07a27f28-0

John Brooks

COLLABORATORS			
	TITLE : 07a27f28-0		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	John Brooks	August 9, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

07a27f28-0

1.1 Linked List Library

```
                    Linked List Library v1.1

~~INTRODUCTION~           --- what this is all about
~~LEGAL~STUFF~~           --- permissions and copyright info

~~LINKLIST.H~~~           --- the header file
~~LINKLIST.LIB~           --- the library file

~~HISTORY~~~~~~           --- what was I thinking??
```

1.2 INTRODUCTION

Well, I finally got tired of re-writing code for linked lists. It would~usually take me a couple of hours cause I had to figure out how to do it all~over again. I took about a week, only a couple of hours a day, and came up~with this library.

A few facts about this thing. There is code written for three basic types of~linked lists, single link, double link, and circular link. There are several~duplicate functions between the types. None of the functions are dependant~upon other functions in the library. Yes, this results in some duplicate code~but the size is about 5k so who really cares. I stayed away from recursion.~This seems to make it just a bit faster.

The library was developed on an Amiga 3000 with the SAS C compiler version 6.3. The code was compiled and tested with the Memlib libray to ensure there were no memory errors or leaks from it.

To help out a bit, there are examples of how to use the library calls in this~document. Enjoy. I hope you like it.

If you have any comments, questions, suggestions, or wish to send gifts or~donations I can be reached at

John Brooks

461 3rd St
Satellite Bch, FL 32937

1.3 LEGAL STUFF

Permission is granted to develop and release software products both public and~commercial with this library. There are no restrictions on the use of it.

Permission is not granted to dissasemble or modify the library or~documentation without permission from the author.

1.4 LINKLIST.H

LINKLIST.H The library header file.

This file allows access into the linklist library. When looking at this file it is not that easy to tell what is going on. Hopefully this will help.

~STRUCTURES~ --- the pre-defined C structures that are used with the library

~FUNCTIONS~~ --- a listing of all the functions in the library.

1.5 STRUCTURES

Linked List Library Structures

Single Link List

```
typedef struct _single {
  struct _single        *next;
  void                  *data;
} Single;
```

Double Link List

```
typedef struct _double {
  struct _double        *next;
  struct _double        *prev;
  void                  *data;
} Double;
```

```
Circle Link List

        typedef struct _circle {
          struct _circle        *next;
          void                  *data;
        } Circle;
```

1.6 FUNCTIONS

```
FUNCTIONS

standard list functions

void    *stdGetNewLink (int)

single list functions

Single  *singleGetNewLink (void)
Single  *singleAttachBegin (Single *, Single *)
Single  *singleAttachEnd (Single *, Single *)
Single  *singleInsertLink (Single *, Single *)
Single  *singleDeleteLink (Single *, Single *)
Single  *singleSearch (void *, Single *)
Single  *singleFindEnd (Single *)
void     singleDestroyList (Single *)

double list functions

Double  *doubleGetNewLink (void)
Double  *doubleAttachBegin (Double *, Double *)
Double  *doubleAttachEnd (Double *, Double *)
Double  *doubleInsertLink (Double *, Double *)
Double  *doubleDeleteLink (Double *, Double *)
Double  *doubleFindBegin (Double *)
Double  *doubleFindEnd (Double *)
Double  *doubleSearch (void *, Double *)
void     doubleDestroyList (Double *)

circular list functions

Circle  *circleGetNewLink (void)
Circle  *circleStartList (Circle *)
Circle  *circleAttachEnd (Circle *, Circle *)
Circle  *circleInsertLink (Circle *, Circle *)
Circle  *circleDeleteLink (Circle *, Circle *)
Circle  *circleSearch (void *, Circle *)
void     circleDestroyList (Circle *)
```

1.7 LINKLIST.LIB

Library Functions

This section will explain in detail each of the functions contained in the library. There will be an example of how to use the code as part of the description.

```
Standard~List~Functions
Single~List~Functions~~
Double~List~Functions~~
Circular~List~Functions
```

1.8 STANDARD FUNCTIONS

Standard Functions

These functions can be used with any linked list. Since they are pretty generic, typedef'ing will more than likely be used to keep the compiler quiet.

```
stdGetNewLink
```

1.9 stdGetNewLink ()

stdGetNewLink

```
void *stdGetNewLink (size) int        size;
```

This function returns a pointer to the amount of memory allocated in the argument size. The pointer will probably have to be typedef'ed into a useful type. This function's primary use is to get memory for any user created structures.

example:

```
#include "linklist.h"

main ()
{
  Single  *link;
  link = (Single *)stdGetNewLink (sizeof (Single));
  ...
}
```

error: returns NULL if the memory could not be allocated.

