utlist: linked list macros for C structures
===
Troy D. Hanson <tdh@tkhanson.net>
v2.3.0, February 2021

Here's a link back to the https://github.com/troydhanson/uthash[GitHub project
page].

Introduction

A set of general-purpose 'linked list' macros for C structures are included with
uthash in `utlist.h`. To use these macros in your own C program, just
copy `utlist.h` into your source directory and use it in your programs.

```
  #include "utlist.h"
```

These macros support the basic linked list operations: adding and deleting
elements, sorting them and iterating over them.

Download
~~~~~~~~
To download the `utlist.h` header file,
follow the links on https://github.com/troydhanson/uthash to clone uthash or get
a zip file,
then look in the src/ sub-directory.

BSD licensed
~~~~~~~~~~~~
This software is made available under the
link:license.html[revised BSD license].
It is free and open source.

Platforms
~~~~~~~~~
The 'utlist' macros have been tested on:

 * Linux,
 * Mac OS X, and
 * Windows, using Visual Studio 2008, Visual Studio 2010, or Cygwin/MinGW.

Using utlist

Types of lists
~~~~~~~~~~~~~~
Three types of linked lists are supported:

- *singly-linked* lists,
- *doubly-linked* lists, and
- *circular, doubly-linked* lists

Efficiency
^^^^^^^^^^
Prepending elements::
 Constant-time on all list types.
Appending::
 'O(n)' on singly-linked lists; constant-time on doubly-linked list.
 (The utlist implementation of the doubly-linked list keeps a tail pointer in
 `head->prev` so that append can be done in constant time).
Deleting elements::
 'O(n)' on singly-linked lists; constant-time on doubly-linked list.
Sorting::
 'O(n log(n))' for all list types.
Insertion in order (for sorted lists)::

```
 'O(n)' for all list types.
Iteration, counting and searching::
 'O(n)' for all list types.

List elements
~~~~~~~~~~~~~
You can use any structure with these macros, as long as the structure
contains a `next` pointer. If you want to make a doubly-linked list,
the element also needs to have a `prev` pointer.

  typedef struct element {
      char *name;
      struct element *prev; /* needed for a doubly-linked list only */
      struct element *next; /* needed for singly- or doubly-linked lists */
  } element;

You can name your structure anything. In the example above it is called
`element`.
Within a particular list, all elements must be of the same type.

Flexible prev/next naming
^^^^^^^^^^^^^^^^^^^^^^^^^
You can name your `prev` and `next` pointers something else. If you do, there is
a <<flex_names,family of macros>> that work identically but take these names as
extra arguments.

List head
~~~~~~~~~
The list head is simply a pointer to your element structure. You can name it
anything. *It must be initialized to `NULL`*.

  element *head = NULL;

List operations
~~~~~~~~~~~~~~~
The lists support inserting or deleting elements, sorting the elements and
iterating over them.

