Implementing a Generic Linked List in C

PUBLISHED ON MAY 2, 2013

Because I am a totally shameless nerd, I find myself writing applications in C from time to time just to make sure I still can. Aside from iOS development, I rarely have to work with C directly (without the help of a superset like C++ or Objective-C), but every once in a while I like to try and challenge myself to write an application in pure C.

I've found that doing this has led to a much more profound understanding of modern languages and has really opened my eyes to the challenges faced by developers who write their own languages or work with compiler optimization (I know a few…it sounds like tough work!).

I also continue to attend several universities in order to further my education and C seems to be the language of choice when trying to focus on the concepts of data structures and algorithms since much of the problem space is left up to the developer to solve as opposed to our modern languages of choice.

Ok enough babbling! The goal of this article is to describe how we would go about creating a generic implementation of a linked list in pure C. So without further ado, let's take a look at the header file.

List Header File

```
#ifndef __LIST_H
#define __LIST_H

// a common function used to free malloc'd objects
typedef void (*freeFunction)(void *);

typedef enum { FALSE, TRUE } bool;

typedef bool (*listIterator)(void *);

typedef struct _listNode {
  void *data;
  struct _listNode *next;
} listNode;

typedef struct {
  int logicalLength;
  int elementSize;
  listNode *head;
  listNode *tail;
  freeFunction freeFn;
} list;

void list_new(list *list, int elementSize, freeFunction freeFn);
void list_destroy(list *list);

void list_prepend(list *list, void *element);
void list_append(list *list, void *element);
```

```c
int list_size(list *list);

void list_for_each(list *list, listIterator iterator);
void list_head(list *list, void *element, bool removeFromList);
void list_tail(list *list, void *element);

#endif
```

list.h view raw (https://gist.github.com/pseudomuto/6334796/raw/13b9fef5b39b826cf4bb4e563e8755ba08a9bd82/list.h) (https://gist.github.com/pseudomuto/6334796#file-list-h) hosted with ❤ by GitHub (https://github.com)

Line 5 declares a function pointer for a generic `free` function that will be called for each element in the list when it is destroyed. The function must be supplied by the caller when they call `list_new`. Essentially we are saying that the name `freeFunction` will be used to mean "a pointer to a function that returns void and accepts a single void * argument containing the item to be freed." - More on this soon.

Next we define a bool type. C does not have a boolean, but things are truthy or false, like JavaScript or Ruby. We use this to our advantage by creating a bool type and specifying FALSE first. Feel free to write a quick test to show that !FALSE is truthy while !TRUE is falsy. (Slightly more info found here http://stackoverflow.com/questions/1921539/using-boolean-values-in-c (http://stackoverflow.com/questions/1921539/using-boolean-values-in-c)).

Finally we define a `listIterator` type that is a pointer to a method that returns a bool and accepts a `void *`. This function will be called for each element in the list during `list_for_each`.

Structs

The next few lines define a couple of structs we will use for the list.

The first is a typical linked list node that has a void * field for storing whatever the implementer likes and a `next` pointer pointing to the next node in the list. This struct will not be used by the caller, rather it will be used by the internal implementation of the linked list.

The second struct is the actual list and is much more interesting. We start by defining a `logicalLength`, which will be used to keep track of the number of elements in the list, and `elementSize`, which will store the size of each element. The element size is important, since C's generics are limited to void pointers, which means we need the caller to tell us how big each element is in order to `malloc/memcpy` values. This information should always be dynamically supplied. Meaning, we (the caller) should use `sizeof(int)` rather than `4` to allow for $^{32}/_{64}$ bit implementations, etc.

Lastly we store two pointers to listNodes, `head` and `tail` respectively, and a pointer to a function that will be called for each list element when the list is destroyed in order to support lists of complex items (like strings or other structs).

Functions

When I develop C code, I'm always working with the premise that we should not rely on knowing anything about the contents of a struct, rather we should use the supplied functions to gain access to the internal information (i.e. we should use `list_size(list *)` to get the size rather than `list.logicalLength` or `list->logicalLength`).

The next few lines define the function prototypes that will be available for working with linked lists. Here's brief summary of each method:

- `list_new` - initializes a linked list to store elements of `elementSize` and to call `freeFunction` for each element when destroying a list
- `list_destroy` - frees dynamically allocated nodes and optionally calls `freeFunction` with each node's `data` pointer
- `list_prepend` - adds a node to the head of the list
- `list_append` - adds a node to the tail of the list
- `list_size` - returns the number of items in the list
- `list_for_each` - calles the supplied iterator function with the data element of each node (iterates over the list)
- `list_head` - returns the head of the list (optionally removing it at the same time)
- `list_tail` - returns the tail of the list

That pretty much sums up the header file. Now down to brass tacks…the implementation.