1.10 Single List Functions

Single List Functions

The following functions are used with the single linked list. If a different structure is used with these functions, problems can occur with the memory bounds of pointers. To ensure this doesn't happen the structures used with the single functions should be the same size as the defined single structure.

singleGetNewLink~
singleAttachBegin
singleAttachEnd~~
singleInsertLink~
singleDeleteLink~
singleSearch~~~~~
singleFindEnd~~~~
singleDestroyList

1.11 singleGetNewLink ()

singleGetNewLink

Single *singleGetNewLink (void)

This function returns a pointer of type single to the allocated memory. Typedef the function if a different pointer type is being used.

example:

```
#include "linklist.h"

main () {

   Single *link;
   link = singleGetNewLink ();
   ...
}
```

error: returns a NULL if the memory could not be allocated.

1.12 singleAttachBegin ()

singleAttachBegin

```
Single *singleAttachBegin (list, link)
Single *list;
Single *link;
```

Attaches the new link to the start of the list and returns the pointer to the start of the new list.

example:

```
#include "linklist.h"
```

```
main ()
{

  Single    *list, link;
  ...

  link = singleGetNewLink ();
  list = singleAttachBegin (list, link);
  ...
}
```

NOTE: The first argument to this function must be the start of the list or a serious leak can and will occur.

error: returns a NULL if either of the arguments is an invalid pointer.

1.13 singleAttachEnd ()

singleAttachEnd

```
Single *singleAttachEnd (list, link)
Single *list;
Single *link;
```

The function returns the pointer to the same list that was passed in as the first argument. The pointer to the link will be attached to the end of the list.

example:

```
#include "linklist.h"

main () {

  Single *list, *link;
  ...

  link = singleGetNewLink ();
  list = singleAttachEnd (list, link);
  ...
}
```

The list variable and argument doesn't have to be the start of the list. It is a little quicker if the list argument is somewhere deeper into the list. The big problem is when a pointer to the start of the list is not maintained a memory leak occurs and also loss of data.

error: returns NULL if either of the arguments is an invalid pointer

1.14 singleInsertLink ()

```
singleInsertLink

Single *singleInsertLink (before, new)
Single *before;
Single *new;

This function is used when a link needs to be inserted somewhere inside  the
list.  To speed up the operation it takes the link that is before the
new link as the first argument so the whole link doesn't have to be
parsed.  This function can be used to attach a link to the end of the
list. The pointer to the first argument is returned.

example:

#include "linklist.h"

main() {
  Single *list, *new_link;
  ...

  list = singleInsertLink (list, new_link);
  ...
}

error: returns NULL if either of the pointers are invalid.
```

1.15 singleDeleteLink ()

```
singleDeleteLink

Single *singleDeleteLink (list, link)
Single *list;
Single *link;

This function is used to remove a link from the list.  Since it is technically
a memory bug to free memory that is not allocated (Memlib complains) the
function makes sure the link is in the list  before freeing and setting the
pointers of the other links.  If the  link is not part of the list it must be
freed by the programmer or attached to the link before it can be freed with
this call.  The arguments are the pointer to the list and the link being
removed.  The pointer to the new list is returned.

example:

#include "linklist.h"

main () {
  Single *list, *link;
  ...

  list = singleDeleteLink (list, link);
  ...
}
```

As with singleAttachEnd the argument list doesn't have to be the start of the list to help speed things up, but the same memory leak and data loss potential is there.

error: returns a NULL if either of the arguments is an invalid pointer.

