Doubly Linked List API

Carl J. Nobile

January 8, 1998

Purpose:

There are many goals to achieve when deciding to write an API. The library of functions should be reenterable, easy to include in an application, platform independent, and reasonably flexible with enough functionality to be usable. These goals can often be contradictory; however, they are achievable with enough forethought and planning.

Writing an API for a link list came about after many years of struggling with data storage problems. I would often write link-list code embedded in my application, exposing all of its innards to the application. This was often a nightmare to weed through as the application grew in functionality and complexity. If more than one link list was needed my beard would thin. So that is how this API was born; I hope it is as useful for you as it has been for me.

Overview:

The API breaks down into six function groups: initialization, status, data modification, pointer manipulation, search, and input/output. The initialization group handles the creation, initializing, and destruction of the link list. The status group functions return various types of information about the link list. The pointer manipulation group allows the arbitrary repositioning of the current-pointer. The data modification group adds and deletes nodes. The search group returns node information based on key data or on absolute node position. The input/output group saves or retrieves node data to or from a disk file.

Type Definitions and Declarations:

The header file should be included with your other includes. There are two header files, one to be used for production and the other for debugging.

`#include <linklist.h>` (Production header)

`#include <dll_dbg.h>` (Debug header)

In the header files you will find:

```
typedef void Info;.
```

This *type definition* is the dummy definition for the data that will be inserted in the link list. Since it is defined as type *void* your data can take on any form; however, I recommend that the data be defined as a *typedef* as in the example below. In the remainder of this document I refer to this definition as type *Info*.

```
typedef struct your_info
   {
   char info_element[SIZE]    /* Your data */
   } YourInfo;
```

The only other thing that needs to be done is the *Info* declaration, of which I generally use three, one each for getting, putting, and finding data. These can be declared globally or as needed.

There are four typedef enumerations that you will need to be aware of: *DLL_Return*, *DLL_Boolean*, *DLL_SrchOrigin*, and *DLL_SrchDir*.

DLL_Return defines all the return values used in the API.

```
typedef enum
   {
   DLL_NORMAL,           /* normal operation */
   DLL_MEM_ERROR,        /* malloc error */
   DLL_ZERO_INFO,        /* sizeof(Info) is zero */
   DLL_NULL_LIST,        /* List is NULL */
   DLL_NOT_FOUND,        /* Record not found */
   DLL_OPEN_ERROR,       /* Cannot open file */
   DLL_WRITE_ERROR,      /* File write error */
   DLL_READ_ERROR,       /* File read error */
   DLL_NOT_MODIFIED,     /* Unmodified list */
   DLL_NULL_FUNCTION     /* NULL function pointer */
   } DLL_Return;
```

Since booleans are rather bothersome in C I have defined them with a DLL_ prefix so as not to collide with other definitions. You could do something like this:

```
#define bool   DLL_Boolean
#define FALSE  DLL_FALSE
#define TRUE   DLL_TRUE
```

or just use my definitions.

```
typedef enum
   {
   DLL_FALSE,
   DLL_TRUE
   } DLL_Boolean;
```

DLL_SrchOrigin and *DLL_SrchDir* are definitions used in the *DLL_SetSearchModes()* function described below in the search group. The state table defaults at initialization to DLL_HEAD and DLL_DOWN. The two default enumerations use the current setting in the state table.

```
typedef enum
   {
   DLL_ORIGIN_DEFAULT,      /* Use current origin setting */
   DLL_HEAD,                /* Set origin to head pointer */
   DLL_CURRENT,             /* Set origin to current pointer */
   DLL_TAIL                 /* Set origin to tail pointer */
   } DLL_SrchOrigin;

typedef enum
   {
   DLL_DIRECTION_DEFAULT,   /* Use current direction setting */
   DLL_DOWN,                /* Set direction to down */
   DLL_UP                   /* Set direction to up */
   } DLL_SrchDir;
```

Initialization Functions:

Since one of the goals was to have the API be reenterant, a function was needed that could create as many incidents of a doubly linked list as would be needed in a single application. However, we first need to define a pointer to our list.

```
List *your_name = NULL;
```

This pointer is then dereferenced again and passed as an argument to *DLL_CreateList().* Its return value is also the address placed in name, but can be tested for a NULL failure condition.

```
List *DLL_CreateList(List **name);
```

After this call the top level structure needs to be initialized by calling *DLL_InitializeList()*. The first argument, *list* is the pointer to the linked list and *infosize* is the size of the previously defined *Info* structure holding the data. This function returns DLL_NORMAL, DLL_ZERO_INFO, or DLL_NULL_LIST.

```
DLL_Return DLL_InitializeList(List *list, size_t infosize);
```

The above two functions are all that is needed to initialize the API; however, you will need to deinitialize the API at some point so there is one last function in this group.

```
void DLL_DestroyList(List **name);
```

Notice also the pointer to pointer notation. This function writes a NULL to the pointer before it exits, so you can test to see if the linked list actually exists.

