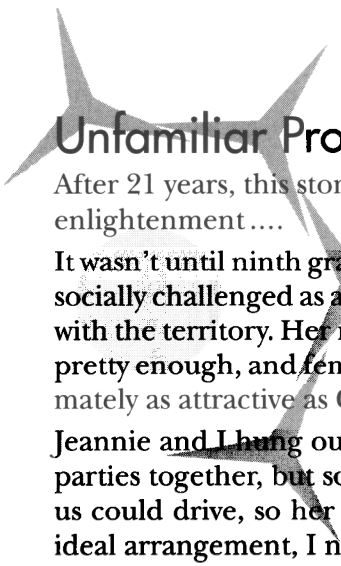
Chapter 15

Linked Lists and Unintended Challenges

Chapter 15



Unfamiliar Problems with Familiar Data Structures

After 21 years, this story still makes me wince. Oh, the humiliations I suffer for your enlightenment....

It wasn't until ninth grade that I had my first real girlfriend. Okay, maybe I was a little socially challenged as a kid, but hey, show me a good programmer who wasn't; it goes with the territory. Her name was Jeannie Schweigert, and she was about four feet tall, pretty enough, and female—and willing to go out with me, which made her approximately as attractive as Cheryl Tiegs, in my book.

Jeannie and I hung out together at school, and went to basketball games and a few parties together, but somehow the two of us were never alone. Being 14, neither of us could drive, so her parents tended to end up chauffeuring us. That's a next-to-ideal arrangement, I now realize, having a daughter of my own (ideal being exiling all males between the ages of 12 and 18 to Tasmania), but at the time, it drove me nuts. You see... ahem... I had never actually kissed Jeannie—or anyone, for that matter, unless you count maiden aunts and the like—and I was dying to. At the same time, I was terrified at the prospect. What if I turned out to be no good at it? It wasn't as if I could go to Kisses 'R' Us and take lessons.

My long-awaited opportunity finally came after a basketball game. For a change, *my* father was driving, and when we dropped her off at her house, I walked her to the door. This was my big chance. I put my arms around her, bent over with my eyes closed, just like in the movies....

And whacked her on the top of the head with my chin. (As I said, she was only about four feet tall.) And I do mean whacked. Jeannie burst into hysterical laughter, tried to calm herself down, said goodnight, and went inside, still giggling. No kiss.

I was a pretty mature teenager, so this was only slightly more traumatic than leading the Tournament of Roses parade in my underwear. On the next try, though, I did manage to get the hang of this kissing business, and eventually even went on to have a child. (Not with Jeannie, I might add; the mind boggles at the mess I could have made of *that* with her.) As it turns out, none of that stuff is particularly difficult; in fact, it's kind of enjoyable, wink, wink, say no more.

When you're dealing with something new, a little knowledge goes a long way. When it comes to kissing, we have to fumble along the learning curve on our own, but there are all sorts of resources to help speed up the learning process when it comes to programming. The basic mechanisms of programming—searches, sorts, parsing, and the like—are well-understood and superbly well-documented. Treat yourself to a book like *Algorithms,* by Robert Sedgewick (Addison Wesley), or Knuth's *The Art of Computer Programming* series (also from Addison Wesley; and where was Knuth with *The Art of Kissing* when I needed him?), or practically anything by Jon Bentley, and when you tackle a new area, give yourself a head start. There's still plenty of room for inventiveness and creativity on your part, but why not apply that energy on top of the knowledge that's already been gained, instead of reinventing the wheel? I know, reinventing the wheel is just the kind of challenge programmers love—but can you really afford to waste the time? And do you honestly think that you're so smart that you can out-think Knuth, who's spent a lifetime at this stuff and happens to be a genius?

Maybe you can—but I sure can't. For example, consider the evolution of my understanding of linked lists.