[width="100%",cols="10<m,10<m,10<m",grid="cols",options="header"]
|===============================================================================
|Singly-linked             | Doubly-linked              | Circular, doubly-
linked
|LL_PREPEND(head,add);     | DL_PREPEND(head,add);      | CDL_PREPEND(head,add);
|LL_PREPEND_ELEM(head,ref,add); | DL_PREPEND_ELEM(head,ref,add); |
CDL_PREPEND_ELEM(head,ref,add);
|LL_APPEND_ELEM(head,ref,add); | DL_APPEND_ELEM(head,ref,add); |
CDL_APPEND_ELEM(head,ref,add);
|LL_REPLACE_ELEM(head,del,add); | DL_REPLACE_ELEM(head,del,add); |
CDL_REPLACE_ELEM(head,del,add);
|LL_APPEND(head,add);      | DL_APPEND(head,add);       | CDL_APPEND(head,add);
|LL_INSERT_INORDER(head,add,cmp); | DL_INSERT_INORDER(head,add,cmp); |
CDL_INSERT_INORDER(head,add,cmp);
|LL_CONCAT(head1,head2);   | DL_CONCAT(head1,head2);    |
|LL_DELETE(head,del);      | DL_DELETE(head,del);       | CDL_DELETE(head,del);
|LL_SORT(head,cmp);        | DL_SORT(head,cmp);         | CDL_SORT(head,cmp);
|LL_FOREACH(head,elt) {...}| DL_FOREACH(head,elt) {...} | CDL_FOREACH(head,elt)
{...}
|LL_FOREACH_SAFE(head,elt,tmp) {...}| DL_FOREACH_SAFE(head,elt,tmp) {...} |
CDL_FOREACH_SAFE(head,elt,tmp1,tmp2) {...}
|LL_SEARCH_SCALAR(head,elt,mbr,val);| DL_SEARCH_SCALAR(head,elt,mbr,val); |
CDL_SEARCH_SCALAR(head,elt,mbr,val);
|LL_SEARCH(head,elt,like,cmp);| DL_SEARCH(head,elt,like,cmp); |
CDL_SEARCH(head,elt,like,cmp);
|LL_LOWER_BOUND(head,elt,like,cmp); | DL_LOWER_BOUND(head,elt,like,cmp); |
```

CDL_LOWER_BOUND(head,elt,like,cmp);
|LL_COUNT(head,elt,count); | DL_COUNT(head,elt,count); |
CDL_COUNT(head,elt,count);
|===

'Prepend' means to insert an element in front of the existing list head (if
any),
changing the list head to the new element. 'Append' means to add an element at
the
end of the list, so it becomes the new tail element. 'Concatenate' takes two
properly constructed lists and appends the second list to the first. (Visual
Studio 2008 does not support `LL_CONCAT` and `DL_CONCAT`, but VS2010 is ok.)
To prepend before an arbitrary element instead of the list head, use the
`_PREPEND_ELEM` macro family.
To append after an arbitrary element element instead of the list head, use the
`_APPEND_ELEM` macro family.
To 'replace' an arbitrary list element with another element use the
`_REPLACE_ELEM`
family of macros.

The 'sort' operation never moves the elements in memory; rather it only adjusts
the list order by altering the `prev` and `next` pointers in each element. Also
the sort operation can change the list head to point to a new element.

The 'foreach' operation is for easy iteration over the list from the head to the
tail. A usage example is shown below. You can of course just use the `prev` and
`next` pointers directly instead of using the 'foreach' macros.
The 'foreach_safe' operation should be used if you plan to delete any of the
list
elements while iterating.

The 'search' operation is a shortcut for iteration in search of a particular
element. It is not any faster than manually iterating and testing each element.
There are two forms: the "scalar" version searches for an element using a
simple equality test on a given structure member, while the general version
takes an
element to which all others in the list will be compared using a `cmp` function.

The 'lower_bound' operation finds the first element of the list which is no
greater
than the provided `like` element, according to the provided `cmp` function.
The 'lower_bound' operation sets `elt` to a suitable value for passing to
`LL_APPEND_ELEM`; i.e., `elt=NULL` if the proper insertion point is at the front
of
the list, and `elt=p` if the proper insertion point is between `p` and `p-
>next`.

The 'count' operation iterates over the list and increments a supplied counter.

The parameters shown in the table above are explained here:

head::
  The list head (a pointer to your list element structure).
add::
  A pointer to the list element structure you are adding to the list.
del::
  A pointer to the list element structure you are replacing or
  deleting from the list.
elt::
  A pointer that will be assigned to each list element in succession (see
  example) in the case of iteration macros; or, the output pointer from
  the search macros.
ref::
  Reference element for prepend and append operations that will be

```
  prepended before or appended after.
  If `ref` is a pointer with value NULL, the new element will be appended to the
  list for _PREPEND_ELEM() operations and prepended for _APPEND_ELEM()
operations.
  `ref` must be the name of a pointer variable and cannot be literally NULL,
  use _PREPEND() and _APPEND() macro family instead.
like::
  An element pointer, having the same type as `elt`, for which the search macro
  seeks a match (if found, the match is stored in `elt`). A match is determined
  by the given `cmp` function.
cmp::
  pointer to comparison function which accepts two arguments-- these are
  pointers to two element structures to be compared. The comparison function
  must return an `int` that is negative, zero, or positive, which specifies
  whether the first item should sort before, equal to, or after the second item,
  respectively. (In other words, the same convention that is used by `strcmp`).
  Note that under Visual Studio 2008 you may need to declare the two arguments
  as `void *` and then cast them back to their actual types.
tmp::
  A pointer of the same type as `elt`. Used internally. Need not be initialized.
mbr::
  In the scalar search macro, the name of a member within the `elt` structure
which
  will be tested (using `==`) for equality with the value `val`.
val::
  In the scalar search macro, specifies the value of (of structure member
  `field`) of the element being sought.
count::
  integer which will be set to the length of the list

