Linked List Library

Programmers Reference

version 1.1

Preface

Well, I finally got tired of writing programs with linked lists in them and forgetting how to do it. I would sit there for about an hour trying to dig up old source code or remembering how to do it. Enough of that. I took about a week and developed a library with some of the basic functions to use with these lists. There are duplicate functions for three types of lists: single (forward only), double (forward and back), and circular (no start no end).

The C structures for these lists are also defined. There is a pointer in them to be used for data. I'm not saying you have to use these structures but it might make programming with this library a bit easier.

This code was developed on an Amiga 3000 with the SAS C compiler version 6.3. It was tested with the Memlib library to ensure there were no memory leaks. That's not to say any code developed with this won't be memory bug free. I'm not taking any responsibility for poor memory handling. If you don't know exactly what a linked list is or how to allocate memory, use it, and then FREE it, please learn before releasing code with this library. Although I tried to make it idiot proof there is only so much one can do. I hope this code is useful.

If you like it and use it a lot I'm always willing to take donations.

Any questions, comments, or suggestions, please feel free to contact me at

John Brooks
461 3rd Ave
Satellite Bch, FL 32937

1. Header File

As with most C libraries a header file is used to tell the compiler how to access and use the library. No difference here. The header file name is *linklist.h*. It allows access into the library called *linklist.lib*. The header file itself is really not that useful for the programmer when trying to figure out what arguments to pass to functions. About the most one can get out of this header file is the argument types. Defined within *linklist.h* are the structures of the three linked lists and the functions contained in the library. The functions are defined as prototypes.

1.1 Structures

The structures for single, double, and circle linked lists are defined in the header file. They all have a pointer to another link, next and/or previous, and a void pointer that can be used to store other data. Since the pointer for the data is void, typedef'ing will more than likely need to be done if the pointer is used in another function call that requires a specific type.

These structures are there as a convenience. A custom structure may be used but it will have to be typedef'ed to keep the compiler from crying. The pointers to the next and/or previous links will need to be in the same order in the structure as they are in the header file so when they are passed to a function it uses the correct memory location to find the next link. Don't do this and expect programs to act strange and probably crash.

single link struct

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

double link struct

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

circular link struct

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

1.2 Functions

As with the structures, several functions for the three types of linked lists are also defined in the header. There are also functions for all types of linked lists. These functions have the prefix std. The functions for single lists start with single, double prefix with double, and

circles have the prefix circle.

A lot of the functions are similar between the three types. In fact there is a lot of duplicate code within the library. None of the functions are dependant to any other library functions. This was done to keep the speed up. Also, there is no recursion in any of the functions. The library is relatively small so space was not a big concern.

When using these functions it would be a good idea to keep track of the head of the list especially the single lists. Because of their nature, when the head pointer is lost it is impossible to find the start of the list and here comes the memory leak. With the double the start can be found and memory leaks are easier to prevent. When using some of the circular functions, such as delete, the 'head' can no longer be 'trusted' to point to the next link correctly so the most stable link closest to the pointer passed into the function is returned. Since the list is circular this is not a big problem.

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 *)
```

2. Library Functions

This chapter 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. In the next chapter there are example programs which hopefully show more on the use of these functions.

2.1 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

```
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.

2.2 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.

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 memory leak can and will occur.

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

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 first 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.

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.

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 to the system.

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 an invalid pointer.

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 NULL if the memory could not be allocated.

singleInsertLink

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

This function is used when a link needs to be inserted somewhere inside the list. This function can be used to attach a link to the end of the list. The preferred way is to use `singleAttachEnd()`. 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.

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.

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

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

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 the arguments are invalid pointers then a NULL is returned.

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 either of the arguments are invalid pointers, a NULL is returned.