Linked Lists

Linked lists are data structures composed of discrete elements, or nodes, joined together with links. In C, the links are typically pointers. Like all data structures, linked lists have their strengths and their weaknesses. Primary among the strengths are: simplicity; speedy sequential processing; ease and speed of insertion and deletion; the ability to mix nodes of various sizes and types; and the ability to handle variable amounts of data, especially when the total amount of data changes dynamically or is not always known beforehand. Weaknesses include: greater memory requirements than arrays (the pointers take up space); slow non-sequential processing, including finding arbitrary nodes; and an inability to backtrack, unless doubly-linked lists are used. Unfortunately, doubly linked lists need more memory, as well as processing time to maintain the backward links.

Linked lists aren't very good for most types of sorts. Insertion and bubble sorts work fine, but more sophisticated sorts depend on efficient random access, which linked

lists don't provide. Likewise, you wouldn't want to do a binary search on a linked list. On the other hand, linked lists are ideal for applications where nothing more than sequential access is needed to data that's always sorted or nearly sorted.

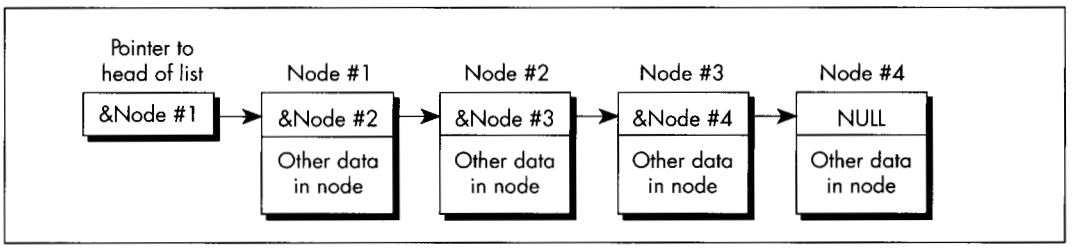
Consider a polygon fill function, for example. Polygon edges are added to the active edge list in x-sorted order, and tend to stay pretty nearly x-sorted, so sophisticated sorting is never needed. Edges are read out of the list in sorted order, just the way linked lists work best. Moreover, linked lists are straightforward to implement, and with linked lists an arbitrary number of polygon edges can be handled with no fuss. All in all, linked lists work beautifully for filling polygons. For an example of the use of linked lists in polygon filling, see my column in the May 1991 issue of *Dr. Dobb's Journal.* Be warned, though, that none of the following optimizations are to be found in that column.

You see, that column was my first heavy-duty use of linked lists, and they seemed so simple that I didn't even open Sedgewick or Knuth. For hashing or Boyer-Moore searching, sure, I'd have done my homework first; but linked lists seemed too obvious to bother. I was much more concerned with the polygon-related aspects of the implementation, and, in truth, I gave the linked list implementation not a moment's thought before I began coding. Heck, I had handled *much* tougher programming problems in the past; surely it would be faster to figure this one out on my own than to look it up.

Not!

The basic concept of a linked list—the one I came up with for that *DDJ* column—is straightforward, as shown in Figure 15.1. A head pointer points to the first node in the list, which points to the next node, which points to the next, and so on, until the last node in the list is reached (typically denoted by a **NULL** next-node pointer). Conceptually, nothing could be simpler. From an implementation perspective, however, there are serious flaws with this model.

The fundamental problem is that the model of Figure 15.1 unnecessarily complicates link manipulation. In order to delete a node, for example, you must change the preceding



The basic concept of a linked list.
Figure 15.1

node's **NextNode** pointer to point to the following node, as shown in Listing 15.1. (Listing 15.2 is the header file LLIST.H, which is **#include**d by all the linked list listings in this chapter.) Easy enough—unless the preceding node happens to be the head pointer, which doesn't *have* a **NextNode** field, because it's not a node, so Listing 15.1 won't work. Cumbersome special code and extra information (a pointer to the head of the list) are required to handle the head-pointer case, as shown in Listing 15.3. (I'll grant you that if you make the next-node pointer the first field in the **LinkNode** structure, at offset 0, then you could successfully point to the head pointer and pretend it was a **LinkNode** structure—but that's an ugly and potentially dangerous trick, and we'll see a better approach next.)

