? 0 : _last.um()->next.um());
}

inline clistf::itr& clistf::itr::operator++() { // ++i
    set (ai()->next.ai());
    if (this->marked()) {
        set(0);
    }
    return *this;
}
```

Listado No. 8

En el Listado No. 8 se pone en práctica la operación "`++itr`" de los iteradores, que sirve para avanzar al siguiente nodo de la lista. Aquí, se usa el método "`ptrbit::set()`" para cambiar el valor del puntero que contiene el iterador, de manera que apunte al siguiente nodo de la lista. Como la operación de incremento se ejecuta únicamente cuando el iterador todavía apunta a un nodo de la lista, entonces el puntero que éste contiene nunca es marcado como el último de la lista. Por eso, al obtener el valor del mismo, se usa el método "`ptrbit::ai()`", el cual no trata de borrar el último bit del puntero, que no puede estar prendido.

En esta adecuación de la lista hay que tomar en cuenta que el último puntero de la misma, que a su vez señala al primer nodo, es un puntero especial, "marcado". En esta condición, al avanzar hacia el siguiente nodo, siempre es posible determinar si se ha llegado al final de la lista, es decir, si el puntero no tiene el bit menos significativo prendido. Por lo tanto entonces el iterador corresponde a un nodo intermedio de la lista, pues sólo el campo "`next`" del último nodo contiene un puntero marcado.

Debido a que la lista es circular (ver Figura No. 2), si el iterador apunta al último nodo, al avanzar hacia el siguiente se obtendrá un puntero dirigido al primer nodo, pero éste estará marcado en su último bit, por eso si al avanzar el puntero del iterador se obtiene un puntero marcado, esto indica que ya se ha cruzado del último al primer nodo, por lo que el operador de avance del iterador debe retornar. Éste es el valor "0" y no un puntero apuntando al primer nodo de la lista, que es lo que "`clistf::end()`" siempre retorna.

```
#include <stddef.h> // offsetof()
template <class T>
class clistfT : public clistf {
private:
    struct node {
        T         e;  // element
        list_link lk; // link field for the list
```

```
            node(T& n)  : e(n), lk() {}
        }; // node

    public:
        class iterator : public itr {
            friend class clistfT;
        public:
            iterator()              : itr ( ) {}
            iterator(itr &o)        : itr (o) {}
            iterator(list_link *o)  : itr (o) {}
            void operator=(itr &x)   { itr::operator=(x); }

    private:
            node* np() {  // Node Pointer
                return ((node*)((char*)vp() - offsetof(node, lk)));
            }
    public:
            T& operator*() { return np()->e; }
        }; // iterator

    public:
        ~clistfT() {
            while (!empty()) {
                iterator i = unlink_after(0);
                delete i.um();
            }
        }
    }; // clistfT<T>
```

Listado No. 9: clase `clistfT<T>`

La clase `clistf` no es una plantilla, pues ha sido construida de manera que cualquier lista, independientemente del tipo de datos que almacene, use el mismo código objeto para incluir sus operaciones. La adecuación de las operaciones de la lista está en el archivo "`clistf.cpp`", y es común a todas. En el Listado No. 9 se muestra la clase `clistfT<T>`, derivada de `clistf`, que en realidad es una envoltura alrededor de la clase `clistf`. Lo mismo ocurre con la clase `clistfT<T>::iterator`, derivado de `clistfT<T>::itr`. El nodo de `clistfT<T>` contiene el campo de enlace "`lk`" que se necesita para enlazar todos los valores de una `clistf`, y además exporta el método "`T& iterator::operator*()`" que sirve para tomar un iterador y, mediante el operador de referencia "`*`", obtener el valor almacenado en la lista. Para ello, es necesario ajustar el puntero del iterador, que apunta al campo de enlace y trasladarlo al principio del nodo de `clistfT<T>`. Este trabajo lo hace la operación "`np()`" (*Node Pointer*) usando la macro "`offsetof()`".

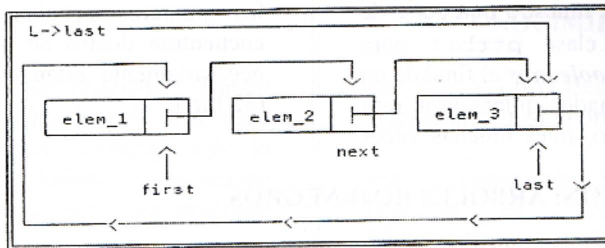

Figura No. 3: `pHere` transformado en `pInt` usando `iterator::np()`

Las versiones de la lista se obtienen al instanciar la plantilla `clistfT<T>`. La forma de lograr que compartan la misma adecuación de las operaciones, es adecuarlas de manera que manipulen los campos de enlace de la lista, en lugar de que manipulen nodos completos. Por otro lado, la operación "`np()`" transforma un puntero a un campo de enlace, en un puntero al nodo, por ello debe restársele el desplazamiento marcado desde el principio del nodo, como se muestra en la Figura No. 3. En [DiM-1999] se encuentra una discusión detallada de cómo lograr la relación entre los métodos de la lista.

3. ¿CUÁNTOS BITS APROVECHAR?

Debido a que la mayor parte de los objetos que residen de la memoria dinámica, quedan alineados en frontera de doble palabra, y como la mayoría de los computadores actuales tiene procesadores de 32 bits, es lógico usar los dos últimos bits de un puntero para almacenar dos valores *booleanos* independientes.

