```c
/* GLIB - Library of useful routines for C programming
 * Copyright (C) 1995-1997  Peter Mattis, Spencer Kimball and Josh MacDonald
 *
 * SPDX-License-Identifier: LGPL-2.1-or-later
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, see <http://www.gnu.org/licenses/>.
 */

/*
 * Modified by the GLib Team and others 1997-2000.  See the AUTHORS
 * file for a list of people on the GLib Team.  See the ChangeLog
 * files for a list of changes.  These files are distributed with
 * GLib at ftp://ftp.gtk.org/pub/gtk/.
 */

/*
 * MT safe
 */

#include "config.h"
```

```
32
33 #include "glist.h"
34 #include "gslice.h"
35 #include "gmessages.h"
36
37 #include "gtestutils.h"
38
39 /**
40  * GList:
41  * @data: holds the element's data, which can be a pointer
    to any kind
42  *        of data, or any integer value using the
43  *        [Type Conversion Macros][glib-Type-Conversion-
   Macros]
44  * @next: contains the link to the next element in the
   list
45  * @prev: contains the link to the previous element in the
    list
46  *
47  * The #GList struct is used for each element in a doubly-
   linked list.
48  **/
49
50 /**
51  * g_list_previous:
52  * @list: an element in a #GList
53  *
54  * A convenience macro to get the previous element in a #
   GList.
55  * Note that it is considered perfectly acceptable to
   access
56  * @list->prev directly.
57  *
58  * Returns: the previous element, or %NULL if there are no
    previous
59  *          elements
60  **/
61
62 /**
63  * g_list_next:
64  * @list: an element in a #GList
65  *
66  * A convenience macro to get the next element in a #GList
   .
```

```
 67  * Note that it is considered perfectly acceptable to
     access
 68  * @list->next directly.
 69  *
 70  * Returns: the next element, or %NULL if there are no
     more elements
 71  **/
 72
 73 #define _g_list_alloc()         g_slice_new (GList)
 74 #define _g_list_alloc0()        g_slice_new0 (GList)
 75 #define _g_list_free1(list)     g_slice_free (GList, list
     )
 76
 77 /**
 78  * g_list_alloc:
 79  *
 80  * Allocates space for one #GList element. It is called
     by
 81  * g_list_append(), g_list_prepend(), g_list_insert() and
 82  * g_list_insert_sorted() and so is rarely used on its
     own.
 83  *
 84  * Returns: a pointer to the newly-allocated #GList
     element
 85  **/
 86 GList *
 87 g_list_alloc (void)
 88 {
 89   return _g_list_alloc0 ();
 90 }
 91
 92 /**
 93  * g_list_free:
 94  * @list: the first link of a #GList
 95  *
 96  * Frees all of the memory used by a #GList.
 97  * The freed elements are returned to the slice allocator
     .
 98  *
 99  * If list elements contain dynamically-allocated memory
     , you should
100  * either use g_list_free_full() or free them manually
     first.
101  *
```

```
102  * It can be combined with g_steal_pointer() to ensure
    the list head pointer
103  * is not left dangling:
104  * |[<!-- language="C" -->
105  * GList *list_of_borrowed_things = …;  /<!-- -->* (
    transfer container) *<!-- -->/
106  * g_list_free (g_steal_pointer (&list_of_borrowed_things
    ));
107  * ]|
108  */
109 void
110 g_list_free (GList *list)
111 {
112   g_slice_free_chain (GList, list, next);
113 }
114
115 /**
116  * g_list_free_1:
117  * @list: a #GList element
118  *
119  * Frees one #GList element, but does not update links
    from the next and
120  * previous elements in the list, so you should not call
    this function on an
121  * element that is currently part of a list.
122  *
123  * It is usually used after g_list_remove_link().
124  */
125 /**
126  * g_list_free1:
127  *
128  * Another name for g_list_free_1().
129  **/
130 void
131 g_list_free_1 (GList *list)
132 {
133   _g_list_free1 (list);
134 }
135
136 /**
137  * g_list_free_full:
138  * @list: the first link of a #GList
139  * @free_func: the function to be called to free each
    element's data
```

```
140  *
141  * Convenience method, which frees all the memory used by
     a #GList,
142  * and calls @free_func on every element's data.
143  *
144  * @free_func must not modify the list (eg, by removing
    the freed
145  * element from it).
146  *
147  * It can be combined with g_steal_pointer() to ensure
    the list head pointer
148  * is not left dangling — this also has the nice
    property that the head pointer
149  * is cleared before any of the list elements are freed,
    to prevent double frees
150  * from @free_func:
151  * |[<!-- language="C" -->
152  * GList *list_of_owned_things = …;  /<!-- -->* (transfer
     full) (element-type GObject) *<!-- -->/
153  * g_list_free_full (g_steal_pointer (&
    list_of_owned_things), g_object_unref);
154  * ]|
155  *
156  * Since: 2.28
157  */
158 void
159 g_list_free_full (GList          *list,
160                   GDestroyNotify  free_func)
161 {
162   g_list_foreach (list, (GFunc) free_func, NULL);
163   g_list_free (list);
164 }
165
166 /**
167  * g_list_append:
168  * @list: a pointer to a #GList
169  * @data: the data for the new element
170  *
171  * Adds a new element on to the end of the list.
172  *
173  * Note that the return value is the new start of the
    list,
174  * if @list was empty; make sure you store the new value.
175  *
```

```
176  * g_list_append() has to traverse the entire list to
    find the end,
177  * which is inefficient when adding multiple elements. A
    common idiom
178  * to avoid the inefficiency is to use g_list_prepend()
    and reverse
179  * the list with g_list_reverse() when all elements have
    been added.
180  *
181  * |[<!-- language="C" -->
182  * // Notice that these are initialized to the empty list
    .
183  * GList *string_list = NULL, *number_list = NULL;
184  *
185  * // This is a list of strings.
186  * string_list = g_list_append (string_list, "first");
187  * string_list = g_list_append (string_list, "second");
188  *
189  * // This is a list of integers.
190  * number_list = g_list_append (number_list,
    GINT_TO_POINTER (27));
191  * number_list = g_list_append (number_list,
    GINT_TO_POINTER (14));
192  * ]|
193  *
194  * Returns: either @list or the new start of the #GList
    if @list was %NULL
195  */
196 GList *
197 g_list_append (GList    *list,
198                gpointer  data)
199 {
200   GList *new_list;
201   GList *last;
202
203   new_list = _g_list_alloc ();
204   new_list->data = data;
205   new_list->next = NULL;
206
207   if (list)
208     {
209       last = g_list_last (list);
210       /* g_assert (last != NULL); */
211       last->next = new_list;
```

```
212       new_list->prev = last;
213
214       return list;
215     }
216   else
217     {
218       new_list->prev = NULL;
219       return new_list;
220     }
221 }
222
223 /**
224  * g_list_prepend:
225  * @list: a pointer to a #GList, this must point to the
    top of the list
226  * @data: the data for the new element
227  *
228  * Prepends a new element on to the start of the list.
229  *
230  * Note that the return value is the new start of the
    list,
231  * which will have changed, so make sure you store the
    new value.
232  *
233  * |[<!-- language="C" -->
234  * // Notice that it is initialized to the empty list.
235  * GList *list = NULL;
236  *
237  * list = g_list_prepend (list, "last");
238  * list = g_list_prepend (list, "first");
239  * ]|
240  *
241  * Do not use this function to prepend a new element to a
     different
242  * element than the start of the list. Use
    g_list_insert_before() instead.
243  *
244  * Returns: a pointer to the newly prepended element,
    which is the new
245  *     start of the #GList
246  */
247 GList *
248 g_list_prepend (GList    *list,
249                 gpointer  data)
```

```
250 {
251   GList *new_list;
252
253   new_list = _g_list_alloc ();
254   new_list->data = data;
255   new_list->next = list;
256
257   if (list)
258     {
259       new_list->prev = list->prev;
260       if (list->prev)
261         list->prev->next = new_list;
262       list->prev = new_list;
263     }
264   else
265     new_list->prev = NULL;
266
267   return new_list;
268 }
269
270 /**
271  * g_list_insert:
272  * @list: a pointer to a #GList, this must point to the
    top of the list
273  * @data: the data for the new element
274  * @position: the position to insert the element. If this
     is
275  *     negative, or is larger than the number of elements
     in the
276  *     list, the new element is added on to the end of
    the list.
277  *
278  * Inserts a new element into the list at the given
    position.
279  *
280  * Returns: the (possibly changed) start of the #GList
281  */
282 GList *
283 g_list_insert (GList    *list,
284                gpointer  data,
285                gint      position)
286 {
287   GList *new_list;
288   GList *tmp_list;
```

```
289
290   if (position < 0)
291     return g_list_append (list, data);
292   else if (position == 0)
293     return g_list_prepend (list, data);
294
295   tmp_list = g_list_nth (list, position);
296   if (!tmp_list)
297     return g_list_append (list, data);
298
299   new_list = _g_list_alloc ();
300   new_list->data = data;
301   new_list->prev = tmp_list->prev;
302   tmp_list->prev->next = new_list;
303   new_list->next = tmp_list;
304   tmp_list->prev = new_list;
305
306   return list;
307 }
308
309 /**
310  * g_list_insert_before_link:
311  * @list: a pointer to a #GList, this must point to the
    top of the list
312  * @sibling: (nullable): the list element before which
    the new element
313  *     is inserted or %NULL to insert at the end of the
    list
314  * @link_: the list element to be added, which must not
    be part of
315  *     any other list
316  *
317  * Inserts @link_ into the list before the given position
    .