Status Functions:

It is often necessary to test for the existence of nodes in a list. The argument *list* in *DLL_IsListEmpty()* is the pointer to the linked list. The return value is TRUE (empty list) or FALSE (non-empty list).

```
DLL_Boolean DLL_IsListEmpty(List *list);
```

To test if there are sufficient memory resources for an additional node we use the following function. The argument *list* is the pointer to the linked list. The return value is TRUE (not enough memory) or FALSE (enough memory).

```
DLL_Boolean DLL_IsListFull(List *list);
```

DLL_IsListEmpty() makes a test on pointers to determine if there are any nodes in the list; the following function looks at an element in the *List* structure to determine how many nodes exist. Both functions can be used to test if any nodes exist, but *DLL_IsListEmpty()* is a more reliable test of the integrity of the linked list.

```
unsigned long DLL_GetNumberOfRecords(List *list);
```

Data Modification:

To add new nodes to the list in sorted or unsorted order use the following function. The first argument, *list*, is the pointer to the linked list; *info* points to an *Info* structure (your data), *pFun* points to a function that does the element compare for the insertion sort. If a NULL pointer is passed instead of the function pointer *DLL_AddRecord()* will add records to the end of the list. The return value is DLL_NORMAL or DLL_MEM_ERROR.

```
DLL_Return DLL_AddRecord(List *list, Info *info,
                         int (*pFun)(Info *, Info *));
```

The function which compares elements can be as simple or as complex as the situation requires. It is passed two arguments: the first *Info* structure is the data from the list, the second comes from the second argument passed to *DLL_AddRecord()*. The return value is the same as strcmp():

Less than zero	arg1 < arg2
Zero	arg1 == arg2
Greater than Zero	arg1 > arg2

Example:

```
int compare(Info *record, Info *compare)
   {
   return(strcmp(record->info_element, compare->info_element));
   }
```

To update a record use *DLL_UpdateCurrentRecord()*. The first argument, *list*, is the pointer to the linked list, *record* points to an *Info* structure (your data). The return value is DLL_NORMAL or DLL_NULL_LIST.

```
DLL_Return DLL_UpdateCurrentRecord(List *list, Info *record);
```

The following function deletes records. It is passed a single argument named *list* which points to the linked list. The return value is DLL_NORMAL or DLL_NULL_LIST.

```
DLL_Return DLL_DeleteCurrentRecord(List *list);
```

The following function deletes the entire list but does not destroy the top level type *List* structure; there is no need to call *DLL_InitializeList()* again, the head, tail, current, and saved pointers are set to NULL, the listsize pointer is set to zero, and the boolean modified is set to TRUE. It is passed a single argument named *list* which points to the linked list. The return value is DLL_NORMAL or DLL_NULL_LIST.

```
DLL_Return DLL_DeleteEntireList(List *list);
```

Pointer Manipulation:

All the pointer manipulation functions affect only the current pointer. The first two of these functions moves the current pointer to the head or the tail of the list. The argument, *list* is the pointer to the linked list. The return value is DLL_NORMAL or DLL_NULL_LIST.

```
DLL_Return DLL_CurrentPointerToHead(List *list);

DLL_Return DLL_CurrentPointerToTail(List *list);
```

To increment or decrement the current pointer use the following functions. The argument *list* is the pointer to the linked list. The return value is DLL_NORMAL, DLL_NULL_LIST, or DLL_NOT_FOUND.

```
DLL_Return DLL_IncrementCurrentPointer(List *list);

DLL_Return DLL_DecrementCurrentPointer(List *list);
```

I often found it necessary to be able to store the *current pointer* and then to restore it later. The two following function fill this need. The argument *list* is the pointer to the linked list. The return value is DLL_NORMAL, or DLL_NOT_FOUND.

```
DLL_Return DLL_StoreCurrentPointer(List *list);

DLL_Return DLL_RestoreCurrentPointer(List *list);
```

Search:

To find a record based on a set of search criteria use the following function. The first argument, *list*, points to the link list; the second argument, *record*, points to an *Info* data structure which will hold the first record found; the third argument, *match*, points to an *Info* data structure which has been previously filled with the search criteria; and the forth argument, *pFun*, points to a function that does the element compare for finding the record. See *DLL_AddRecord()* for a description on how to write the function pointed to by *pFun*. The return value is DLL_NORMAL, DLL_NULL_LIST, DLL_NOT_FOUND, or DLL_NULL_FUNCTION.

```
DLL_Return DLL_FindRecord(List *list, Info *record, Info *match,
                          int (*pFun)(Info *, Info *));
```

NOTE: There are no FindNext or FindPrior functions; however, they are on my to-do list.