```
class ptrbit {
    // ...
public:
    void* vp() const { return (void*) ((long)_p & ~(1L+2L)); }

    int  marked() { return (int)((unsigned long)_p & 1UL); }
    void mark()   { (unsigned long)_p |=  1UL; }
    void unmark() { (unsigned long)_p &= ~1UL; }  // bit 0

    int  checked() { return (((unsigned long)_p & 2L)?1:0); }
    void check()   { (unsigned long)_p |=  2L; }
    void uncheck() { (unsigned long)_p &= ~2L; }  // bit 1
    int bit(int n) {
        return ( n==0 ? marked() : checked() );
    }
    void bit(int n, int v) {
         (n == 0 ? ( v==0 ? unmark()  : mark()  )
                 : ( v==0 ? uncheck() : check() ) ); }

    static int aligned(const void *p)
         { return 0 == ((unsigned long)p & (1L+2L)); }
    // ...
}; // ptrbit
```

Listado No. 10: Aprovechamiento de dos bits del puntero

En el Listado No. 10 se muestra una parte de la adecuación de la clase `ptrbit` para almacenar dos valores ***booleanos*** al final de un puntero. Si un programador quiere usar más bits, debe ser cuidadoso, pues muchas veces los punteros señalan a campos que se encuentran dentro de un objeto, los que no necesariamente están en frontera de palabra (32 bits).

4. OTRA APLICACIÓN: ÁRBOLES ROJI-NEGROS

```
ptrbit <class Key, class Ty, class Kf, class P, class A>

class tree {
    typedef typename
        A::rebind<void>::other::pointer Genptr;
    enum redblack { Red, Black }; // bit values