318  *
319  * Returns: the (possibly changed) start of the #GList
320  *
321  * Since: 2.62
322  */
323 GList *
324 g_list_insert_before_link (GList *list,
325                            GList *sibling,
326                            GList *link_)
327 {
```

```
328   g_return_val_if_fail (link_ != NULL, list);
329   g_return_val_if_fail (link_->prev == NULL, list);
330   g_return_val_if_fail (link_->next == NULL, list);
331
332   if (list == NULL)
333     {
334       g_return_val_if_fail (sibling == NULL, list);
335       return link_;
336     }
337   else if (sibling != NULL)
338     {
339       link_->prev = sibling->prev;
340       link_->next = sibling;
341       sibling->prev = link_;
342       if (link_->prev != NULL)
343         {
344           link_->prev->next = link_;
345           return list;
346         }
347       else
348         {
349           g_return_val_if_fail (sibling == list, link_);
350           return link_;
351         }
352     }
353   else
354     {
355       GList *last;
356
357       for (last = list; last->next != NULL; last = last->
    next) {}
358
359       last->next = link_;
360       last->next->prev = last;
361       last->next->next = NULL;
362
363       return list;
364     }
365 }
366
367 /**
368  * g_list_insert_before:
369  * @list: a pointer to a #GList, this must point to the
    top of the list
```

```
370  * @sibling: the list element before which the new
     element
371  *     is inserted or %NULL to insert at the end of the
     list
372  * @data: the data for the new element
373  *
374  * Inserts a new element into the list before the given
     position.
375  *
376  * Returns: the (possibly changed) start of the #GList
377  */
378 GList *
379 g_list_insert_before (GList    *list,
380                       GList    *sibling,
381                       gpointer  data)
382 {
383   if (list == NULL)
384     {
385       list = g_list_alloc ();
386       list->data = data;
387       g_return_val_if_fail (sibling == NULL, list);
388       return list;
389     }
390   else if (sibling != NULL)
391     {
392       GList *node;
393
394       node = _g_list_alloc ();
395       node->data = data;
396       node->prev = sibling->prev;
397       node->next = sibling;
398       sibling->prev = node;
399       if (node->prev != NULL)
400         {
401           node->prev->next = node;
402           return list;
403         }
404       else
405         {
406           g_return_val_if_fail (sibling == list, node);
407           return node;
408         }
409     }
410   else
```

```
411     {
412       GList *last;
413 
414       for (last = list; last->next != NULL; last = last->
    next) {}
415 
416       last->next = _g_list_alloc ();
417       last->next->data = data;
418       last->next->prev = last;
419       last->next->next = NULL;
420 
421       return list;
422     }
423 }
424 
425 /**
426  * g_list_concat:
427  * @list1: a #GList, this must point to the top of the
    list
428  * @list2: the #GList to add to the end of the first #
    GList,
429  *     this must point  to the top of the list
430  *
431  * Adds the second #GList onto the end of the first #
    GList.
432  * Note that the elements of the second #GList are not
    copied.
433  * They are used directly.
434  *
435  * This function is for example used to move an element
    in the list.
436  * The following example moves an element to the top of
    the list:
437  * |[<!-- language="C" -->
438  * list = g_list_remove_link (list, llink);
439  * list = g_list_concat (llink, list);
440  * ]|
441  *
442  * Returns: the start of the new #GList, which equals @
    list1 if not %NULL
443  */
444 GList *
445 g_list_concat (GList *list1,
446                GList *list2)
```

```
447 {
448   GList *tmp_list;
449
450   if (list2)
451     {
452       tmp_list = g_list_last (list1);
453       if (tmp_list)
454         tmp_list->next = list2;
455       else
456         list1 = list2;
457       list2->prev = tmp_list;
458     }
459
460   return list1;
461 }
462
463 static inline GList *
464 _g_list_remove_link (GList *list,
465                      GList *link)
466 {
467   if (link == NULL)
468     return list;
469
470   if (link->prev)
471     {
472       if (link->prev->next == link)
473         link->prev->next = link->next;
474       else
475         g_warning ("corrupted double-linked list detected
    ");
476     }
477   if (link->next)
478     {
479       if (link->next->prev == link)
480         link->next->prev = link->prev;
481       else
482         g_warning ("corrupted double-linked list detected
    ");
483     }
484
485   if (link == list)
486     list = list->next;
487
488   link->next = NULL;
```

```
489   link->prev = NULL;
490
491   return list;
492 }
493
494 /**
495  * g_list_remove:
496  * @list: a #GList, this must point to the top of the
    list
497  * @data: the data of the element to remove
498  *
499  * Removes an element from a #GList.
500  * If two elements contain the same data, only the first
    is removed.
501  * If none of the elements contain the data, the #GList
    is unchanged.
502  *
503  * Returns: the (possibly changed) start of the #GList
504  */
505 GList *
506 g_list_remove (GList         *list,
507                gconstpointer  data)
508 {
509   GList *tmp;
510
511   tmp = list;
512   while (tmp)
513     {
514       if (tmp->data != data)
515         tmp = tmp->next;
516       else
517         {
518           list = _g_list_remove_link (list, tmp);
519           _g_list_free1 (tmp);
520
521           break;
522         }
523     }
524   return list;
525 }
526
527 /**
528  * g_list_remove_all:
529  * @list: a #GList, this must point to the top of the
```

```
529 list
530  * @data: data to remove
531  *
532  * Removes all list nodes with data equal to @data.
533  * Returns the new head of the list. Contrast with
534  * g_list_remove() which removes only the first node
535  * matching the given data.
536  *
537  * Returns: the (possibly changed) start of the #GList
538  */
539 GList *
540 g_list_remove_all (GList         *list,
541                    gconstpointer  data)
542 {
543   GList *tmp = list;
544
545   while (tmp)
546     {
547       if (tmp->data != data)
548         tmp = tmp->next;
549       else
550         {
551           GList *next = tmp->next;
552
553           if (tmp->prev)
554             tmp->prev->next = next;
555           else
556             list = next;
557           if (next)
558             next->prev = tmp->prev;
559
560           _g_list_free1 (tmp);
561           tmp = next;
562         }
563     }
564   return list;
565 }
566
567 /**
568  * g_list_remove_link:
569  * @list: a #GList, this must point to the top of the
    list
570  * @llink: an element in the #GList
571  *
```

```
572  * Removes an element from a #GList, without freeing the
     element.
573  * The removed element's prev and next links are set to %
     NULL, so
574  * that it becomes a self-contained list with one element
     .
575  *
576  * This function is for example used to move an element
     in the list
577  * (see the example for g_list_concat()) or to remove an
     element in
578  * the list before freeing its data:
579  * |[<!-- language="C" -->
580  * list = g_list_remove_link (list, llink);
581  * free_some_data_that_may_access_the_list_again (llink->
     data);
582  * g_list_free (llink);
583  * ]|
584  *
585  * Returns: the (possibly changed) start of the #GList
586  */
587 GList *
588 g_list_remove_link (GList *list,
589                     GList *llink)
590 {
591   return _g_list_remove_link (list, llink);
592 }
593
594 /**
595  * g_list_delete_link:
596  * @list: a #GList, this must point to the top of the
     list
597  * @link_: node to delete from @list
598  *
599  * Removes the node link_ from the list and frees it.
600  * Compare this to g_list_remove_link() which removes the
      node
601  * without freeing it.
602  *
603  * Returns: the (possibly changed) start of the #GList
604  */
605 GList *
606 g_list_delete_link (GList *list,
607                     GList *link_)
```

```
608 {
609   list = _g_list_remove_link (list, link_);
610   _g_list_free1 (link_);
611
612   return list;
613 }
614
615 /**
616  * g_list_copy:
617  * @list: a #GList, this must point to the top of the
    list
618  *
619  * Copies a #GList.
620  *
621  * Note that this is a "shallow" copy. If the list
    elements
622  * consist of pointers to data, the pointers are copied
    but
623  * the actual data is not. See g_list_copy_deep() if you
    need
624  * to copy the data as well.
625  *
626  * Returns: the start of the new list that holds the same
     data as @list
627  */
628 GList *
629 g_list_copy (GList *list)
630 {
631   return g_list_copy_deep (list, NULL, NULL);
632 }
633
634 /**
635  * g_list_copy_deep:
636  * @list: a #GList, this must point to the top of the
    list
637  * @func: (scope call): a copy function used to copy
    every element in the list
638  * @user_data: user data passed to the copy function @
    func, or %NULL
639  *
640  * Makes a full (deep) copy of a #GList.
641  *
642  * In contrast with g_list_copy(), this function uses @
    func to make
```

```
643  * a copy of each list element, in addition to copying
    the list
644  * container itself.