List Implementation

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "list.h"

void list_new(list *list, int elementSize, freeFunction freeFn)
{
  assert(elementSize > 0);
  list->logicalLength = 0;
  list->elementSize = elementSize;
  list->head = list->tail = NULL;
  list->freeFn = freeFn;
}

void list_destroy(list *list)
{
  listNode *current;
  while(list->head != NULL) {
    current = list->head;
    list->head = current->next;

    if(list->freeFn) {
      list->freeFn(current->data);
    }

    free(current->data);
    free(current);
  }
}

void list_prepend(list *list, void *element)
{
  listNode *node = malloc(sizeof(listNode));
  node->data = malloc(list->elementSize);
  memcpy(node->data, element, list->elementSize);

```

```c
  node->next = list->head;
  list->head = node;

  // first node?
  if(!list->tail) {
    list->tail = list->head;
  }

  list->logicalLength++;
}

void list_append(list *list, void *element)
{
  listNode *node = malloc(sizeof(listNode));
  node->data = malloc(list->elementSize);
  node->next = NULL;

  memcpy(node->data, element, list->elementSize);

  if(list->logicalLength == 0) {
    list->head = list->tail = node;
  } else {
    list->tail->next = node;
    list->tail = node;
  }

  list->logicalLength++;
}

void list_for_each(list *list, listIterator iterator)
{
  assert(iterator != NULL);

  listNode *node = list->head;
  bool result = TRUE;
  while(node != NULL && result) {
    result = iterator(node->data);
    node = node->next;
  }
}

void list_head(list *list, void *element, bool removeFromList)
{
  assert(list->head != NULL);

  listNode *node = list->head;
  memcpy(element, node->data, list->elementSize);

  if(removeFromList) {
    list->head = node->next;
    list->logicalLength--;

    free(node->data);
    free(node);
  }
}

void list_tail(list *list, void *element)
```

```
{
  assert(list->tail != NULL);
  listNode *node = list->tail;
  memcpy(element, node->data, list->elementSize);
}

int list_size(list *list)
{
  return list->logicalLength;
}
```

list.c **view raw (https://gist.github.com/pseudomuto/6334796/raw/13b9fef5b39b826cf4bb4e563e8755ba08a9bd82/list.c)** **(https://gist.github.com/pseudomuto/6334796#file-list-c)** hosted with ❤ by **GitHub (https://github.com)**

We start by including some system headers and of course `list.h`. I generally try to include as few headers as possible, and only in the implementation files whenever possible. Here are the included headers and a quick description of why we need them.

- `stdlib.h` - in order to use `malloc/free`
- `string.h` - in order to use `memcpy`
- `assert.h` - in order to use the `assert` macro
- `list.h` - in order to have linked list typedefs, structs and function prototypes available to us

list_new

This function takes three arguments: a pointer to the list, the size of the elements being stored, and a function to be called for each element when the list is destroyed.

This function will assert that the `elementSize` supplied is greater than 0. You could work with a list of elements of size <= 0, but not with my implementation! It then sets the default values for `logicalLength`, `head` and `tail`, followed by the `freeFunction` (which can be `NULL` for simple/stack types).

list_destroy

This function takes a single argument; a pointer to the list to be destroyed.

It will free each node's `data` and the node itself. If a *freeFunction* was supplied with the call to `list_new` for this list, it will be called on each node's `data` element before freeing any malloc'd (that's now a word) memory.

list_prepend

This method takes two arguments; a pointer to the list and a void * pointing to the element to be inserted.

We start by creating a new node to store this data and dynamically allocating `list->elementSize` bytes to store the data. We then copy the raw bytes sent in via element to `node->data`.

If the caller passes in an `int` value (and presumably `sizeof(int)` during `list_new`), we copy that integer to `node->data`. If however the caller sends in a pointer to something (like a `char **`) we copy the pointer, not the thing being pointed to. These situations require the caller to pass in a custom *freeFunction* that will be called to free the memory. (The example below shows how to store strings).

Lastly we update `head` to point to the new node, and `tail` and `logicalLength` are updated to reflect the changes.

list_append

This method is almost identical to prepend, except that it puts the new node at the end of the list rather than at the beginning.

list_for_each

This method takes two arguments; a pointer to the list and a `listIterator` function to be called for each node. We then get a hold of `head`, and store a boolean value indicating whether or not to continue.

The iterator function will be passed a pointer to the node's `data` element. I've elected not to copy the pointer here as done in `head/tail`, but to simply pass the original pointer to the iterator function. We stop iterating when either we are out of nodes, or the iterator function returns `FALSE`.

list_head

This method takes three arguments; a pointer to the list, a void * representing a place to populate with the node's `data` and a bool indicating whether or not we should remove the item from the list.

We start by finding the head node (easy enough with `list->head`) and copying `list->elementSize` bytes from the node's `data` into the address specified by *element*.

If the caller supplied `TRUE` for the last argument, we remove the head node from the list, updating `head/tail` values along the way, and free the malloc'd memory for `node->data` and `node`.

list_tail

This method accepts two arguments; a pointer to the list (this should start to seem like Deja Vu by now) and a void * indicating the address to copy the tail node's `data` to.

list_size

The method take a single argument; a pointer to the list. It returns the number of nodes in that list (via `list->logicalLength`).

Finally…How To Use It