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, the function will not execute. No error condition is returned.

doubleFindBegin

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

This function accepts the list as its only arguement 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: if the pointer passed to the function is not valid, then a NULL is returned.

doubleFindEnd

```
Double *doubleFindEnd (list)
Double *list;
```

This function is identical to the previous 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: a NULL is returned if the argument is an invalid pointer.

doubleGetNewLink

```
Double *doubleGetNewLink (void)
```

This function returns a pointer to the new link.

example:

```
#include "linklist.h"

main ()
{
   Double *link;
   ...

   link = doubleGetNewLink ();
   ...

}
```

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

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. This function can be used to attach a link to the end of the list. The

preferred method is with `doubleAttachEnd()`. Do not use this function to attach a link to the head of the list, use `doubleAttachBegin()` instead.

example:

```
#include "linklist.h"

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

   list = doubleAttachEnd (list, doubleGetNewLink());
   link = list->next;
   ...

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

error: if either of the pointers passed into the function are invalid then a NULL is returned.

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: if the pointer to the list is invalid, a NULL is returned.

2.4 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, i.e. 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.

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

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

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: no error condition is returned. If the pointer passed to the function is invalid, the function returns without releasing the memory back to the system causing a possible memory leak.

circleGetNewLink

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

The function returns a pointer to the new link.

example:

```
#include "linklist.h"

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

error: if the memory could not be allocated then a NULL pointer is returned.

circleInsertLink

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

This function is used to insert a link somewhere in the list. This function can be used to attach links to the proverbial 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: a NULL is returned if either of the arguments passed are invalid pointers.

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 returns 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 pointer passed into the function is invalid then a NULL is returned.

circleStartList

```
Circle *circleStartList (list)
Circle *list;
```

This is a very important function. The first link in the list must be passed into this function before the list can be used at all. 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 argument is an invalid pointer then the function will return NULL.

3. Example Code

3.1 Code example for single linked lists

single_test.c

```
/*============================================================
  == PROGRAM NAME : single_test.c                           ==
  == ------------------------------------------------------- ==
  == DESCRIPTION : program to test the linklist library     ==
  ============================================================
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "linklist.h"

void print_list (Single *);

main (argc, argv)
int     argc;
char  **argv;

{
  char *numbers[5] = {"one", "two", "three", "four", "five"};
  int  i;

  Single  *list, *link;
  Single  *temp_list;

  /*
   * start the linked list
   */
  list = singleGetNewLink ();
  list->data = numbers[0];

  /*
   * start putting the rest of the links into
   * the list
   */
  for (i = 1; i < 4; i ++) {
    link = singleGetNewLink ();
    link->data = numbers[i];
    list = singleAttachEnd (list, link);

  }

  printf ("Initial list\n");
  print_list (list);

  /*
   * insert a link in the middle
```

```
     */
    link = singleGetNewLink ();
    link->data = numbers[4];

    temp_list = list;
    temp_list = temp_list->next;
    temp_list = singleInsertLink (temp_list, link);

    printf ("List after insert\n");
    print_list (list);

    /*
     * put a link at the start;
     */
    link = singleGetNewLink ();
    link->data = (char *)malloc (strlen("Start of list") +1);
    strcpy (link->data, "Start of list");
    list = singleAttachBegin (list, link);
    print_list (list);

    /*
     * try the search and destroy stuff
     */

    if ((link = singleSearch (numbers[3], list)))
      list = singleDeleteLink (list, link);

    printf ("List after the search and delete\n");
    print_list (list);

    /*
     * get rid of the last link
     */
    list = singleDeleteLink (list, singleFindEnd (list));
    printf ("List after deleting the last link\n");
    print_list (list);

    /*
     * get rid of the list
     */

    singleDestroyList (list);

    return (0);
}

/*
 * print_list ()
 */

void print_list (list)
Single  *list;

{
```

```
  while (list->next != NULL) {
    printf ("data = %s\n", list->data);
    list = list->next;
  }

  /*
   * print the last link
   */
  printf ("data = %s\n", list->data);
}
```