645  *
646  * @func, as a #GCopyFunc, takes two arguments, the data
    to be copied
647  * and a @user_data pointer. On common processor
    architectures, it's safe to
648  * pass %NULL as @user_data if the copy function takes
    only one argument. You
649  * may get compiler warnings from this though if
    compiling with GCC's
650  * `-Wcast-function-type` warning.
651  *
652  * For instance, if @list holds a list of GObjects, you
    can do:
653  * |[<!-- language="C" -->
654  * another_list = g_list_copy_deep (list, (GCopyFunc)
    g_object_ref, NULL);
655  * ]|
656  *
657  * And, to entirely free the new list, you could do:
658  * |[<!-- language="C" -->
659  * g_list_free_full (another_list, g_object_unref);
660  * ]|
661  *
662  * Returns: the start of the new list that holds a full
    copy of @list,
663  *     use g_list_free_full() to free it
664  *
665  * Since: 2.34
666  */
667 GList *
668 g_list_copy_deep (GList     *list,
669                   GCopyFunc  func,
670                   gpointer   user_data)
671 {
672   GList *new_list = NULL;
673
674   if (list)
675     {
676       GList *last;
677
678       new_list = _g_list_alloc ();
```

```
679       if (func)
680         new_list->data = func (list->data, user_data);
681       else
682         new_list->data = list->data;
683       new_list->prev = NULL;
684       last = new_list;
685       list = list->next;
686       while (list)
687         {
688           last->next = _g_list_alloc ();
689           last->next->prev = last;
690           last = last->next;
691           if (func)
692             last->data = func (list->data, user_data);
693           else
694             last->data = list->data;
695           list = list->next;
696         }
697       last->next = NULL;
698     }
699
700   return new_list;
701 }
702
703 /**
704  * g_list_reverse:
705  * @list: a #GList, this must point to the top of the
    list
706  *
707  * Reverses a #GList.
708  * It simply switches the next and prev pointers of each
    element.
709  *
710  * Returns: the start of the reversed #GList
711  */
712 GList *
713 g_list_reverse (GList *list)
714 {
715   GList *last;
716
717   last = NULL;
718   while (list)
719     {
720       last = list;
```

```
721       list = last->next;
722       last->next = last->prev;
723       last->prev = list;
724     }
725
726   return last;
727 }
728
729 /**
730  * g_list_nth:
731  * @list: a #GList, this must point to the top of the
    list
732  * @n: the position of the element, counting from 0
733  *
734  * Gets the element at the given position in a #GList.
735  *
736  * This iterates over the list until it reaches the @n-th
     position. If you
737  * intend to iterate over every element, it is better to
    use a for-loop as
738  * described in the #GList introduction.
739  *
740  * Returns: the element, or %NULL if the position is off
741  *     the end of the #GList
742  */
743 GList *
744 g_list_nth (GList *list,
745             guint  n)
746 {
747   while ((n-- > 0) && list)
748     list = list->next;
749
750   return list;
751 }
752
753 /**
754  * g_list_nth_prev:
755  * @list: a #GList
756  * @n: the position of the element, counting from 0
757  *
758  * Gets the element @n places before @list.
759  *
760  * Returns: the element, or %NULL if the position is
761  *     off the end of the #GList
```

```
762  */
763 GList *
764 g_list_nth_prev (GList *list,
765                  guint  n)
766 {
767   while ((n-- > 0) && list)
768     list = list->prev;
769
770   return list;
771 }
772
773 /**
774  * g_list_nth_data:
775  * @list: a #GList, this must point to the top of the
    list
776  * @n: the position of the element
777  *
778  * Gets the data of the element at the given position.
779  *
780  * This iterates over the list until it reaches the @n-th
     position. If you
781  * intend to iterate over every element, it is better to
    use a for-loop as
782  * described in the #GList introduction.
783  *
784  * Returns: the element's data, or %NULL if the position
785  *     is off the end of the #GList
786  */
787 gpointer
788 g_list_nth_data (GList *list,
789                  guint  n)
790 {
791   while ((n-- > 0) && list)
792     list = list->next;
793
794   return list ? list->data : NULL;
795 }
796
797 /**
798  * g_list_find:
799  * @list: a #GList, this must point to the top of the
    list
800  * @data: the element data to find
801  *
```

```
802  * Finds the element in a #GList which contains the given
     data.
803  *
804  * Returns: the found #GList element, or %NULL if it is
    not found
805  */
806 GList *
807 g_list_find (GList         *list,
808              gconstpointer  data)
809 {
810   while (list)
811     {
812       if (list->data == data)
813         break;
814       list = list->next;
815     }
816
817   return list;
818 }
819
820 /**
821  * g_list_find_custom:
822  * @list: a #GList, this must point to the top of the
    list
823  * @data: user data passed to the function
824  * @func: (scope call): the function to call for each
    element.
825  *     It should return 0 when the desired element is
    found
826  *
827  * Finds an element in a #GList, using a supplied
    function to
828  * find the desired element. It iterates over the list,
    calling
829  * the given function which should return 0 when the
    desired
830  * element is found. The function takes two #
    gconstpointer arguments,
831  * the #GList element's data as the first argument and
    the
832  * given user data.
833  *
834  * Returns: the found #GList element, or %NULL if it is
    not found
```

```
835  */
836 GList *
837 g_list_find_custom (GList         *list,
838                     gconstpointer  data,
839                     GCompareFunc   func)
840 {
841   g_return_val_if_fail (func != NULL, list);
842
843   while (list)
844     {
845       if (! func (list->data, data))
846         return list;
847       list = list->next;
848     }
849
850   return NULL;
851 }
852
853 /**
854  * g_list_position:
855  * @list: a #GList, this must point to the top of the
    list
856  * @llink: an element in the #GList
857  *
858  * Gets the position of the given element
859  * in the #GList (starting from 0).
860  *
861  * Returns: the position of the element in the #GList,
862  *     or -1 if the element is not found
863  */
864 gint
865 g_list_position (GList *list,
866                  GList *llink)
867 {
868   gint i;
869
870   i = 0;
871   while (list)
872     {
873       if (list == llink)
874         return i;
875       i++;
876       list = list->next;
877     }
```

```c

  return -1;
}

/**
 * g_list_index:
 * @list: a #GList, this must point to the top of the list
 * @data: the data to find
 *
 * Gets the position of the element containing
 * the given data (starting from 0).
 *
 * Returns: the index of the element containing the data,
 *     or -1 if the data is not found
 */
gint
g_list_index (GList         *list,
              gconstpointer  data)
{
  gint i;

  i = 0;
  while (list)
    {
      if (list->data == data)
        return i;
      i++;
      list = list->next;
    }

  return -1;
}

/**
 * g_list_last:
 * @list: any #GList element
 *
 * Gets the last element in a #GList.
 *
 * Returns: the last element in the #GList,
 *     or %NULL if the #GList has no elements
 */
```

```
920 GList *
921 g_list_last (GList *list)
922 {
923   if (list)
924     {
925       while (list->next)
926         list = list->next;
927     }
928
929   return list;
930 }
931
932 /**
933  * g_list_first:
934  * @list: any #GList element
935  *
936  * Gets the first element in a #GList.
937  *
938  * Returns: the first element in the #GList,
939  *     or %NULL if the #GList has no elements
940  */
941 GList *
942 g_list_first (GList *list)
943 {
944   if (list)
945     {
946       while (list->prev)
947         list = list->prev;
948     }
949
950   return list;
951 }
952
953 /**
954  * g_list_length:
955  * @list: a #GList, this must point to the top of the
    list
956  *
957  * Gets the number of elements in a #GList.
958  *
959  * This function iterates over the whole list to count
    its elements.
960  * Use a #GQueue instead of a GList if you regularly need
     the number
```

```
961  * of items. To check whether the list is non-empty, it
     is faster to check
962  * @list against %NULL.
963  *
964  * Returns: the number of elements in the #GList
965  */
966 guint
967 g_list_length (GList *list)
968 {
969   guint length;
970
971   length = 0;
972   while (list)
973     {
974       length++;
975       list = list->next;
976     }
977
978   return length;
979 }
980
981 /**
982  * g_list_foreach:
983  * @list: a #GList, this must point to the top of the
     list
984  * @func: (scope call): the function to call with each
     element's data
985  * @user_data: user data to pass to the function
986  *
987  * Calls a function for each element of a #GList.
988  *
989  * It is safe for @func to remove the element from @list
     , but it must
990  * not modify any part of the list after that element.
991  */
992 /**
993  * GFunc:
994  * @data: the element's data
995  * @user_data: user data passed to g_list_foreach() or
     g_slist_foreach()
996  *
997  * Specifies the type of functions passed to
     g_list_foreach() and
998  * g_slist_foreach().