LISTING 15.1 L15-1.C

```
/* Deletes the node in a linked list that follows the indicated node.
   Assumes list is headed by a dummy node, so no special testing for
   the head-of-list pointer is required.  Returns the same pointer
   that was passed in. */

#include "llist.h"
struct LinkNode *DeleteNodeAfter(struct LinkNode *NodeToDeleteAfter)
{
   NodeToDeleteAfter->NextNode =
         NodeToDeleteAfter->NextNode->NextNode;
   return(NodeToDeleteAfter);
}
```

LISTING 15.2 LLIST.H

```
/* Linked list header file. */
#define MAX_TEXT_LENGTH 100   /* longest allowed Text field */
#define SENTINEL  32767       /* largest possible Value field */

struct LinkNode {
   struct LinkNode *NextNode;
   int Value;
   char Text[MAX_TEXT_LENGTH+1];
   /* Any number of additional data fields may by present */
};
struct LinkNode *DeleteNodeAfter(struct LinkNode *);
struct LinkNode *FindNodeBeforeValue(struct LinkNode *, int);
struct LinkNode *InitLinkedList(void);
struct LinkNode *InsertNodeSorted(struct LinkNode *,
   struct LinkNode *);
```

LISTING 15.3 L15-3.C

```
/* Deletes the node in the specified linked list that follows the
   indicated node. List is headed by a head-of-list pointer; if the
   pointer to the node to delete after points to the head-of-list
   pointer, special handling is performed. */
#include "llist.h"
struct LinkNode *DeleteNodeAfter(struct LinkNode **HeadOfListPtr,
   struct LinkNode *NodeToDeleteAfter)
{
   /* Handle specially if the node to delete after is actually the
      head of the list (delete the first element in the list) */
   if (NodeToDeleteAfter == (struct LinkNode *)HeadOfListPtr) {
      *HeadOfListPtr = (*HeadOfListPtr)->NextNode;
```

```
    } else {
        NodeToDeleteAfter->NextNode =
              NodeToDeleteAfter->NextNode->NextNode;
    }
    return(NodeToDeleteAfter);
}
```

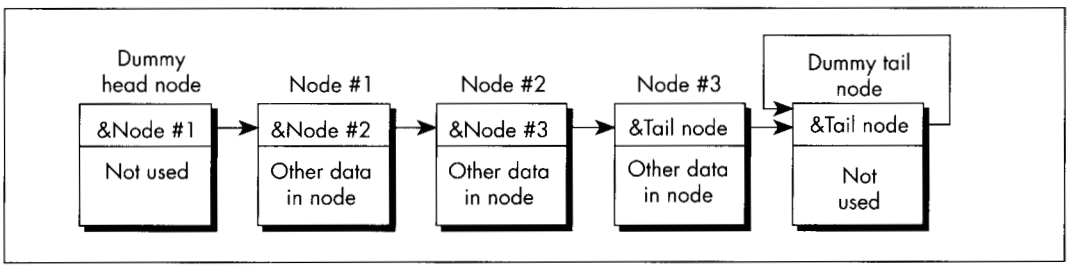
However, it is true that if you're going to store a variety of types of structures in your linked lists, you should start each node with the **LinkNode** field. That way, the link pointer is in the same place in *every* structure, and the same linked list code can handle all of the structure types by casting them to the base link-node structure type. This is a less than elegant approach, but it works. C++ can handle data mixing more cleanly than C, via derivation from a base link-node class.

Note that Listings 15.1 and 15.3 have to specify the linked-list delete operation as "delete the *next* node," rather than "delete this node," because in order to relink it's necessary to access the **NextNode** field of the node preceding the node to be deleted, and it's impossible to backtrack in a singly linked list. For this reason, singly-linked list operations tend to work with the structure preceding the one of interest—and that makes the problem of having to special-case the head pointer all the more acute.

Similar problems with the head pointer crop up when you're inserting nodes, and in fact in all link manipulation code. It's easy to end up working with either pointers to pointers or lots of special-case code, and while those approaches work, they're inelegant and inefficient.