Example
~~~~~~~
This example program reads names from a text file (one name per line), and
appends each name to a doubly-linked list. Then it sorts and prints them.

.A doubly-linked list
--------------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utlist.h"

#define BUFLEN 20

typedef struct el {
    char bname[BUFLEN];
    struct el *next, *prev;
} el;

int namecmp(el *a, el *b) {
    return strcmp(a->bname,b->bname);
}

el *head = NULL; /* important- initialize to NULL! */

int main(int argc, char *argv[]) {
    el *name, *elt, *tmp, etmp;

    char linebuf[BUFLEN];
    int count;
    FILE *file;

    if ( (file = fopen( "test11.dat", "r" )) == NULL ) {
        perror("can't open: ");
```

```
        exit(-1);
    }

    while (fgets(linebuf,BUFLEN,file) != NULL) {
        if ( (name = (el *)malloc(sizeof *name)) == NULL) exit(-1);
        strcpy(name->bname, linebuf);
        DL_APPEND(head, name);
    }
    DL_SORT(head, namecmp);
    DL_FOREACH(head,elt) printf("%s", elt->bname);
    DL_COUNT(head, elt, count);
    printf("%d number of elements in list\n", count);

    memcpy(&etmp.bname, "WES\n", 5);
    DL_SEARCH(head,elt,&etmp,namecmp);
    if (elt) printf("found %s\n", elt->bname);

    /* now delete each element, use the safe iterator */
    DL_FOREACH_SAFE(head,elt,tmp) {
      DL_DELETE(head,elt);
      free(elt);
    }

    fclose(file);

    return 0;
}
--------------------------------------------------------------------------------
```

[[flex_names]]
Other names for prev and next
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If the `prev` and `next` fields are named something else, a separate group of macros must be used. These work the same as the regular macros, but take the field names as extra parameters.

These "flexible field name" macros are shown below. They all end with `2`. Each operates the same as its counterpart without the `2`, but they take the name of the `prev` and `next` fields (as applicable) as trailing arguments.

```
[width="100%",cols="10<m,10<m,10<m",grid="cols",options="header"]
|===============================================================================
|Singly-linked                           | Doubly-linked
| Circular, doubly-linked
|LL_PREPEND2(head,add,next);             | DL_PREPEND2(head,add,prev,next);
| CDL_PREPEND2(head,add,prev,next);
|LL_PREPEND_ELEM2(head,ref,add,next);    |
DL_PREPEND_ELEM2(head,ref,add,prev,next);  |
CDL_PREPEND_ELEM2(head,ref,add,prev,next);
|LL_APPEND_ELEM2(head,ref,add,next);     |
DL_APPEND_ELEM2(head,ref,add,prev,next);   |
CDL_APPEND_ELEM2(head,ref,add,prev,next);
|LL_REPLACE_ELEM2(head,del,add,next);    |
DL_REPLACE_ELEM2(head,del,add,prev,next);  |
CDL_REPLACE_ELEM2(head,del,add,prev,next);
|LL_APPEND2(head,add,next);              | DL_APPEND2(head,add,prev,next);
| CDL_APPEND2(head,add,prev,next);
|LL_INSERT_INORDER2(head,add,cmp,next);  |
DL_INSERT_INORDER2(head,add,cmp,prev,next); |
CDL_INSERT_INORDER2(head,add,cmp,prev,next);
|LL_CONCAT2(head1,head2,next);           | DL_CONCAT2(head1,head2,prev,next);
|
|LL_DELETE2(head,del,next);              | DL_DELETE2(head,del,prev,next);
| CDL_DELETE2(head,del,prev,next);
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
|LL_SORT2(head,cmp,next);                    | DL_SORT2(head,cmp,prev,next);
| CDL_SORT2(head,cmp,prev,next);
|LL_FOREACH2(head,elt,next) {...}            | DL_FOREACH2(head,elt,next) {...}
| CDL_FOREACH2(head,elt,next) {...}
|LL_FOREACH_SAFE2(head,elt,tmp,next) {...} | DL_FOREACH_SAFE2(head,elt,tmp,next)
{...}   | CDL_FOREACH_SAFE2(head,elt,tmp1,tmp2,prev,next) {...}
|LL_SEARCH_SCALAR2(head,elt,mbr,val,next); |
DL_SEARCH_SCALAR2(head,elt,mbr,val,next);   |
CDL_SEARCH_SCALAR2(head,elt,mbr,val,next);
|LL_SEARCH2(head,elt,like,cmp,next);       | DL_SEARCH2(head,elt,like,cmp,next);
| CDL_SEARCH2(head,elt,like,cmp,next);
|LL_LOWER_BOUND2(head,elt,like,cmp,next);  |
DL_LOWER_BOUND2(head,elt,like,cmp,next);    |
CDL_LOWER_BOUND2(head,elt,like,cmp,next);
|LL_COUNT2(head,elt,count,next);           | DL_COUNT2(head,elt,count,next);
| CDL_COUNT2(head,elt,count,next);
|===============================================================================

// vim: set tw=80 wm=2 syntax=asciidoc:
```