```

```
 999  */
1000 void
1001 g_list_foreach (GList    *list,
1002                 GFunc     func,
1003                 gpointer  user_data)
1004 {
1005   while (list)
1006     {
1007       GList *next = list->next;
1008       (*func) (list->data, user_data);
1009       list = next;
1010     }
1011 }
1012
1013 static GList*
1014 g_list_insert_sorted_real (GList    *list,
1015                            gpointer  data,
1016                            GFunc     func,
1017                            gpointer  user_data)
1018 {
1019   GList *tmp_list = list;
1020   GList *new_list;
1021   gint cmp;
1022
1023   g_return_val_if_fail (func != NULL, list);
1024
1025   if (!list)
1026     {
1027       new_list = _g_list_alloc0 ();
1028       new_list->data = data;
1029       return new_list;
1030     }
1031
1032   cmp = ((GCompareDataFunc) func) (data, tmp_list->data
     , user_data);
1033
1034   while ((tmp_list->next) && (cmp > 0))
1035     {
1036       tmp_list = tmp_list->next;
1037
1038       cmp = ((GCompareDataFunc) func) (data, tmp_list->
     data, user_data);
1039     }
1040
```

```
1041   new_list = _g_list_alloc0 ();
1042   new_list->data = data;
1043
1044   if ((!tmp_list->next) && (cmp > 0))
1045     {
1046       tmp_list->next = new_list;
1047       new_list->prev = tmp_list;
1048       return list;
1049     }
1050
1051   if (tmp_list->prev)
1052     {
1053       tmp_list->prev->next = new_list;
1054       new_list->prev = tmp_list->prev;
1055     }
1056   new_list->next = tmp_list;
1057   tmp_list->prev = new_list;
1058
1059   if (tmp_list == list)
1060     return new_list;
1061   else
1062     return list;
1063 }
1064
1065 /**
1066  * g_list_insert_sorted:
1067  * @list: a pointer to a #GList, this must point to the
     top of the
1068  *     already sorted list
1069  * @data: the data for the new element
1070  * @func: (scope call): the function to compare elements
     in the list. It should
1071  *     return a number > 0 if the first parameter comes
     after the
1072  *     second parameter in the sort order.
1073  *
1074  * Inserts a new element into the list, using the given
     comparison
1075  * function to determine its position.
1076  *
1077  * If you are adding many new elements to a list, and
     the number of
1078  * new elements is much larger than the length of the
     list, use
```

```
1079  * g_list_prepend() to add the new items and sort the
      list afterwards
1080  * with g_list_sort().
1081  *
1082  * Returns: the (possibly changed) start of the #GList
1083  */
1084 GList *
1085 g_list_insert_sorted (GList        *list,
1086                       gpointer      data,
1087                       GCompareFunc  func)
1088 {
1089   return g_list_insert_sorted_real (list, data, (GFunc)
      func, NULL);
1090 }
1091
1092 /**
1093  * g_list_insert_sorted_with_data:
1094  * @list: a pointer to a #GList, this must point to the
      top of the
1095  *     already sorted list
1096  * @data: the data for the new element
1097  * @func: (scope call): the function to compare elements
      in the list. It should
1098  *     return a number > 0 if the first parameter  comes
      after the
1099  *     second parameter in the sort order.
1100  * @user_data: user data to pass to comparison function
1101  *
1102  * Inserts a new element into the list, using the given
      comparison
1103  * function to determine its position.
1104  *
1105  * If you are adding many new elements to a list, and
      the number of
1106  * new elements is much larger than the length of the
      list, use
1107  * g_list_prepend() to add the new items and sort the
      list afterwards
1108  * with g_list_sort().
1109  *
1110  * Returns: the (possibly changed) start of the #GList
1111  *
1112  * Since: 2.10
1113  */
```

```
1114 GList *
1115 g_list_insert_sorted_with_data (GList           *list,
1116                                 gpointer         data,
1117                                 GCompareDataFunc  func,
1118                                 gpointer
     user_data)
1119 {
1120   return g_list_insert_sorted_real (list, data, (GFunc)
     func, user_data);
1121 }
1122
1123 static GList *
1124 g_list_sort_merge (GList     *l1,
1125                    GList     *l2,
1126                    GFunc     compare_func,
1127                    gpointer  user_data)
1128 {
1129   GList list, *l, *lprev;
1130   gint cmp;
1131
1132   l = &list;
1133   lprev = NULL;
1134
1135   while (l1 && l2)
1136     {
1137       cmp = ((GCompareDataFunc) compare_func) (l1->data
     , l2->data, user_data);
1138
1139       if (cmp <= 0)
1140         {
1141           l->next = l1;
1142           l1 = l1->next;
1143         }
1144       else
1145         {
1146           l->next = l2;
1147           l2 = l2->next;
1148         }
1149       l = l->next;
1150       l->prev = lprev;
1151       lprev = l;
1152     }
1153   l->next = l1 ? l1 : l2;
1154   l->next->prev = l;
```

```
1155
1156   return list.next;
1157 }
1158
1159 static GList *
1160 g_list_sort_real (GList    *list,
1161                   GFunc     compare_func,
1162                   gpointer  user_data)
1163 {
1164   GList *l1, *l2;
1165
1166   if (!list)
1167     return NULL;
1168   if (!list->next)
1169     return list;
1170
1171   l1 = list;
1172   l2 = list->next;
1173
1174   while ((l2 = l2->next) != NULL)
1175     {
1176       if ((l2 = l2->next) == NULL)
1177         break;
1178       l1 = l1->next;
1179     }
1180   l2 = l1->next;
1181   l1->next = NULL;
1182
1183   return g_list_sort_merge (g_list_sort_real (list,
     compare_func, user_data),
1184                             g_list_sort_real (l2,
     compare_func, user_data),
1185                             compare_func,
1186                             user_data);
1187 }
1188
1189 /**
1190  * g_list_sort:
1191  * @list: a #GList, this must point to the top of the
     list
1192  * @compare_func: (scope call): the comparison function
     used to sort the #GList.
1193  *     This function is passed the data from 2 elements
     of the #GList
```

```
1194  *     and should return 0 if they are equal, a negative
      value if the
1195  *     first element comes before the second, or a
     positive value if
1196  *     the first element comes after the second.
1197  *
1198  * Sorts a #GList using the given comparison function.
     The algorithm
1199  * used is a stable sort.
1200  *
1201  * Returns: the (possibly changed) start of the #GList
1202  */
1203 /**
1204  * GCompareFunc:
1205  * @a: a value
1206  * @b: a value to compare with
1207  *
1208  * Specifies the type of a comparison function used to
     compare two
1209  * values.  The function should return a negative
     integer if the first
1210  * value comes before the second, 0 if they are equal,
     or a positive
1211  * integer if the first value comes after the second.
1212  *
1213  * Returns: negative value if @a < @b; zero if @a = @b;
     positive
1214  *          value if @a > @b
1215  */
1216 GList *
1217 g_list_sort (GList        *list,
1218              GCompareFunc  compare_func)
1219 {
1220   return g_list_sort_real (list, (GFunc) compare_func,
     NULL);
1221 }
1222
1223 /**
1224  * g_list_sort_with_data:
1225  * @list: a #GList, this must point to the top of the
     list
1226  * @compare_func: (scope call): comparison function
1227  * @user_data: user data to pass to comparison function
1228  *
```

```
1229  * Like g_list_sort(), but the comparison function
      accepts
1230  * a user data argument.
1231  *
1232  * Returns: the (possibly changed) start of the #GList
1233  */
1234 /**
1235  * GCompareDataFunc:
1236  * @a: a value
1237  * @b: a value to compare with
1238  * @user_data: user data
1239  *
1240  * Specifies the type of a comparison function used to
      compare two
1241  * values.  The function should return a negative
      integer if the first
1242  * value comes before the second, 0 if they are equal,
      or a positive
1243  * integer if the first value comes after the second.
1244  *
1245  * Returns: negative value if @a < @b; zero if @a = @b;
      positive
1246  *          value if @a > @b
1247  */
1248 GList *
1249 g_list_sort_with_data (GList            *list,
1250                        GCompareDataFunc  compare_func,
1251                        gpointer          user_data)
1252 {
1253   return g_list_sort_real (list, (GFunc) compare_func,
      user_data);
1254 }
1255
1256 /**
1257  * g_clear_list: (skip)
1258  * @list_ptr: (not nullable): a #GList return location
1259  * @destroy: (nullable): the function to pass to
      g_list_free_full() or %NULL to not free elements
1260  *
1261  * Clears a pointer to a #GList, freeing it and,
      optionally, freeing its elements using @destroy.
1262  *
1263  * @list_ptr must be a valid pointer. If @list_ptr
      points to a null #GList, this does nothing.
```

```
1264  *
1265  * Since: 2.64
1266  */
1267 void
1268 (g_clear_list) (GList          **list_ptr,
1269                 GDestroyNotify   destroy)
1270 {
1271   GList *list;
1272
1273   list = *list_ptr;
1274   if (list)
1275     {
1276       *list_ptr = NULL;
1277
1278       if (destroy)
1279         g_list_free_full (list, destroy);
1280       else
1281         g_list_free (list);
1282     }
1283 }
1284
```

```c
/* GLIB - Library of useful routines for C programming
 * Copyright (C) 1995-1997  Peter Mattis, Spencer Kimball and Josh MacDonald
 *
 * SPDX-License-Identifier: LGPL-2.1-or-later
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, see <http://www.gnu.org/licenses/>.
 */

/*
 * Modified by the GLib Team and others 1997-2000.  See the AUTHORS
 * file for a list of people on the GLib Team.  See the ChangeLog
 * files for a list of changes.  These files are distributed with
 * GLib at ftp://ftp.gtk.org/pub/gtk/.