```
#include <stdio.h>
#include <string.h>

#include "list.h"

void list_with_ints();
void list_with_strings();

bool iterate_int(void *data);
bool iterate_string(void *data);
void free_string(void *data);

int main(int argc, char *argv[])
{
  printf("Loading int demo...\n");
```

```c
  list_with_ints();
  list_with_strings();
}

void list_with_ints()
{
  int numbers = 10;
  printf("Generating list with the first %d positive numbers...\n", numbers);

  int i;
  list list;
  list_new(&list, sizeof(int), NULL);

  for(i = 1; i <= numbers; i++) {
    list_append(&list, &i);
  }

  list_for_each(&list, iterate_int);

  list_destroy(&list);
  printf("Successfully freed %d numbers...\n", numbers);
}

void list_with_strings()
{
  int numNames = 5;
  const char *names[] = { "David", "Kevin", "Michael", "Craig", "Jimi" };

  int i;
  list list;
  list_new(&list, sizeof(char *), free_string);

  char *name;
  for(i = 0; i < numNames; i++) {
    name = strdup(names[i]);
    list_append(&list, &name);
  }

  list_for_each(&list, iterate_string);

  list_destroy(&list);
  printf("Successfully freed %d strings...\n", numNames);
}

bool iterate_int(void *data)
{
  printf("Found value: %d\n", *(int *)data);
  return TRUE;
}

bool iterate_string(void *data)
{
  printf("Found string value: %s\n", *(char **)data);
  return TRUE;
}

void free_string(void *data)
{
  free(*(char **)data);
}
```

view raw (https://gist.github.com/pseudomuto/6334796/raw/13b9fef5b39b826cf4bb4e563e8755ba08a9bd82/sample_app.c)
sample_app.c (https://gist.github.com/pseudomuto/6334796#file-sample_app-c) hosted with ❤ by GitHub (https://github.com)

TAGS: C (/tags/c), DATA STRUCTURES (/tags/data-structures)

4 Comments

1 Login ▾

G

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

♡ 4 Share

Best Newest Oldest

Jonathan Bakebwa Rivers
6 years ago

Viewing this in 2018. And I cannot say how thankful I am for this. Cheers mate!

1 0 Reply • Share ›

baetek
2 years ago

Hello, great article although i would rather use some kind of for
loop packed macro to traverse every element of the list, for example:

```
#define foreach(list) for(listNode *iterator = list->head; iterator != NULL; iterator =
iterator->next)
```

and usage in code:

```
foreach(list) {
printf("%d -> ", *(int *)iterator->value);
}
```

What do you think about this approach?

0 0 Reply • Share ›

altimadc
4 years ago

So it seems every link list implementation I find does not include the obviously needed function that allows one to loop through the list and delete some nodes (anywhere in the middle or wherever). This is a common need in filtering a link list. Of course I could write it separately but it seems like a basic task that should be included in any LL library. I must be missing something.

0 0 Reply • Share ›

Cleber Machado
4 years ago edited

2019 and I couldn't agree more with Mr. Rivers. You saved my day!
Although I have 2 things to ask.

Cloud you post another example using a structure as data?

Why don't you call the snippet below in the list_head method when the **removeFromList** is true?

```
if(list->freeFn) {
list->freeFn(current->data);
}
```

0 0 Reply • Share ›

Subscribe **Privacy** **Do Not Sell My Data**

< prev | (https://pseudomuto.com/2012/12/application-templates-for-rails/) blog (/blog) | next > (https://pseudomuto.com/2013/06/implementing-a-generic-stack-in-c/)

home (https://pseudomuto.com/)