Dummies and Sentinels

A far better approach is to use a *dummy node* for the head of the list, as shown in Figure 15.2. I invented this one for myself the next time I encountered linked lists, while designing a seed fill function for MetaWindows, back during my tenure at Metagraphics Corp. But I could have learned it by spending five minutes with Sedgewick's book.



Using a dummy head and tail node with a linked list.
Figure 15.2

The next-node pointer of the head node, which points to the first real node, is the only part of the head node that's actually used. This way the same code works on the head node as on the rest of the list, so there are no special cases.

Likewise, there should be a separate node for the tail of the list, so that every node that contains real data is guaranteed to have a node on either side of it. In this scheme, an empty list contains two nodes, as shown in Figure 15.3. Although it is not necessary, the tail node may point to itself as its own next node, rather than contain a **NULL** pointer. This way, a deletion operation on an empty list will have no effect—quite unlike the same operation performed on a list terminated with a **NULL** pointer. The tail node of a list terminated like this can be detected because it will be the only node for which the next-node pointer equals the current-node pointer.

Figure 15.3 is a giant step in the right direction, but we can still make a few refinements. The inner loop of any code that scans through such a list has to perform a special test on each node to determine whether the tail has been reached. So, for example, code to find the first node containing a value field greater than or equal to a certain value has to perform two tests in the inner loop, as shown in Listing 15.4.

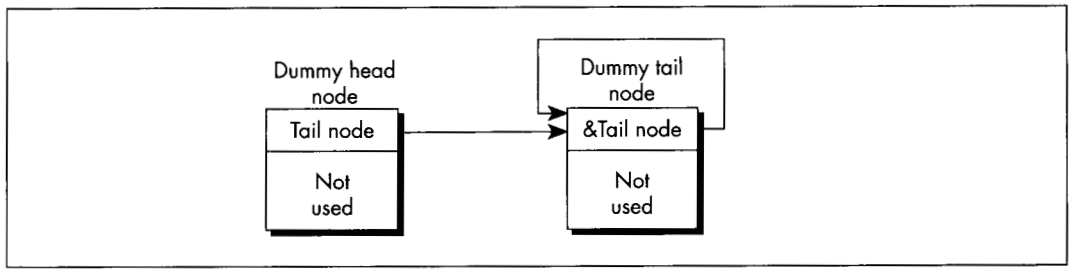
LISTING 15.4 L15-4.C

```
/*    Finds the first node in a linked list with a value field greater
      than or equal to a key value, and returns a pointer to the node
      preceding that node (to facilitate insertion and deletion), or a
      NULL pointer if no such value was found. Assumes the list is
      terminated with a tail node pointing to itself as the next node. */
#include <stdio.h>
#include "llist.h"
struct LinkNode *FindNodeBeforeValueNotLess(
   struct LinkNode *HeadOfListNode, int SearchValue)
{
   struct LinkNode *NodePtr = HeadOfListNode;

   while ( (NodePtr->NextNode->NextNode != NodePtr->NextNode) &&
         (NodePtr->NextNode->Value < SearchValue) )
      NodePtr = NodePtr->NextNode;

   if (NodePtr->NextNode->NextNode == NodePtr->NextNode)
      return(NULL);     /* we found the sentinel; failed search */
   else
      return(NodePtr);  /* success; return pointer to node preceding
                           node that was >= */
}
```

Suppose, however, that we make the tail node a *sentinel* by giving it a value that is guaranteed to terminate the search, as shown in Figure 15.4. The list in Figure 15.4 has a sentinel with a value field of 32,767; since we're working with integers, that's the highest possible search value, and is guaranteed to satisfy any search that comes down the pike. The success or failure of the search can then be determined outside the loop, if necessary, by checking for the tail node's special pointer—but the inside of the loop is streamlined to just one test, as shown in Listing 15.5. Not all linked lists