 */

#ifndef __G_LIST_H__
#define __G_LIST_H__

#if !defined (__GLIB_H_INSIDE__) && !defined (GLIB_COMPILATION)
```

```
31 #error "Only <glib.h> can be included directly."
32 #endif
33
34 #include <glib/gmem.h>
35 #include <glib/gnode.h>
36
37 G_BEGIN_DECLS
38
39 typedef struct _GList GList;
40
41 struct _GList
42 {
43   gpointer data;
44   GList *next;
45   GList *prev;
46 };
47
48 /* Doubly linked lists
49  */
50 GLIB_AVAILABLE_IN_ALL
51 GList*   g_list_alloc                    (void)
   G_GNUC_WARN_UNUSED_RESULT;
52 GLIB_AVAILABLE_IN_ALL
53 void     g_list_free                     (GList
              *list);
54 GLIB_AVAILABLE_IN_ALL
55 void     g_list_free_1                   (GList
              *list);
56 #define  g_list_free1                    g_list_free_1
57 GLIB_AVAILABLE_IN_ALL
58 void     g_list_free_full                (GList
              *list,
59                       GDestroyNotify    free_func);
60 GLIB_AVAILABLE_IN_ALL
61 GList*   g_list_append                   (GList
              *list,
62                       gpointer          data)
   G_GNUC_WARN_UNUSED_RESULT;
63 GLIB_AVAILABLE_IN_ALL
64 GList*   g_list_prepend                  (GList
              *list,
65                       gpointer          data)
   G_GNUC_WARN_UNUSED_RESULT;
66 GLIB_AVAILABLE_IN_ALL
```

```
67 GList*   g_list_insert                  (GList
              *list,
68                      gpointer          data,
69                      gint              position)
   G_GNUC_WARN_UNUSED_RESULT;
70 GLIB_AVAILABLE_IN_ALL
71 GList*   g_list_insert_sorted           (GList
              *list,
72                      gpointer          data,
73                      GCompareFunc      func)
   G_GNUC_WARN_UNUSED_RESULT;
74 GLIB_AVAILABLE_IN_ALL
75 GList*   g_list_insert_sorted_with_data (GList
              *list,
76                      gpointer          data,
77                      GCompareDataFunc  func,
78                      gpointer          user_data)
   G_GNUC_WARN_UNUSED_RESULT;
79 GLIB_AVAILABLE_IN_ALL
80 GList*   g_list_insert_before           (GList
              *list,
81                      GList            *sibling,
82                      gpointer          data)
   G_GNUC_WARN_UNUSED_RESULT;
83 GLIB_AVAILABLE_IN_2_62
84 GList*   g_list_insert_before_link      (GList
              *list,
85                      GList            *sibling,
86                      GList            *link_)
   G_GNUC_WARN_UNUSED_RESULT;
87 GLIB_AVAILABLE_IN_ALL
88 GList*   g_list_concat                  (GList
              *list1,
89                      GList            *list2)
   G_GNUC_WARN_UNUSED_RESULT;
90 GLIB_AVAILABLE_IN_ALL
91 GList*   g_list_remove                  (GList
              *list,
92                      gconstpointer     data)
   G_GNUC_WARN_UNUSED_RESULT;
93 GLIB_AVAILABLE_IN_ALL
94 GList*   g_list_remove_all              (GList
              *list,
95                      gconstpointer     data)
```

```
 95 G_GNUC_WARN_UNUSED_RESULT;
 96 GLIB_AVAILABLE_IN_ALL
 97 GList*   g_list_remove_link             (GList
              *list,
 98                      GList            *llink)
    G_GNUC_WARN_UNUSED_RESULT;
 99 GLIB_AVAILABLE_IN_ALL
100 GList*   g_list_delete_link             (GList
              *list,
101                      GList            *link_)
    G_GNUC_WARN_UNUSED_RESULT;
102 GLIB_AVAILABLE_IN_ALL
103 GList*   g_list_reverse                  (GList
              *list) G_GNUC_WARN_UNUSED_RESULT;
104 GLIB_AVAILABLE_IN_ALL
105 GList*   g_list_copy                     (GList
              *list) G_GNUC_WARN_UNUSED_RESULT;
106
107 GLIB_AVAILABLE_IN_2_34
108 GList*   g_list_copy_deep                (GList
              *list,
109                      GCopyFunc         func,
110                      gpointer          user_data)
    G_GNUC_WARN_UNUSED_RESULT;
111
112 GLIB_AVAILABLE_IN_ALL
113 GList*   g_list_nth                      (GList
              *list,
114                      guint             n);
115 GLIB_AVAILABLE_IN_ALL
116 GList*   g_list_nth_prev                 (GList
              *list,
117                      guint             n);
118 GLIB_AVAILABLE_IN_ALL
119 GList*   g_list_find                     (GList
              *list,
120                      gconstpointer     data);
121 GLIB_AVAILABLE_IN_ALL
122 GList*   g_list_find_custom              (GList
              *list,
123                      gconstpointer     data,
124                      GCompareFunc      func);
125 GLIB_AVAILABLE_IN_ALL
126 gint     g_list_position                 (GList
```

```
126             *list,
127                     GList           *llink);
128 GLIB_AVAILABLE_IN_ALL
129 gint     g_list_index                  (GList
                *list,
130                     gconstpointer     data);
131 GLIB_AVAILABLE_IN_ALL
132 GList*   g_list_last                   (GList
                *list);
133 GLIB_AVAILABLE_IN_ALL
134 GList*   g_list_first                  (GList
                *list);
135 GLIB_AVAILABLE_IN_ALL
136 guint    g_list_length                 (GList
                *list);
137 GLIB_AVAILABLE_IN_ALL
138 void     g_list_foreach                (GList
                *list,
139                     GFunc             func,
140                     gpointer          user_data);
141 GLIB_AVAILABLE_IN_ALL
142 GList*   g_list_sort                   (GList
                *list,
143                     GCompareFunc      compare_func)
    G_GNUC_WARN_UNUSED_RESULT;
144 GLIB_AVAILABLE_IN_ALL
145 GList*   g_list_sort_with_data         (GList
                *list,
146                     GCompareDataFunc  compare_func,
147                     gpointer          user_data)
    G_GNUC_WARN_UNUSED_RESULT;
148 GLIB_AVAILABLE_IN_ALL
149 gpointer g_list_nth_data               (GList
                *list,
150                     guint             n);
151
152 GLIB_AVAILABLE_IN_2_64
153 void     g_clear_list                  (GList
              **list_ptr,
154                                         GDestroyNotify
        destroy);
155
156 #define  g_clear_list(list_ptr, destroy)       \
157   G_STMT_START {                               \
```

```
158     GList *_list;                                   \
159                                                     \
160     _list = *(list_ptr);                            \
161     if (_list)                                      \
162       {                                             \
163         *list_ptr = NULL;                           \
164                                                     \
165         if ((destroy) != NULL)                      \
166           g_list_free_full (_list, (destroy));      \
167         else                                        \
168           g_list_free (_list);                      \
169       }                                             \
170   } G_STMT_END                                      \
171   GLIB_AVAILABLE_MACRO_IN_2_64
172
173
174 #define g_list_previous(list)           ((list) ? (((
    GList *)(list))->prev) : NULL)
175 #define g_list_next(list)           ((list) ? (((GList
     *)(list))->next) : NULL)
176
177 G_END_DECLS
178
179 #endif /* __G_LIST_H__ */
180
```

```c
#ifndef _LINUX_LIST_H
#define _LINUX_LIST_H

#include <linux/types.h>
#include <linux/stddef.h>
#include <linux/poison.h>
#include <linux/const.h>
#include <linux/kernel.h>

/*
 * Simple doubly linked list implementation.
 *
 * Some of the internal functions ("__xxx") are useful when
 * manipulating whole lists rather than single entries, as
 * sometimes we already know the next/prev entries and we can
 * generate better code by using them directly rather than
 * using the generic single-entry routines.
 */

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    WRITE_ONCE(list->next, list);
    list->prev = list;
}

#ifdef CONFIG_DEBUG_LIST
extern bool __list_add_valid(struct list_head *new,
                  struct list_head *prev,
                  struct list_head *next);
extern bool __list_del_entry_valid(struct list_head *entry);
#else
static inline bool __list_add_valid(struct list_head *new,
                struct list_head *prev,
                struct list_head *next)
{
    return true;
```

```
42 }
43 static inline bool __list_del_entry_valid(struct list_head
    *entry)
44 {
45     return true;
46 }
47 #endif
48
49 /*
50  * Insert a new entry between two known consecutive
   entries.
51  *
52  * This is only for internal list manipulation where we
   know
53  * the prev/next entries already!
54  */
55 static inline void __list_add(struct list_head *new,
56                   struct list_head *prev,
57                   struct list_head *next)
58 {
59     if (!__list_add_valid(new, prev, next))
60         return;
61
62     next->prev = new;
63     new->next = next;
64     new->prev = prev;
65     WRITE_ONCE(prev->next, new);
66 }
67
68 /**
69  * list_add - add a new entry
70  * @new: new entry to be added
71  * @head: list head to add it after
72  *
73  * Insert a new entry after the specified head.
74  * This is good for implementing stacks.