The *DLL_FindRecord()* function uses a predefined state table which can be altered with the next function. The state table defaults at initialization are DLL_HEAD and DLL_DOWN. The return value is DLL_NORMAL or DLL_NOT_MODIFIED.

```
DLL_Return DLL_SetSearchModes(List *list, DLL_SrchOrigin origin,
                              DLL_SrchDir dir);
```

This state table takes the following form:

```
typedef struct search_modes
   {
   DLL_SrchOrigin search_origin;
   DLL_SrchDir    search_dir;
   } DLL_SearchModes;
```

To determine how the current state table is set use the next function which returns a pointer to type *DLL_SearchModes*.

```
DLL_SearchModes *DLL_GetSearchModes(List *list);
```

NOTE: The above structure is a copy of the actual state table, so saving its pointer and setting the variables directly will have no effect.

The next three functions will get a record based on the position of the current pointer. They are passed two arguments, the first is *list* which points to the linked list and the second is *record* which points to an *Info* data structure which will hold the found record. The return values for *DLL_GetCurrentRecord()* are DLL_NORMAL or DLL_NULL_LIST and the return values for *DLL_GetPriorRecord()* and *DLL_GetNextRecord()* are DLL_NORMAL, DLL_NULL_LIST, or DLL_NOT_FOUND.

```
DLL_Return DLL_GetCurrentRecord(List *list, Info *record);

DLL_Return DLL_GetPriorRecord(List *list, Info *record);

DLL_Return DLL_GetNextRecord(List *list, Info *record);
```

NOTE: *DLL_GetPriorRecord()* and *DLL_GetNextRecord()* retrieve the record linked to the current record not a record based on any search criteria.

Input/Output:

Input and output functions would not generally be considered to be part of a linked list API; however, without these functions one would need to set the current pointer to the head or tail of the list and then make repeated calls to some of the DLL_Get functions. This of course would be time consuming both for the programmer and for the routine, so I have decided to include two functions for saving and loading a file from disk.

When saving a list, pass to *DLL_SaveList()* the arguments *list* which points to the linked list and then *path*, which points to a string containing the filename and it's path. The return value is DLL_NORMAL, DLL_NULL_LIST, DLL_OPEN_ERROR, DLL_WRITE_ERROR, or DLL_NOT_MODIFIED. If a list has not been modified it will not be saved and *DLL_SaveList()* will return DLL_NOT_MODIFIED.

```
DLL_Return DLL_SaveList(List *list, const char *path);
```

To load a list from disk, pass to *DLL_LoadList()* the arguments: *list*, which points to the linked list, *path*, which points to a string containing the filename and it's path, and *pFun*, which points to a function that does the element comparison for an insertion sort of the incoming file. Sorting a file will set the modified pointer to TRUE. See *DLL_AddRecord()* for a description on how to write the function pointed to by *pFun*. The return value is DLL_NORMAL, DLL_MEM_ERROR, DLL_OPEN_ERROR, or DLL_READ_ERROR. If a NULL pointer is passed instead of the function pointer *DLL_LoadList()* will add records to the end of the list in the same order that they were previously saved and the modified pointer will be set to FALSE.

```
DLL_Return DLL_LoadList(List *list, const char *path,
                        int (*pFun)(Info *, Info *));
```

Methodology:

In order to make the linked list reenterant a top level *List* structure is created which holds all the necessary information about each list created. This structure is hidden from the application developer along with the *Node* structure which contains three pointers to the user created *Info* structure. The *List* structure contains nine elements: *head*, *tail*, *current*, and *saved* defined as *Node* pointer types; *infosize* defined as a size_t; *listsize* defined as an unsigned long; *modified* defined as a boolean; and *search_origin* and *search_dir* . defined as enumerations for the state table. The *Node* structure contains three elements: *info* defined as an *Info* type; *next* and *prior* defined as *Node* pointers.