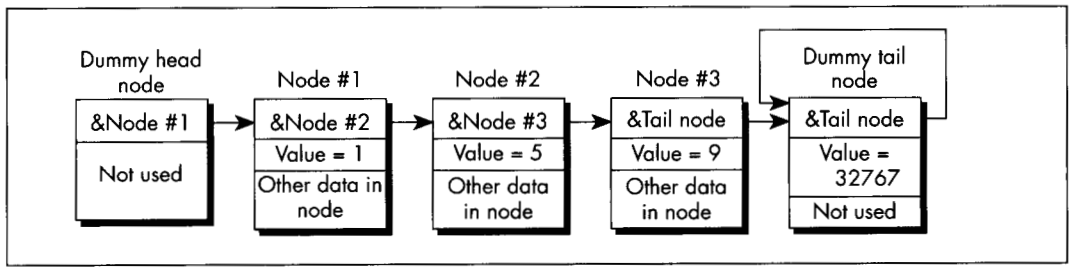


Representing an empty list.
Figure 15.3

lend themselves to sentinels, but the performance benefits are considerable for those that do.

LISTING 15.5 L15-5.C

```
/* Finds the first node in a value-sorted linked list that
   has a Value field greater than or equal to a key value, and
   returns a pointer to the node preceding that node (to facilitate
   insertion and deletion), or a NULL pointer if no such value was
   found. Assumes the list is terminated with a sentinel tail node
   containing the largest possible Value field setting and pointing
   to itself as the next node. */
#include <stdio.h>
#include "llist.h"
struct LinkNode *FindNodeBeforeValueNotLess(
   struct LinkNode *HeadOfListNode, int SearchValue)
{
   struct LinkNode *NodePtr = HeadOfListNode;
   while (NodePtr->NextNode->Value < SearchValue)
      NodePtr = NodePtr->NextNode;
   if (NodePtr->NextNode->NextNode == NodePtr->NextNode)
      return(NULL);      /* we found the sentinel; failed search */
   else
      return(NodePtr);   /* success; return pointer to node preceding
                            node that was >= */
}
```



List terminated by a sentinel.
Figure 15.4

Circular Lists

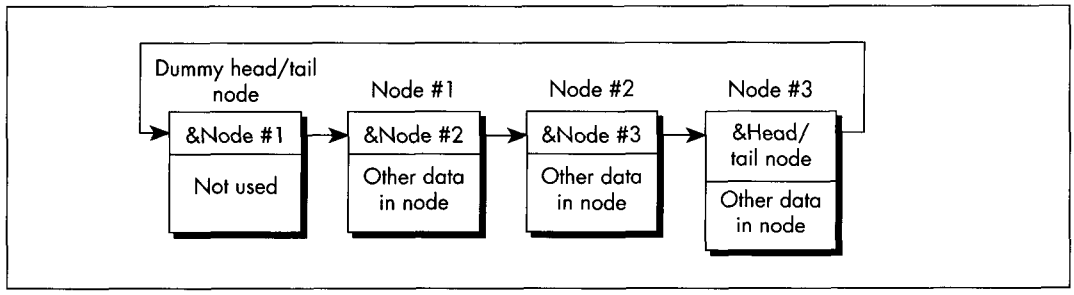
One minor but elegant refinement yet remains: Use a single node as both the head *and* the tail of the list. We can do this by connecting the last node back to the first through the head/tail node in a circular fashion, as shown in Figure 15.5. This head/tail node can also, of course, be a sentinel; when it's necessary to check for the end of the list explicitly, that can be done by comparing the current node pointer to the head pointer. If they're equal, you're at the head/tail node.

Why am I so fond of this circular list architecture? For one thing, it saves a node, and most of my linked list programming has been done in severely memory-constrained environments. Mostly, though, it's just so *neat;* with this setup, there's not a single node or inner-loop instruction wasted. Perfect economy of programming, if you ask me.

I must admit that I racked my brains for quite a while to come up with the circular list, simple as it may seem. Shortly after coming up with it, I happened to look in Sedgewick's book, only to find my nifty optimization described plain as day; and a little while after *that,* I came across a thread in the algorithms/computer.sci topic on BIX that described it in considerable detail. Folks, the information is out there. Look it up *before* turning on your optimizer afterburners!