75  */
76 static inline void list_add(struct list_head *new, struct
   list_head *head)
77 {
78     __list_add(new, head, head->next);
79 }
80
81
```

```c
/**
 * list_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static inline void list_add_tail(struct list_head *new,
struct list_head *head)
{
    __list_add(new, head->prev, head);
}

/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we
know
 * the prev/next entries already!
 */
static inline void __list_del(struct list_head * prev,
struct list_head * next)
{
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}

/**
 * list_del - deletes entry from list.
 * @entry: the element to delete from the list.
 * Note: list_empty() on entry does not return true after
 this, the entry is
 * in an undefined state.
 */
static inline void __list_del_entry(struct list_head *
entry)
{
    if (!__list_del_entry_valid(entry))
        return;

    __list_del(entry->prev, entry->next);
}
```

```
121
122 static inline void list_del(struct list_head *entry)
123 {
124     __list_del_entry(entry);
125     entry->next = LIST_POISON1;
126     entry->prev = LIST_POISON2;
127 }
128
129 /**
130  * list_replace - replace old entry by new one
131  * @old : the element to be replaced
132  * @new : the new element to insert
133  *
134  * If @old was empty, it will be overwritten.
135  */
136 static inline void list_replace(struct list_head *old,
137                 struct list_head *new)
138 {
139     new->next = old->next;
140     new->next->prev = new;
141     new->prev = old->prev;
142     new->prev->next = new;
143 }
144
145 static inline void list_replace_init(struct list_head *
    old,
146                     struct list_head *new)
147 {
148     list_replace(old, new);
149     INIT_LIST_HEAD(old);
150 }
151
152 /**
153  * list_del_init - deletes entry from list and
    reinitialize it.
154  * @entry: the element to delete from the list.
155  */
156 static inline void list_del_init(struct list_head *entry)
157 {
158     __list_del_entry(entry);
159     INIT_LIST_HEAD(entry);
160 }
161
162 /**
```

```
163  * list_move - delete from one list and add as another's
     head
164  * @list: the entry to move
165  * @head: the head that will precede our entry
166  */
167 static inline void list_move(struct list_head *list,
     struct list_head *head)
168 {
169     __list_del_entry(list);
170     list_add(list, head);
171 }
172
173 /**
174  * list_move_tail - delete from one list and add as
     another's tail
175  * @list: the entry to move
176  * @head: the head that will follow our entry
177  */
178 static inline void list_move_tail(struct list_head *list,
179                   struct list_head *head)
180 {
181     __list_del_entry(list);
182     list_add_tail(list, head);
183 }
184
185 /**
186  * list_is_last - tests whether @list is the last entry
     in list @head
187  * @list: the entry to test
188  * @head: the head of the list
189  */
190 static inline int list_is_last(const struct list_head *
     list,
191                 const struct list_head *head)
192 {
193     return list->next == head;
194 }
195
196 /**
197  * list_empty - tests whether a list is empty
198  * @head: the list to test.
199  */
200 static inline int list_empty(const struct list_head *head
     )
```

```
201 {
202     return READ_ONCE(head->next) == head;
203 }
204 
205 /**
206  * list_empty_careful - tests whether a list is empty and
     not being modified
207  * @head: the list to test
208  *
209  * Description:
210  * tests whether a list is empty _and_ checks that no
    other CPU might be
211  * in the process of modifying either member (next or
    prev)
212  *
213  * NOTE: using list_empty_careful() without
    synchronization
214  * can only be safe if the only activity that can happen
215  * to the list entry is list_del_init(). Eg. it cannot be
     used
216  * if another CPU could re-list_add() it.
217  */
218 static inline int list_empty_careful(const struct
    list_head *head)
219 {
220     struct list_head *next = head->next;
221     return (next == head) && (next == head->prev);
222 }
223 
224 /**
225  * list_rotate_left - rotate the list to the left
226  * @head: the head of the list
227  */
228 static inline void list_rotate_left(struct list_head *
    head)
229 {
230     struct list_head *first;
231 
232     if (!list_empty(head)) {
233         first = head->next;
234         list_move_tail(first, head);
235     }
236 }
237 
```

```
238 /**
239  * list_is_singular - tests whether a list has just one
    entry.
240  * @head: the list to test.
241  */
242 static inline int list_is_singular(const struct list_head
     *head)
243 {
244     return !list_empty(head) && (head->next == head->prev
    );
245 }
246
247 static inline void __list_cut_position(struct list_head *
    list,
248         struct list_head *head, struct list_head *entry)
249 {
250     struct list_head *new_first = entry->next;
251     list->next = head->next;
252     list->next->prev = list;
253     list->prev = entry;
254     entry->next = list;
255     head->next = new_first;
256     new_first->prev = head;
257 }
258
259 /**
260  * list_cut_position - cut a list into two
261  * @list: a new list to add all removed entries
262  * @head: a list with entries
263  * @entry: an entry within head, could be the head itself
264  *  and if so we won't cut the list
265  *
266  * This helper moves the initial part of @head, up to and
267  * including @entry, from @head to @list. You should
268  * pass on @entry an element you know is on @head. @list
269  * should be an empty list or a list you do not care
    about
270  * losing its data.
271  *
272  */
273 static inline void list_cut_position(struct list_head *
    list,
274         struct list_head *head, struct list_head *entry)
275 {
```

```
276     if (list_empty(head))
277         return;
278     if (list_is_singular(head) &&
279         (head->next != entry && head != entry))
280         return;
281     if (entry == head)
282         INIT_LIST_HEAD(list);
283     else
284         __list_cut_position(list, head, entry);
285 }
286 
287 static inline void __list_splice(const struct list_head *
    list,
288                  struct list_head *prev,
289                  struct list_head *next)
290 {
291     struct list_head *first = list->next;
292     struct list_head *last = list->prev;
293 
294     first->prev = prev;
295     prev->next = first;
296 
297     last->next = next;
298     next->prev = last;
299 }
300 
301 /**
302  * list_splice - join two lists, this is designed for
    stacks
303  * @list: the new list to add.
304  * @head: the place to add it in the first list.
305  */
306 static inline void list_splice(const struct list_head *
    list,
307                 struct list_head *head)
308 {
309     if (!list_empty(list))
310         __list_splice(list, head, head->next);
311 }
312 
313 /**
314  * list_splice_tail - join two lists, each list being a
    queue
315  * @list: the new list to add.
```

```
316  * @head: the place to add it in the first list.
317  */
318 static inline void list_splice_tail(struct list_head *
    list,
319                 struct list_head *head)
320 {
321     if (!list_empty(list))
322         __list_splice(list, head->prev, head);
323 }
324 
325 /**
326  * list_splice_init - join two lists and reinitialise the
     emptied list.
327  * @list: the new list to add.
328  * @head: the place to add it in the first list.
329  *
330  * The list at @list is reinitialised
331  */
332 static inline void list_splice_init(struct list_head *
    list,
333                     struct list_head *head)
334 {
335     if (!list_empty(list)) {
336         __list_splice(list, head, head->next);
337         INIT_LIST_HEAD(list);
338     }
339 }
340 
341 /**
342  * list_splice_tail_init - join two lists and
    reinitialise the emptied list
343  * @list: the new list to add.
344  * @head: the place to add it in the first list.
345  *
346  * Each of the lists is a queue.
347  * The list at @list is reinitialised
348  */
349 static inline void list_splice_tail_init(struct list_head
     *list,
350                      struct list_head *head)
351 {
352     if (!list_empty(list)) {
353         __list_splice(list, head->prev, head);
354         INIT_LIST_HEAD(list);
```

```
355     }
356 }
357 
358 /**
359  * list_entry - get the struct for this entry
360  * @ptr:    the &struct list_head pointer.
361  * @type:   the type of the struct this is embedded in.
362  * @member: the name of the list_head within the struct.
363  */
364 #define list_entry(ptr, type, member) \
365     container_of(ptr, type, member)
366 
367 /**
368  * list_first_entry - get the first element from a list
369  * @ptr:    the list head to take the element from.
370  * @type:   the type of the struct this is embedded in.
371  * @member: the name of the list_head within the struct.
372  *
373  * Note, that list is expected to be not empty.
374  */
375 #define list_first_entry(ptr, type, member) \
376     list_entry((ptr)->next, type, member)
377 
378 /**
379  * list_last_entry - get the last element from a list
380  * @ptr:    the list head to take the element from.
381  * @type:   the type of the struct this is embedded in.
382  * @member: the name of the list_head within the struct.
383  *
384  * Note, that list is expected to be not empty.
385  */
386 #define list_last_entry(ptr, type, member) \
387     list_entry((ptr)->prev, type, member)
388 
389 /**
390  * list_first_entry_or_null - get the first element from
    a list
391  * @ptr:    the list head to take the element from.
392  * @type:   the type of the struct this is embedded in.
393  * @member: the name of the list_head within the struct.
394  *
395  * Note that if the list is empty, it returns NULL.
396  */
397 #define list_first_entry_or_null(ptr, type, member) ({ \
```

```
398     struct list_head *head__ = (ptr); \
399     struct list_head *pos__ = READ_ONCE(head__->next); \
400     pos__ != head__ ? list_entry(pos__, type, member) :
    NULL; \
401 })
402
403 /**
404  * list_next_entry - get the next element in list
405  * @pos:    the type * to cursor
406  * @member: the name of the list_head within the struct.