1.16 singleSearch ()

singleSearch

```
Single *singleSearch (data, list)
void   *data;
Single *list;
```

This function may be a little tricky and rarely used. It requires some pointer that will be compared to the Single->data part of the structure. The other argument is the list to be searched. It returns the link if found or NULL if not.

example:

```
#include "linklist.h"

main() {
  Single *list, *link;
  char   *data = "data";
  ...

  link->data = data;
  ...

  if ( !(singleSearch (data, list)))
    printf ("Data not found\n");

  ...
}
```

error: returns a NULL if the second argument is an invalid pointer.

1.17 singleFindEnd ()

singleFindEnd

```
Single *singleFindEnd (list)
Single *list;
```

This function is pretty much self explanatory. Its only argument is the list and it returns a pointer to the last link of that list.

example:

```
#include "linklist.h"
```

```
main () {
  Single *list;
  ...

  /* remove the last link of the list */
  list = singleDeleteLink (list, singleFindEnd (list));
  ...
}
```

error: NULL is returned if the argument is invalid.

1.18 singleDestroyList ()

singleDestroyList

```
void singleDestroyList (list)
Single *list;
```

The use of this function is pretty much straight forward - destroy the whole list. The function parses through the list to find the end and then one by one frees the memory to each link.

NOTE: This function does not free the memory pointed to by the Single->data pointer. It would be simple to do this but if it was a pointer to another structure that had memory pointer in it, there would be serious memory leaks. Any memory that has been allocated inside this pointer must be freed before calling this function.

example:

```
#include "linklist.h"

main()

{
  Single *list;
  ...

  singleDestroyList (list);
  exit (0);
}
```

error: no error condition is returned. If the pointer is invalid the function returns without releasing the memory.

1.19 DOUBLE LIST FUNCTIONS

Double List Functions

These following functions are for double linked lists. Again, the predefined structures don't have to be used but make sure that any user defined strucures

have the same basic format to prevent any weird memory and/or pointer problems.

```
        doubleGetNewLink~
        doubleAttachBegin
        doubleAttachEnd~~
        doubleInsertLink~
        doubleDeleteLink~
        doubleFindBegin
        doubleFindEnd~~~~
        doubleSearch
        doubleDestroyList
```

1.20 doubleGetNewLink ()

```
doubleGetNewLink

Double *doubleGetNewLink (void)

Returns a pointer to the new link of type Double.

example:

#include "linklist.h"

main ()
{
        Double *link;
        link = doubleGetNewLink ();
}

error: returns a NULL if the memory could no be allocated.
```

1.21 doubleAttachEnd ()

```
doubleAttachBegin

Double *doubleAttachBegin (list, link)
Double *list;
Double *link;

This function parses through the list and attaches the link to the start.  The ↩
   pointer
to the new list is returned, not necessarily the start of the list.

example:

#include "linklist.h"

main ()
{
        Double *list;
        ...
```

```
        list = doubleAttachBegin (list, doubleGetNewLink());
        ...
}
```

As with the attach function with the single lists, the first argument can be any link on the list. The only difference here is that a pointer to the head doesn't have to be kept since the start can be found.

error: returns a NULL if either of the pointers are invalid.

1.22 doubleAttachEnd ()

doubleAttachEnd

```
Double *doubleAttachEnd (list, link)
Double *list;
Double *link;
```

This function is identical to doubleAttachBegin except that it puts the link at the end of the list.

example:

```
#include "linklist.h"

main()
{
        Double *list;
        ...

        list = doubleAttachEnd (list, doubleGetNewLink());
        ...
}
```

error: if either of the pointers are invalid a NULL is returned.

1.23 doubleInsertLink ()

doubleInsertLink

```
Double *doubleInsertLink (before, new)
Double *before;
Double *new;
```

This function is almost identical to singleInsertLink. It is used to insert a link somewhere in the list.

example:

```
#include "linklist.h"

main ()
{
        Double *list, *link2;
        ...

        list = doubleAttachEnd (list, doubleGetNewLink());
        ...

        link2 = doubleGetNewLink();
        list = doubleInsertLink (list, link2);
        ...
}
```

error: returns a NULL if either of the arguments are invalid pointers.