3.2 Code example for double linked lists

double_test.c

```
/*============================================================
  == PROGRAM NAME : double_test.c                           ==
  == ------------------------------------------------------- ==
  == DESCRIPTION :  test the double linked list             ==
  ============================================================
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <exec/memory.h>
#include <proto/exec.h>
#include "linklist.h"

void print_list (Double *);

main (argc, argv)
int    argc;
char **argv;

{

  char  *ones[5] = {"one", "two", "three", "four", "five"};
  char  *tens[5] = {"ten", "twenty", "thirty", "fourty", "fifty"};
  int    i;

  Double  *list, *link;

  /*
   * start the linked list
   */
  list = doubleGetNewLink ();
  list->data = ones[0];

  /*
   * add some more links
   */
```

```
  for (i = 1; i < 5; i ++) {
    link = doubleGetNewLink ();
    link->data = ones[i];
    list = doubleAttachEnd (list, link);
  }

  printf ("List after initial creation\n");
  print_list (list);

  /*
   * put a new link at the start and insert
   * some others in the middle
   */
  link = doubleGetNewLink ();
  link->data = tens[0];
  list = doubleAttachBegin (list, link);

  for (i = 1; i < 5; i ++) {
    link = doubleGetNewLink ();
    link->data = tens[i];
    list = doubleInsertLink (doubleSearch (ones[i-1], list), link);
  }

  printf ("after the inserts\n");
  print_list (list);

  /*
   * get rid of the first and last links
   */
  list = doubleDeleteLink (list, doubleFindBegin (list));
  list = doubleDeleteLink (list, doubleFindEnd (list));

  printf ("After removing the first and last links\n");
  print_list (list);

  /*
   * now that we are done, get rid
   * of the whole list.
   */
  doubleDestroyList (list);

}

/*
 * print_list ()
 */

void print_list (list)
Double  *list;

{

  /*
   * find the start of the list
```

```
     */

    while (list->prev != NULL)
      list = list->prev;

    /*
     * now print it
     */

    while (list->next != NULL) {
      printf ("data = %s\n", list->data);
      list = list->next;
    }

    /*
     * print the last link
     */
    printf ("data = %s\n", list->data);

}
```

3.3 Code example for circular linked lists

circle_test.c

```
/*============================================================
  == PROGRAM NAME : circle_test.c                           ==
  == ------------------------------------------------------ ==
  == DESCRIPTION : test the circular list                   ==
  ============================================================
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <exec/memory.h>
#include <proto/exec.h>
#include "linklist.h"

void print_list (Circle *);

main (argc, argv)
int    argc;
char **argv;

{

  char  *ones[5] = {"one", "two", "three", "four", "five"};
  char  *tens[5] = {"ten", "twenty", "thirty", "fourty", "fifty"};
  int    i;

  Circle  *list, *link;
```

```
/*
 * start the list
 */
list = circleGetNewLink ();
list->data = ones[0];

/*
 * initialize the list
 * doesn't need to be here since 'circleGetNewLink ()'
 * initializes the list but here for user created
 * list structures
 */

list = circleStartList (list);

/*
 * add some more links
 */
for (i = 1; i < 5; i ++) {
  link = circleGetNewLink ();
  link->data = ones[i];
  list = circleAttachEnd (list, link);
}

printf ("After inital list create\n");
print_list (list);

/*
 * insert some new links
 */
for (i = 0; i < 5; i ++) {
  link = circleGetNewLink ();
  link->data = tens[i];
  list = circleInsertLink (circleSearch (ones[i], list), link);
}

printf ("After the insert\n");

/*
 * The following print will look like a bug since the 'data = one'
 * is no longer on the top.  This happened because of the way the
 * insert was performed.  Since it is a circular list the end is
 * attached to the start and there is no loss of data.
 */
print_list (list);

/*
 * delete a link
 */
list = circleDeleteLink (list, circleSearch (ones[3], list));

printf ("After the delete\n");
print_list (list);
```

```
  /*
   * done with the test, get rid of the list
   */
  circleDestroyList (list);

}

/*
 * print_list ()
 */

void print_list (list)
Circle  *list;

{

  Circle  *temp_list;

  /*
   * start printing the list and then loop
   * until the next link in temp_list is
   * the same as the list passed in.
   */

  temp_list = list;

  while (temp_list->next != list) {
    printf ("data = %s\n", temp_list->data);
    temp_list = temp_list->next;
  }

  /*
   * print that last link
   */
  printf ("data = %s\n", temp_list->data);

}
```

Appendix A

Code History and Changes

Version 1.0
Initial release.

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.

Index