407  */
408 #define list_next_entry(pos, member) \
409     list_entry((pos)->member.next, typeof(*(pos)), member
    )
410
411 /**
412  * list_prev_entry - get the prev element in list
413  * @pos:    the type * to cursor
414  * @member: the name of the list_head within the struct.
415  */
416 #define list_prev_entry(pos, member) \
417     list_entry((pos)->member.prev, typeof(*(pos)), member
    )
418
419 /**
420  * list_for_each-   iterate over a list
421  * @pos:    the &struct list_head to use as a loop cursor.
422  * @head:   the head for your list.
423  */
424 #define list_for_each(pos, head) \
425     for (pos = (head)->next; pos != (head); pos = pos->
    next)
426
427 /**
428  * list_for_each_prev   -   iterate over a list backwards
429  * @pos:    the &struct list_head to use as a loop cursor.
430  * @head:   the head for your list.
431  */
432 #define list_for_each_prev(pos, head) \
433     for (pos = (head)->prev; pos != (head); pos = pos->
    prev)
434
435 /**
436  * list_for_each_safe - iterate over a list safe against
```

```
436 removal of list entry
437  * @pos:    the &struct list_head to use as a loop cursor.
438  * @n:      another &struct list_head to use as temporary
     storage
439  * @head:   the head for your list.
440  */
441 #define list_for_each_safe(pos, n, head) \
442     for (pos = (head)->next, n = pos->next; pos != (head
    ); \
443         pos = n, n = pos->next)
444
445 /**
446  * list_for_each_prev_safe - iterate over a list
    backwards safe against removal of list entry
447  * @pos:    the &struct list_head to use as a loop cursor.
448  * @n:      another &struct list_head to use as temporary
     storage
449  * @head:   the head for your list.
450  */
451 #define list_for_each_prev_safe(pos, n, head) \
452     for (pos = (head)->prev, n = pos->prev; \
453          pos != (head); \
454          pos = n, n = pos->prev)
455
456 /**
457  * list_for_each_entry  -   iterate over list of given
    type
458  * @pos:    the type * to use as a loop cursor.
459  * @head:   the head for your list.
460  * @member: the name of the list_head within the struct.
461  */
462 #define list_for_each_entry(pos, head, member
    )                \
463     for (pos = list_first_entry(head, typeof(*pos),
    member);    \
464          &pos->member != (head);                    \
465          pos = list_next_entry(pos, member))
466
467 /**
468  * list_for_each_entry_reverse - iterate backwards over
    list of given type.
469  * @pos:    the type * to use as a loop cursor.
470  * @head:   the head for your list.
471  * @member: the name of the list_head within the struct.
```

```
472  */
473 #define list_for_each_entry_reverse(pos, head, member
    )           \
474     for (pos = list_last_entry(head, typeof(*pos), member
    );      \
475          &pos->member != (head);                    \
476          pos = list_prev_entry(pos, member))
477
478 /**
479  * list_prepare_entry - prepare a pos entry for use in
    list_for_each_entry_continue()
480  * @pos:    the type * to use as a start point
481  * @head:   the head of the list
482  * @member: the name of the list_head within the struct.
483  *
484  * Prepares a pos entry for use as a start point in
    list_for_each_entry_continue().
485  */
486 #define list_prepare_entry(pos, head, member) \
487     ((pos) ? : list_entry(head, typeof(*pos), member))
488
489 /**
490  * list_for_each_entry_continue - continue iteration over
     list of given type
491  * @pos:    the type * to use as a loop cursor.
492  * @head:   the head for your list.
493  * @member: the name of the list_head within the struct.
494  *
495  * Continue to iterate over list of given type,
    continuing after
496  * the current position.
497  */
498 #define list_for_each_entry_continue(pos, head, member
    )       \
499     for (pos = list_next_entry(pos, member);            \
500          &pos->member != (head);                    \
501          pos = list_next_entry(pos, member))
502
503 /**
504  * list_for_each_entry_continue_reverse - iterate
    backwards from the given point
505  * @pos:    the type * to use as a loop cursor.
506  * @head:   the head for your list.
507  * @member: the name of the list_head within the struct.
```

```
508  *
509  * Start to iterate over list of given type backwards,
     continuing after
510  * the current position.
511  */
512 #define list_for_each_entry_continue_reverse(pos, head,
     member)     \
513     for (pos = list_prev_entry(pos, member);            \
514          &pos->member != (head);                    \
515          pos = list_prev_entry(pos, member))
516
517 /**
518  * list_for_each_entry_from - iterate over list of given
     type from the current point
519  * @pos:    the type * to use as a loop cursor.
520  * @head:   the head for your list.
521  * @member: the name of the list_head within the struct.
522  *
523  * Iterate over list of given type, continuing from
     current position.
524  */
525 #define list_for_each_entry_from(pos, head, member
     )           \
526     for (; &pos->member != (head);                  \
527          pos = list_next_entry(pos, member))
528
529 /**
530  * list_for_each_entry_from_reverse - iterate backwards
     over list of given type
531  *                                    from the current
     point
532  * @pos:    the type * to use as a loop cursor.
533  * @head:   the head for your list.
534  * @member: the name of the list_head within the struct.
535  *
536  * Iterate backwards over list of given type, continuing
     from current position.
537  */
538 #define list_for_each_entry_from_reverse(pos, head,
     member)     \
539     for (; &pos->member != (head);                  \
540          pos = list_prev_entry(pos, member))
541
542 /**
```

```
543  * list_for_each_entry_safe - iterate over list of given
     type safe against removal of list entry
544  * @pos:    the type * to use as a loop cursor.
545  * @n:      another type * to use as temporary storage
546  * @head:   the head for your list.
547  * @member: the name of the list_head within the struct.
548  */
549 #define list_for_each_entry_safe(pos, n, head, member
     )           \
550     for (pos = list_first_entry(head, typeof(*pos),
     member),    \
551          n = list_next_entry(pos, member);          \
552           &pos->member != (head);                   \
553           pos = n, n = list_next_entry(n, member))
554
555 /**
556  * list_for_each_entry_safe_continue - continue list
     iteration safe against removal
557  * @pos:    the type * to use as a loop cursor.
558  * @n:      another type * to use as temporary storage
559  * @head:   the head for your list.
560  * @member: the name of the list_head within the struct.
561  *
562  * Iterate over list of given type, continuing after
     current point,
563  * safe against removal of list entry.
564  */
565 #define list_for_each_entry_safe_continue(pos, n, head,
     member)         \
566     for (pos = list_next_entry(pos, member
     ),              \
567          n = list_next_entry(pos, member);             \
568           &pos->member != (head);                      \
569           pos = n, n = list_next_entry(n, member))
570
571 /**
572  * list_for_each_entry_safe_from - iterate over list from
      current point safe against removal
573  * @pos:    the type * to use as a loop cursor.
574  * @n:      another type * to use as temporary storage
575  * @head:   the head for your list.
576  * @member: the name of the list_head within the struct.
577  *
578  * Iterate over list of given type from current point,
```

```
578 safe against
579  * removal of list entry.
580  */
581 #define list_for_each_entry_safe_from(pos, n, head,
    member)             \
582     for (n = list_next_entry(pos, member
    );                  \
583          &pos->member != (head);                    \
584          pos = n, n = list_next_entry(n, member))
585
586 /**
587  * list_for_each_entry_safe_reverse - iterate backwards
    over list safe against removal
588  * @pos:    the type * to use as a loop cursor.
589  * @n:      another type * to use as temporary storage
590  * @head:   the head for your list.
591  * @member: the name of the list_head within the struct.
592  *
593  * Iterate backwards over list of given type, safe
    against removal
594  * of list entry.
595  */
596 #define list_for_each_entry_safe_reverse(pos, n, head,
    member)     \
597     for (pos = list_last_entry(head, typeof(*pos), member
    ),      \
598          n = list_prev_entry(pos, member);          \
599          &pos->member != (head);                    \
600          pos = n, n = list_prev_entry(n, member))
601
602 /**
603  * list_safe_reset_next - reset a stale
    list_for_each_entry_safe loop
604  * @pos:    the loop cursor used in the
    list_for_each_entry_safe loop
605  * @n:      temporary storage used in
    list_for_each_entry_safe
606  * @member: the name of the list_head within the struct.
607  *
608  * list_safe_reset_next is not safe to use in general if
    the list may be
609  * modified concurrently (eg. the lock is dropped in the
    loop body). An
610  * exception to this is if the cursor element (pos) is
```

```
610 pinned in the list,
611  * and list_safe_reset_next is called after re-taking the
     lock and before
612  * completing the current iteration of the loop body.
613  */
614 #define list_safe_reset_next(pos, n, member)                \
615     n = list_next_entry(pos, member)
616
617 /*
618  * Double linked lists with a single pointer list head.
619  * Mostly useful for hash tables where the two pointer
    list head is
620  * too wasteful.
621  * You lose the ability to access the tail in O(1).