1.24 doubleDeleteLink ()

doubleDeleteLink

```
Double *doubleDeleteLink (list, link)
Double *list;
Double *link;
```

This function is used to remove a link from the list. The function will not free any memory allocated by the Double->data pointer. This memory, and any memory further in, must be freed before the link is destroyed in order to prevent memory leaks. The pointer to the new list is returned.

example:

```
#include "linklist.h"

main ()
{
        Double *list, *link;
        ...

        list = doubleDeleteLink (list, link);
        ...
}
```

error: if the pointers are invalid, a NULL is returned.

1.25 doubleFindBegin ()

doubleFindBegin

```
Double *doubleFindBegin (list)
Double *list;
```

```
This function accepts the list as its only argumnt and then parses back through
it returning the pointer to the first link in the list.

example:

#include "linklist.h"

main ()
{
        Double *list, *link;
        ...

        list = doubleAttachBegin (list, link);
        /* make sure that list is pointing to the head */
        list = doubleFindBegin (list);
        ...
}

error: a NULL is returned if the pointer is invalid.
```

1.26 doubleFindEnd ()

```
doubleFindEnd

Double *doubleFindEnd (list)
Double *list;

This function is identical to the doubleFindBegin except that it finds the end of
the list instead of the start.

example:

#include "linklist.h"

main ()
{
        Double *list;
        ...

        /* remove the last link in the list */
        list = doubleDeleteLink (list, doubleFindEnd (list));
        ...
}

error: if the argument is an invalid pointer, NULL is returned.
```

1.27 doubleSearch ()

```
@[fg shine}doubleSearch

Double *doubleSearch (data, list)
void   *data;
```

```
Double *list;

The first argument to this function is a pointer to the data that will be
searched for.  The function parses through the list, the second argument,
comparing the pointer to the Double->data pointer in the structure.  If
found the pointer to the link is returned otherwise NULL is.

example:

#include "linklist.h"

main ()
{
        Double *list, *link;
        char   *data = "data";
        ...

        link->data = data;
        ...

        if ( !(doubleSearch (data, list)))
                printf ("Data not found\n");
        ...
}

error: a NULL is returned if the second argument is an invalid pointer.
```

1.28 doubleDestroyList ()

```
doubleDestroyList

void doubleDestroyList (list)
Double *list;

The use of this function is pretty much straight forward - destroy the whole
list.  The function parses through the list to find the end and then one by
one frees the memory to each link.

NOTE: This function does not free the memory pointed to by the Double->data
pointer.  It would be simple to do this but if it was a pointer to another
structure that had memory pointer in it, there would be serious memory leaks.
Any memory that has been allocated inside this pointer must be freed before
calling this function.

example:

#include "linklist.h"

main ()
{
        Double *list;
        ...

        doubleDestroyList (list);
        exit (0);
```

```
}
```

error: if the pointer passed to the function is invalid it will not execute. No error condition is returned.

1.29 CIRCULAR LIST FUNCTIONS

Circular List Functions

The following functions are to be used with circular linked lists. These functions use links that only point in one direction (the next link) and end up connecting back at the start. Because of the sometimes unique behavior of circular links, some of the functions here may not behave in the expected way, ie. a pointer to a link other than expected. These may appear as bugs but there should be no great concern as the functions return the most reliable pointers and since the list is circular there is no loss of data through these functions.

```
circleGetNewLink~
circleStartList
circleAttachEnd
circleInsertLink
circleDeleteLink
circleSearch
circleDestroyList
```

1.30 circleGetNewLink ()

circleGetNewLink

```
Circle *circleGetNewLink (void)
```

The function returns a pointer to the new link.

example:

```
#include "linklist.h"

main ()
{
        Circle *link;
        link = circleGetNewLink ();
}
```

error: a NULL is returned if the memory couldn't be allocated.