    struct Node {
        Genptr   Left;
        Genptr   Parent;  // the pointers
        Genptr   Right;
        redblack Color;   // node color
        Ty       Value;   // info field
    };
   // more stuff deleted...
}; // tree
```

Listado 11: Definición del nodo de un árbol roji-negro [Pla-1997]

También se puede usar la clase `ptrbit` para mejorar la adecuación de los ***containers*** asociativos de la biblioteca STL, lo cual generalmente se hace usando un árbol roji-negro. Los nodos de este tipo de árbol deben estar decorados por un valor, que se ejecuta usando un campo de tipo "`Color`" en el que se almacena uno de dos valores: `red` o `black` [Sha-1998], como se muestra en el extracto de la clase "`tree`" en el Listado No. 11. Debido a que este campo sólo puede tomar dos valores, se permite usar una variable ***booleana***. La forma de reducir el tamaño del nodo es eliminar el campo "`redblack`", y en consecuencia hay que almacenar su valor en alguno de los punteros.

Para eliminar el campo "`redblack`" no se puede cambiar el de "`Value`" del nodo, pues éste lo usa el programador cliente de la clase y en él no se puede almacenar ningún valor. Por eso no queda más que codificar el color del nodo en el bit menos significativo del puntero, que señala al nodo, en el campo "`Left`" o "`Right`".

Por otra parte, los algoritmos de manipulación del árbol deben ser modificados, para que incluyan el color del nodo, que antes estuvo almacenado en el campo "`redblack`", del bit menos significativo del puntero que apunta nodo. No resulta difícil hacer esta modificación pero sí tedioso. Para ajustar los algoritmos, resulta trabajar con una clase derivada de "`ptrbit`" que permita usar verificación fuerte de tipos, la que se puede obtener derivando de "`ptrbit`" una clase similar a la "`pointer<T>`" propuesta en el Listado No. 5.

5. CONCLUSIÓN

Es posible almacenar datos en los bits menos significativos de un puntero, lo que puede ayudar a mejorar la eficiencia de algunos algoritmos. El programador debe sopesar el incremento en eficiencia del uso de espacio con el aumento en la complejidad de los algoritmos, en lugar de usar punteros

decorados con valores *booleanos*.

En general es complicado programar usando punteros, y será más difícil hacerlo si se usan "decorados". En algunas ocasiones vale la pena hacer el esfuerzo adicional, como ocurre al realizar la lista circular que se ha mostrado en este artículo.

6. RECONOCIMIENTOS

Lic. Carlos Loría Beeche tuvo la gentileza de criticar una versión preliminar de este artículo.

Esta investigación se realizó dentro del proyecto de investigación 326-98-391 "Polimorfismo uniforme más eficiente", inscrito ante la Vicerrectoría de Investigación de la Universidad de Costa Rica. La Escuela de Ciencias de la Computación e Informática aportó fondos para este trabajo.

7. BIBLIOGRAFÍA

[DiM-1999] Di Mare, Adolfo: ***C Parametrized Lists***, Technical Report ECCI-99-02, Escuela de Ciencias de la Computación e Informática, Universidad de Costa Rica, 1999.
http://www.di-mare.com/adolfo/p/c-list.htm

[DiM-2000] Di Mare, Adolfo: ***C Iterators***, Revista Acta Académica, Número 26, pp [14-30], ISSN 1017-7507, Mayo 2000.
http://www.uaca.ac.cr/acta/2000may/c-iter.htm
http://www.di-mare.com/adolfo/p/c-iter.htm

[Mus-1994] David R. Musser: ***The Standard Template Library*** *(documento disponible en Internet)*, Computer Science Department, Rensselaer Polytechnic Institute, 1994.
http://www.cs.rpi.edu/~musser/stl.html

[Nel-1995] Nelson, Mark: ***C++ Programmer's Guide to the Standard Template Library***, IDG Books Worldwide, ISBN 1-56884-314-3, 1995.

[Pla-1997] Plauger, P. J.: ***Implementing Associative Containers***, C/C++ Users Journal, Vol.15 No.5, pp [8, 10, 12, 14, 16, 18, 19], Mayo 1997.

[Sha-1998] Shankel, Jason: ***STL's Red-Black Trees***, Dr. Dobb's Journal, No.284, pp [54, 56, 58, 60], Abril 1998.

8. ACERCA DEL AUTOR

Adolfo Di Mare: Investigador costarricense en la Escuela de Ciencias de la Computación e Informática [ECCI] de la Universidad de Costa Rica [UCR], en donde ostenta el rango de Profesor Catedrático. Trabaja en las tecnologías de Programación e Internet. Es Maestro Tutor del Stvdivm Generale de la Universidad Autónoma de Centro América [UACA], en donde ostenta el rango de Catedrático y funge como Consiliario Académico. Obtuvo la Licenciatura en la Universidad de Costa Rica, la Maestría en Ciencias en la Universidad de California, Los Angeles [UCLA], y el Doctorado (Ph.D.) en la Universidad Autónoma de Centro América.