622  */
623
624 #define HLIST_HEAD_INIT { .first = NULL }
625 #define HLIST_HEAD(name) struct hlist_head name = {  .
    first = NULL }
626 #define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
627 static inline void INIT_HLIST_NODE(struct hlist_node *h)
628 {
629     h->next = NULL;
630     h->pprev = NULL;
631 }
632
633 static inline int hlist_unhashed(const struct hlist_node
     *h)
634 {
635     return !h->pprev;
636 }
637
638 static inline int hlist_empty(const struct hlist_head *h)
639 {
640     return !READ_ONCE(h->first);
641 }
642
643 static inline void __hlist_del(struct hlist_node *n)
644 {
645     struct hlist_node *next = n->next;
646     struct hlist_node **pprev = n->pprev;
647
648     WRITE_ONCE(*pprev, next);
649     if (next)
```

```
650         next->pprev = pprev;
651 }
652
653 static inline void hlist_del(struct hlist_node *n)
654 {
655     __hlist_del(n);
656     n->next = LIST_POISON1;
657     n->pprev = LIST_POISON2;
658 }
659
660 static inline void hlist_del_init(struct hlist_node *n)
661 {
662     if (!hlist_unhashed(n)) {
663         __hlist_del(n);
664         INIT_HLIST_NODE(n);
665     }
666 }
667
668 static inline void hlist_add_head(struct hlist_node *n,
    struct hlist_head *h)
669 {
670     struct hlist_node *first = h->first;
671     n->next = first;
672     if (first)
673         first->pprev = &n->next;
674     WRITE_ONCE(h->first, n);
675     n->pprev = &h->first;
676 }
677
678 /* next must be != NULL */
679 static inline void hlist_add_before(struct hlist_node *n,
680                     struct hlist_node *next)
681 {
682     n->pprev = next->pprev;
683     n->next = next;
684     next->pprev = &n->next;
685     WRITE_ONCE(*(n->pprev), n);
686 }
687
688 static inline void hlist_add_behind(struct hlist_node *n,
689                     struct hlist_node *prev)
690 {
691     n->next = prev->next;
692     WRITE_ONCE(prev->next, n);
```

```
693     n->pprev = &prev->next;
694 
695     if (n->next)
696         n->next->pprev  = &n->next;
697 }
698 
699 /* after that we'll appear to be on some hlist and
    hlist_del will work */
700 static inline void hlist_add_fake(struct hlist_node *n)
701 {
702     n->pprev = &n->next;
703 }
704 
705 static inline bool hlist_fake(struct hlist_node *h)
706 {
707     return h->pprev == &h->next;
708 }
709 
710 /*
711  * Check whether the node is the only node of the head
    without
712  * accessing head:
713  */
714 static inline bool
715 hlist_is_singular_node(struct hlist_node *n, struct
    hlist_head *h)
716 {
717     return !n->next && n->pprev == &h->first;
718 }
719 
720 /*
721  * Move a list from one list head to another. Fixup the
    pprev
722  * reference of the first entry if it exists.
723  */
724 static inline void hlist_move_list(struct hlist_head *old
    ,
725                    struct hlist_head *new)
726 {
727     new->first = old->first;
728     if (new->first)
729         new->first->pprev = &new->first;
730     old->first = NULL;
731 }
```

```
732
733 #define hlist_entry(ptr, type, member) container_of(ptr,
    type,member)
734
735 #define hlist_for_each(pos, head) \
736     for (pos = (head)->first; pos ; pos = pos->next)
737
738 #define hlist_for_each_safe(pos, n, head) \
739     for (pos = (head)->first; pos && ({ n = pos->next; 1
    ; }); \
740          pos = n)
741
742 #define hlist_entry_safe(ptr, type, member) \
743     ({ typeof(ptr) ____ptr = (ptr); \
744        ____ptr ? hlist_entry(____ptr, type, member) :
    NULL; \
745     })
746
747 /**
748  * hlist_for_each_entry - iterate over list of given type
749  * @pos:    the type * to use as a loop cursor.
750  * @head:   the head for your list.
751  * @member: the name of the hlist_node within the struct.
752  */
753 #define hlist_for_each_entry(pos, head, member
    )                \
754     for (pos = hlist_entry_safe((head)->first, typeof(*(
    pos)), member);\
755          pos;                                \
756          pos = hlist_entry_safe((pos)->member.next,
    typeof(*(pos)), member))
757
758 /**
759  * hlist_for_each_entry_continue - iterate over a hlist
    continuing after current point
760  * @pos:    the type * to use as a loop cursor.
761  * @member: the name of the hlist_node within the struct.
762  */
763 #define hlist_for_each_entry_continue(pos, member
    )           \
764     for (pos = hlist_entry_safe((pos)->member.next,
    typeof(*(pos)), member);\
765          pos;                                \
766          pos = hlist_entry_safe((pos)->member.next,
```

```
766 typeof(*(pos)), member))
767
768 /**
769  * hlist_for_each_entry_from - iterate over a hlist
    continuing from current point
770  * @pos:    the type * to use as a loop cursor.
771  * @member: the name of the hlist_node within the struct.
772  */
773 #define hlist_for_each_entry_from(pos, member
    )                \
774     for (; pos;                            \
775          pos = hlist_entry_safe((pos)->member.next,
    typeof(*(pos)), member))
776
777 /**
778  * hlist_for_each_entry_safe - iterate over list of given
     type safe against removal of list entry
779  * @pos:    the type * to use as a loop cursor.
780  * @n:      another &struct hlist_node to use as
    temporary storage
781  * @head:   the head for your list.
782  * @member: the name of the hlist_node within the struct.
783  */
784 #define hlist_for_each_entry_safe(pos, n, head, member
    )       \
785     for (pos = hlist_entry_safe((head)->first, typeof(*
    pos), member);\
786          pos && ({ n = pos->member.next; 1; });         \
787          pos = hlist_entry_safe(n, typeof(*pos), member))
788
789 #endif
790
```

```c
//
// Created by hfwei on 2023/12/20.
//

#include <stdio.h>
#include <assert.h>
#include "ll/ll.h"

#define NUM 10

void SitAroundCircle(LinkedList *list, int num);
void KillUntilOne(LinkedList *list);
int GetSurvivor(const LinkedList *list);

int main(void) {
  printf("I hate the Josephus game!\n");

  LinkedList list;
  Init(&list);

  SitAroundCircle(&list, NUM);
  // Print(&list);

  KillUntilOne(&list);
  int survivor = GetSurvivor(&list);
  printf("%d : %d\n", NUM, survivor);

  Free(&list);

  return 0;
}

void SitAroundCircle(LinkedList *list, int num) {
  for (int i = 1; i <= num; i++) {
    Append(list, i);
  }
}

void KillUntilOne(LinkedList *list) {
  Node *node = list->head;

  while (!IsSingleton(list)) {
    // use node to delete node->next
    Delete(list, node);
```

```
45      node = node->next;
46    }
47  }
48
49  int GetSurvivor(const LinkedList *list) {
50    assert(IsSingleton(list));
51
52    return GetHeadVal(list);
53  }
```

```
1 add_executable(josephus josephus.c ll/ll.c)
```

```
 1 # `13-linked-list`
 2
 3 - CMakeLists.txt (`ll/ll.c`)
 4 - `struct node *`
 5 - `#ifndef`
 6
 7 ## Intrusive list
 8
 9 - [Data Structures in the Linux Kernel: Doubly linked list
   ](https://0xax.gitbooks.io/linux-insides/content/
   DataStructures/linux-datastructures-1.html)
10   - [types.h](https://github.com/torvalds/linux/blob/
   16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/
   types.h)
11     - [list_head](https://github.com/torvalds/linux/blob/
   16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/
   types.h#L184)
12   - [list.h](https://github.com/torvalds/linux/blob/
   16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/
   list.h)
13   - [misc.c as an application](https://github.com/torvalds
   /linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/
   drivers/char/misc.c)
14
15 ```c++
16 struct list_head {
17   struct list_head *next, *prev;
18 };
19 ```
20
21 ## [linux](https://github.com/torvalds/linux)
22
23 - [types.h](https://github.com/torvalds/linux/blob/
   16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/
   types.h)
24   - [list_head](https://github.com/torvalds/linux/blob/
   16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/
   types.h#L184)
25 - [list.h](https://github.com/torvalds/linux/blob/
   16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/
   list.h)
26 - [How does the kernel implements Linked Lists?](https://
   kernelnewbies.org/FAQ/LinkedLists)
27   > Also illustrate how to use the list_head structure.
```

````
28
29 ## [`GList`](https://gitlab.gnome.org/GNOME/glib/-/blob/
   bc56578a087fc4eda0204b361d75162a4144546d/glib/glist.c) in
30
31 `GNOME/glibc`
32
33 - [glist.h](https://gitlab.gnome.org/GNOME/glib/-/blob/
   main/glib/glist.h)
34 - [glist.c](https://gitlab.gnome.org/GNOME/glib/-/blob/
   main/glib/glist.c)
35
36 ```c++
37 typedef struct _GList GList;
38
39 struct _GList
40 {
41   gpointer data;
42   GList *next;
43   GList *prev;
44 };
45 ```
46
47 - [Docs for List](https://docs.gtk.org/glib/struct.List.
   html)
48
49 ## `glibc`
50
51 - [list_t.h](https://github.com/bminor/glibc/blob/master/
   include/list_t.h)
52 - [list.h](https://github.com/bminor/glibc/blob/master/
   include/list.h)
````