1.31 circleStartList ()

```
circleStartList

Circle *circleStartList (list)
Circle *list;

This is a very important function.  This function starts the circular
pointer.  It returns the pointer to the start of the list.

example:

#include "linklist.h"

main ()
{
        Circle *list;

        list = circleGetNewLink ();
        list = circleStartList (list);
        ...
}

error: if the pointer is invalid, a NULL is returned.
```

1.32 circleAttachEnd ()

```
circleAttachEnd

Circle *circleAttachEnd (list, link)
Circle *list;
Circle *link;

This function attaches the link to the 'end' of the list.  There really is
no end in a circular list.  The function pareses through the list until it
finds the start and then it attaches the link just before the start and
points the link to the start completing the circle again.

example:

#include "linklist.h"

main ()
{
        Circle *list;
        ...

        list = circleAttachEnd (list, circleGetNewLink());
        ...
}

error: if either of the pointers are invalid, a NULL is returned.
```

1.33 circleInsertLink ()

```
circleInsertLink

Circle *circleInsertLink (before, new)
Circle *before;
Circle *new;

This function is used to insert a link somewhere in the list.  To speed up
the process the first argument is the link that will be before
the new link.  This function can be used to attach links to the
preverbial end of the list.

example:

#include "linklist.h"

main ()
{
        Circle *list, *link;

        list = circleStartList (circleGetNewLink());
        link = circleGetNewLink ();
        /* attach link to the end of the list */
        list = circleInsertList (list, link);
        /* now there are two links in the list */
        ...
}

error: if the arguements are invalid pointers, a NULL is returned.
```

1.34 circleDeleteLink ()

```
circleDeleteLink

Circle *circleDeleteLink (list, link)
Circle *list;
Circle *link;

This is one of the functions that may not return the expected pointer into
the list.  If the pointer being passed in as the list is the one destroyed
then the pointer becomes unstable.  To avoid loss of data the link that
points to this link is the one that is passed back.  This function does not
free any memory that is allocated deeper in the Circle->data pointer
This memory must be freed before the link is destroyed.

example:

#include "linklist.h"

main ()
{
        Circle *list, *link;
        ...
```

```
        list = circleDeleteLink (list, link);
        ...
}
```

error: a NULL is returned if either of the pointers are invalid.

1.35 circleSearch

```
circleSearch

Circle *circleSearch (data, list)
void   *data;
Circle *list;
```

As with the other search functions this one also parses through the list looking at the Circle->data pointer. If a match is found it returns the pointer to the link and return NULL if one is not.

example:

```
#include "linklist.h"

main ()
{
        Circle *list, *link;
        char   *data = "data";
        ...

        link->data = data;
        ...

        if ( (circleSearch (data, list)))
                printf ("Found %s\n", data);
        ...
}
```

error: if the second pointer is invalid, a NULL is returned.

1.36 circleDestroyList ()

```
circleDestroyList

void circleDestroyList (list)
Circle *list;
```

The use of this function is pretty much straight forward - destroy the whole list. The function parses through the list to find the end and then one by one frees the memory to each link.

NOTE: This function does not free the memory pointed to by the Circle->data pointer. It would be simple to do this but if it was a pointer to another

```
structure that had memory pointer in it, there would be serious memory leaks.
Any memory that has been allocated inside this pointer must be freed before
calling this function.

example:

#include "linklist.h"

main ()
{
        Circle *list;
        ...

        circleDestroyList (list);
        exit (0);
}

error: if the pointer is not valid, the function will return without
releasing the memory.  No error condition is returned.
```

1.37 HISTORY

```
Version 1.0
        The initial release of the library.

Version 1.1

Added a new function singleAttachBegin() to the library.

Changed the functions singleInsertLink(), doubleInsertLink(), and
circleInsertLink().

singleInsertLink (before, after, link);                 /*v 1.0*/
singleInsertLink (before, link);      /*v 1.1*/

doubleInsertLink (before, after, link);                 /*v 1.0*/
doubleInsertLink (before, link);      /*v 1.1*/

circleInsertLink (before, after, link);                 /*v 1.0*/
circleInsertLink (before, link);      /*v 1.1*/

The function circleStartList(), which creates a one link circular link to
initialize the list, is no longer needed.  Now circleGetNewLink()
correctly initializes the new list by doing this function. It is still
left in the library for compatability to version 1.0 and to initialize
any user created list structures.

All of the functions that take link pointers as arguments now have
error checking in them.  If an invalid pointer is passed into the
function a NULL pointer is returned.  This keeps the library from
crashing any programs due to SEGV errors.
```