Listings 15.1 and 15.6 together form a suite of C functions for maintaining a circular linked list sorted by ascending value. (Listing 15.5 requires modification before it will work with circular lists.) Listing 15.7 is an assembly language version of **InsertNodeSorted()**; note the tremendous efficiency of the scanning loop in **InsertNodeSorted()**—four instructions per node!—thanks to the dummy head/tail/sentinel node. Listing 15.8 is a simple application that illustrates the use of the linked-list functions in Listings 15.1 and 15.6.

Contrast Figure 15.5 with Figure 15.1, and Listings 15.1, 15.5, 15.6, and 15.7 with Listings 15.3 and 15.4. Yes, linked lists are simple, but not so simple that a little knowledge doesn't make a substantial difference. Make it a habit to read Knuth or Sedgewick or the like before you write a single line of code.



Representing a circular list.
Figure 15.5

```
   while (NodePtr->NextNode->Value < SearchValue)
      NodePtr = NodePtr->NextNode;
   NodeToInsert->NextNode = NodePtr->NextNode;
   NodePtr->NextNode = NodeToInsert;
   return(NodePtr);
}
```

LISTING 15.7 L15-7.ASM

```
; C near-callable assembly function for inserting a new node in a
; linked list sorted by ascending order of the Value field. The list
; is circular; that is, it has a dummy node as both the head and the
; tail of the list. The dummy node is a sentinel, containing the
; largest possible Value field setting. Tested with TASM.
MAX_TEXT_LENGTH equ 100         ;longest allowed Text field
SENTINEL equ  32767             ;largest possible Value field
LinkNode struc
NextNode dw     ?
Value    dw     ?
Text     db     MAX_TEXT_LENGTH+1 dup(?)
;*** Any number of additional data fields may by present ***
LinkNode ends

        .model  small
        .code

; Inserts the specified node into a ascending-value-sorted linked
; list, such that value-sorting is maintained. Returns a pointer to
; the node after which the new node is inserted.
; C near-callable as:
; struct LinkNode *InsertNodeSorted(struct LinkNode *HeadOfListNode,
;      struct LinkNode *NodeToInsert)
parms   struc
        dw      2 dup (?)       ;pushed return address & BP
HeadOfListNode dw       ?       ;pointer to head node of list
NodeToInsert dw         ?       ;pointer to node to insert
parms   ends

        public  _InsertNodeSorted
_InsertNodeSorted proc  near
        push    bp
        mov     bp,sp                           ;point to stack frame
        push    si                              ;preserve register vars
        push    di
        mov     si,[bp].NodeToInsert            ;point to node to insert
        mov     ax,[si].Value                   ;search value
        mov     di,[bp].HeadOfListNode          ;point to linked list in
                                                ; which to insert
SearchLoop:
        mov     bx,di                           ;advance to the next node
        mov     di,[bx].NextNode                ;point to following node
        cmp     [di].Value,ax                   ;is the following node's
                                                ; value less than the value
                                                ; from the node to insert?
        jl      SearchLoop                      ;yes, so continue searching
                                                ;no, so we have found our
                                                ; insert point
        mov     ax,[bx].NextNode                ;link the new node between
        mov     [si].NextNode,ax                ; the current node and the
        mov     [bx].NextNode,si                ; following node
```

```
        mov     ax,bx                   ;return pointer to node
                                        ; after which we inserted
        pop     di                      ;restore register vars
        pop     si
        pop     bp
        ret
_InsertNodeSorted endp
        end
```

LISTING 15.8 L15-8.C

```
/* Sample linked list program. Tested with Borland C++. */
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#include "llist.h"

void main()
{ int Done = 0, Char, TempValue;
   struct LinkNode *TempPtr, *ListPtr, *TempPtr2;
   char TempBuffer[MAX_TEXT_LENGTH+3];

   if ((ListPtr = InitLinkedList()) == NULL) {
       printf("Out of memory\n");
       exit(1);
   }
   while (!Done) {
      printf("\nA=add; D=delete; F=find; L=list all; Q=quit\n>");
      Char = toupper(getche());
      printf("\n");
      switch (Char) {
         case 'A':               /* add a node */
            if ((TempPtr = malloc(sizeof(struct LinkNode))) == NULL)
            {
               printf("Out of memory\n  );
               exit(1);
             }
            printf("Node value: ");
            scanf("%d", &TempPtr->Value);
            if ((FindNodeBeforeValue(ListPtr,TempPtr->Value))!=NULL)
            {  printf("*** value already in list; try again ***\n");
               free(TempPtr);
            } else {printf("Node text: ");
               TempBuffer[0] = MAX_TEXT_LENGTH;
               cgets(TempBuffer);
               strcpy(TempPtr->Text, &TempBuffer[2]);
               InsertNodeSorted(ListPtr, TempPtr);
               printf("\n");
            }
            break;
         case 'D':               /* delete a node */
            printf("Value field of node to delete: ");
            scanf("%d", &TempValue);
            if ((TempPtr = FindNodeBeforeValue(ListPtr, TempValue))
                 != NULL) {
               TempPtr2 = TempPtr->NextNode; /* -> node to delete */
               DeleteNodeAfter(TempPtr);     /* delete it */
               free(TempPtr2);               /* free its memory */
```

```
            } else {
               printf("*** no such value field in list ***\n")
            break;
         case 'F':                  /* find a node */
            printf("Value field of node to find: ");
            scanf("%d", &TempValue);
            if ((TempPtr = FindNodeBeforeValue(ListPtr, TempValue))
                  != NULL)
               printf("Value: %d\nText: %s\n",
                 TempPtr->NextNode->Value, TempPtr->NextNode->Text);
            else
               printf("*** no such value field in list ***\n");
            break;
         case 'L':                  /* list all nodes */
            TempPtr = ListPtr->NextNode;  /* point to first node */
            if (TempPtr == ListPtr) {     /* empty if at sentinel */
               printf("*** List is empty ***\n");
            } else {
               do {printf("Value: %d\n  Text: %s\n", TempPtr->Value,
                        TempPtr->Text);
                  TempPtr = TempPtr->NextNode;
               } while (TempPtr != ListPtr);
            }
            break;
         case 'Q':
            Done = 1;
            break;
         default:
            break;
      }
   }
}
```

Hi/Lo in 24 Bytes

In one of my *PC TECHNIQUES* "Pushing the Envelope" columns, I passed along one of David Stafford's fiendish programming puzzles: Write a C-callable function to find the greatest or smallest unsigned **int**. Not a big deal—except that David had *already* done it in 24 bytes, so the challenge was to do it in 24 bytes or less.

Such routines soon began coming at me from all angles. However (and I hate to say this because some of my correspondents were *very* pleased with the thought that they had bested David), no one has yet met the challenge—because most of you folks missed a key point. When David said, "Write a function to find the greatest or smallest unsigned **int** in 24 bytes or less," he meant, "Write the **hi** and the **lo** functions in 24 bytes or less—*combined.*"

Oh.

Yes, a 24-byte hi/lo function is possible, anatomically improbable as it might seem. Which I guess goes to show that when one of David's puzzles seems less than impossible, odds are you're missing something. Listing 15.9 is David's 24-byte solution, from which a lot may be learned if one reads closely enough.

LISTING 15.9 L15-9.ASM

```
; Find the greatest or smallest unsigned int.
; C callable (small model); 24 bytes.
; By David Stafford.
; unsigned hi( int num, unsigned a[] );
; unsigned lo( int num, unsigned a[] );

                public _hi, _lo

_hi:            db      0b9h            ;mov cx,immediate
_lo:            xor     cx,cx
                pop     ax              ;get return address
                pop     dx              ;get count
                pop     bx              ;get pointer
                push    bx              ;restore pointer
                push    dx              ;restore count
                push    ax              ;restore return address
save:           mov     ax,[bx]
top:            cmp     ax,[bx]
                jcxz    around
                cmc
around:         ja      save
                inc     bx
                inc     bx
                dec     dx
                jnz     top

                ret
```

Before I end this chapter, let me say that I get a lot of feedback from my readers, and it's much appreciated. Keep those cards, letters, and email messages coming. And if any of you know Jeannie Schweigert, have her drop me a line and let me know how she's doing these days....